Remarks:

Acknowledgments

Thank you to Miss Nicole for guiding us on our assignment, thank you Jessica Li on Kaggle for providing the dataset, and thanks to all those researchers who wrote their papers that we used on our related works.

Abstract

The goal of this project is to utilize existing computer vision techniques to improve the accuracy and reliability of existing car classification models. Our group has extensively reviewed 8 papers on their approach to car identification and classification. We have proposed several modifications and improvements to enhance both the accuracy and effectiveness of currently accessible artificial neural networks, mainly Convolutional Neural Networks (CNNs). Based on our extensive research in the literature reviews, we found that CNN architectures such as ResNet and VGGNet are extremely useful in handling image processing tasks thanks to their capability in image identification and classification. For example, Ahmed S. Algamdi et al. (2023) achieved a whopping 99.7% accuracy using a VGG model, which is the best combination in terms of performance we had reviewed. We also discovered that transfer learning and deep learning can significantly increase the model's accuracy in classification while addressing problems such as gradient fading. Our study compares various CNN architectures, mainly ResNet and VGGNet to figure out which model is the best suited for car model classification. In this study, we hope to achieve improved accuracy and scalability in different circumstances by utilizing modern

preprocessing methods. After further discussions, our group decided to implement three different experiments that focused on pre-processing, data representation, and model evaluation, comparing our own pre-trained VGG-16 and AlexNet models with ResNet-101. We concluded that the best model combination which is VGG-16, took 40 mins to run while also achieving a commendable accuracy of 66.16%. Based on our findings, we recommend employing the VGG-16 model for the task of classifying car models. To achieve better accuracy and reliability in the future, additional research and testing in practical applications are necessary. This includes experimenting with different data augmentation methods to produce more varied images and experimenting with fine-tuning layers to extract more useful features during feature extraction.

Table of contents

1.0 Introduction

1.1 Background

Due to the rapid growth and production of cars over the past years, the demand for car classification and identification models have increased. In recent years, car manufacturers are under pressure to create more cars than in the past as the automotive industry continues to grow and evolve in order to meet the growing demands of customers (Alexandros Kelaiditis, 2023). Furthermore, given that car industries differ significantly throughout nations, this results in the production of both a substantial number of cars and a wide range of distinctive car types. The problem arises due to the large amount of car models available, making it a difficult task for humans to accurately identify the make and model of a car. The process of classifying cars becomes substantially more difficult due to several factors such as camera angles, lighting conditions, etc. As a result, traditional car classification methods are deemed not suitable due to them being prone to flaws and human errors as well as their inability to scale. Thus, they often come up short of the strict standards. With recent advancements of computer vision however, Convolutional Neural Networks (CNNs) have become a liable solution for most image processing tasks. CNNs are particularly effective for tasks such as object identification and classification as they are good at identifying distinctive patterns and characteristics within the images

. Additionally, the implementation of transfer learning and data augmentation methods will significantly improve the model's accuracy in classification while addressing problems such as gradient fading. With these advancements in computer vision, our group's aim for this project is to utilize existing computer vision techniques such as normalization and data augmentation to improve the accuracy and reliability of existing car classification models.

1.2 Research Goal

The primary goal of this research is to develop and validate a robust computer vision model capable of accurately identifying the make, model, and year of vehicles from images captured in various environmental conditions. This model aims to leverage advanced deep learning techniques, specifically convolutional neural networks(CNN), to process and analyze visual data, distinguishing between subtle differences in vehicle design and features.

1.3 Research Objectives

The objective of this research is to modify and evaluate existing artificial neural networks such as convolutional neural networks capable of identifying the make, model, and year of vehicles from images fed into the model. This involves:

1. Data preprocessing:
    a. Acquire a diverse and comprehensive dataset of vehicle images, including different makes, model, and years.
    b. Implementing preprocessing techniques like unfreezing classification layer and fine tuning higher level features to enhance image quality, normalize data, and address challenges such as occlusions, shadows, and reflections
2. Model comparison:
    a. Modify existing convolutional neural network (CNN) architecture using transfer learning in order to optimize the model for vehicle recognition tasks.
    b. Implement advanced machine learning techniques to improve the model's ability to distinguish between visually similar vehicle models.
        - Choosing the most suitable performance metrics

2.0 Related Works

Reference	Domain	Dataset	Feature Selection	Model	Performance
(Buzzelli & Luca Segantin, 2021)	Image classification of car models	Stanford cars dataset containing 16,185 image classification pairs across 196 classes	Data augmentation- random rotations of 20 degrees from anchor, random translations and color jittering. Classification method-Two-Step Cascade: This approach splits the problem into two sub-tasks: First predicting the car type(12 classes), then directing it to a specific classifier for make, model, and year(MMY) classification.	Multiple CNN architectures were used for testing such as Resnet-50	The best performing model achieved a Top-1 accuracy of 91.05% and a Top-5 accuracy of 98.88% which was achieved by Resnet-50.
(Simoni et al., 2021)	Improving Car Model Classification through vehicle keypoint localization	Pascal3D+ dataset that includes 10 sub-categories of car models. This dataset contains 4081 training images and	Leveraging both visual and pose information, leads to better results than using either models independently	Combining Stacked-Hourglass model and ResNeXt-101 model	By combining the two models together, the classification accuracy was improved by +1.3% over the baseline

		1024 testing images			
(Chigateri et al., 2022)	System for detecting car models based on machine learning	Stanford cars dataset	- Activation function: The activation function used is ReLU(Rectified Linear Unit) - Training Data: The network is trained on nearly 16,000 photos of cars from over 30 distinct automobile models and manufacturers	Resnet-34 CNN model	The model achieved 80 percent test accuracy and reduction of training loss from 4.0 to less than one
(Alghamdi et al., 2023)	Vehicle Classification Using Deep Feature Fusion and Genetic Algorithms	Stanford car dataset that contains almost 196 cars and vehicles	- Data augmentation like image flipping and standardizing image sizing with 8 classes each containing 1000 images after augmentation - Guided filter applied to improve colors of images - Feature extraction using 5 different combinations of convolutional layers with each combination containing 2-3 convolution layers. - Feature selection using Genetic Algorithm (GA) Optimization: Used to reduce feature	VGG 16	Achieve accuracy of 99.7%

			complexity. GA prioritizes important features while discarding less relevant ones - Classification with SVM Kernels: The selected features are classified using various SVM kernels, including Linear SVM, Cubic SVM, Quadratic SVM, and Medium Gaussian SVM		
(Stjepan Ložnjak et al., 2020)	Automobile classification using transfer learning on Resnet Neural Network architecture	Stanford cars dataset	- Image Features: Extracting relevant features from car images such as color, shape, texture, and edges. - Transfer Learning: Utilize pretrained models to extract high-level features - Dimensionality Reduction: Applying techniques such as Principal Component Analysis (PCA) or t-SNE to reduce feature dimensionality	Resnet-152 Neural Network	The model didn't change much. Best achieved accuracy was 88.82% on the 15th epoch
(Hassan et al., 2021)	An Empirical Analysis of Deep	Stanford cars dataset & VMMRDB dataset	- Data augmentation like random erasing, horizontal flip,	ResNet-152, Inception-ResNet-v2 Xception,	The model that had the best accuracy was Xception was 92.45% and ResNet-152 was 92%

	Learning Architectures for Vehicle Make and Model Recognition		random crop, resizing, sharpen, and rotation. - Texture Descriptors: Use texture features like Local Binary Patterns (LBP) or Haralick features to capture surface characteristics - Deep Learning Features: Utilize pre-trained deep learning models example CNN to extract high-level features from car images	DenseNet-201, MobileNet-v1, DenseNet-121	
(Shi Hao Tan et al., 2020)	Spatially Recalibrated CNN for Vehicle Type Recognition	BIT-Vehicle Dataset. Consists of 9850 images of different viewpoints	- T-distributed Stochastic Neighbor Embedding(TSNE)is used to project 4,096 dimensional features extracted from the penultimate fully connected layer into two dimensions for visual inspection	CaffeNet	The model achieved an accuracy of 94.17%
(Lu et al., 2022)	A novel part-level feature extraction method for fine-grained vehicle recognition	Stanford Cars & CompCars	- Data augmentation included standardizing image sizes and also horizontal flipping - Best results were derived from merging 4th and 5th convolutional layers	ResNet-101	The proposed model achieved an accuracy of 97.7%

			- Feature grouping module to aggregate the strongly correlated part information of local regions - Feature fusion module to complement multi-scale information of part-level features		
(Chen et al., 2021)	Learning to locate for fine-grained image recognition	Stanford car, Stanford dog, and CUB-200-2011	- Datasets were converted to COCO format and all images size were standardized to 500x500	ResNet-50	The proposed model achieved an accuracy of 94.08%
(Zhu et al., 2021)	Intelligent Transportation, Vehicle type recognition	Stanford Cars Database	- Global Average Pooling is utilized to combine features effectively and prevent overfitting - Residual connections included to avoid gradient loss - 1x1 convolutions to enhance the network's nonlinear feature extraction capability.	NIN VGGNet GoogLENet	The proposed model Network in Network achieved an accuracy of 97.2%

2.1 Gap Analysis

From these reviews we can observe several notable gaps in our computer vision-based car model prediction. To begin with, although a variety of feature selection techniques are utilized, there is a lack of exploration into more advanced methods such as automated machine learning (AutoML) for a more meta-learning approach. Such techniques will be necessary to integrate to possibly enhance our feature selection, thus improving the overall model performance. Training splits of the dataset could be introduced to generative models to augment them further.

Moreover, while high accuracy has been recorded in several models such as VGG 16 and DenseNet201, the lack of focus on other performance metrics such as model size, computational efficiency and inference time, may result in a less accurate model as it will not take into account the practical applicability of such models. Exploring lightweight models for deployment on edges could prove to be beneficial. Most of the research focused on broach vehicle recognition tasks instead of real world applications like real-time traffic monitoring. A model that is able to perform multiple related tasks at once could enhance the reliability of the results. Lastly, one prominent gap observed is the lack of ability for the model to recognise new or unseen car models without the aid of intensive additional training. To combat this the use of few-shot and zero-shot learning approaches should be taken into account in more research. By addressing these gaps, future research can significantly advance the field of computer vision-based car model recognition, pushing the boundaries of current methodologies and improving practical deployment in real-world scenarios.

In terms of the models we propose, VGG16 provides better performance in terms of accuracy for image classification tasks. This can be explained by the fact that it uses deeper architecture and more sophisticated feature extraction capabilities. For example, in a study done on a variety of different ImageNet datasets, the Top-5 Accuracy for the models VGG16 and AlexNet were 92.7% and 84.6% respectively. (Krizhevsky et al., 2012) This however, comes at a cost, VGG16 has a much higher computational complexity requirement resulting in longer training and inference times compared to AlexNet. Furthermore, studies that chose to do extensive preprocessing of the dataset like unfreezing classification layers or those who fine tuned higher level features for better performance of classifiers, personalizing the model to focus on our owned defined classes, improving the overall accuracy of the results.

2.2 Scope of research

By using Resnet-101 as our benchmark index, we will assess the performance of both sophisticated neural networks models in this study, which are AlexNet and VGG-16. In order to verify the validity and effectiveness of these models, we will be carrying out extensive testing and evaluation on a variety of datasets during the course of our assessment procedure. We will be applying several preprocessing methods such as image augmentation and normalization in order to guarantee a comprehensive evaluation. These preprocessing methods are used with the intention of enhancing the data input quality, examining precisely how the methods will affect AlexNet and ResNet-101's output performance in comparison to VGG-16. Besides that, we will look at the effects of various textual representation approaches, evaluating the way variations in data technique and arrangement might impact these models' results. We will be evaluating the effects of the preprocessing methods on AlexNet and VGG-16 to determine which augmentation techniques work most effectively in enhancing these model's performance and also demonstrate how these methods improve the predictive capability of these models. This comparison of AlexNet and VGG-16 to Resnet-101 will assist in defining the pros and cons of each model, which will guide future studies on neural network architectures along with its applications.

3.0 Methodology

3.1 Dataset

The data set that is seen the most in the literature review above is the Stanford car dataset. Stanford Car dataset consists of 16,185 images with 196 classes of different car models and makes up to the year 2012. Compared to the hundreds of datasets available out there, stanford dataset has a great coverage of a wide range of car make and models, spanning from different conditions and angles to help in the training and testing of specific model architectures. By using a dataset of this significance, the training and testing will be able to achieve better accuracy and consistency compared to datasets that have fewer images and variations for their images.

3.2 Data Preparation

3.2.1 Data Collection

The Stanford Cars Dataset above is available on multiple sources, including Kaggle, among others. It is ensured to be the right one without any additional sets that can hamper or interfere with our process unnecessarily.

3.2.2 Data Preprocessing

Data preprocessing involves transforming raw data into a format that is suitable for analysis and modeling, and consists of several steps that might or might not be necessary depending on our goal, such as data loading and cleaning, data augmentation, transformation, reduction, and feature engineering (Markus, 2024). One of the reasons why data preprocessing is essential is to ensure that the dataset used in the process is of the highest quality and proper for further processing. Raw data collected at first can have many errors, inconsistencies, incompleteness, and other flaws that can result in improper and misleading results.

Furthermore, raw data collected can have too many dimensions and be too complex for our intended process. It is often high dimensional, noisy, and heterogenous, making it way harder to directly analyze in this stage. This is why data preprocessing can come in handy, by using preprocessing techniques called dimensionality reduction, or feature scaling, so the overall data can be simplified enough and sufficiently complex for proper processing by our model.

Moreover, since our goal in this research is to use computer vision and classification for primarily images, we need to perform the technique of image preprocessing before we carry on. The technique of image preprocessing itself refers to the manipulation and engineering of raw image data given and turning it into a workable and meaningful format for our model. This allows us to remove and get rid of unwanted, unneeded features and distortions, and emphasize on specific attributes and qualities that we want to focus on for our model to process. These are very crucial to be implemented before feeding the image data into the machine learning models.

For all our data preprocessing, The following steps were undertaken sequentially:

Data Loading

- The intended dataset with available images is downloaded to a local machine and loaded into the environment by using appropriate deep learning libraries that are well known and well documented, such as PyTorch or TensorFlow. For example, we can import TensorFlow libraries by 'import tensorflow', then we can use 'tf.keras.preprocessing.image_dataset_from_directory' and put it into a train dataset variable.
- The image dataset has a certain path in our directory that needs to be located accurately and passed down to the functions to be preprocessed. Also, labels refer to the attributes or classes contained in the dataset that the image belongs to. The function 'tf.keras.preprocessing.image_dataset_from_directory' in TensorFlow automatically handles the mapping of the image and their corresponding labels based on the directory structure.

Data Augmentation:

To enhance the robustness of the model, data augmentation techniques were applied to the training images. This included transformations such as:

- Rotation - Rotation of the image refers to turning the image around its center by some degree resulting in different angles. This will help the model to process the image of cars orientation invariantly.
- Horizontal and vertical flipping - Flipping of the image is very beneficial as it can horizontally and vertically flip and change the car's position, helping the model to learn about the car regardless of side.
- Random cropping and scaling - Random cropping involves selecting a random portion of the image and resizing it back to the desired size. Scaling adjusts the size of the image randomly.
- Color jittering (adjusting brightness, contrast, saturation) - Color jittering involves randomly changing the brightness, contrast, saturation, and hue of the image to make the model robust to color variations.

These techniques help in increasing the diversity of the training data and prevent overfitting.

Image Resizing:

- All images were resized to a standard dimension (e.g., 224x224 pixels) to ensure uniformity across the dataset. This standardization is necessary for feeding images into convolutional neural networks (CNNs).

Normalization:

- Pixel values of images were normalized to a range of [0, 1] by dividing by 255. This helps in speeding up the convergence of the neural network training process.
- Additionally, mean subtraction and standard deviation scaling were applied based on the dataset statistics.

Label Encoding:

- Car class labels were encoded into numerical values using label encoding techniques. This is essential for training the machine learning model, which requires numerical inputs for categorical variables.

Train-Test Split Verification:

- The predefined train-test split was verified to ensure there is no data leakage. This ensures that the testing set remains completely unseen during the training process.

More importantly, in data preprocessing, we would have to unfreeze layers and this simply means to remove the classifications layer of pre-trained models as there are 196 car classes in this dataset and we will only need a selected number of classes of models in our assignment. Additionally, it is important to state that the dataset we have chosen have classes in the form of MMY (model-make-year) but in our assignment we will only be classifying cars based on models, and so any car images that do not fall under the classes we have trained the model under will automatically be classified as "rejection" class and this idea was inspired from one of our literature reviews (Buzzelli & Luca Segantin, 2021).

We can also fine tune our feature extractor by removing the higher feature layers while keeping the pre-trained lower level feature layers so that we can train the model we have chosen to identify the specific car model classes we need for the assignment and this will optimize the performance and accuracy of our classifier which will allow for higher accuracy score of classifying car models.

3.2.3 Image Representation and Feature Extraction

The technique of image representation involves taking the raw pixel values and transforming them into meaningful features that can be utilized by the machine learning models for training. To extract and get these meaningful features, this often can be achieved by techniques such as feature extraction and fine-tuning existing pre-trained models.

Pre-trained Models and Transfer Learning:

- Pre-trained models refers to machine learning models that are already existing and were trained for specific tasks on specific dataset, and are currently usable. These pre-trained models such as VGG16, ResNet101, InceptionV3 that were trained on large datasets like

ImageNet, can then be utilized to extract features from the Cars dataset. This is possible because these pre-trained models have already learned rich feature representations.

- Transfer learning refers to the technique that involves taking a pre-trained model and utilizing it for adaptation to a new dataset. This technique involves removing the final classification layer and replacing it with the new layer adapted to the new dataset, which in this case is suited to the Cars dataset.

Feature Extraction:

- Feature extraction involves using the convolutional base of a pre-trained model to extract features from the images. These features are then used as input to a new classifier.
- This can be done by removing the top layer of the pre-trained model and using the output of the convolutional layers as feature representations.

