Vehicle Classification with VGG-16

Team Members:

Kaustubh (21UCS108)
Saumya Saraswat (21UCS184)
Manas Vashisht (21UCS125)

Abstract

Our project implements a Convolutional Neural Network, which is currently state-of-the-art in image recognition technologies. In our project, we use the primary structure of the VGG-16 Convolutional Neural Network, initially submitted for the ImageNet ILSVRC Challenge by the Visual Geometry Group (VGG) at the University of Oxford.

Our project presents two approaches to vehicle classification using VGG-16; first, we use the weights of the pre-trained VGG-16 network submitted by the University of Oxford. Using transfer learning, our first implementation freezes the weights of VGG-16 layers with the original pre-trained weights (From the ILSVRC Submission). It connects it to a trainable, fully connected layer for classification. The fully connected layer is then trained on our dataset for classification. In our second approach, we build a VGG-16 Neural Network from scratch and train the entire network, from the convolutional layers to the fully connected layers on our specific dataset for the same classification. We then compare the accuracy and try to justify the results from the two approaches in our project.

Introduction

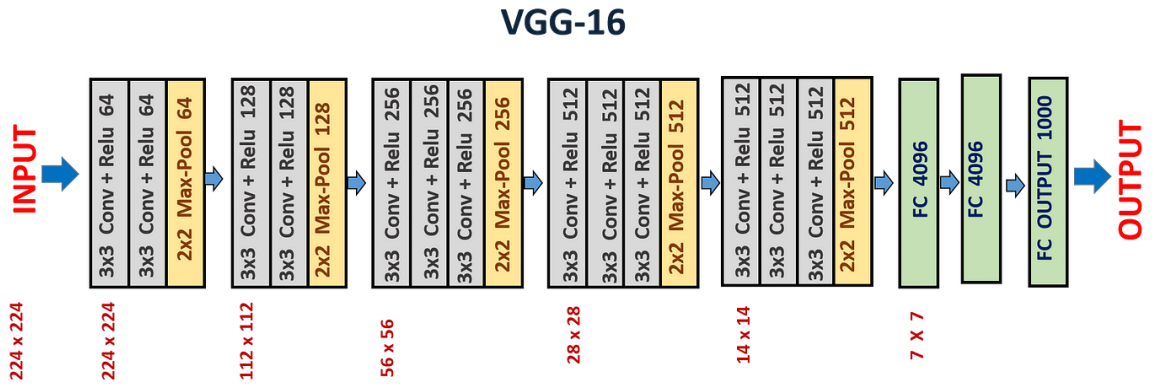
Our project intends to classify vehicles on the Stanford Cars Dataset**[1]**, which contains 16,165 images of 196 cars. The project addresses the problem of utilizing three-dimensional object representations for multi-view object class detection and scene comprehension. The growing area of fine-grained recognition in computer vision is becoming more critical because it helps us notice subtle differences in appearance, which has practical, real-world applications. This dataset focusing on cars is particularly advantageous, providing robust training and testing sets essential for developing models adept at distinguishing one car from another, irrespective of the perspective the image might be from.

This is a stepping stone in the broader field of object recognition, where models can be built to recognise objects in images, irrespective of the spatial orientation of the object in the image. Moreover, this specific application also has the scope of being deployed in various traffic safety compliance systems to accurately identify vehicles using a combination of the registration number and the make and model of cars. This can help build a more accurate and reliable system for automatically ensuring and monitoring traffic and road safety rules compliance.

About VGG-16

VGG stands for the “Visual Geometry Group”. It is a research group based at The University Of Oxford. They work on Convolutional networks (ConvNets), which currently set the state of the art in visual recognition. The project aims to investigate how the ConvNet depth affects their accuracy in large-scale image recognition settings. The VGG layers only use 3x3 filters with stride 1 in each convolutional layer.

The VGG-16 network is a version of the VGG convolutional neural network with 13 convolutional layers, followed by 3 fully connected layers. A schematic of VGG-16 has been given below:



The above schematic is for the ImageNet VGG-16 network**[2]**, as evident by the 1000 output neurons. We use this as the basis for our neural network and make appropriate modifications, using two approaches to perform vehicle identification on our dataset.

