# AIML CAPSTONE PROJECT

**Computer Vision Stanford Car Detection**

# PROJECT MEMBER

| SR NO | NAME | EMAIL ID | GITHUB PROJECT REPO |
|---|---|---|---|
| 1 | Rupali Botre | botrerupalid@gmail.com | https://github.com/bamachrn/car-detection-cv2 |
| 2 | Bama Charan Kundu | bamachrn@gmail.com | https://github.com/bamachrn/car-detection-cv2 |

**TABLE OF CONTENTS**

# 1. PROJECT DESCRIPTION

**DOMAIN:** Automotive. Surveillance.

**CONTEXT:**

Computer vision can be used to automate supervision and generate action appropriate action trigger if the event is predicted from the image of interest. For example, a car moving on the road can be easily identified by a camera as make of the car, type, colour, number plates etc.

**DATA DESCRIPTION:**

The Cars dataset contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, Year, e.g.

2012 Tesla Model S or 2012 BMW M3 coupe.

- **Train Images**: Consists of real images of cars as per the make and year of the car.
- **Test Images**: Consists of real images of cars as per the make and year of the car.
- **Train Annotation**: Consists of bounding box region for training images.
- **Test Annotation**: Consists of bounding box region for testing images.

Dataset has been attached along with this project. Please use the same for this capstone project. Original link to the dataset: https://www.kaggle.com/jutrera/stanford-car-dataset-by-classes-folder

Reference: 3D Object Representations for Fine-Grained Categorisation, Jonathan Krause, Michael Stark, Jia Deng, Li Fei-Fei 4th IEEE

Workshop on 3D Representation and Recognition, at ICCV 2013 (3dRR-13). Sydney, Australia. Dec. 8, 2013.

**PROJECT OBJECTIVE:**

Design a DL based car identification model.

# 2. MILESTONE 1

## 2.1 Summary of Problem Statement

- The Stanford Car Dataset will be utilized to build a vehicle recognition predictive model. The goal of the model is to classify a car's year, make and model given an input image.
- With technology development and image recognition methods being increasingly accurate, many business and government applications arise. Among them, recognition technologies are often used for security and/or tracking purposes. The Stanford Car Dataset will be utilized to build a vehicle recognition predictive model.
- The ultimate goal of the model is to classify a car's make and model given an input image. This model could be further developed to be used in creating a mobile application that assists users in identifying cars of interest.
  The users would simply take a picture of the vehicle of interest and the application would return information (Make and Model) regarding the recognized vehicle.
- The Stanford Cars Dataset is a large collection of vehicle images produced by Dr. Jonathan Krause and his team at Stanford University. To generate an initial list of car labels, the authors crawled an unspecified popular car website to create a list of all cars from 1990 to 2012.
- Because many car models do not change their appearances across model years, the authors merged car classes with similar visual features using a technique called perceptual hashing. This technique compares two media objects such as images to see if they are different from each other. In this study, the authors used Hamming distance, the difference between two strings of numbers that represent the pictures, as a measure to determine the dissimilarities between car classes. After the initial round of perceptual hashing, the authors generated 197 classes of cars.
- To expand on the pool of car images, Dr. Krause and his team collected car images from Flickr, Google, and Bing. While these search engines allowed the authors to collect many images, they needed to verify that the collected car images were from the correct car classes.
- To verify the identities of these images, the authors utilized Amazon Mechanical Turk (AMT) workers to annotate the car images with the correct car labels. The car identification task contained an image of the car that needed to have its identity verified, an image of the actual car from the class of interest, and an image of a car from a different class that could easily be mixed up as an image from the target class. Based on the two images with confirmed classes, the workers must decide whether the unverified car image was from the class of interest. If not, the workers annotated the image with the correct class label. For the workers to qualify for this task, they needed to pass a series of tests that contained some of the most difficult cars to identify.
- To determine the quality of annotations, the authors used a technique called Get Another Label (GAL), which is an algorithm using expectation-maximum, a type of maximum-likelihood algorithm that estimates values for model parameters for incomplete data, that estimates the probability that an image is from a certain class while also determining the quality of a worker based on their correct annotations. The criteria for GAL for the car annotation task were:
  1) an agreement of workers on the correct car class of an image and

2) the ability of workers to identify “gold standard” images, which were images that the authors knew the correct labels.

- After the GAL probabilities of a candidate image exceeded a certain threshold, the image was put into the target class. GAL was also used to further weed out poor quality workers by assigning more and more images to users that have low scores, discouraging them to continue the task. After obtaining the set of images with assigned classes, the authors utilized a different group of AMT workers to assign bounding boxes, the section of an image that contains the target object, using a technique presented by Fei-Fei et. al. To further remove duplicate images, the authors used another round of perceptual hashing on the images based on the bounding boxes, yielding a total of 16,185 images with 196 classes of cars.

## 2.2 Approach to EDA and Pre-processing

- The Kaggle Stanford Cars Dataset contains total 16,185 images of cars. There are a total of 196 classes of cars in this dataset. The data is split in half to be used as training and testing sets. The data also comes with class labels and bounding boxes for all images.

- The classes are typically at the level of Year, Make and Model (e.g. 2012 Tesla Model S or 2012 BMW M3 coupe). The sizes of each image are different. Utilization of the bounding boxes is essential in the pre-processing phase to first obtain images that focus on the objects of interest, which in this case are the vehicles. The actual images are in JPG format, but the data comes zipped in TGZ/TAR format.
- Pre-processing steps were dependent both on the modelling method and the available resources. The original dataset contained a collection of images with varying heights, widths, and colour channels. In order to simplify problem and model complexity, images were normalized and resized to a particular height and width deemed suitable for model construction.

