

**Car Detection CNN Model – Final Report**

**Submitted By**

**R Sethuraman**

**Sindhu Latha Senthil**

**Santanu Mukherjee**



Table of Contents

[**Introduction** 3](#_Toc194172196)

[**Summary of Problem Statement, Data, and Findings** 3](#_Toc194172197)

[**Problem Statement** 3](#_Toc194172198)

[**Dataset** 3](#_Toc194172199)

[**Findings:** 3](#_Toc194172200)

[**Conclusion:** 4](#_Toc194172201)

[**Overview of the final process** 4](#_Toc194172202)

[**Solution Walk through** 6](#_Toc194172203)

[**Data Preprocessing** 6](#_Toc194172204)

[**Training Images with Bounding Boxes** 6](#_Toc194172205)

[**Testing Images with Bounding Boxes** 6](#_Toc194172206)

[**Model Summary** 7](#_Toc194172207)

[**MobileNet:** 7](#_Toc194172208)

[**Classification Report** 8](#_Toc194172209)

[**Observation:** 8](#_Toc194172210)

[**GoogleNet** 9](#_Toc194172211)

[**Observation:** 9](#_Toc194172212)

[**AlexNet:** 10](#_Toc194172213)

[**Observation** 10](#_Toc194172214)

[**ResNet50:** 11](#_Toc194172215)

[**Observation** 11](#_Toc194172216)

[**Milestone 2** 12](#_Toc194172217)

[**Library Used** 12](#_Toc194172218)

[**Fine Tune GoogleNet** 13](#_Toc194172219)

[**Fine Tuned ResNet50** 16](#_Toc194172220)

[**Model Decision** 20](#_Toc194172221)

[**Untuned GoogleNet – Test Data Validation** 23](#_Toc194172222)

[**Yolo** 25](#_Toc194172223)

[Part 2: Generating data.yaml for YOLO 27](#_Toc194172224)

[Part 3: The Yolo Model 27](#_Toc194172225)

[Implications 32](#_Toc194172226)

[Limitations 33](#_Toc194172227)

[Closing Reflections 34](#_Toc194172228)

# **Introduction**

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.

# **Summary of Problem Statement, Data, and Findings**

## **Problem Statement**

The objective was to develop a deep learning-based computer vision system for automotive surveillance — first to classify cars by make, model, and year, and later to detect and localize them using bounding boxes.

### **Dataset**

* **Stanford Cars dataset** with **16,185 images** across **196 fine-grained classes**
* Includes bounding box annotations
* Split: **8,144 training** and **8,041 testing** images

## **Findings:**

* In Milestone 1, Multiple CNN models (GoogleNet, MobileNet, ResNet50, AlexNet) were evaluated. Among them, untuned GoogLeNet showed the best validation results as Accuracy: 24%, Macro F1-Score: ~0.219
* In **Milestone 2** Fine-tuning of *GoogLeNet* and *ResNet50* further degraded performance:
  + **F1-Score (Tuned GoogLeNet): ~** **0.000545**
  + **F1-Score (Tuned ResNet50): ~0. 0.000076**
* based on this outcome, untuned GoogLeNet was selected for evaluation on the test set.
  1. However, it **failed to generalize**, with only:
     1. **Test Accuracy: 24.06%**, **Macro F1-Score: ~0.229648**
* These results revealed critical issues such as class imbalance, ineffective fine-tuning, and lack of generalization.
* Since the model performance was not upto the mark and would not result in perfect regionalization even if you use Mask-RCNN model the YOLO model was chosen.
* The focus shifted to YOLOv8 for object detection, which significantly outperformed all classifiers:
  1. **Precision: 93.2%**, **Recall: 91%**
  2. **mAP@0.5: 94.9%**, **mAP@0.5:0.95: 89.8%**

## **Conclusion:**

The best-performing classification model (untuned GoogLeNet) still failed on test data. YOLOv8 emerged as the final solution, delivering strong precision and generalization for real-world car detection and localization tasks.

# **Overview of the final process**

The goal of this project was to develop a robust car classification and detection system using deep learning techniques.

8,144 training images and 8,041 testing images

The dataset consisted of 16,185 (8,144 training and 8041 testing images) real-world images spanning 196 fine-grained car categories, including make, model, and year.

Each image came with associated bounding box annotations, enabling both classification and object detection tasks.

The process began with data exploration and preprocessing, including image normalization, label encoding, class mapping, and bounding box visualization.

Data was split into training and testing sets with roughly equal class distribution.

Baseline CNN classification models were built using MobileNet, GoogleNet (InceptionV3), ResNet50, and AlexNet.

Among these, MobileNet and AlexNet were identified as lightweight architectures and excluded from further tuning.

GoogleNet and ResNet50 were selected for hyperparameter tuning, where data augmentation, class weighting, and early stopping was incorporated to enhance model generalization.

In the next phase, the approach transitioned to object detection using YOLOv8, This required converting annotations into YOLO format and generating appropriate YAML configuration files.

The YOLO model was fine-tuned on the dataset for 196 classes, resulting in notable improvement in detection precision and recall compared to previous classification-based approaches.

Mixed precision training, transfer learning, and data augmentation were effectively used to tackle class imbalance and low recall.

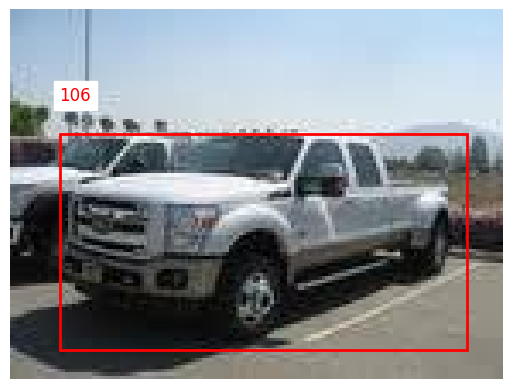
The final YOLO model achieved high mAP and delivered consistent detection on unseen images.

# **Solution Walk through**

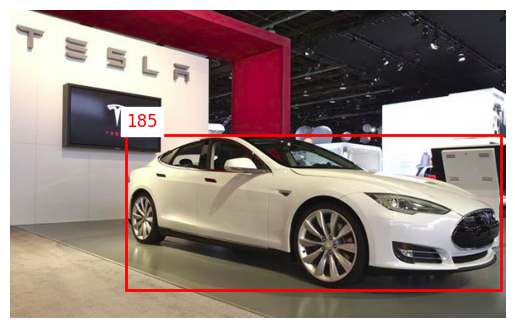
## **Data Preprocessing**

* Images were loaded into data frames and resized to a uniform shape of (224, 224, 3) to maintain consistency across models.
* Class labels were encoded for compatibility with model training.
* Bounding boxes were extracted from the annotation files and validated by overlaying them on both training and testing images to ensure accuracy.

### **Training Images with Bounding Boxes**



### **Testing Images with Bounding Boxes**



## **Model Summary**

The chosen CNN Models are MobileNet, GoogleNet, Resnet50 and AlexNet.