3.2.4 Data Exploration and Analysis

Class Distribution:

- We can analyze the distribution of the attributes or classes so that any imbalances in the dataset can be identified and corrected. The way to do this is to plot the number of images per class and ensure that the images in classes of the datasets are distributed equally.
- Some available techniques such as class-weight adjustment, undersampling, or oversampling can be considered if significant imbalances were detected.

Sample Visualization:

Random samples or top heads of the rows from the dataset are visualized and presented, so the quality and variety of the images can be inspected, along with their corresponding attributes.

This step is important for us so that we can improve our understanding of the dataset and identify any anomalies, errors, outliers, and hidden features.

3.3 Models

The models selected for this assignment are VGG16 and AlexNet and will be compared to ResNet-101 as the best model from literature review as a benchmark. The reason for ResNet-101 being chosen as the benchmark is that although VGG16 technically has the highest accuracy, ResNet-101 has the most usage.

3.4 Model Evaluation

This section compares and assesses the capabilities of AlexNet and VGG16, two well-known convolutional neural network (CNN) designs. These models have been chosen because of their historical relevance and broad usage in the computer vision community. The foundational benchmark is AlexNet, which transformed picture classification through its creative application of dropout regularization and ReLU activations. VGG16 provides a more intricate method of feature extraction and is renowned for its deeper architecture and usage of smaller convolutional filters. The objective is to comprehend the different benefits and shortcomings of these architectures through a thorough comparative analysis, with a focus on accuracy, computing efficiency, and suitability for various image recognition applications.

3.4.1 Introduction to AlexNet

AlexNet is a deep convolutional neural network (CNN) developed in 2012, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It was a pioneering model that made major contributions to the field of computer vision. By obtaining a top-5 error rate of 17.0% and a top-1 error rate of 37.5%, which was a notable improvement over earlier techniques, the model won the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), garnering major attention and recognition (Chen et al., 2022).

3.4.2 Architecture of AlexNet

AlexNet is composed of three fully connected layers before five convolutional layers. Convolutional layers employ max pooling for spatial reduction and ReLU activation to extract features using filters with different sizes and strides (Singh et al., 2022). These features are processed for classification by the fully connected layers, which result in an output layer with softmax activation for 1000-class predictions (Chen et al., 2022),

No.	Layer Type	Filters	Size/Stride	Output Size
1	Input	-	-	224x224x3
2	Convolutional	96	11x11 / 4 / 0	55x55x96
3	Max Pooling	-	3x3 / 2 / 0	27x27x96
4	Convolutional	256	5x5 / 1 / 2	27x27x256
5	Max Pooling	-	3x3 / 2 / 0	13x13x256
6	Convolutional	384	3x3 / 1 / 1	13x13x384
7	Convolutional	384	3x3 / 1 / 1	13x13x384
8	Convolutional	256	3x3 / 1 / 1	13x13x256
9	Max Pooling	-	3x3 / 2 / 0	6x6x256
10	Fully Connected	-	-	4096

11	dropout	-	-	4096
12	Fully Connected	-	-	4096
13	dropout	-	-	4096
14	Fully Connected	-	-	1000
15	Softmax	-	-	1000

(Chen et al., 2022)

3.4.3 Introduction to VGG16

VGG16 is a popular convolutional neural network architecture used for image classification and feature extraction, developed by the Visual Geometry Group in 2014. Key features of VGG16 include deep architecture with a depth of 16 layers inclusive of the convolutional layers, max pooling layers, and the fully connected layers. VGG16 boasts a simple and uniform structure where all convolutional layers use a 3x3 filter (Albashish et al., 2021). High accuracy is achieved by VGG16 on classification tasks such as ImageNet, while being highly effective for feature extraction which is required for all computer vision tasks. VGG16 is widely used for computer vision tasks because it supports weight sharing or transfer learning with pretrained weights on the ImageNet dataset, allowing the model to adapt to new tasks.

3.2.4 Architecture of VGG16

Diving deep into the architecture of VGG16 shows that VGG16 comprises 13 convolutional layers and 3 fully connected layers, organized into 5 blocks. Each block is followed by a max pooling layer, with the output for classification being softmax (Albashish et al., 2021).

No.	Layer Type	Filters	Size/Stride	Output Size
1	Input	-	-	224 x 224 x 3
2	Convolutional	64	3 x 3 / 1	224 x 224 x 64
3	Convolutional	64	3 x 3 / 1	224 x 224 x 64
4	Max Pooling	-	2 x 2 / 2	112 x 112 x 64
5	Convolutional	128	3 x 3 / 1	112 x 112 x 128
6	Convolutional	128	3 x 3 / 1	112 x 112 x 128
7	Max Pooling	-	2 x 2 / 2	56 x 56 x 128
8	Convolutional	256	3 x 3 / 1	56 x 56 x 256
9	Convolutional	256	3 x 3 / 1	56 x 56 x 256
10	Convolutional	256	3 x 3 / 1	56 x 56 x 256
11	Max Pooling	-	2 x 2 / 2	28 x 28 x 256
12	Convolutional	512	3 x 3 / 1	28 x 28 x 256

13	Convolutional	512	3 x 3 / 1	28 x 28 x 256
14	Convolutional	512	3 x 3 / 1	28 x 28 x 256
15	Max Pooling	-	2 x 2 / 2	14 x 14 / 512
16	Convolutional	512	3 x 3 / 1	14 x 14 / 512
17	Convolutional	512	3 x 3 / 1	14 x 14 / 512
18	Convolutional	512	3 x 3 / 1	14 x 14 / 512
19	Max Pooling	-	2 x 2 / 2	7 x 7 x 512
20	Fully Connected	-	-	4096
21	Fully Connected	-	-	4096
22	Fully Connected	-	-	1000
23	Softmax	-	-	1000 (Classes)

(Albashish et al., 2021)

3.4.5 Comparative Summary

When comparing the performance and depth of both models, AlexNet boasts 8 (5 convolutional, 3 fully connected) layers whereas VGG16 uses 16 (13 convolutional, 3 fully connected) layers. This means that while AlexNet architecture is shallower, this allows the architecture to be simpler. VGG16 has a deeper, and more complex architecture due to the increased number of hidden layers in the network. These architectural differences cause AlexNet to have only basic hierarchical feature extraction capabilities while VGG16 is capable of more advanced hierarchical feature learning. Another performance impact is the accuracy, AlexNet is respectable for its time, but proves less effective on complex datasets, while VGG16 can provide a high accuracy with better performances on more complex datasets.

AlexNet has a lower parameter count than VGG16. It is ~80 million on AlexNet while VGG16 has a parameter count of ~138 million. This lower number of parameters in AlexNet allows for the model to be faster and require less powerful hardware to function. When looking at VGG16 it has a greater training time and requires more powerful hardware in comparison.

Usability of the model is an important aspect when choosing a model for computer vision tasks. AlexNet has effective transfer learning while with less generalizable features when compared with VGG16 which has highly effective transfer learning capabilities with robust feature extraction. Looking at data augmentation support of both models, AlexNet pioneered the use of data augmentation techniques, and VGG16 is commonly used in conjunction with data augmentation. AlexNet has strong historic support from the community and developers while VGG16 has extensive current support and resources.

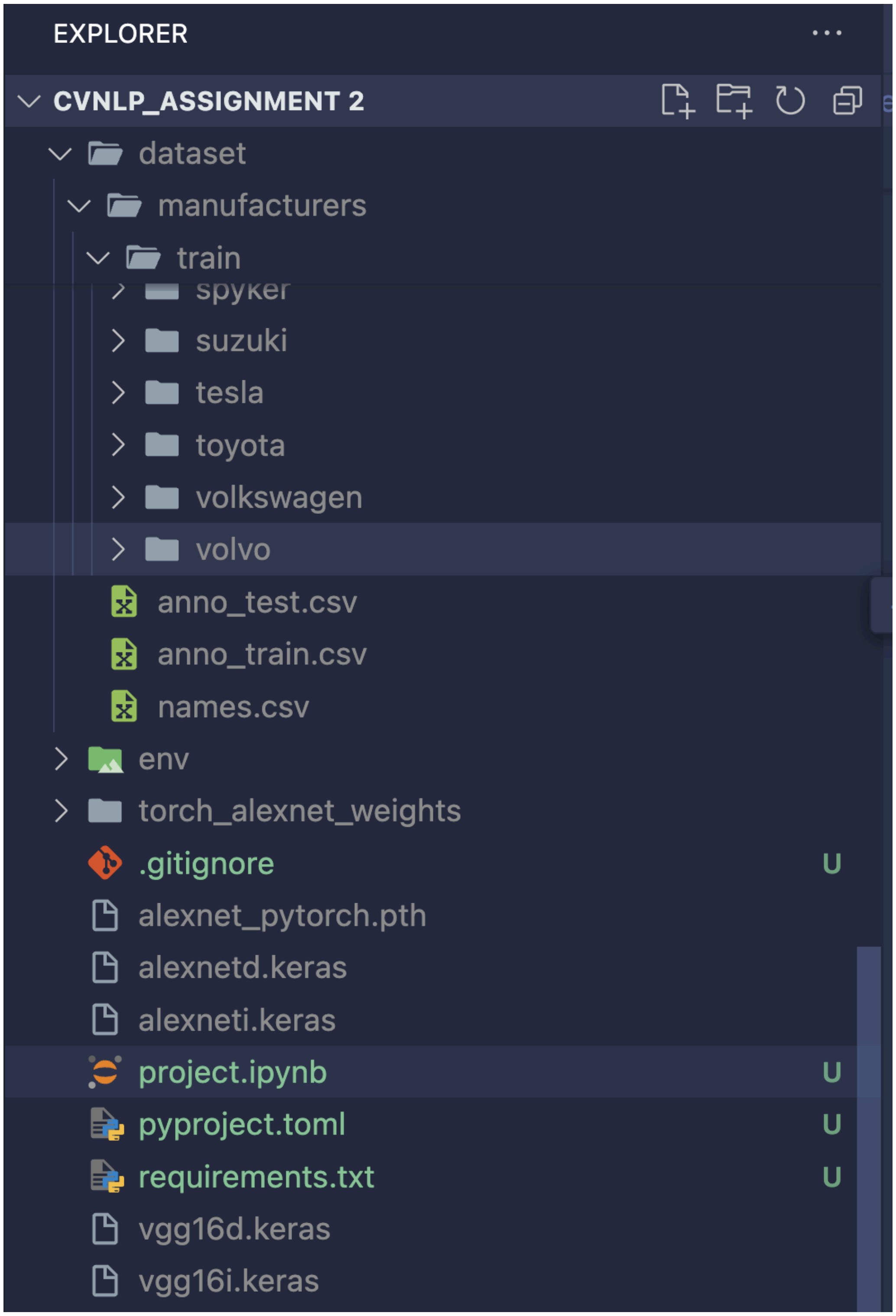
In conclusion, the choice of VGG16 and AlexNet for this comparative analysis emphasizes the variety and development of convolutional neural network architectures. AlexNet is not as good for sophisticated jobs due to its limited hierarchical feature extraction capabilities and lesser accuracy on complicated datasets, even though its simpler, shallower structure gives advantages in speed and lower hardware requirements. On complicated datasets, VGG16 excels at providing sophisticated hierarchical feature learning and improved accuracy thanks to its deeper design of 16 layers. VGG16 is the recommended option for complex computer vision problems due to its strong feature extraction and excellent transfer learning capabilities, even with its higher processing requirements and longer training durations. Furthermore, the research community's broad support for it now and its compatibility with modern data augmentation approaches reinforce its selection. To end the performance metrics we can use to measure our model's accuracy we can use F1-score and confusion matrix besides our precision and accuracy score.

3.5 Summary of methodology

To summarize our methodology, our model we've chosen is VGG16 and the reason we've chosen it is because it has a deep design of 16 layers and smaller filters which allow it to extract features at a more higher hierarchical level as compared to AlexNet and hence is generally more accurate in classifying classes. Next our data preprocessing process is very crucial since our chosen dataset, Stanford cars dataset has a size of 16,185 images and 196 classes and we need to carry out data augmentation as it helps prevent overfitting during training and helps the model have a more varied understanding of the existing car models in the dataset like for example we can provide cropped version of images at different parts of the car like it's headlights, bumper, car doors, logo or flipped versions and different degrees of rotation so that the model classifying accuracy is optimized. Afterwards we need to unfreeze the classification layers and fine tune the higher level features so that we can train the model properly to classify. **1.0 Implementation and Results**

4.0 Project Documentation

Folder Structure

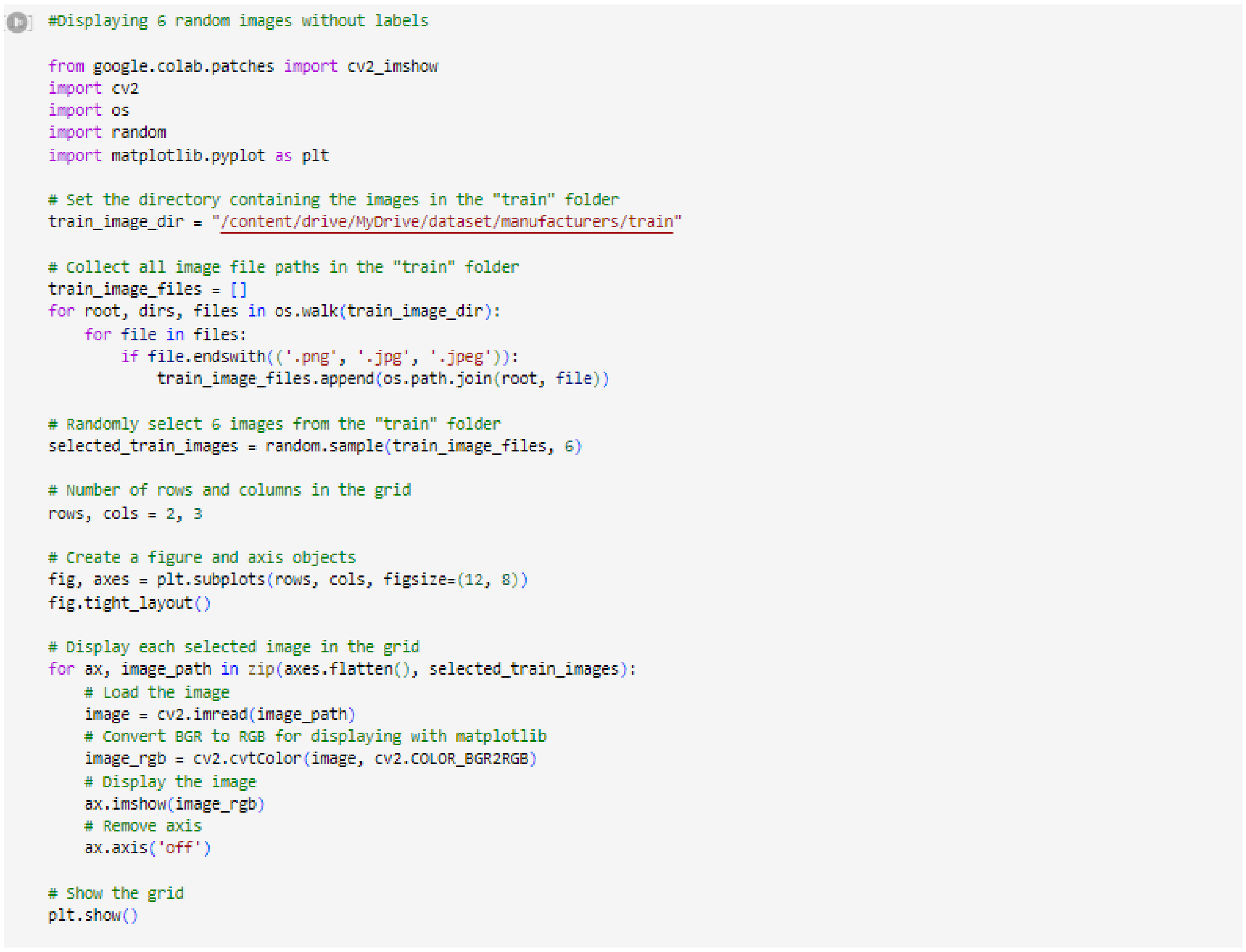


```
EXPLORER
CVNLP_ASSIGNMENT 2
  dataset
    manufacturers
      train
        spyker
        suzuki
        tesla
        toyota
        volkswagen
        volvo
    anno_test.csv
    anno_train.csv
    names.csv
  env
  torch_alexnet_weights
  .gitignore                U
  alexnet_pytorch.pth
  alexnetd.keras
  alexneti.keras
  project.ipynb             U
  pyproject.toml            U
  requirements.txt          U
  vgg16d.keras
  vgg16i.keras
```

- dataset/:
  - manufacturers/:
    - train/: Contains subfolders for each car manufacturer (e.g., spyker, suzuki, tesla, toyota, volkswagen, volvo). Each of these subfolders likely contains images of cars for training.
  - anno_test.csv: Annotations for the test dataset.
  - anno_train.csv: Annotations for the training dataset.
  - names.csv: Probably contains the names of the car manufacturers or classes.
- env/: Might be the environment setup files, such as virtual environment configurations.
- torch_alexnet_weights/: Directory that contains the weights extracted from the AlexNet PyTorch model, saved as .npy files.
- .gitignore: Standard Git file specifying files and directories to ignore in version control.
- alexnet_pytorch.pth: Saved PyTorch model file for AlexNet.
- alexnetd.keras, alexneti.keras: Saved Keras model files for AlexNet, alexnetd for the model trained without pretrained weights, and alexneti for the model with pretrained weights.
- project.ipynb: Jupyter Notebook file for the project, likely containing code for training, evaluating, and analyzing the models.
- pyproject.toml: Configuration file for the project, possibly specifying dependencies and other project settings.
- requirements.txt: Text file listing Python dependencies required for the project.
- vgg16d.keras, vgg16i.keras: Saved Keras model files for VGG16, vgg16d for the model trained without pretrained weights, and vgg16i for the model with pretrained weights.