Literature Review:

Here are some prerequisites one must understand to understand our project fully:

Image Representation and Preprocessing:

- **Pixels and Channels:** Images are essentially a grid where each pixel helps represent the image's colour and intensity. Three channels (red, green and blue) are combined to form a colour image.
- **Normalisation:** Resizing the dataset to a standard scale is an essential part of any deep learning project, as each image may have different sizes and variations in brightness and contrast. Normalisation helps in bringing all the model's inputs to a standard scale.

Feature Extraction:

- **Convolutional Neural Network:** The convolutional Neural Network (CNN) is the main component of the whole project. It helps extract specific details like edges, shapes and textures from an image and use them to classify the image.
- **Feature Maps:** Feature maps are the output of each convolution layer when applied to the images. CNNs produce multiple layers of feature maps that capture different image details.
- **Pooling Layer:** This layer helps pick the most appropriate activation in a region by learning more robust representations of the input data. It also reduces the spatial dimension of the input, which reduces the computational cost of the network.
- **Fully Connected Layers:** This layer helps introduce non-linearity, allowing the network to learn complex relationships between features.

Machine Learning and Model Training:

- **Supervised Learning:** Vehicle Classification is a supervised learning task where a model learns from labelled data to make predictions or decisions for new or unseen data.
- **Optimization:** There is a need to minimise the difference between the predicted and actual values so that the model works well on new or unseen data efficiently.
- **Training and Testing split:** The dataset is divided into training and testing datasets. The model is trained on the training dataset and evaluated on the unseen set to make the model predict accurately.

Evaluation and Metrics:

- **Accuracy:** The most common metric refers to the proportion of correct predictions the model makes. It also shows how well the model can generalise its learning from the training data to unseen data.

- **Loss Function:** This metric helps us quantify the classification error. It is essential for classification tasks as the final output isn't numeric. However, we can still quantify the classification loss using an effective loss function despite only present class labels. In our project, we use the Cross-Entropy Loss function for the same.

Various other alternative models can be used for vehicle classification. These models have some potential drawbacks compared to VGG-16:

- **ResNet:** This architecture is newer than VGG 16, which helps solve the problem of vanishing and exploding gradients problem by introducing skip and residual connections. However, they can be more complex to implement and require more computational resources.
- **InceptionNet:** This architecture is also a newer one implemented after VGGNet emerged. This architecture is also famous due to the feature extraction improvement. It also introduces the concept of an auxiliary classifier that helps push useful gradients to the lower layers. Despite various advantages, this architecture risks overfitting and can be computationally expensive compared to the VGG network.
- **Ensemble Methods:** Combining multiple layers can improve generalizability and robustness compared to a single model. However, they can be more complex to interpret and require careful hyperparameter tuning.

Related Works

Here are some highly influential and impactful publications in this area, each offering unique insights and contributions:

1. **"Fine-Grained Image Classification for Vehicle Makes and Models using Convolutional Neural Networks" by Zhu et al. (2019):**
    - This paper focuses on the fine-grained classification of car makes and models. It achieves state-of-the-art accuracy on the Standard Cars dataset using VGG-16 with fine-tuning.
    - This paper evaluates the potential of data augmentation, transfer learning, and ensemble learning to improve performance.
    - This publication is the starting point for vehicle classification using VGG networks. It helps understand the challenges and common approaches for vehicle classification with VGG.
2. **"Transfer Learning for Vehicle Make and Model Recognition using Deep Learning" by Li et al. (2020):**
    - This paper focuses on transfer learning on vehicle classification with limited data, utilising pre-trained VGG-16 models.
    - This paper helps maximise performance with limited data by exploring fine-tuning and data augmentation strategies.
    - This publication uses the concept of VGG-16 vehicle classification with limited training data and helps understand the benefits and challenges of transfer learning.
3. **"An Empirical Analysis of Deep Learning Architectures for Vehicle Make and Model Recognition" by Khan et al.(2021)**
    - This paper helps analyse the performance of vehicle classification using different architectures of deep learning, including VGG-16, ResNet, and MobileNet, for vehicle make and model recognition.
    - It also explores the impact of optimisation techniques, regularisation strategies, and ensemble methods on accuracy.
    - This paper provides a broader perspective on the suitability of using different deep-learning architectures for vehicle classification and also explores valuable insights into hyperparameter tuning and optimisation strategies.
4. **"Towards Scalable and Efficient Vehicle Make and Model Recognition" by Yang et al.(2022)**

    - This paper leverages knowledge distillation and channel pruning to optimize VGG-16 for vehicle classification at scale.
    - It highlights the advantages of knowledge distillation for improving while reducing model size, making it suitable for resource-constrained environments.
    - It also provides valuable insights into using VGG-16 for vehicle classification.

