COMPUTER VISION AUTOMOBILE CLASSIFICATION USING VARIOUS DEEP CONVOLUTIONAL NEURAL NETWORK ARCHITECTURES

THESIS

RESEARCH PROJECT

Christensen Mario Frans - 2301963316

BINUS INTERNATIONAL UNIVERSITY

JAKARTA

BATCH 2023

COMPUTER VISION AUTOMOBILE CLASSIFICATION USING VARIOUS DEEP CONVOLUTIONAL NEURAL NETWORK ARCHITECTURES

THESIS

Proposed as a requirement to obtain a Bachelor's Degree in Computer Science

Christensen Mario Frans - 2301963316

BINUS INTERNATIONAL UNIVERSITY

JAKARTA

BATCH 2023

COMPUTER VISION AUTOMOBILE CLASSIFICATION USING VARIOUS DEEP CONVOLUTIONAL NEURAL NETWORK ARCHITECTURES

THESIS

Prepared by:

Christensen Mario Frans
2301963316

Approved by:

Supervisor

Nunung Nurul Qomariah, S.Kom., M.T.I., Ph.D.
Lecturer Code: D6211

BINUS INTERNATIONAL UNIVERSITY
JAKARTA
BATCH 2023

STATEMENT FROM THE BOARD OF EXAMINERS

We, the members of the Board of Examiners for the S-1 Thesis Defense,
Hereby declare that

CHRISTENSEN MARIO FRANS (2301963316)

Who presented an S-1 Thesis entitle

COMPUTER VISION AUTOMOBILE CLASSIFICATION USING VARIOUS DEEP CONVOLUTIONAL NEURAL NETWORK ARCHITECTURES

Successfully passed the S-1 Thesis Defense Examination conducted on 10 July 2023

		Name	Signature
1.	Chair	Ardimas Andi Purwita, S.T., M.T., Ph.D.	.....
2.	Member	Dr. Maria Seraphina Astriani, S.Kom., MTI.	.....
3.	Supervisor	Nunung Nurul Qomariyah, S.Kom., M.T.I., Ph.D.	.....

JWC Campus
Jl. Hang Lekir I No.6
Senayan, Jakarta 10270
Indonesia

t. +6221 720 2222, 720 3333
f. +6221 720 8569, 720 5555
e. inquiry-jwc@binus.edu
international.binus.ac.id

EXCLUSIVE RIGHT STATEMENT

Dengan ini, saya/kami,
With this, I/We,

Nama (*Name*):
Christensen Mario Frans

NIM (*Student ID*):
2301963316

Judul Tesis (*Thesis Title*):
Computer Vision Automobile Classification Using Various Deep Convolutional Neural Network Architectures

Memberikan kepada Universitas Bina Nusantara hak non-eksklusif untuk menyimpan, memperbanyak, dan menyebarluaskan tesis saya/kami, secara keseluruhan atau hanya sebagian atau hanya ringkasannya saja, dalam bentuk format tercetak atau elektronik.

Hereby grant to my/our school, Bina Nusantara University, the non-exclusive right to archive, reproduce, and distribute my/our thesis, in whole or in part, whether in the form of a printed or electronic format.

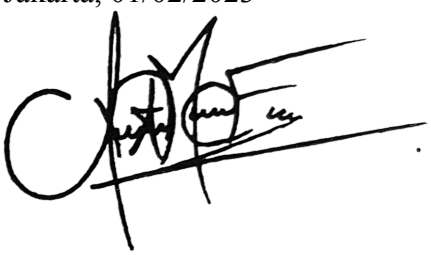
Menyatakan bahwa saya/kami, akan mempertahankan hak exclusive saya/kami, untuk menggunakan seluruh atau sebagian isi tesis saya/kami, guna pengembangkan karya di masa depan, misalnya dalam bentuk artikel, buku, perangkat lunak, ataupun sistem informasi.

I/We acknowledge that I/we retain exclusive rights of my/our thesis by using all or part of it in a future work or output, such as an article, a book, software, or information system.

Catatan: Pernyataan ini dibuat dalam 2 (dua) bahasa, Indonesia dan Inggris, dan apabila terdapat perbedaan penafsiran, maka yang berlaku adalah versi Bahasa Indonesia.

Note: This Statement is made in 2 (two) languages, Indonesian and English, and in the case of a different interpretation, the Indonesian version shall prevail.

Jakarta, 01/02/2023



Christensen Mario Frans
2301963316

EXCLUSIVE RIGHT STATEMENT

Dengan ini, saya/kami,
With this, I/We,

Nama (*Name*):
Christensen Mario Frans

NIM (*Student ID*):
2301963316

Judul Tesis (*Thesis Title*):
Computer Vision Automobile Classification Using Various Deep Convolutional Neural Network Architectures

Memberikan kepada Universitas Bina Nusantara hak non-eksklusif untuk menyimpan skripsi karya saya, secara keseluruhan atau hanya sebagian atau hanya ringkasannya saja, dalam bentuk format tercetak atau elektronik.

Hereby grant to my/our school, Bina Nusantara University, the non-exclusive right to archive my/our thesis, in whole or in part, whether in the form of a printed or electronic format.

Menyatakan bahwa saya tidak akan menggunakan seluruh atau sebagian isi skripsi saya, dalam bentuk apapun juga termasuk dalam penelitian karya ilmiah saya selanjutnya.

I/We acknowledge that I/we will not use all or part of it in a future work or output, in any form, including for my next research.

Catatan: Pernyataan ini dibuat dalam 2 (dua) bahasa, Indonesia dan Inggris, dan apabila terdapat perbedaan penafsiran, maka yang berlaku adalah versi Bahasa Indonesia.

Note: This Statement is made in 2 (two) languages, Indonesian and English, and in the case of a different interpretation, the Indonesian version shall prevail.

Jakarta, 01/02/2023

Christensen Mario Frans
2301963316

BINUS INTERNATIONAL UNIVERSITY

Bachelor's Degree in Computer Science

2023 Even Semester

Computer Vision Automobile Classification Using Various Deep Convolutional Neural Network Architectures

Christensen Mario Frans - 2301963316

ABSTRACT

This research entails the general & technical overview of modern Machine Learning algorithms, as well as how it is able to potentially benefit human beings by assisting real-world problems. The main objective in this research is to prove that Computer Vision & Deep Learning is able to classify automobiles, in order to provide automation and replace the manual labor in categorizing cars, which is essential for a more efficiently designed parking layout to reduce parking congestion.

This project incorporates multiclass image classification on different car types, using multiple Deep Convolutional Neural Networks (DCNN). The architectures experimented in this paper includes the ResNet-50, Inception V3, DenseNet 201, Xception, a Custom-Built DCNN, as well as another DenseNet 201 pre-trained with ImageNet weights (Transfer Learning). Hence, the performances of these models will be evaluated upon the experimentations.

Among the 6 different DCNN architectures, the DenseNet 201 model is able to achieve the best performance in terms of Accuracy and F1 Scores of 76% and 0.71 respectively. Additionally, this research has also concluded that removing backgrounds from the dataset images, or in other words removing unnecessary noise from our input features, does have a significant positive impact on the models' performances and their training times.

The outcomes yielded from the conducted experiments has proven that Deep Learning is able to successfully classify various automobile types. Thus, it could be helpful in automating and replacing the manual labor needed in categorizing cars, for the alternative parking layout that is designed to improve the efficiency of space allocations for modern parking lots. Hence, Computer Vision could indirectly contribute to decreasing parking congestion.

Key Words

Computer Vision, Deep Learning, Deep Convolutional Neural Network, Transfer Learning

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Background & Motivation

Automotive drivers experienced trouble looking for parking spots at least once in their lifetimes. According to USA Today, drivers spend on average about 17 hours every year just to look for parking spots. The numbers are much higher for larger cities, such as New York, where drivers spend a staggering average of 107 hours. [1] This results in additional costs for both the driver and environment aside from just time alone; more fuel consumed & purchased, emissions, traffic congestions, and many more.

This leads to the core motivation for this research. To prevent parking congestion, an efficient alternative would be to carefully study the cars that are entering these parking areas, and segment them into different categories based on their sizes. Then, we can segregate parking spots for smaller and larger cars, just like how we typically separate lots for motorbikes and cars. Hence, smaller cars will find more available lots; such as lower ceiling areas, shorter available distances from curbs, tighter backout path widths, & narrower corners. On the other hand, the saved spaces could be used for larger cars to park. Therefore, the final layout will leave more available parking spaces for all.

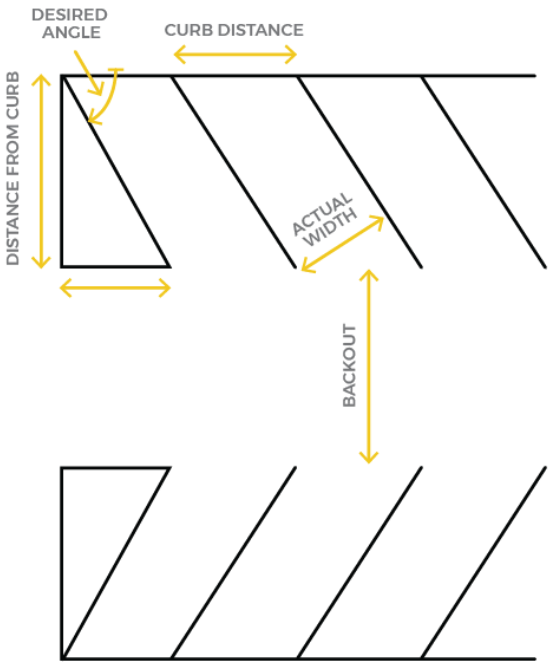


Figure 1.1: Typical Car Park Layout & Crucial Measurements [2]

Unfortunately, this method still requires a manual process in labeling and directing all vehicles that are entering the building, leading to additional costs of hiring staff. Moreover, it is impractical to completely rely on employees, as it is an exhausting yet monotonous task to standby & tell drivers where to park for hours straight; potentially causing human errors. Alternatively, a sign could be placed in the entrance so drivers can determine themselves where to park, to prevent the need for manual labor. However, some drivers are not so familiar with automobiles, thus this will rather confuse them, or cause them to park in the wrong section.

Specifically, this idea could be implemented with the help of Machine Learning. Instead of only reading car plate numbers and handing out parking tickets, the camera in the entrance booth could also be used to classify every car entering the building using Computer Vision. Utilizing the car image captured by the camera as an input, the trained Deep Learning model predicts and tells the drivers which category their vehicle belongs to, alongside its type's dedicated parking sections.

Computer Vision is a field of Artificial Intelligence (AI) that aims to provide computers the ability to interpret and understand visual data, such as images and videos. [3]

Deep Learning is a type of Machine Learning that involves the use of Artificial Neural Networks (ANNs) that are designed to mimic the way human brains work, allowing them to "learn" from large amounts of unstructured data; such as images, just like human beings. Hence, Deep Convolutional Neural Networks (DCNNs) are simply ANNs with Convolutions that geometrically enhance its learning capabilities. [4]

1.2 Scope

In order to incorporate Computer Vision to assist the above-mentioned more efficiently designed parking layout, it is necessary to first prove that it is able to visually classify different automobile types. Ideally, more than one Machine Learning algorithm shall be considered for prototyping and experimentation. Therefore, this research entails the performance comparison between 6 different

Deep Convolutional Neural Networks (DCNN); 5 base and 1 Pre-Trained (transfer learning) models, in performing multiclass car types classification. The following DCNN architectures are used in this research:

1. ResNet50
2. Inception V3
3. DenseNet 201
4. DenseNet 201 (Pre-Trained Imagenet)
5. Xception
6. Custom-built DCNN Architecture

The data used to train and evaluate these models are images of multiple different cars from the Stanford AI Cars Dataset. Initially, the data consisted of 16185 car pictures, divided into 196 different classes that were named after the *year-make-model* of the cars; for example: “2012 Tesla Model S”, or “2012 BMW M3 Coupe”. These categories are separated into an approximately equal amount of train & test portions; about 8090 images each. [5] Since we would like our Machine Learning models to classify car body types instead of *year-make-model*, we then modified the dataset into such categories instead, resulting only the following 7 classes:

1. Sedan
2. Sport
3. SUV
4. MPV
5. Hatchback
6. Wagon
7. Truck

1.3 Aims

This research aims to achieve the following:

1. Build and evaluate which amongst the 6 DCNN architectures experimented in this research; ResNet50, Inception V3, DenseNet 201, DenseNet 201 (Pre-trained Imagenet), Xception, and Custom-Built DCNN Architecture, is able to achieve the best classification results possible, or in other words the best performing model.

2. Train & predict the models on 2 types of dataset:
    a. Image data without background removed
    b. Image data with background removed

    Evaluate how removing backgrounds of the image data has impact on each of the model's performances

1.4 Benefits

This project proposes a solution utilizing Computer Vision to automate the process of manually labeling cars, which is essential for the alternative parking layout design for modern parking lots, in order to improve the efficiency of space allocations and thereby to reduce parking congestion. Therefore, this research conveys how Deep Learning could indirectly contribute to reducing parking congestion, which carries many benefits to its stakeholders & the environment; less fuel consumed & purchased, emissions, time drivers spent to look for a parking spot, and many more.

On top of that, the whole process can be fully automated; by reverse engineering the OCR softwares that are already installed in typical parking lots to automatically read car plates, to also capture the vehicle's image and pass them into a trained Machine Learning model which predicts the classes of the entering cars. Therefore, it is likely that not much additional expenses in hardware equipment or software modification is needed.

Finally, the Deep Learning models developed in this research could be further enhanced by retraining it with more car pictures if needed; car manufacturers typically release new models every few years, therefore it is preferable for the data to always be updated to maintain or even improve its performance levels.

1.5 Structure

This research is broken down into different chapters, each having their own respective agenda, as described in a nutshell below:

- Chapter 1: Introduction
    - This chapter entails the project's background & motivation, problem, proposed solution, scope, aims, and its potential benefits to stakeholders.

- Chapter 2: Theoretical Foundation
    - This section elaborates the theories behind our research; providing detailed physical characteristics of each car body type, and the general & technical overview of Machine Learning; including Deep Learning for classification, Deep Convolutional Neural Networks (DCNNs), Transfer Learning, common performance evaluation metrics, as well as some real-world examples of DCNNs.
- Chapter 3: Problem Analysis
    - This part further demonstrates the problem, related work, proposed solution incorporating Machine Learning, and finally how Deep Learning can potentially contribute to the real world based on the experiments in this research.
- Chapter 4: Solution Architecture
    - This chapter reveals and illustrates the end-to-end solution design; including data collection & organization, data preprocessing, selected DCNN architectures alongside their training methodologies, as well as the evaluation metrics that will be used to measure their performances.
- Chapter 5: Result Analysis
    - Upon experimentation, this segment will thoroughly elaborate each of our models' prediction outputs using various evaluation metrics.
- Chapter 6: Discussion
    - This section analyzes & compares the results yielded by different DCNN architectures (from Chapter 5) amongst one another. Hence, some of the aims of this research will also be justified; such as the top performing model, and the impact of removing backgrounds on the dataset.
    - Additional discussion will also be included for other experimentations conducted during the development of this project, that were not selected for our final modeling; DCNN architecture alternatives, hyperparameters considered, image preprocessing techniques, etc.
- Chapter 7: Conclusion
    - This final chapter concludes the crucial observations of the project, proving that it has achieved the research's main objectives, and most

importantly how Computer Vision & Deep Learning can potentially benefit stakeholders by supporting efficient parking layouts to reduce parking congestion.

- It will also suggest any further improvements for the project that could be made for future research.

CHAPTER 2: THEORETICAL FOUNDATION

2.1 Car Body Types & Sizes

Since its very first invention in 1886, automobiles have revolutionized the world's transportation systems, providing the ability to transport both passengers and goods from one place to another much more conveniently; without needing constant manpower or animals to keep the carriage moving, while providing the driver with full control of the speed, acceleration, and direction of the cabin. Moreover, the private vehicle could be parked right in front of any suburban home, providing owners with easy access to transportation at any time. As a result, when gasoline-powered cars, such as the Ford Model T, began its mass production in the early 20th century, it quickly gained popularity and replaced primitive non-engine vehicles; including animal-drawn carriages, bicycles, and rickshaws. [6]

Figure 2.1: The World's First Car [7]

As demand kept surging for such vehicles over the next few decades, cars were not only used to transport people, but also vast amounts of freight from one place to another, especially during the World War II era. This paved the way for the engineering of different car sizes serving different purposes.