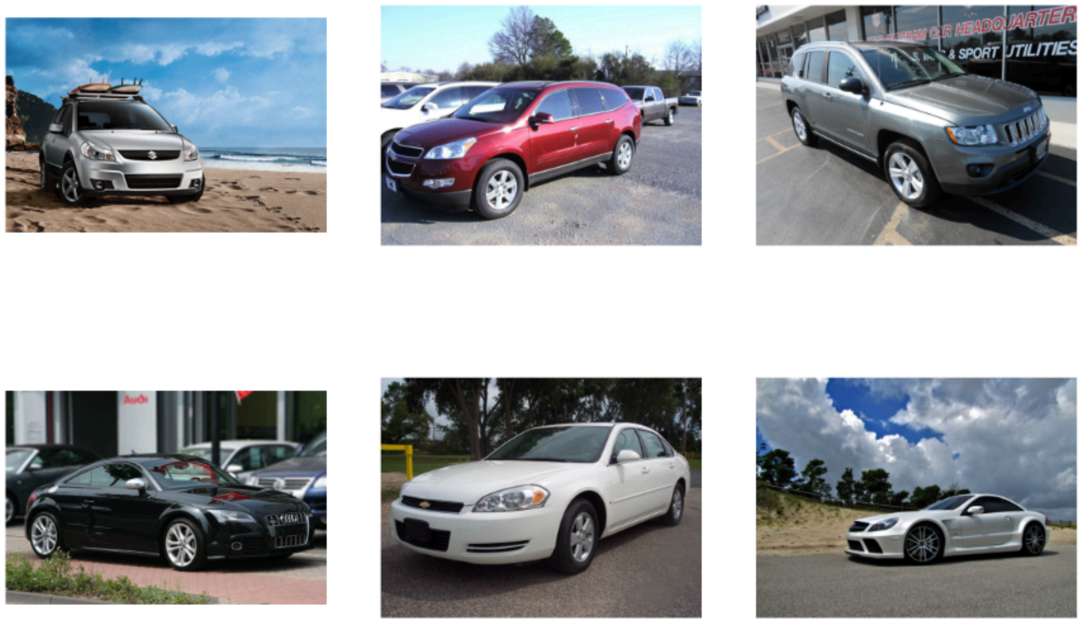
4.1 EDA and Data Preparation

The main purpose of our EDA was to display a few sample images and also to display the number of images after train, test, and validation split. To begin, we decided to display 6 random images from any of the 10 classes of cars without labels. The code below was used to display those images.



```
#Displaying 6 random images without labels

from google.colab.patches import cv2_imshow
import cv2
import os
import random
import matplotlib.pyplot as plt

# Set the directory containing the images in the "train" folder
train_image_dir = "/content/drive/MyDrive/dataset/manufacturers/train"

# Collect all image file paths in the "train" folder
train_image_files = []
for root, dirs, files in os.walk(train_image_dir):
    for file in files:
        if file.endswith(('.png', '.jpg', '.jpeg')):
            train_image_files.append(os.path.join(root, file))

# Randomly select 6 images from the "train" folder
selected_train_images = random.sample(train_image_files, 6)

# Number of rows and columns in the grid
rows, cols = 2, 3

# Create a figure and axis objects
fig, axes = plt.subplots(rows, cols, figsize=(12, 8))
fig.tight_layout()

# Display each selected image in the grid
for ax, image_path in zip(axes.flatten(), selected_train_images):
    # Load the image
    image = cv2.imread(image_path)
    # Convert BGR to RGB for displaying with matplotlib
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # Display the image
    ax.imshow(image_rgb)
    # Remove axis
    ax.axis('off')

# Show the grid
plt.show()
```

Since we did not use set.seed() function the images will be randomly loaded everytime and won't be same but in this it does not matter since the purpose of EDA is just to show a few random images to get an idea of the dataset.

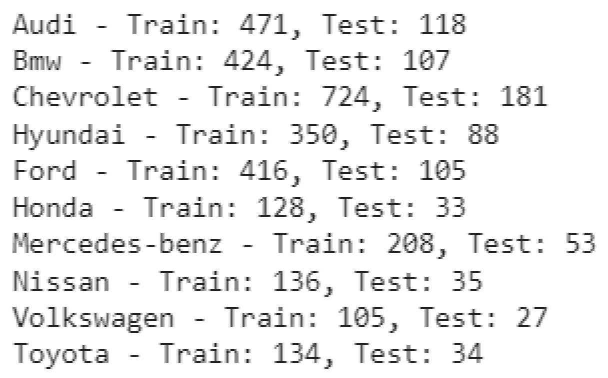


Next, we displayed another set of 6 random images but with labels. The code is the same except for an additional section which also gets the labels of the images as it is being selected.

```
# Randomly select images and their corresponding labels
selected_indices = random.sample(range(len(image_files)), num_images)
selected_images = [image_files[i] for i in selected_indices]
selected_labels = [labels[i] for i in selected_indices]
```

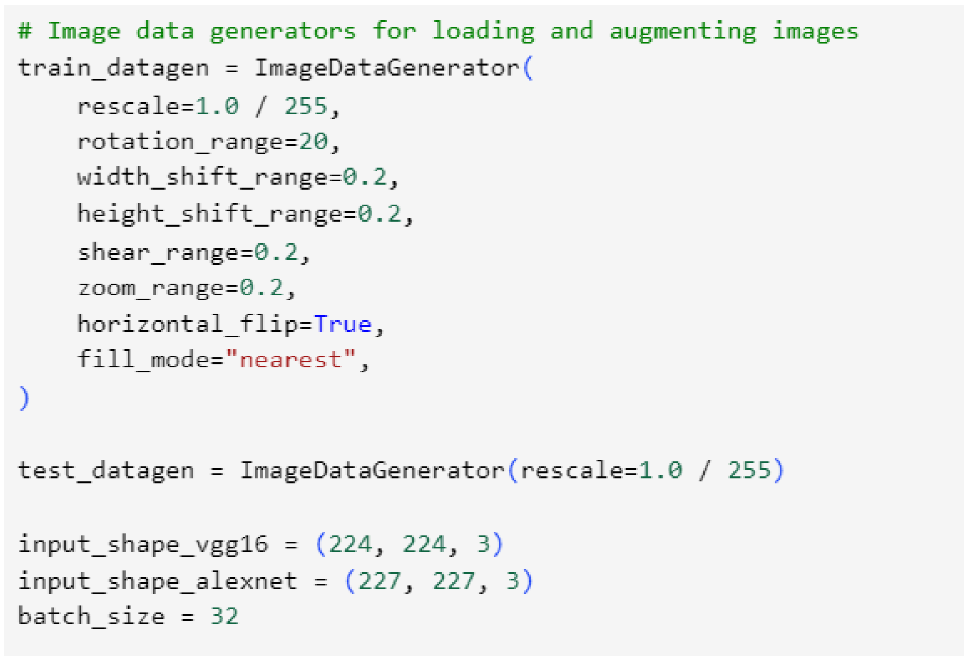
This will be the following output (the images displayed will change every time you run the code)

The following screenshot shows the total number of images in the train and test before data augmentation



```
Audi - Train: 471, Test: 118
Bmw - Train: 424, Test: 107
Chevrolet - Train: 724, Test: 181
Hyundai - Train: 350, Test: 88
Ford - Train: 416, Test: 105
Honda - Train: 128, Test: 33
Mercedes-benz - Train: 208, Test: 53
Nissan - Train: 136, Test: 35
Volkswagen - Train: 105, Test: 27
Toyota - Train: 134, Test: 34
```

For our data augmentation we carred out image rescaling, rotation, image shifting and translations, random zooms, horizontal flipping and other process like shear and fill_mode.



```
# Image data generators for loading and augmenting images
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest",
)

test_datagen = ImageDataGenerator(rescale=1.0 / 255)

input_shape_vgg16 = (224, 224, 3)
input_shape_alexnet = (227, 227, 3)
batch_size = 32
```

Furthermore, our dataset initially almost had a 50/50 split for train and test set so we utilized random shuffle function to randomly split the dataset 80/20 for train and test.

```
random.shuffle(images)

split_point = int(len(images) * 0.8)
train_images = images[:split_point]
test_images = images[split_point:]

train_manufacturer_path = os.path.join(new_path, 'train', manufacturer)
test_manufacturer_path = os.path.join(new_path, 'test', manufacturer)
```

This was the following results from the random shuffle

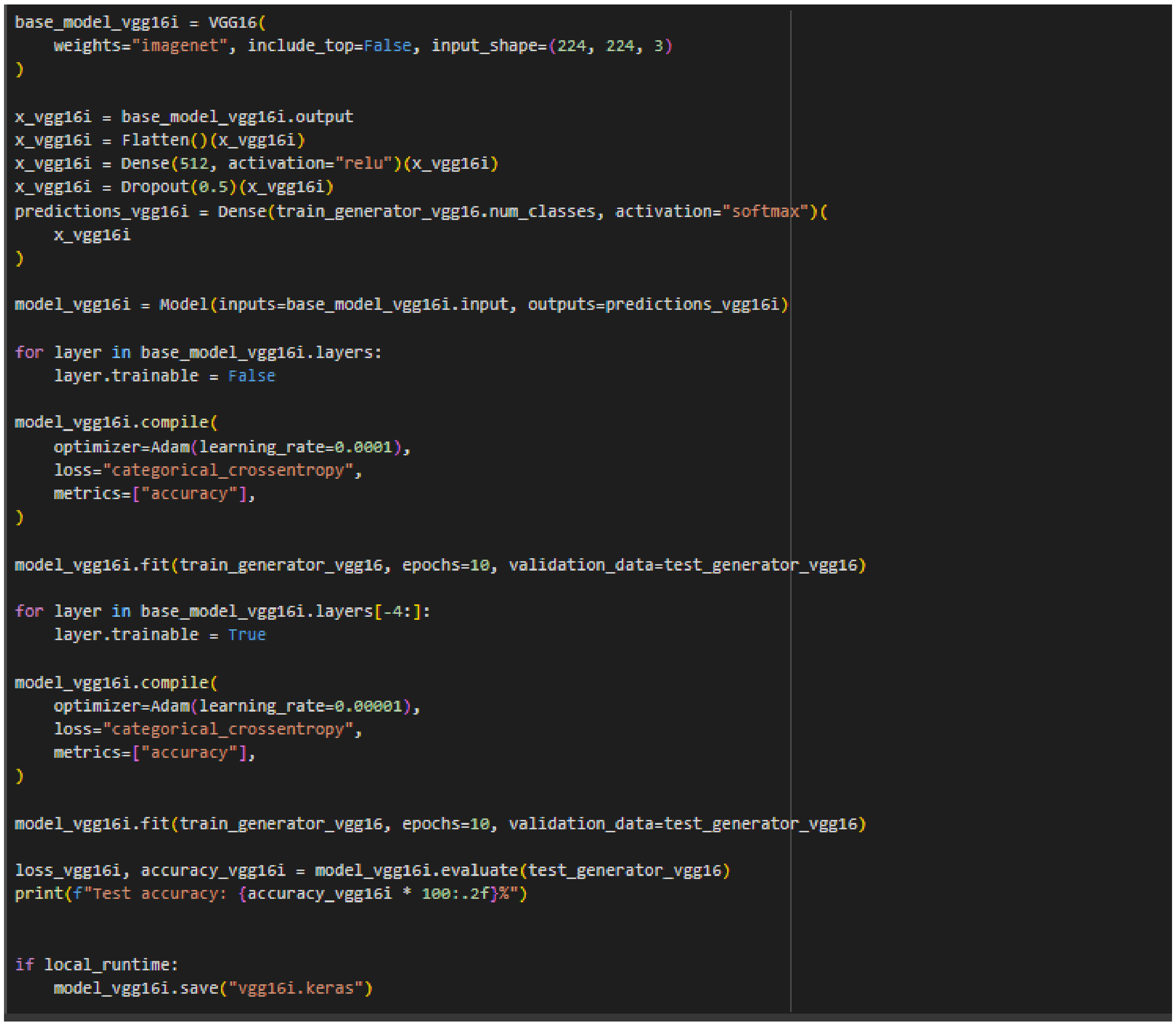
```
Found 6059 images belonging to 10 classes.
Found 1616 images belonging to 10 classes.
```

6059 is for the train set and 1616 is for the test set.

4.2 Modeling

4.2.1.1 Model 1

Experiment 1: VGG16 with pretrained weights



```
base_model_vgg16i = VGG16(
    weights="imagenet", include_top=False, input_shape=(224, 224, 3)
)

x_vgg16i = base_model_vgg16i.output
x_vgg16i = Flatten()(x_vgg16i)
x_vgg16i = Dense(512, activation="relu")(x_vgg16i)
x_vgg16i = Dropout(0.5)(x_vgg16i)
predictions_vgg16i = Dense(train_generator_vgg16.num_classes, activation="softmax")(
    x_vgg16i
)

model_vgg16i = Model(inputs=base_model_vgg16i.input, outputs=predictions_vgg16i)

for layer in base_model_vgg16i.layers:
    layer.trainable = False

model_vgg16i.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)

model_vgg16i.fit(train_generator_vgg16, epochs=10, validation_data=test_generator_vgg16)

for layer in base_model_vgg16i.layers[-4:]:
    layer.trainable = True

model_vgg16i.compile(
    optimizer=Adam(learning_rate=0.00001),
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)

model_vgg16i.fit(train_generator_vgg16, epochs=10, validation_data=test_generator_vgg16)

loss_vgg16i, accuracy_vgg16i = model_vgg16i.evaluate(test_generator_vgg16)
print(f"Test accuracy: {accuracy_vgg16i * 100:.2f}%")


if local_runtime:
    model_vgg16i.save("vgg16i.keras")
```

VGG16(weights="imagenet"): Loads the VGG16 model pre-trained on the ImageNet dataset. include_top=False: Excludes the top (fully connected) layers of the network, as we will add custom top layers for our specific classification task. input_shape=input_shape: Sets the input shape to match the images in the dataset (224x224 with 3 color channels). ‘model.add(base_model): Adds the pre-trained VGG16 model as the base of the new model.

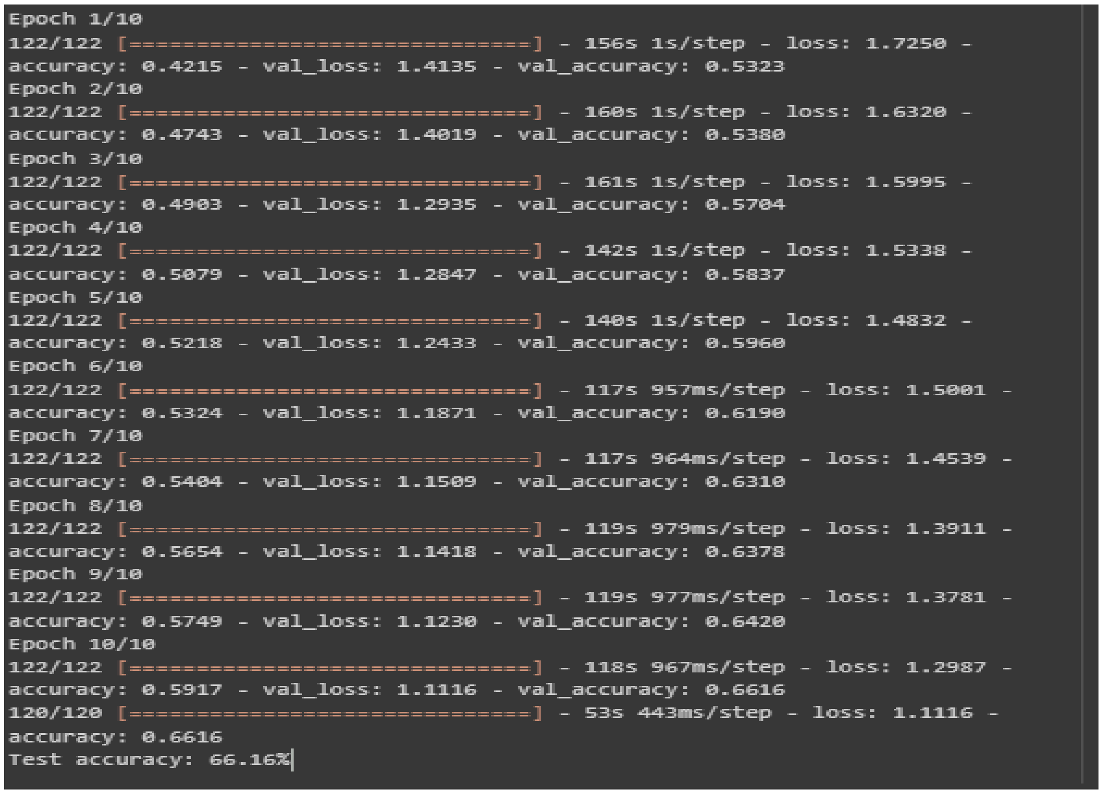
Flatten(): Flattens the output of the convolutional layers to a 1D array. Dense(4096, activation="relu"): Adds a fully connected layer with 4096 units and ReLU activation. Dropout(0.5): Adds dropout regularization to prevent overfitting (drops 50% of the units randomly). Dense(num_classes, activation="softmax"): Adds the output layer with a number of units equal to the number of classes and softmax activation for multi-class classification.

Most Importantly these lines were used to train the model:

- train_generator_vgg16: Provides the training data.
- steps_per_epoch=steps_per_epoch_vgg16i: Specifies the number of steps per epoch. epochs=10: Trains for 10 epochs.
- validation_data=test_generator_vgg16: Provides the validation data for evaluating the model.