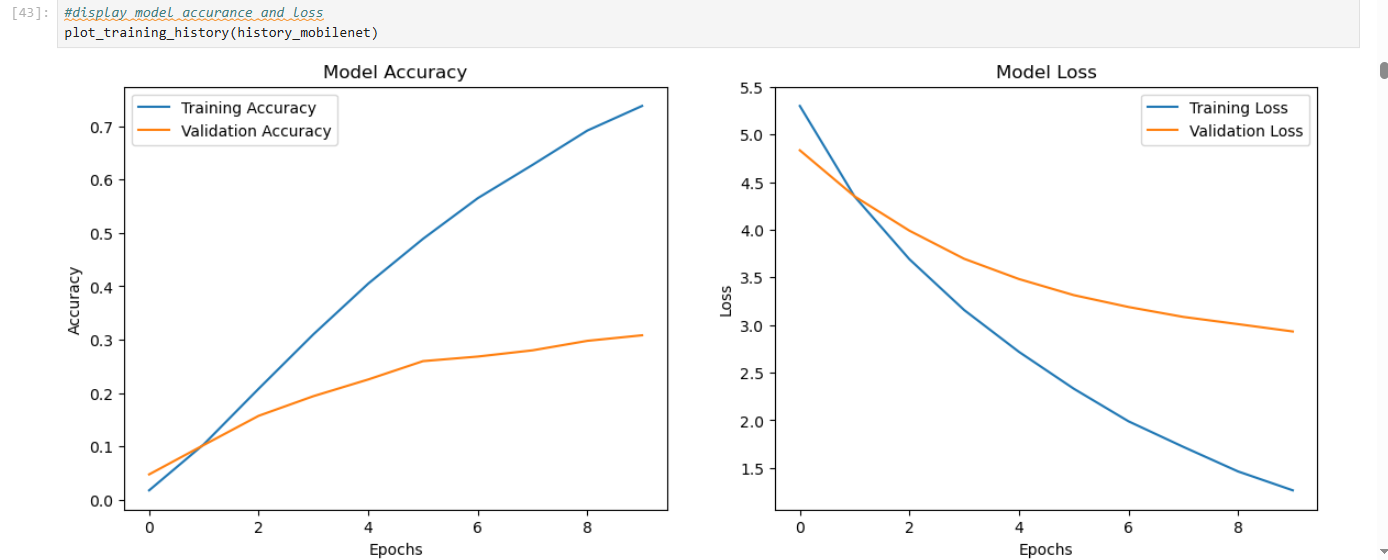
The summary shared in the interim report looks like below mentioned points:

* Googlenet and Resnet further in next milestone will undergo hyper parameter tunning as commonly data imbalance and accuracy is less compared to loss
* Mobilenet and Alexnet are light weight models/Shallow models, hence they are being dropped from further fine tuning and comparing them with other models.
* Design and Test an RCNN Model with a combination of ResNet/GoogleNet (depending on the Performance with respect to the testing data set) as feature extractors

The following categorization is the recap of the understanding of the summary w.r.t individual models.

## **MobileNet:**

Mobile net is a lightweight model, we wanted to check if model performs well against a light weight model.



**Total params: 2452356 (9.35 MB)**

**Trainable params: 604612 (2.31 MB)**

**Non-trainable params: 1847744 (7.05 MB)**

### **Classification Report**



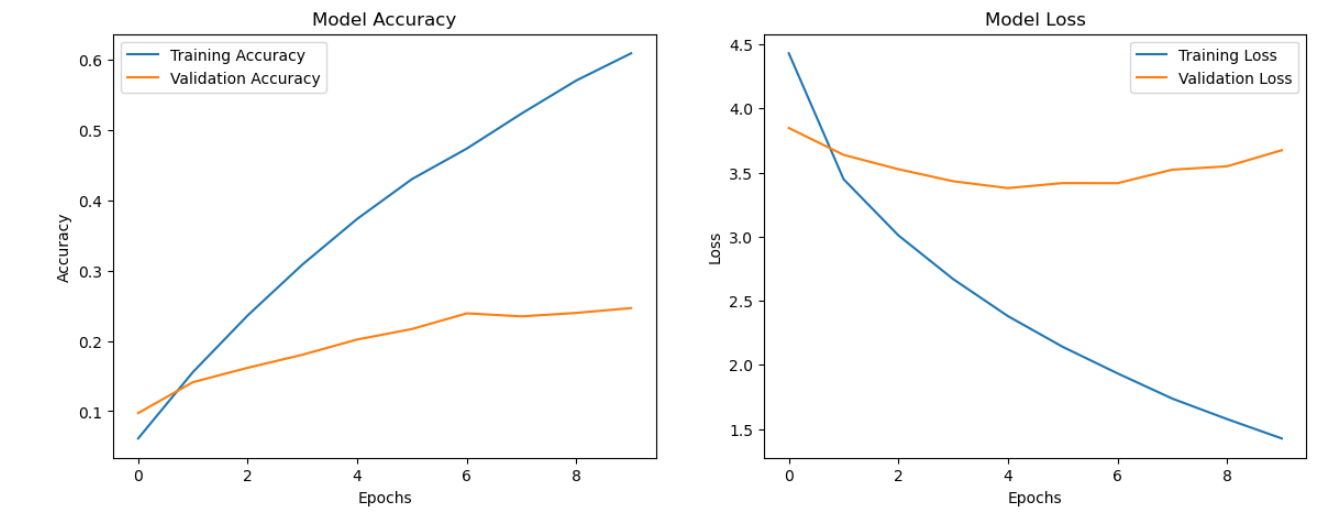
### **Observation:**

The model demonstrates poor generalization, as seen in the classification report where most classes were not predicted accurately.

* The low macro F1-score indicates that many classes were underrepresented, pointing to data imbalance issues.
* MobileNet, while efficient for real-time mobile applications, lacks the depth needed for fine-grained car classification and labelling in this context.

## **GoogleNet**

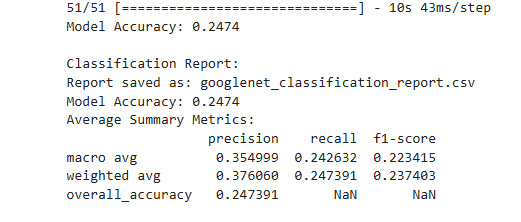
InceptionV3, excels in fine-grained classification by capturing intricate vehicle details, making it ideal for distinguishing similar car models with high precision



