1 Hello Computer Vision

Computer vision is an interdisciplinary field that deals with enabling computers and systems to derive meaningful information from digital images, videos and other visual inputs, and take appropriate action or make relevant recommendations. It involves acquiring, processing, analyzing and understanding digital images and videos to produce numerical or symbolic information in order to achieve tasks such as object detection, recognition, motion analysis, scene reconstruction, and event detection. Computer vision has applications in diverse domains including robotics, autonomous vehicles, medical imaging, security and surveillance, industrial automation, and human-computer interaction. At its core, computer vision combines methods from areas like signal and image processing, pattern recognition, machine learning, geometry, physics, and other computer science fields. After reading this tutorial, you will know:

- The role of deep learning, particularly convolutional neural networks (CNNs), in revolutionizing vehicle detection and classification tasks.
- The working principles of CNN architectures for object detection, localization, and vehicle type classification.
- The concept of transfer learning and the advantages of leveraging pre-trained CNN models for vehicle detection and classification.
- Practical considerations and challenges involved in deploying real-world vehicle detection and classification systems, including datasets, annotation tools, varying conditions, and hardware requirements.

Let's get started.

1.1 Importance of Vehicle Detection and Classification

The ability to accurately detect and classify vehicles plays a crucial role in the development and implementation of Intelligent Transportation Systems (ITS). ITS aim to improve transportation efficiency, safety, and environmental sustainability by leveraging advanced technologies, including computer vision, sensor networks, and communication systems.

One of the primary applications of vehicle detection and classification is in traffic monitoring and management. By identifying and tracking vehicles on roads and highways, ITS can provide real-time traffic data, enabling better traffic flow optimization, incident detection, and congestion mitigation strategies. Accurate vehicle classification (e.g., cars, trucks, buses) further enhances these capabilities, as different vehicle types have varying impacts on traffic patterns and road infrastructure.

Moreover, vehicle detection and classification are essential for improving road safety and preventing accidents. Advanced driver assistance systems (ADAS) and autonomous vehicles rely heavily on these technologies to perceive their surroundings, identify potential hazards, and make informed decisions. By accurately detecting and classifying vehicles, pedestrians, and other obstacles, these systems can take appropriate actions to avoid collisions and ensure the safety of all road users.

Another important application is in toll collection and enforcement systems. Vehicle detection and classification enable accurate vehicle identification and toll calculation based on vehicle type, ensuring fair and efficient toll collection processes. Additionally, law enforcement agencies can utilize these technologies for traffic violation detection and enforcement, such as identifying vehicles that exceed speed limits or run red lights.

Furthermore, the data generated from vehicle detection and classification systems can be analyzed to uncover valuable insights into traffic patterns, congestion hotspots, and travel behaviors. These insights can inform transportation planning, infrastructure development, and traffic forecasting models, leading to more efficient and

sustainable transportation systems.

In the context of smart cities, vehicle detection and classification play a vital role in various applications. Smart traffic lights and signals can adapt to real-time traffic conditions by detecting and classifying approaching vehicles, optimizing signal timings and reducing congestion and emissions. Parking management systems can leverage these technologies to monitor parking availability, guide drivers to available spots, and enforce parking regulations based on vehicle types.

Environmental monitoring is another important application, as vehicle detection and classification can aid in estimating emissions and noise levels based on vehicle types and volumes, enabling cities to implement more effective environmental policies and mitigation strategies.

Beyond transportation, vehicle detection and classification have commercial and industrial applications in logistics and fleet management. By tracking and monitoring vehicles, companies can optimize routing, improve resource utilization, and enhance operational efficiency. Additionally, these technologies are essential for the development and deployment of autonomous vehicles and advanced driver assistance systems (ADAS), enabling safer and more efficient transportation solutions.

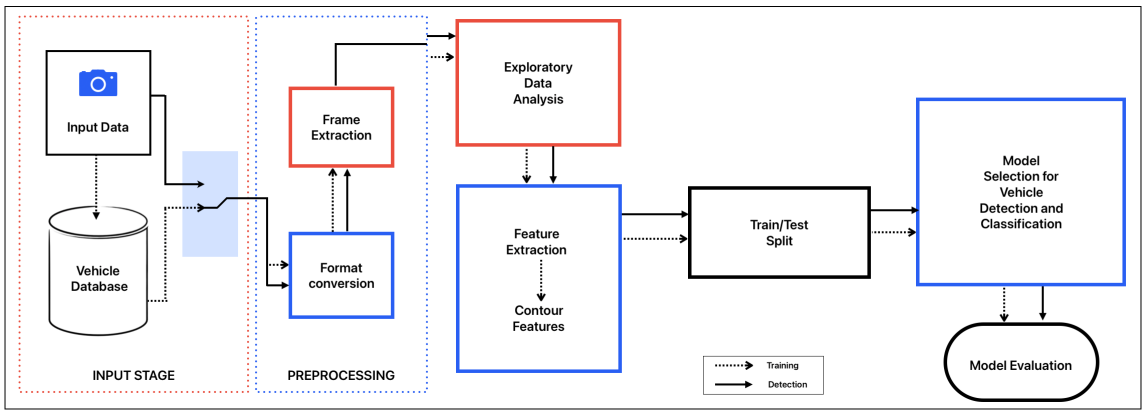


Figure 1: Diagrams of autonomous vehicles or ADAS systems, highlighting the role of vehicle detection and classification.

In the domain of security and surveillance, vehicle detection and classification can assist in identifying suspicious vehicles, tracking potential threats, and enhancing overall public safety measures.
As technology continues to evolve, the importance of vehicle detection and classification is expected to grow further. Potential future applications include integration with smart infrastructure, predictive maintenance of transportation systems, and enhanced traffic simulations for urban planning purposes. Ongoing research efforts are focused on developing more accurate and robust computer vision algorithms, leveraging advances in deep learning and multi-sensor fusion techniques.

Useful References:

- *Intelligent Transportation Systems*
- *Deep Learning Techniques for Vehicle Detection and Classification from Images/Videos: A Survey*
- *Vehicle Detection and Classification via YOLOv8 and Deep Belief Network over Aerial Image Sequences*
- *Vehicle Detection and Classification Using Convolutional Neural Networks*

- *Road traffic: Vehicle detection and classification*
- *Vehicle Detection and Classification: A Review*
- *Efficient Feature Selection and Classification for Vehicle Detection*

1.2 Challenges in Vehicle Detection and Classification

Vehicle detection and classification, while crucial for various applications, present numerous challenges that must be addressed to achieve accurate and reliable results. These challenges stem from the inherent complexities of real-world environments, the diversity of vehicle types and appearances, and the computational constraints of implementing these systems in practical scenarios.

One of the most significant challenges is dealing with varying environmental and lighting conditions. Illumination levels can fluctuate dramatically based on time of day, weather, and location, making it difficult for computer vision algorithms to consistently detect and classify vehicles. Additionally, adverse weather conditions such as rain, snow, or fog can obscure or distort visual information, leading to potential errors or missed detections. Occlusion and obstructed views further complicate the task of vehicle detection and classification. Partial occlusion, where only a portion of a vehicle is visible, can make it challenging to accurately identify and classify the vehicle type. Complete occlusion, where the vehicle is entirely blocked from view, can result in missed detections altogether. Obstructed viewpoints, such as those caused by buildings, trees, or other objects in the line of sight, can also impact the accuracy of detection and classification algorithms.

The sheer diversity of vehicle types and appearance variations presents another significant challenge. Algorithms must be capable of accurately distinguishing between different vehicle classes, such as cars, trucks, buses, and motorcycles, while also accounting for variations in make, model, color, and customizations. This diversity can lead to overlapping visual characteristics, making it difficult to establish clear boundaries for classification. Motion and viewpoint challenges further complicate the task. High vehicle speeds can result in motion blur, making it difficult to accurately detect and classify moving vehicles. Additionally, varying viewpoints and camera angles can significantly alter the appearance of vehicles, requiring algorithms to be robust to these changes. Proper camera and sensor positioning is crucial to ensure optimal performance and minimize occlusions or obstructed views.

Real-time performance and computational constraints are also important considerations, especially in embedded systems or resource-constrained environments. Vehicle detection and classification algorithms must be computationally efficient to meet real-time processing requirements, while also adhering to memory and storage limitations. Energy efficiency is particularly crucial for applications involving mobile or autonomous systems.

Data quality and availability pose another challenge. Robust vehicle detection and classification models require access to diverse and representative datasets, encompassing a wide range of vehicle types, environmental conditions, and viewpoints. However, obtaining and annotating such datasets can be time-consuming and labor-intensive. Privacy and security concerns related to collecting and using visual data from public spaces can also present legal and ethical challenges.

Finally, ensuring robustness and generalization is crucial for the successful deployment of vehicle detection and classification systems. Algorithms must be capable of handling edge cases and unusual scenarios that may not have been encountered during training. Additionally, models should be adaptable to new environments and conditions, minimizing the need for extensive retraining or fine-tuning. Transfer learning and model generalization techniques can help address these challenges, but they require careful consideration and validation.

Useful References:

- *Deep Learning Techniques for Vehicle Detection and Classification from Images/Videos: A Survey*
- *Challenges of Video-Based Vehicle Detection and Tracking in Intelligent Transportation Systems*
- *Vehicle Detection and Classification via YOLOv8 and Deep Belief Network over Aerial Image Sequences*
- *Challenges of Video-Based Vehicle Detection and Tracking in Intelligent Transportation Systems*
- *What are the challenges in developing an effective vehicle detection system for military surveillance?*
- *Parallel Implementation of a Video-based Vehicle Speed Measurement System for Municipal Roadways*

1.3 Deep Learning for Vehicle Detection and Classification: An Overview

The field of deep learning has revolutionized the way we approach complex problems in computer vision, including vehicle detection and classification. At the heart of this revolution lies a powerful class of artificial neural networks known as Convolutional Neural Networks (CNNs). These networks, inspired by the biological visual cortex, have proven to be remarkably effective in extracting and learning intricate patterns and features from visual data.

The story of deep learning's rise to prominence is one of perseverance and breakthrough discoveries. Although the foundations of artificial neural networks date back to the 1950s, it wasn't until the early 2000s that researchers began to unlock the true potential of deep learning with the advent of powerful computing resources and large-scale datasets. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) played a pivotal role in this transformation, serving as a catalyst for the development and refinement of deep learning techniques for image classification and object detection.

Convolutional Neural Networks, a specialized type of deep neural network, have emerged as the workhorse of computer vision tasks. Their unique architecture, comprising layers of convolution and pooling operations, allows them to efficiently capture and learn hierarchical features from raw pixel data. The convolution operation performs feature extraction by sliding learnable filters over the input image, while pooling operations downsample the feature maps, introducing translation invariance and reducing computational complexity. Over the years, researchers have proposed numerous CNN architectures, each building upon the successes and limitations of its predecessors. Pioneering models such as AlexNet, VGGNet, and GoogLeNet paved the way for more advanced architectures like ResNet, Inception, and DenseNet, which addressed challenges like vanishing gradients and computational efficiency through innovative techniques like skip connections and dense connectivity.

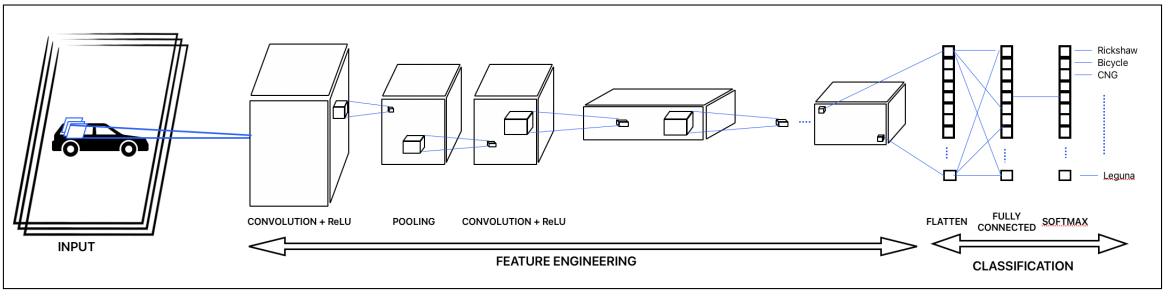


Figure 2: Diagrams illustrating the architecture and building blocks of Convolutional Neural Networks (CNNs)

Object detection, a crucial task in computer vision and a prerequisite for vehicle detection and classification, has also been transformed by deep learning. Early region-based methods like R-CNN and Fast R-CNN employed a two-stage approach, first proposing potential object regions and then classifying each region. Faster R-CNN

streamlined this process by incorporating a Region Proposal Network (RPN) to predict region proposals directly from the feature maps. However, the computational overhead of these methods led to the development of single-shot detectors like YOLO and SSD, which perform object detection in a single forward pass, making them more efficient for real-time applications. Adapting these powerful object detection techniques to the specific domain of vehicle detection and classification presents its own set of challenges and considerations. While CNNs excel at learning discriminative features, their performance can be impacted by factors such as occlusion, varying viewpoints, and the diversity of vehicle types and appearances. Researchers have explored various strategies to address these challenges, including tailoring CNN architectures for vehicle-specific tasks, leveraging transfer learning and fine-tuning techniques, and incorporating multi-task learning and multi-sensor fusion approaches.

To train robust vehicle detection and classification models, access to high-quality datasets is essential. Publicly available datasets such as KITTI, BDD100K, and Cityscapes have played a crucial role in advancing research in this area. However, the diversity and representativeness of these datasets can still be improved, prompting researchers to explore data augmentation techniques, such as geometric transformations, synthetic data generation, and domain adaptation methods. The training process for deep learning models involves carefully crafted loss functions and evaluation metrics tailored to the specific task at hand. For object detection, metrics like mean Average Precision (mAP) and Intersection over Union (IoU) are commonly used to assess model performance. Optimization algorithms like Stochastic Gradient Descent (SGD) and its variants, combined with effective hyperparameter tuning strategies, are employed to navigate the complex loss landscapes and converge to optimal model parameters.