- The Stanford Car Dataset will be utilized to build a vehicle recognition predictive model.

- The goal of the model is to classify a car's year, make and model given an input image.

- The dataset contained no missing values, so no imputations or data removal was required due to the nature of image data. In terms of Exploratory Data Analysis, the class labels were split to explore the individual Make, Model, Type and Year levels of the labels.

- The string-formatted labels were split by a space, then the output of that were categorized into the Make, Model, Type and Year levels. This was tricky, since some the Make and Model levels had different lengths (for instance, Aston Martin vs. BMW in the Make level and Sonata vs. F-450 Super Duty Crew in the Model level). This extraction of class label levels was performed to the best of our abilities.

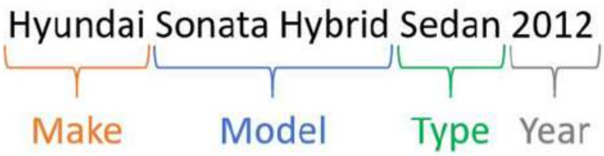
- There were 196 classes originally. Because of this high total class number, the levels of class labels were analysed with the hopes of reducing the total class number. Initially the class labels were analyzed by human eyes.

- While the Stanford dataset contained pre-split training and testing data, 50:50 data.

## 2.3 Data and Findings

- The original dataset defined a ‘class’ as the combination of make, model, and year.

- This yields 196 individual and unique classes. An example of one of these classes is shown in Figure. The class levels were parsed into the components also shown in Figure. It may be possible to extract more useful information by separating these characteristics.




- The following table provides specific descriptive statistics from the entire original dataset.

- The image dimensions (height, width, and channels) were added to support future modelling decisions. Due to the way that the image of this dataset was created, a thorough Exploratory Data Analysis of the original class distributions was highly desired.

| | ClassNo | Class | Make | Model | Type | Year | Image Height | Image Width | Color Channels |
|---|---|---|---|---|---|---|---|---|---|
| **Type** | Integer | String | String | String | String | Integer | Integer | Integer | Integer |
| **Uniques** | 196 | 196 | 49 | 177 | 13 | 16 | Several | Several | 3 |
| **Mean** | - | - | - | - | - | 2009.56 | 308 | 573 | - |
| **Std Dev** | - | - | - | - | - | 4.43 | 214 | 375 | - |

Adding class name to image file as follow:

| | image_file | x0 | y0 | x1 | y1 | class | class_name | car_image_path |
|---|---|---|---|---|---|---|---|---|
| **0** | 00001.jpg | 39 | 116 | 569 | 375 | 14 | Audi TTS Coupe 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **1** | 00002.jpg | 36 | 116 | 868 | 587 | 3 | Acura TL Sedan 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **2** | 00003.jpg | 85 | 109 | 601 | 381 | 91 | Dodge Dakota Club Cab 2007 | /content/drive/My Drive/AIML/Capstone project/... |
| **3** | 00004.jpg | 621 | 393 | 1484 | 1096 | 134 | Hyundai Sonata Hybrid Sedan 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **4** | 00005.jpg | 14 | 36 | 133 | 99 | 106 | Ford F-450 Super Duty Crew Cab 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **5** | 00006.jpg | 259 | 289 | 515 | 416 | 123 | Geo Metro Convertible 1993 | /content/drive/My Drive/AIML/Capstone project/... |
| **6** | 00007.jpg | 88 | 80 | 541 | 397 | 89 | Dodge Journey SUV 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **7** | 00008.jpg | 73 | 79 | 591 | 410 | 96 | Dodge Charger Sedan 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **8** | 00009.jpg | 20 | 126 | 1269 | 771 | 167 | Mitsubishi Lancer Sedan 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **9** | 00010.jpg | 21 | 110 | 623 | 367 | 58 | Chevrolet Traverse SUV 2012 | /content/drive/My Drive/AIML/Capstone project/... |
| **10** | 00011.jpg | 51 | 93 | 601 | 393 | 49 | Buick Verano Sedan 2012 | /content/drive/My Drive/AIML/Capstone project/... |

**Bounding Box**

Image classification involves assigning a class label to an image, whereas object localization involves drawing a bounding box around one or more objects in an image. Object detection is more challenging and combines these two tasks and draws a bounding box around each object of interest in the image and assigns them a class label.

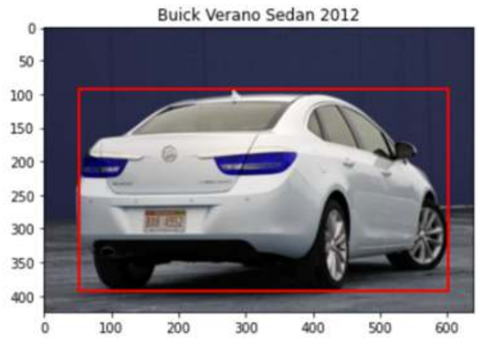
Following image showing with Bounding Box

```
In [ ]: show_image(df)
```

Showing Image with Bounding Box

```
In [ ]: df['car_image_path']
Out[ ]: '/content/drive/My Drive/AIML/Capstone project/car_data/car_data/train/Buick Verano Sedan 2012/00011.jpg'
```

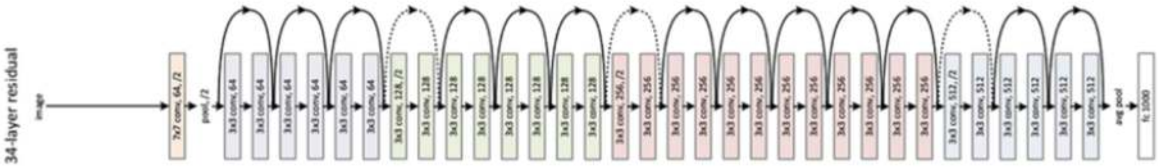
```
In [ ]: show_image_with_bb(df)
```



# 3. MILESTONE 2

## 3.1 RESNET

- We started building deeper custom CNNs and exploring other state-of-the-art architectures such as ResNet34
- Batch normalization is applied after convolution layers in order to help speed up training and allow an infinite positive-negative range for weights to be evaluated in. Linear activation is used to prevent further degradation or loss of information.
- Some model-specific image pre-processing is done prior to training. The model is trained in batches using augmented images since it is believed that there may not be enough data for the large number of target classes proposed. Image augmentation has greatly improved the rate at which models converge on a solution and the maximum accuracy overall.