Modern cars are built into many different types, such as Sedans, Hatchbacks, Wagons, Sport Cars, Multi-Purpose Vehicles (MPVs), Sport Utility Vehicles (SUVs),

Trucks, and many more. Each car type has its own unique functionality built superior to others; for instance, sport cars could race at much higher speeds but with less passenger capacity (usually two), whereas SUVs are built withstand wild offroad conditions and are much larger in size, and luxury sedans are often used to chauffeur top executives at supreme comfort.

Above all superiority, the simplest way to distinguish between these vehicle types are actually their sizes and dimensions. The picture below illustrates a typical car dimension measurements; length, height, width, and wheelbase:

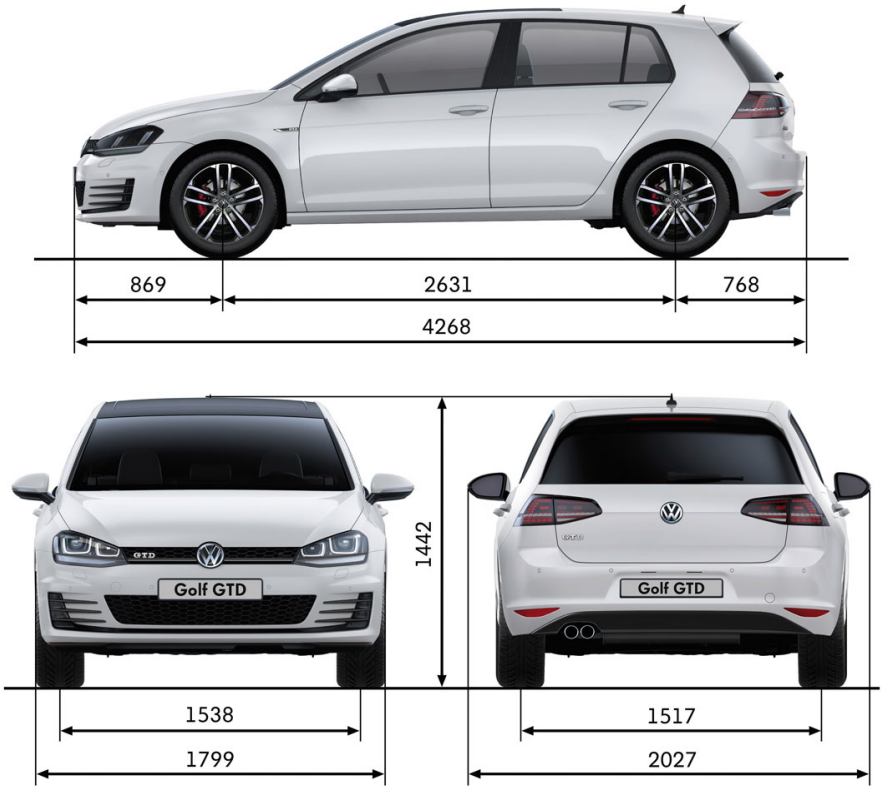


Figure 2.2: Typical Car Dimension Calculations [8]

Note: For the purpose of this research, we will only be looking at the 3 main dimensions; length, height, and width. This is because their measurements will be the main identifier of the car's physical size for parking lots, unlike wheelbase lengths.

Moving forward, to mathematically get the physical size of a car, relative to its length, width, and height, we could simply calculate its volume. This will be the perceived total space needed by the car to park, as visualized below:

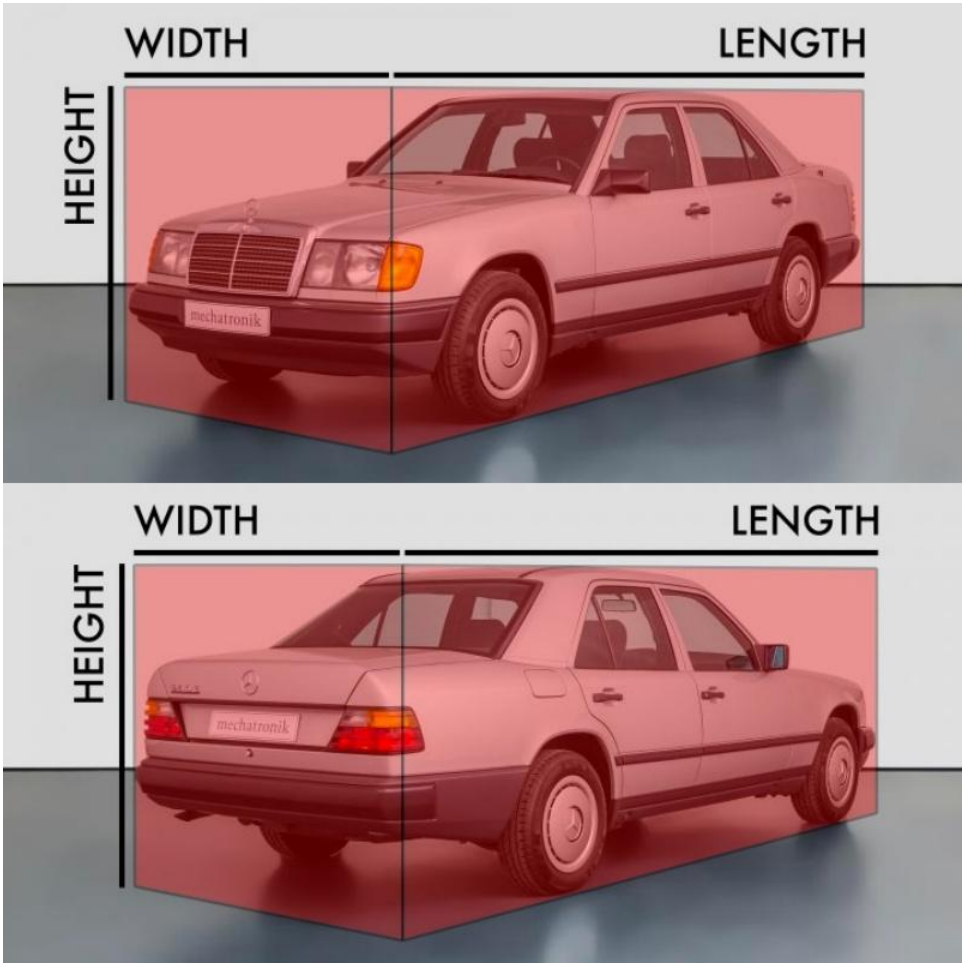


Figure 2.3: Car Volume [9]

The table below depicts the dimensions of different car models, corresponding to their body types:

Table 2.1: Different Car Dimensions

Type	Sub-Type	Model Example	Length (cm)	Height (cm)	Width (cm)	Volume (m^3)
Sedan	Compact	Honda City 2021	454.9	148.9	174.8	11.84
Sedan	Saloon	BMW 5 Series 2022	496.3	146.7	186.8	13.60
Sedan	Full-Sized	Mercedes-Benz S-Class 2022	528.9	150.3	210.9	16.77
Hatchback	Sub-Compact	Fiat 500 Abarth 2019	366.7	149.1	162.8	8.90
Hatchback	Compact	Volkswagen Golf 2021	425.7	147.8	179.8	11.31

Wagon	Medium-Sized	Chevrolet HHR 2010	440.5	157.75	172.75	12.04
Wagon	Station Wagon	Mazda 6 Estate 2020	480	145	184	12.81
SUV	Compact	Maserati Levante 2019	500.3	167.9	196.8	16.53
SUV	Medium-Sized	BMW X7 2019	515.1	180.5	200	18.60
SUV	Full-Sized	Cadillac Escalade ESV 2022	576.6	194.1	206	23.06
Sport Car	Roadster	Mazda MX-5 2021	391.5	122.5	173.5	8.32
Sport Car	Convertibles	Ferrari Portofino 2018	458.6	131.8	193.8	11.71
Sport Car	Supercar	McLaren 720S 2017	454.3	119.6	205.9	11.19
MPV	Minivan	Toyota Kijang Innova 2020	473.5	179.5	183	15.55
MPV	Van	Mercedes-Benz V-Class 2020	514	188	192.8	18.63
MPV	Cargo Van	Mercedes-Benz Sprinter 2021	593.2	233.1	202	27.93
Truck	Medium-Sized	Nissan Frontier 2015	522	178.1	184.9	17.19
Truck	Full-Sized	Ford F-450 Super Duty 2022	676.2	208.5	243.8	34.37

[10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27]

We can summarize the table above by taking the average dimensions and volumes of each car body type, resulting the new data below:

Table 2.2: Average Car Type's Dimensions

Type	**Average Length (cm)**	**Average Height (cm)**	**Average Width (cm)**	**Average Volume (m^2)**
Hatchback	396.2	148.45	171.3	10.1
MPV	526.9	200.2	192.6	20.71
SUV	530.67	180.83	200.93	19.39
Sedan	493.37	148.63	190.83	14.07
Sport Car	434.8	124.63	191.07	10.41
Truck	599.1	193.3	214.35	25.78
Wagon	463.75	152.75	179.75	12.71

As shown in the table above, different car body types do take up varying amounts of space. On average, Trucks require the largest amount of space, followed by MPVs, SUVs, and Sedans consecutively. Alternatively, Hatchbacks, Sport Cars, and Wagons have the least amount of volumes.

2.2 Machine Learning

Machine learning is a field of Artificial Intelligence (AI) that involves the use of algorithms and statistical models that enable computers to automatically learn and improve from data it is fed and trained, without the need of being explicitly programmed, nor interventions from any humans. Usually, the more data being trained, the better the performance of these AI models. [28]

The history of Machine Learning dates all the way back to the early 1940s, when the first mathematical model of Neural Networks was described by Walter Pitts and Warren McCulloch in a scientific paper titled "A Logical Calculus of the Ideas Immanent in Nervous Activity". In 1949, Donald Hebb's published a book, “The Organization of Behavior”, containing theories about the relationship between the biological human brain behavior and Neural Networks, which became an important foundation for the development of Machine Learning in the upcoming decade. Just one year later, Alan Turing invented the Turing Test; a scientific method to check whether or not a computer has actual intellectual capabilities, by programming it to trick a human into believing that it is also human. Soon after, Turing presented this principle in his paper “Computing Machinery and Intelligence”, beginning with the question: "Can machines think? [29]

Eventually in 1952, IBM computer pioneer Arthur Samuel developed the first ever computer learning program to play the game of checkers. The IBM computer that ran the program gradually improved in performance the more it played the game; analyzing winning strategies and using them for future games. Five years later, an American psychologist named Frank Rosenblatt, designed the perceptron, which was the first ever Neural Network for computers that is able to simulate the thought processes of a typical human brain. The next major advancement in Machine Learning occurred in 1967 with the development of the "nearest neighbor" algorithm, which allowed computers to study basic pattern recognition. This algorithm could be used to map the most efficient routes for a traveling salesman, ensuring that all cities are visited during a short tour, while beginning from a random location. In 1979, a group of students at Stanford University created the "Stanford Cart," a self-driving vehicle which was able to navigate around a room while

avoiding obstacles on its own. Just a couple years later, Gerald Dejong came up with the Explanation Based Learning (EBL) concept, allowing a computer to analyze training data and thereby generating a general rule to follow while discarding any unimportant data. [29]

It wasn't until the 1990s that the foundation of Machine Learning models shifted from knowledge-based to data-driven instead, and scientists began to gather large amounts of data to feed and train the programs they have built. Later that decade, IBM's Deep Blue surprised the globe when it successfully beat the world champion at chess. [29] The innovation carried on for the next couple decades, leading to where it stands today.

Although the term "Machine Learning" was invented as early as the 1950s by Arthur Samuel, it did not take off until the past 3 decades when the term started becoming familiar in public ears. This is because it still took decades of research to enhance the high-level algorithms developed by experts from all around the world, let alone readily implemented to assist our everyday lives. [30] Moving forward to the present days, Machine Learning has never been more involved in assisting humans and corporations. Some of its common applications include Computer Vision, Speech Recognition, Natural Language Processing, Predictive Analytics, Fraud & Email Spam Detection, and many more. Even so, Machine Learning algorithms and architectures are still being improved by researchers up to this date, in hopes to innovate the subject even deeper and thereby come up with even more powerful models that are able to assist more complex tasks.

In general, there are 3 different categories of machine learning; among which are:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

Supervised Learning

Supervised Learning involves training a model on a labeled dataset, where the correct output is provided for each particular entry in the training set. Thus, this is

where the terminology “supervised” comes from. The model is able to generate predictions upon discovering geometrical relationships or underlying patterns between this input-output mapping. [31] The most commonly heard examples of Supervised learning are Regression and Classification.

Regression is used to forecast a continuous output; such as prices, sales volume, or probability, where the output is a real-valued number. The goal is to minimize the sum of squared/absolute differences between the predicted and true values as much as possible. Common techniques for regression include Linear & Polynomial regression. [32]

Classification is used to predict a categorical output, such as a label or a class, where the output is a discrete value. The goal is to predict the correct class for each entry in the dataset, by attempting to attain the highest possible accuracy score during training. Common techniques for classification include Decision Trees, Logistic Regression, K-Nearest Neighbors, and Support Vector Machines. Classification is commonly implemented for labeling or flagging tasks; such as Spam & Fraud Detection, Patient Illness Diagnosis, Customer Churns, and many more. [33]

Unsupervised Learning

Unsupervised learning involves training a model on an unlabeled dataset, where the model attempts to find patterns and similarities between the data, and thereby create artificial segmentations and clusters, without the need of any guidance. [34] Unlike Supervised Learning, the goal of Unsupervised Learning is to discover the inherent structure and relationships in the data, rather than to make specific predictions or decisions. [35]

Reinforcement Learning

Reinforcement learning is a type of machine learning in which an agent learns to interact with an environment using trial-and-error, and thereby determine a policy with a goal of maximizing its cumulative reward over time. [36]

Below are the basic components that elaborates the Reinforcement Learning problem [36]:

- Environment: The virtual surroundings in which the agent is able to simulate and interact with.
- State: The agent's current condition.
- Reward: Environmental feedback based on the agent's action/s.
- Policy: A technique to map an agent's state towards their future actions.
- Value: The potential benefit that an agent would gain upon performing an action in a certain state.