There are various other state-of-the-art approaches to this project. Here are some of them

- **Deeper and Wider architecture:** EfficientNet- V3 and Swin Transformers balance with efficiency, making them suitable for real-time applications.
- **Attention Mechanisms:** Transformers and their derivatives focus on specific car components and subtle details, leasing to better fine-grained classification.
- **Ensemble and Meta-Learning:** combining multiple models or meta-learning approaches can improve robustness and adapt to new car classes with limited data.
- **Data augmentation and pre-processing:** weak supervision, domain-specific augmentation, and self-supervised learning techniques improve performance and reduce reliance on a large labelled dataset.

Methodology

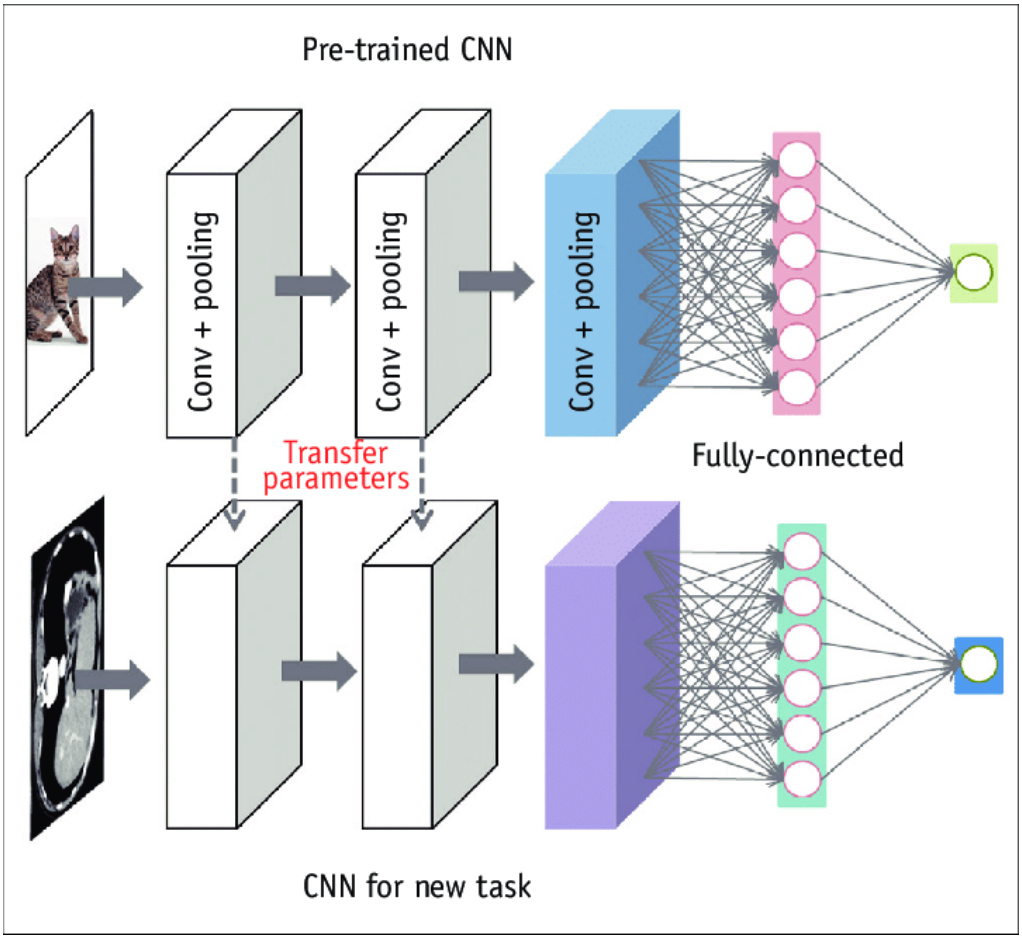
Our project compares two approaches to perform the vehicle classification task on the Stanford Cars Dataset. Our first approach involves using Transfer Learning, while the second approach trains a VGG-16 structured neural network and trains the entire network from scratch. We shall see more about the individual approaches below.