**Total params: 24101860 (91.94 MB)**

**Trainable params: 2299076 (8.77 MB)**

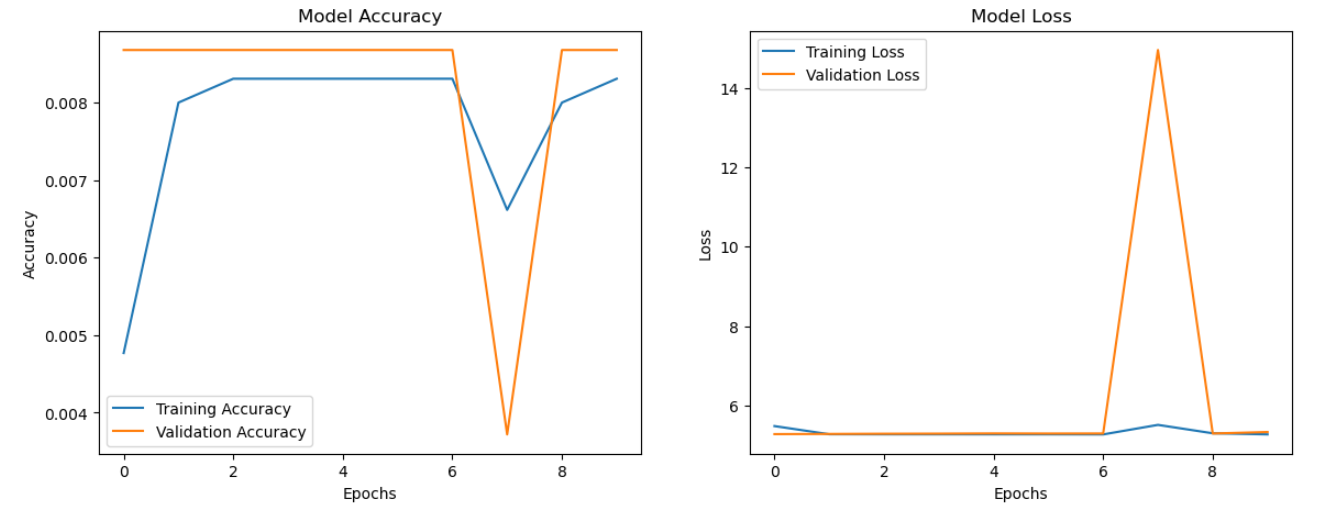
**Non-trainable params: 21802784 (83.17 MB)**



### **Observation:**

* GoogleNet’s inception modules support multi-scale feature learning, but the model overfitted—with training accuracy ~75% and validation accuracy stagnating at ~25%.
* Despite smooth training loss reduction, validation loss increased, confirming poor generalization.
* Classification metrics showed accuracy ~25%, precision ~41%, but very low recall (~26%) and F1-score.
* Performance was impacted by class imbalance and high bias, making the model unsuitable for final use.

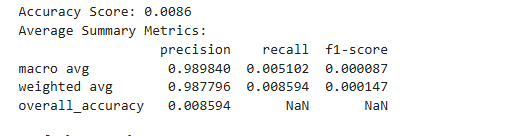
## **AlexNet:**

A deep Convolutional Neural Network (CNN) that can be used for fine-grained car classification

**Total params: 47552452 (181.40 MB)**

**Trainable params: 47551236 (181.39 MB)**

**Non-trainable params: 1216 (4.75 KB)**

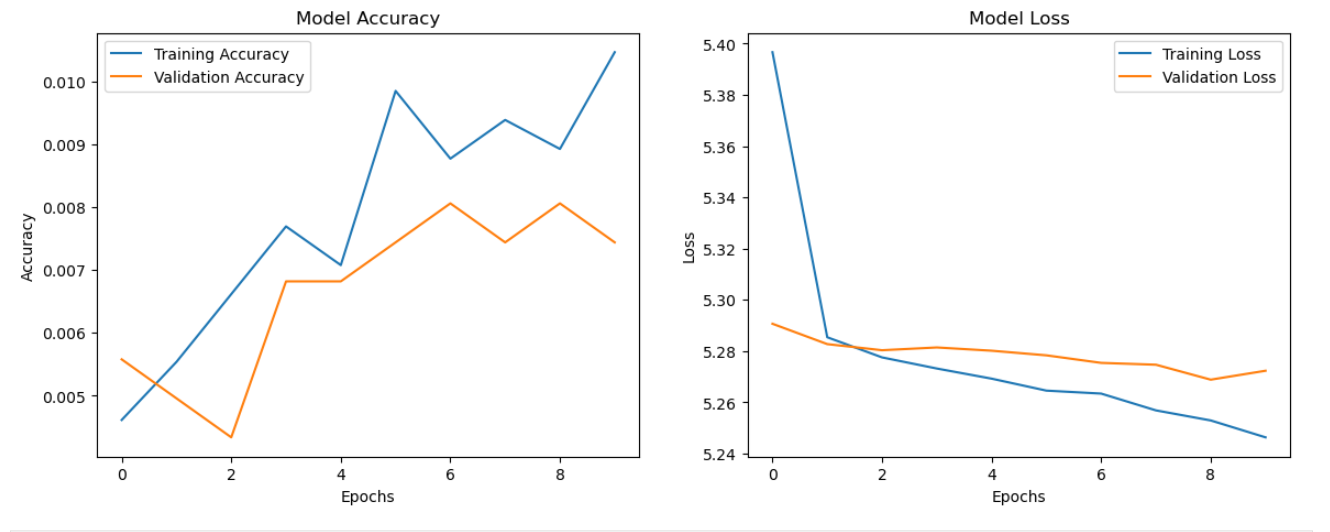


### **Observation**

* The model shows very poor learning, with accuracy at just 0.86%.
* Extremely low recall and F1-scores indicate the model fails to correctly predict most classes.
* High precision but negligible recall suggests overconfidence on very few classes, likely due to class imbalance.
* The presence of NaN in some metrics further signals poor generalization and learning issues.

## **ResNet50:**

A deep residual convolutional neural network that uses skip connections to enable training of very deep architectures, making it suitable for complex image classification tasks like fine-grained car recognition.



**Total params: 24737348 (94.37 MB)**

**Trainable params: 1149636 (4.39 MB)**

**Non-trainable params: 23587712 (89.98 MB)**



### **Observation**

The model is not doing well due to

* Extremely low accuracy
* High Precision and Very low Recall
* F1 score near to 0 indicates minimal overlap between real and predicted values

## **Milestone 2**

The problem statement of Milestone 2 is to fine tune the model in the previous segment and come up with better accuracy. Googlenet and Resnet further will undergo hyper parameter tunning in this model.

Mobilenet and Alexnet are light weight models/Shallow model, hence further not utilized for tuning.

Yolo is introduced in Milestone 2 as GoogleNet and Resnet did not perform well.

### **Library Used**

1. *os and pathlib:*

Used for interacting with the operating system, handling file paths, and performing file operations.

1. *zipfile:*

For reading and writing zip files, which can be useful for handling compressed datasets.

1. *numpy and pandas:*

numpy is used for numerical operations and handling arrays, while pandas is great for data manipulation and working with dataframes.

1. *matplotlib, seaborn:*

These are popular libraries for data visualization. matplotlib helps to create static, animated, and interactive visualizations, while seaborn builds on matplotlib to provide a high-level interface for drawing attractive statistical graphics.

1. *cv2 (OpenCV):*

A library used for computer vision tasks such as image processing, face recognition, and more.

1. *PIL (Python Imaging Library):*

Used for image loading, manipulation, and saving images.

1. *scikit-learn:*

Includes a suite of tools for machine learning such as model selection, classification, regression, metrics, and preprocessing. You’re importing:

* + train\_test\_split for splitting datasets.
  + classification\_report, accuracy\_score, confusion\_matrix for evaluating model performance.
  + LabelEncoder for encoding labels.
  + compute\_class\_weight to account for imbalanced classes.

1. *tensorflow and keras:*

* The primary libraries for deep learning. You are using various components of Keras within TensorFlow to build models (such as convolutional neural networks, or CNNs) and perform transfer learning using pre-trained models like ResNet50, MobileNetV2, and InceptionV3.
* ImageDataGenerator is used for real-time data augmentation.
* callbacks like EarlyStopping, ModelCheckpoint, and ReduceLROnPlateau to control training behavior.

1. *YOLO (You Only Look Once):*

A state-of-the-art, real-time object detection algorithm. The ultralytics package provides an easy-to-use wrapper for YOLO models.