2.3 Computer Vision

Computer Vision is a field of Artificial Intelligence (AI) that aims to provide computers the ability to interpret and understand visual data, such as images and videos. This allows them to extract useful information from these sources as well as to make crucial decisions or recommendations. [3]

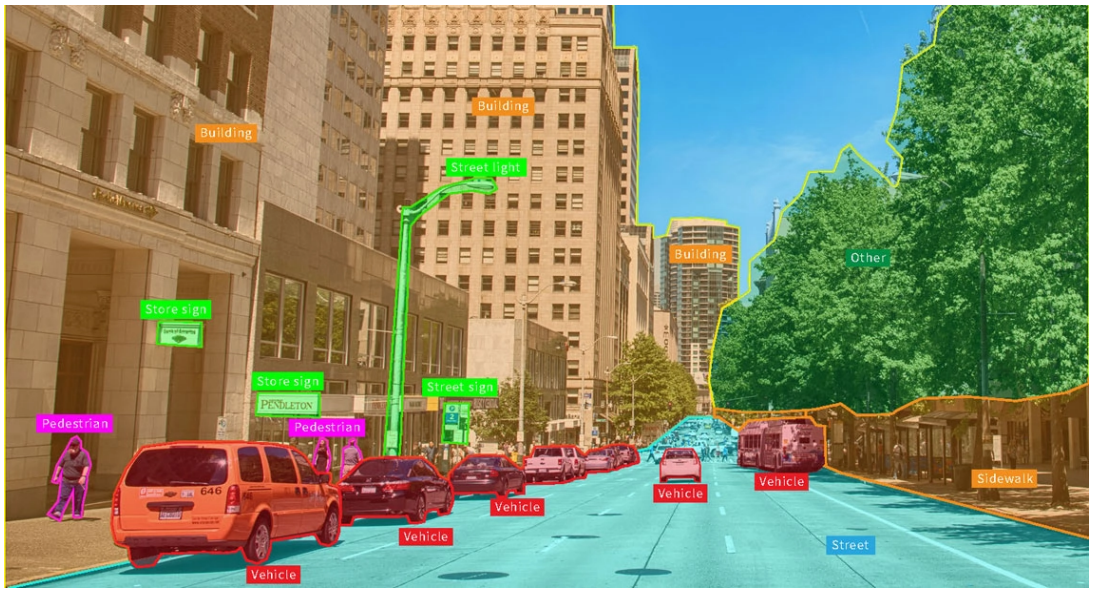


Figure 2.4: Object Detection with Computer Vision [37]

Computer vision relies heavily on the use of large amounts of data to train and improve its performance. This process involves repeatedly analyzing the data and using it to learn the subtle differences between various objects or images. For instance, to teach a computer to classify fruits, it would need to be fed with

numerous images of different fruit types, so that it can learn to distinguish one fruit from another, even if it has many visible defects. [3]

Computer Vision is used to perform a variety of tasks, including [3]:

1. Image Classification; which analyzes an image and assigns it to a specific class or category. For example, a social media firm might use this technology to automatically identify and remove inappropriate images uploaded by users.
2. Object Detection; which entails the use of image classification to identify a specific object and then counting the number of times it appears in an image or video. This could be used to detect defects on an assembly line or identify machinery that needs maintenance.
3. Object Tracking; this involves following or tracking an object once it has been detected. This task is often performed using a sequence of images or a real-time video recording. Such applications are beneficial for autonomous vehicles, which need to track objects like pedestrians, other cars, or road infrastructure in order to avoid collisions and obey traffic regulations.
4. Content-Based Image Retrieval; this consists of using Computer Vision to search and collect images from a large database, however, based on the features of the pictures rather than their associated metadata tags. This task may also involve automatic image labeling, which replaces traditional manual tagging on collected data. This technique is widely used in digital asset management systems and can improve the accuracy of search and retrieval.

Images Representations in Computers

Modern images are represented in computers as pixels, with a specific color assigned to each of them. The calculation for total number of pixels present on an image is practically straightforward, which is simply multiplying together the number of pixels horizontally and vertically. For example, an image with width and height of 100 px and 200 px respectively will have a total of 20,000 pixels. [38]

Alternatively for colors, since modern computers are able to read up to 16.7 million different colors, it will be extremely resourceful to store each of these different colors directly. Therefore, each possible color of a pixel in the image is defined by the combination of main colors with different intensity levels instead, translated into hexadecimal codes. The most commonly used model defining the color of each of these pixels is known as the RGB; which is a combination of the 3 primary colors: Red, Blue, & Green. Each of these main colors can have 256 different values of intensity (a range of between 0 to 255 inclusive). [38]

	Red	Green	Blue	Hexadecimal code
	0	0	0	#000000
	255	255	255	#FFFFFF
	255	0	0	#FF0000
	0	255	0	#00FF00
	0	0	255	#0000FF
	255	128	0	#FF8000
	255	255	0	#FFFF00
	128	128	128	#808080

Figure 2.5: RGB Model Table Translated into Hexadecimal Codes [38]

2.4 Deep Learning

Deep Learning is a type of Machine Learning that involves the use of Artificial Neural Networks (ANNs) with three or more layers. These networks are designed to mimic the way human brains work, allowing them to "learn" from large amounts of data just like human beings. [4]

The term “Deep Learning” was first introduced in the year 2006 by a British-Canadian cognitive psychologist and computer scientist, in order to describe the newly invented Neural Network architecture that allows computers to distinguish between objects and texts in images or videos. [29]

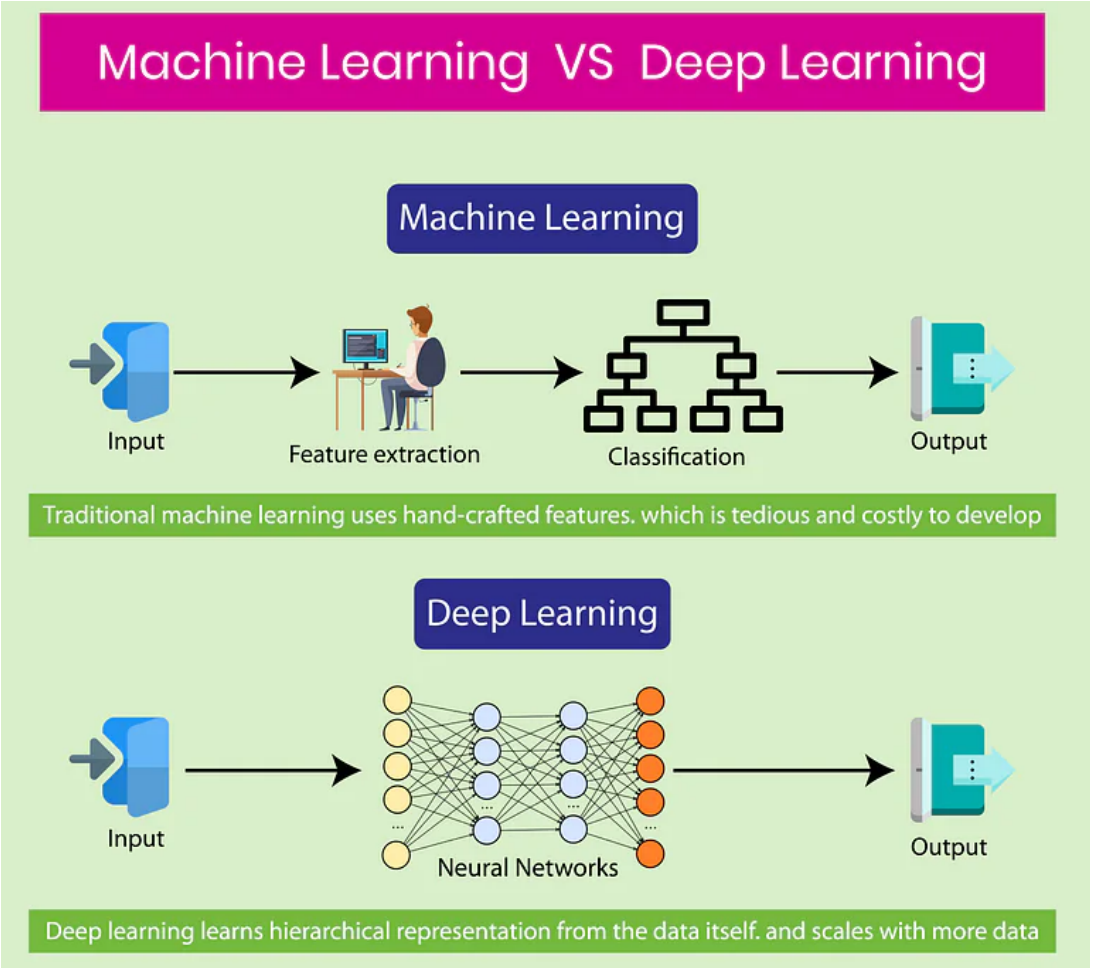


Figure 2.6: Machine Learning Versus Deep Learning [39]

Deep Learning differs itself from classical Machine Learning in terms of the type of data it works with and the methods it uses to learn. While Machine Learning algorithms typically work on structured & labeled data organized into tables, Deep Learning algorithms have the ability to process unstructured data such as text and images without requiring any pre-processing. These algorithms are also able to automate the feature extraction process, reducing the need for labeling features manually. Through the use of Gradient Descent, Forward & Backward Propagation, Deep Learning algorithms are able to automatically adjust and optimize themselves for accuracy and precision. While a Neural Network with a single hidden layer can still make predictions, adding more hidden layers will usually help to improve the accuracy of their predictions. [4]

As an example, given a set of pictures of different pets, which needs to be categorized as ‘cat’, ‘dog’, ‘hamster’, and etc. A Deep Learning algorithm could

determine which specific feature; such as ears, mouth, or eyes, that is the most crucial in distinguishing the different animals, and use this information to make future predictions for a new picture of an animal. [4]

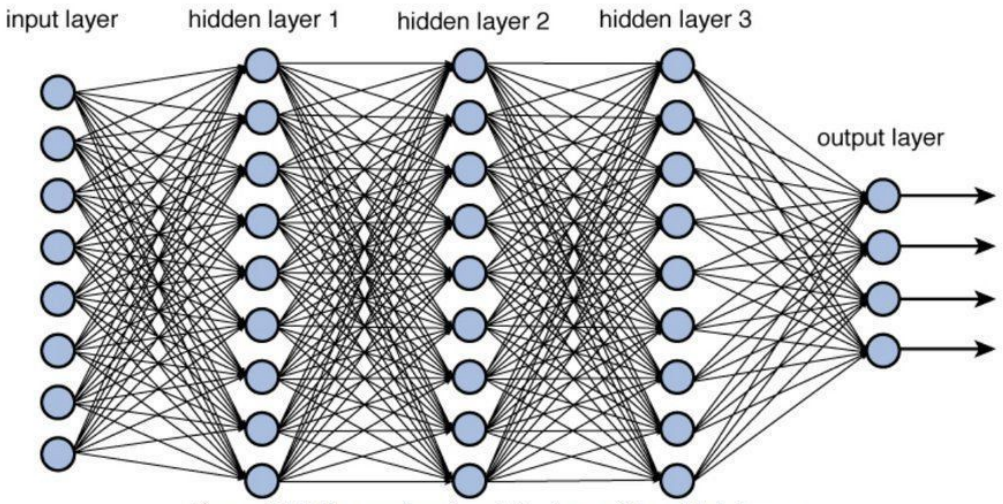


Figure 2.7: A Typical Deep Neural Network Architecture [40]

Although the illustration above represents a simple layout of a typical Deep Neural Network, these architectures can become very complex, and there are various types of Artificial Neural Networks designed to assist specific problems or datasets. For instance, Convolutional Neural Networks (CNNs) are often used in Computer Vision and image classification tasks, while Recurrent Neural Networks (RNNs) are primarily used in Natural Language and Speech Recognition problems, since it also takes into account the sequences and time-series behavior of the data. [4]

2.4.1 Forward & Backward Propagation

As shown in the figure above, Deep Neural Networks are made up of multiple layers of interconnected nodes. The input and output layers of a Deep Neural Network are known as ‘visible layers’, whereas the layers in between them are called ‘hidden layers’. The input layer is where the model receives the data for processing, and the output layer is where the final prediction or classification is generated. Hence, the computational process moving through the network is known as Forward Propagation. Another process involved in Deep Neural Networks is called Backward Propagation, which utilizes algorithms such as the infamous Gradient Descent, to compute prediction errors and thereby use it to adjust the Weights and Biases of its

preceding nodes to minimize this error value in the future, hence moving in reverse through the layers. Together, both Forward and Backward Propagation allows the Neural Network to generate predictions and optimize the error amounts automatically, leading to an increase in its performance over more training iterations. [4]

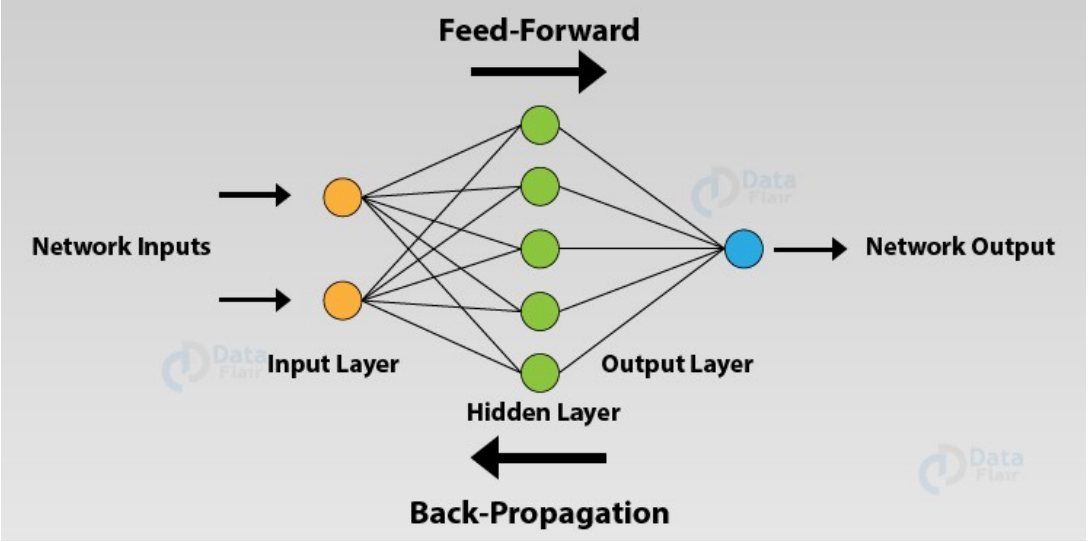


Figure 2.8: Forward & Backward Propagation Illustration [42]

2.4.2 A Single Neuron

As mentioned before, Artificial Neural Networks (ANNs) are constructed with three to many layers, each typically consisting of multiple nodes per layer. These nodes are scientifically referred to as Neurons, which mathematically functions to mimic each biological neuron activity in human brains. [43]

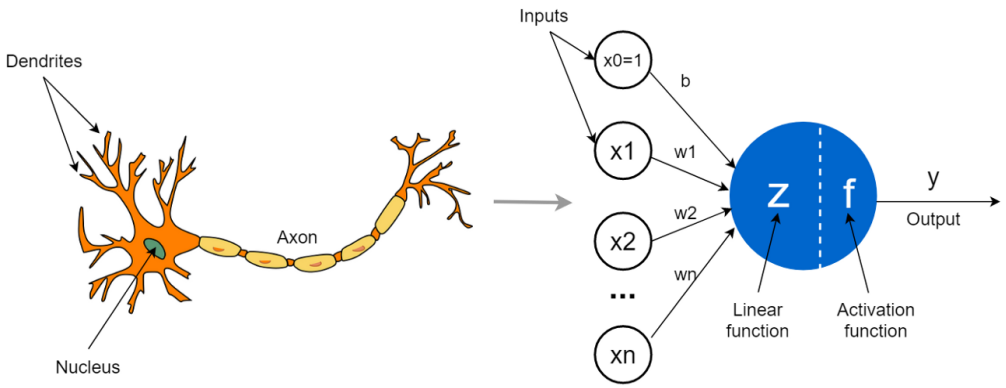


Figure 2.9: DCNN Nodes Compared to Biological Brain Neurons [41]

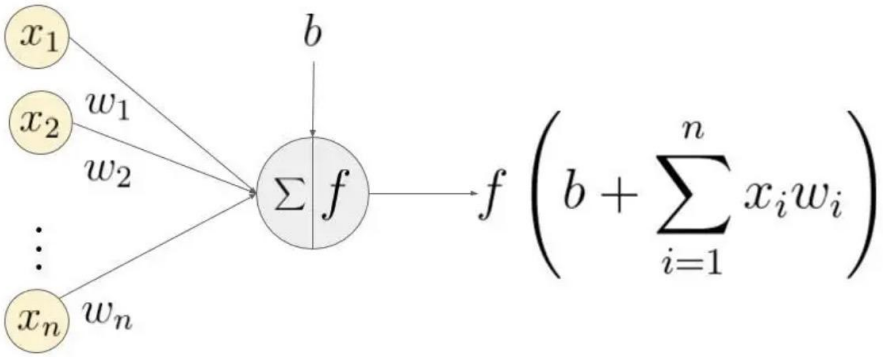
There are a few main components in an artificial Neuron; which are Weights, Biases, and an Activation Function. The goal of having these three components is to find a nonlinear relationship among the input values that fits best for the overall model, by transforming them numerically.