Approach 1: Transfer Learning Approach

Key Idea:

Using the Transfer learning approach, we can use already pre-trained networks as feature extractors and modify subsequent layers to suit our classification task. The significant advantage of this approach is the reduced training time for the network. Pre-trained networks will already be capable of giving us a refined set of features from the input image. By training just a fully connected network connected to the pre-trained network, we can directly use these refined features to perform the classification task.

This approach also has far fewer trainable parameters, so it will be easier to train just our fully connected layer's weights over the training set. Such networks offer solid performance with minimal training effort required compared to other approaches. Such approaches are used to train networks when we need quick and stable results or just want to prototype a certain model's use case before dedicating large-scale resources towards training it.



The Transfer Learning Approach For Training Models

Network Structure:

```
# Load VGG16 pre-trained on ImageNet
print('Loading pre-trained VGG16 model...')
model = VGG16(
    include_top=False, weights='imagenet', input_shape=(224, 224, 3)
)

# Freeze the pre-trained weights
for layer in model.layers:
    layer.trainable = False

# Add a new trainable FC layers
flatten_layer = Flatten()(model.layers[-1].output)
dense_layer = Dense(4096, activation='relu', name='fc1')(flatten_layer)
dense_layer2 = Dense(4096, activation='relu', name='fc2')(x)
output_tensor = Dense(196, activation='softmax', name='predictions')(x)

# Creating Model
model = Model(inputs=model.input, outputs=output_tensor, name=vgg16_tl)
```

In this approach, we build a VGG-16 network by invoking the library function. We construct the network using the pre-trained weights for the ImageNet dataset, and by specifying *include_top = False,* we can obtain all the pre-trained convolutional layers for our network.

These pre-trained weights are then frozen to allow us to use the ImageNet weights as a feature extractor. The output from the convolutional layers is then flattened and connected to 2 fully connected layers, followed by a single output layer, akin to the original VGG-16 network.

The dimensions of the fully connected layer are specified in the image below:

```
flatten_layer = Flatten()(model.layers[-1].output)
dense_layer = Dense(4096, activation='relu', name='fc1')(flatten_layer)
dense_layer2 = Dense(4096, activation='relu', name='fc2')(x)
output_tensor = Dense(196, activation='softmax', name='predictions')(x)
```

The final output layer has 196 output neurons, each corresponding to a single possible output class. Moreover, a softmax activation function is used as the activation function for the output layer to give the most appropriate output.

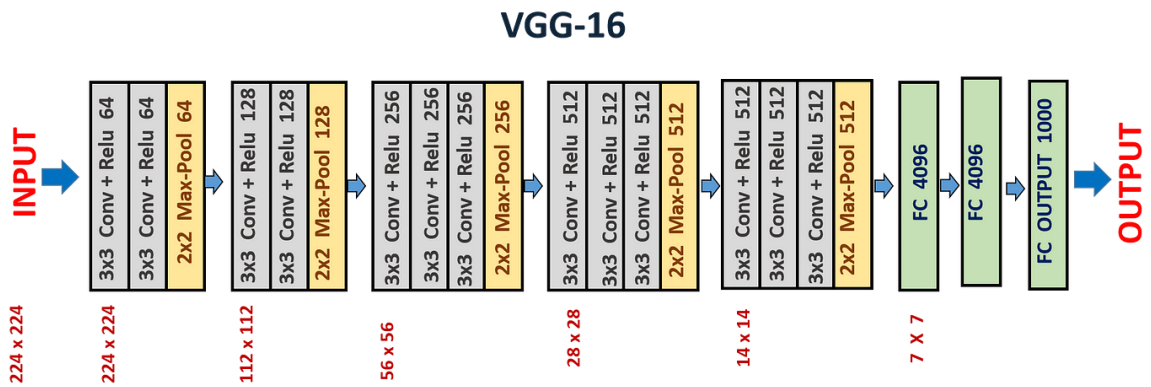
Approach 2: Training a VGG-16 Neural Network from Scratch

Key Idea:

We build a VGG-16 network with all the required layers from scratch. We train the entire network, from the convolutional layers to the fully connected layers, on our training dataset. This is similar to the classical approach towards training neural networks for a task. This approach promises better performance on our specific dataset than a more general-purpose model developed using transfer learning, as seen in Approach 1.