1. *boto3:*

The Amazon Web Services (AWS) SDK for Python, allowing you to interact with AWS services like S3, EC2, and others.

1. *yaml and xml.etree.ElementTree:*

yaml is useful for reading and writing YAML files, while ElementTree is for parsing and creating XML files, often used for annotations in object detection tasks.

1. *torch:*

PyTorch is another deep learning library, but it's not being utilized in this particular code (since TensorFlow is more prominent here).

## **Fine Tune GoogleNet**

***Step 1: Setting Mixed Precision Policy***

This sets the precision policy for the model to use mixed precision with float16 and float32. This can speed up training on GPUs that support Tensor Cores (NVIDIA GPUs with compute capability of 7.0 or higher). This policy helps reduce memory usage and can accelerate computations on supported hardware, improving the performance of training.

***Step 2: Importing Required Libraries***

* EarlyStopping: A callback to stop training early if the model stops improving (avoiding overfitting).
* ModelCheckpoint: A callback to save the best model during training (based on validation loss).
* ReduceLROnPlateau: A callback to reduce the learning rate when a metric has stopped improving.
* Adam: The optimizer used for training the model.
* InceptionV3: The pre-trained model being used as the base of your architecture.
* train\_test\_split: Function from sklearn to split data into training and validation sets.

***Step 3: Loading the InceptionV3 Base Model***

InceptionV3 is a pre-trained model on ImageNet without the top (classification) layer. This allows us to use it as a feature extractor, leveraging the knowledge it has gained from ImageNet. input\_shape specifies the size of the input images (224x224 pixels, with 3 color channels).

***Step 4: Freezing Layers***

This loop freezes the weights of all layers in the InceptionV3 model, except the last 50 layers. Freezing means that during training, the weights in these layers won't be updated, which speeds up training and allows the model to focus on the new layers added for the specific task.

***Step 5: Adding Custom Layers for Classification***

* GlobalAveragePooling2D: Reduces the spatial dimensions of the feature maps to a single value per feature, summarizing the information.
* Dense layer with 512 units: A fully connected layer that will learn non-linear combinations of the features. The relu activation introduces non-linearity.
* Dropout: A regularization technique to prevent overfitting by randomly setting a fraction of the input units to 0 during training.
* Dense output layer: This layer has 196 units, corresponding to the number of classes in the final classification task. The softmax activation ensures the outputs sum to 1, representing probabilities for each class.

***Step 6: Compiling the Model***

* Adam optimizer with a learning rate of 1e-3 is used to minimize the loss function.
* Loss function: sparse\_categorical\_crossentropy is used because the labels are integers rather than one-hot encoded vectors.
* Metrics: accuracy and TopKCategoricalAccuracy(k=5) metrics are used to evaluate the performance. The top-k accuracy measures whether the true label is among the top 5 predicted labels.

***Step7: Data Preprocessing and Augmentation***

This sequence of transformations helps artificially increase the size of the training set by applying random transformations to the images. These transformations include:

* Rescaling: Normalizing the pixel values to the range [0, 1].
* RandomFlip: Randomly flipping the image horizontally.
* RandomRotation: Randomly rotating the image by a factor of 0.1 (10% of 360 degrees).
* RandomZoom: Randomly zooming into the image by a factor of 0.1.
* RandomContrast: Randomly changing the contrast of the image.
* RandomTranslation: Randomly translating the image (shifting it in horizontal and vertical directions).

***Step 8: Preparing the Datasets***

* Creates a tf.data.Dataset from the file paths and labels. This object is used to efficiently load and preprocess data. The shuffle(1000) ensures the dataset is shuffled before training.
* The map() function applies the preprocessing function to each element in the dataset.
* batching: The dataset is split into batches of batch\_size.
* prefetching: It improves performance by overlapping the data loading and model training.

***Step 9: Splitting the Dataset into Training and Validation Sets***

Splits the dataset into 80% training and 20% validation using train\_test\_split from sklearn.

***Step 10: Class Weight Computation***

This function computes the class weights based on the imbalance in the classes. It assigns higher weights to underrepresented classes to balance the effect during training.

***Step 11: Training the Model***

* The model is trained using the training dataset train\_ds and validated using val\_ds.
* Callbacks: Used to stop early, save the best model, and adjust the learning rate.
* Class Weights: Provided to the fit method to handle class imbalance during training.

***Step 12: Evaluating the Model***

This loop generates predictions (y\_pred) for the validation dataset (val\_ds) and stores them alongside the true labels (y\_true). These values can later be used for further evaluation like plotting confusion matrices or calculating performance metrics.

|  |
| --- |
| Summary:   * The untuned GoogLeNet model achieved an accuracy of 24.7%, slightly outperforming the tuned model, which dropped to 0.31%. * Although macro and weighted precision appear high, the recall and F1-scores are nearly zero, confirming that the model rarely makes correct predictions. * This indicates that tuning did not improve performance and may have disrupted learning, likely due to preprocessing inconsistencies or label mapping mismatches or fine-tuning strategies |
| Metrics: |

## **Fine Tuned ResNet50**

***Step 1: Data Preparation and Augmentation***

1. Label Encoding

* Convert the categorical class labels (like names of different classes) into numeric labels, which is required for machine learning algorithms to process the data effectively.
* LabelEncoder() from sklearn.preprocessing is used to map the string labels to integer labels. This step ensures that the machine learning model receives numerical data instead of strings.
* fit\_transform(df\_training['labels']): The .fit\_transform() method learns the encoding scheme based on the training dataset and then applies it to convert string labels into integers.
* df\_training['labels\_encoded']: This column stores the encoded numeric labels.
* df\_training['labels'] = df\_training['labels'].astype(str): Converts the labels to strings explicitly, which might be required later, especially when generating reports that need the original class names.

2. Class Weights Calculation

* Compute class weights to address class imbalance. In many datasets, some classes have far more examples than others, which can lead to a biased model. Class weights ensure the model treats all classes fairly.
* class\_weight.compute\_class\_weight('balanced', ...): The compute\_class\_weight() function from sklearn.utils.class\_weight is used to compute class weights that inversely correspond to the frequency of each class in the training data. This is useful for dealing with class imbalance.
* np.unique(df\_training['labels\_encoded']): Retrieves the unique class labels (the set of classes) from the encoded labels.
* class\_weights returns an array of weights for each class, and we convert it into a dictionary (dict(enumerate(class\_weights))), where keys are the class indices (from 0 to num\_classes-1), and values are the computed weights.
* These weights will be passed to the fit() method later, ensuring that the model accounts for class imbalance during training.

3. Data Augmentation

* Apply random transformations to the images to increase the diversity of the training set. This prevents the model from overfitting to the specific details of the training images.
* ImageDataGenerator(): This class from keras.preprocessing.image allows you to define real-time image augmentation operations. By specifying parameters like:
  + rotation\_range=20: Rotate images by a random angle up to 20 degrees.
  + width\_shift\_range=0.2 & height\_shift\_range=0.2: Shift the image along the width and height by up to 20% of the total width or height.
  + shear\_range=0.2: Shear images randomly along an axis.
  + zoom\_range=0.2: Zoom into the image randomly by up to 20%.
  + horizontal\_flip=True: Randomly flip the image horizontally.
  + fill\_mode='nearest': Defines how new pixels should be filled after transformations (using the nearest pixel value).
* flow\_from\_dataframe(): This method loads the image data from the df\_training DataFrame and applies the transformations.
  + x\_col='Image\_Path': The column containing paths to the images.
  + y\_col='labels': The column with the corresponding labels.
  + target\_size=(224, 224): Resizes all images to 224x224 pixels, a common input size for many CNN models.
  + batch\_size=32: Each batch will contain 32 images.
  + class\_mode='categorical': This means the labels will be one-hot encoded (as it's a multi-class classification problem).