- The same learning rate tuning process was performed, but different weight decay values were explored for the ResNet34 model. Utilizing the One-Cycle-Policy, the ResNet34 model was able to achieve 72.95% validation accuracy after just 10 epochs. This model building phase proved the effectiveness of the One-Cycle-Policy in terms of both performance and time costs.
- The ResNet34 model is the best performing model at this moment. Further training of this model will be performed in the future, in addition to exploring with different learning rates at different phases. Some of the most confused (mis-classified) images and areas that the ResNet34 model focuses on during classification.

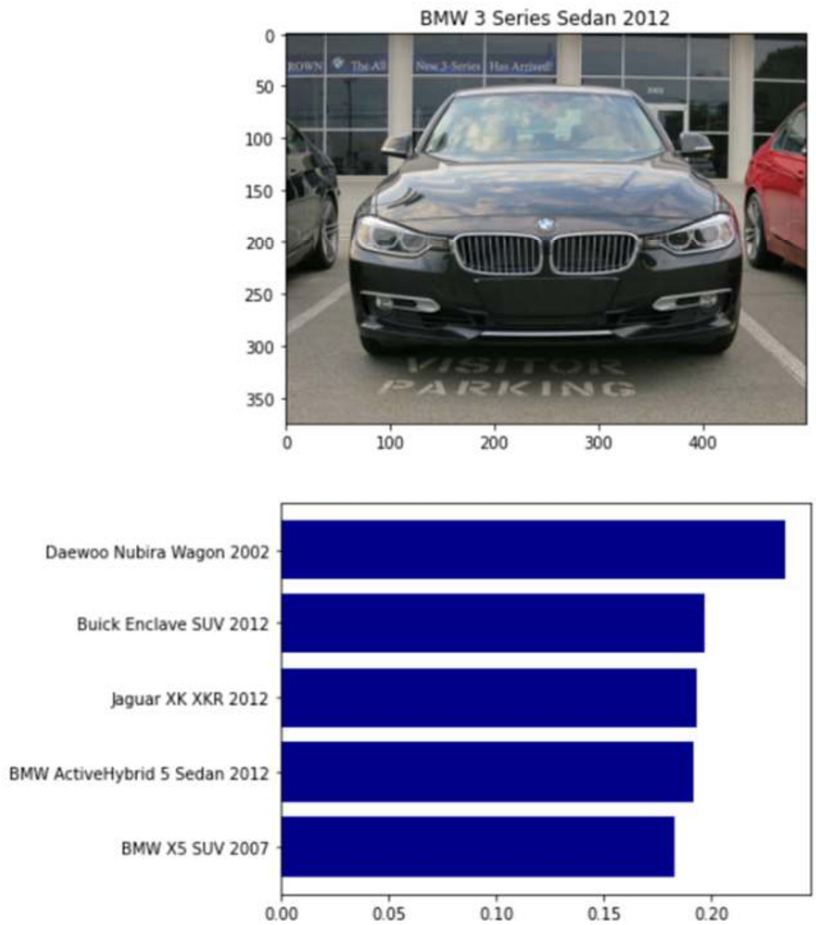
```
No. epochs: 1,     Training Loss: 5.172     Valid Loss: 4.686     Valid Accuracy: 0.056
No. epochs: 2,     Training Loss: 1.487     Valid Loss: 3.478     Valid Accuracy: 0.219
No. epochs: 2,     Training Loss: 1.487     Valid Loss: 3.478     Valid Accuracy: 0.219
No. epochs: 2,     Training Loss: 4.619     Valid Loss: 2.59      Valid Accuracy: 0.362
No. epochs: 2,     Training Loss: 4.619     Valid Loss: 2.59      Valid Accuracy: 0.362
No. epochs: 3,     Training Loss: 1.736     Valid Loss: 2.055     Valid Accuracy: 0.484
No. epochs: 3,     Training Loss: 1.736     Valid Loss: 2.055     Valid Accuracy: 0.484
No. epochs: 4,     Training Loss: 0.286     Valid Loss: 1.68      Valid Accuracy: 0.565
No. epochs: 4,     Training Loss: 0.286     Valid Loss: 1.68      Valid Accuracy: 0.565
No. epochs: 4,     Training Loss: 1.547     Valid Loss: 1.446     Valid Accuracy: 0.612
No. epochs: 4,     Training Loss: 1.547     Valid Loss: 1.446     Valid Accuracy: 0.612
No. epochs: 5,     Training Loss: 0.529     Valid Loss: 1.52      Valid Accuracy: 0.597
No. epochs: 5,     Training Loss: 0.529     Valid Loss: 1.52      Valid Accuracy: 0.597
No. epochs: 5,     Training Loss: 1.336     Valid Loss: 1.134     Valid Accuracy: 0.681
No. epochs: 5,     Training Loss: 1.336     Valid Loss: 1.134     Valid Accuracy: 0.681
No. epochs: 6,     Training Loss: 0.543     Valid Loss: 0.827     Valid Accuracy: 0.779
No. epochs: 6,     Training Loss: 0.543     Valid Loss: 0.827     Valid Accuracy: 0.779
No. epochs: 7,     Training Loss: 0.188     Valid Loss: 0.801     Valid Accuracy: 0.787
No. epochs: 7,     Training Loss: 0.188     Valid Loss: 0.801     Valid Accuracy: 0.787
No. epochs: 7,     Training Loss: 0.631     Valid Loss: 0.78      Valid Accuracy: 0.792
No. epochs: 7,     Training Loss: 0.631     Valid Loss: 0.78      Valid Accuracy: 0.792
No. epochs: 8,     Training Loss: 0.33      Valid Loss: 0.771     Valid Accuracy: 0.797
No. epochs: 8,     Training Loss: 0.33      Valid Loss: 0.771     Valid Accuracy: 0.797
No. epochs: 9,     Training Loss: 0.083     Valid Loss: 0.761     Valid Accuracy: 0.799
No. epochs: 9,     Training Loss: 0.083     Valid Loss: 0.761     Valid Accuracy: 0.799
No. epochs: 9,     Training Loss: 0.493     Valid Loss: 0.762     Valid Accuracy: 0.796
No. epochs: 9,     Training Loss: 0.493     Valid Loss: 0.762     Valid Accuracy: 0.796
No. epochs: 10,    Training Loss: 0.249     Valid Loss: 0.758     Valid Accuracy: 0.799
No. epochs: 10,    Training Loss: 0.249     Valid Loss: 0.758     Valid Accuracy: 0.799
No. epochs: 10,    Training Loss: 0.644     Valid Loss: 0.758     Valid Accuracy: 0.8
No. epochs: 10,    Training Loss: 0.644     Valid Loss: 0.758     Valid Accuracy: 0.8
```