As deep learning models continue to grow in size and complexity, hardware considerations become increasingly important. The widespread adoption of Graphics Processing Units (GPUs) and the emergence of Tensor Processing Units (TPUs) have facilitated the training and deployment of large-scale deep learning models, enabling researchers and practitioners to push the boundaries of what is achievable in vehicle detection and classification.

Once trained, deep learning models must be deployed and integrated into real-world systems and applications. Model compression and optimization techniques, such as quantization and pruning, can help reduce the memory footprint and computational requirements of these models, enabling efficient deployment on edge devices and embedded systems. Real-time inference is crucial for applications like autonomous driving and advanced driver assistance systems (ADAS), where split-second decisions can mean the difference between safety and potential accidents.

As the field of deep learning for vehicle detection and classification continues to evolve, researchers are exploring a multitude of exciting directions and emerging trends. Advanced CNN architectures and techniques, such as attention mechanisms, capsule networks, and transformer-based models, hold promise for improving performance and addressing long-standing challenges. Multi-task learning and multi-sensor fusion approaches aim to leverage complementary information from various sources, such as cameras, LiDAR, and radar, to enhance the robustness and reliability of vehicle detection and classification systems. The pursuit of Explainable AI (XAI) and model interpretability has become increasingly important, particularly in safety-critical applications like autonomous vehicles. Developing deep learning models that can provide transparent and human-understandable explanations for their decisions is crucial for building trust and enabling effective debugging and improvement.

Useful References:

- *Deep Learning for Object Detection: A Comprehensive Review*
- *Object Detection with Deep Learning: A Review*
- *Deep Learning for Generic Object Detection: A Survey*
- *ImageNet Large Scale Visual Recognition Challenge, 2015.*
- *ImageNet Classification with Deep Convolutional Neural Networks, 2012.*

- *Object Detection with Deep Learning: A Review, 2018.*
- *A Survey of Modern Object Detection Literature using Deep Learning, 2018.*
- *Deep Learning for Generic Object Detection: A Survey, 2018.*

2 Deep Learning

Deep learning, a subset of machine learning, is a powerful technique that extracts intricate patterns from vast amounts of data using artificial neural networks – sophisticated computing systems designed to mimic the human brain's ability to analyze and process information. These artificial neural networks form the foundation of artificial intelligence, tackling complex problems that would be impractical or impossible to solve through traditional human or statistical methods. In contrast to traditional machine learning algorithms, which rely on manually defined and engineered features, deep learning takes a more intelligent approach. It learns these features directly from the data itself, eliminating the time-consuming and often brittle process of hand-engineering features. This scalable and adaptable approach has revolutionized the way we approach complex data analysis tasks. At the heart of deep learning lies a multi-layered structure of interconnected neurons, each layer building upon the insights gleaned from the previous one. This architecture draws inspiration from the pioneering work of Walter Pitts and Warren McCulloch, who, in 1943, created one of the earliest computer models based on the neural networks of the human brain. Combining algorithms and mathematics, they developed a concept called "threshold logic," laying the foundation for mimicking the thought process through computational means. Deep learning's ability to automatically extract relevant features from raw data has propelled its adoption across diverse domains, from computer vision and natural language processing to speech recognition and drug discovery. By leveraging the power of artificial neural networks, deep learning algorithms can uncover intricate patterns and relationships that would otherwise remain hidden, unlocking unprecedented insights and driving innovation in ways previously unimaginable.

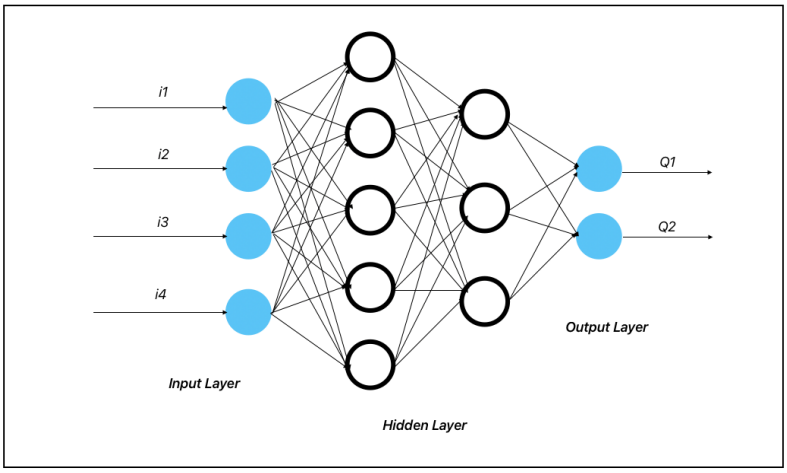


Figure 3: A visual representation of the multi-layered architecture of a deep neural network, showing the input layer, hidden layer's, and output layer. This diagram can help illustrate the flow of information and computations through the network.

At the core of deep learning lies a sophisticated architecture consisting of three distinct layers: the input layer, the hidden layer's, and the output layer. This multi-layered structure forms the backbone of the learning process, enabling the neural network to extract insights from raw data and deliver meaningful results. The input layer acts as the gateway, accepting the initial features or data points from the external world. However, it performs no computations; its sole purpose is to pass this information seamlessly to the subsequent hidden layer's. The

hidden layer's constitute the heart of the deep learning model, where the real magic happens. Concealed from the outside world, these layers perform intricate computations and transformations on the input features, leveraging the power of artificial neurons and their interconnections. It is within these hidden layers that the network learns to recognize patterns, extract meaningful representations, and uncover complex relationships buried deep within the data. Finally, the output layer serves as the interface between the neural network and the external world, presenting the learned insights and predictions. This layer takes the distilled knowledge accumulated through the hidden layers and translates it into a form that can be interpreted and utilized by humans or other systems. The true strength of deep learning lies in its ability to automatically discover and learn these intricate representations and patterns from raw data, without the need for explicit feature engineering. Through this multi-layered architecture, deep learning models can adapt and evolve, continuously refining their understanding and improving their performance as more data is processed. This self-learning capability has revolutionized various domains, enabling breakthroughs in areas ranging from image recognition and natural language processing to autonomous systems and predictive analytics.

A single Neuron/Perceptron (Forward Propagation) is the basic structural building block of Deep Learning. The output of a single Neuron is $\hat{y} = g(w_0 + X^T W)$ that's shown in Eq. (1), where X and W are given by:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} X_i \cdot w_i) = g(w_0 + X^T W) \tag{1}$$

The activation function defines the output of a neuron/perceptron given an input or set of inputs (output of multiple neurons). The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

1. Linear Activation: No transformation occurs between the input and output. Therefore, the output is the same as the input.
2. Non-Linear Activation: Non-linear activations increase the functional capacity of the neural network because they allow it to represent non-linear relationships between features in the input.
   (a) Sigmoid Function: $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
   (b) Hyperbolic Tangent: $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
   (c) Rectified Linear Unit (ReLU): $g(z) = \max(0, z) = \begin{cases} 0 & \text{if } z \text{ is negative} \\ z & \text{if } z \text{ is positive} \end{cases}$

Loss functions play an important role in any machine learning model. It measure how far an estimated value is from its true value. Loss is the penalty for misclassification of a learning model. It's a number indicating how bad the model's prediction was on a single instance. If the model's prediction is correct the loss is zero; otherwise, the loss is greater. The goal of a learning model is to find a set of weights and biases that have low/minimum loss, on average, across all instances.
Empirical Risk Minimisation (ERM) is a fundamental concept in machine learning that we can't know exactly how well the learning algorithm will work in real-life (the true risk), as testing learning algorithm on new real-world data is very expensive and costly process (known as goal standard). Because, we don't know the true distribution of data that the algorithm will work on, but we can instead measure its performance on a known set of training data (the "empirical" risk). The empirical loss measures the total loss over the entire dataset.

$$\frac{1}{n}\sum_{i=1}^{n} L(Predicted, \hat{y}^{(i)}, Actual, y^{(i)}) \tag{2}$$

2.1 Convolutional Neural Networks

Convolutional neural networks (CNNs), a subtype of deep learning models, are frequently used for image processing tasks including segmentation, object detection, and classification. They have also been applied in a number of fields, including as natural language processing and medical imaging. Since they excel in identifying spatial hierarchies and patterns in data, CNNs are well suited for applications that use grids, such as image processing.

A CNN is made up of several layers, each of which serves a particular purpose. Let's mathematically analyze the fundamental components and explain how each of them functions:

1. Input Layer (I): In an image classification task, the input is a 3D tensor representing an image. Suppose the input image has dimensions width (W), height (H), and channels (C, typically 3 for RGB images). The input layer can be represented as-

$$I \in R \wedge (W \times H \times C)$$

2. Convolutional Layer (Conv): The convolution operation is the heart of a CNN. It involves applying a set of learnable filters (kernels) to the input image. Each filter has a small receptive field and slides across the input to produce feature maps. Mathematically, the convolution operation can be written as:

$$y_i, j = b + \sum_{m=0}^{F-1}\sum_{n=0}^{F-1} x_i + m, j +_n w_{m,n}$$

where $y_{i,j}$ is the output value at position (i,j), b is the bias term, $x_i + m, j +_n$ is the input value at position (i+m,j+n), $w_{m,n}$ is the filter value at position (m,n), and F is the filter size. This equation assumes that the input and the filter have the same number of channels and that the stride and padding are both equal to 1.

3. Activation Function (e.g., ReLU):A convolutional layer or a fully linked layer's output is transformed into a non-linear form using an activation function. Typically, it is applied to each value in the output feature map element-by-element. The neural network's performance, capacity for learning, and the kinds of predictions it is capable of making may all be influenced by the activation function. Many other activation function types exist, including sigmoid, tanh, ReLU, Leaky ReLU, softmax, etc. The job and the data determine which activation function should be used, and each activation function has advantages and disadvantages of its own.
For example, Rectified Linear Unit (ReLU) is applied element-wise to introduce non-linearity:

$$A_{(i,j)} = max(0, F_{(i,j)})$$

4. Pooling Layer (e.g., Max Pooling): A pooling layer is a type of layer in a convolutional neural network (CNN) that reduces the spatial dimensions of the input feature maps while preserving the depth (i.e., the number of channels). The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions, and applying a pooling operation to each region. The pooling operation can be either max pooling or average pooling, which summarize the maximum or the average value of the elements in each region, respectively. The output of the pooling layer is a new feature map that has a smaller size but retains the most important features from the input. Pooling layers are used to improve the performance and efficiency of CNNs by reducing the number of parameters and computations, as well as to introduce some translation invariance to the features.

   Max pooling is a common technique:

$$P_{(i,j)} = max(A_{(2i,2j)}, A_{(2i,2j+1)}, A_{(2i+1,2j)}, A_{(2i+1,2j+1)})$$

5. Fully Connected Layer (FC): A fully connected layer in a convolutional neural network (CNN) is a layer that connects every unit in the input feature map to every unit in the output feature map. A fully connected layer performs a linear transformation followed by an optional activation function. A fully connected layer can be used for various purposes, such as classification, regression, or dimensionality reduction. This layer can be seen as a special case of a convolutional layer, where the filter size is equal to the input size and the number of filters is equal to the output size. This means that a fully connected layer can be replaced by a convolutional layer with the same parameters. However, a fully connected layer usually has more parameters and computations than a convolutional layer, as it does not exploit the spatial structure of the input.
   After several convolutional and pooling layers, fully connected layers are used to make predictions. These layers are essentially densely connected neural networks:

   $$O = \sigma(W_x + b)$$

   where O represents the output vector, W is the weight matrix, x is the input vector, b is the bias vector, and σ is an activation function.

6. Output Layer: The output layer function of a CNN is to produce the final output of the network based on the features extracted by the previous layers. The output layer is usually a fully connected layer that connects every unit in the input feature map to every unit in the output feature map. The output layer can have different numbers and types of units depending on the task and the data. For example, for image classification, the output layer can have as many units as the number of classes, and use a softmax activation function to produce a probability distribution over the classes.

2.2 Transfer Learning

Transfer learning is a powerful technique that has revolutionized the realm of deep learning. It involves taking an existing model, trained for one task, and adapting it for a new, related task. This ingenious approach allows us to harness the knowledge and patterns the model has already learned, providing a significant head start. Imagine training a simple model to recognize backpacks in images. Through transfer learning, we can take this model's understanding and apply it to recognizing other objects like sunglasses or hats. It's akin to building upon a solid foundation, rather than starting anew from the ground up.

