Project Report - DA 526

"Distracted Driver Detection"

GITHUB

Team:

Raj Katkar (224156008)

Gavit Vaibhav Ishwar (224156013)

Aniket Zope (224156014)

PV Rohith Kumar (224156017)

Sushant Pargaonkar (224156020)

Problem Statement

Our goal in this project is to detect *if the car driver is driving safely or performing any activity that might result in an accident or any harm to others.* With the increasing number of vehicles on the road and advancements in technology, driver distraction has become a significant concern in road safety. Distracted driving is a leading cause of accidents, injuries, and fatalities on the road.

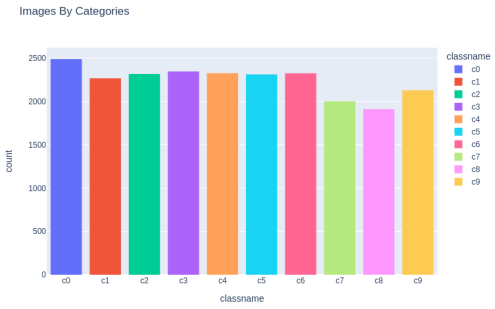
Therefore, there is a need to develop an *efficient and accurate system for detecting driver distraction in real-time.* The **problem statement** is to *design a driver distraction detection system that can accurately and quickly detect, classify distracted driving behaviors in real-time using machine learning and deep learning algorithms.*

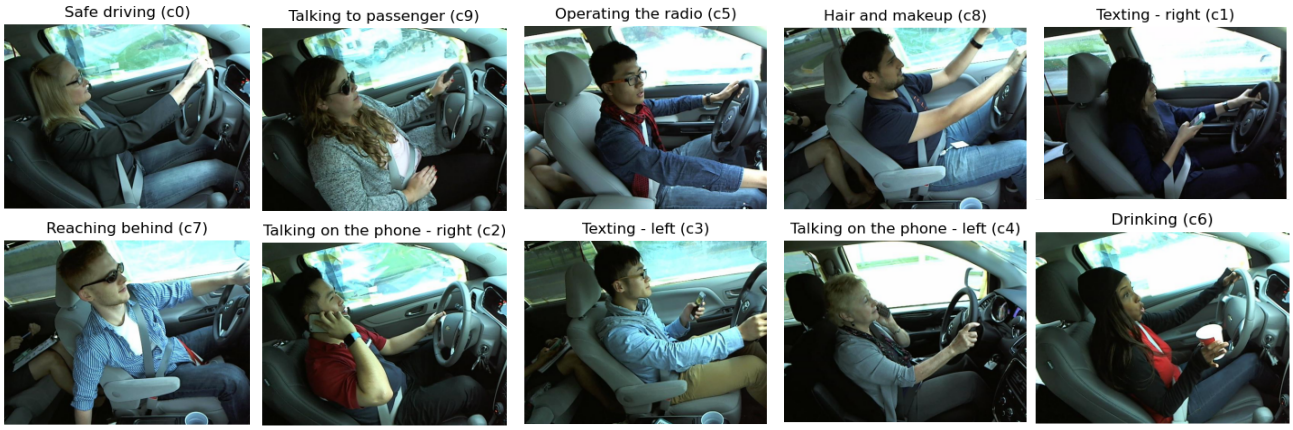
Dataset

We have used the dataset from **Kaggle (State Farm Distracted Driver Detection).**

Key Features of the State Farm Distracted Driver Detection Dataset are:

- **Number of images (Training set):** 22,424.
- **Number of images (Testing set):** 79,726.
- **Number of classes:** 10
- **Image size:** 640x480 pixels
- **Image format:** JPG
- **Class distribution:** uneven, ranging from 2,000 to 2,800 images per class
- **Data source and collection method:** dashcams in actual vehicles driven by State Farm customers.