4.2.1.1 Summary of Model 1

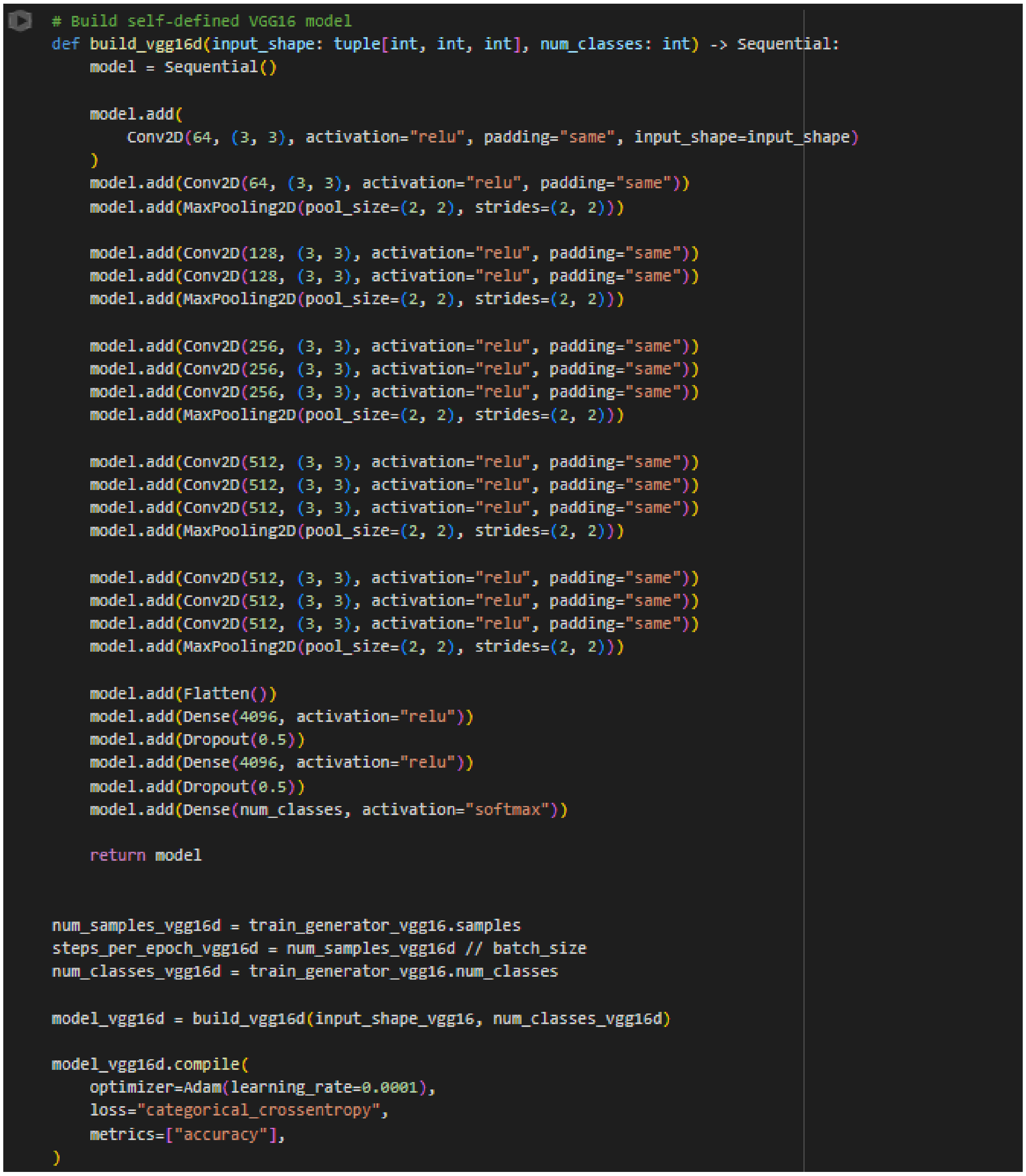
A VGG16 model pre-trained on ImageNet, excluding its top classification layers, is used as a base. New fully connected layers, including flatten, dense, dropout, and a final softmax layer for classification, are added. The model is compiled using the Adam optimizer and categorical cross-entropy loss before being trained on a training data generator, with performance evaluated against a test data generator. The results of these models are shown below:



```
Epoch 1/10
122/122 [==============================] - 156s 1s/step - loss: 1.7250 -
accuracy: 0.4215 - val_loss: 1.4135 - val_accuracy: 0.5323
Epoch 2/10
122/122 [==============================] - 160s 1s/step - loss: 1.6320 -
accuracy: 0.4743 - val_loss: 1.4019 - val_accuracy: 0.5380
Epoch 3/10
122/122 [==============================] - 161s 1s/step - loss: 1.5995 -
accuracy: 0.4903 - val_loss: 1.2935 - val_accuracy: 0.5704
Epoch 4/10
122/122 [==============================] - 142s 1s/step - loss: 1.5338 -
accuracy: 0.5079 - val_loss: 1.2847 - val_accuracy: 0.5837
Epoch 5/10
122/122 [==============================] - 140s 1s/step - loss: 1.4832 -
accuracy: 0.5218 - val_loss: 1.2433 - val_accuracy: 0.5960
Epoch 6/10
122/122 [==============================] - 117s 957ms/step - loss: 1.5001 -
accuracy: 0.5324 - val_loss: 1.1871 - val_accuracy: 0.6190
Epoch 7/10
122/122 [==============================] - 117s 964ms/step - loss: 1.4539 -
accuracy: 0.5404 - val_loss: 1.1509 - val_accuracy: 0.6310
Epoch 8/10
122/122 [==============================] - 119s 979ms/step - loss: 1.3911 -
accuracy: 0.5654 - val_loss: 1.1418 - val_accuracy: 0.6378
Epoch 9/10
122/122 [==============================] - 119s 977ms/step - loss: 1.3781 -
accuracy: 0.5749 - val_loss: 1.1230 - val_accuracy: 0.6420
Epoch 10/10
122/122 [==============================] - 118s 967ms/step - loss: 1.2987 -
accuracy: 0.5917 - val_loss: 1.1116 - val_accuracy: 0.6616
120/120 [==============================] - 53s 443ms/step - loss: 1.1116 -
accuracy: 0.6616
Test accuracy: 66.16%
```

4.2.1.2 Model 1

Experiment 2: VGG16 without pretrained weights



```python
# Build self-defined VGG16 model
def build_vgg16d(input_shape: tuple[int, int, int], num_classes: int) -> Sequential:
    model = Sequential()

    model.add(
        Conv2D(64, (3, 3), activation="relu", padding="same", input_shape=input_shape)
    )
    model.add(Conv2D(64, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Conv2D(128, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(128, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Conv2D(256, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(256, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(256, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(Conv2D(512, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Flatten())
    model.add(Dense(4096, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation="softmax"))

    return model


num_samples_vgg16d = train_generator_vgg16.samples
steps_per_epoch_vgg16d = num_samples_vgg16d // batch_size
num_classes_vgg16d = train_generator_vgg16.num_classes

model_vgg16d = build_vgg16d(input_shape_vgg16, num_classes_vgg16d)

model_vgg16d.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
```

We have to define and build our own vgg16 model from scratch to use as a comparator with pretrained weights:

- Conv2D(64, (3, 3), padding="same", activation="relu", input_shape=input_shape): Adds a 2D convolutional layer with 64 filters, 3x3 kernel size, same padding, and ReLU activation. The input_shape specifies the input dimensions.
- MaxPooling2D(pool_size=(2, 2)): Adds a max pooling layer with 2x2 pool size to reduce the spatial dimensions of the output volume.

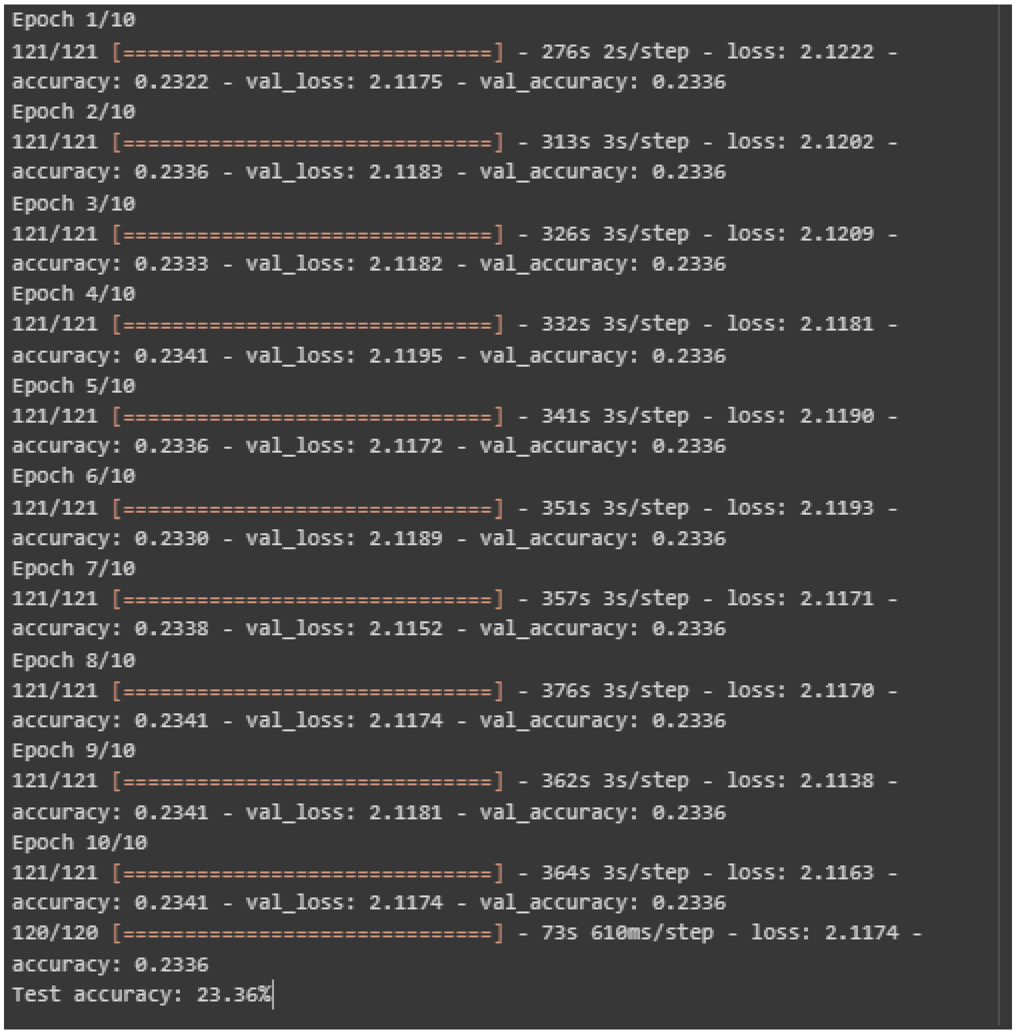
We add two different types of layers for different purposes:

- Convolutional Layers: Add layers with increasing filter sizes (128, 256) to capture more complex features.
- Pooling Layers: Reduce the spatial dimensions after each block of convolutional layers.

The Training and compiling of this model is practically the same as the vgg16 model with training weights so we will not be showing it here to save time.

4.2.1.2 Summary of Model 1, Experiment 2

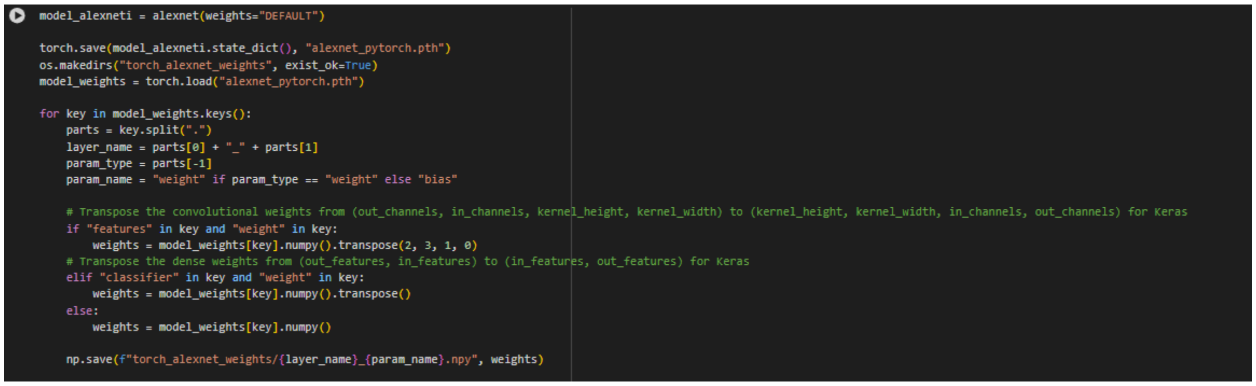
A custom convolutional neural network architecture inspired by VGG16 is constructed using convolutional, max pooling, flatten, dense, and dropout layers. This model is compiled using the Adam optimizer and categorical cross-entropy loss before being trained on a training data generator and evaluated against a test data generator. The results are shown below:



```
Epoch 1/10
121/121 [==============================] - 276s 2s/step - loss: 2.1222 -
accuracy: 0.2322 - val_loss: 2.1175 - val_accuracy: 0.2336
Epoch 2/10
121/121 [==============================] - 313s 3s/step - loss: 2.1202 -
accuracy: 0.2336 - val_loss: 2.1183 - val_accuracy: 0.2336
Epoch 3/10
121/121 [==============================] - 326s 3s/step - loss: 2.1209 -
accuracy: 0.2333 - val_loss: 2.1182 - val_accuracy: 0.2336
Epoch 4/10
121/121 [==============================] - 332s 3s/step - loss: 2.1181 -
accuracy: 0.2341 - val_loss: 2.1195 - val_accuracy: 0.2336
Epoch 5/10
121/121 [==============================] - 341s 3s/step - loss: 2.1190 -
accuracy: 0.2336 - val_loss: 2.1172 - val_accuracy: 0.2336
Epoch 6/10
121/121 [==============================] - 351s 3s/step - loss: 2.1193 -
accuracy: 0.2330 - val_loss: 2.1189 - val_accuracy: 0.2336
Epoch 7/10
121/121 [==============================] - 357s 3s/step - loss: 2.1171 -
accuracy: 0.2338 - val_loss: 2.1152 - val_accuracy: 0.2336
Epoch 8/10
121/121 [==============================] - 376s 3s/step - loss: 2.1170 -
accuracy: 0.2341 - val_loss: 2.1174 - val_accuracy: 0.2336
Epoch 9/10
121/121 [==============================] - 362s 3s/step - loss: 2.1138 -
accuracy: 0.2341 - val_loss: 2.1181 - val_accuracy: 0.2336
Epoch 10/10
121/121 [==============================] - 364s 3s/step - loss: 2.1163 -
accuracy: 0.2341 - val_loss: 2.1174 - val_accuracy: 0.2336
120/120 [==============================] - 73s 610ms/step - loss: 2.1174 -
accuracy: 0.2336
Test accuracy: 23.36%
```

4.2.2.1 Model 2

Experiment 1: AlexNet with Pretrained weights

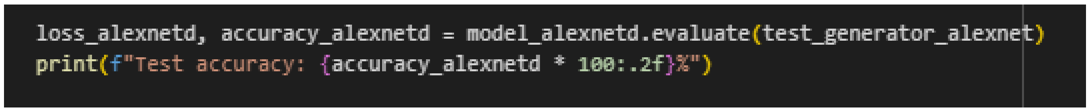


```
model_alexneti = alexnet(weights="DEFAULT")

torch.save(model_alexneti.state_dict(), "alexnet_pytorch.pth")
os.makedirs("torch_alexnet_weights", exist_ok=True)
model_weights = torch.load("alexnet_pytorch.pth")

for key in model_weights.keys():
    parts = key.split(".")
    layer_name = parts[0] + "_" + parts[1]
    param_type = parts[-1]
    param_name = "weight" if param_type == "weight" else "bias"

    # Transpose the convolutional weights from (out_channels, in_channels, kernel_height, kernel_width) to (kernel_height, kernel_width, in_channels, out_channels) for Keras
    if "features" in key and "weight" in key:
        weights = model_weights[key].numpy().transpose(2, 3, 1, 0)
    # Transpose the dense weights from (out_features, in_features) to (in_features, out_features) for Keras
    elif "classifier" in key and "weight" in key:
        weights = model_weights[key].numpy().transpose()
    else:
        weights = model_weights[key].numpy()

    np.save(f"torch_alexnet_weights/{layer_name}_{param_name}.npy", weights)
```

The AlexNet model is imported from the library torch, and the code ‘torchvision.models’ provides us access to various pretrained models, in this case AlexNet. alexnet(pretrained=default): This command loads the AlexNet model pre-trained on the ImageNet dataset. The pretrained=default argument indicates that we want to use the pre-trained weights. model_alexneti.classifier[6]: Accesses the last fully connected layer of AlexNet’s classifier. In PyTorch, the classifier part of AlexNet is defined as a sequence of layers, and the sixth layer is the final fully connected layer. We also use different lines of code to transpose the convolutional and dense weights for Keras.

This modification replaces the original output layer of AlexNet, which was designed for 1000 classes (as per the ImageNet dataset), with a new layer tailored to the number of classes in our specific problem.

Similar to VGG16 it uses Maxpooling2d and conv2d layers for convolutional and pooling layers. Once again training and compiling of this model undergoes a very similar process to the two above.



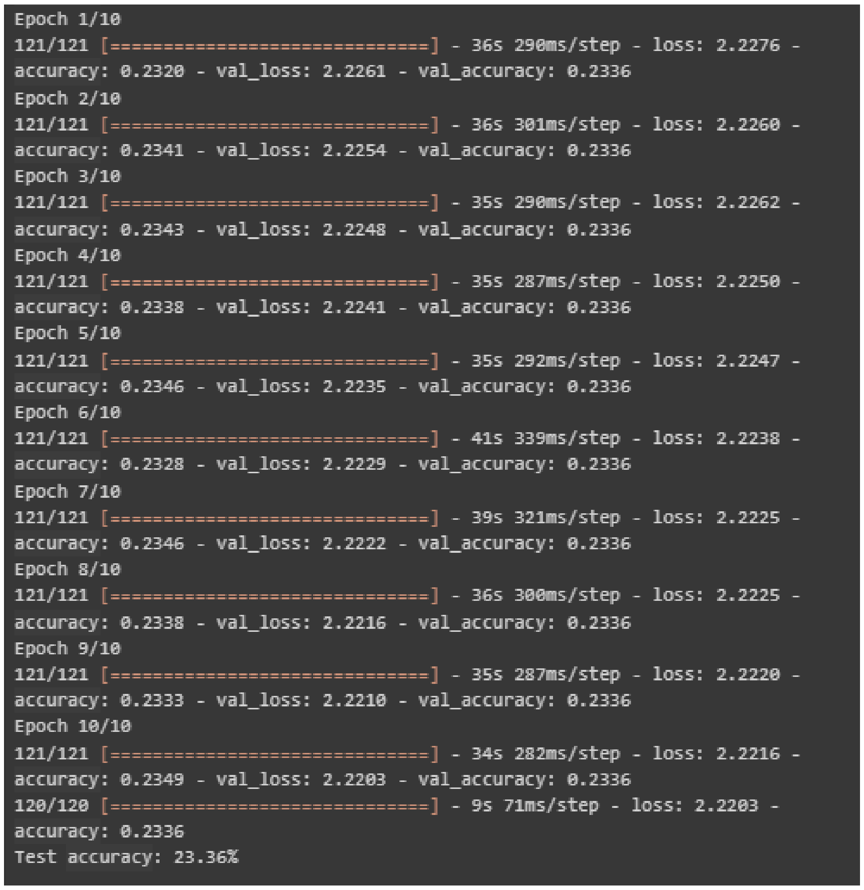
```
loss_alexnetd, accuracy_alexnetd = model_alexnetd.evaluate(test_generator_alexnet)
print(f"Test accuracy: {accuracy_alexnetd * 100:.2f}%")
```

The method .evaluate() runs the forward pass of the model on the test data, computes the loss, and evaluates the accuracy.