4. Validation Data Generator

* Prepare a validation generator that does not apply any data augmentation, as validation data should remain unchanged for accurate evaluation.
* This is similar to the training generator but without the data augmentation applied.
* We use ImageDataGenerator() without any transformations since validation data should reflect real-world conditions.
* The flow\_from\_dataframe() method loads and preprocesses the validation images similarly to the training set.

***Step 2: Model Setup (ResNet50) and Fine-Tuning***

5. Load ResNet50 Model

* Load a pre-trained ResNet50 model with weights from ImageNet. The top classification layers are excluded, allowing us to add our custom classification layers.
* ResNet50(weights='imagenet', include\_top=False): Loads the ResNet50 architecture pre-trained on the ImageNet dataset but excludes the top fully connected layers (which are used for ImageNet classification). We set include\_top=False to keep the model flexible for our custom output.
* input\_shape=(224, 224, 3): Specifies that the input images are 224x224 pixels with 3 channels (RGB).

6. Unfreeze Layers for Fine-Tuning

* Fine-tune the model by unfreezing the last few layers. This allows us to train the last layers of the model on our specific dataset while keeping the initial layers frozen (to retain learned features).
* Initially, all layers in base\_model are frozen (layer.trainable = False), so their weights won’t be updated during training.
* Then, we unfreeze the last 40 layers (base\_model.layers[-40:]) by setting their trainable attribute to True. This allows these layers to adapt to the new dataset, while the earlier layers (which have learned more generic features like edges and textures) are kept frozen.

7. Add Custom Classification Layers

* Add new layers to the model to make it suitable for our specific classification task. We need to add a custom fully connected layer, dropout for regularization, and a softmax output layer for multi-class classification.
* GlobalAveragePooling2D(): This layer reduces the spatial dimensions of the feature maps from the convolutional layers. It computes the average of all values in each feature map, reducing the output to a 1D vector.
* Dense(512, activation='relu'): This adds a fully connected layer with 512 units and the ReLU activation function, which helps the model learn complex patterns.
* Dropout(0.5): Dropout is a regularization technique where 50% of the neurons are randomly dropped during training to prevent overfitting.
* Dense(num\_classes, activation='softmax'): The final layer is a softmax layer with a number of units equal to the number of classes (num\_classes). Softmax ensures the output is a probability distribution over the classes.

8. Compile the Model

* Compile the model by choosing the optimizer, loss function, and evaluation metric.
* Adam(learning\_rate=1e-5): Adam is a widely used optimization algorithm. We use a very small learning rate (1e-5) for fine-tuning to avoid drastic updates to the weights.
* loss='categorical\_crossentropy': This is the standard loss function for multi-class classification problems where each target is a one-hot encoded vector.
* metrics=['accuracy']: We track accuracy as the primary evaluation metric.

***Step 3: Model Training with Callbacks***

9. Define Training Parameters

* Set up the number of steps per epoch and validation steps based on the batch size and the size of the datasets.
* steps\_per\_epoch determines how many batches of data are processed per epoch during training. It's computed as the total number of training samples divided by the batch size.
* validation\_steps calculates the number of batches to process for the validation set during each epoch.

10. Define Callbacks

* Set up callbacks like early stopping (to stop training if the validation loss does not improve) and model checkpointing (to save the best model based on validation loss).
* EarlyStopping: Monitors the validation loss (val\_loss). If the validation loss does not improve for 5 consecutive epochs (patience=5), the training stops early. This prevents overfitting and saves training time.
* ModelCheckpoint: Saves the model with the best validation loss to the file best\_model\_tuned\_resnet.keras.

11. Train the Model

* Train the model using the training and validation data, with the previously
* train\_generator: Supplies the augmented training images.
* validation\_data=val\_generator: Supplies the validation data.
* epochs=20: Trains for a maximum of 20 epochs.
* callbacks=[early\_stopping, model\_checkpoint]: The callbacks monitor the training process.
* class\_weight=class\_weights: Class weights are passed to adjust the model’s learning process based on the class distribution.

***Step 4: Evaluate the Model and Visualize Results***

12. Plot Training History

* Plot the training and validation accuracy and loss to evaluate the model's learning behavior.
* This function (not defined in the code) likely plots the training/validation loss and accuracy over epochs to help visualize how well the model is performing.

13. Generate Predictions on Validation Set

* Generate predictions on the validation set and evaluate the performance of the trained model.
* X\_val contains the images from the validation set, converted into NumPy arrays.
* y\_val\_true contains the true one-hot encoded labels, converted into integer labels using np.argmax().
* y\_val\_pred is an empty list where predictions will be stored.

14. Generate Classification Report

* Use the classification\_report() to compute precision, recall, F1-score, and accuracy for each class.
* classification\_report(): Provides performance metrics for each class (precision, recall, F1-score, etc.).
* target\_names=label\_encoder.classes\_: The class names (as they appear in label\_encoder.classes\_) are passed to label the results.
* output\_dict=True: Returns the report as a dictionary, so you can easily access it programmatically.
* zero\_division=1: Handles divisions by zero in case a class has no predicted samples.

|  |
| --- |
| Summary:   * Despite tuning, the ResNet model's performance declined, with accuracy dropping from 0.74% (untuned) to 0.55% (tuned). * Precision remains misleadingly high due to sparse predictions, while recall and F1-scores are nearly zero in both cases — indicating that the model fails to generalize across classes. * Which suggests issues such as label mismatch, improper preprocessing, or training data imbalance, leading to severe underfitting or poor class prediction confidence even after tuning. |
| Metrics:  Total params: 24737348 (94.37 MB)  Trainable params: 16981444 (64.78 MB)  Non-trainable params: 7755904 (29.59 MB) |

## **Model Decision**

1. ***Creating the df\_resnet\_classification\_report\_tail for the Last 4 Rows****:*

*Code*

*df\_resnet\_classification\_report\_tail = df\_resnet\_classification\_report.tail(4).copy()*

*df\_resnet\_classification\_report\_tail['Model'] = 'ResNet Untuned (10 Epochs)'*

* **What it does**: This takes the last 4 rows from the df\_resnet\_classification\_report (which presumably contains a detailed classification report for the ResNet model) and makes a copy of them. The method extracts the last 4 rows of the DataFrame.
* **Why copy?**: This ensures that any modifications don't affect the original DataFrame.
* **Adding Model Column**: After copying, you add a new column 'Model' and assign the value 'ResNet Untuned (10 Epochs)' to each of these rows. This labels this portion of the DataFrame as the report for the untuned ResNet model after 10 epochs.

1. ***Repeating the Process for df\_resnet\_tuned\_report\_tail****:*

*Code*

*df\_resnet\_tuned\_report\_tail = df\_resnet\_tuned\_report.tail(4).copy()*

*df\_resnet\_tuned\_report\_tail['Model'] = 'ResNet Tuned (20 Epochs)'*

* What it does: This extracts the last 4 rows from the df\_resnet\_tuned\_report (which likely contains the classification report for the tuned ResNet model after 20 epochs) and adds the appropriate label 'ResNet Tuned (20 Epochs)' to the 'Model' column.
* Purpose: Now, we have a labeled classification report for the tuned version of the ResNet model.