- After 10 epochs of training, the ResNet34 model was able to obtain 0.644 training loss and 0.758 testing loss, as illustrated in the table and graph above. Compared to the previous loss vs.
- Small gap between the training and the testing loss could be interpreted as the model's ability to perform similarly regardless of training/testing set it uses. In ResNet34's case, the training loss was much higher than the testing loss during the beginning stages of the learning phase.
- This meant that the model had room for improvement by learning more from the training data. After 10 epochs of fitting, both the training and testing loss were improved ResNet34 was successful. This did not occur during the 10 epochs of model fitting. For ResNet34 we have used pytorch and cuda. PyTorch provide useful abstractions to reduce amounts of boilerplate code and speed up model development.
- The tensor object created in this way is on the CPU by default. As a result, any operations that we do using this tensor object will be carried out on the CPU. This ability makes PyTorch very versatile because computations can be selectively carried out either on the CPU or on the GPU.

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
```

- ResNet34 showing top 5 predication of the car. As shown in following figure:

```
In [58]: cardir='/BMW 3 Series Sedan 2012/06582'
         plot_solution(cardir, model)
```



## 3.2 Mobile Net

We shall be using Mobilenet as it is lightweight in its architecture. It uses depthwise separable convolutions which basically means It uses depthwise separable convolutions which basically means it performs a single convolution on each colour channel rather than combining all three and flattening it.

This has the effect of filtering the input channels. Or as the authors of the paper explain clearly: “For MobileNets the depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs the depthwise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size”.

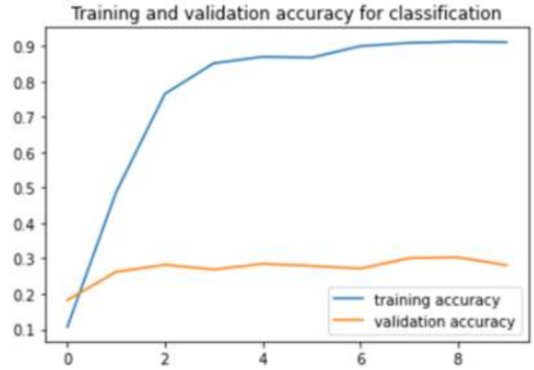
Table 1. MobileNet Body Architecture

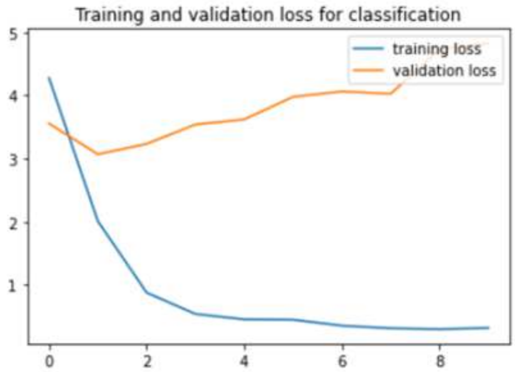
| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| 5× Conv dw / s1<br>Conv / s1 | $3 \times 3 \times 512$ dw<br>$1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$<br>$14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

- We have used the MobileNetV2 model with weights - ‘imagenet’ and Activation Function as Softmax.
- Used two Dense Layers 1024 Relu and 196 Sofmax.
- Created Bounding Box model.