Weights can be thought of as a coefficient that measures the importance of the input value (which could be from a numeric input source or another Neuron in the previous layer) passed into the Neuron. [44] On the other hand, Bias is a constant value added to this weighted input, prior to it being passed into the Activation Function. Therefore in simpler terms, and with respect to the Activation Function, the weight determines the effectiveness of the input, whereas the bias is a constant that acts as an intercept to adjust its output values. [45]

Altogether, a Neuron works by computing the weighted average of the input value it receives, which the sum of this is passed through the nonlinear Activation Function to yield an output value. [43] Finally, as mentioned before, the Weights and Biases will be fine-tuned automatically through Forward & Backward Propagation over the training iterations, in order to optimize the error values of the architecture's predictions.

2.4.3 Activation Functions

As mentioned in the previous paragraph, Activation Function transforms the aggregated input value to generate an output value that will be used as an input for a Neuron in the next layer. This computation process carries on until it reaches the final output layer, revealing the final predicted values. [43] The image below depicts how a Neuron computes its output:



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

Figure 2.10: Function Depicting how a Neuron Computes an Output [41]

There are multiple types of Activation Functions, such as Sigmoid, Tanh, Softmax, Rectified Linear Unit (ReLU), Exponential Linear Unit (ELU), and more. Each of these functions are tailored to solve specific problems. For instance, the Sigmoid and Tanh functions are typically used in the final layer of a Neural Network for Binary Classification, whereas Softmax could be adopted for both Binary & Multiclass Classification. On the other hand, Regression problems could either utilize the ReLU, or may not need any Activation Function at all in its final layer. Regardless, the ReLU is the most commonly used Activation Function within the hidden layers.

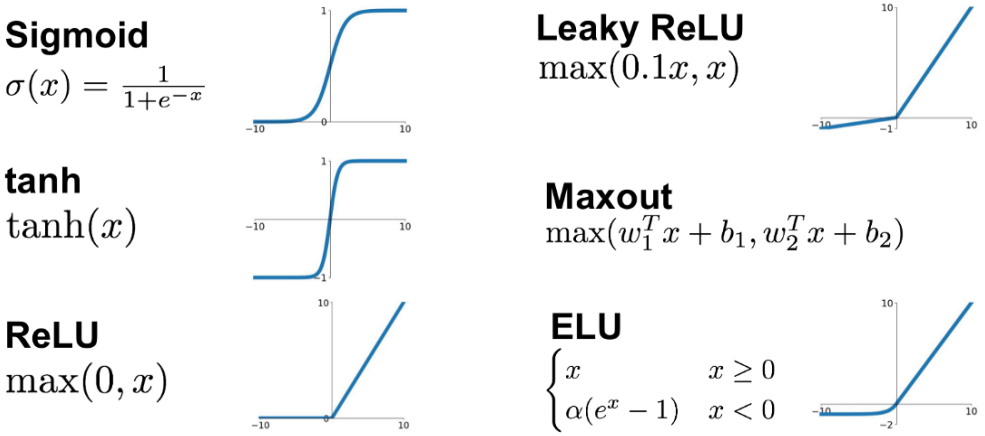


Figure 2.11: Formula & Graphs of Each Activation Function [46]

Another reason for having multiple Activation Functions is because some functions have their own benefits over one another. For example, one of the most well known

functions for Binary Classification, the Sigmoid, converts the input into an output value within a range of 0 to 1 of its ‘S’ shaped curve. This is reasonable if it were to be used in the final prediction layer, as its range directly maps the probability of a binary outcome. However, if it’s also used throughout the hidden layers, it will cause an issue within the Neural Network known as the Vanishing Gradient problem. This is because during Backward Propagation, the gradient of the Loss Function needs to be computed to minimize the error value, and thereby to adjust the Weights and Biases. With the Chain Rule of partial derivatives, this formula could be represented as the product of all the Activation Functions’ gradients, thus the updated Weights of each node will now be dependent on them. Yet, the derivative of the Sigmoid function could only reach a maximum value of 0.25, and when there are more layers with this Activation Function, this means that their derivatives will be multiplied with one another. Mathematically, when a small number less than 1 is multiplied with another small number less than 1, it yields an even smaller number. Hence, this is where the name “Vanishing Gradient" comes from; the product value will keep depleting and eventually approaching 0. This computational flaw leads the model to learn at a slower rate, or in some extreme cases stopping it from learning. Alternatively, the ReLU Activation Function could be used within the hidden layers instead. [47]

2.4.4 Loss Functions

A Loss Function in a Neural Network is a mathematical function returning a real number that represents the overall errors of its prediction results, or in other words a method to judge how well the model performs, on a certain training iteration. Over each training iteration, this Loss Function output is utilized by the Deep Learning architecture to optimize its Weights & Biases for the next iteration, with the goal of decreasing the prediction’s error value in the future. As mentioned in earlier sections, this process is accomplished through Forward & Backward Propagation, as well as Gradient Descent. Likewise, there are multiple Loss Functions available to use depending on the model’s task, as depicted below [48]:

- Regression:
    - Mean Squared Error (MSE)
    - Mean Absolute Error (MAE)

- Binary Classification:
    - Binary Cross-Entropy
    - Categorical Cross-Entropy
    - Sparse Categorical Cross-Entropy
- Multiclass Classification:
    - Categorical Cross-Entropy
    - Sparse Categorical Cross-Entropy

2.4.5 Gradient Descent

A gradient is a slope of a curve at a given point of a function. Gradient Descent is an iterative optimization mechanism implemented to locate the minimum or maximum of a certain function. Given a function, the algorithm iteratively computes the derivative of the next point using the slope value from its current point, scales it based on its learning rate, and then either subtract or add this calculated value from the current position depending on whether it is looking for the minimum and maximum point respectively. The mechanism repeats until it reaches a convergence point, where the derivative/slope value is the least. This is because the turning points of a function will have a derivative of 0. Similarly, this technique is extremely useful for training Neural Networks; in minimizing the amount of errors yielded from the Loss Function on behalf of the predictions, in order to achieve the best possible results. [49]

Practically, the Gradient Descent algorithm could be thought of as a person looking for the lowest point in a valley, where he/she computes the valley's slope every few steps (learning rate) until they reach its lowest point.

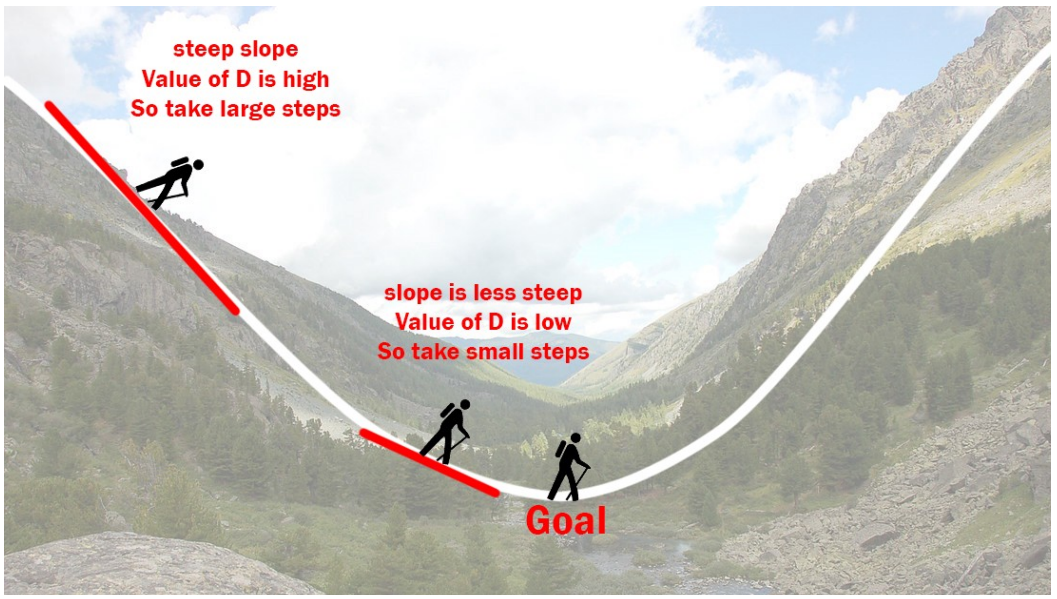


Figure 2.12: Gradient Descent Analogy [50]

Likewise, the Gradient Descent algorithm has following formula:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

Figure 2.13: Gradient Descent Formula

The parameter η parameter symbolizes the learning rate, which may have a powerful effect on the Gradient Descent's performance. The smaller the learning rate, the more iterations that will be computed, therefore it will take longer for the algorithm to converge to its optimum point. On the other hand, if the learning rate is too high, the algorithm may not converge to the function's most optimal point at all, since taking larger steps will also risk skipping past it. [49]

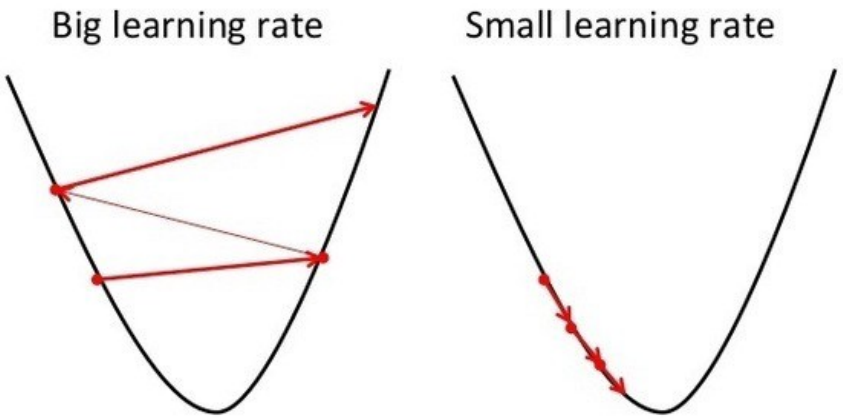


Figure 2.14: Illustration of how Learning Rate Affects Convergence [51]

2.4.6 Optimizers

In general, Optimizers are mechanisms applied in Neural Networks that are used to modify its characteristics, such as the weights, biases, and learning rates, in order to minimize the amount of losses while training. Although there are multiple Optimizers available; such as Adaptive Moment Estimation (Adam), Momentum, and Stochastic Gradient Descent (SGD), they are all based on the core Gradient Descent approach, however modified and improved in their own respective ways for better efficiency and performance. [52]

Indeed, there are specific advantages and disadvantages of each methodology. Regardless, the most commonly used and the default Optimizer for most Deep Learning applications is Adam. This is because upon research on different problems, it has been proven that the Adam Optimizer achieves better results overall, with lesser computational time, as well as requiring the least amount of parameters as compared to the other mechanisms. [53]

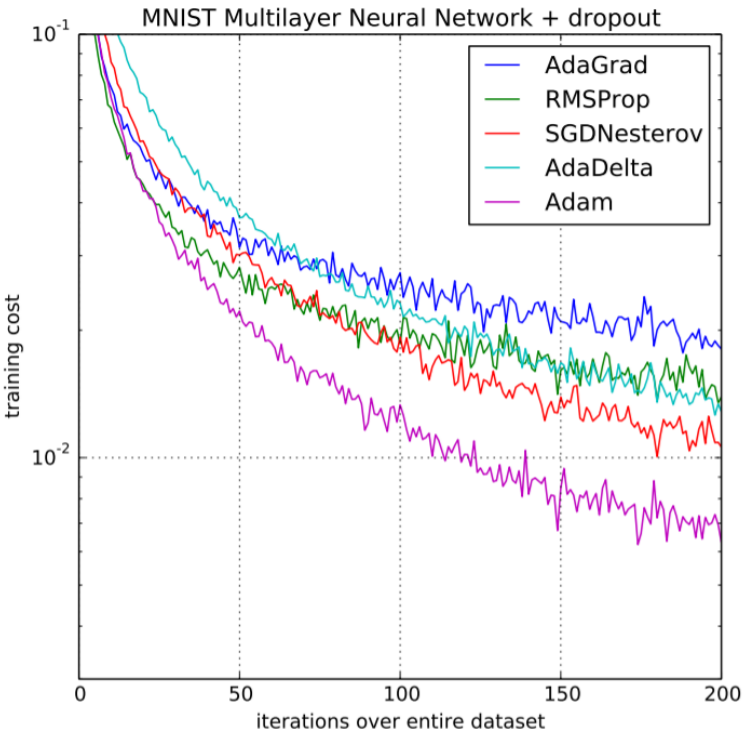


Figure 2.15: Training Cost Comparisons Between Multiple Optimizers [54]

There are two sub mechanisms used in the Adam Optimizer; Momentum and Root Mean Square Propagation (RMSP). Momentum essentially speeds up the process of Gradient Descent itself, by taking in account the exponentially weighted average of the gradients, thereby allowing the algorithm to converge to the minima at a much faster pace. On the other hand, RMSP is an adaptive learning rate approach; controlling its learning rate to make sure that there is minimum oscillation required to reach the global minimum loss, while still taking an ideally large enough step size to bypass any local minimum complications. The Adam Optimizer acquires the positive attributes of these two algorithms, allowing it to achieve the global minimum much more efficiently. [54]

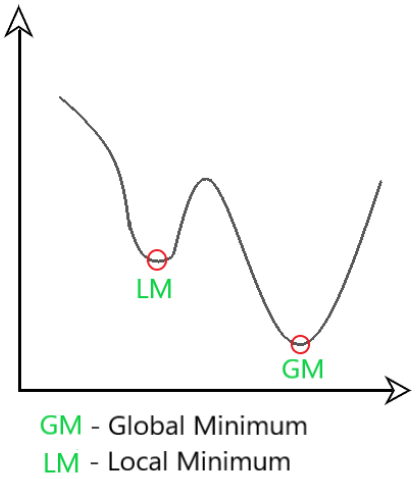


Figure 2.16: Spotlighting the Local and Global Minimums in a Loss Graph [54]

2.5 Deep Convolutional Neural Networks (DCNNs)

In short, Deep Convolutional Neural Networks (DCNNs) are simply Artificial Neural Networks (ANNs) with Convolutions. Due to their enhanced capabilities in handling and identifying patterns in three-dimensional data, they are widely applied for Computer Vision tasks, such as image classification and object detection. This not only includes images, but also videos too. [55] Likewise, more in-depth details on how convolutions work will be discussed in this section.

2.5.1 Convolution Layer

As mentioned in the Computer Vision section, images are represented as pixels in computers, and have a color assigned to every single pixel. This means that every

colored picture is a three-dimensional element. Having an ANN to learn from such types of data will require very complex computations, and will often be extremely resourceful, especially when picture sizes are large, and if there were vast amounts of images to be fed into the algorithm for training. Therefore, the main function of a Convolution layer is to extract the high-level properties of every single image in order to simplify calculations, as well as to find patterns in the image. [56] To enhance its capabilities, a Deep Convolutional Neural Network (DCNN) may consist of multiple layers of Convolution to distinguish between different feature maps.

Convolutions could be thought of as a feature transformation method. There are two main ways of achieving this; Blurring and Edge Detection. Visually from our naked eyes alone, this is just as equivalent to applying filters to images in photoshop, as illustrated in the examples below:

Figure 2.17: Gaussian Blur, Before (Left) and After (Right) [57]

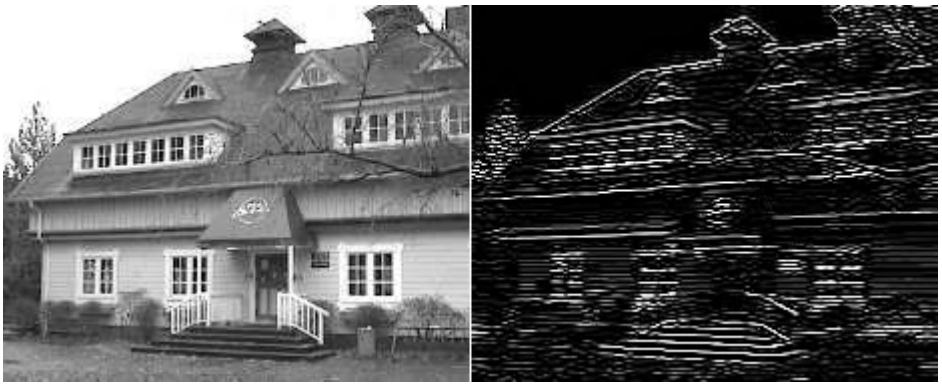


Figure 2.18: Edge Detection, Before (Left) and After (Right) [57]

Passing the image into the Convolution layer will produce such output images, with fewer pixels, however maintaining as many relevant features as possible.