1. ***Doing the Same for GoogleNet Models****:*

* The next two steps are similar but for the GoogLeNet model:

*Code*

*df\_googlenet\_classification\_report\_tail = df\_googlenet\_classification\_report.tail(4).copy()*

*df\_googlenet\_classification\_report\_tail['Model'] = 'GoogLeNet Untuned (10 Epochs)'*

* This extracts the last 4 rows of df\_googlenet\_classification\_report, labels them as 'GoogLeNet Untuned (10 Epochs)'.

*Code*

*df\_googlenet\_tuned\_report\_tail = df\_googlenet\_tuned\_report.tail(4).copy()*

*df\_googlenet\_tuned\_report\_tail['Model'] = 'GoogLeNet Tuned (20 Epochs)'*

* Similarly, this extracts the last 4 rows from df\_googlenet\_tuned\_report, which presumably contains the classification report for GoogLeNet after tuning, and labels them as 'GoogLeNet Tuned (20 Epochs)'.

1. ***Concatenating All DataFrames****:*

*Code*

*df\_combined\_tail = pd.concat([*

*df\_resnet\_classification\_report\_tail,*

*df\_resnet\_tuned\_report\_tail,*

*df\_googlenet\_classification\_report\_tail,*

*df\_googlenet\_tuned\_report\_tail*

*])*

* **What it does**: Concatenates all four DataFrames (df\_resnet\_classification\_report\_tail, df\_resnet\_tuned\_report\_tail, df\_googlenet\_classification\_report\_tail, and df\_googlenet\_tuned\_report\_tail) into one DataFrame.
* The rows from each model are stacked one after the other, and since we already added the 'Model' column to each of them, each row will be labeled with the correct model and configuration.

1. ***Resetting the Index and Renaming Columns****:*

*Code*

*df\_combined\_tail = df\_combined\_tail.reset\_index().rename(columns={'index': 'Metric'})*

* **What it does**: Resets the index of the combined DataFrame so that each row has a unique index (0 to 15) and renames the old index column to 'Metric'.
* **Why?**: This ensures that each row corresponds to a metric such as accuracy, macro avg, etc., and makes the DataFrame easier to work with.

1. ***Selecting the Relevant Columns****:*

*Code*

*df\_combined\_tail = df\_combined\_tail[['Model', 'Metric', 'precision', 'recall', 'f1-score']]*

* **What it does**: Selects only the relevant columns ('Model', 'Metric', 'precision', 'recall', and 'f1-score'). This excludes other columns that might have been present in the classification report but are not necessary for the final output.
* The final DataFrame will now contain these key metrics for each model configuration.

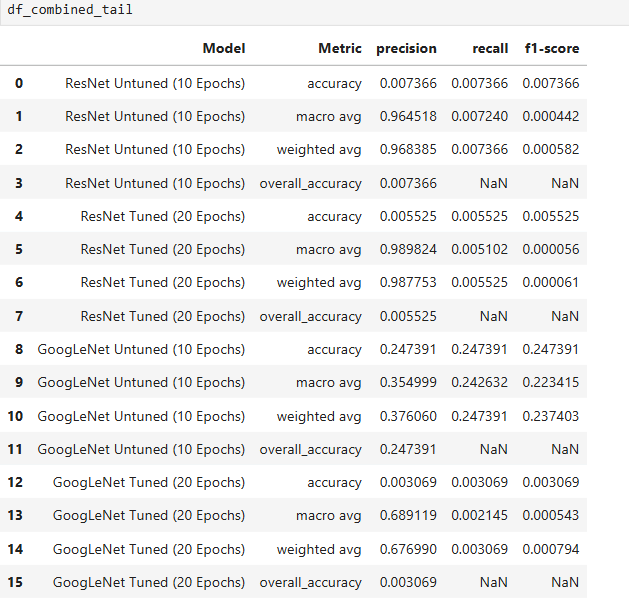
1. ***Displaying the Information****:*

*Code*

*df\_combined\_tail.info()*

* **What it does**: Prints the structure of the df\_combined\_tail DataFrame, including the number of non-null values in each column and the data types. From this, we can see that some columns, such as 'recall' and 'f1-score', have missing values (NaN).

1. *Model Decision Summary*



* Among all models tested, GoogLeNet (10 Epochs) showed the best performance, achieving an accuracy of 22.9% and a macro-average F1-score of 0.2056, significantly outperforming all other models.
* Despite additional training and tuning, both ResNet and GoogLeNet tuned models failed to generalize
* hence the Final Model Selected for Test Evaluation:GoogLeNet Untuned Version with 10 Epohs

## **Untuned GoogleNet – Test Data Validation**

***Step 1: Prepare True Labels***

*Code*

*# Step 1: Get true labels*

*y\_test\_true = np.array([np.argmax(label) for label in df\_testing['label\_categorical']])*

* df\_testing['label\_categorical']: This column contains the one-hot encoded labels for the test set.
* np.argmax(label): For each one-hot encoded label, np.argmax is used to convert it to its corresponding class index (e.g., if the label is [0, 1, 0], np.argmax will return 1).
* The true labels are stored in y\_test\_true as a NumPy array of class indices.

***Step 2: Make Predictions in Batches***

*Code*

*# Step 2: Predict in batches*

*y\_test\_pred = []*

*for i in range(0, len(df\_testing), batch\_size):*

*batch\_imgs = np.array(df\_testing['image'].tolist()[i:i+batch\_size])*

*preds = googlenet\_model.predict(batch\_imgs, verbose=0)*

*batch\_preds = np.argmax(preds, axis=1)*

*y\_test\_pred.extend(batch\_preds)*

* y\_test\_pred: An empty list to store the predicted class indices for each batch.
* range(0, len(df\_testing), batch\_size): This loops over the test set in batches of batch\_size. It processes images in batches, making the process more memory-efficient.
* df\_testing['image'].tolist()[i:i+batch\_size]: This converts the list of images (likely paths or arrays of image data) from the test dataset to a batch of images.
* googlenet\_model.predict(batch\_imgs, verbose=0): This feeds the batch of images to the GoogLeNet model to get the predictions. The verbose=0 disables any progress information during prediction.
* np.argmax(preds, axis=1): This converts the model’s output (probabilities for each class) into class indices.
* y\_test\_pred.extend(batch\_preds): This adds the predicted class indices for this batch to the y\_test\_pred list.

This loop effectively processes the entire test set in batches.