The major downside of this approach comes from the fact that it has a much larger number of learnable parameters than a transfer learning approach. Consequently, these networks take more time and epochs to train until optimal performance is achieved. However, properly trained models using this approach will be more reliable

and perform much better than a transfer learning-based model, as the entire network is trained on the application-specific dataset, resulting in a more stable and accurate model overall. This shows the potential of this approach to train final, stable models for a specific application after choosing the best possible prototype.



The Classical VGG-16 Network Architecture

Network Structure:

```python
def vgg16(input_shape=(224, 224, 3), num_classes=1000):
  input_tensor = Input(shape=input_shape)
  # Block 1
  x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(input_tensor)
  x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)
  # Block 2
  x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
  x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)
  # Block 3
  x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
  x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
  x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)
  # Block 4
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)
  # Block 5
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
  x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)
  # Flatten and dense layers
  x = Flatten(name='flatten')(x)
  x = Dense(4096, activation='relu', name='fc1')(x)
  x = Dense(4096, activation='relu', name='fc2')(x)
  output_tensor = Dense(num_classes, activation='softmax', name='predictions')(x)
  # Create model
  model = Model(inputs=input_tensor, outputs=output_tensor, name='vgg16')
  return model
```

As seen above, we define a function to return a classical VGG-16 model. We also use *num_classes = 1000*, setting the default value for this parameter as 1000, similar to the classical VGG-16 model. This parameter is later modified to 196 when calling this function to build a model for our project, as seen below:

```
model = vgg16(num_classes = 196)
```

Here, no weights are frozen. Instead, similar to the typical approach to training neural network models, we train the entire network throughout the training set to update all the network parameters. This gives us a final model better suited for this specific classification task.

Loss Function

Both our approaches use the same loss function, i.e. the Categorical Cross Entropy Loss function to ensure a fair comparison between the two. It is a standard loss function used for calculating the loss for multi-class classification.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Mathematical Representation of the Loss Function

Optimizer

Both our models use the same optimizer to ensure a fair comparison between the two models. Our optimizer is a form of Stochastic Gradient Descent with Nesterov Momentum, as seen below:

```
opt = SGD(learning_rate=0.001, momentum=0.9, nesterov=True)
model.compile(
    loss='categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)
```

While this optimizer uses more parameters, and requires more epochs to reach convergence, it has experimentally proven to be a higher performance model as it requires lesser overall training time to reach convergence. The momentum parameter also ensures that the change in the parameters is also dependent on the previous parameter changes, this prevents large fluctuations in the values of the gradient observed during training.

Regularizers

We have not used L1 or L2 regularizers in our model. Instead, to prevent overfitting we make use of the following callbacks, as seen in the image below:

```
early_stopping = EarlyStopping(patience=10, restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(factor=0.5, patience=3)
callbacks = [early_stopping, lr_scheduler]
```

The *callback* list is provided to the *model.fit()* method when training the models for both our approaches. Early stopping allows the training to stop early if, for 10 continuous changes, the performance seems to deteriorate; and

restores the best weights from those 10 epochs. The *lr_scheduler* allows for modifying the learning rate for each iteration of the Stochastic Gradient Descent. The learning rate is reduced by a factor of 0.5 if a plateau is reached for 3 consecutive updates.