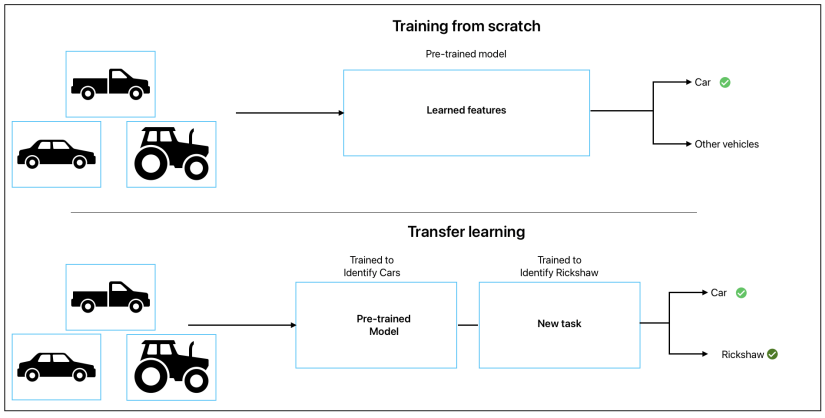


Figure 4: Utilizing Transfer Learning for Vehicle Detection and Classification

The true brilliance of transfer learning lies in its ability to overcome the inherent challenges of deep learning, where extracting patterns from vast amounts of data can be a formidable undertaking. By leveraging pre-trained models

that have already learned from massive datasets, transfer learning offers a potent solution, especially when labeled training data is scarce. This technique has found widespread applications in computer vision and natural language processing, revolutionizing fields like image recognition, object detection, and language understanding. Its advantages are manifold: not only does it save precious training time, but it also enhances the performance and accuracy of neural networks, even with limited data. Transfer learning has opened new vistas for innovation, enabling researchers and developers to tackle increasingly complex problems by standing on the shoulders of giants – harnessing the knowledge and insights gained from previous endeavors. It has become an indispensable tool in the deep learning arsenal, unlocking new possibilities and driving progress in ways once thought unattainable.

Table 1: Model Overview

No	Model	Core Model	Architecture	Input Matrix	Year
1	AlexNet	CNN	5-Conv, 3-FC, 3 Max-pool Layers	(256, 256, 3)	2012
2	GoogleNet	CNN+Inception	59-Conv, 9-Inception, 1-FC, 5 Max-pool Layers, Avg-pool Layers	(224, 224, 3)	2014
3	ResNet-50	CNN+Residual Connection	48-Conv, 1-Max-pool Layer, 1-Avg-pool Layers	(224, 224, 3)	2015
4	VGG-19	CNN	19-Conv, 3-FC, 5 Max-pool Layer	(224, 224, 3)	2014
5	Xception	CNN (inspired by inception)	36-Conv, FC layer, Max-pool Layer, Global Avg-pool Layers	(299, 299, 3)	2016
6	InceptionV3	CNN+Inception	86-Conv, 11-Inception, 3-FC, Max-pool Layer, Global Avg-pool Layers	(256, 256, 3)	2015
7	DenseNet-121	Dense CNN	103-Conv, 4-Transition layers, 1-Global Avg-pool Layers	(224, 224, 3)	2016
8	SqueezeNet	Compact CNN	8-Fire Module, 2-Conv Layers	(224, 224, 3)	2016
9	NASNetMobile	CNN	36-Conv, 4-Transition Layers, FC Layers, Global Avg Pool Layers	(224, 224, 3)	2018
10	MobileNetV2	Region-based CNN	86-Conv, Residual Blocks, Global Avg-pool Layer	(224, 224, 3)	2018

2.3 AlexNet

AlexNet is a convolutional neural network (CNN) architecture created in 2012 by Alex Krizhevsky and coworkers. It won the ImageNet Large Scale Visual Recognition Challenge with a top-5 error of 15.3%, which was significantly lower than the previous best-in-class models. It has eight layers, 5 of which are convolutional and 3 of which are

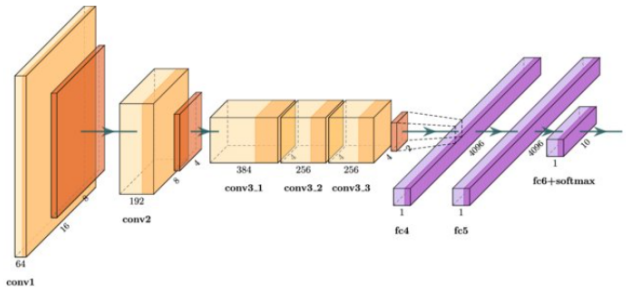


Figure 5: Architecture of AlexNet

completely linked. In order to decrease memory use and speed up training, the network is divided into two parallel streams, each executing on a single GPU.

1. Convolutional Layers: AlexNet's convolutional layer is a layer that convolutions the input data, which is a linear combination of the input and a filter (or kernel) matrix. A convolutional layer may extract characteristics like as edges, shapes, textures, and so on from input data. There are five convolutional layers in AlexNet, and each layer has a distinct number of filters, padding, strides, and filter sizes. The 1st convolutional layer has 96 11x11 filters with a stride of 4 and no padding, resulting in a 55*55*96 output feature map. The 2nd convolutional layer has 256 5*5 filters with a stride of 1 and a padding of 2, resulting in a 27*27*256 output feature map. The 3rd convolutional layer has 384 3*3 filters with a stride of 1 and a padding of 1, resulting in a 13*13*384 output feature map. The 4th convolutional layer has 256 3*3 filters with a stride of 1 and a padding of 1, resulting in a 13*13*256 output feature map. The 5th convolutional layer has 256 3*3 filters

with a stride of 1 and a padding of 1, resulting in a 13*13*256 output feature map.

$$y = f(W \times x + b)$$

$x \rightarrow input$
$y \rightarrow output$
$W \rightarrow$ Weight Matrix
$b \rightarrow$ bias vector
$f \rightarrow$ Activation Function

2. Pooling Layers: There are three pooling layers in AlexNet, each with a unique filter size and stride. Using max pooling with a filter size of 3*3 and a stride of 2, the first pooling layer is applied after the first convolutional layer, producing an output feature map with the dimensions 27*27*96. Following the second convolutional layer, the second pooling layer is performed, using max pooling with the same parameters to produce an output feature map with dimensions of 13*13*256. Following the fifth convolutional layer, the third pooling layer is used, producing an output feature map with the dimensions 6*6*2563 using max pooling with a filter size of 3*3 and a stride of 2.

$$y = max(x)$$

3. Normalization layers: Normalization layers of AlexNet are layers that perform a normalization operation on the output of a convolutional layer, which is a feature map with multiple channels. Normalization layers aim to reduce the internal covariate shift, which is the change in the distribution of layer activations due to the change in network parameters during training. Normalization layers can also enhance the generalization ability of the network, prevent overfitting, and speed up the convergence.
AlexNet uses local response normalization (LRN) layers, which are a type of divisive normalization. LRN layers normalize the activations across nearby channels at the same spatial location. LRN layers apply the following formula to each activation -
$x_{i,j,k}$ where,
$i \rightarrow$ batch index
$j \rightarrow$ channel index
$k \rightarrow$ spatial index

$$y_{i,j,k} = \frac{x_{i,j,k}}{(k + \alpha \sum_{l=max(0,j-\frac{n}{2})}^{min=(N-1,j+\frac{n}{2})} x_{i,l,k}^2)}$$

where,
$N \rightarrow$ total number of channel
$n \rightarrow$ size of the local region
$k \rightarrow constant$
$\alpha \rightarrow$ scaling parameter(e.g., 0.0001)
$\beta \rightarrow$ exponent parameter(e.g., 0.75)

AlexNet has two LRN layers, each applied after the first and second convolutional layers, respectively. The LRN layers have a filter size of 5, a scaling parameter of 0.0001, an exponent parameter of 0.75, and a constant of 2.

4. Fully Connected Layers: AlexNet has three fully connected layers, each with different numbers of neurons. The first fully connected layer has 4096 neurons, the second one has 4096 neurons as well, and the third one has 1000 neurons, corresponding to the number of classes in ImageNet. The output of the last fully connected layer is passed through a softmax function, which produces the final prediction of the network.

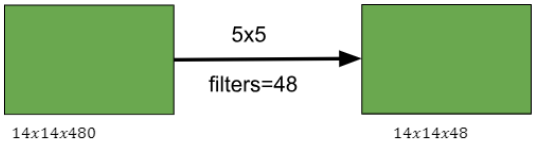
2.4 GoogleNet

Google researchers, with the support of numerous institutions, debuted Google Net (also known as Inception V1) in the research publication "Going Deeper with Convolutions" published in 2014. This architecture took first place in the 2014 ILSVRC image categorization competition. It has provided a notable reduction in error rate when compared to prior winners AlexNet (ILSVRC 2012 Winner), ZF-Net (ILSVRC 2013 Winner), and VGG (ILSVRC 2014 Runner Up). In the midst of the architecture, this design uses techniques like global average pooling and 1-1 convolutions

1. Features of GoogleNet: GoogleNet uses a variety of approaches, including global average pooling and 1X1 convolution, in order to create deeper architecture.For example:

   - 1x1 convolution: One-to-one convolution is a feature of the inception architecture. The architecture's weights and biases were reduced in number by these convolutions. Reducing the parameters also deepens the architecture. Here is an illustration of a 1:1 convolution —

     (a) If we want to perform 6×6 convolution having 54 filters without using 1×1 convolution as intermediate:



     → Total Number of operations:(14 x 14 x 48) x (5 x 5 x 480) = 112.9 M

     (b) With 1x1 convolution:

$$(14 \times 14 \times 16) \times (1 \times 1 \times 480) + (14 \times 14 \times 48) \times (5 \times 5 \times 16) = 1.5M + 3.8M = 5.3M$$

     Which is smaller than 112.9M.

2. Global Average Pooling: Prior designs, like AlexNet, used the entirely connected layers at the network's edge. The majority of the parameters in many designs, which increase computation costs, are located in these entirely connected layers.
   In the GooLeNet design, a method called global average pooling is used at the network's edge. A 7x7 feature map is averaged down to a 1x1 size in this layer. This also lowers the number of trainable parameters to 0 and improves the top-1's accuracy by 0.6

3. Inception Module: The inception module is distinct from older systems like AlexNet and ZF-Net. In this architecture, a convolution size is allocated to each layer.The final output is created in the Inception module by stacking the outputs of the parallel 1x1, 3x3, 5x5, and 3x3 max pooling operations that were performed at the input. Theoretically, convolution filters of different sizes will be better able to handle objects of different scales.

4. Auxiliary Classifier for Training: There are a few intermediate classifier branches scattered throughout the Inception architecture that are only used during training. These branches comprise a softmax classification layer, two layers with 1024 outputs each and complete connections between them, a layer with 55 average

pooling and a stride of 3, a layer with 11 convolutions and 128 filters, and two layers with 1000 outputs each. The loss generated by these layers weighed in at 0.3 of the total loss. These layers assist in addressing the gradient vanishing issue in addition to regularization.
The layers of Google Net are described in depth below-

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 6: GoogleNet Model

The overall architecture consists of 22 levels. The architecture was developed with computing efficiency in consideration. The architecture is made to function on solitary devices with constrained computing power. The architecture connects the outputs of the Inception (4a) and Inception (4d) layers to two further classifier layers.

Here are the architectural details of auxiliary classifiers:

(a) A typical pooling layer with a 5x5 filter size and stride 3.

(b) 1x1 convolution with 128 filters for dimension reduction and ReLU activation.

(c) A fully linked layer with 1025 outputs and ReLU activation

(d) Dropout Regularization with a ratio of 0.7 dropouts

(e) A 1000-class softmax classifier that produces results comparable to the primary softmax classifier.

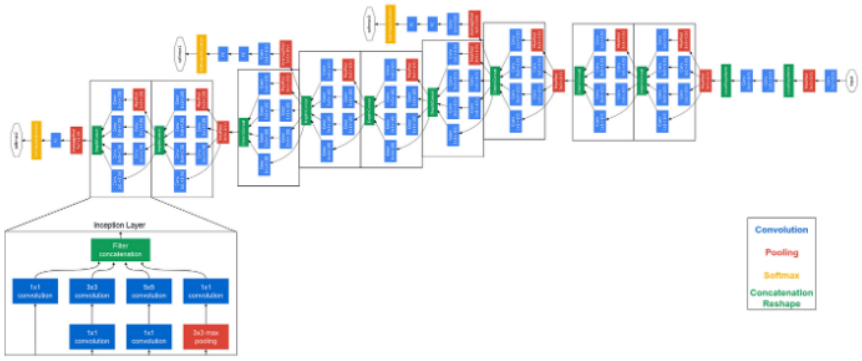


Figure 7: Architecture of GoogleNet

2.5 ResNet-50

The deep convolutional neural network architecture ResNet-50, also known as "Residual Network-50," was first introduced in the paper titled "Deep Residual Learning for Image Recognition" by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, which took home the Best Paper Award at the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Several computer vision applications, most notably image classification, typically employ ResNet-50, a variant of the original ResNet architecture.

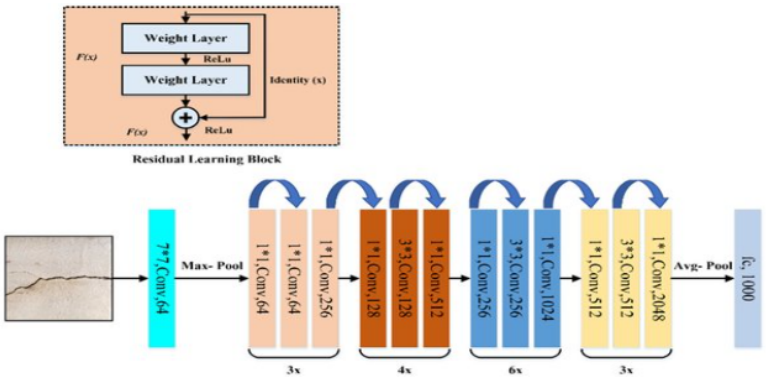


Figure 8: Architecture of ResNet-50

The usage of residual connections, commonly referred to as skip connections or shortcut connections, is the main innovation in the ResNet architecture. By overcoming the vanishing gradient problem, which can make it difficult to train deep networks without residual connections, these connections allow for the development of extremely deep neural networks. ***The ResNet-50 architecture's main characteristics and elements are listed below***—

1. Depth: ResNet-50 has 50 layers and is a deep network. Each of these blocks or stages, which make up these layers, has a distinct amount of residual units. In a nutshell, the architecture is as follows:

   (a) The first layer of convolution is the max-pooling layer.