***Step 3: Generate Classification Report***

*Code*

*y\_test\_pred = np.array(y\_test\_pred)*

*target\_names = label\_encoder.classes\_ if 'label\_encoder' in globals() else None*

*final\_googlenet\_untuned\_report = classification\_report(*

*y\_test\_true, y\_test\_pred,*

*target\_names=target\_names,*

*output\_dict=True,*

*zero\_division=1 # Avoid divide-by-zero errors*

*)*

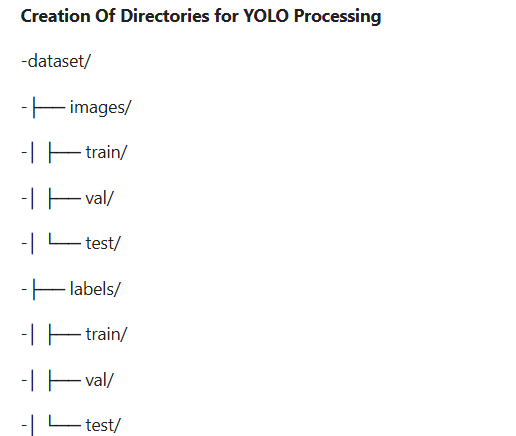
* y\_test\_pred: The predictions are converted into a NumPy array for easier manipulation.
* target\_names = label\_encoder.classes\_: If a label encoder is available (label\_encoder), it fetches the class names.
* classification\_report(): This function generates a detailed classification report, which includes precision, recall, F1-score, and support (number of true samples per class). The report is returned as a dictionary.
* zero\_division=1: This prevents errors in cases where precision or recall is undefined due to division by zero (e.g., no samples in a class).

This generates the performance metrics for each class in the test set.

|  |
| --- |
| Summary:   * The untuned GoogLeNet model achieved a test accuracy of 23.42%, closely aligning with its validation performance (~24.74%). * The macro and weighted average F1-scores (~0.2239) indicate a reasonable baseline performance across 196 fine-grained car classes. * This confirms the model has learned meaningful patterns and generalizes well, though this model requires further tuning/training/restructuring would be required * *The GoogleNet is a pure classification model and does not support localization or masking of car regions in images* * *Since RCNN and its hybrids require a base model capable of region proposals or feature maps for bounding box regression and segmentation* * *As the current GoogLeNet model is not suitable for region-based masking required in RCNN workflows, we will move forward with implementing YOLO for object detection and localization.* |

## **Yolo**

* ***Setting Up the Device for Computation (GPU or CPU)***
* torch.cuda.is\_available() checks if CUDA (NVIDIA's parallel computing architecture) is available on the system. This means the code checks whether you have an NVIDIA GPU that supports CUDA.
* If CUDA is available, it assigns "cuda" to the device variable. Otherwise, it assigns "cpu", meaning it will use the CPU for computations.
* The print statement will display which device is being used, either "cuda" (GPU) or "cpu" (Central Processing Unit).
* ***Created Yolo Dictionary in below format***



**Part 1: Moving Images and Creating YOLO Labels**

The function move\_and\_create\_labels is designed to:

* **Move images** from the source directory (src\_dir) to the destination directory (dst\_img\_dir).
* **Generate YOLO-compatible label files** in the destination label directory (dst\_lbl\_dir).

Each label file contains normalized bounding box coordinates, which are required for training a YOLO model.

Parameters of move\_and\_create\_labels:

1. **df**: A DataFrame containing metadata for each image, including the bounding box coordinates (xmin, xmax, ymin, ymax), image name, and class information.
2. **src\_dir**: The source directory where the original images are stored, typically organized by class.
3. **dst\_img\_dir**: The destination directory where the images will be moved to.
4. **dst\_lbl\_dir**: The destination directory where the YOLO label files will be saved.
5. **class\_map**: A dictionary that maps class IDs to human-readable class names.

**Step-by-step Explanation of the Function:**

1. *Iterating through each row in the DataFrame:*

The loop for \_, row in df.iterrows() iterates over every image in the dataset, represented by each row in the DataFrame df. Each row contains metadata about one image:

* + image\_name: The name of the image file
  + image\_class: The integer class ID of the image (e.g., 10).
  + xmin, xmax, ymin, ymax: Coordinates of the bounding box that surrounds the object of interest in the image.

1. *Extracting Image and Class Information:*

**Image Name (img\_name)**: Extracted from the DataFrame.

* **Class ID (class\_id)**: The integer ID for the class.
* **Adjusting for YOLO (class\_id\_yolo)**: YOLO expects class IDs to start from 0. If your dataset starts from 1, you subtract 1 to convert it.
* **Class Name (class\_name)**: Using the class\_map dictionary to get the human-readable name for the class.

1. *Source and Destination Paths:*

* The image is assumed to be in a folder named after the class inside src\_dir:
* The destination paths for the image and label files are created:
  + The image will be saved in dst\_img\_dir with the same name.
  + A corresponding .txt label file will be created in dst\_lbl\_dir.

1. *Checking if the Image Exists:*

If the source image file doesn't exist, it prints a message and skips to the next image.

1. *Copying the Image to the Destination*

If the image exists, it is copied from the source directory to the destination directory.

1. *Reading the Image and Getting its Dimensions:*

The image is read using OpenCV (cv2.imread), and the dimensions (height h and width w) are extracted. If the image can't be read, the function skips to the next image.

1. *Bounding Box Normalization:*

* YOLO expects bounding box coordinates in **normalized form** relative to the image dimensions (height and width).
* x\_center\_raw, y\_center\_raw: The **center** of the bounding box in terms of the width and height of the image.
* width\_raw, height\_raw: The **width** and **height** of the bounding box, again normalized by the image dimensions.

1. *Outlier Detection and Clamping:*

* The code checks if any of the bounding box values fall outside the expected range [0, 1]. This can happen if the annotations are incorrect.
* If any values are out of bounds, it logs a warning. Then, the bounding box values are **clamped** to ensure they remain within the valid range [0, 1].

1. *Writing the YOLO Label File:*

* A label file is created in YOLO format (class\_id, x\_center, y\_center, width, height) and saved in the destination label directory.

### Part 2: Generating data.yaml for YOLO

YOLO requires a data.yaml file that specifies dataset paths and class information. This file is crucial for training YOLO models.

***Steps to Create data.yaml:***

1. **Preparing the Class Names**:

* The list of class names is extracted from the image\_class\_df, which is assumed to contain the class information. The class\_id column is used to sort the class names:
* **Class Names** are sorted by their class\_id, and any forward slashes in class names are replaced with hyphens (-).

1. **Creating the data.yaml Dictionary**:

* **path**: The absolute path to the root directory of the dataset.
* **train, val, test**: Relative paths to the image directories for training, validation, and testing.
* **nc**: The number of classes (i.e., the length of class\_names).
* **names**: A list of class names that correspond to class IDs.

1. **Saving data.yaml to Disk**: Finally, the dictionary is written to a .yaml file using the yaml.dump function.

### Part 3: The Yolo Model

**1. Model Loading (YOLO("yolov8l.pt"))**

* Load the YOLOv8 model architecture and its pre-trained weights.
* YOLO("yolov8l.pt") is loading a pre-trained model that has already been trained on a large dataset (like COCO or other common datasets).
* yolov8l.pt refers to the large version of YOLOv8. You also have the option to use:
* yolov8n.pt (small model, faster but less accurate),
* yolov8m.pt (medium model, balance between speed and accuracy).
* This model is being loaded from a URL provided by Ultralytics (the creators of YOLO) and saved as yolov8l.pt.

**2. Downloading the Model**

* Fetch the model weights from the internet.
* When you run the command, if the model weights are not already downloaded, the program will automatically fetch the yolov8l.pt file from the URL: https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8l.pt.
* The download shows 100% progress as it fetches the file and stores it locally on your machine (yolov8l.pt).