Additional Details

We use the technique of Data Augmentation to increase the size of the training dataset further artificially. This technique takes the initial training dataset and performs various transformations on the dataset's images, like rotations, width shifts, height shifts, zooming, shearing, horizontal flipping, etc, to automatically create more labelled data in addition to the base dataset, which we used for training our model.

```
from keras.preprocessing.image import ImageDataGenerator
aug = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

Our Specifications for Image Augmentation

Experimental Setup

Once the above-described methodology and the required structures for it had been established, we trained the two models using the code below:

```
# Train model
H = model.fit(
    aug.flow(X_train, y_train, batch_size=32),
    validation_data=(X_val, y_val),
    steps_per_epoch=len(X_train)//32,
    epochs=50,
    callbacks=callbacks,
    verbose=1
)
```

Both Approach 1 and Approach 2 Models were trained using this identical code to ensure consistency

For all our used methods, we used the example source code we accumulated through our deep learning classes, as well as the Keras documentation and made appropriate modifications to those, specific to our intended application.

Also, as seen in the code above, augmentation is only performed for the training split of the dataset. No such modification is performed for the testing set.

Our Dataset

We used the Stanford Cars Dataset for our project. This dataset contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the Make, Model, and Year. For example: the 2012 Tesla Model S or 2012 BMW M3 coupe.

For our project, we have preprocessed the dataset by performing random cropping and augmentation to artificially increase its size for training and attempt to create a better-trained model overall. This was done after the standard operations of resizing all the images to be compatible with the maximum dimensions of the VGG Network, which is (224,224,3).

We also had to download the dataset separately and upload it to our colab instance, as we could not load it directly from a web source into our runtime instance. This, however, wasn't a significant issue in the long run, as we also chose to save the preprocessed dataset to reduce processing loads on subsequent runs of our entire code.

Our Data Preprocessing

First, all our dataset's images were read and resized to a size of 256*256 to prepare it for further processing. The images were added to a list called *img_list.* This list was then appended to an *imgs_array* array, as seen below

```
img_list = []
for img in img_paths:
    img = cv2.imread(img)
    try:
      img = cv2.resize(img, (256, 256))
    except:
      break
    img_list.append(img)

imgs_array = np.array(img_list)
```

A similar process was followed for our labels. They were turned into discrete numerical values for us so we could train the model, and were added to another array as well, as seen below:

```
# Initialize the list of labels
labels_list = []
# Read and extract data from the labels text file
f = open(LABELS, 'r')
for row in f:
    labels_list.append(int(row))
f.close()
# Labels in array format
labels = np.array(labels_list)
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
print('Labels created!')
```

This gives us another array of labels represented as discrete numerical values. This allows us to use our selected loss function to compute loss values during training.

After this, the RGB values for all the training images were mean-normalized, followed by random cropping to ensure the appropriate size of 224*224, which is compatible with the VGG-16 network. This process, allows us to create completely unique images with sufficient variations in the positioning of the subject which can help us train an effective network for the classification task. The code for the same has been shown below:

```
# Training set
# Prepare RGB values for mean normalization
R, G, B = 0, 0, 0
print('Applying mean normalization and resizing images...')
for img in X_train:
    b, g, r = cv2.mean(img)[:3]
    R += r
    G += g
    B += b

R = R / len(X_train)
G = G / len(X_train)
B = B / len(X_train)

X_train_new = []
# Apply mean normalization and random cropping
from sklearn.feature_extraction.image import extract_patches_2d
for img in X_train:
    img[:,:,0] - B
    img[:,:,1] - G
    img[:,:,2] - R

    img = extract_patches_2d(img, (224, 224), max_patches=1)[0]
    X_train_new.append(np.array(img))

X_train = np.array(X_train_new)
```

The testing set was only randomly cropped, with no mean normalisation. The normalization was skipped to ensure that feature normalization wouldn't skew the accuracy of our model when tested against a testing set.

```
X_test_new = []
# Test set
for img in X_test:
    img[:,:,0] - B
    img[:,:,1] - G
    img[:,:,2] - R

    img = cv2.resize(img, (224, 224))
    X_test_new.append(np.array(img))

X_test = np.array(X_test_new)
```

Our Train-Test Split

```
# Split the dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    imgs_array, labels,
    test_size=0.2,
    stratify=labels
)
```

As seen above, we opted for an 80/20 Train-Test split for our dataset. The *imgs_array* in the above screenshot is an array of all the resized images from the original dataset.

Furthermore, the test dataset is again split into two more datasets by the following code:

```
# Split validation and test sets
(X_val, X_test, y_val, y_test) = train_test_split(
    X_test, y_test,
    test_size=0.5,
    stratify=y_test
)
```

This gives us separate datasets for the final testing and continuous validation after each epoch.

The above approach ensures that the final test is performed using truly unseen values by the model, as the testing dataset is only ever used for the final model accuracy testing.

At the end of all these steps, the final images and labels were saved locally on the machine to ensure that these processing steps would not have to be redone every time we tried to train our networks, or simply run our model.