   (b) There are four phases, each with a number of residual units.

   (c) A pooling layer with a global average.

   (d) A categorization layer that is completely linked.

2. Residual Units: ResNet-50 has numerous residual units for each level. Two or three convolutional layers, batch normalization, and ReLU activation functions constitute a residual unit. The important concept is that residual units learn the residual (difference) between the input and the desired output rather than the desired mapping directly. Before passing through the subsequent unit, the output of one residual unit is added to the input (shortcut connection).

3. Bottleneck Architecture: ResNet-50's residual units have a bottleneck design, which helps to decrease computational costs while retaining excellent performance. A typical ResNet-50 residual unit has the following structure: 1x1 convolution, 3x3 convolution, and another 1x1 convolution.

4. Global Average Pooling: ResNet-50 leverages global average pooling rather than fully connected layers with a high number of parameters for final classification. This method computes the average of each feature map's overall spatial dimensions, providing a tiny fixed-size tensor that may be used directly for classification.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 9: ResNet-50 Model

5. Output Layer: The number of neurons in the final layer, which is completely linked, corresponds to the number of classes in the classification work. Class probabilities are obtained by using a softmax activation function.

ResNet-50 has proven that it can perform exceptionally well in image classification benchmarks, including the ImageNet Large Scale Visual Recognition Challenge. Widespread use and adaption of this approach have been observed in the areas of object recognition, semantic segmentation, and other computer vision applications. Pre-trained ResNet-50 models are often utilized by researchers and professionals as a starting point for a number of computer vision applications because to their effectiveness and efficiency.

2.6 VGG-19

University of Oxford researchers Simonyan and Zisserman created the VGG-19 model. It has 19 layers (16 connected, 3 fully linked), strictly applies 33 filters with stride and pad of 1, and has 22 max-pooling layers with stride 2. When compared to AlexNet, the VGG-19 is a CNN that has more layers and is more complicated. In order to reduce the number of parameters in such deep networks, it makes the best use of its modest 33 filters and 7.3% error rate over all convolutional layers. The VGG Net article is still one of the most significant ones even if the VGG-19 model did not win the top prize at ILSVRC 2014 since it reinforced the notion that CNNs need to have a significant number of layers in order for this hierarchical representation of visual input to work.

In the ILSVRC 2014, the VGG-19 model, which includes 138M total parameters, placed second for classification

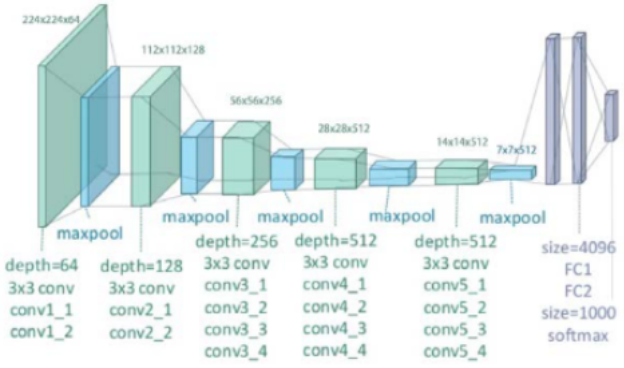


Figure 10: Architecture of VGG-19

and first for localization. A subset of the ImageNet database, on which this model was trained, is used in the

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The VGG-19 has 1,000 categories in which it can classify images, including keyboard, mouse, pencil, and several animal categories. More than a million photos were used for training. Consequently, the model has generated rich feature representations for a range of images.

Description:

- VGG-19 is a deep neural network architecture known for its simplicity and ease of implementation.
- The key characteristic of VGG is the use of small 3x3 convolutional filters with a stride of 1, which enables it to learn fine-grained features.
- Max-pooling layers are used to reduce spatial dimensions while increasing the receptive field of the network.
- The fully connected layers at the end of the network serve as a classifier. In the original paper, VGG-19 was trained for image classification on the ImageNet dataset, which contains 1.2 million training images and 1000 categories.
- VGG-19 has a total of 19 weight layers (16 convolutional layers and 3 fully connected layers), hence the name "VGG-19."
- While VGG-19 performed well on various image classification tasks, it is considered relatively deep for its time, and training it from scratch on large datasets can be computationally expensive.
- It is common to use pre-trained VGG-19 models for transfer learning, where the network's weights are fine-tuned on specific tasks or datasets.

VGG-19 and its variants have had a significant impact on the development of deep learning architectures, serving as a basis for more advanced models while also being useful for practical applications in computer vision.

2.7 Xception

François Chollet introduced the deep convolutional neural network architecture known as Xception in 2016. It is an adaptation of the Inception architecture that utilizes depthwise separable convolutions to lower the network's parameter count and increase network effectiveness.
The convolutional layers of the Xception architecture are followed by batch normalization and activation processes. The usage of depthwise separable convolutions, which are made up of two distinct convolutions: a depthwise convolution and a pointwise convolution, is the main invention of Xception. The pointwise convolution applies a 1x1 filter to combine the output of the depthwise convolution whereas the depthwise convolution applies a single filter to each input channel. Comparatively to conventional convolutions, this method minimizes the number of parameters and computing complexity.
It has been demonstrated that the Xception model outperforms the Inception model on a number of picture classification tasks, including ImageNet2, CIFAR-103, and Pneumonia Detection. It is also commonly used for various purposes, including deep fake detection, face swapping, and semantic segmentation.

2.7.1 Mathematical Representation

Let's denote the input image to Xception architecture as X, which has dimension H $\times$ W $\times$ C, where H represents the height, W represents the width, and C represents the number of channels. The architecture consists of a series of convolutional layers and non-linear activations

1. Entry Flow: The input image X is initially given a conventional convolutional layer. This layer's output, designated A_1, has the dimensions $H \times W \times F_1$, where F_1 is the number of filters in this layer.

   Then, we apply a set of depthwise separable convolutions, which are made up of depthwise and pointwise convolutions in succession. The dimensions of A_i, which stands for the output of each depthwise separable convolution, are H, W, and F_i. In the entrance flow, this procedure is performed a total of N times.

2. Middle Flow: The entry flow's properties are further refined in the middle flow, which is made up of a stack of identical modules. Similar to the entry flow, each module is made up of a number of depthwise separable convolutions. Each module's output is combined with its input in order to maintain the gradient signal during training.

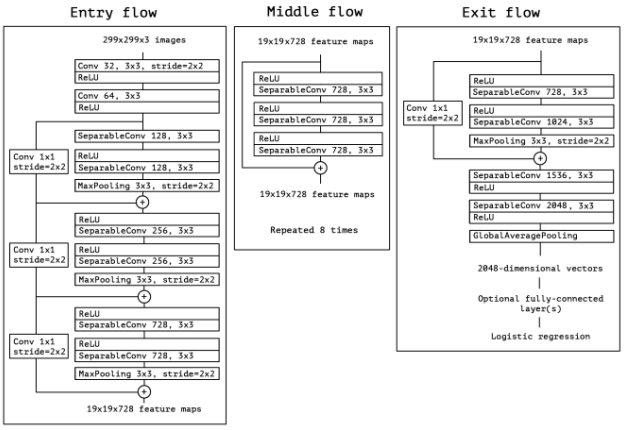


Figure 11: Architecture of Xception

3. Exit Flow: The Xception architecture's last component, the exit flow, tries to create the final categorization. A depthwise separable convolution is the first step, followed by an average pooling layer. To acquire the final class probabilities, the output of the average pooling layer is flattened into a vector and fed into a fully connected layer with softmax activation.

The Xception model, which contains 71 layers, can categorize images into 1000 different groups. The entry flow, the middle flow, and the exit flow are its three basic components. The entry flow applies several common convolutions and max pooling operations on the input image to prepare it for the following layers. The model may learn complicated features since the middle flow iteratively repeats the same block of depthwise separable convolutions eight times. The middle flow's output is combined with further depthwise separable convolutions, global average pooling, and a softmax layer in the exit flow, which is followed by the final prediction.

2.8 InceptionV3

Popular deep learning architecture InceptionV3 was created for image classification challenges. As an expansion of the original Inception architecture, Google introduced it in 2015. To better depict complicated patterns in images, InceptionV3 combines a variety of different-sized filters to collect data at various sizes.It was designed for image classification tasks which is an extension of the original Inception architecture. InceptionV3 applies a range of different-sized filters to collect data at various sizes to better describe intricate patterns in images.

The major modifications done on the Inception V3 model are —

1. Factorization into Smaller Convolutions
2. Spatial Factorization into Asymmetric Convolutions
3. Utility of Auxiliary Classifiers
4. Efficient Grid Size Reduction

2.8.1 Mathematical Representation

Let's denote the input image to the InceptionV3 architecture as X, which has dimension H × W × C, where H represents the height, W represents the width, and C represents the number of channels. The architecture consists of a series of modules, each employing various sizes of filters to extract features from the input.

1. Basic Convolution Block:The InceptionV3 architecture begins with a fundamental convolutional block that applies a series of convolutional layers with varying filter sizes on the input picture X. After each convolutional layer, batch normalization and a non-linear activation (typically ReLU) are used. These layers' outputs are then combined to generate a single output.
2. Inception Module:The Inception module serves as the foundation of InceptionV3. To collect features at various scales, it uses numerous simultaneous convolutional procedures with various filter sizes.
    - 1x1 Convolution: This operation applies 1x1 filters to reduce the dimensionality of the input. It helps to reduce computational complexity.
    - 3x3 Convolution: This operation applies 3x3 filters to capture more spatial information in the input.
    - 5x5 Convolution: This operation applies 5x5 filters to capture features on a larger receptive field.
    - Max Pooling: This operation applies max pooling with a stride of 1 to capture the most important features.

The final output of the Inception module is generated by concatenating the results of various simultaneous techniques. In total, the inception V3 model is made up of 42 layers which is a bit higher than the previous inception V1 and

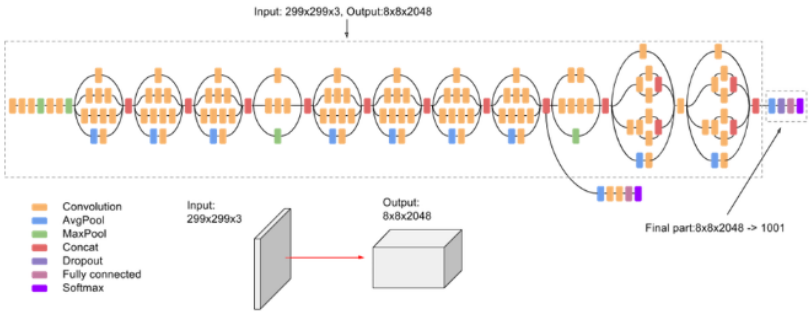


Figure 12: Architecture of InceptionV3

V2 models.

2.9 DenseNet-121

The first convolutional layer, which gets the input, is the only one in a standard feed-forward convolutional neural network (CNN) that receives the output of the convolutional layer before it. This convolutional layer then creates an output feature map, which is then passed on to the subsequent convolutional layer. As a result, there are "L" direct connections for each layer, one from one to the next.

The 'vanishing gradient' problem, however, becomes more evident as the CNN's layers grow in number and depth. This implies that when the channel for information from the input to the output layers lengthens, it may cause some information to 'vanish' or disappear, which hinders the network's capacity to train efficiently. This issue is solved by DenseNets by altering the typical CNN design and streamlining the connection structure between layers. The name "Densely Connected Convolutional Network" comes from the fact that each layer in a DenseNet design is closely connected to every other layer. There are L(L+1)/2 direct connections for layer 'L'.

1. Connectivity: The feature maps from all the preceding layers are concatenated and utilized as inputs in each layer rather than being averaged. Because duplicate feature mappings are removed, DenseNets require fewer parameters than an equivalent standard CNN, allowing for feature reuse. The feature-maps from all previous layers, x0,...,xl-1, are therefore input for the lth layer, where [x0,x1,...,xl-1] is the concatenation of the feature-maps, or the output from all previous layers (0,...,l-1). To make implementation simpler, Hl's many inputs are combined into a single tensor.

$$x_l = H_l([x_0, x_1, x_2, ..., x_{l-1}])$$

2. DenseBlocks: When the size of feature maps varies, it is not possible to utilize the concatenation method. However, downsampling of layers, which minimizes the size of feature-maps through dimensionality reduction to enable faster calculation rates, is a crucial component of CNNs. In order to do this, DenseNets are divided into DenseBlocks, where the size of the feature maps inside a block stays constant but the number of filters between them varies. Transition Layers are the layers in between the blocks that cut the number of channels in half compared to the number of channels currently in use.

3. Growth Rate: The characteristics may be regarded as the network's overall status. After passing through each thick layer, the size of the feature map increases because each layer adds 'K' features on top of the global state (pre-existing features). The network's growth rate, denoted by the parameter "K," controls how much information is added to each layer of the network. If each function H l generates k feature maps, the lth layer has—

$$k_l = k_0 + k \times (l - 1)$$

   input feature maps, where k0 is the number of channels in the input layer. Unlike existing network architectures, DenseNets can have very narrow layers.

4. Bottleneck Layers:Despite the fact that each layer only generates k output feature maps, there can be a significant amount of input, especially for subsequent levels. To increase computing efficiency and speed, a 1x1 convolution layer might be added as a bottleneck layer before each 3x3 convolution.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figure 13: Table of DenseNet-121

The table above provides a summary of the many architectures used to build the ImageNet database. The number of pixels shifted across the input matrix called the stride. The filters are moved 'n' pixels at a time when the stride parameter is set to 'n' (the default value is 1).

By understanding the table using the DenseNet-121 design, we can observe that each dense block includes a variety of layers (repetitions) with two convolutions each; a bottleneck layer with a 1x1-sized kernel and a convolution layer with a 3x3 kernel.