Alternatively, it is also possible to have the output image of equivalent size as the input image. This is done by adding a "padding" of pixels with zero values around the input image so that it is larger in size, hence producing in an output of the desired size.

The Convolutional Neural Network Filter, or scientifically referred to as a Kernel, is a matrix that will be element-wise multiplied repeatedly with the matrices of different portions belonging to the input image, resulting an output known as the Convolved Image. [58] Likewise, more of which will be explained in the example below:

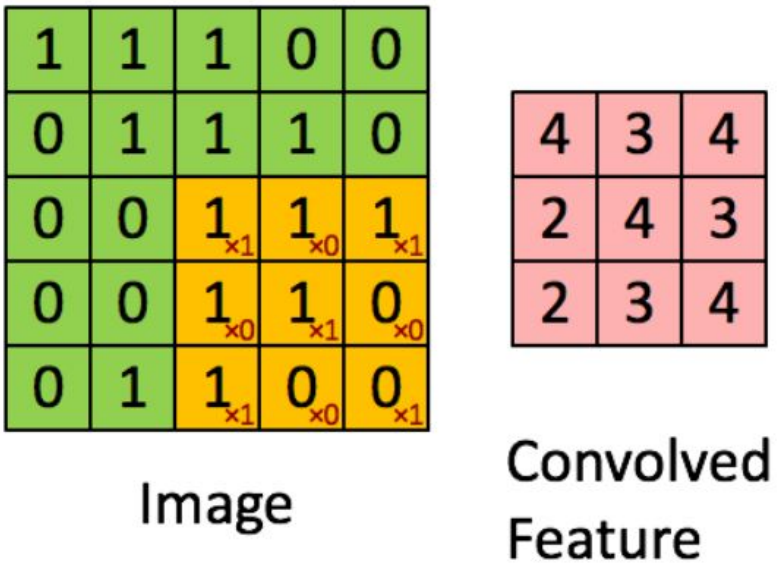


Figure 2.19: A Single Convolution Example [58]

In the figure above, it can be seen that the input image (green area) has a dimension of 5 ✕ 5 ✕ 1; having 5 pixels both horizontally and vertically, as well as another dimension for the RGB color itself. [58] This results the equation:

$$D = 5 \times 5 \times 1$$

On the other hand, the Convolution Kernel (orange area) has a 3✕3 matrix of:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Similarly, this gives the equation:

$$K = 3 \times 3$$

Next, the Filter will be placed overlapping the input image. From here, only the pixels that are being overlapped by the Kernel will be included as the "current" matrix to be element-wise multiplied with the Kernel's matrix, as described in the equation below [58]:

Kernel Matrix Image Matrix

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The multiplication of these 2 matrices will yield the following calculation:

$$1 \times 1 + 0 \times 1 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

Since the Kernel matrix overlaps the bottom-right corner of the image, this means that the result, 4, will be the value for the bottom-right pixel of the Convolved Feature [58]:

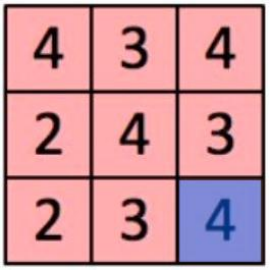


Figure 2.20: Matrix Multiplication Example Result [58]

The Convolution operation begins this element-wise multiplication of matrices from the top-left corner, moving to the right with a step value defined as the "Stride" until

it reaches the top-right corner. Then, the Kernel proceeds to the second row; moving down by the "Stride" value, and repeats the process from left to right. The procedure carries on until the Kernel reaches the bottom-right corner of the image, as depicted in the figure below [58]:

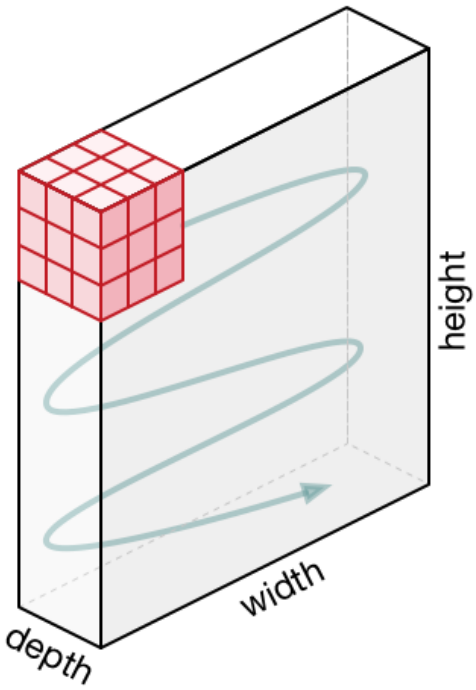


Figure 2.21: The Convolution Operation [58]

Regardless, the examples above only elaborates how a single Filter works to transform an image into a Convolved Feature. In general, Convolution layers typically have more than one Filter, each containing distinct matrix values. The goal of this is to have different Kernels searching for different patterns present in the image. For instance, to create a DCNN for facial recognition, the model needs to be able to distinguish the various features present in different human faces, such as nose, eyes, ears, etc. Hence, having multiple Filters of different values will also compute different Convolved Features as outputs. Finally, all of these Convolved Features will be stacked together as different channels, concluding a 3-Dimensional output of typically larger depth than the input image itself, as shown in the figure below:

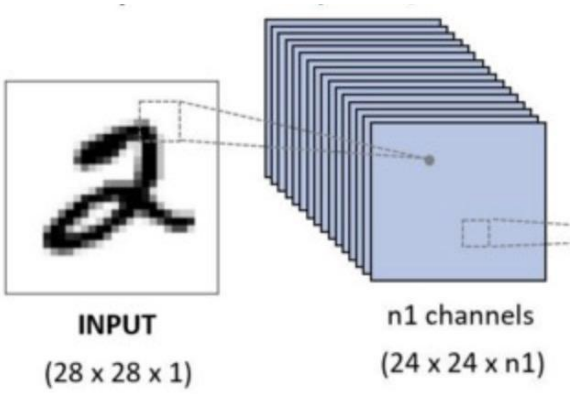


Figure 2.22: Before and After a Convolution Operation [58]

Like the Weights & Biases, these Kernel matrices' values of the Convolution layer are gradually optimized automatically; with Gradient Descent through Forward & Backward propagation of the whole DCNN architecture.

2.5.2 Pooling Layer

If the Convolution layer intends to extract the high-level features of an image, the Pooling layer functions to summarize these Convolved images' features, alongside downsizing them even more to further simplify computations. There are 2 main approaches to this; Max Pooling and Average Pooling.

Similar to the Convolution operation, Max Pooling selects the maximum values of the image that is overlapped by a Kernel that moves around the image, whereas the Average Pooling takes the average values instead. For each of the examples in the figure below, the Kernel size is of 2 × 2 each, shrinks a Convolved image of size 4 × 4 into a 2 × 2 Pooled image:

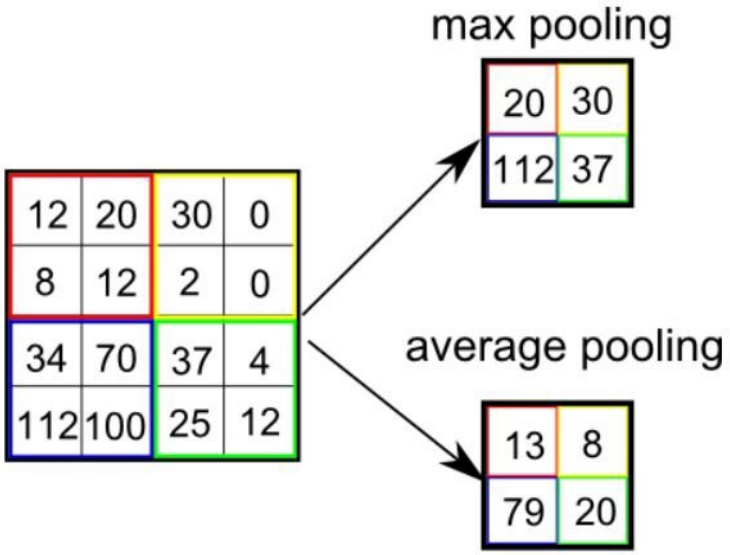


Figure 2.23: Maximum and Average Pooling Results

The reason behind why the Convolution and Pooling layers work is because, recalling that each Convolution layer extracts the high-level features of the input, this means that when a similar pattern is discovered in a certain portion of an image, the element-wise multiplication will compute a higher number for that portion of the output matrix. As mentioned before, each Convolved Feature will then be summarized by getting the maximum or average values through the Pooling layer. Hence in other words, only the dominant conclusions of pattern recognition will be maintained. Typically in constructing Deep Convolutional Neural Networks, we often have multiple Convolution and Pooling layers of the same Kernel sizes. Therefore, upon downsizing the image after passing through a layer, the Filter of the next layer will now cover a larger portion of the image. Thus, the process repeats to find patterns in larger portions of the image.

Another benefit of implementing Max Pooling to the Convolved images is Noise Reduction. This is because some of the pixels in the image may contain imprecise values, if compared to the other pixels. Therefore, generalizing these values will not only reduce dimensions, but also to take out irrelevant data. Unfortunately, Average Pooling may not be utilizing this capability as it simply functions to take the average value of the pixels in the Kernel.

2.5.3 Fully Connected Layer

The final layer of a typical Deep Convolutional Neural Network prior to revealing its output is known as the Fully Connected layer. This layer however, is not only limited to CNNs. This is because, since Neural Networks consist of a set of dependent nonlinear functions with their own respective Neurons, therefore this layer functions to apply linear transformation towards the input vector by taking the dot product of it with a Weights Matrix as follows [59]:

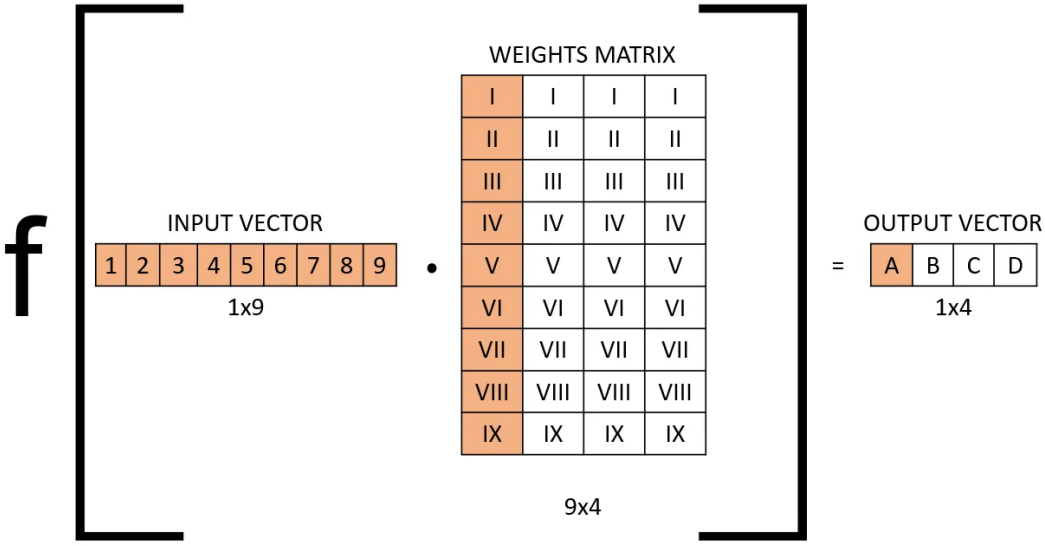


Figure 2.24: Geometrical Illustration of a Fully Connected Layer

As shown in the picture above, the input is a 1 × 9 vector, and the Weights Matrix is of 9 × 4. This gives an output vector of 1 × 4. Additionally, the numbers inside the Weights Matrix depicted in the image above are labeled as variables for simple interpretation, whereas in reality, these values will be automatically generated and optimized as the model is being trained. [59]

This layer is named “Fully Connected” as all of the possible combinations and connections between the input and output layer are considered. This conveys that every single value within the input vector has the influence towards every single output value in the output vector, as described in the figure below [59]:

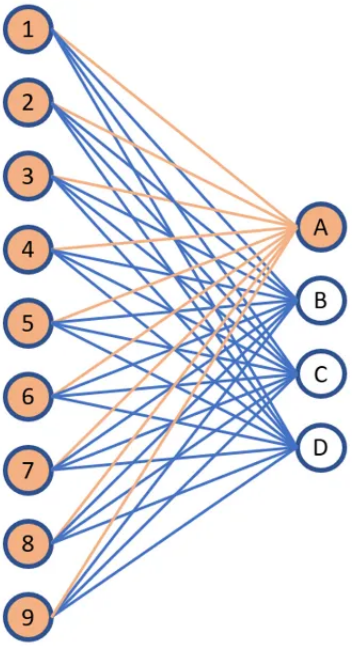


Figure 2.25: Fully Connected Layer Blueprint [59]

2.5.4 Constructing a Deep Convolutional Neural Network

Together, when all of the layers mentioned in this subchapter are utilized, we are able to form a fully working Deep Convolutional Neural Network for multiclass image classification, as shown in the figure below:

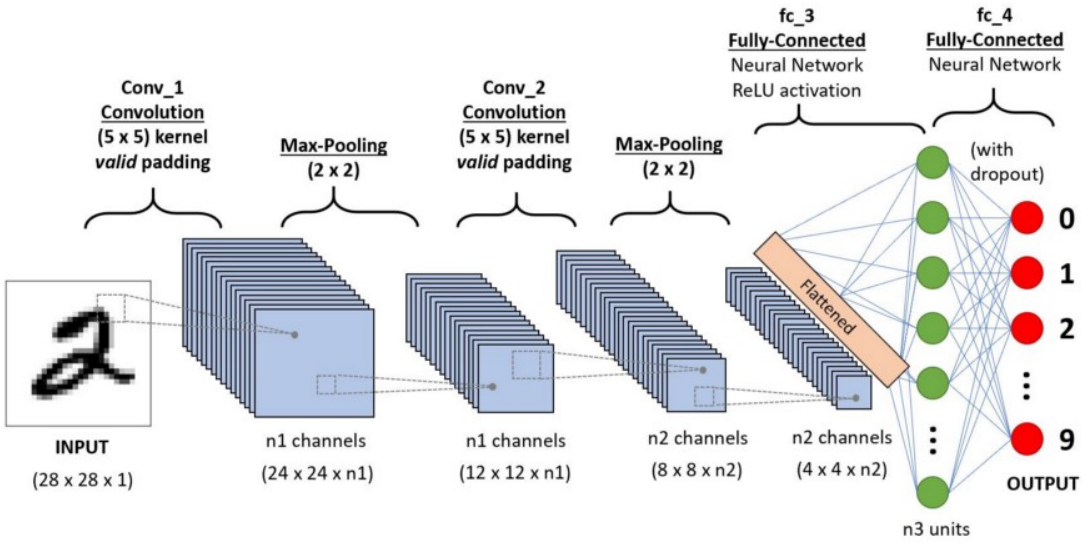


Figure 2.26: Example of a Deep Convolutional Neural Network Architecture [58]

In a nutshell, the first Convolution Layer extracts the high-level features from the raw input images, while slightly decreasing its dimensions from 28 × 28 × 1 into 24 × 24 × 1. Next, the Max Pooling Layer down-sizes the Convolved Features by taking the maximum values of its pixels, while decreasing its dimension to 12 × 12 × 1. Then, it is followed by another two sets of Convolution & Max Pooling Layers, that consecutively extracts high-level features from their previous layer, also transforming the output image's dimension into 4 × 4 × 1. Finally, the Fully Connected layer converts all of the nonlinear outcomes into linear form by multiplying its output vector values with a Weights Matrix, generating an output layer where the category predictions are revealed.

2.6 Transfer Learning

Transfer Learning is a concept of Machine Learning where models that were previously developed and trained to solve problems are reused again as the beginning point to train a new dataset of another related problem. Regardless, in order for Transfer Learning to work, the features of both datasets from the first and second tasks must be somewhat related or similar to one another [60]. For instance, models that were used to distinguish between trucks could be used to recognize different cars.