- loss_alexnetd: This variable captures the loss of the model on the test dataset.
- accuracy_alexnetd: This variable captures the accuracy of the model on the test dataset.

4.2.2.1 Summary of Model 2, Experiment 1

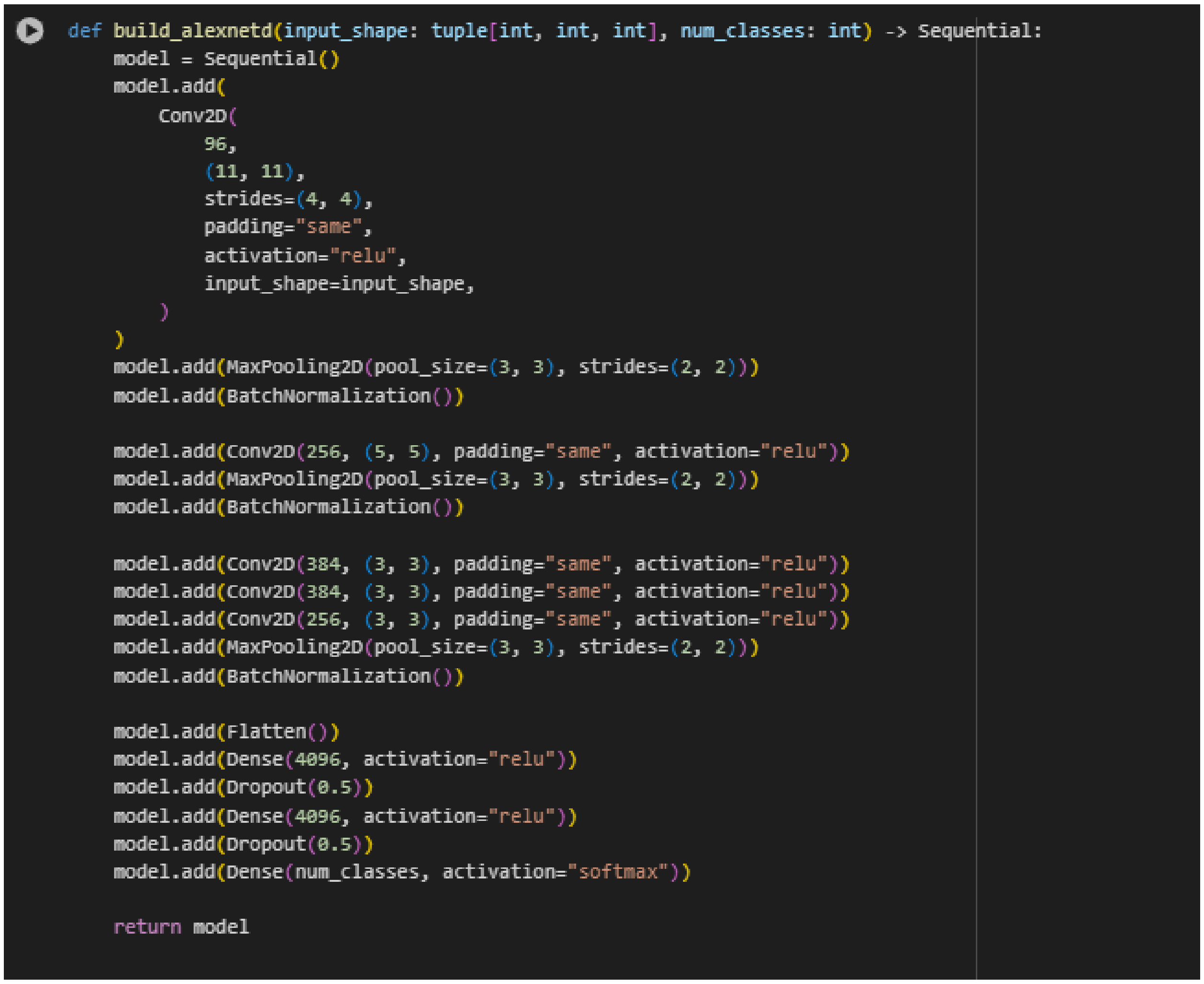
In short we replace the last fully connected layer with a new one matching the number of classes. The results after a 10 step epoch process are shown below.



```
Epoch 1/10
121/121 [==============================] - 36s 290ms/step - loss: 2.2276 -
accuracy: 0.2320 - val_loss: 2.2261 - val_accuracy: 0.2336
Epoch 2/10
121/121 [==============================] - 36s 301ms/step - loss: 2.2260 -
accuracy: 0.2341 - val_loss: 2.2254 - val_accuracy: 0.2336
Epoch 3/10
121/121 [==============================] - 35s 290ms/step - loss: 2.2262 -
accuracy: 0.2343 - val_loss: 2.2248 - val_accuracy: 0.2336
Epoch 4/10
121/121 [==============================] - 35s 287ms/step - loss: 2.2250 -
accuracy: 0.2338 - val_loss: 2.2241 - val_accuracy: 0.2336
Epoch 5/10
121/121 [==============================] - 35s 292ms/step - loss: 2.2247 -
accuracy: 0.2346 - val_loss: 2.2235 - val_accuracy: 0.2336
Epoch 6/10
121/121 [==============================] - 41s 339ms/step - loss: 2.2238 -
accuracy: 0.2328 - val_loss: 2.2229 - val_accuracy: 0.2336
Epoch 7/10
121/121 [==============================] - 39s 321ms/step - loss: 2.2225 -
accuracy: 0.2346 - val_loss: 2.2222 - val_accuracy: 0.2336
Epoch 8/10
121/121 [==============================] - 36s 300ms/step - loss: 2.2225 -
accuracy: 0.2338 - val_loss: 2.2216 - val_accuracy: 0.2336
Epoch 9/10
121/121 [==============================] - 35s 287ms/step - loss: 2.2220 -
accuracy: 0.2333 - val_loss: 2.2210 - val_accuracy: 0.2336
Epoch 10/10
121/121 [==============================] - 34s 282ms/step - loss: 2.2216 -
accuracy: 0.2349 - val_loss: 2.2203 - val_accuracy: 0.2336
120/120 [==============================] - 9s 71ms/step - loss: 2.2203 -
accuracy: 0.2336
Test accuracy: 23.36%
```

4.2.2.2 Model 2

Experiment 2: AlexNet without Pretrained weights



```
def build_alexnetd(input_shape: tuple[int, int, int], num_classes: int) -> Sequential:
    model = Sequential()
    model.add(
        Conv2D(
            96,
            (11, 11),
            strides=(4, 4),
            padding="same",
            activation="relu",
            input_shape=input_shape,
        )
    )
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(BatchNormalization())

    model.add(Conv2D(256, (5, 5), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(BatchNormalization())

    model.add(Conv2D(384, (3, 3), padding="same", activation="relu"))
    model.add(Conv2D(384, (3, 3), padding="same", activation="relu"))
    model.add(Conv2D(256, (3, 3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(BatchNormalization())

    model.add(Flatten())
    model.add(Dense(4096, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation="softmax"))

    return model
```

Different types of layers were used and their respective lines are shown below:

- Conv2D(96, (11, 11), strides=(4, 4), padding="same", activation="relu", input_shape=input_shape): Adds a 2D convolutional layer with 96 filters, 11x11 kernel size, stride of 4, same padding, and ReLU activation.
- MaxPooling2D(pool_size=(3, 3), strides=(2, 2)): Adds a max pooling layer with 3x3 pool size and stride of 2.

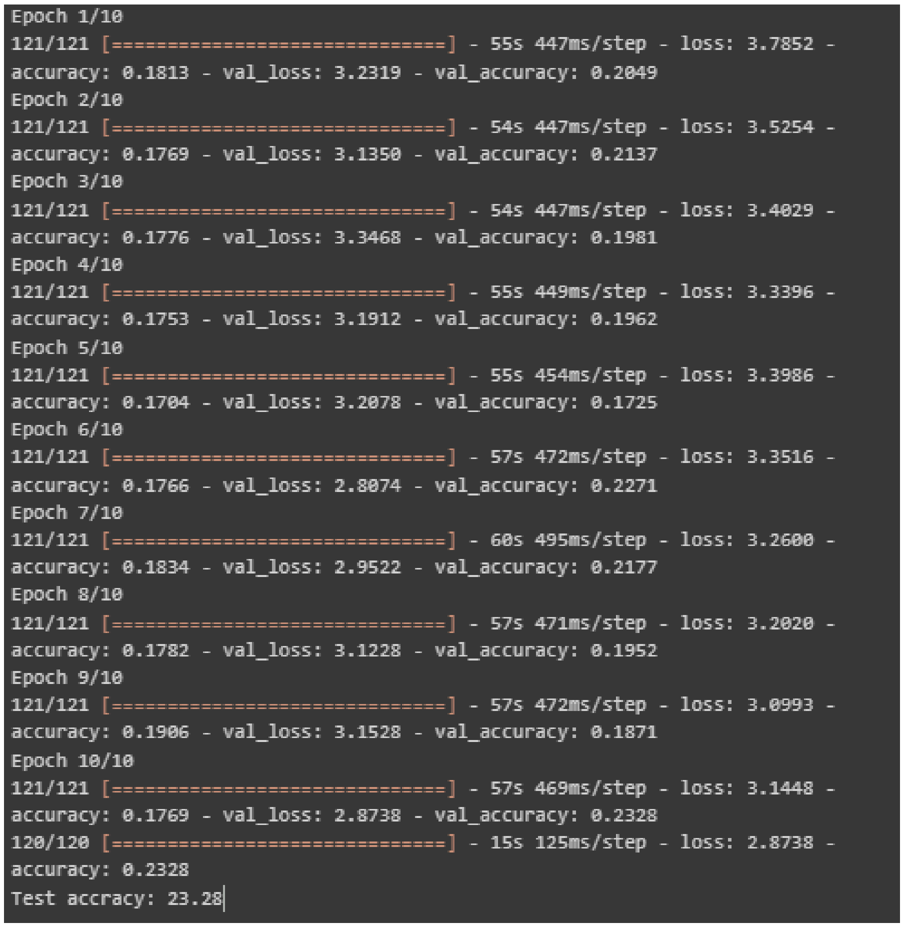
- BatchNormalization(): Adds batch normalization to normalize the output of the convolutional layer.

These layers are stacked on top of one another with increasing filter sizes from 256 to 384 on top of additional convolutional layers in order to help the model capture more complex features in our car database.

The final bit of code starting from ‘model.add(Flatten()) is used to add fully connected layers and dropout regularization in order to prevent overfitting by ensuring no images in the dataset are codependent of each other.

4.2.2.2 Summary of Model 2, Experiment 2

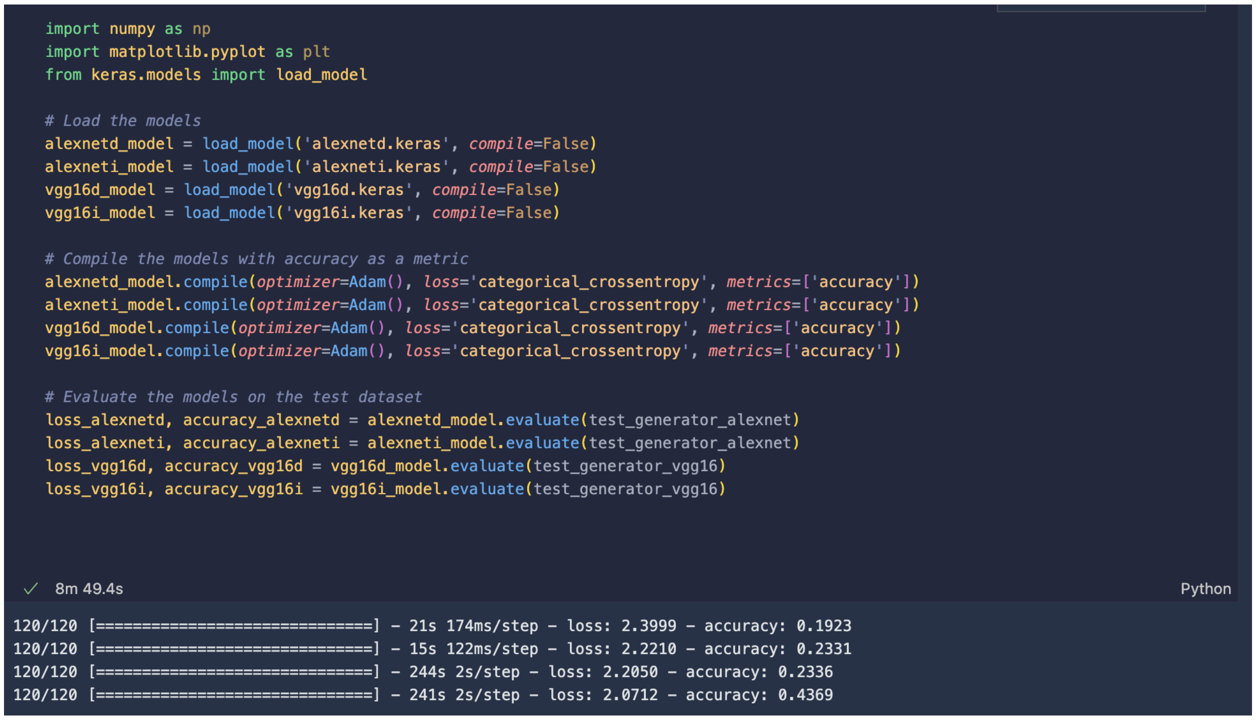
We built a custom AlexNet like architecture with Conv2D, MaxPooling2D, Flatten, Dense, Dropout and BatchNormalization layers. The results are shown below:



```
Epoch 1/10
121/121 [==============================] - 55s 447ms/step - loss: 3.7852 -
accuracy: 0.1813 - val_loss: 3.2319 - val_accuracy: 0.2049
Epoch 2/10
121/121 [==============================] - 54s 447ms/step - loss: 3.5254 -
accuracy: 0.1769 - val_loss: 3.1350 - val_accuracy: 0.2137
Epoch 3/10
121/121 [==============================] - 54s 447ms/step - loss: 3.4029 -
accuracy: 0.1776 - val_loss: 3.3468 - val_accuracy: 0.1981
Epoch 4/10
121/121 [==============================] - 55s 449ms/step - loss: 3.3396 -
accuracy: 0.1753 - val_loss: 3.1912 - val_accuracy: 0.1962
Epoch 5/10
121/121 [==============================] - 55s 454ms/step - loss: 3.3986 -
accuracy: 0.1704 - val_loss: 3.2078 - val_accuracy: 0.1725
Epoch 6/10
121/121 [==============================] - 57s 472ms/step - loss: 3.3516 -
accuracy: 0.1766 - val_loss: 2.8074 - val_accuracy: 0.2271
Epoch 7/10
121/121 [==============================] - 60s 495ms/step - loss: 3.2600 -
accuracy: 0.1834 - val_loss: 2.9522 - val_accuracy: 0.2177
Epoch 8/10
121/121 [==============================] - 57s 471ms/step - loss: 3.2020 -
accuracy: 0.1782 - val_loss: 3.1228 - val_accuracy: 0.1952
Epoch 9/10
121/121 [==============================] - 57s 472ms/step - loss: 3.0993 -
accuracy: 0.1906 - val_loss: 3.1528 - val_accuracy: 0.1871
Epoch 10/10
121/121 [==============================] - 57s 469ms/step - loss: 3.1448 -
accuracy: 0.1769 - val_loss: 2.8738 - val_accuracy: 0.2328
120/120 [==============================] - 15s 125ms/step - loss: 2.8738 -
accuracy: 0.2328
Test accracy: 23.28
```

4.3 Summary of Implementation and Results

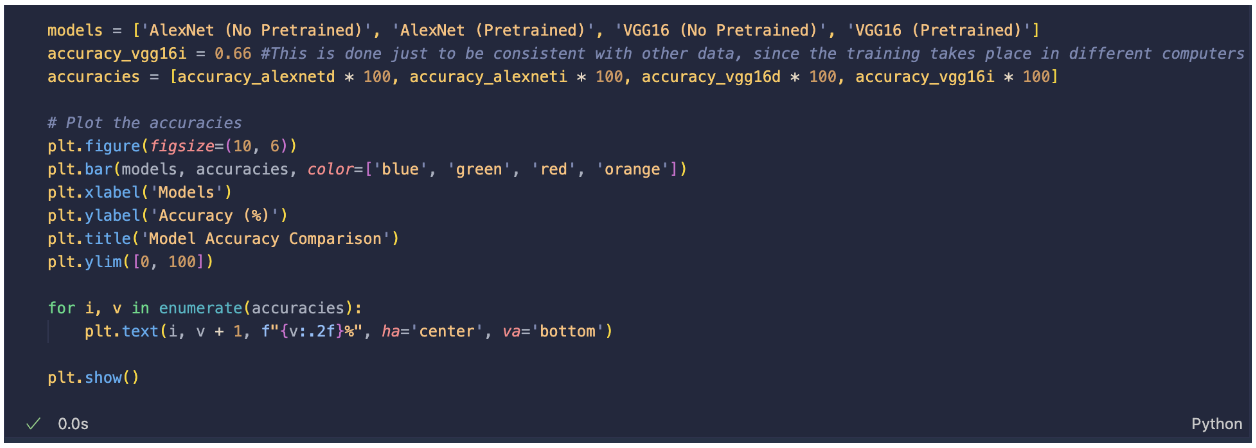
4.3.1 Utilize Models For Accuracy Comparisons



```python
import numpy as np
import matplotlib.pyplot as plt
from keras.models import load_model

# Load the models
alexnetd_model = load_model('alexnetd.keras', compile=False)
alexneti_model = load_model('alexneti.keras', compile=False)
vgg16d_model = load_model('vgg16d.keras', compile=False)
vgg16i_model = load_model('vgg16i.keras', compile=False)

# Compile the models with accuracy as a metric
alexnetd_model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
alexneti_model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
vgg16d_model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
vgg16i_model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# Evaluate the models on the test dataset
loss_alexnetd, accuracy_alexnetd = alexnetd_model.evaluate(test_generator_alexnet)
loss_alexneti, accuracy_alexneti = alexneti_model.evaluate(test_generator_alexnet)
loss_vgg16d, accuracy_vgg16d = vgg16d_model.evaluate(test_generator_vgg16)
loss_vgg16i, accuracy_vgg16i = vgg16i_model.evaluate(test_generator_vgg16)
```