Also, each transition layer has a 1x1 convolutional layer and a 2x2 average pooling layer with a stride of 2.

Thus, the layers present are as follows:

I. Basic convolution layer with 64 filters of size 7X7 and a stride of 2

II. Basic pooling layer with 3x3 max pooling and a stride of 2

III. Dense Block 1 with 2 convolutions repeated 6 times

IV. Transition layer 1 (1 Conv + 1 AvgPool)

V. Dense Block 2 with 2 convolutions repeated 12 times

VI. Transition layer 2 (1 Conv + 1 AvgPool)

VII. Dense Block 3 with 2 convolutions repeated 24 times

VIII. Transition layer 3 (1 Conv + 1 AvgPool)

IX. Dense Block 4 with 2 convolutions repeated 16 times

X. Global Average Pooling layer- accepts all the feature maps of the network to perform classification

XI. Output layer

DenseNet-121 consists of 120 Convolutions and 4 AvgPool.Deeper layers can use features that were retrieved earlier because all levels, even those in the same dense block and transition layers, distribute their weights over multiple inputs.The layers in the second and third dense blocks provide the output of the transition layers the least weights since they produce a lot of reproduced information.Additionally, even though the final layers depend on the weights of the entire dense block, it's possible that higher-level features are created further into the model because there appeared to be a greater emphasis on final feature maps in trials.

2.10 SqueezeNet

SqueezeNet is a compact deep learning network optimized for resource-constrained inference. DeepScale and UC Berkeley academics introduced it in 2016. SqueezeNet uses fire modules, which are created to decrease the amount of parameters while maintaining representational power, to strike a compromise between model size reduction and accuracy.

Certain techniques are used by the SqueezeNet model to reduce the bulk of parameters. Which are:

1. Substituting 1x1 filters for the 3x3 filters
2. Limiting the use of 3x3 filters to the input channels
3. Later downsampling at the network

Details are discussed below:

1. Replacing the 3x3 filters with 1x1 filters: Due to budget constraints, the model employed the use of 1x1 filters as opposed to the traditional 3x3 filters. Thus the model has 9 times fewer parameters than a traditional filter.
2. Decreasing the number of input channels to 3x3 filters: As the filter size is reduced to 1x1, the number of input channels also needs to be reduced. This is done using the squeeze layer, which will be discussed later.
3. Downsampling later in the network: By downsampling later in the network, we are able to get larger activation maps for the convolution layers.

   In a convolutional network, each convolution layer produces an output activation map with a spatial resolution that is at least 1x1 and often much larger than 1x1. The height and width of these activation maps are controlled by:

   I. The size of the input data (256x256 images) and

II. The choice of layers in which to downsample in the CNN architecture.

By setting the (stride > 1) in a few of the convolution or pooling layers, downsampling is built into CNN designs. Most layers in the network will have tiny activation maps if the layers closest to the input layer have big strides. The network will have many layers with high activation maps if, on the other hand, the majority of layers have a stride of 1, and the strides bigger than 1 are clustered around the classifier at the end of the network.Strategies 1 and 2 are mainly to reduce the number of parameters. Strategy 3 is all about maximizing accuracy with a restricted amount of parameters.

Eight Fire modules (fire2-9) are placed before a final convolution layer (conv10) in the SqueezeNet design, which starts with an isolated convolution layer (conv1). From the start of the network until the finish, the number of filters per fire module steadily increases. In accordance with Strategy 3, SqueezeNet places pooling relatively late, following layers conv1, fire4, fire8, and conv10, with a stride of 2. Below is a demonstration of the SqueezeNet architecture's whole Architecture.

SqueezeNet is displayed on the left, followed by SqueezeNet with a basic bypass, and SqueezeNet with a sophisticated bypass.

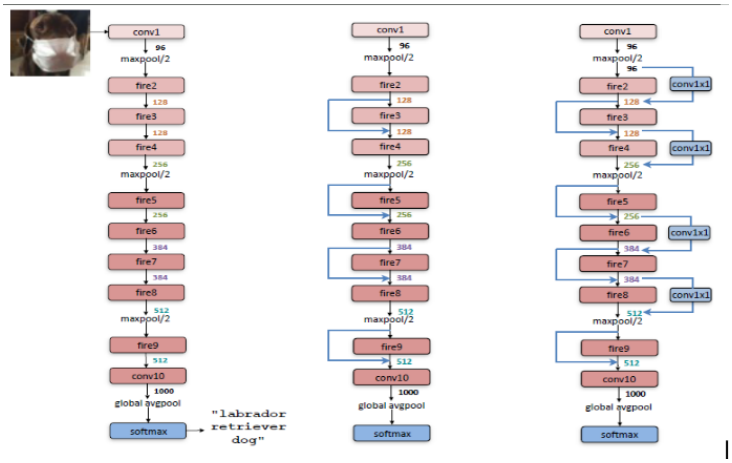


Figure 14: Architecture of SqueezeNet

We can say,Squeezenet is a CNN architecture that has 50 times fewer parameters than AlexNet but still maintains AlexNet level accuracy.

2.11 NASNetMobile

Neural Search Architecture (NAS) Network is the name of the machine learning model known as NASNet. In 2018, Google researchers introduced the NASNetMobile architecture (Neural Architecture Search Network Mobile), a deep learning model. It is made to maintain high performance in a mobile environment while effectively balancing model size and processing resources. Neural architecture search (NAS) methods that automate the building of deep learning networks led to the development of NASNetMobile.

1. Input Layer: NasNetMobile takes an input image with a fixed size, typically 224x224 pixels.
2. Stem Convolution:
   - The input image is first passed through a series of convolutional layers to extract low-level features.
   - The stem convolution includes two convolutional layers with batch normalization and ReLU activation functions.

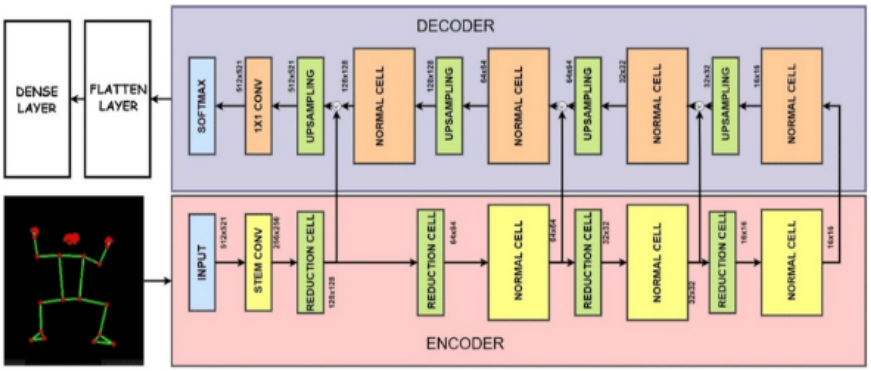


Figure 15: Architecture of NASNetMobile

3. Cell Blocks:
    - NasNetMobile consists of multiple cell blocks that are stacked on top of each other.
    - Each cell block contains two types of cells: normal cells and reduction cells.
    - Normal cells are responsible for capturing features, while reduction cells reduce the spatial dimensions of the feature maps.
4. Normal Cell:
    - The normal cell consists of a series of operations performed in parallel.
    - These operations include 1x1 and 3x3 convolutions, max-pooling, average-pooling, and skip connections.
    - The output of these operations is concatenated to form the cell's output.
5. Reduction Cell:
    - The reduction cell also performs operations in parallel.
    - It uses larger convolutions and pooling operations to reduce spatial dimensions.
    - The output of these operations is concatenated to form the cell's output.
6. Intermediate Layers: After the cell blocks, NasNetMobile includes additional convolutional layers to further process the feature maps.
7. Global Average Pooling:The feature maps are globally average-pooled to reduce their spatial dimensions to 1x1.
8. Fully Connected Layer:
    - A fully connected layer is added to the network to perform the final classification.
    - The number of neurons in this layer is typically equal to the number of classes in the classification task.
9. Output Layer: The output layer usually employs a softmax activation function to convert the network's final predictions into class probabilities.

Like other models in the NASNet family, NASNetMobile is renowned for its effective architecture and effective performance across a range of image classification workloads. It was created to have competitive accuracy and be computationally efficient, making it suited for embedded and mobile systems.

2.12 MobileNetV2

A convolutional neural network (CNN) architecture called MobileNetV2 was created for mobile and resource-constrained devices. In 2018, Google researchers announced it as an upgrade from the original MobileNet design. MobileNetV2 strives for improved accuracy while keeping efficient processing and model size.

For a variety of computer vision applications, such as image classification, object identification, and more, MobileNetV2 is a mobile-friendly convolutional neural network (CNN) architecture. It replaces the original MobileNet architecture and aims to increase performance while preserving deployment effectiveness for mobile and embedded devices.

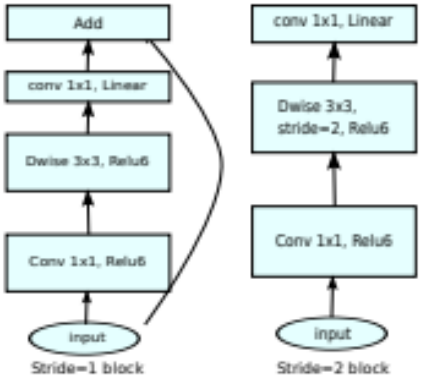


Figure 16: Architecture of MobileNetV2

Below is an overview of the MobileNetV2 architecture:

1. Input Layer:
   - MobileNetV2 typically takes an input image with a size of 224x224 pixels, which is a common size for many image classification tasks.
2. Initial Convolution:
   - The input image is passed through an initial convolutional layer, which is a standard 3x3 convolution with batch normalization and ReLU activation.
   - This layer extracts initial features from the input image.
3. Inverted Residual Blocks:
   - The core building blocks of MobileNetV2 are called inverted residual blocks.
   - Each inverted residual block consists of three key components:
     – Bottleneck Layer:

- A 1x1 depthwise convolution (pointwise convolution with 1x1 kernel) to reduce the number of input channels.
- Batch normalization and ReLU activation.

– Depthwise Convolution:

- A depthwise separable convolution, which includes a depthwise convolution (3x3) and pointwise convolution (1x1).
- Batch normalization and ReLU activation are applied after each convolution.
- Depthwise convolution reduces computational cost by applying a separate 3x3 convolution to each input channel.

– Linear Bottleneck:

- Another 1x1 pointwise convolution to expand the number of channels back to the original width.
- Batch normalization but no ReLU activation.

- These blocks are designed to reduce computation while capturing important features.

4. Inverted Residual Block Sequences:
   - Multiple inverted residual blocks are stacked together, forming sequences of these blocks.
   - The number of blocks and their specific configurations may vary depending on the desired model size and task.
5. Feature Pyramid:
   - MobileNetV2 often includes additional layers for feature pyramid construction, which is useful for object detection tasks.
   - These layers capture features at multiple scales within the network.
6. Global Average Pooling (GAP):
   - The feature maps are globally average-pooled to produce a fixed-size feature vector.
   - This feature vector is used for final classification or regression tasks.
7. Fully Connected Layer (Optional):
   - In some versions of MobileNetV2, a fully connected layer may be added for classification tasks.
   - Typically followed by a softmax activation for classification or a linear activation for regression.

MobileNetV2 is well-known for its ability to balance model size, computational cost, and performance. It has been frequently used in applications for mobile and edge devices where resource limitations are an issue. Users can select the best MobileNetV2 variations for their particular use cases by adjusting the width multipliers and input resolutions to meet their individual accuracy and computational needs.

3 Vehicle Classification Datasets

In this chapter, we have presented the six vehicle classification datasets: BIT_Vehicle, IDD, DhakaAI, Poribohon-BD, Sorokh-Poth, and VTID2 that are used to train the pre-trained CNN models. The key characteristics of the datasets are tabulated in Table 2.

Table 2: Vehicle Classification Datasets Details.

No	Dataset	Vehicle Classes	Total Images	Size of Dataset
1	BIT-Vehicle	6	9,580	2.8 GB
2	IDD	9	10,004	18.5 GB
3	DhakaAI	21	6,956	1.6 GB
4	Poribohon-BD	15	9,058	7.6 GB
5	Sorokh-Poth	10	9,809	9.4 GB
6	VTID2	5	4,356	194 MB

3.1 BIT-Vehicle

In 2015, Dong et al. introduced the BIT-Vehicle dataset, which originated from the Beijing Institute of Technology, China. Total 9,850 images are captured from two cameras across various locations and time frames, including front, rear, and side views. The vehicle images are classifier into six categories which include: Microbus, Minivan, Bus, Sedan, Truck, and SUV containing 883, 476, 558, 5922, 822, and 1392 images respectively. With a size of 2.8 GB, it's a large dataset ideal for measuring performance of image classification pre-trained models for vehicle classification and detection.

3.2 IDD

Varma et al. presented the India Driving Dataset (IDD) in 2018, which is a substantial resource for semantic segmentation and object recognition. The IDD dataset includes 10,004 images with 34 annotated classes having seven vehicle types i.e. motorcycle, bicycle, auto-rickshaw, car, truck, bus, and caravan. The number of vehicles per class is as follows: 4238, 61, 1565, 8164, 2340, 1724, and 50 respectively. The dataset captured the subtleties of Indian road scenes, collected from Hyderabad and Bangalore. The IDD dataset totals 18.5 GB and focuses on road scene understanding in unstructured Indian environments.