"In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them, to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task." ~ MachineLearningMastery.com [60]

This fashion of Transfer Learning is known as Inductive Transfer, meaning that the scope or bias of the model that was previously trained on the first task is narrowed down by fitting it on a different but similar dataset [60].

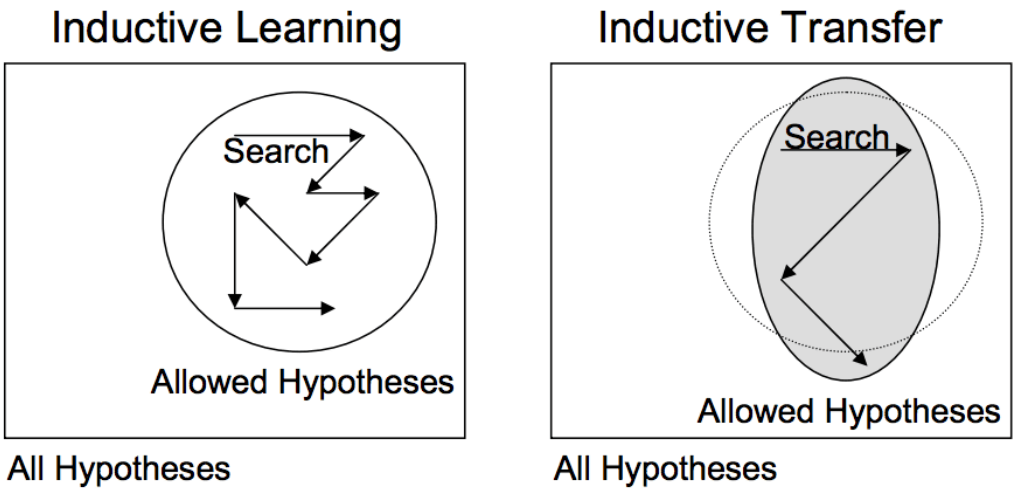


Figure 2.27: Inductive Learning & Transfer [60]

Transfer Learning is very popular in modern Deep Learning applications, mainly for Computer Vision and Natural Language Processing, knowing the time and computational resources needed to construct and train models for such projects from scratch [60].

In general, there are two approaches in which developers could utilize Transfer Learning; Develop Model and Pre-Trained Model approach. The contrast between the two is simply if the model for the first task was built by the developers themselves, or retrieved from another source respectively. Pre-Trained Models are commonly published by research institutions that have previously trained it on multiple different complex or large datasets, such as ImageNet. Developers are able to choose from a wide variety of Pre-Trained models as well as data pools that they have been trained on using widely known APIs like Tensorflow [60].

In this research experiment, we have implemented Transfer Learning as an alternative for just the DenseNet 201, as this is the latest architecture we have. Regardless, in later chapters we will see if this is an effective approach to training our models given the performance it concludes.

2.7 Performance Evaluation Metrics

To monitor and judge the performance of Machine Learning models, Evaluation Metrics are utilized. These metrics are different from Loss Functions that were

mentioned in earlier subchapters. Loss Functions measure a model's performance while training, for optimization through Gradient Descent and Forward & Backward propagation. Evaluation Metrics, on the other hand, are mathematical formulas that compare the model's prediction results with the dataset's actual target values to conclude its performance [61]. These metrics could be computed for the model's in-sample (training) and out-of-sample (testing) prediction scores.

There are several Evaluation Metrics that could be chosen from, depending on the type of problem to be solved. For regression tasks, the most commonly used metrics are Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), or R-Squared. Alternatively, the most popular metrics for classification problems are Accuracy, F1, Precision, and Recall Scores. Additionally, the Classification Report is also used to compile these metric scores altogether, not only to have a clearer picture of the overall model's performance, but also that for every single category [61].

Since this research entails a Multiclass Classification problem, this subchapter will only look into and explain the Classification Evaluation Metrics.

Accuracy Score

Accuracy Score is the simplest amongst all other evaluation metrics, which is simply calculated by dividing the number of correct predictions by the total number of predicted entries. The outcome is also often multiplied by 100 to convert it to a percentage form [61].

$$\text{Accuracy Score} = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions} \times 100$$

Accuracy Scores are also the most commonly used metric for Classification tasks, as it quickly measures how well models perform based on the number of correct predictions. However, there are also many cases in which Accuracy Scores aren't relevant, such as when a model is trained and used to predict imbalance datasets; where there are inconsistent number of entries for each category. For instance, the

ratio between class A to class B is 3:1 in a Binary Classification problem. Therefore, even if the model predicts all the targets as class A, the final Accuracy Score will still be 75%, which is supposedly marked as a decent performance. Similarly in more extreme cases, where the ratio of class A to class B could be 1:99, if all the predictions are of class B, the model will achieve a 99% accuracy. Alternatively, Precision, Recall, and F1 Scores are more relevant to be used in these types of scenarios.

Positives and Negatives

Before moving forward to the other remaining metrics, it is an important prerequisite to understand Positives and Negatives in model predictions. Taking a Binary Classification problem as a simple example, Positives are predicted true category values, whereas Negatives are the predicted false values. For instance, in a task to predict if a patient has cancer or not, the cancer predictions will be noted as Positives, whereas the non-cancer predictions are Negatives.

True Positives is basically the total number of correctly predicted values of a true category; such as the number of patients with cancer that is correctly identified. On the other hand, False Positives is the number of wrongly identified cancer patients. Similarly, True Negatives are the precisely predicted patients without cancer, and False Negatives are patients with cancer but are identified as non cancerous.

Precision Score

In short, Precision Score is a computation to measure how many predicted true items are relevant, by dividing the total number of True Positives by the total number of Positive predictions [61].

$$\text{Precision Score} = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Relevance is important in evaluating a model's performance, especially for imbalanced datasets, as it measures how well the model predicts one category. Therefore, even if the predictions are imbalanced, the Negatives are neglected for

this calculation. A high Precision Score, usually close to 1, signals that all the class' true predictions by the model are barely incorrect. On the other hand, models that have a lower than 0.5 Precision Score signifies that most of the predicted class' true values are incorrect [61].

Recall Score

In predictions for a Binary Classification problem, and based on the definition in earlier subsections in this subchapter, True Positives and False Negatives are the actual true categories in the dataset. The Recall Score is a measure of how well the model performs for this category, by dividing the total number True Positives over the sum of True Positives' and False Negatives' counts [61].

$$\text{Recall Score} = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

In contrast to Precision Score, the Recall score calculates the percentage of truly predicted values that have been predicted correctly, out of all the actual true values in the dataset. Thus, this is also a very useful metric for evaluating a model's performance on imbalanced datasets. A high Recall Score, preferably near to 1, indicates that the model successfully predicted nearly all the actual true class values precisely, whereas a low Recall Score of less than 0.5 tells that the model failed to predict most of the actual true values present in the dataset [61].

F1 Score

In addition to Precision and Recall Scores, the F1 Score is the combination of both of these metrics, estimating the harmonic mean of the two values [61], having defined by the mathematical formula below:

$$F_1\text{−score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

Figure 2.28: F1 Score Formula [62]

In evaluating Classification models, F1 Scores are perhaps the most commonly used metric aside from Accuracy Scores, especially for imbalanced datasets. However, it

is usually followed by its preceding metrics, Precision and Recall Scores, which helps to identify the model's performance in their different respective areas. Generally, the ideal judgment of a model's performance using F1 Score is the following:

F1 score	Interpretation
> 0.9	Very good
0.8 - 0.9	Good
0.5 - 0.8	OK
< 0.5	Not good

Figure 2.29: F1 Score Benchmarks [63]

Classification Report

In addition to Precision, Recall, and F1 Scores individually, researchers could also compile all of these metrics in a single Classification Report table to have a clearer picture of the model's performance. On top of each metric value, it typically also includes their averages, as well as the Support values; which is the total number of occurrences for each specific category [64]. The table below illustrates an example Classification Report for a Binary Classification task:

	precision	recall	f1-score	support
0	0.92	0.93	0.92	12500
1	0.93	0.92	0.92	12500
micro avg	0.92	0.92	0.92	25000
macro avg	0.92	0.92	0.92	25000
weighted avg	0.92	0.92	0.92	25000

Figure 2.30: Classification Report Example [64]

Classification Reports are essential for all Classification problems, convincing researchers the metric scores for each category. This is even more important for Multiclass Classification problems, where each category needs to be evaluated properly in order to achieve the best performance possible.

2.8 Real-World Examples of DCNNs

Many of these popular Deep Learning algorithms, as well as Transfer Learning techniques, have been deployed to solve real-world problems. This section will provide a few examples, as well as elaborate specifically how they have been implemented.

COVID-19 Classification with Chest X-Ray Images using the ResNet-50 Architecture

Due to the surging COVID-19 cases from all over the world, it has become very impractical to use test kits such as RT-PCR to diagnose patients with respiratory misalignments. This could be due to the test needing a relatively long turn-around time, low sensitivity and accuracy rate, as well as insufficient test kits in different geographical areas. Instead, in the article titled “Transfer Learning with Fine Tuned Deep CNN ResNet50 Model for Classifying COVID-19 from Chest X-Ray Images”, authors attempted to implement Transfer Learning using the ResNet50 architecture to classify patients using Chest X-Ray images from the COVID-19 Radiography Database. In this work, 10 Pre-Trained weights with a total of 14530 samples (roughly 7200 samples of each category) have been experimented [65]:

1. ChestX-ray14
2. Chexpert
3. ImageNet
4. ImageNet_ChestX-ray14
5. ImageNet_Chexpert
6. iNat2021_Supervised
7. iNat2021_Supervised_From_Scratch
8. iNat2021_Mini_SwAV_1k
9. Moco_V1
10. Moco_V2

Thus, achieving the Accuracy and Loss results for each weight below:

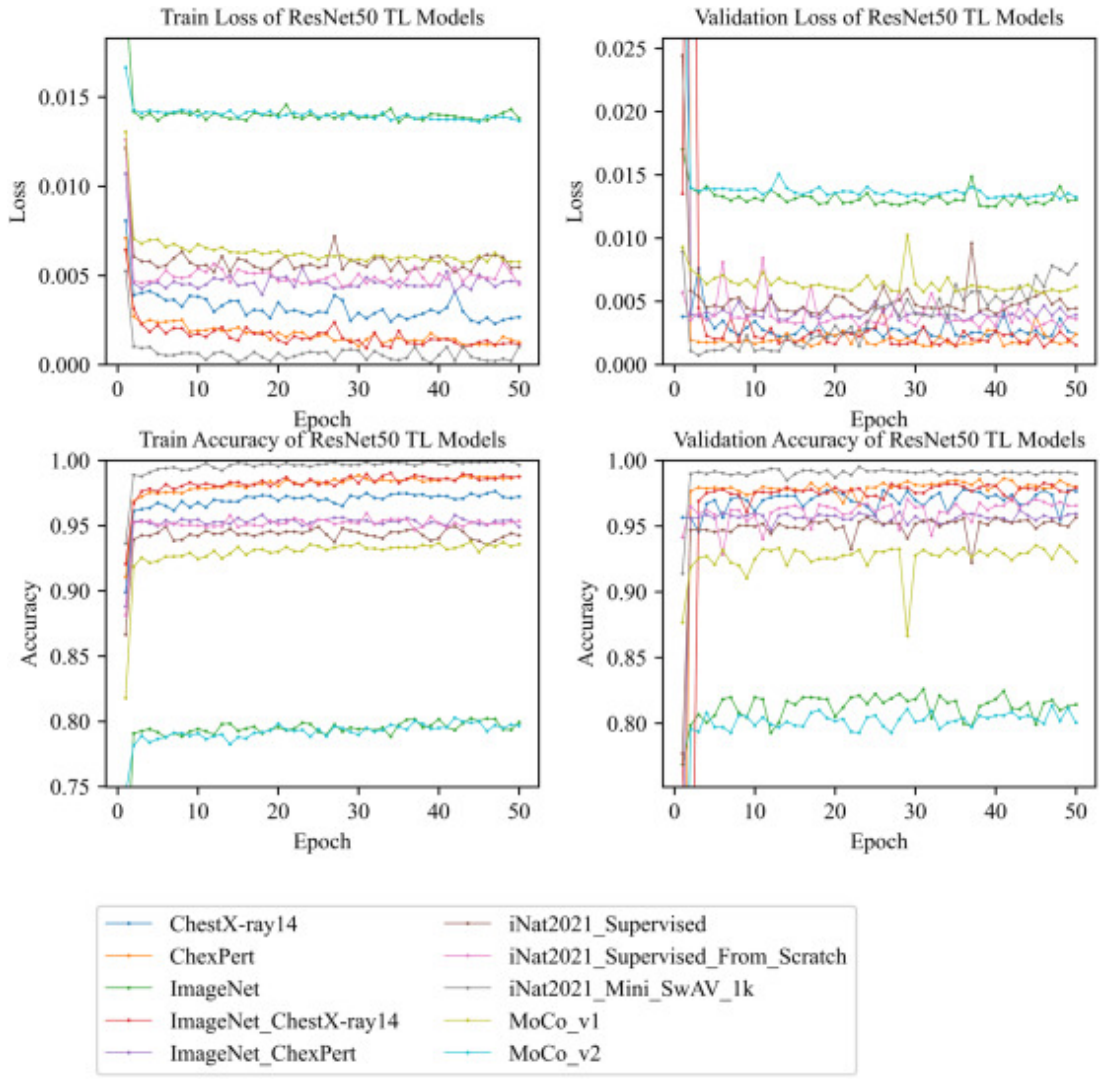


Figure 2.31: ResNet50 Results on Predicting COVID-19 Patients [65]

From the visualization above, we can conclude that the ResNet50 architecture is able to achieve an Accuracy of larger than 90% for most of the participating weights, and only the ImageNet and MoCo_V2 dataset that it performed slightly above the 80% mark, for both Training and Validation sets [65]. In general, models with an 80% could be considered as good. Hence, overall, it shows that the ResNet50 model is able to diagnose COVID-19 patients reliably with just their chest X-Ray images.