- Created IOU function for finding out the difference between provided bounding box and predicted bounding box.
- For Bounding Box Model, we achieved 19% accuracy with 50 Epoch.

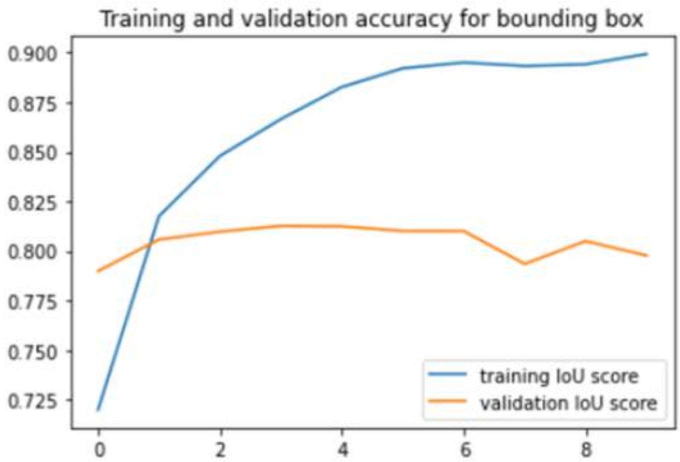
```
Epoch 40/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1527 - accuracy: 0.9636 - val_loss: 17.9163 - val_accuracy: 0.1903
Epoch 41/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1630 - accuracy: 0.9652 - val_loss: 16.9450 - val_accuracy: 0.1962
Epoch 42/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1889 - accuracy: 0.9646 - val_loss: 16.9776 - val_accuracy: 0.1992
Epoch 43/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1402 - accuracy: 0.9707 - val_loss: 16.2439 - val_accuracy: 0.2028
Epoch 44/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1042 - accuracy: 0.9767 - val_loss: 17.4875 - val_accuracy: 0.2059
Epoch 45/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1569 - accuracy: 0.9655 - val_loss: 18.5217 - val_accuracy: 0.2041
Epoch 46/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1155 - accuracy: 0.9750 - val_loss: 18.2261 - val_accuracy: 0.2047
Epoch 47/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1221 - accuracy: 0.9724 - val_loss: 19.8259 - val_accuracy: 0.2056
Epoch 48/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1422 - accuracy: 0.9706 - val_loss: 19.7778 - val_accuracy: 0.2066
Epoch 49/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1995 - accuracy: 0.9697 - val_loss: 21.5963 - val_accuracy: 0.2005
Epoch 50/50
255/255 [==============================] - 25s 97ms/step - loss: 0.1868 - accuracy: 0.9638 - val_loss: 19.9988 - val_accuracy: 0.1986
```

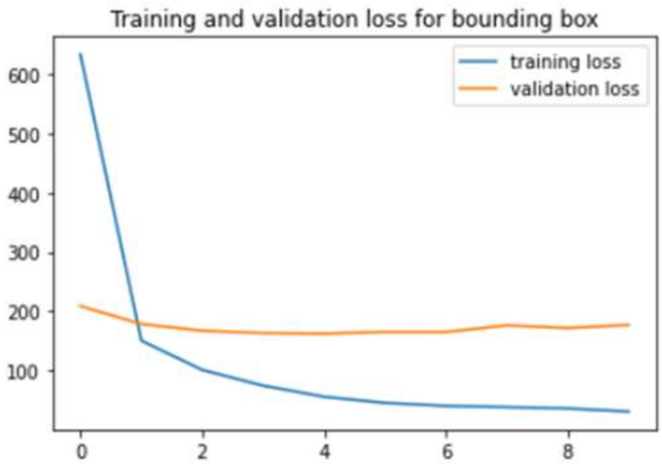
- We have achieved following Training Validation Accuracy and Loss as shown in following graph:



Bounding Box Accuracy




- We have used the MobileNetV2 model with weights from - ‘Imagenet’ and Activation Function as Softmax.
- For bounding box, we have taken the output of top layer from MobileNet
- Added two Layers on top of it
    - 2Dconv layer with kernel size 7X7 naming co-ords. This layer gives the output as (1,1,4) array
    - Reshape layer to generate co-ords for the bounding box.
- We have trained classification model with 10 epoch resulting maximum of 80% validation IoU and 92% training IoU.
- Created IOU function for finding out the difference between provided bounding box and predicted bounding box.
- We have got the following matrix for the IoU and loss

Classification Model:

```
Epoch 1/20
255/255 [==============================] - 390s 2s/step - loss: 5.1628 - accuracy: 0.0519 - val_loss: 3.7387 - val_accuracy: 0.1490
Epoch 2/20
255/255 [==============================] - 388s 2s/step - loss: 2.3693 - accuracy: 0.3968 - val_loss: 3.7691 - val_accuracy: 0.2052
Epoch 3/20
255/255 [==============================] - 394s 2s/step - loss: 1.1144 - accuracy: 0.6939 - val_loss: 4.0246 - val_accuracy: 0.2291
Epoch 4/20
255/255 [==============================] - 387s 2s/step - loss: 0.5627 - accuracy: 0.8370 - val_loss: 4.6468 - val_accuracy: 0.2408
Epoch 5/20
255/255 [==============================] - 389s 2s/step - loss: 0.4338 - accuracy: 0.8750 - val_loss: 4.9579 - val_accuracy: 0.2503
Epoch 6/20
255/255 [==============================] - 384s 2s/step - loss: 0.3323 - accuracy: 0.9085 - val_loss: 5.4263 - val_accuracy: 0.2405
Epoch 7/20
255/255 [==============================] - 377s 1s/step - loss: 0.3213 - accuracy: 0.9031 - val_loss: 6.0623 - val_accuracy: 0.2327
Epoch 8/20
255/255 [==============================] - 395s 2s/step - loss: 0.3558 - accuracy: 0.9032 - val_loss: 6.0866 - val_accuracy: 0.2379
Epoch 9/20
255/255 [==============================] - 361s 1s/step - loss: 0.3596 - accuracy: 0.9099 - val_loss: 6.4702 - val_accuracy: 0.2359
Epoch 10/20
255/255 [==============================] - 377s 1s/step - loss: 0.3109 - accuracy: 0.9155 - val_loss: 6.7739 - val_accuracy: 0.2389
Epoch 11/20
255/255 [==============================] - 384s 2s/step - loss: 0.3153 - accuracy: 0.9195 - val_loss: 7.5135 - val_accuracy: 0.2362
Epoch 12/20
255/255 [==============================] - 365s 1s/step - loss: 0.3748 - accuracy: 0.9088 - val_loss: 7.4548 - val_accuracy: 0.2355
Epoch 13/20
255/255 [==============================] - 370s 1s/step - loss: 0.3504 - accuracy: 0.9116 - val_loss: 7.8351 - val_accuracy: 0.2283
Epoch 14/20
255/255 [==============================] - 357s 1s/step - loss: 0.3453 - accuracy: 0.9194 - val_loss: 7.9049 - val_accuracy: 0.2375
Epoch 15/20
255/255 [==============================] - 363s 1s/step - loss: 0.3086 - accuracy: 0.9340 - val_loss: 8.5340 - val_accuracy: 0.2333
Epoch 16/20
255/255 [==============================] - 362s 1s/step - loss: 0.2692 - accuracy: 0.9347 - val_loss: 9.0235 - val_accuracy: 0.2257
Epoch 17/20
255/255 [==============================] - 379s 1s/step - loss: 0.2532 - accuracy: 0.9453 - val_loss: 10.0757 - val_accuracy: 0.2275
Epoch 18/20
255/255 [==============================] - 399s 2s/step - loss: 0.3606 - accuracy: 0.9261 - val_loss: 9.8534 - val_accuracy: 0.2271
Epoch 19/20
255/255 [==============================] - 392s 2s/step - loss: 0.3539 - accuracy: 0.9294 - val_loss: 10.7820 - val_accuracy: 0.2285
Epoch 20/20
255/255 [==============================] - 387s 2s/step - loss: 0.3449 - accuracy: 0.9335 - val_loss: 11.6147 - val_accuracy: 0.2246
```

Bounding Box Model:

```
Epoch 1/10
255/255 [==============================] - 308s 1s/step - loss: 1705.1549 - IoU: 0.6113 - val_loss: 210.0194 - val_IoU: 0.7913
Epoch 2/10
255/255 [==============================] - 290s 1s/step - loss: 161.9168 - IoU: 0.8132 - val_loss: 177.7499 - val_IoU: 0.8037
Epoch 3/10
255/255 [==============================] - 281s 1s/step - loss: 103.9605 - IoU: 0.8453 - val_loss: 167.7060 - val_IoU: 0.8099
Epoch 4/10
255/255 [==============================] - 282s 1s/step - loss: 73.2904 - IoU: 0.8683 - val_loss: 162.3988 - val_IoU: 0.8147
Epoch 5/10
255/255 [==============================] - 296s 1s/step - loss: 55.0930 - IoU: 0.8839 - val_loss: 164.1105 - val_IoU: 0.8103
Epoch 6/10
255/255 [==============================] - 281s 1s/step - loss: 44.2537 - IoU: 0.8942 - val_loss: 164.0532 - val_IoU: 0.8134
Epoch 7/10
255/255 [==============================] - 290s 1s/step - loss: 40.8739 - IoU: 0.8908 - val_loss: 166.1725 - val_IoU: 0.8112
Epoch 8/10
255/255 [==============================] - 277s 1s/step - loss: 39.8851 - IoU: 0.8890 - val_loss: 165.9476 - val_IoU: 0.8095
Epoch 9/10
255/255 [==============================] - 292s 1s/step - loss: 32.2053 - IoU: 0.8976 - val_loss: 170.2238 - val_IoU: 0.8069
Epoch 10/10
255/255 [==============================] - 291s 1s/step - loss: 24.4905 - IoU: 0.9060 - val_loss: 173.6742 - val_IoU: 0.8048
```

# 4. MILESTONE 3

## A. Pickled model from Milestone 2:

We have saved our model for MobileNet and ResNet34.

1. ResNet34

We build ReseNet34 model with 10 epochs. It took 3-hour time for training. This model is implemented in Pytorch. We have saved that model as shown in following screenshot:

**Step 5: Save the Model**

```
In [38]: # Saving: feature weights, new model.fc, index-to-class mapping, optimiser state, and No. of epochs
checkpoint = {'state_dict': model.state_dict(),
              'model': model.fc,
              'class_to_idx': train_data.class_to_idx,
              'opt_state': optimizer.state_dict,
              'num_epochs': epochs}

torch.save(checkpoint, '/content/drive/My Drive/AIML/Capstone Project/my_checkpoint1.pth')
```

```
In [ ]: json_file = model.to_json()
with open(project_path+"classification_mobilenet.json", "w") as file:
   file.write(json_file)
# serialize weights to HDF5
model.save_weights(project_path+"classification_mobilenet.h5")
```

**Step 6: Load the Model**

```
In [39]: # Write a function that loads a checkpoint and rebuilds the model

def load_checkpoint(filepath):

    checkpoint = torch.load(filepath)

    #model.load_state_dict(checkpoint['state_dict'])
    model.load_state_dict(checkpoint['state_dict'], strict=False)
    model.class_to_idx = checkpoint['class_to_idx']

    return model
```

```
In [40]: # Loading model
model = load_checkpoint('/content/drive/My Drive/AIML/Capstone Project/my_checkpoint1.pth')
# Checking model i.e. should have 196 output units in the classifier
print(model)