```
print('Saving datasets...')
# Save the preprocessed dataset
np.save('/content/drive/MyDrive/DataSet/archive/X_train', X_train)
np.save('/content/drive/MyDrive/DataSet/archive/y_train', y_train)
np.save('/content/drive/MyDrive/DataSet/archive/X_val', X_val)
np.save('/content/drive/MyDrive/DataSet/archive/y_val', y_val)
np.save('/content/drive/MyDrive/DataSet/archive/X_test', X_test)
np.save('/content/drive/MyDrive/DataSet/archive/y_test', y_test)
print('Datasets saved!')
```

Machine Configuration

Our entire neural network was trained on a free Google Colab Instance. This allowed the team members to work effectively to modify and build the project.

Specifications

Python 3 Google Compute Engine Backend.
Free T4 GPU Runtime Instance
System RAM: 12.7GB
GPU VRAM: 15GB
Disk Space: 78.2GB

Additional Details

Our first approach uses transfer learning. The pre-trained feature extractor, in that case, is a VGG-16 model with the weights from the ImageNet Dataset. It gives us decent accuracy with much less computational expense for training. However, it doesn't bias our accuracy as the pre-trained feature extractor has a much broader scope than vehicle classification. So, it doesn't bring any bias objectively.

Both models were trained for a maximum number of 50 epochs. The initial learning rate for both the models we trained was set at 0.01 with a momentum of 0.9.

Results

Metrics Used

For assessing the quality of our models, we used the following metrics:

- **Accuracy:** This refers to the accuracy of the network against the training dataset. This was continuously evaluated during training. Training over multiple epochs resulted in a definite increase in the value of this metric.
- **Validation Accuracy:** The value of this metric is calculated at the end of every epoch. This metric is obtained upon validating the model obtained at the end of each epoch against the validation set. The values of this metric can be used to determine if overfitting or underfitting occurs at a particular point during training.
- **Testing Accuracy:** This metric is calculated after training has concluded. The model is tested against the separate testing dataset to get a final accuracy value. We use the value of this metric to determine the overall performance of our final obtained model in our project.

Results for Approach 1: Transfer Learning Approach

Our final test accuracy was **74%.**

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer.
26/26 [==============================] - 15s 190ms/step - loss: 4394.7334 - accuracy: 0.0074
Model accuracy is 0.74%
```

The screenshot above shows a value of 0.0074. This discrepancy is due to a change in the TensorFlow library we used. The floating point representations were modified to save memory when training and testing models.

Results for Approach 2: Training a VGG-16 Neural Network from Scratch

Our final test accuracy was **86%.**

```
26/26 [==============================] - 544s 21s/step - loss: 5.2727 - accuracy: 0.0086
Model accuracy is 0.86%
```

The screenshot above shows a value of 0.0086. This discrepancy is due to a change in the TensorFlow library we used. The floating point representations were modified to save memory when training and testing models.

Ablation Studies

For Approach 1: Transfer Learning Approach

We can remove specific sections of the pre-trained feature extractor to get an idea of the features each layer extracts. This gives us a comprehensive overview of the network's knowledge representation and could help us understand what additional changes can be made to the layers to improve the accuracy of the network obtained from this approach. It can also help us assess the need for more convolutional layers after the pre-trained layers to improve the accuracy of our neural network further despite the cost in performance. We can also identify certain pre-trained layers that can be modified or removed without majorly impacting the accuracy of our model while making it more lightweight and faster to train.

For Approach 2: Training a VGG-16 Neural Network from Scratch

Similar to the study for Approach 1 above, we can understand what features our network extracts when trained explicitly on our dataset by removing specific sections from our network. An analysis of the same can help us assess if we can reduce the overall number of layers to perform the classification task without significant loss in accuracy. This would help us obtain an even more lightweight neural network.