Related Work

- Driver Distraction Recognition Based on Physiological Features
- Driver Distraction Recognition Based on Vehicle Control State
- **Driver Distraction Recognition Based on Visual Features**
    - The visual-feature-based distraction detection method distinguishes the driver's distraction behavior through a variety of image processing techniques. Compared with the above two methods, this method is more user-friendly and lower in cost.
    - **Seshadri et al.** combined the original pixel and histogram of gradient orientation (HOG) features with the AdaBoost classifier to detect distracted behaviors such as using mobile phones, and achieved better results on self-made datasets.
- **Abouelnaga et al.** first created a similar image data set by imitating the distracted driving competition data set on Kaggle and then eliminated redundant information through face, hand positioning, and skin segmentation. Finally, they used convolutional neural networks and genetic algorithms to classify the driver's distracted behavior, which achieved an **accuracy of 84.64%.**
- **Baheti et al.** proposed an improved algorithm for VGG-16, which can not only detect the driver's distracted behavior but also identify the causes.
- **Ou et al.** solved the problem of insufficient driver distraction behavior data through the transfer learning of the ResNet-50 model pre-trained on ImageNet, and proved that it has strong robustness to illumination changes.

Methods, Experiments and Results

With the understanding of the problem statement and several brainstorming sessions we moved on to creating models for our dataset. The basic idea was to construct the following:

- **PCA and Autoencoder with KNN classifier**
- **Custom CNN 01**
- **Custom CNN 02**
- **VGG16**
- **RESNET**
- **MobileNET**

PCA with KNN Classifier

Applying PCA to extract features and reduce dimension and Classify using KNN classifier.
Original size of image = 640 x 480.
Problem Faced (1) - Computation of Covariance became difficult as RAM was overflowing, so applied preprocessing.
Preprocessing - Reduced 50 pixels from all four sides of the images and then resized them to 180 x 127.
Searched for different libraries for Matrix multiplications. "Cupy" library performed faster than the "Numpy" library.

Problem Faced (2) - Computation of Eigenvectors and Eigenvalues. To make use of the GPU for faster computation used pyTorch library, converted matrices to tensors and then performed operations.
And then applied KNN classifiers with random K values, 31, 51, 71. There were 10 classes and around 500 images of each class in test data, so selected K values of 51.
Result : Accuracy : 50.39 %
Confusion Matrix :

Predicted

Actual

```
[[447   2  10   0   0   8   0   4   0   0]
 [  0  62   1 126 130   0   0 100   0  81]
 [  0   1 466   1   0   0   5   0   0   0]
 [  0 126   0  48 123   0   0 105   0  67]
 [  1 128   1 106  41   0   0  87   0  76]
 [  5   1   4   0   0 447   4   0   0   0]
 [  1   0  17   1   1   7 436   1   1   0]
 [  0 120   1 107 124   0   0  28   0  70]
 [ 13   1  48   0   2   7  16   0 295   0]
 [  0 132   0  90 105   0   0  83   0  17]]
```

Autoencoder with KNN Classifier

Autoencoder to extract features and classify using KNN classifier.
Input dimension = 180 X 127 =22860. In each layer the dimension is reduced by a factor of 3. Output of the encoder is of size 282. Autoencoder is trained to regenerate the input at the output layer.
Tried to improve or better the model performance by experimenting.
Experiments: Varying learning rate, Varying batch size Varying number of epochs. Default learning rate 0f 0.001 gave the best results. Large batch size slowed down the convergence rate, or the loss was higher. Used optimal batch size of 128. Increased the number of epochs from 100 to 400 for better results.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 7620)              174200820

 dense_1 (Dense)             (None, 2540)              19357340

 dense_2 (Dense)             (None, 847)               2152227

 dense_3 (Dense)             (None, 282)               239136

 dense_4 (Dense)             (None, 847)               239701

 dense_5 (Dense)             (None, 2540)              2153920

 dense_6 (Dense)             (None, 7620)              19362420

 dense_7 (Dense)             (None, 22860)             174216060

=================================================================
Total params: 391,921,624
Trainable params: 391,921,624
Non-trainable params: 0
```

And then applied KNN classifiers with random K values of 31, 51, 71.
Result : Accuracy : 50.77 %
Confusion Matrix:

Predicted

Actual