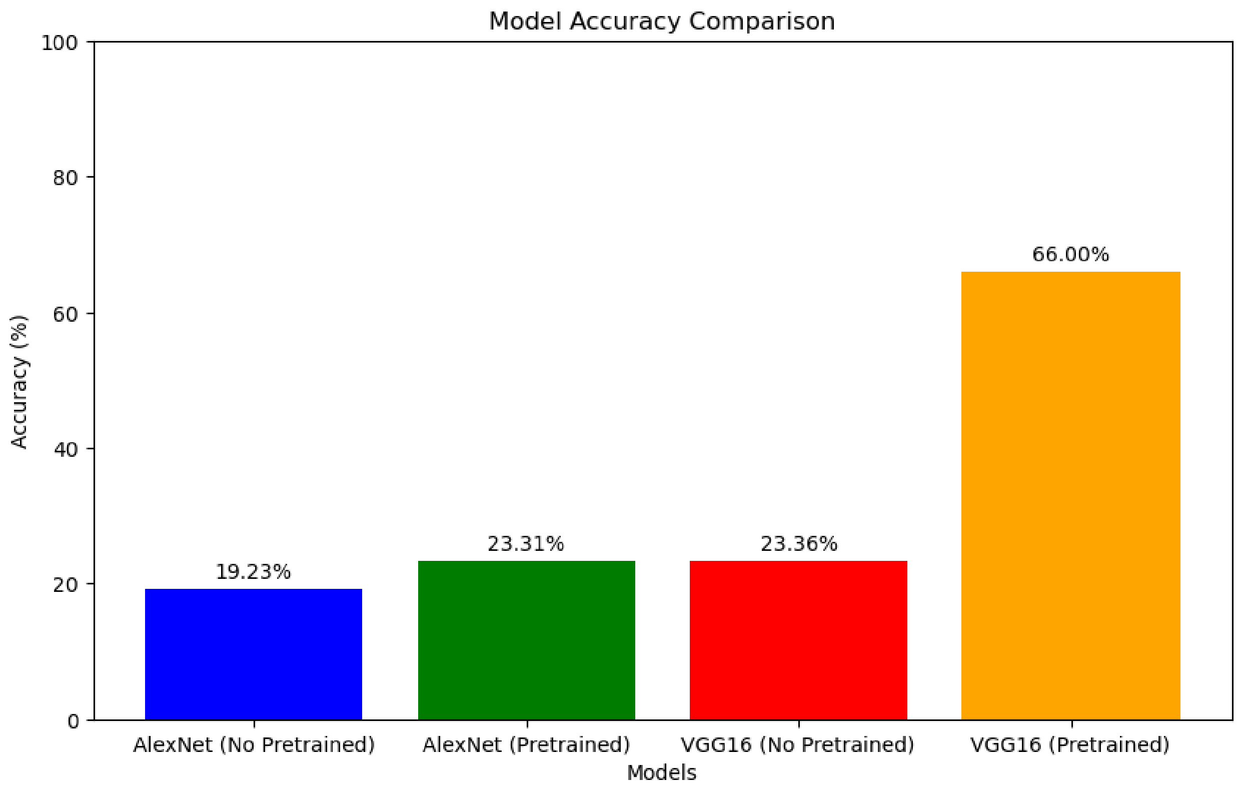
✓ 8m 49.4s Python

```
120/120 [==============================] - 21s 174ms/step - loss: 2.3999 - accuracy: 0.1923
120/120 [==============================] - 15s 122ms/step - loss: 2.2210 - accuracy: 0.2331
120/120 [==============================] - 244s 2s/step - loss: 2.2050 - accuracy: 0.2336
120/120 [==============================] - 241s 2s/step - loss: 2.0712 - accuracy: 0.4369
```

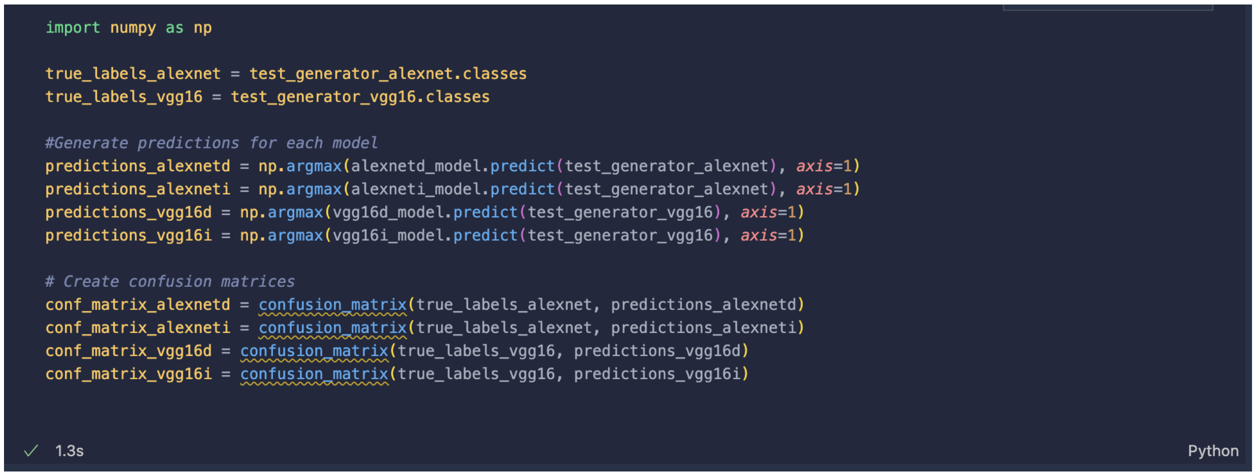


```python
models = ['AlexNet (No Pretrained)', 'AlexNet (Pretrained)', 'VGG16 (No Pretrained)', 'VGG16 (Pretrained)']
accuracy_vgg16i = 0.66 #This is done just to be consistent with other data, since the training takes place in different computers
accuracies = [accuracy_alexnetd * 100, accuracy_alexneti * 100, accuracy_vgg16d * 100, accuracy_vgg16i * 100]

# Plot the accuracies
plt.figure(figsize=(10, 6))
plt.bar(models, accuracies, color=['blue', 'green', 'red', 'orange'])
plt.xlabel('Models')
plt.ylabel('Accuracy (%)')
plt.title('Model Accuracy Comparison')
plt.ylim([0, 100])

for i, v in enumerate(accuracies):
    plt.text(i, v + 1, f"{v:.2f}%", ha='center', va='bottom')

plt.show()
```

✓ 0.0s Python

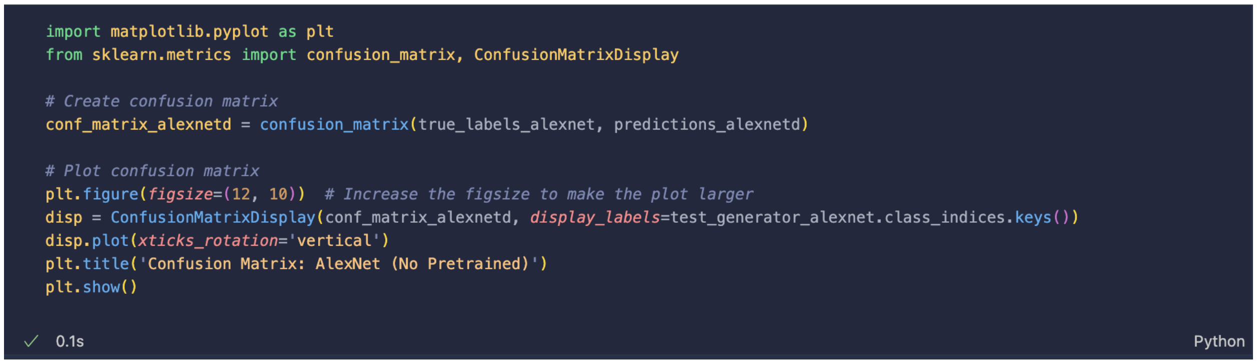


4.3.2 Utilize Models For Confusion Matrix Predictions



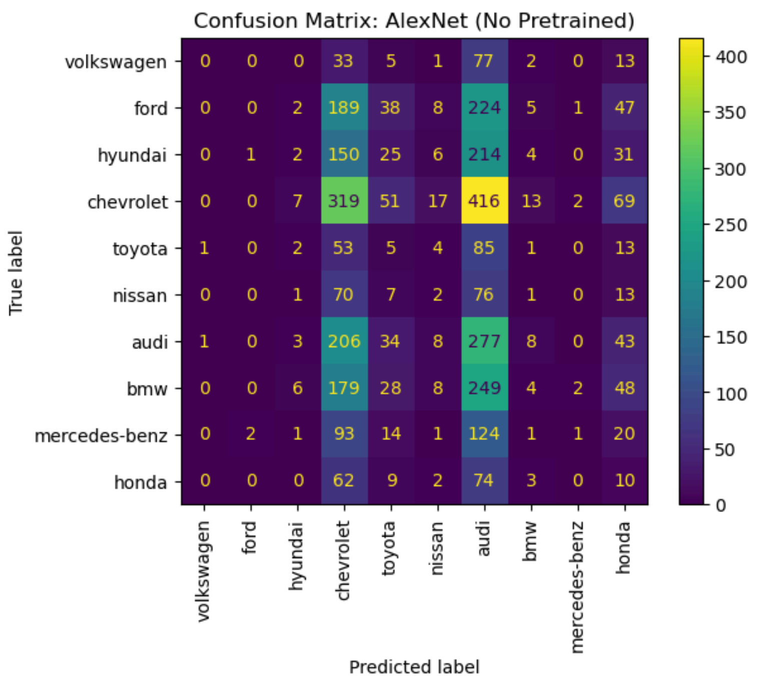
```python
import numpy as np

true_labels_alexnet = test_generator_alexnet.classes
true_labels_vgg16 = test_generator_vgg16.classes

#Generate predictions for each model
predictions_alexnetd = np.argmax(alexnetd_model.predict(test_generator_alexnet), axis=1)
predictions_alexneti = np.argmax(alexneti_model.predict(test_generator_alexnet), axis=1)
predictions_vgg16d = np.argmax(vgg16d_model.predict(test_generator_vgg16), axis=1)
predictions_vgg16i = np.argmax(vgg16i_model.predict(test_generator_vgg16), axis=1)

# Create confusion matrices
conf_matrix_alexnetd = confusion_matrix(true_labels_alexnet, predictions_alexnetd)
conf_matrix_alexneti = confusion_matrix(true_labels_alexnet, predictions_alexneti)
conf_matrix_vgg16d = confusion_matrix(true_labels_vgg16, predictions_vgg16d)
conf_matrix_vgg16i = confusion_matrix(true_labels_vgg16, predictions_vgg16i)
```

4.3.2.1 Plotting Confusion Matrix For Alexnet Without Pretrained Weight



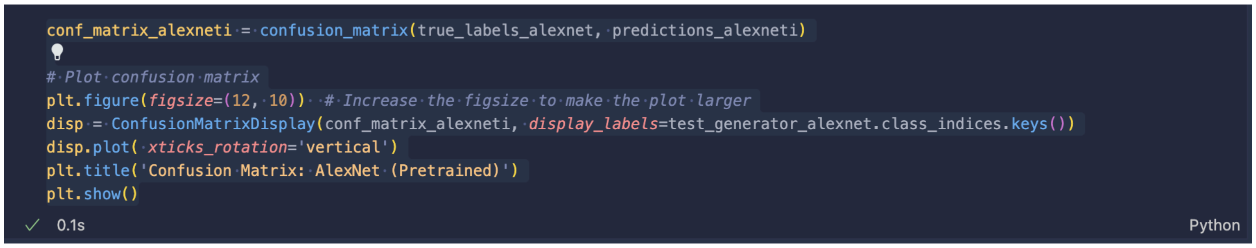
```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Create confusion matrix
conf_matrix_alexnetd = confusion_matrix(true_labels_alexnet, predictions_alexnetd)

# Plot confusion matrix
plt.figure(figsize=(12, 10))  # Increase the figsize to make the plot larger
disp = ConfusionMatrixDisplay(conf_matrix_alexnetd, display_labels=test_generator_alexnet.class_indices.keys())
disp.plot(xticks_rotation='vertical')
plt.title('Confusion Matrix: AlexNet (No Pretrained)')
plt.show()
```

```
✓ 0.1s                                                                    Python
```

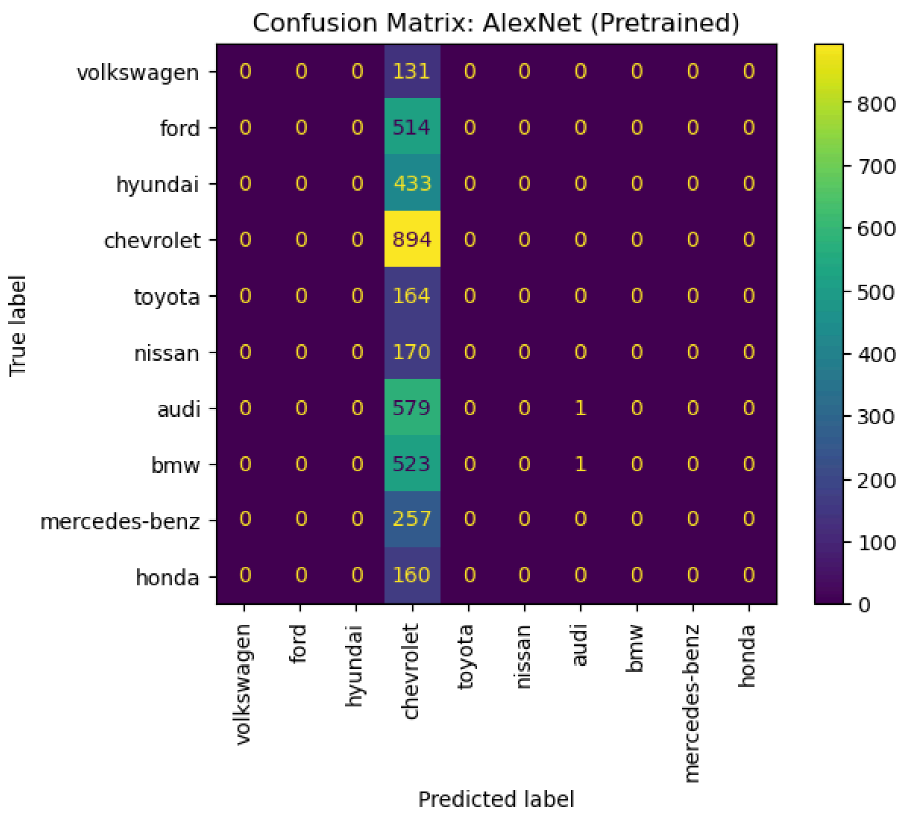


	volkswagen	ford	hyundai	chevrolet	toyota	nissan	audi	bmw	mercedes-benz	honda
volkswagen	0	0	0	33	5	1	77	2	0	13
ford	0	0	2	189	38	8	224	5	1	47
hyundai	0	1	2	150	25	6	214	4	0	31
chevrolet	0	0	7	319	51	17	416	13	2	69
toyota	1	0	2	53	5	4	85	1	0	13
nissan	0	0	1	70	7	2	76	1	0	13
audi	1	0	3	206	34	8	277	8	0	43
bmw	0	0	6	179	28	8	249	4	2	48
mercedes-benz	0	2	1	93	14	1	124	1	1	20
honda	0	0	0	62	9	2	74	3	0	10

4.3.2.2 Plotting Confusion Matrix For Alexnet With Pretrained Weight

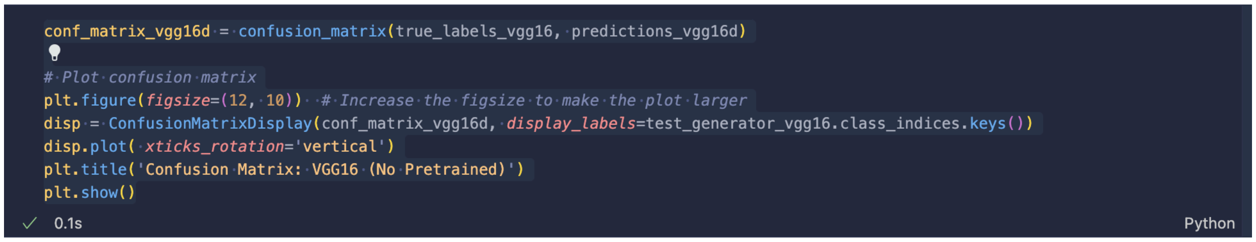


```python
conf_matrix_alexneti = confusion_matrix(true_labels_alexnet, predictions_alexneti)

# Plot confusion matrix
plt.figure(figsize=(12, 10))  # Increase the figsize to make the plot larger
disp = ConfusionMatrixDisplay(conf_matrix_alexneti, display_labels=test_generator_alexnet.class_indices.keys())
disp.plot( xticks_rotation='vertical')
plt.title('Confusion Matrix: AlexNet (Pretrained)')
plt.show()
```