Ancient Dynasty Mural Classification with the Inception V3 Architecture

It is practically very challenging for historians to identify the exact period in which ancient murals were created. This is due to several artificial and natural factors, such

as similarities in content and style amongst caricatures, how the item had been preserved, low image resolutions, and many more. The research paper "Application of a modified Inception-v3 model in the dynasty-based classification of ancient murals" elaborates how Deep Learning could be adopted to solve such complexities through Multiclass Classification. Similar to the previous case, the author adopted a Pre-Trained model with Transfer Learning technique on the ImageNet dataset, consisting of a total of 9700 images with 6 different Dynasty categories. Surprisingly, the final model is able to achieve an Accuracy, F1, and Recall Scores of 88.4%, 88.36%, and 88.32% respectively, as depicted in the charts below [66]:

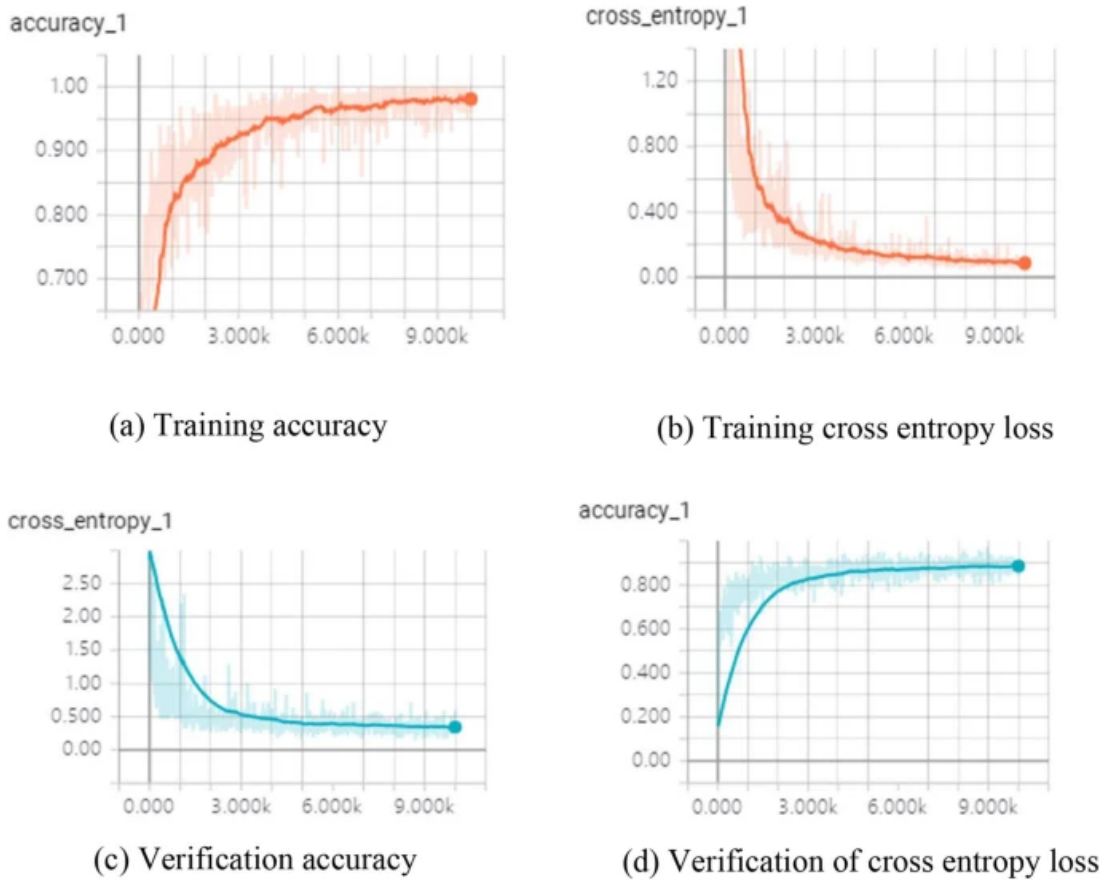


Figure 2.32: Inception V3 Results on Ancient Dynasty Mural Classification [66]

Based on the results yielded by the model above, it could be concluded that the Inception V3 architecture is able to achieve a satisfactory score of nearly 90%, and could thereby be helpful for historians to predict the Dynasties in which ancient murals originate from.

Mask Detection using DenseNet-201 Architecture

During the early days of the COVID-19 pandemic, the use of masks had been an obligation for everyone to prevent the widespread spread of the virus. This led researchers to develop a Computer Vision detection system to classify whether or not people are wearing masks in public areas; utilizing Transfer Learning with a Pre-Trained MobileNetV2 Architecture, which achieved an F1 Score of just 0.67. According to a research titled "A Deep Learning Using DenseNet201 to Detect Masked or Non-masked Face", the authors hypothesized that this is not an accurate enough score, and alternatively proposed a new detection system with the DenseNet-201 architecture instead. In this study, authors trained this model using the Masked Face Recognition Dataset (MFRD), consisting of over 92,000 images of 525 people (90,000 Masked and 2,203 Non-Masked images), which is an imbalance dataset. These images are then split into an 80% and 20% Training and Validation sets respectively [67]. Upon comparison, they were able to conclude the following Accuracy scores for the previous and current models:

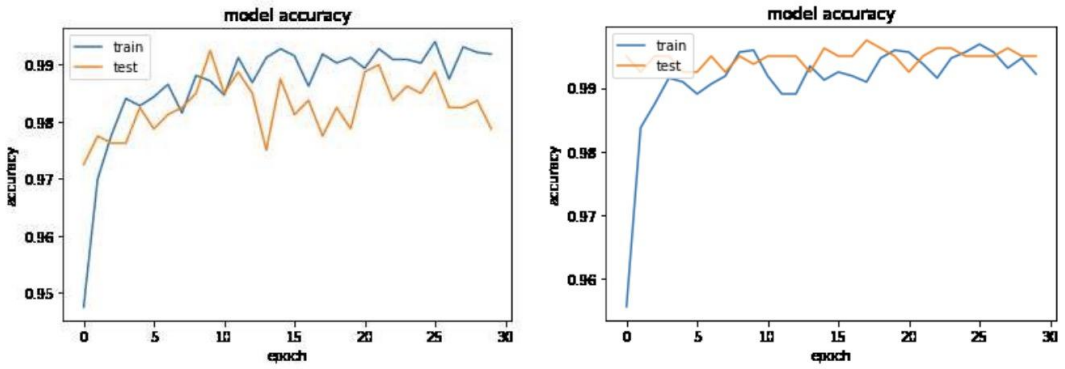


Figure 2.33: MobileNetV2 (left) VS DenseNet201 (right) Model Accuracies Comparison in Masked and Non-Masked Binary Classification [67]

From the illustration above, although both models are able to achieve an exceptional accuracy of over 97%, it could be conveyed vividly that the DenseNet-201 model was able to achieve a more stable and higher average Accuracy Score on both its Training and Testing sets. Moreover, the DenseNet-201 architecture was able to surpass MobileNetV2 with a much higher Recall and F1 Scores, 0.97 and 0.98 respectively, as depicted in the table below [67]:

THE TESTING RESULTS OF MODEL BUILDING WITH MOBILENETV2 AND DENSENET201

Model CNN	Recall	Precision	F-Measure
MobileNetV2	0.50	1	0.67
DenseNet201	0.97	1	0.98

Figure 2.34: MobileNetV2 and DenseNet201 Model Evaluation Comparison in Masked and Non-Masked Binary Classification [67]

Therefore in conclusion, the DenseNet-201 is able to competitively outperform the MobileNetV2 architecture in this Binary Classification task by miles, and could thereby replace it for this Mask Detection system.

CHAPTER 3: PROBLEM ANALYSIS

In this chapter, the problem with an alternative parking layout to reduce parking congestion will be discussed thoroughly, as well as the proposed solution supporting it with Machine Learning. The subsections in this chapter will also include any related works, as well as how this research is able to contribute to the fields of Computer Vision.

3.1 Problem

Finding a car parking spot has never been more difficult in modern times. With the rise in world population, demand for personal convenience, and terrible public transportation services in less developed countries, led to the rise in the need for private automobiles all over the world. Unfortunately, this further leads to more congestion, pollution, and fuel consumption not only on roads, but also in public parking areas too. As discussed earlier in Chapter 1, drivers spend an average of 17 hours every year just to look for parking spots. The numbers are much higher in larger metropolitan areas like New York City, where the number could go up to 107 hours [1]. Hence, parking congestion is a major problem in densely populated metropolitan cities.

Figure 3.1: Parking Congestion in Mumbai, India [68]

Modern day cars have many different body types, such as Sedans, Hatchbacks, Wagons, Sport Cars, Multi-Purpose Vehicles (MPVs), Sport Utility Vehicles (SUVs), Trucks, and many more. This results in cars being built in different widths, lengths, and heights. In general, almost all parking lot sizes are engineered based only on large cars, which are usually SUVs, MPVs, and Trucks. This is definitely reasonable, as designing parking lots for large cars will technically fit smaller cars as well. Alternatively, instead of engineering parking spots based on large cars to fit all cars, we can segregate different parking sections for different types of cars, just like how we typically separate lots for motorbikes and cars. In this manner, smaller vehicles will find more suitable spots to park; such as lower ceiling areas, shorter available distances from curbs, tighter backout path widths, & narrower corners. On the other hand, the saved spaces could be given for larger cars to park. Additionally, for larger parking buildings, they can have different floor levels for different car types for convenience. Thereby, the final layout will leave more available parking spaces for all. Furthermore, with this structure, drivers will also be able to immediately locate sections where their car belongs and thereby needing less time looking for parking spots.

Nonetheless, in order to implement the aforementioned alternative parking layout, it requires a very manual labor process to carefully study the cars that are entering these parking spots, before labeling them into different categories based on their sizes, and telling drivers where to park. Unfortunately, this leads to additional costs of hiring staff. Moreover, it is impractical to completely rely on employees, as it is an exhausting yet monotonous task to standby & tell drivers where to park for hours straight; potentially causing human errors. Alternatively, a sign could be placed in the entrance so drivers can determine themselves where to park, in order to prevent the need for manual labor. However, some drivers are not so familiar with automobiles, thus this may lead them to confusion, or worse parking in the wrong section.

These days, it is common to hear of how Machine Learning has been implemented in our daily lives to benefit both individuals or corporations with various tasks or operations, including automation. Hence similarly, can Machine Learning be incorporated to assist the aforementioned problem?

3.2 Related Works

In October 2020, a paper titled “Real-Time Vehicle Classification” was published to IEEE (Institute of Electrical and Electronics Engineers) by 3 authors from Bangladesh. The research was aimed to reduce the number of road accidents in their country by using Deep Convolutional Neural Networks to classify 4 of the most common types of cars in real time. The project came with multiple challenges, ranging from variation of shapes and colors in the image dataset. Fortunately, the final output reveals a 97% accuracy, which concluded a decent performance on the real-time test dataset [69].

Another research published in February of 2021 titled “Convolutional Neural Network Based Vehicle Classification in Adverse Illuminous Conditions for Intelligent Transportation Systems” came with an objective to innovate the effectiveness of the current traffic control and highway automation systems. The authors claimed that the existing solutions were only trained on very limited and small datasets, and thus aren’t able to cater real-time road traffic conditions. Alternatively, Deep Learning is incorporated to solve the above matters. In addition, the 10,000 image dataset, containing 6 different categories of vehicle types, that were used in training the models are mixed with random noises, such as weather conditions and illumination factors to improve its robustness in real-time applications. The project then assess the performances of pretrained AlexNet, GoogleNet, Inception-V3, VGG, and ResNet models that were fine tuned based on the artificial dataset. Next, the best performing model, which is the ResNet architecture, was improved by adding a new classification block on the network. Moreover, to ensure generalization, the authors also fine tuned the network on the public VeRi dataset engulfing 50,000 images, which have also been categorized into 6 different vehicle classes. Finally, the proposed vehicle classification method was evaluated, revealing a 99.68% Accuracy, 99.65% Precision, and 99.56% F1 Scores [70].

A recent project published in 2022 titled “CNN-Based Classification for Highly Similar Vehicle Model Using Multi-Task Learning” aspires to improve the existing intelligent traffic law enforcement systems, by implementing vehicle make and

model classification in scenarios where traffic violations had been committed but the license plate failed to be obtained. The main challenge in this research is the fact that there are vehicles of different make and model vehicles but are very similar in visual appearances. To overcome this issue, a fine-grained Convolutional Neural Network classifier with multi-task learning is incorporated in the paper. The proposed approach begins with extracting features from the input images using the VGG-16 architecture. The extracted features are then split into 2 separate branches for classification; one for vehicle make, and the other one for vehicle model. Finally, the performance of the proposed method was evaluated with the InaV-Dash dataset, containing images from various Indonesian cars with very similar visual appearances. This result yields an Accuracy Scores of 98.73% and 97.69%, for vehicle make and model classification respectively [71].

3.3 Proposed Solution

As mentioned in the Problem section above, it requires a manual process to run the more efficiently-designed parking layout. Fortunately, this operation could be automated with the help of Deep Learning and Computer Vision in classifying different car body types, upon the car arriving at the ticket booth prior to entering the building.

Specifically, instead of only reading car plate numbers and handing out parking tickets, the OCR camera in the entrance booth could also be used to capture the car's image and pass it into a trained Deep Learning model. Utilizing this image as an input, it predicts the category of the entering car. The predicted results could then be printed out on the parking ticket handed out to the driver, which they can use to locate the dedicated parking section where their car should belong.

In order to incorporate Computer Vision to assist such operation, it is crucial to first prove that it is able to visually classify different automobile types. Ideally, more than one Deep Learning algorithm shall be considered for prototyping & experimentation. Hence, this research entails the performance comparison between 6 different Deep Convolutional Neural Networks (DCNN) in performing multiclass automobile types classification; including 5 base and 1 Pre-Trained (Transfer Learning) models:

1. ResNet50

2. Inception V3
3. DenseNet 201
4. DenseNet 201 (Pre-Trained Imagenet Weights)
5. Xception
6. Custom-built DCNN Architecture

Likewise, more of which will be elaborated in Chapter 4.

3.4 Research Contribution

This project proposes a solution utilizing Computer Vision to automate the process of manually labeling cars, which is essential for the alternative parking layout designed to improve the efficiency of space allocations in modern parking lots, and thereby to reduce parking congestion. Hence, this paper elaborates how Deep Learning could indirectly contribute to reducing parking congestion, which carries many benefits to its stakeholders & the environment; less fuel consumed & purchased, emissions, time drivers spent to look for a parking spot, and many more.

Furthermore, the whole process can be fully automated; by reverse engineering the OCR softwares that are already used in typical parking lots to automatically read car plates, to also capture the car's body image and pass them into a trained Deep Learning model which predicts the classes of the entering cars. Thus, it is likely that not much additional expenses in hardware equipment or software modification is needed.

This project not only compares multiple base Deep Convolutional Neural Network models' performances trained on a given dataset, but also involves that of Transfer Learning; a Pre-Trained DenseNet-201 network carrying ImageNet weights. Moreover, it also provides a comparative study on how removing the images' backgrounds, or in other words removing noises from the input images have an effect on each model's performance. Therefore, this allows for consideration to be implemented in the future.

This project is simply a proof of concept for the extent of Deep Learning in solving existing general issues. Hence, the models in this project could still be optimized

further in many ways to achieve better performances; such as training it on larger car datasets, more or newer vehicle models (car manufacturers typically release new models every few years), tuning their hyperparameters, and many more. Likewise, all the source codes for the experiments conducted in this research have been attached in Appendices C to I of the paper.

CHAPTER 4: SOLUTION ARCHITECTURE

This segment entails each specific step in which the experiments were conducted. This includes the dataset chosen, data preprocessing methodologies, dataset statistics overview, Deep Learning architectures used for modeling, and finally the methods used for evaluation. The flowchart below depicts the steam of each experimentation process:

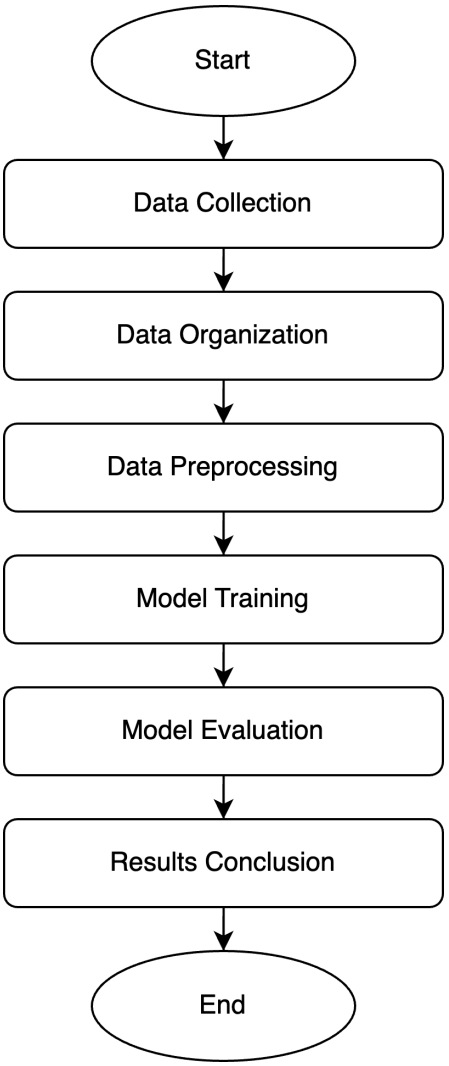


Figure 4.1: Solution Architecture Overview

4.1 Data Collection & Organization

This project will make use of the Stanford AI Cars Dataset to train and evaluate our models. The data consisted of 16185 car pictures of multiple different types, making it ideal and sufficient for training our models. Initially, these images are categorized into 196 different classes. These classes are of the *year-make-model* of the cars; for

instance: “2012 Tesla Model S”, or “2012 BMW M3 Coupe”. These classes are separated into an approximately equal amount of train & test portions; about 8090 images each [5].