```
[[441   0   9   2   0  11   1   0   6   1]
 [  2  57   2 134 124   1   0  94   3  83]
 [  2   0 452   0   0   0  18   0   1   0]
 [  6 131   0  41 118   5   0  92   1  75]
 [  3 131   1  95  46   2   0  88   1  73]
 [  9   0   7   0   0 440   4   0   1   0]
 [  2   1  22   1   0   6 428   0   5   0]
 [  4 120   0 109 118   1   0  37   1  60]
 [ 15   0  28   2   2   9  13   0 313   0]
 [  5 121   4  88 101   1   0  84   0  23]]
```

Custom CNN 01

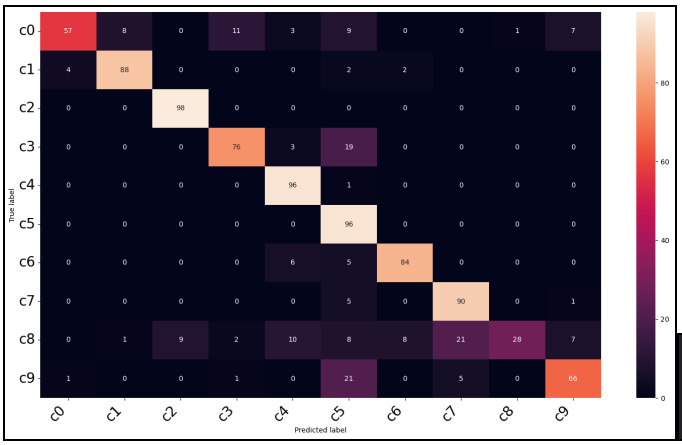
After creating and training multiple CNN's, This CNN has only 27,034 parameters. We added batch normalization and dropouts to reduce overfitting.

Test Accuracy: 86% (on normally splitting the dataset).

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lambda (Lambda)             (None, 224, 224, 1)       0

 conv2d (Conv2D)             (None, 110, 110, 16)      416

 batch_normalization (BatchN  (None, 110, 110, 16)     64
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 55, 55, 16)       0
 )

 dropout (Dropout)           (None, 55, 55, 16)        0

 conv2d_1 (Conv2D)           (None, 27, 27, 16)        2320

 batch_normalization_1 (Batc  (None, 27, 27, 16)       64
 hNormalization)

 conv2d_2 (Conv2D)           (None, 6, 6, 32)          4640

 batch_normalization_2 (Batc  (None, 6, 6, 32)         128
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 3, 3, 32)         0
 2D)

 dropout_2 (Dropout)         (None, 3, 3, 32)          0

 flatten (Flatten)           (None, 288)               0

 dense (Dense)               (None, 64)                18496

 batch_normalization_3 (Batc  (None, 64)               256
 hNormalization)

 dropout_3 (Dropout)         (None, 64)                0

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 27,034
Trainable params: 26,778
Non-trainable params: 256
_________________________________________________________________
```

Custom CNN 02

Sensing something wrong, we tried to modify the CNN architecture by introducing *batch normalization* and *dropout layer* in an *attempt to minimize the test error.* The new **Custom CNN 02** as we call it had the following results:




```
Accuracy: 0.791346
Precision: 0.813326
Recall: 0.791346
F1 score: 0.779416
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 222, 222, 32)      896

 max_pooling2d_3 (MaxPooling  (None, 74, 74, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 72, 72, 64)        18496

 max_pooling2d_4 (MaxPooling  (None, 24, 24, 64)       0
 2D)

 conv2d_6 (Conv2D)           (None, 22, 22, 128)       73856

 batch_normalization_1 (Batc  (None, 22, 22, 128)      512
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 7, 7, 128)        0
 2D)

 conv2d_7 (Conv2D)           (None, 5, 5, 256)         295168

 dropout_3 (Dropout)         (None, 5, 5, 256)         0

 flatten_1 (Flatten)         (None, 6400)              0

 dense_3 (Dense)             (None, 1024)              6554624

 dropout_4 (Dropout)         (None, 1024)              0

 dense_4 (Dense)             (None, 256)               262400

 dropout_5 (Dropout)         (None, 256)               0

 dense_5 (Dense)             (None, 10)                2570

=================================================================
Total params: 7,208,522
Trainable params: 7,208,266
Non-trainable params: 256
_________________________________________________________________
```

With just **15 epochs** we achieved training accuracy of **72.69%** and still improving – meaning if we trained it for *more than 15 epochs* we would have closed it down to *90+ and we did.* Also, the **test accuracy was 79.13%**. Contemplating for a moment regarding the CNN architecture that we built and exchanging intense eye gaze with our mates thinking we somehow built the *best CNN architecture for this dataset.* So, we tried to test it on **unseen data** and below were the results:

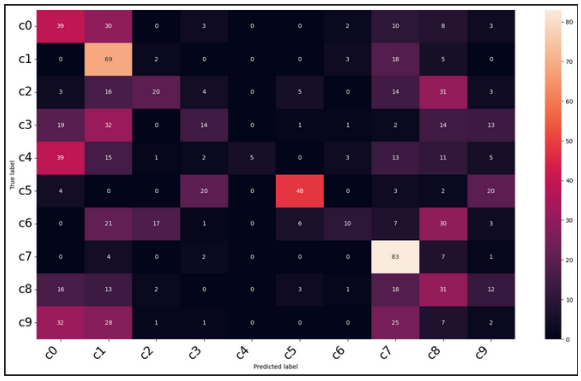
Not so surprising for us, we digged deeper into what could

have gone wrong and found the following errors made by us:

- We used the same data to – *train, validate and test.*
- Upon realizing the above mistake and rectifying it we still got pretty bad accuracy on the *new unseen test data*.
- As a result, we investigated further to see what might have gone wrong and discovered that *our training data contains many photographs of the same subject inside a class*, each with a tiny difference in height or width and/or angle. Due to the fact that the model was trained using a lot of the same data that it was attempting to forecast, this was leading to a **data leakage issue.**

Our solution to data leakage:

To counter the issue of data leakage, we observed the dataset more keenly and got to know that there are **26 unique drivers** present namely: *p002, p012, p014, p015, p016, p021, p022, p024, p026, p035, p039, p041, p042, p045, p047, p049, p050, p051, p052, p056, p061, p064, p066, p072, p075 and p081.* We then divided the dataset based on the above IDs as in – *first 20 IDs in train and remaining 6 in test* instead of a random split. Now, after fitting our model with this new dataset we see some *"realistic"* results:



- *Custom **CNN 01*** accuracy:
  - **Before data leakage solution:** 86%.
  - **After data leakage solution:** *27%.*
- *Custom **CNN 02*** accuracy:
  - **Before data leakage solution:** *79.13%*
  - **After data leakage solution:** *32.03%*
    - The heatmap depicts the same story.

Also for completeness sake, while performing preprocessing we randomly changed brightness and contrast using a random number generator. And later applied data augmentation over the same. To improve the results further we head on to other deep neural networks mentioned next.

VGG16 for Distracted Driver Detection

VGG16 is a well-suited choice for distracted driver detection as it can effectively extract meaningful features from images. With 16 layers, including convolutional and fully connected layers, it captures spatial information and contributes to accurate classification. By utilizing pre-trained weights from ImageNet, it demonstrates a strong ability to recognize diverse objects and patterns.

Why VGG16 for Distracted Driver Detection?

- Depth and Complexity: VGG16 can learn intricate features, crucial for detecting distracted drivers.
- Transfer Learning: Pre-trained weights from ImageNet enable efficient training with limited labeled data.

- Generalization: VGG16's ability to capture spatial and semantic information aids in detecting various forms of driver distraction.

Preprocessing

- Data augmentation techniques (random rotations, shifts, shearing, zooming) are applied using ImageDataGenerator to diversify the training set and improve generalization.
- Pixel values are normalized using the preprocess_input function to align with the preprocessing steps used during VGG16 training on ImageNet.

Model Training

During the model training phase, the pretrained VGG16 layers are frozen to prevent their weights from being updated. This approach allows us to leverage the learned representations and focus on training the newly added layers specific to the distracted driver detection task. We add additional layers, including fully connected layers with dropout regularization to reduce overfitting, and a final softmax layer for multi-class classification. The model is compiled with the Adam optimizer, categorical cross-entropy loss function, and accuracy as the evaluation metric.

Model Evaluation

The model is evaluated using a validation set to monitor loss and accuracy during training. The training history is observed over each epoch for convergence and overfitting. Finally, the model's generalization performance is assessed on a separate test set.

Model Performance

During the training process, the model underwent 10 epochs using a batch size of 128.

After training, the model was evaluated on a separate test set, yielding the following results:

Limitations of VGG16 for Distracted Driver Detection

- Data Leakage: Improper separation of training and validation datasets can lead to data leakage, resulting in overly optimistic performance estimates.
- Limited Contextual Understanding: VGG16's focus on visual features may limit its ability to capture contextual cues such as facial expressions and hand movements, which are important for distracted driver detection.
- Computational Complexity

VGG16 for Distracted Driver Detection with Driver-ID Based Data Splitting

We present an improved approach for training a VGG16-based model using driver-ID based data splitting to address the issue of data leakage.