3.3 DhakaAI

In 2020, Shihavuddin and Rashid launched a public dataset called DhakaAI, which is specifically designed to develop traffic and transportation related applications based on Dhaka city, capital of Bangladesh. The dataset consists of 3,953 images that capture various traffic scenarios, making it a valuable resource for training algorithms in tasks such as object detection, tracking, and traffic flow analysis. The dataset is divided into three subsets: the training set comprising 3003 images, the test set containing 500 images, and test-2 with 450 images. The DhakaAI dataset comprises 21 vehicle classes including car (5467), bus (3327), motorbike (2280), CNG (2986), rickshaw (3536), truck (1492), pickup (1224), minivan (934), SUV (857), van (755), bicycle (459), auto-rickshaw (372), human hauler (169), wheelbarrow (119), ambulance (70), minibus (95), taxi (60), army- vehicle (43), scooter (38), police-car(32), and garbage-van (3). It has a size of 1.6 GB, making it well-suited to addressing the specific challenges of Bangladeshi traffic scenarios. The dataset created for the AI-based Dhaka Traffic Detection Challenge is intended to evaluate the effectiveness of sophisticated techniques in identifying and detecting traffic vehicles.

3.4 Poribohon-BD

A dataset Poribohon-BD was presented by Tabassum et al. in 2020, has been specifically designed for vehicle classification tasks in Bangladesh. It has a total size of 7.5 GB and contains 9,058 images of 15 native Bangladeshi vehicle types which are Bicycle (1617), Boat (1974), Bus (3711), Car (1698), CNG (3214), Easy-bike (2062), Horse-cart (1581), Launch (332), Leguna (1686), Motorbike (746), Rickshaw (3386), Tractor (509), Truck (1673), Van (2057), Wheelbarrow (605), captured from diverse positions, perspectives, lighting conditions, weather conditions, and backdrops. The images are taken from roads and highways in Bangladesh using smartphone cameras where 1791 images are produced through data augmentation methods, and approximately 4,000 images are obtained from Facebook. The authors highlighted the dataset's practical applications, particularly in areas such as traffic

monitoring and intelligent transportation systems.

3.5 Sorokh-poth

Recently in 2023, Sana et al. introduced the Sorokh- Poth dataset that includes 9,809 annotated images of ten Bangladeshi vehicle types, including auto-rickshaw (1004), bike (1007), bus (1181), bicycle (1048), car (1136), CNG (1048), leguna (642), rickshaw (750), truck (1074), and van(1038) with a size of 9.4 GB. The images were collected through smartphones and social media platforms. During the image capture process, various factors such as weather conditions, backgrounds, vehicle direction, and lighting conditions were taken into consideration.

3.6 VTID2

In 2021, Boonsirisumpun and Surinta [55] introduced the Vehicle Type Image Dataset (VTID2) to investigate the most commonly used car types in Thailand. The dataset consists of 4,356 images categorised into five vehicle classes containing sedans, pickups, hatchbacks, SUVs, and other vehicle types. The number of vehicles per class is as follows: 230, 1240, 606, 680, and 600 respectively which has a total size of 194 MB. Two cameras were installed at the university front gate to record the images during the daytime for four weeks between July and December 2018.

4 Data Preparation and Preprocessing

4.1 Enhancing Dataset Accessibility and Compatibility

In our initial exploration, we encountered datasets in various formats, some of which were not immediately compatible with popular deep learning frameworks like PyTorch or TensorFlow. This highlighted the importance of data preprocessing and the need for standardized dataset formats to facilitate seamless integration with transfer learning techniques.

```python
import tensorflow as tf
import h5py
import numpy as np

def convert_to_standard_format(images, labels, filename, format='tfrecord'):
    if format == 'tfrecord':
        # Define function to serialize image and label into TFRecord example
        def serialize_example(image, label):
            feature = {
                'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[tf.io.serialize_tensor(image).numpy()])),
                'label': tf.train.Feature(int64_list=tf.train.Int64List(value=[label])),
            }
            example = tf.train.Example(features=tf.train.Features(feature=feature))
            return example.SerializeToString()

        # Write TFRecord file
        with tf.io.TFRecordWriter(filename) as writer:
            for image, label in zip(images, labels):
                tf_example = serialize_example(image, label)
                writer.write(tf_example)

    elif format == 'hdf5':
        # Write HDF5 file
        with h5py.File(filename, 'w') as f:
            f.create_dataset('images', data=images)
            f.create_dataset('labels', data=labels)

    else:
        raise ValueError("Invalid format. Please choose 'tfrecord' or 'hdf5'.")
```

The convert_to_standard_format function takes in images, labels, a filename, and an optional format parameter to convert the given data into a standardized format for machine learning tasks. If the specified format is 'tfrecord', it leverages TensorFlow's TFRecord binary file format and serializes each image and its corresponding label into a Protobuf example using the serialize_example function. These serialized examples are then written to the specified file using a TFRecordWriter. Alternatively, if the format is 'hdf5', it utilizes the HDF5 (Hierarchical Data Format 5) file format, creating two separate datasets within the HDF5 file to store the images and labels respectively. This function streamlines the process of data preprocessing and standardization, enabling efficient storage and organization of large amounts of data in formats compatible with deep learning frameworks like TensorFlow. By providing a centralized approach to data conversion, it facilitates seamless integration of image data into machine learning pipelines, ensuring consistency and compatibility across various stages of the modeling process.

4.2 Dataset Analysis and Balancing

To better understand the characteristics of our dataset, we calculated basic statistics, including the total number of images, the resolution (width x height) of images, and the image formats (e.g., JPEG, PNG). This analysis provided valuable insights into the dataset's composition and helped identify any potential issues or inconsistencies. During this analysis, we discovered that some image samples of certain classes were significantly more numerous than others, which could lead to class imbalance issues. To mitigate this problem, we employed sampling techniques to ensure a more balanced representation of classes within our dataset. Addressing class imbalance was crucial, as it can significantly impact the performance and generalization ability of machine learning models trained on the dataset.

Next step was to load the actual data. We utilized the h5py library, a powerful Python interface for reading and writing HDF5 files, to access the dataset stored in the 'vtid2-dataset.h5' file. The code below demonstrates how we efficiently loaded the image data and corresponding labels:

```
import h5py

# Load vtid2-dataset
with h5py.File('/kaggle/input/vtid2-dataset/vtid2_dataset.h5', 'r') as file:
    images = file['images'][:]
    labels = file['labels'][:]

# Print the total number of images and labels
print('Total number of images:', len(images))
print('Total number of labels:', len(labels))
```

In this code snippet, we first imported the h5py library. Then, we opened the 'vtid2_dataset.h5' file in read mode using the h5py.File function and a context manager (with statement) to ensure proper resource handling. We retrieved the 'images' and 'labels' datasets from the opened HDF5 file using dataset indexing (file['images'] and file['labels']), and stored them in the images and labels variables, respectively. The slice notation [:] was used to load the entire dataset into memory. Finally, we printed the total number of images and labels to verify that the loading process was successful and to get a sense of the dataset's size.

By following this approach, we ensured that the dataset was loaded efficiently and consistently, preparing the data for further preprocessing and modeling steps. Then we proceeded to resize and preprocess the images to prepare them for training our deep learning model. For that, We first imported the necessary libraries: Xception and preprocess_input from the tensorflow.keras.applications.xception module, Image from the Pillow library, and numpy for array operations.

Next, we resized all images to a consistent size of (224, 224) using the Pillow library's Image.resize function. This step ensured that our images were compatible with the input size expected by the Xception model. We converted the resized images from Pillow's Image objects to NumPy arrays, as this is the format required by the deep learning libraries.

```
from tensorflow.keras.applications.xception import Xception, preprocess_input
from PIL import Image
```

```
import numpy as np

resized_images = [Image.fromarray((image * 255).astype(np.uint8)).resize((224, 224)) for image in
    images]
resized_images = np.array([preprocess_input(np.array(image)) for image in resized_images])
```

We applied the preprocess_input function provided by the Xception model to preprocess the resized images. This step typically involves normalization and other transformations specific to the model's training process, ensuring that the input data is in the correct format and range for optimal performance.

```
import numpy as np

# Sample code
unique_classes, class_counts = np.unique(labels, return_counts=True)
for class_label, count in zip(unique_classes, class_counts):
    print(f"Class {class_label.decode('utf-8')}: {count} samples")
```

Class b'autorickshaw': 1565 samples
Class b'bicycle': 61 samples
Class b'bus': 1724 samples
Class b'car': 8164 samples
Class b'caravan': 50 samples
Class b'motorcycle': 4238 samples
Class b'truck': 2340 samples

In the datasets, including IDD and DhakaAI, sampling was used to make the dataset balanced by oversampling the minority classes. Following code snippet implements a sampling strategy to balance the dataset. It encodes the labels using LabelEncoder, flattens the images, and then iterates over each class. For classes with a count below a certain threshold (2340), it adds all the samples to the oversampled set.

```
threshold_low_samples = 2340

label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)
label_mapping = {i: label for i, label in enumerate(label_encoder.classes_)}

flattened_images_reshaped = np.array([img.flatten() for img in resized_images])
num_samples, num_features = flattened_images_reshaped.shape

oversampled_images = []
oversampled_labels = []

to_sample_images = []
to_sample_labels = []

unique_classes, class_counts = np.unique(encoded_labels, return_counts=True)

for class_label, count in zip(unique_classes, class_counts):
    class_indices = np.where(encoded_labels == class_label)[0]

    if count <= threshold_low_samples:
        to_sample_images.extend(flattened_images_reshaped[class_indices])
        to_sample_labels.extend(encoded_labels[class_indices])
    else:
        num_samples_to_keep = threshold_low_samples

        indices_to_keep = np.random.choice(class_indices, size=num_samples_to_keep, replace=False)
        oversampled_images.extend(flattened_images_reshaped[indices_to_keep])
        oversampled_labels.extend(encoded_labels[indices_to_keep])

if len(to_sample_images) >= 3:
```

```
    smote = SMOTE(sampling_strategy='auto', random_state=42, k_neighbors=2)
    oversampled_class_images, oversampled_class_labels = smote.fit_resample(to_sample_images,
    to_sample_labels)

    oversampled_images.extend(oversampled_class_images)
    oversampled_labels.extend(oversampled_class_labels)
```

For classes above the threshold, it randomly selects a fixed number of samples (2340) and adds them to the oversampled set. Additionally, it uses the SMOTE (Synthetic Minority Over-sampling Technique) algorithm to generate synthetic samples for the minority classes.

4.3 Data Augmentation for Diversity

Recognizing the importance of diversity in training data, we utilized data augmentation techniques to introduce variations and increase the overall size of our datasets including Bit-Vehicle, Poribohon-BD, Sorokh-Poth and VTID2. For example, certain classes were underrepresented, so we considered augmenting the minority classes through techniques like brightness adjustment, flipping, rotating, zooming, and shifting. This approach helped improve the robustness and generalization capabilities of our models.

```
def adjust_brightness(image, factor):
    image = image.astype(np.float32)
    augmented_image = image + factor
    augmented_image = np.clip(augmented_image, 0, 255)
    augmented_image = augmented_image.astype(np.uint8)
    return augmented_image

def flip_image(image, flip_code):
    return cv2.flip(image, flip_code)

def rotate_image(image, angle):
    rows, cols = image.shape[:2]
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1)
    return cv2.warpAffine(image, M, (cols, rows))

def zoom_image(image, zoom_factor):
    rows, cols = image.shape[:2]
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), 0, zoom_factor)
    return cv2.warpAffine(image, M, (cols, rows))

def shift_image(image, dx, dy):
    rows, cols = image.shape[:2]
    M = np.float32([[1, 0, dx], [0, 1, dy]])
    return cv2.warpAffine(image, M, (cols, rows))
```

```
augmented_images = []
augmented_labels = []

for img, label in zip(resized_images, labels):
    if label == b'Bus':
        augmented_img_brightness = adjust_brightness(img, 50)
        augmented_img_flip_horizontal = flip_image(img, 1)
        augmented_img_flip_vertical = flip_image(img, 0)
        augmented_img_rotate = rotate_image(img, 30)
        augmented_img_zoom = zoom_image(img, 1.2)
        augmented_img_shift = shift_image(img, 20, 20)
        augmented_images.extend([
            augmented_img_brightness,
            augmented_img_flip_horizontal,
            augmented_img_flip_vertical,
            augmented_img_rotate,
            augmented_img_zoom,
            augmented_img_shift,
        ])
```

```
        augmented_labels.extend([label] * 6)
    if label == b'Minivan':
        augmented_img_brightness = adjust_brightness(img, 50)
        augmented_img_flip_horizontal = flip_image(img, 1)
        augmented_img_flip_vertical = flip_image(img, 0)
        augmented_img_rotate = rotate_image(img, 30)
        augmented_img_zoom = zoom_image(img, 1.2)
        augmented_img_shift = shift_image(img, 20, 20)
        augmented_images.extend([
            augmented_img_brightness,
            augmented_img_flip_horizontal,
            augmented_img_flip_vertical,
            augmented_img_rotate,
            augmented_img_zoom,
            augmented_img_shift,
        ])
        augmented_labels.extend([label] * 6)

    if label == b'Microbus':
        augmented_img_flip_horizontal = flip_image(img, 1)
        augmented_img_flip_vertical = flip_image(img, 0)
        augmented_images.extend([
            augmented_img_flip_horizontal,
            augmented_img_flip_vertical,
        ])
        augmented_labels.extend([label] * 2)

    if label == b'SUV':
        augmented_img_rotate = rotate_image(img, 30)
        augmented_images.extend([
            augmented_img_rotate,
        ])
        augmented_labels.extend([label] * 1)

    if label == b'Truck':
        augmented_img_flip_horizontal = flip_image(img, 1)
        augmented_img_flip_vertical = flip_image(img, 0)
        augmented_img_rotate = rotate_image(img, 30)
        augmented_images.extend([
            augmented_img_flip_horizontal,
            augmented_img_flip_vertical,
            augmented_img_rotate,
        ])
        augmented_labels.extend([label] * 3)

    else:
        augmented_images.append(img)
        augmented_labels.append(label)
```

The provided code performs data augmentation on the BIT-Vehicle dataset by applying various transformations to the images based on their class labels. The augmentation techniques used include adjusting brightness, flipping horizontally and vertically, rotating, zooming, and shifting images. Here's a breakdown of the process:

The code iterates over each image and its corresponding label in the dataset. It performs different augmentation operations based on the class label of the image. For example:

- For images labeled as "Bus" or "Minivan", it applies six different augmentation techniques: adjusting brightness, flipping horizontally, flipping vertically, rotating by 30 degrees, zooming by a factor of 1.2, and shifting the image by 20 pixels in both x and y directions. These augmented images and their respective labels are added to the lists of augmented images and labels.
- For images labeled as "Microbus", it performs two augmentation operations: flipping horizontally and flipping vertically. The augmented images and their labels are added to the respective lists.