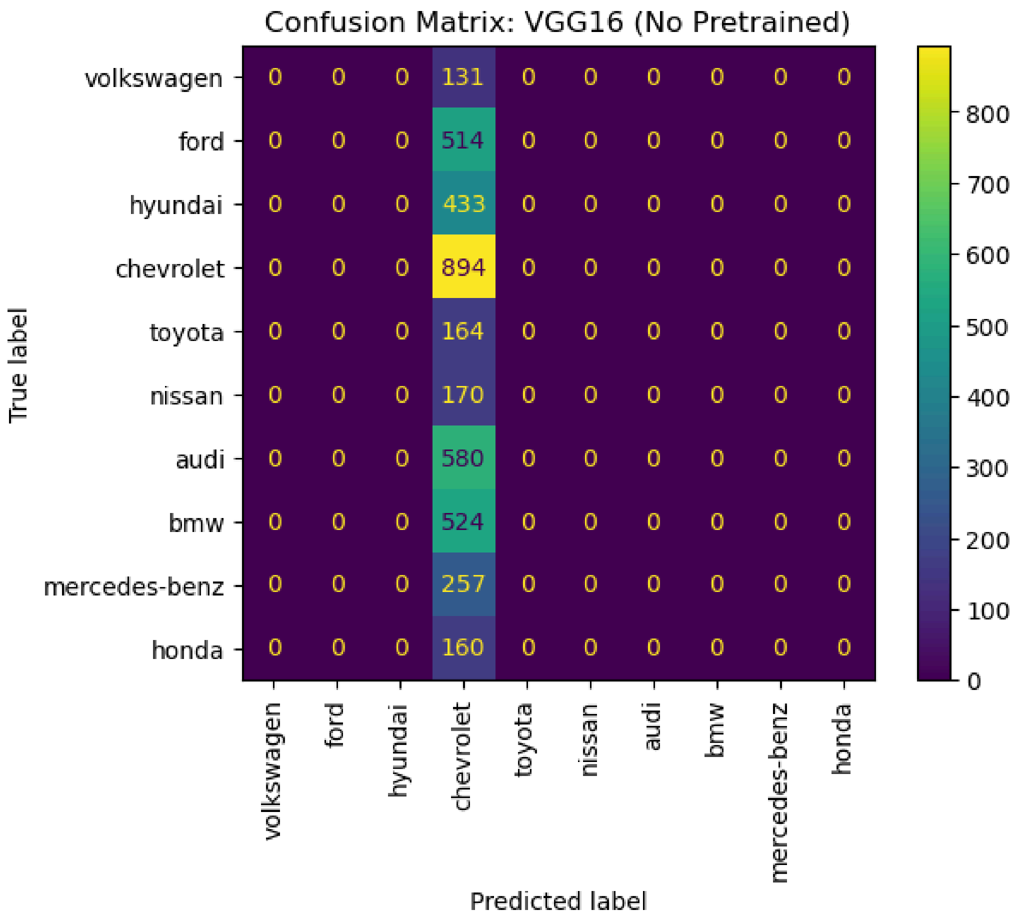


	volkswagen	ford	hyundai	chevrolet	toyota	nissan	audi	bmw	mercedes-benz	honda
volkswagen	0	0	0	131	0	0	0	0	0	0
ford	0	0	0	514	0	0	0	0	0	0
hyundai	0	0	0	433	0	0	0	0	0	0
chevrolet	0	0	0	894	0	0	0	0	0	0
toyota	0	0	0	164	0	0	0	0	0	0
nissan	0	0	0	170	0	0	0	0	0	0
audi	0	0	0	579	0	0	1	0	0	0
bmw	0	0	0	523	0	0	1	0	0	0
mercedes-benz	0	0	0	257	0	0	0	0	0	0
honda	0	0	0	160	0	0	0	0	0	0

4.3.2.3 Plotting Confusion Matrix For VGG16 Without Pretrained Weight

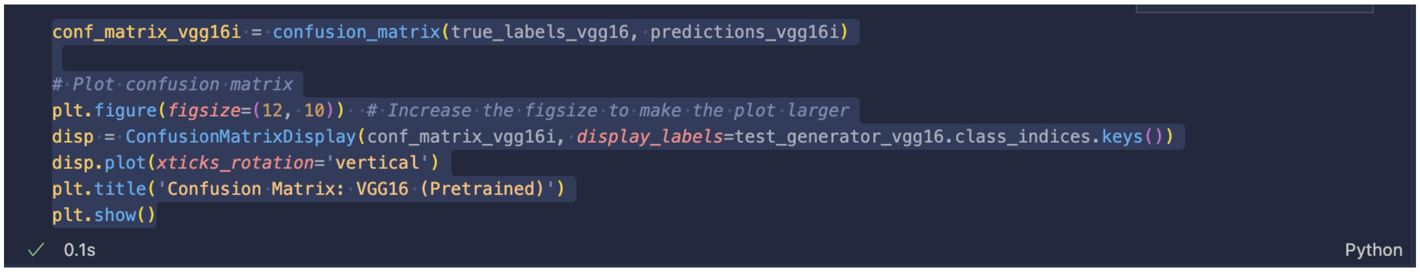


```python
conf_matrix_vgg16d = confusion_matrix(true_labels_vgg16, predictions_vgg16d)

# Plot confusion matrix
plt.figure(figsize=(12, 10))  # Increase the figsize to make the plot larger
disp = ConfusionMatrixDisplay(conf_matrix_vgg16d, display_labels=test_generator_vgg16.class_indices.keys())
disp.plot( xticks_rotation='vertical')
plt.title('Confusion Matrix: VGG16 (No Pretrained)')
plt.show()
```
✓ 0.1s Python



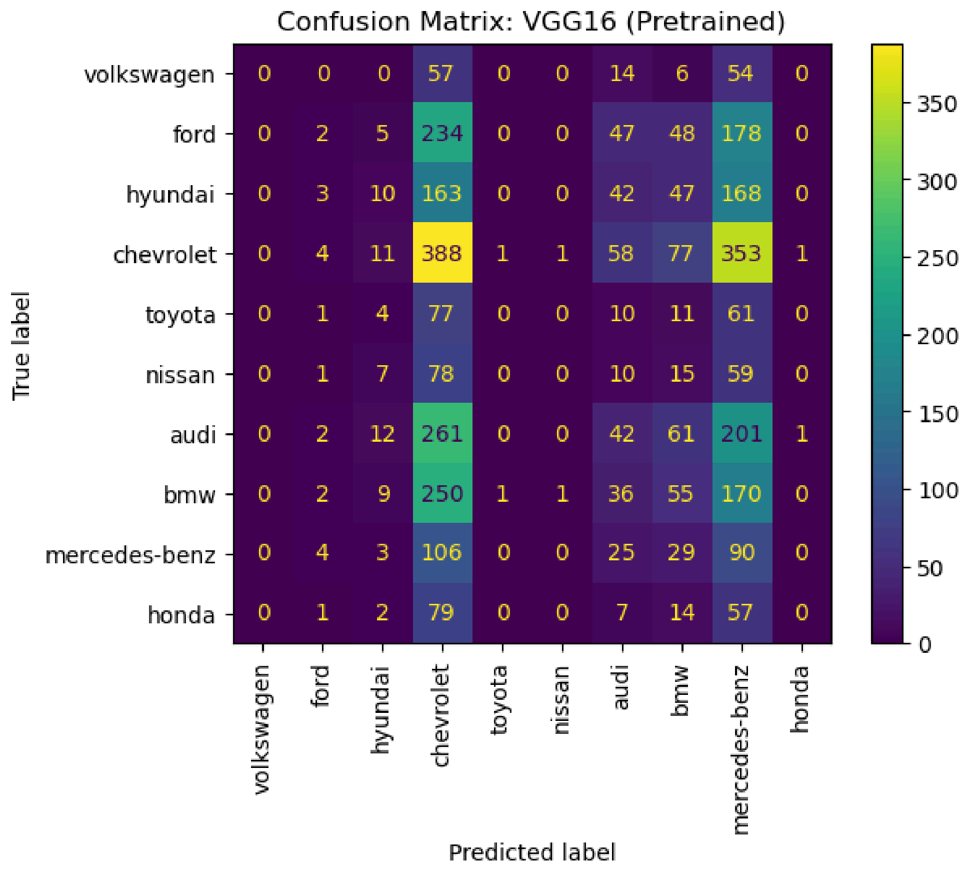
True label \ Predicted label	volkswagen	ford	hyundai	chevrolet	toyota	nissan	audi	bmw	mercedes-benz	honda
volkswagen	0	0	0	131	0	0	0	0	0	0
ford	0	0	0	514	0	0	0	0	0	0
hyundai	0	0	0	433	0	0	0	0	0	0
chevrolet	0	0	0	894	0	0	0	0	0	0
toyota	0	0	0	164	0	0	0	0	0	0
nissan	0	0	0	170	0	0	0	0	0	0
audi	0	0	0	580	0	0	0	0	0	0
bmw	0	0	0	524	0	0	0	0	0	0
mercedes-benz	0	0	0	257	0	0	0	0	0	0
honda	0	0	0	160	0	0	0	0	0	0

4.3.2.4 Plotting Confusion Matrix For VGG16 With Pretrained Weight



```python
conf_matrix_vgg16i = confusion_matrix(true_labels_vgg16, predictions_vgg16i)

# Plot confusion matrix
plt.figure(figsize=(12, 10))  # Increase the figsize to make the plot larger
disp = ConfusionMatrixDisplay(conf_matrix_vgg16i, display_labels=test_generator_vgg16.class_indices.keys())
disp.plot(xticks_rotation='vertical')
plt.title('Confusion Matrix: VGG16 (Pretrained)')
plt.show()
```

✓ 0.1s Python



True label \ Predicted label	volkswagen	ford	hyundai	chevrolet	toyota	nissan	audi	bmw	mercedes-benz	honda
volkswagen	0	0	0	57	0	0	14	6	54	0
ford	0	2	5	234	0	0	47	48	178	0
hyundai	0	3	10	163	0	0	42	47	168	0
chevrolet	0	4	11	388	1	1	58	77	353	1
toyota	0	1	4	77	0	0	10	11	61	0
nissan	0	1	7	78	0	0	10	15	59	0
audi	0	2	12	261	0	0	42	61	201	1
bmw	0	2	9	250	1	1	36	55	170	0
mercedes-benz	0	4	3	106	0	0	25	29	90	0
honda	0	1	2	79	0	0	7	14	57	0

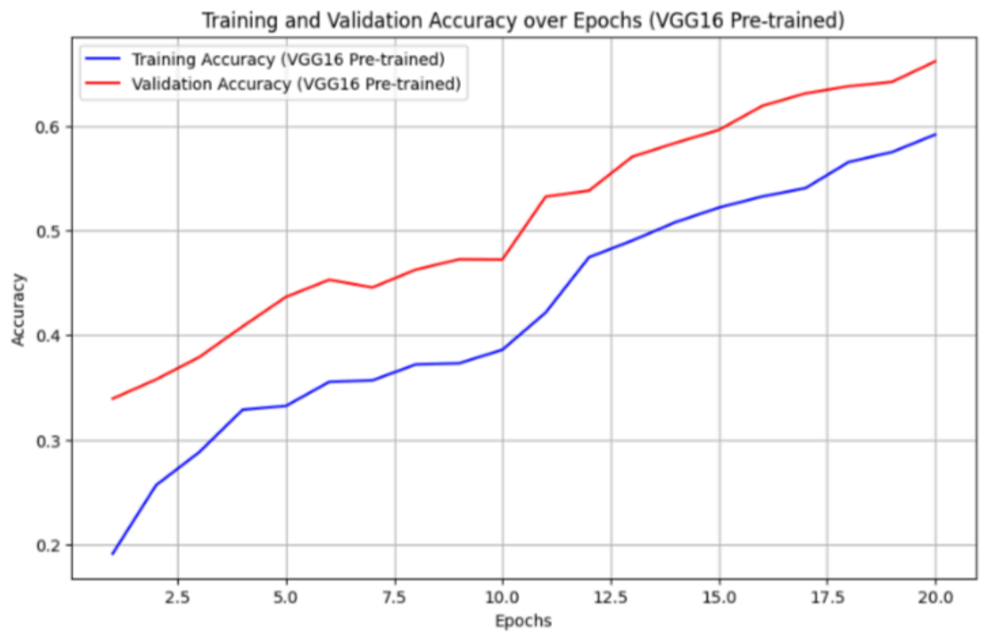
4.3.3 Utilizing models for Accuracy plots

Using this section of code which uses matplotlib.pyplot libraries to plot the epochs against the accuracy. The accuracy were all collected from the summary of the accuracy which are shown above in 4.2.

4.3.3.1 Plotting Accuracy plot for VGG16 with Pretrained Weight

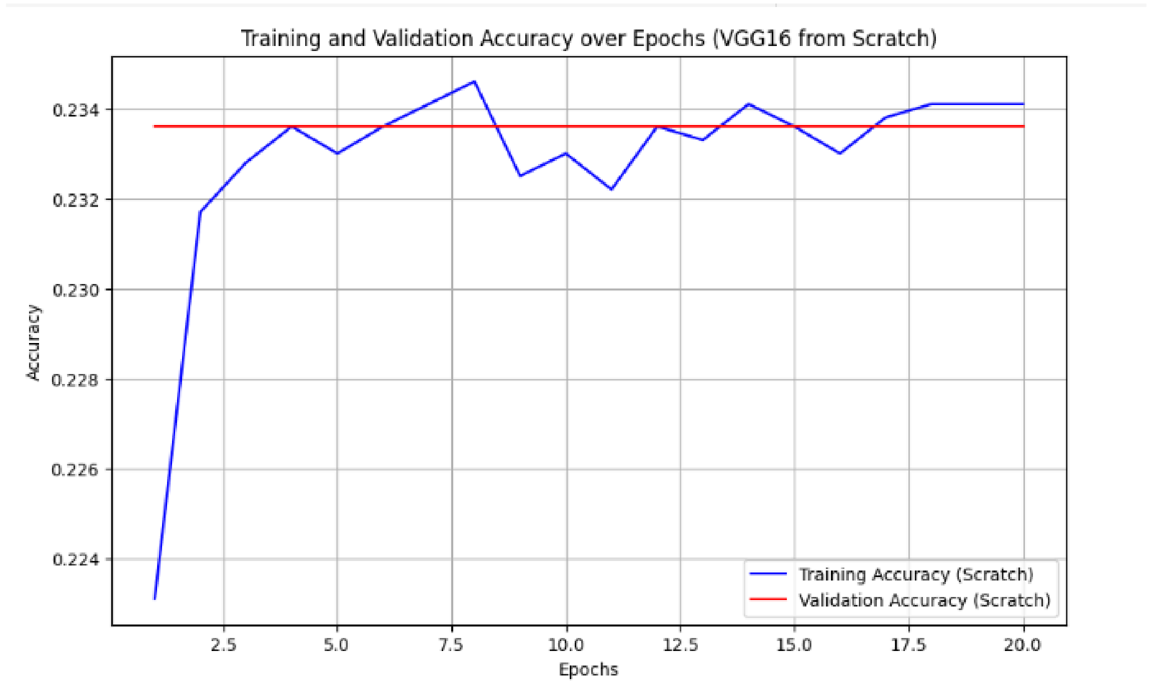
The code and figure below displays the accuracy plot for VGG16 model with pre-trained weights

```python
#Accuracy plot for VGG16 with pre-trained weights

import matplotlib.pyplot as plt

# Epochs
epochs = list(range(1, 21))

# Training and validation accuracy for each epoch (first and second run)
train_accuracy = [0.1911, 0.2564, 0.2881, 0.3286, 0.3322, 0.3552, 0.3567, 0.3719, 0.3730, 0.3859,
                  0.4215, 0.4743, 0.4903, 0.5079, 0.5218, 0.5324, 0.5404, 0.5654, 0.5749, 0.5917]
val_accuracy = [0.3392, 0.3575, 0.3789, 0.4082, 0.4364, 0.4528, 0.4455, 0.4625, 0.4724, 0.4722,
                0.5323, 0.5380, 0.5704, 0.5837, 0.5960, 0.6190, 0.6310, 0.6378, 0.6420, 0.6616]

# Plotting the accuracy
plt.figure(figsize=(10, 6))
plt.plot(epochs, train_accuracy, label='Training Accuracy (VGG16 Pre-trained)', color='blue')
plt.plot(epochs, val_accuracy, label='Validation Accuracy (VGG16 Pre-trained)' , color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy over Epochs (VGG16 Pre-trained)')
plt.legend()
plt.grid(True)
plt.show()
```



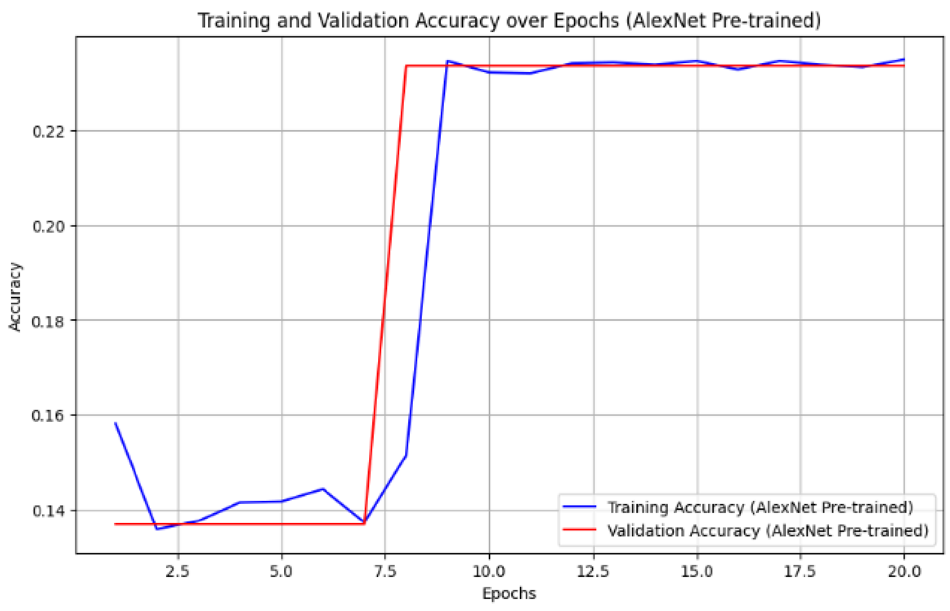
4.3.3.2 Plotting Accuracy plot for VGG16 Without Pretrained Weights

The same code is used except the values of accuracy are changed when plotting the VGG16 model that is trained from scratch.