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

**2. MobileNet**

- We have saved two MobileNet Models one for Classification and Bounding Box.
- This model took 65 secs per epochs for Training.
- We have saved the model Weights in h5 format.
- Also We have saved the model in JSON format for further GUI development.

```
json_file = classification_model.to_json()
with open(project_path+"classification_mobilenet.json", "w") as file:
   file.write(json_file)
# serialize weights to HDF5
classification_model.save_weights(project_path+"classification_mobilenet.h5")
```

```
json_file = bb_model.to_json()
with open(project_path+"bb_mobilenet.json", "w") as file:
   file.write(json_file)
# serialize weights to HDF5
bb_model.save_weights(project_path+"bb_mobilenet.h5")
```

```
classify_file_name = project_path+"classification_mobilenet.h5"
classification_model.save(classify_file_name)
```

```
bb_file_name = project_path+"bb_mobilenet.h5"
bb_model.save(bb_file_name)
```

```
json_file_path = project_path+"classification_mobilenet.json"
json_file_path
```

**Classification Model:**

- We have taken input as resized training images (224,224,3)
- Also, we have taken the classes as OneHotEncoding.
- We have applied Dropout and Batch Normalization on top of it.
- This Model has 196 output neurons for generating 196 classes as output.
- This Model achieved maximum 32% accuracy after 20 epochs.
- There are 67,118,852 Total Parameters, 65,856,260 Trainable parameters and 2,262,592 Non-Trainable parameters as shown in following screenshot.

```
Conv_1_bn (BatchNormalization)  (None, 7, 7, 1280)   5120        Conv_1[0][0]
__________________________________________________________________________________________________
out_relu (ReLU)                 (None, 7, 7, 1280)   0           Conv_1_bn[0][0]
__________________________________________________________________________________________________
dropout_4 (Dropout)             (None, 7, 7, 1280)   0           out_relu[0][0]
__________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 7, 7, 1280)   5120        dropout_4[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 62720)        0           batch_normalization_4[0][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 1024)         64226304    flatten_2[0][0]
__________________________________________________________________________________________________
dropout_5 (Dropout)             (None, 1024)         0           dense_6[0][0]
__________________________________________________________________________________________________
batch_normalization_5 (BatchNor (None, 1024)         4096        dropout_5[0][0]
__________________________________________________________________________________________________
dense_7 (Dense)                 (None, 512)          524800      batch_normalization_5[0][0]
__________________________________________________________________________________________________
dense_8 (Dense)                 (None, 196)          100548      dense_7[0][0]
==================================================================================================
Total params: 67,118,852
Trainable params: 64,856,260
Non-trainable params: 2,262,592
__________________________________________________________________________________________________
```

**Bounding Box Model:**

- We have taken input as resized training images (224,224,3) with the bounding box.
- This Model has 4 output neurons for generating 4 co-ordinates for bounding box as an output.
- We have used 32 Batch sized for all the images.
- This Model achieved maximum 85% IOU (Intersection Over Union) after 10 epochs.
- We got very good IOU for Bounding Box.
- There are 2,508,868 Total Parameters, 250,884 Trainable Parameters and 257,981 Non-Trainable parameters as shown in following screenshot.

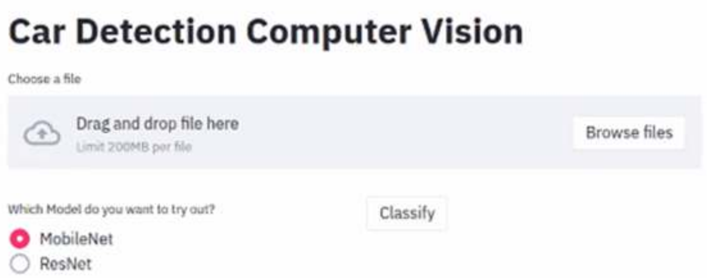
```
block_16_depthwise (DepthwiseCo (None, 7, 7, 960)    8640        block_16_expand_relu[0][0]
__________________________________________________________________________________________________
block_16_depthwise_BN (BatchNor (None, 7, 7, 960)    3840        block_16_depthwise[0][0]
__________________________________________________________________________________________________
block_16_depthwise_relu (ReLU)  (None, 7, 7, 960)    0           block_16_depthwise_BN[0][0]
__________________________________________________________________________________________________
block_16_project (Conv2D)       (None, 7, 7, 320)    307200      block_16_depthwise_relu[0][0]
__________________________________________________________________________________________________
block_16_project_BN (BatchNorma (None, 7, 7, 320)    1280        block_16_project[0][0]
__________________________________________________________________________________________________
Conv_1 (Conv2D)                 (None, 7, 7, 1280)   409600      block_16_project_BN[0][0]
__________________________________________________________________________________________________
Conv_1_bn (BatchNormalization)  (None, 7, 7, 1280)   5120        Conv_1[0][0]
__________________________________________________________________________________________________
out_relu (ReLU)                 (None, 7, 7, 1280)   0           Conv_1_bn[0][0]
__________________________________________________________________________________________________
coords (Conv2D)                 (None, 1, 1, 4)      250884      out_relu[0][0]
__________________________________________________________________________________________________
reshape (Reshape)               (None, 4)            0           coords[0][0]
==================================================================================================
Total params: 2,508,868
Trainable params: 250,884
Non-trainable params: 2,257,984
__________________________________________________________________________________________________
```

## B. Clickable UI based interface

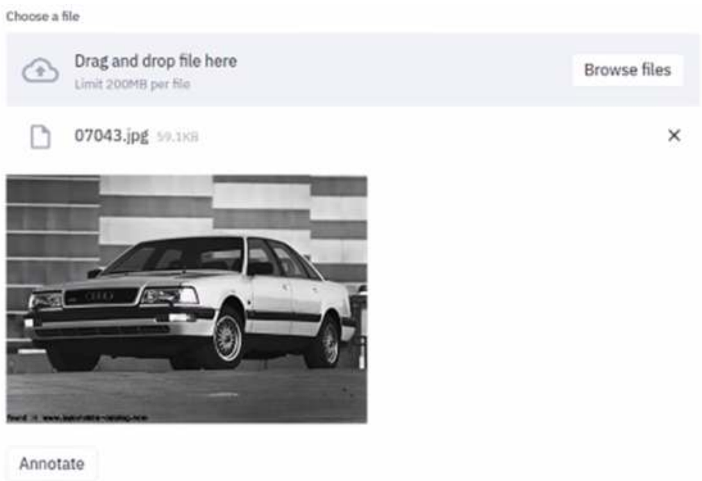
We have developed a clickable UI Interface in Docker Container Image.