- For images labeled as "SUV", it applies a single augmentation technique: rotating the image by 30 degrees. The augmented image and its label are added to the lists.
- For images labeled as "Truck", it performs three augmentation operations: flipping horizontally, flipping vertically, and rotating by 30 degrees. The augmented images and their labels are added to the respective lists.
- For images belonging to any other class, no augmentation is performed, and the original image and its label are added to the lists of augmented images and labels.

```
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(augmented_labels)
label_mapping = {i: label for i, label in enumerate(label_encoder.classes_)}
labels = to_categorical(encoded_labels, num_classes)
unique_labels_set = set(tuple(label) for label in labels)
for unique_label in unique_labels_list:
    print(unique_label)
```

Provided code first uses LabelEncoder to convert categorical labels into numerical format, mapping 'hatchback' to 0, 'other' to 1, 'pickup' to 2, 'sedan' to 3, and 'suv' to 4. It then performs one-hot encoding on these numerical labels using PyTorch's nn.functional.one_hot, creating a binary tensor representation where each column represents a class, with 1 indicating the corresponding class and 0 for the rest. The resulting one-hot encoded tensor is converted to float data type, suitable for machine learning models that work with numerical data.

After augmentation process we got augmentation summary as follow.

```
print('Total number of augmented images: ',len(augmented_images))
print('Total number of augmented labels: ',len(augmented_labels))

unique_classes_aug, class_counts_aug = np.unique(augmented_labels, return_counts=True)
for class_label, count in zip(unique_classes_aug, class_counts_aug):
    print(f"Class {class_label}: {count} samples")
```

Total number of augmented images: 20714
Total number of augmented labels: 20714
Class b'Bus': 3885 samples
Class b'Microbus': 2580 samples
Class b'Minivan': 3269 samples
Class b'SUV': 2744 samples
Class b'Sedan': 5776 samples
Class b'Truck': 2460 samples

5 Model Training & Evaluation

5.1 Enhancing Dataset Accessibility and Compatibility

This chapter will cover the process of training and evaluating pre-trained models from popular deep learning frameworks like PyTorch and TensorFlow, as well as the state-of-the-art object detection model YOLOv8 from Ultralytics. It will provide a comprehensive guide to fine-tuning these models on custom datasets and evaluating their performance using industry-standard metrics.

5.2 Fine-tuning Pre-trained Models with PyTorch

In this sub-chapter, we will explore the process of leveraging pre-trained models like AlexNet, GoogLeNet, SqueezeNet, and MobileNetV2 to fine-tune them on a custom vehicle dataset. These models, initially trained on large-scale datasets like ImageNet, have learned robust feature representations that can be transferred to new tasks with

relatively small datasets. By freezing the earlier layers and fine-tuning the later layers, we can adapt these models to our specific vehicle classification task, while benefiting from their prior learning.

At first, Import required libraries for data handling, image manipulation, plotting, machine learning, and model evaluation.

```
import torch
import h5py
import cv2
from PIL import Image
import numpy as np
import pandas as pd
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
import seaborn as sns
import torchvision.transforms as transforms
import torchvision.models as models
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import classification_report, roc_curve, confusion_matrix
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder, label_binarize
```

After data preprocessing phase, we got dataset, then we do following to prepare data for model training.

```
class ImageDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        x = self.data[index]
        y = self.labels[index]

        if self.transform:
            x = self.transform(x)

        return x, y
transform = transforms.Compose([

    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])


dataset = ImageDataset(np.array(augmented_images), labels, transform=transform)
```

The provided code defines a custom PyTorch dataset class ImageDataset that inherits from torch.utils.data.Dataset. It overrides the ___init___, ___len___, and ___getitem___ methods to handle the dataset initialization, length determination, and data sample retrieval, respectively. The ___init___ method stores the input data, labels, and an optional transformation. The ___len___ method returns the length of the dataset, while the ___getitem___ method retrieves a single data sample (image and label) at the specified index, applies the transformation if provided, and returns the transformed image and label as a tuple. Additionally, a transformation pipeline is defined using torchvision.transforms.Compose, which converts the numpy array to a PIL Image, resizes it to 224x224 pixels, and converts it to a PyTorch tensor. Finally, an instance of the ImageDataset class is created with the augmented_images, labels, and the defined transform.

Then we move on to setting up the pretrained models for fine-tuning. The code first loads the pretrained GoogLeNet model from PyTorch's torchvision.models module using models.googlenet(pretrained=True). This model was initially trained on the ImageNet dataset for image classification tasks.

```
model = models.googlenet(pretrained=True)
num_input_features = model.fc.in_features
model.fc = nn.Linear(num_input_features, num_classes)
use_cuda = torch.cuda.is_available()
if use_cuda:
model.cuda()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Next, it retrieves the number of input features from the last fully connected layer (model.fc.in_features). This is necessary because we will replace the final layer with a new one to match the number of classes in our custom dataset.

The final fully connected layer is then replaced with a new linear layer nn.Linear(num_input_features, num_classes), where num_classes is the number of classes in our dataset.

The code checks if a CUDA-enabled GPU is available using torch.cuda.is_available(). If available, it moves the model to the GPU using model.cuda() and sets the device accordingly using device = torch.device("cuda"). Otherwise, it uses the CPU. Finally, the model is moved to the selected device (device) using model.to(device), preparing it for training or inference on the appropriate hardware.

```
learning_rate = 0.0001
epochs = 3
batch_size = 32
k = 3

kf = KFold(n_splits=k, shuffle=True)
accuracy_values = []
weighted_precision_values = []
weighted_recall_values = []
weighted_f1_score_values = []


all_true_labels = []
all_pred_labels = []
confusion_matrices = []

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for fold, (train_index, test_index) in enumerate(kf.split(dataset), 1):
    print("Fold:", fold)
    train_sampler = torch.utils.data.SubsetRandomSampler(train_index)
    test_sampler = torch.utils.data.SubsetRandomSampler(test_index)

    train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=
    train_sampler)
    test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=test_sampler
    )

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for batch_idx, (inputs, labels) in enumerate(train_loader):
          inputs = inputs.to(device)
          labels = labels.to(device)
```

```
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            print(f"\rEpoch {epoch+1}/{epochs}, Batch {batch_idx+1}/{len(train_loader)}, Loss: {loss.item():.4f}", end='')

        print(f"\rEpoch {epoch+1}/{epochs}, Loss: {running_loss / len(train_loader):.4f}")

    model.eval()
    y_true = []
    y_pred = []

    for batch_idx, (inputs, labels) in enumerate(test_loader):
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        predicted = F.softmax(outputs, dim=1)

        y_true.extend(labels.cpu().tolist())
        y_pred.extend(predicted.cpu().tolist())

    print(f"\rEvaluation: Batch {batch_idx+1}/{len(test_loader)}", end='\n')

    y_true_labels = np.argmax(y_true, axis=1)
    y_pred_labels = np.argmax(y_pred, axis=1)

    cm = confusion_matrix(y_true_labels, y_pred_labels)
    report = classification_report(y_true_labels, y_pred_labels, output_dict=True)

    confusion_matrices.append(cm)
    accuracy_values.append(report['accuracy'])
    weighted_precision_values.append(report['weighted avg']['precision'])
    weighted_recall_values.append(report['weighted avg']['recall'])
    weighted_f1_score_values.append(report['weighted avg']['f1-score'])

    all_true_labels.extend(y_true_labels)
    all_pred_labels.extend(y_pred)


avg_accuracy = np.mean(accuracy_values)
avg_weighted_precision = np.mean(weighted_precision_values)
avg_weighted_recall = np.mean(weighted_recall_values)
weighted_f1_score_values = np.mean(weighted_f1_score_values)

print('Average accuracy:', avg_accuracy)
print('Average weighted precision:', avg_weighted_precision)
print('Average weighted recall:', avg_weighted_recall)
print('Average weighted f1_score:', weighted_f1_score_values)
```

Accuracy: 0.999475272202545
Weighted Average Precision: 0.999475574987886
Weighted Average Recall: 0.999475272202545
Weighted Average f1-score: 0.9994752293491107

In training phase, the code sets hyperparameters like learning rate, epochs, batch size, and number of folds (k) for K-fold cross-validation using scikit-learn's KFold. It initializes lists to store evaluation metrics across folds. The loss function (nn.CrossEntropyLoss) and optimizer (torch.optim.Adam) are defined.

For each fold, the dataset is split into train and test indices using KFold.split(). PyTorch's SubsetRandomSampler and DataLoader are used to create train and test loaders. The model is moved to the appropriate device (GPU/CPU).

The training loop iterates over train loader batches for each epoch. Outputs and loss are computed, gradients are backpropagated, and the optimizer updates model parameters. The training loss is monitored.

After training, the model is set to evaluation mode. The evaluation loop iterates over test loader batches, computing outputs and converting them to probabilities using softmax. True labels and predicted probabilities are collected.

The true and predicted labels are converted to class indices. The confusion matrix and classification report (accuracy, precision, recall, F1-score) are computed and stored for the current fold. After all folds, the average accuracy, weighted precision, recall, and F1-score across folds are calculated and printed.

5.3 Transfer Learning with TensorFlow

In this sub-chapter, we will explore the process of leveraging pre-trained models like ResNet-50, VGG19, DenseNet-121, Xception, InceptionV3, and NASNetMobile from TensorFlow to fine-tune them on a custom vehicle dataset. These models, initially trained on the large-scale ImageNet dataset, have learned robust feature representations that can be transferred to new tasks with relatively small datasets. By freezing the earlier layers and fine-tuning the later layers, we can adapt these models to our specific vehicle classification task, while benefiting from their prior learning on ImageNet.

At first, Import required libraries for data handling, image manipulation, plotting, machine learning, and model evaluation.

```
import h5py
import cv2
from PIL import Image
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Dense, Flatten, Dropout, GlobalAveragePooling2D,
    BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
```

Then, we load the pre-trained VGG19 model from TensorFlow's keras.applications module, specifying the weights='imagenet' parameter to initialize the model with the weights pre-trained on the ImageNet dataset. The include_top=False parameter ensures that the fully connected layers at the top of the model are excluded, as we will add our own custom layers for our specific task. The input_shape parameter is set to (224, 224, 3), specifying the input image dimensions and the number of channels.

The output of the last convolutional layer from the base VGG19 model (base_model.layers[-1].output) is retrieved and passed through a GlobalAveragePooling2D layer to reduce the spatial dimensions and obtain a feature vector.

The feature vector is then passed through a series of fully connected (Dense) layers and dropout (Dropout) layers. These additional layers are responsible for learning the task-specific features and mapping them to the desired output classes.

After these custom layers, the final Dense layer with num_classes units and a softmax activation is added, representing the output layer for the classification task, where num_classes is the number of classes in our custom vehicle dataset.

The complete model is then constructed using Keras' Model class, taking the input from the base VGG19 model (base_model.input) and the output from the final Dense layer (predictions).

```
base_model = VGG19(weights='imagenet', include_top=False, input_shape=(224, 224, 3))


x = base_model.layers[-1].output

x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

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

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

To prevent the pre-trained weights of the VGG19 model from being updated during fine-tuning, we freeze all the layers in the base model by setting layer.trainable = False for each layer in base_model.layers.

The learning rate for the optimizer is set to a specific value (learning_rate = 0.0001), and an Adam optimizer instance is created. The model is compiled with the specified optimizer, loss function (categorical_crossentropy), and evaluation metrics (accuracy), preparing it for training on our custom vehicle dataset.

```
k = 3
epochs = 3
batch_size = 32
kf = KFold(n_splits=k, shuffle=True)

accuracy_values = []
precision_values = []
recall_values = []
f1_score_values = []

confusion_matrices = []
all_true_labels = []
all_pred_labels = []
model_history = []

for fold, (train_index, test_index) in enumerate(kf.split(original_images), 1):
    print("Fold:", fold)
    X_train_fold, X_test_fold = original_images[train_index], original_images[test_index]
    y_train_fold, y_test_fold = labels[train_index], labels[test_index]

    history = model.fit(X_train_fold, y_train_fold, epochs=epochs, batch_size=batch_size)

    y_pred = model.predict(X_test_fold)
    y_pred_labels = np.argmax(y_pred, axis=1)
    y_true_labels = np.argmax(y_test_fold, axis=1)


    cm = confusion_matrix(y_true_labels, y_pred_labels)
    report = classification_report(y_true_labels, y_pred_labels, output_dict=True)

    confusion_matrices.append(cm)
    accuracy_values.append(report['accuracy'])
```

```
    precision_values.append(report['weighted avg']['precision'])
    recall_values.append(report['weighted avg']['recall'])
    f1_score_values.append(report['weighted avg']['f1-score'])

    all_true_labels.extend(y_true_labels)
    all_pred_labels.extend(y_pred)
    model_history.append(history)

avg_accuracy = np.mean(accuracy_values)
avg_weighted_precision = np.mean(precision_values)
avg_weighted_recall = np.mean(recall_values)
avg_weighted_f1_score = np.mean(f1_score_values)

print('Average accuracy:', avg_accuracy)
print('Average weighted precision:', avg_weighted_precision)
print('Average weighted recall:', avg_weighted_recall)
print('Average weighted f1 score:', avg_weighted_f1_score)
```

Accuracy: 0.999475272202545
Weighted Average Precision: 0.999475574987886
Weighted Average Recall: 0.999475272202545
Weighted Average f1-score: 0.9994752293491107

In training phase, we set the number of folds (k=3) for K-fold cross-validation using scikit-learn's KFold, the number of epochs (epochs=3), and the batch size (batch_size=32). It initializes lists to store evaluation metrics, confusion matrices, true labels, predicted labels, and model training histories across all folds. The KFold object is created with n_splits=k and shuffle=True.