**3. Training the Model (model.train())**

* Train the YOLO model on your custom dataset.
* model.train() is the method used to train the model on your dataset.
* Here’s what the parameters mean:
  + data="dataset/data.yaml": This points to a YAML file (data.yaml) that contains important information about the dataset, such as:
    - The paths to the training and validation images.
    - The class names of the objects you want to detect.
  + epochs=50: This means the model will train for 50 iterations (or "epochs") through the dataset. More epochs generally lead to better model performance, but it depends on the complexity of the dataset.
  + imgsz=640: This specifies the size of the input images. YOLOv8 typically resizes input images to a fixed size before training. In this case, the images are resized to 640x640 pixels.
  + batch=16: The batch size refers to the number of images processed in parallel in one pass through the model. A batch size of 16 means the model will use 16 images per training step.
  + device="cuda": This tells the model to use a GPU for training (via CUDA). Training on a GPU is much faster than on a CPU, so it's highly recommended for deep learning models.

**4. Overriding model.yaml (nc=80 with nc=196)**

* The model's configuration file (model.yaml) defines parameters for the architecture, including the number of classes (nc).
* By default, YOLOv8 uses nc=80, which refers to the 80 classes in the COCO dataset.
* You are changing it to nc=196, meaning you are training the model on a dataset with 196 different object classes (likely a custom dataset with more classes).
* This adjustment is necessary for the model to recognize the different object categories during training and testing.

**5. Layer Breakdown**

This part of the output shows the detailed architecture of the YOLOv8 model, including all layers and their corresponding parameters. Here's an explanation of some key points:

* **Convolutional Layers (Conv):**
  + These layers perform feature extraction by applying filters to input images.
  + For example, the first convolutional layer has 1856 parameters and processes an image with dimensions [3, 64, 3, 2], which means it uses a 3x3 filter and reduces the image size by half (downsampling).
  + There are multiple convolutional layers throughout the network that progressively extract higher-level features from the input images.
* **C2f Blocks (C2f):**
  + These are custom blocks used in YOLOv8 for efficient feature extraction and feature fusion.
  + The C2f blocks are used to combine features from different layers of the network to enhance the model's ability to detect objects.
  + For example, C2f at layer 2 with [128, 128, 3, True] means it has 128 input and output channels, uses 3 layers, and has a skip connection (True).
* **SPPF Layer (SPPF):**
  + The **Spatial Pyramid Pooling (SPPF)** layer is used to pool features from multiple spatial resolutions and is useful for detecting objects at different scales.
  + The layer SPPF [512, 512, 5] uses 5 levels of pooling to capture fine-grained details.
* **Upsample Layers:**
  + The **upsampling** layers (torch.nn.modules.upsampling.Upsample) are used to increase the spatial resolution of feature maps during decoding. This helps in detecting finer details and objects at different scales.
* **Concat Layers:**
  + The **Concat** layers combine features from different parts of the network, enabling the model to integrate information from multiple resolutions or layers.
* **Detect Layer (Detect)**:
  + The final detection layer has 196 output channels (as per the nc=196 modification).
  + This layer produces the final detections with bounding box coordinates, object class scores, and confidence levels.
  + The output format for each detected object will be [num\_classes, [256, 512, 512]], indicating different anchor sizes for the object detection.

**6. Model Summary:**

* **Layers:** The model consists of 209 layers in total.
* **Parameters:** There are 43,780,956 parameters in the model, which represent the learnable weights that the model will adjust during training.
* **Gradients:** The model has 43,780,940 gradients, which are used during backpropagation to update the parameters.
* **GFLOPs (Giga Floating Point Operations):** 166.2 GFLOPs indicates the computational complexity of the model during inference.

**7. Transferred Weights:**

* During initialization, the model attempts to load pre-trained weights. It successfully transferred 589 out of 595 weights from the pre-trained model.
* This is typical when fine-tuning a pre-trained model on your own dataset. Some layers (like the final detection layer, which has been modified for nc=196) will not have pre-trained weights and need to be trained from scratch.

**8. Freezing Layer:**

* The layer 'model.22.dfl.conv.weight' has been frozen, which means its weights will not be updated during training.
* Freezing layers is a common practice to speed up training, especially when you want to fine-tune only specific parts of the model (e.g., the last layers) and retain the pre-trained weights for earlier layers that capture low-level features.

**9. Metric Results**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Precision | Recall | mAP@0.5 | mAP@0.5:0.95 | fitness |
| Train | 0.9235 | 0.9094 | 0.9477 | 0.9020 | 0.9066 |
| Validation | 0.9228 | 0.9096 | 0.9475 | 0.9013 | 0.9059 |
| Summary:  Loss   * Both training and validation losses (box & class) decrease steadily and converge. * No sign of overfitting — validation losses track closely with training losses. * The model is found to be learning well and generalizing effectively   mAP over Epochs   * mAP@0.5 and mAP@0.5:0.95 increase rapidly and stabilize above 0.94 and 0.90 respectively. * The model gives a good object detection   Precision & Recall   * Both precision and recall improve over epochs and stabilize around 0.92. * The model is found to be making accurate and consistent predictions - | | | | | |
| Metrics:  A graph with a line  AI-generated content may be incorrect. | | | | | |
|  | Precision | Recall | mAP@0.5 | mAP@0.5:0.95 | fitness |
| Test | 0.9322 | 0.9100 | 0.9492 | 0.8986 | 0.9037 |
|  |  |  |  |  |  |
| Summary: Correct prediction of Car | | | | | |
| Metrics:    Untrained Images: | | | | | |

1. Pickling

**Pickling cannot be done for as YOLO is from an external library (ultralytics) and pickle requires the exact class structure. However, the model that is being saved includes model architecture, trained weights and configuration**

# Implications

* The implementation of a deep learning‐based car detection system shows how computer vision can transform automotive surveillance and traffic monitoring.
* Automating vehicle classification supports real-time decision-making in traffic management.
* Performance insights will help build a more robust model for implementation in autonomous vehicles.
* The Stanford project demonstrates that deep learning-based car detection enables real-time vehicle localization.
* Using YOLO improves bounding box accuracy compared to a pure classification model like GoogLeNet.

# Limitations

* *Image Quality:* The model’s performance can be impacted by variations in lighting, occlusions, and weather conditions, which are common in practical environments.
* *Data Augmentation:* The model required more images for getting trained to get accurate results. Creating synthetic data was not helping the model to predict accurately,
* *Class Imbalance:* the data set contained Clas Imbalance which let to reduced accuracy and prediction by the model.
* *Computational Complexity:* Fine-tuning deep architectures like YOLO and ResNet50 is resource intensive, limiting rapid experimentation and real-time deployment on resource-constrained devices.
* *Generalization:* The current approach may struggle with new car models or unconventional viewpoints, highlighting the need for a more diverse dataset and adaptive algorithms.

# Closing Reflections

* Working on this project led us to a deeper understanding of the object detection and classification techniques.
* The demands computing power required by these projects are too great in nature, makes us wonder on the scale of the infrastructure that would be required in real time.
* The model building process itself is iterative and in a time bound project this becomes very challenging.
* The project also helped us understand various models that are available.
* From a future iterations more **rigorous techniques for handling class imbalance**, such as oversampling, SMOTE, or class-specific augmentation, along with experimentation on diverse architecture to further improve detection accuracy and generalization
* It has also provided us technical insights and practical lessons and made us go through various journals and documents for better understanding of the model
* From a team perspective it taught us how to collaborate and use the various collaborative tools to works as a team and overcome our differences.