The scope of such an ablation study in this approach is far greater than approach 1, as the original VGG architecture was intended to classify between 1000 different classes of the ImageNet dataset. The Stanford Cars dataset doesn't have the same level of diversity within the dataset. All our dataset's objects are similar; thus, a deeper network might be learning and using certain unnecessary features for classification. This leaves a lot of room for improvement in the architecture of the network itself, with the potential for a lightweight network to perform the classification task competently.

Discussion

The results obtained aligned with our initial assumptions going into the project. We clearly see that the model trained using Approach 2 performs far better than the Model trained using Approach 1. The following reasons may be why:

- **Regarding Training Time:** Approach 1's average training time per epoch was significantly lower than the training time for Approach 2. This was in line with our initial assumptions and is caused by the fact that Approach 2 has more learnable parameters than Approach 1.
- **Regarding Testing Accuracy Values:** The testing accuracy values obtained from the model trained using Approach 2 were better than those of the model trained using Approach 1. This is because of the following reasons:
    - In Approach 2, by training the entire network over the training dataset, the weights of the convolutional layers get updated to observe features relevant to our specific dataset instead of a broader set of features which the ImageNet pre-trained VGG-16 network can observe.
    - This reduces the border application scope of the model from Approach 2 but makes it excellent for our specific task here. Thus, our model weight from Approach 2 will not be very effective for tasks besides Vehicle classification or similar tasks.

Where the Project Fails:

While our model from Approach 2 is superior in comparison, both are fairly deep and complex. The hidden layers use many neurons, resulting in performance degradation. Currently, this model will be too slow for real-time vehicle classification.

Future Scope

This project shows a proof of concept for vehicle classification. If we can perform more effective preprocessing of our input images (From our dataset in this case) and train a model on those, we can train a model to perform much faster and almost real-time classification. Such a development will allow this model to work with a vehicle number identification model to build effective automatic traffic rule compliance systems with zero to minimal human intervention.

A system of this kind would be able to identify traffic rule defaulters, obtain the vehicle's registration number and verify it by using a classification model against the government registration database to notify the defaulter and the relevant authorities automatically.

Moreover, intuitively, by adding skip connections to the existing architecture, making it more akin to the ResNet architecture, we can improve the accuracy of such a neural network. It will also provide more effective paths for the gradients to follow and allow the weight of certain identified features to propagate further in the network for more effective classification. In our case, features like vehicle shape, headlight shape, logo shapes, body contours, etc., when propagated this way, would make classification more effective as many brands and even car models have distinct styles that can be identified without relying on extremely specific details.

Conclusion

This project shows the various ways a network following the VGG-16 architecture can be trained to perform the task of vehicle classification with relatively decent accuracy. We also highlight the various tradeoffs in following one approach over the other when training a neural network for vehicle classification using the Stanford Cars dataset.

Furthermore, the project highlights the excellent use of transfer learning techniques to prototype relatively accurate models quickly, which require lesser computational resources and training time for a slight tradeoff in accuracy. Such approaches can be extremely useful for quickly prototyping models for AI-based systems with minimal computational overhead.

Lastly, the project also highlights the effectiveness of Deep Convolutional Neural Networks in object classification tasks and gives a brief insight into how they function to perform this task. It shows how training the network in different ways can affect the types of features extracted and used for classification. This can be built upon to build networks that can perform this task with even better accuracy.

References

[1] J. Krause, J. Hamblen, and L. Berg, "Stanford cars dataset 1.0," in Proc. Stanford Artificial Intelligence Lab Technical Report, 2013.
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

Contributions of Team Members

	Name	ID	Percentage Contribution
	Kaustubh	21UCS108	*33.34%*
	Saumya Saraswat	21UCS184	*33.33%*
	Manas Vashisht	21UCS125	*33.33%*
			***Note**: x + y + z = 100%*

Appendix:

1. https://www.robots.ox.ac.uk/~vgg/research/very_deep/
2. https://keras.io/api/applications/vgg/
3. https://www.researchgate.net/figure/Transfer-learning-Transfer-learning-is-process-of-taking-pretrained-model-usually_fig1_338540456
4. https://machinelearningmastery.com/gradient-descent-with-nesterov-momentum-from-scratch/
5. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/