For each fold, the dataset is split into train and test indices using kf.split(original_images). The training and testing data (images and labels) are extracted using these indices. The model is trained on the training data for the specified number of epochs and batch size using model.fit(), and the training history is stored. The model predicts on the test data, and the predicted labels are computed using numpy.argmax().

The true labels are extracted from the test labels using numpy.argmax(). The confusion matrix and classification report (accuracy, precision, recall, F1-score) are computed and appended to their respective lists for the current fold. The true and predicted labels are also stored in separate lists.

After all folds, the average accuracy, weighted precision, weighted recall, and weighted F1-score are calculated across all folds and printed, providing an overall estimate of the model's performance.

5.4 Vehicle Classification with YOLOv8

The YOLOv8 image classification follows a specific folder structure. The folder structure typically consists of 'train', 'val', and 'test' directories. Within each, there are subdirectories named after the different classes, containing the respective image files for that class. This hierarchical structure organizes the dataset by splitting images into training, validation, and testing sets, with class-specific subdirectories for easy data management.

The provided code defines a function create_folders that structures a dataset for image classification using YOLOv8 by creating necessary folders and subfolders based on train, validation, and test splits. It creates a main folder, followed by 'train', 'val', and 'test' subfolders, and within each subfolder, it creates class-specific subfolders for each unique label. The function iterates over the image and label arrays, normalizes the image pixel values, and saves the normalized images as JPEG files within the corresponding class subfolder, using a specific naming convention. It also prints the number of samples for each class in the train, validation, and test datasets. This structured organization facilitates easy access and processing of the data for training and evaluation with object detection and image classification models like YOLOv8.

```
def create_folders(folder_path, augmented_images, augmented_labels, train_images, train_labels,
    val_images, val_labels, test_images, test_labels):
```

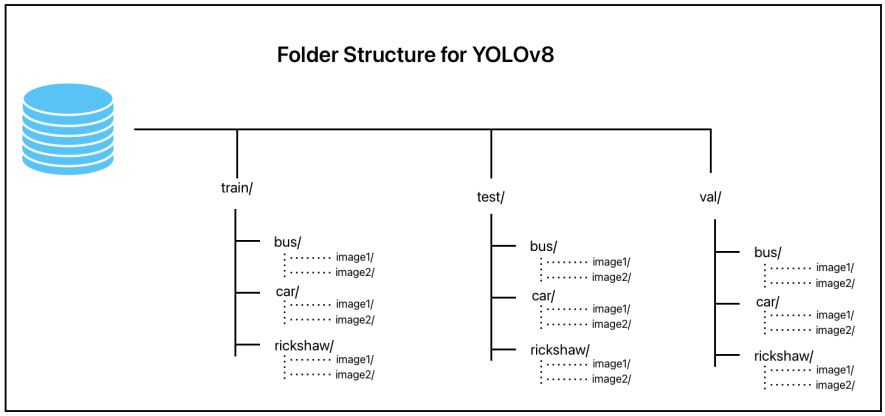


Figure 17: Folder structure for YOLOv8 image classification.

```
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
    else:
        print('Exists!')


    train_folder_path = os.path.join(folder_path, 'train')

    if not os.path.exists(train_folder_path):
        os.makedirs(train_folder_path)
    else:
        print('Exists!')

    val_folder_path = os.path.join(folder_path, 'val')

    if not os.path.exists(val_folder_path):
        os.makedirs(val_folder_path)
    else:
        print('Exists!')


    test_folder_path = os.path.join(folder_path, 'test')

    if not os.path.exists(test_folder_path):
        os.makedirs(test_folder_path)
    else:
        print('Exists!')

    distinct_labels = np.unique(augmented_labels)
    # Create folders for each labels in 'train', 'test' & 'val' folders

    for label in distinct_labels:
        train_label_folder = os.path.join(train_folder_path, str(label.decode('utf-8')))
        if not os.path.exists(train_label_folder):
            os.makedirs(train_label_folder)

        val_label_folder = os.path.join(val_folder_path, str(label.decode('utf-8')))
        if not os.path.exists(val_label_folder):
            os.makedirs(val_label_folder)

        test_label_folder = os.path.join(test_folder_path, str(label.decode('utf-8')))
        if not os.path.exists(test_label_folder):
            os.makedirs(test_label_folder)

```

```
    # Convert and store 'JPEG' files from image array
    unique_classes_aug, class_counts_aug = np.unique(train_labels, return_counts=True)
    print('Train Data:')
    for class_label, count in zip(unique_classes_aug, class_counts_aug):
        print(f"Class {class_label}: {count} samples")

    for i, (image, label) in enumerate(zip(train_images, train_labels)):
        label_folder = os.path.join(train_folder_path, str(label.decode('utf-8')))
        filename = f'image_{i}.jpeg'
        filepath = os.path.join(label_folder, filename)

        normalized_image = cv2.normalize(image, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
        cv2.imwrite(filepath, normalized_image, [cv2.IMWRITE_JPEG_QUALITY, 95])

    # Convert and store 'JPEG' files from image array
    unique_classes_aug, class_counts_aug = np.unique(val_labels, return_counts=True)
    print('Validation Data')
    for class_label, count in zip(unique_classes_aug, class_counts_aug):
        print(f"Class {class_label}: {count} samples")

    for i, (image, label) in enumerate(zip(val_images, val_labels)):
        label_folder = os.path.join(val_folder_path, str(label.decode('utf-8')))
        filename = f'image_{i}.jpeg'
        filepath = os.path.join(label_folder, filename)

        normalized_image = cv2.normalize(image, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
        cv2.imwrite(filepath, normalized_image, [cv2.IMWRITE_JPEG_QUALITY, 95])

    # Convert and store 'JPEG' files from image array
    unique_classes_aug, class_counts_aug = np.unique(test_labels, return_counts=True)
    print('Test Data')
    for class_label, count in zip(unique_classes_aug, class_counts_aug):
        print(f"Class {class_label}: {count} samples")

    for i, (image, label) in enumerate(zip(test_images, test_labels)):
        label_folder = os.path.join(test_folder_path, str(label.decode('utf-8')))
        filename = f'image_{i}.jpeg'
        filepath = os.path.join(label_folder, filename)

        normalized_image = cv2.normalize(image, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
        cv2.imwrite(filepath, normalized_image, [cv2.IMWRITE_JPEG_QUALITY, 95])
```

The code implements a k-fold cross-validation strategy to split the augmented dataset into train, validation, and test sets for each fold. It sets the number of folds to 5 and creates a KFold object from scikit-learn. After generating a random permutation of indices, it iterates over each fold, splitting the indices into train and test sets using the KFold object. From the train indices, it randomly selects a subset (1/6th) as the validation set indices. Then, it creates train, validation, and test sets by indexing the augmented images and labels with the respective indices. Finally, for each fold, it calls the create_folders function, passing the augmented images, labels, and the train, validation, and test sets, to structure the dataset for YOLOv8 image classification by creating the necessary folders and subfolders and saving the images in the appropriate locations.

```
k = 5
kf = KFold(n_splits=k, shuffle=True)

np.random.seed(42)
indices = np.random.permutation(len(augmented_images))

for fold, (train_indices, test_indices) in enumerate(kf.split(indices), 1):
    print("Fold:", fold)

    val_size = len(train_indices) / 6
    # Randomly select indices for the validation set
    val_indices = np.random.choice(train_indices, size=int(val_size), replace=False)

```

```
    # Remove the selected indices from the training set
    train_indices = [idx for idx in train_indices if idx not in val_indices]


    train_images, train_labels = [augmented_images[i] for i in train_indices], [augmented_labels[i] for i in train_indices]
    val_images, val_labels = [augmented_images[i] for i in val_indices], [augmented_labels[i] for i in val_indices]
    test_images, test_labels = [augmented_images[i] for i in test_indices], [augmented_labels[i] for i in test_indices]

    print(len(train_labels), len(test_labels), len(val_labels))

    create_folders(f'/Users/monir/Documents/Research/yolo/vtid2_dataset_KFold/fold{fold}', augmented_images, augmented_labels, train_images, train_labels, val_images, val_labels, test_images, test_labels)
```

Then, we first imported the YOLO class from the Ultralytics library, which provides an implementation of the YOLOv8 object detection model. It then specifies the directory where the structured dataset is located, loads the pre-trained 'yolov8n-cls.pt' model, and initiates the training process with the specified dataset, setting the number of epochs to 10, image size to 224, and enabling plotting of training metrics.

```
from ultralytics import YOLO

directory = '/Users/monir/Documents/Research/yolo/vtid2_dataset_KFold/fold1'

model = YOLO('yolov8n-cls.pt')
results = model.train(data=directory, epochs=10, imgsz=224, plots=True)
```

After training, we performed model validation on the fold1 dataset using the best trained weights obtained from the previous training step. It loads the 'best.pt' model weights, creates a YOLO object, and calls the 'val' method on this object to evaluate the model's performance on the validation data. The validation metrics, including top1 and top5 accuracy, are then accessible through the 'metrics' object returned by the 'val' method.

```
folder_path_test1 = '/Users/monir/Documents/Research/yolo/vtid2_dataset_KFold/fold1/test1'
images_path_test1 = []
#indices_dict = None
y_true1 = []
predictions_test = []

# Loading test images from test folder to create images path array for prediction
# Storing true labels in 'y_true' array
if os.path.exists(folder_path_test1) and os.path.isdir(folder_path_test1):
    folders = [file for file in os.listdir(folder_path_test1) if not file.startswith('.')]

    #indices_dict = {label: index for index, label in enumerate(folders)}

    for item in folders:
        folder = folder_path_test1 + '/' + item
        if os.path.exists(folder) and os.path.isdir(folder):
            image_array = [folder + '/' + file for file in os.listdir(folder) if not file.startswith('.')]
            for file in image_array:
                y_true1.append(item)

            images_path_test1.extend(image_array)

else:
    print(f"The folder '{folder_path_test1}' does not exist.")


print(len(images_path_test1))
print(len(y_true1))

```

```
#Prediction of all images from test data for fold1
results = model_fold1(images_path_test1, stream=True)

names_dict = None
predictions_test1 = []
itr = True

for value in results:
    if itr:
        names_dict = value[0].names
        itr = False
    predictions_test1.append(value.probs.data.numpy()) #Store prediction tensors value

predictions_test.extend(predictions_test1)




#predictions, names_dict
y_pred1 = []

predictions.append(predictions_test)
print(len(predictions_test))

for item in predictions_test:
    y_pred1.append(names_dict[np.argmax(item)])

map_dict = {value: key for key, value in names_dict.items()}

y_true1 = [map_dict[value] for value in y_true1]
y_pred1 = [map_dict[value] for value in y_pred1]

print(names_dict)
y_true.extend(y_true1)
y_pred.extend(y_pred1)

print(len(y_true1), len(y_pred1))
print(len(y_true), len(y_pred))
```

After that, the code performs prediction on the test data for fold1 using the previously validated model. It first creates an empty list images_path_test1 to store the paths of test images and another list y_true1 to store the true labels. The code then iterates through the 'test1' folder inside the 'fold1' directory, creates a dictionary indices_dict mapping class labels to indices, and appends the image paths and true labels to the respective lists.

Next, it creates an instance of the YOLO model loaded with the best weights from the previous step and uses it to make predictions on the test images. The predictions are stored in the predictions_test1 list as numpy arrays. The code iterates through the prediction results, extracting the class names dictionary names_dict.

After obtaining the predictions, the code converts the true labels and predicted labels from indices to their corresponding class names using the names_dict. It appends the true labels to y_true and predicted labels to y_pred lists, which are likely used for further evaluation or analysis. Finally, it prints the lengths of y_true1, y_pred1, y_true, and y_pred lists, as well as the names_dict.

```
cm = confusion_matrix(y_true, y_pred)
report = classification_report(y_true, y_pred, output_dict=True)

print('Accuracy: ', report['accuracy'])
print('Weighted Average Precision: ', report['weighted avg']['precision'])
print('Weighted Average Recall: ', report['weighted avg']['recall'])
print('Weighted Average f1-score: ', report['weighted avg']['f1-score'])
```

Accuracy: 0.999475272202545
Weighted Average Precision: 0.999475574987886

Weighted Average Recall: 0.999475272202545
Weighted Average f1-score: 0.9994752293491107

Finally, we calculated the confusion matrix and classification report for evaluating the model's performance on the test data. It computes the confusion matrix using the true labels (y_true) and predicted labels (y_pred). Then, it generates a classification report as a dictionary, which includes metrics like accuracy, precision, recall, and f1-score for each class and their weighted averages. Finally, it prints the overall accuracy, weighted average precision, weighted average recall, and weighted average f1-score from the classification report.