Data Splitting

To prevent data leakage, we use a driver-ID based data splitting approach. Each driver's images are assigned to a single data split, avoiding the presence of the same driver in multiple splits. This strategy prevents the model from learning driver-specific patterns and enhances its ability to generalize to unseen drivers.

Model Training

Data is split into training, validation, and test sets, ensuring drivers are not present in multiple splits to avoid biases. Pretrained VGG16 layers are frozen, additional layers are added and trained, and the model is compiled with the Adam optimizer, categorical cross-entropy loss, and accuracy metric.

After training, the model was evaluated on a separate test set, yielding the following results:

```
146/146 - 32s - loss: 2.3231 - accuracy: 0.4178 - 32s/epoch - 219ms/step
Test accuracy: 0.41777873039245605
146/146 [==============================] - 33s 225ms/step
```

Although the driver-ID based data splitting strategy helps mitigate data leakage, it may still have limitations. In cases where there are only a few images per driver, it may be challenging to create well-balanced splits, potentially affecting the model's performance. Additionally, this approach assumes that drivers' behavior i s consistent over time, which may not always hold true.

RESNET50

At first images were processed by resizing and reshaping. As RESNET's input image size is 224x224, we reshaped our training instance to match it. Also applied standard Image augmentation techniques from Tensorflow Image Data Generator function.

Now, we have used ResNet50 from Keras, which consists of approximately 25 million parameters and 107 layers. The default size of the model is around 98MB. ResNet50 produces a 7x7 feature map with 2048 channels as output after the 107 layers. This poses a challenge when using a neural network because the number of parameters to train increases rapidly if we directly flatten the last layer.

To address this challenge, we have applied a global average pooling 2D layer, which takes the maximum value from each 7x7 kernel and outputs a 2048-dimensional vector. At this point, we have two choices:

1. Directly use the 2048-dimensional vector as input to the final layer to obtain the output for the three classes. This approach assumes that the global average pooling layer has already captured sufficient information to classify the input accurately.
2. Add additional nonlinear layers after the global average pooling layer to further refine the features and make more complex corrections before reaching the final output layer. This approach allows for more flexibility and capacity to learn intricate patterns and relationships within the data.

The decision between these two choices depends on the complexity of the classification task, the amount of available training data, and the performance of the model. Experimenting with both approaches and evaluating their respective performances on validation or test data can help determine which option works best for the specific problem at hand.

Quantization-Aware Training

Quantization-Aware Training emulates inference-time quantization, creating a model that downstream tools will use to produce actually quantized models. The quantized models use lower-precision (e.g. 8-bit instead of 32-bit float), leading to benefits during deployment.

There are several ways to implement Quantization-Aware Training. You can choose to Quantize the entire model, i.e. all layers, or Quantize only selected layers of the model. The question is, if quantization reduces the size of the model, won't it be better to just quantize the whole model, so why do we have the option to quantize only some layers? The reason is simple: Quantizing the model can have a negative effect on accuracy. Thus, we can selectively quantize model layers to explore the trade-off between accuracy, speed, and model size. Quantization-Aware Training is a fine-tuning method. Therefore, we need to create a Quantize Aware model and train it again. In the case of quantization after training, we do not have to retrain the model.

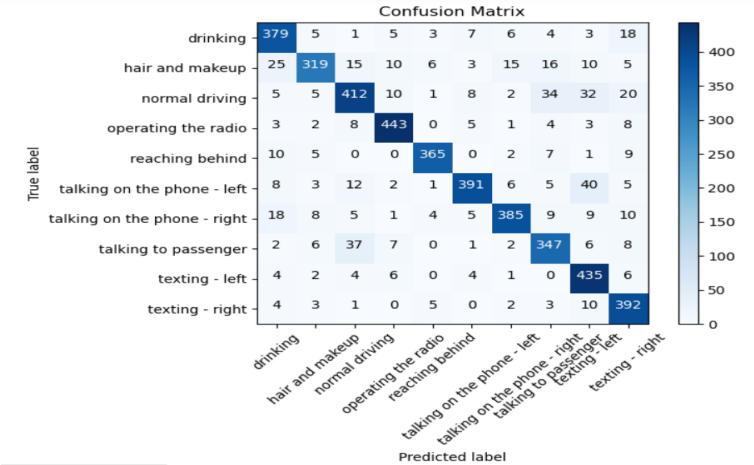
Here we have quantized only a few layers so that instead of quantizing the entire model, we will only be applying quantization to dense layers using the function “apply_quantization_to_dense”. Then we annotate the dense layers using “quantize_annotate_layer()” from tensorflow. After this we apply this quantization of the dense layer function to our model using the tensorflow “clone_model” function. Now our model is quantized which means it is aware which layers weights need to be quantized. Rest of the training model is the same as that of any regular deep learning model.