Please Click Here  to access the GUI for Stanford Image Classification.

1. We have created container based on Python 3.8-slim-buster.
2. We have implemented ‘requirement.txt’ for installing the dependencies like tensorflow, streamlit, opencv-contrib-python etc.
3. This is implemented using Docker file.
4. We have used streamlit api for adding clickable buttons and showing Images and Bounding Box.
5. ‘App.py’ is the streamlit api server.
6. ‘util.py’ is the implementation of function.
7. When We run the container, we are landing on following homepage:

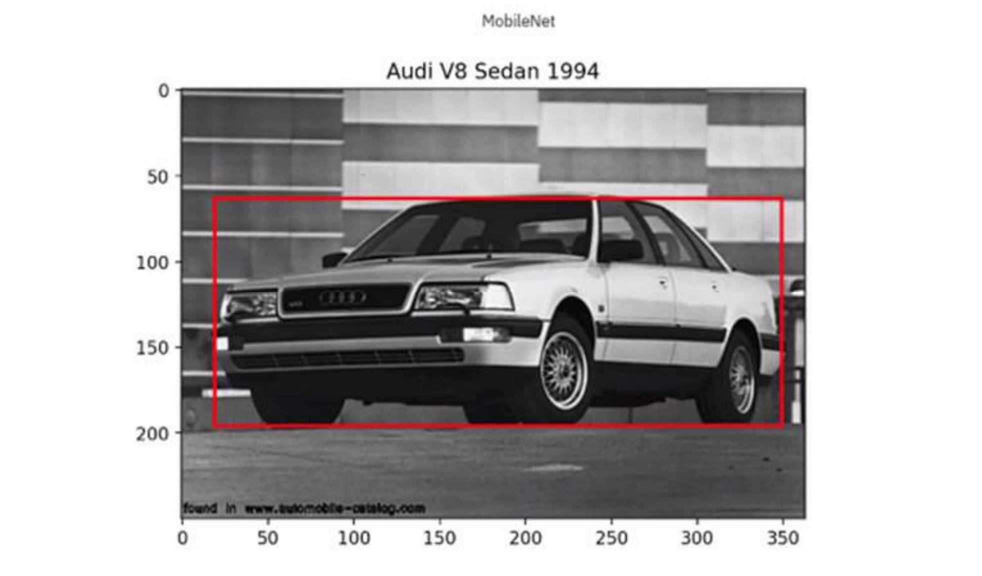



8. Click Browse file and upload an any image from test folder.
9. When we click ‘Annotate’ button, it mapped the Image Name to the browsing image as shown in following screenshot.

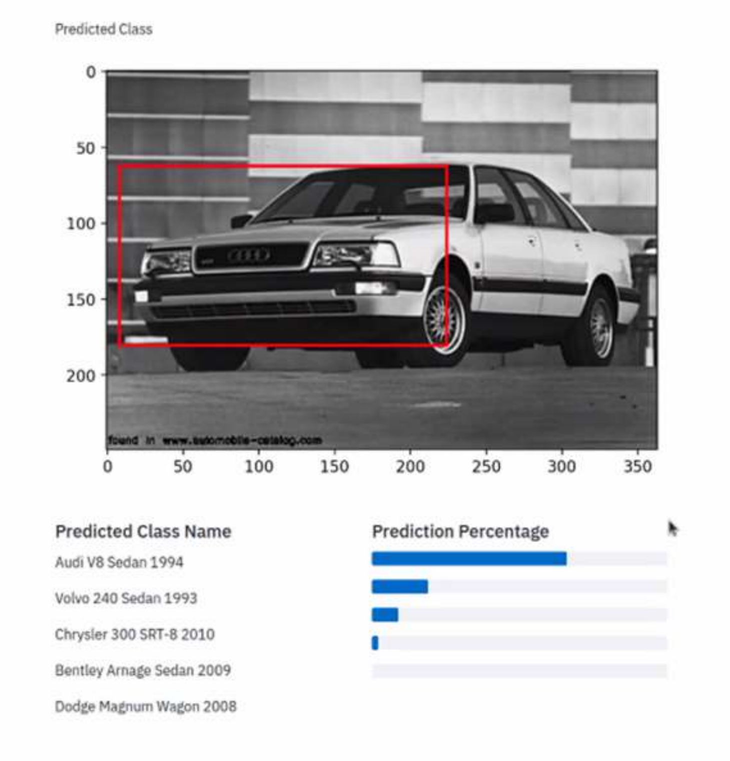


10. We have implemented this GUI using MobileNet Only.
11. Select a MobileNet Model from the option set and Click ‘Classify’ button.


12. When you click ‘Classify Image’, Original Images gets loaded with annotation as shown in following screenshot.



13. MobileNet Predicts Bounding Box and Top Five Image Classes predication as shown below:



# 5. CONCLUSIONS

The performances achieved in the ResNets and MobileNet providing top five predication to classify the Stanford Car.

Please Access our project using this github repository.