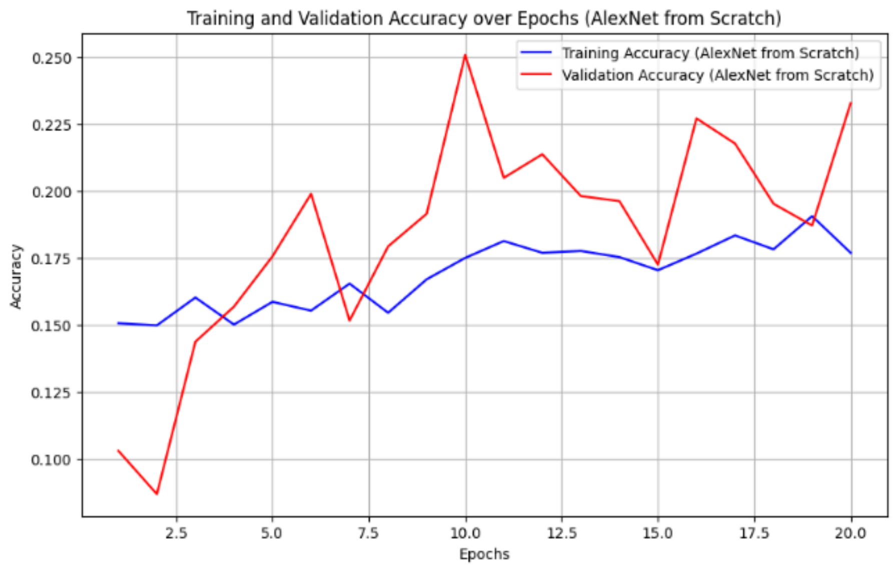
4.3.3.3 Plotting Accuracy plot for AlexNet with Pretrained Weight

Now moving onto the Alexnet model with pre-trained weights, the same code is used except the values of accuracy are changed



4.3.3.4 Plotting Accuracy plot for AlexNet without Pretrained Weight

Finally for the AlexNet model that is trained from scratch, yet again the same code is used with the accuracy. However we can see the model we trained perhaps could've been trained more since it hasn't touched the validation accuracy and hence has room for improvement in terms of accuracy.



5.0 Analysis and Recommendations

In this section we will compare the benchmark algorithm which is Resnet-101 and our own trained VGG 16 and Alexnet used in our experiments to classify cars by their model. Firstly, we will compare the preprocessing techniques between both models

Resnet-101 (Benchmark)	VGG 16 (Our pre-trained model)	Alexnet (Our pre-trained model)
Data Loading, Data Augmentation, Image resizing, Normalization, Label encoder and Train-test-split verification	Data reorganization and Data Filtering and Loading	Data reorganization and Data Filtering and Loading

Since our model was to classify car models only, we downsampled the Stanford car dataset and used the model to classify 10 types of models of cars which were Audi, Hyundai, Toyota, Chevrolet, BMW, Nissan, Ford, Volkswagen, Honda, and Mercedes Benz. For our pre-trained model, the model is designed to preprocess a dataset of car images by reorganizing them based on manufacturers and then setting up a data generator for deep learning purposes. For data filtering and loading, the model takes the parameters such as directory, manufacturers list , batch_size and lists all the classes within the specified directory and filters them based on the provided manufacturers lists. This step assisted in downsampling the dataset from 16,185 images to 7675 images. Downsampling reduces the computational power needed to make predictions as having many images that are redundant can cause the model to use excessive amounts of computational power.

Comparison between models

Resnet-101 (Benchmark)	VGG-16 (Our pre-trained model)	Alexnet (Our pre-trained model)
Deep convolutional neural network	Deep convolutional neural network	Deep convolutional neural network

Performance

The performance of our VGG-16 pre-trained model was significantly higher compared to the pre-trained model of Alexnet which was 66.16% compared to 23.36%. Our VGG-16 model definitely has room for improvement as achieving a high accuracy model necessitates us to carry out extensive experimentation and tuning. Our model may be acceptable for less critical applications with a smaller dataset size or as a starting point for further improvements. However, the relatively low accuracy suggests it may not be suitable for high-stakes or highly precise tasks without significant enhancements. Furthermore, the train-test-split ratio we used was 80% of training and 20% for testing as the default. The 80% of data for training ensures that the both models VGG 16 and Alexnet have access to a substantial amount of data to learn from which is vital in deep learning models with large datasets. More training data means the models have a better understanding of the underlying patterns in the car dataset, leading to improved learning and generalization.

Comments

The VGG-16 pretrained model significantly outperforms the Alexnet pre-trained model in terms of accuracy and reliability. Both models have achieved the purpose of classifying different car classes by the car model. Although in part 3.4 of model evaluation, it is mentioned that Alexnet had a top-5 error rate of 17.0% and a top-1 error rate of 37.5%, we did not see this happen when we used the pre-trained model of Alexnet. This demonstrates that not every model is built for a specific task that guarantees a high accuracy rate. VGG-16 is a better model altogether as mentioned before it has a deep design of 16 layers and smaller filters which allow it to extract features at a more higher hierarchical level. Additionally, despite our VGG-16 model producing an accuracy of 66.16% it still was significantly lower than others that have trained using the same dataset such as (Alghamdi et al., 2023) in our literature review and this goes to show how different data augmentation and fine-tuning methods can drastically affect the accuracy of the model . Steps to improve our model are stated in the next paragraph.

Recommendations:

1. Data Augmentation and Data preparation

Applying transformation to the data such as rotation, zoom, shift, shear, flip and brightness adjustments can help create a more diverse training sample for the model to work with. By doing so, the model is able to generalize better by learning from a wider variety of examples. To improve further we can try experimenting with very differing values in those methods to see if it may help improve the classification performance of the model. On top of this we can also experiment more with colours and see how saturation, contrast, and other lighting techniques can help with improving the model accuracy.

Another method we could've tried to focus on was random cropping to focus on more important features that can be crucial to identify different car classes. We could also normalize the data to ensure that the pixel values of images are normalized for example we can scale the car image to a range of [0,1] or [-1,1] and also carry out proper preprocessing techniques ensure that the data fed into the model is consistent and can lead the model to better convergence during training. Finally we could also try experimenting with different batch sizes and see how much of an impact it can have in training the model.

2. Model Architecture and tuning

In the future, we can try improving VGG16 model's performance in classifying car models by considering expanding its architecture by adding more layers or neurons, or exploring deeper models like VGG19. This approach can help capture more intricate features and patterns, improving the model's accuracy. Fine-tuning using different number of layers for unfreezing and freeazing these weights on our specific dataset can make the model more adept at recognizing subtle differences between car models and additionally also utilzing different optimizers rather than Adam such as SGD.

3. Prevent overfitting

Although data augmentation can help reduce overfitting, we should try other methods such as early stopping where we stop the training once the accuracy starts to plato or drop, such as in

4.3.2 in training VGG16 without pre-trained weights earlier on to 8 epoch, our accuracy is no longer improving and is instead starting to drop at certain points and this led to waste of computational resources and time to train. There are also regularization techniques will help to control overfitting and improve the model's ability to generalize such as weight decay or L2 Regularization and dropout which prevents the model from relying on one or few specific features that may prevent the model from not learning other important features that can help in detection.

4. Stratified splitting

We can implement this method to ensure that if car model class like "chevrolet" which takes up 724 images and around 12% of the total training set of 6059 images, and 181 images which is also close to 12 % of the total test set of 1616 images. Similarly pretty much all our car model classes takes up around the same % of total train and test set and this allows for more unbiased and accurate training.

6.0 Conclusion

Using the Stanford Cars Dataset we chose VGG16 and AlexNet to train and compare against the benchmark of Resnet101 in our assignment which is a more complex and deeper

architecture capable of extracting in depth features to allow for more accurate classification. Despite knowing AlexNet would perform relatively poorly we wanted to see how big the difference it would be between an older and less complex CNN compared to VGG16 and the results not only show VGG16 is vastly superior compared to AlexNet but it also shows how despite using pre-trained weights AlexNet still did not improve much in terms of accuracy which shows its feature extracting is not nearly as good as compared to more advanced models.
For VGG16, the best accuracy was 66.16% which is far below the benchmark of our literature review which was 97.7%. Additionally, our model also did not even compare to the 99.7% that was achieved by (Alghamdi et al., 2023). Despite our model having a lower score however, it still managed to have a significantly higher accuracy than the VGG16 we trained from scratch which was only at 23.36% this shows how important pre-trained weights are especially those trained on a dramatically larger dataset like ImageNet. One strength our model had however was that it didn’t take nearly as long to train as we thought and also took less resources than ResNet101 would’ve taken and hence VGG16 can take advantage of its less complex architecture to train at a faster pace and while having a deeper architecture than AlexNet which allows it to have a better accuracy at classification.
For our recommendation we recommend experimenting with VGG19 and other models like ResNet50 and ResNet101 to compare how a similar architecture with an increased number of layers performs relative to VGG16. These comparisons may provide insights into understanding how important network depth are on model accuracy. Additionally, we suggest changing up data augmentation values to produce varying rotation angles, adjusting lighting conditions and zoom, changing batch sizes, and also implementing cropping strategies and also to normalize the data. These techniques may help the model focus more on distinctive car features, potentially improving classification accuracy. Another area for improvement would be standardizing the dataset size across all car models. This approach would address the imbalance where some classes, like "Audi" and "Chevrolet," have significantly more images compared to "Toyota" and "Honda." Ensuring a balanced dataset, even after augmentation, could lead to more equitable training and better overall model performance. Moreover we can implement stratified splitting for more fair and unbiased split of our % of car images in each class. Finally we can also prevent overfitting by using regularization techniques like weight decay or dropout that prevents the model from relying on few specific features only when training.

7.0 References

1. **Revisiting the CompCars Dataset for Hierarchical Car Classification: New Annotations, Experiments, and Results By Marco Buzzelli, Luca Segantin Container: Sensors Publisher: Multidisciplinary Digital Publishing Institute Year: 2021 Volume: 21 Issue: 2 DOI: 10.3390/s21020596 URL: https://www.mdpi.com/1424-8220/21/2/596**

2. **Improving Car Model Classification through Vehicle Keypoint Localization By Alessandro Simoni, Andrea D'Eusanio, Stefano Pini, Guido Borghi, Roberto Vezzani Container: IRIS UNIMORE (University of Modena and Reggio Emilia) Publisher: University of Modena and Reggio Emilia Year: 2021 DOI: 10.5220/0010207803540361 URL: https://iris.unimore.it/handle/11380/1229971?mode=complete**

3. **System for detecting car models based on machine learning** By Keerthana B Chigateri, Sujay Suryavamshi, Shrihari Rajendra Container: Materials today: proceedings Publisher: Elsevier BV Year: 2022 Volume: 52 DOI: 10.1016/j.matpr.2021.11.335 URL: https://www.sciencedirect.com/science/article/abs/pii/S2214785321073478?casa_token=pqyn3myj4zsAAAAA%3AnhQ7139N4tYOZZaPC9RmIFVunTpPXoLnIbeAYE3j0BYgC7G_5FkZzqVS8ZjTXLxkWFYCQ06Wa5-oA

4. **End-to-End Car Make and Model Classification using Compound Scaling and Transfer Learning** By Omar BOURJA, Abdelilah MAACH, Zineb ZANNOUTI, Hatim DERROUZ, Hamd AIT ABDELALI, Rachid OULAD HAJ THAMI, Francois BOURZEIX Container: International Journal of Advanced Computer Science and Applications Year: 2022 Volume: 13 Issue: 5 DOI: 10.14569/ijacsa.2022.01305111 URL: https://thesai.org/Downloads/Volume13No5/Paper_111-End_to_End_Car_Make_and_Model_Classification_using_Compound_Scaling.pdf

5. **Automobile classification using transfer learning on ResNet neural network architecture** By Stjepan Ložnjak, Tin Kramberger, Ivan Cesar, Renata Kramberger Container: Polytechnic and design Year: 2020 Volume: 8 Issue: 01 DOI: 10.19279/tvz.pd.2020-8-1-18 URL: https://hrcak.srce.hr/clanak/352672

6. **Vehicle Classification Using Deep Feature Fusion and Genetic Algorithms** By Ahmed S Alghamdi, Ammar Saeed, Muhammad Kamran, Khalid T Mursi, Wafa Sulaiman Almukadi Container: Electronics Publisher: Multidisciplinary Digital Publishing Institute Year: 2023 Volume: 12 Issue: 2 DOI: 10.3390/electronics12020280 URL: https://www.mdpi.com/2079-9292/12/2/280

7. **An Empirical Analysis of Deep Learning Architectures for Vehicle Make and Model Recognition** By Aqsa Hassan, Mohsin Ali, Nouman M Durrani, Muhammad Atif Tahir Container: IEEE access Publisher: Institute of Electrical and Electronics Engineers Year: 2021 Volume: 9 DOI: 10.1109/access.2021.3090766 URL: https://ieeexplore.ieee.org/document/9460843

8. **Hierarchical System for Car Make and Model Recognition on Image Using Neural Networks** By Ivan Fomin, Ivan Nenahov, Aleksandr Bakhshiev Year: 2020 DOI: 10.1109/icieam48468.2020.9112026 URL: https://ieeexplore.ieee.org/document/9112026

9. **A novel part-level feature extraction method for fine-grained vehicle recognition**

   By Lei Lu, Ping Wang, Yijie Cao Container: Pattern recognition Publisher: Elsevier BV Year: 2022 Volume: 131 DOI: 10.1016/j.patcog.2022.108869 URL: https://www.sciencedirect.com/science/article/abs/pii/S0031320322003508?casa_token=Pc9CuvTwBvIAAAAA%3AFfgIEwwyePcyuPuzbokRxs6tIIVlU8LTS_OuqpQHWiTd0zuXXmCdIlgFkgW0HZwQP0JIFEDKIAOMgw

10. **Learning to locate for fine-grained image recognition**

    By Jiamin Chen, Jianguo Hu, Shiren Li Container: Computer vision and image understanding Publisher: Elsevier BV Year: 2021 Volume: 206 DOI: 10.1016/j.cviu.2021.103184 URL: https://www.sciencedirect.com/science/article/abs/pii/S107731422100028X

11. **Hybrid quantum ResNet for car classification and its hyperparameter optimization** By Asel Sagingalieva, Mo Kordzanganeh, Andrii Kurkin, Artem Melnikov, Daniil

Kuhmistrov, Michael Perelshtein, Alexey Melnikov, Andrea Skolik, David Von Dollen Container: Quantum Machine Intelligence/Quantum machine intelligence Publisher: Springer Science+Business Media Year: 2023 Volume: 5 Issue: 2 DOI: 10.1007/s42484-023-00123-2 URL: https://link.springer.com/article/10.1007/s42484-023-00123-2

12. **Deep CNN Model based on VGG16 for Breast Cancer Classification**

    By Dheeb Albashish, Rizik Al-Sayyed, Azizi Abdullah, Mohammad Hashem Ryalat, Nedaa Ahmad Almansour Year: 2021 DOI: 10.1109/icit52682.2021.9491631 URL: https://ieeexplore.ieee.org/document/9491631

13. **AlexNet Convolutional Neural Network for Disease Detection and Classification of Tomato Leaf** By Hsing-Chung Chen, Agung Mulyo Widodo, Andika Wisnujati, Mosiur Rahaman, Jerry Chun-Wei Lin, Liukui Chen, Chien-Erh Weng Container: Electronics Publisher: Multidisciplinary Digital Publishing Institute Year: 2022 Volume: 11 Issue: 6 DOI: 10.3390/electronics11060951 URL: https://www.mdpi.com/2079-9292/11/6/951

14. **AlexNet architecture based convolutional neural network for toxic comments classification** By Inderpreet Singh, Gulshan Goyal, Anmol Chandel Container: Journal of King Saud University. Computer and information sciences/Mağalaẗ ğam'aẗ al-malīk Saud : ùlm al-ḥasib wa al-ma'lumat Publisher: Elsevier BV Year: 2022 Volume: 34 Issue: 9 DOI: 10.1016/j.jksuci.2022.06.007 URL: https://www.sciencedirect.com/science/article/pii/S1319157822002026?via%3Dihub

15. **Car make and model classification from image** By Alexandros Kelaiditis Container: ResearchGate Publisher: unknown Year: 2023 URL:

https://www.researchgate.net/publication/372159544_Car_make_and_model_classification_from_image

16. Why Data Preprocessing is Necessary in Data Science

    By Gideon Markus Container: Medium Publisher: Medium Year: 2024 URL: https://medium.com/@gideonmarkus/why-data-preprocessing-is-necessary-in-data-science-546235345fdb#:~:text=One%20of%20the%20primary%20reasons,and%20lead%20to%20erroneous%20conclusions

17. ImageNet Classification with Deep Convolutional Neural Networks

    By Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton Container: Communications of the ACM Year: 2012 Volume: 60 Issue: 6 DOI: 10.1145/3065386 URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

18. Vehicle Type Recognition Algorithm Based on Improved Network in Network

    By Erxi Zhu, Min Xu, De Chang Pi Container: Complexity Publisher: Hindawi Publishing Corporation Year: 2021 Volume: 2021 DOI: 10.1155/2021/6061939 URL: https://onlinelibrary.wiley.com/doi/10.1155/2021/6061939

19. How to Measure Model Performance in Computer Vision: A Comprehensive Guide

    By Zoumana Keita Container: Encord.com Publisher: Encord Blog Year: 2023 URL: https://encord.com/blog/measure-model-performance-computer-vision/#:~:text=Cla

ssification%20models%20can%20be%20evaluated,%2Dscore%2C%20and%20confusion%20matrix