Accuracy(%)	Training Accuracy	Val Accuracy	Test Accuracy
RESNET w/o QAT	98%	98%	45%
RESNET with QAT	98%	96%	20%
RESNET w/o QAT - [2Million Parameter Learning] @Epoch 20	20% @20 epoch	21% @20 epoch	15%
Random Guess [i.e. randomly stating activity of the driver = 1/10 probability of being correct]	10	10	10

**QAT - Quantization aware training*

MobileNET

The MobileNetV2 model is instantiated from the tf.keras.applications module, with the specified input shape and excluding the top classification layers. Pre-trained weights, trained on the ImageNet dataset, are used to initialize the model.



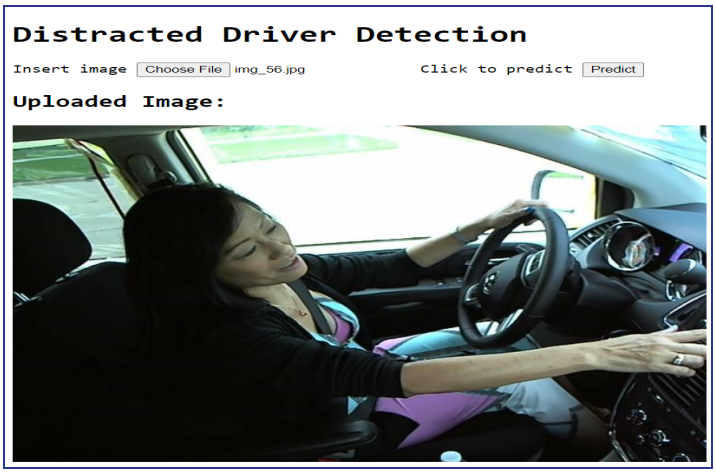
The weights of the MobileNetV2 base model are frozen by setting trainable=False. This ensures that the pre-trained weights remain unchanged during training, allowing us to focus on training the added layers.

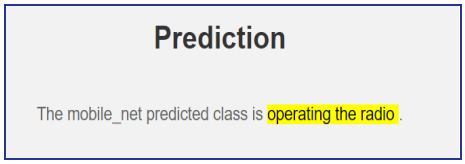
The model structure is defined using the Sequential model from TensorFlow. The base model is added as the first layer, serving as a feature extractor. A GlobalAveragePooling2D layer is then added to perform global average pooling, reducing spatial dimensions while preserving channel information. Finally, a Dense layer with a softmax activation function is appended for classification, with the number of units corresponding to the number of classes in the problem ie.10.

We achieved a **test accuracy** of **0.8624303232998886.**

We have **deployed the model on a local host**, and by running the app.py file, the saved model weights are loaded. When a user clicks on predict in the index.html, the model performs classification on the uploaded image and displays the results on the predict.html page.

We have also worked with **real-time implementation**, where each frame of the video is classified individually after *being broken down into frames.*





Conclusions

Clearly, *modern deep learning models perform much better than traditional machine learning models.* Transfer Learning based models perform exceptionally well. We observed the **problem of data leakage,** that is if the drivers in the training set are also present in the test set, then we are not testing it correctly and the model may not perform well on the new drivers. So, we make sure that the drivers which are present in the training set are not present in the test set. And we observe that all models perform poorly on test data. Even VGG got only about 40% accuracy.

As there are only 22K images in total, the solution *we can try is to get a lot more images from a lot more drivers and then train these models*. We hope that this will improve the results and they will generalize better on unseen drivers.

Model	Accuracy *(With Data Leakage)*	Accuracy *(Without Data Leakage)*
Custom CNN 01	86% (@20 epochs)	27%
Custom CNN 02	79.13% (@13 epochs)	32.03%
VGG16	99.44% (@10 epochs)	41.77%
MobileNET	86.24% (@10 epochs)	-