Figure 4.2: Stanford AI Cars Dataset Preview [5]

As mentioned before, since we would like our Machine Learning models to classify car body types instead of *year-make-model*, we then need to modify the dataset into such categories instead. The first step is to compile both the dataset’s train and test portions altogether. This is because the initial division will not be implementable for our project later on, as we will also require a validation portion too. Thus, we will need to manually split the data into 3 sets rather than 2; for training, test and validation. Then, since the dataset consists of images that are already organized into folders named as their categories, or in other words pictures of the same cars are already separated into their respective folders labeled as their *year-make-model*, the next job is to manually determine which body type each of these car belongs to, and finally segregate their images accordingly based on it. This transforms the whole dataset to only having the following 7 classes:

1. Sedan
2. Sport
3. SUV
4. MPV
5. Hatchback
6. Wagon
7. Truck

4.2 Preprocessing Data

To make sure that the models can achieve the best possible outcomes, the next maneuver is to preprocess and clean the data. This involves both removing unnecessary features, and formatting the images and colors into a coherent format to be fed into the models later on.

Removing Backgrounds

The first preprocessing step was to remove the backgrounds for all the images in the dataset. This is important as it neglects the unnecessary noises in features, so that it could be seamlessly interpreted when trained by our models. To help achieve this, the Python libraries RemBG and PIL (Python Image Library) were utilized.

Figure 4.3: Removing Dataset Image Backgrounds Preview

Regardless, a duplicate of the unmodified dataset is kept. This will later be used to compare our models' performances working on the dataset with and without the backgrounds removed.

Normalization: Rescaling Colors

As mentioned in earlier chapters, for RGB images, the intensity of each Red, Blue, and Green color channels are represented as numbers within a range of 0 to 255. The next preprocessing step is to rescale these values to a range between 0 and 1. This is

done by dividing all the color channel values by its maximum possible value, which is 255.

The premise to doing this is because in vast datasets like the ones used for this project, some images are likely to have higher color ranges, while others have lower color ranges. This may be due to varying image qualities, luminosity when the photographs were taken, and many more. Nonetheless, since all of these images are being fed into the same models, and thereby sharing the same weights and learning rates, images with higher ranges typically contribute to stronger losses as compared to those with lower ranges. Such extremes will later affect their sum values' adjustments during Backward Propagation. In other words, images with larger color ranges will own more votes in determining the model's weights, as compared to those with smaller ranges. Additionally, such extremes in loss values do also affect the model's learning rate; images with stronger losses will require smaller learning rates, whereas those with weaker losses are more suitable with larger rates. Therefore, normalizing these color ranges in images is a necessity and allows them to evenly contribute to the total loss values during model training [72].

Likewise, thanks to the high-level Tensorflow API, such preprocessing could be done almost instantly by simply setting the rescale parameter in the "ImageDataGenerator" class, which is also used to load the images into the model.

Normalization: Resizing Images

The Stanford AI Cars Dataset used for this project contains images of different pixel dimensions. Prior to feeding these images into our Deep Convolutional Neural Network models, it is essential to normalize these image sizes to make sure that the number of features are standardized for each and every input. Since the concern revolves around image sizes, resizing them all to a conventional target dimension will do the work.

There are a couple reasons for performing this preprocessing step. The first reason is generally, both Machine and Deep Learning algorithms require a standardized set of input features for each and every sample in the dataset. Hence, having an irregular amount of features throughout the entries will cause the model to fail learning it in

the first place. The second reason is since we are working on image datasets, the dimensions of these input images also heavily dictate the time complexities when training these Convolutional Neural Networks. This is because technically, Deep Learning algorithms will learn faster on smaller images as they have fewer features listed in them. Alternatively, having an image of just twice the size will require quadruple the amount of effort to train, as there are four times as many pixels [73]. On the other hand however, reducing image sizes also meant that there are bound to be details lost from the original input. Therefore, setting the conventional dimension also requires a comprehensive consideration and balance between such tradeoffs. With regards to this project and upon the broad considerations above, all pictures in the dataset is rescaled to a target size of 100 × 100 pixels.

Similar to rescaling the colors which was mentioned in the previous subchapter, resizing images can also be done with the "ImageDataGenerator" class of the Tensorflow API, by simply setting the target size variable of its "flow_from_directory" function.

4.3 Dataset Statistics

Upon preprocessing the dataset, they are now standardized and are ready to be trained by the models. Prior to doing so, it is also important to comprehend the distribution of each category. Also, as described earlier in this chapter, we would need to split the dataset into 3 subsets for training, validation, and testing each. All of which will be demonstrated in this subchapter.

Category Distribution

Upon looking into the dataset, the amount of entries for each category could be broken down as follows:

Table 4.1: Dataset Category Distribution

Category	Number of Images
SUV	2846
MPV	1076
Sedan	4154

Sport	4537
Hatchback	1558
Wagon	411
Truck	1603

From the table above, it could be seen that there are indeed a varying amount of entries for each category. In the original dataset downloaded from the internet, this may not be the case for the 196 classes. However, since we have previously reorganized the classes manually into the above 7 classes of car types only, we are now left with the imbalanced dataset with the statistics shown above. Unfortunately, the imbalance in the dataset is pretty extreme among the classes. For instance, there are over 11 times more Sport car images than that of Wagon. Similarly, there are roughly 3 times as many Sport car images than both Hatchbacks and Trucks each. As a matter of fact, out of the total 16185 images, 28% and 25% of them are Sports and Sedans respectively, making them both the most dominant classes that adds up to over half the total dataset.

In general there are 2 main techniques in balancing such imbalanced dataset, which are Undersampling and Oversampling. Undersampling involves keeping all the data from the class with the least amount of entries, and decreasing the number of data for the other classes in order to match with it. Alternatively, Oversampling entails maintaining the amount of entries for the category with the largest number of data, but increasing the number of entries for the other classes to compromise with it; which may involve methods such as duplicating or generating synthetic data.

Unfortunately on behalf of this project, the Undersampling technique will not be ideal as this means that much of the data that could potentially be used for training will be discarded. In fact, by standardizing the entry sizes of each class to that of the smallest class; the Wagon, with only 411 images. This means that we will retain a total of only 2877 images, which is approximately just 17% of the total dataset. On the other hand, Oversampling is also nonoptimal for our dataset, as this will potentially cause the model to be biased towards the feature values within the duplicated or synthetically modified entries of a certain class. Specifically, for each Oversampled class, models will mainly take in account the most dominant cars'

year-make-models' features, such as color and design, in order to predict that category. Whereas in general, test data entries may contain much more robust features, and there may be cars of multiple different *year-make-models* that also belong to this category. This leads to the model potentially overfitting upon evaluating it on a test set. Hence, the dataset is maintained in this manner and carried on to the next stage.

Train, Test, and Validation Split

Generally in building Deep Learning models, developers split their dataset into 3 independent portions; which are train, test, and validation sets. As described in its name, the train set will be fed into the model for training. On the other hand, the validation set is a separate portion used to evaluate the model's performance while tuning its hyperparameters, hence will also be used throughout the training process. Alternatively, the test set is another dataset portion that is isolated from the training process, or in other words data that have never been seen by the model, that will be used to judge its predictive performance in the field.

Initially, the original Stanford AI Cars dataset is split into training & test portions of 50% each, thereby having roughly 8000 images in each set. Intuitively, this may not be enough for training the models to get the most optimized results. Moreover, we also require a validation subset which is not included.

Therefore, the portions were recompiled, reorganized, and split into a training size of 80%, validation & test sizes of 10% each. This way, our model will have approximately 13000 images for training, while having about 1600 pictures for validation, and the leftover of approximately 1600 images that have never been seen by the model will be used to evaluate its performance.

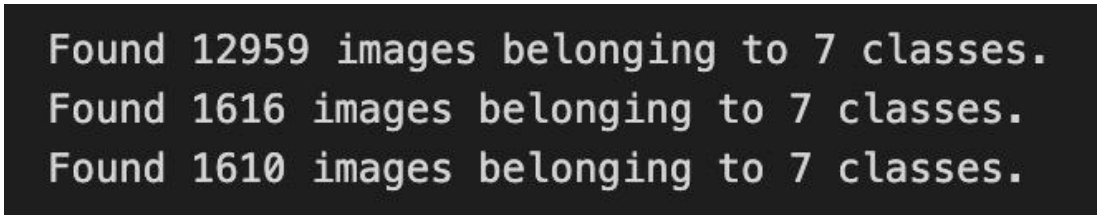


Figure 4.4: Dataset Train, Test, & Validation Portions

4.4 Popular DCNN Architectures Implemented in this Research

Over the past few decades, there have been many well-known Deep Convolutional Neural Network architectures that have been developed by researchers in order to solve Computer Vision classification problems. These architectures innovate to become increasingly lengthy throughout the years, resonating with the complexity of the problems being solved. For instance, when the LeNet-5 architecture was published in 1998, it only consisted of 5 layers. Such construction would seem very simple in our present day, however, it was revolutionary back in its early days. Similarly, the AlexNet introduced in 2012 and the VGG-14 launched in 2014 both consisted of 8 and 16 layers respectively. [74]

Nowadays, such architectures are widely available and implementable to the public thanks to various popular Deep Learning libraries and APIs like Tensorflow and PyTorch; both of which are developed by well-reputed technology companies like Google and Meta. This subchapter will elaborate on the few well-known Deep Convolutional Neural Network architectures that are used in this project, and are thereby also available in the Tensorflow API.

4.4.1 ResNet-50 (2015)

Conventionally when building a Neural Network, Machine Learning developers tend to stack more and more layers with a goal of improving their model's performance. As mentioned in the previous Convolution Layer subchapter, adding more layers could indeed help solve complex problems with more efficiency as different layers could be used to learn different features. However, adding too many layers to the network could also cause its accuracy levels to get saturated, and eventually the model's performance will deteriorate on both training and testing datasets. Overfitting is not the culprit of this issue, but it is rather the network or functions' initialization, or even the infamous vanishing or exploding gradient problems. Deep Residual Networks are aimed to tackle such unprecedented problems, implementing the use of residual blocks in order to increase the performance of the models. This method utilizes the concept of "Skip Connections" in the core of its residual blocks, enabling the model to learn an identity function on its own. In return, this prevents the deeper level layers from performing worse than that of the earlier layers.

Additionally, it also improves the efficiency of training deeper networks while minimizing the percentage of errors [75].

The term “ResNet” is a short abbreviation for Residual Network, consisting of 50 layers. Hence, it is where the name ResNet-50 originates from. Specifically, the architecture consist of the following chronological layers [76]:

1. A Convolution Layer:
    a. 7 × 7 Kernel with 64 Filters
2. A Max Pooling Layer
3. 9 Convolution Layers; the following 3 configurations iterated 3 times:
    a. 3 × 3 Kernel, 64 Filters
    b. 1 × 1 Kernel, 64 Filters
    c. 1 × 1 Kernel, 256 Filters
4. 12 Convolution Layers; the following 3 configurations iterated 4 times:
    a. 1 × 1 Kernel, 128 Filters
    b. 3 × 3 Kernel, 128 Filters
    c. 1 × 1 Kernel, 512 Filters
5. 18 Convolution Layers; the following 3 configurations iterated 6 times:
    a. 1 × 1 Kernel, 256 Filters
    b. 3 × 3 Kernel, 256 Filters
    c. 1 × 1 Kernel, 1024 Filters
6. 9 Convolution Layers; the following 3 configurations iterated 3 times:
    a. 1 × 1 Kernel, 512 Filters
    b. 3 × 3 Kernel, 512 Filters
    c. 1 × 1 Kernel, 2048 Filters
7. An Average Pooling Layer

This architecture was first introduced in a 2015 paper, “Deep Residual Learning for Image Recognition”, written by four brilliant researchers. However, it quickly gained much attention amongst the Artificial Intelligence field when it achieved the top rank in the 2015 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), with just a little over 3.5% error in its final result. Additionally, it also won the ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation within the ILSVRC & COCO competitions in the same year. [75]

4.4.2 Inception V3 (2015)

The Inception V3 is a Deep Convolutional Neural Network architecture with 42 layers that was developed and released by a team at Google in 2015, and is mainly applied for Computer Vision applications to classify images. As the name suggests, it is the third and more superior version of the Inception V1 model that was released a year before. Compared to its predecessors, it is much more optimized, efficient, less computationally costly, however its speed is not as compromised as it consists of more layers [77].

Similar to the ResNet-50, the Inception V3 model aims to solve a specific issue when it comes to stacking multiple layers of Convolution on a Deep Neural Network, causing the model's performance to drop. If the ResNet-50 aims to resolve issues of vanishing or exploding gradient problems, the Inception V3 was developed with a goal of preventing overfitting of the model. In order to achieve so, the Inception architecture utilizes multiple filters of Convolution with different sizes on every layer. Therefore, instead of having deeper layers, the model has more parallel layers, making it "wider" rather than "deeper". In addition, the model has the ability to reduce the dimension of its features much more than its competitors in its time, by factoring larger Convolutions into smaller ones. For example, instead of having a 5 × 5 Convolution layer that may be computationally inefficient, the model breaks it down to a pair of 3 × 3 Convolution layers. On top of that it is also able to split a 3 × 3 Convolution into a 3 × 1 Convolution and a 1 × 3 Convolution. Upon inspection, the 2-layer alternative is able to impressively achieve about 33% more efficiency on average [77].

Likewise, the figure below illustrates the Inception V3 overall architecture:

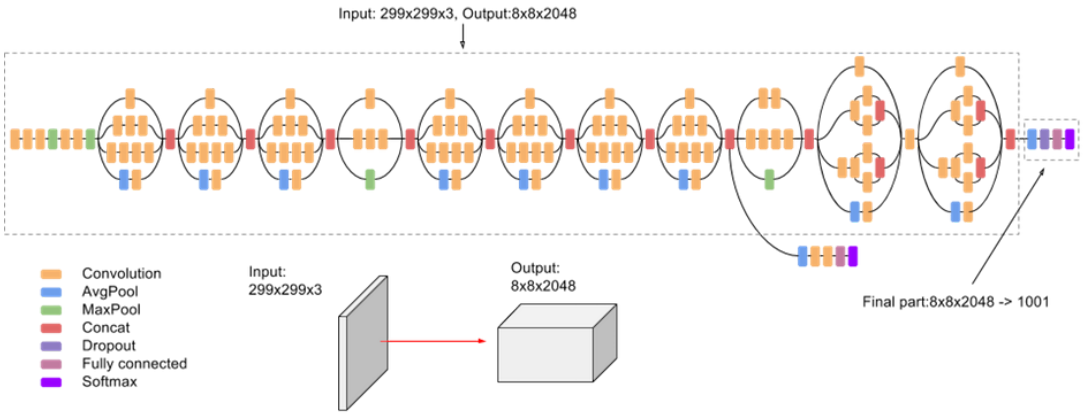


Figure 4.5: Inception V3 Architecture [77]

4.4.3 DenseNet 201 (2017)

The DenseNet is a Deep Neural Network architecture with a combined total of 201 layers. It was jointly developed by Facebook AI Research (FAIR), Cornwell University, and Tsinghua University, and was first introduced to the public through a 2017 Conference on Computer Vision and Pattern Recognition (CVPR) paper, which also attained the Best Paper award with over 2000 citations [78].

Unlike the ResNet and Inception architectures, DenseNet was designed based on the foundational theory that Convolutional Neural Networks with more layers are able to achieve better accuracy and efficiency results if the Neurons between these layers have shorter connections with one another [79]. Hence, it is where the term Dense Neural Network originates from. Thus, having more “dense” layers compared to its rival architectures at the time, it was able to yield better accuracy performance. The model is able to do so thanks to bits of its architecture that differs from its opposing models, like the ResNet, or any standard Deep Convolutional Neural Network layout, as explained in the following Figures and paragraphs [78]:

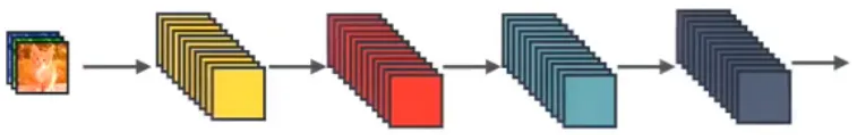


Figure 4.6: A Standard Deep Convolutional Neural Network Architecture [78]

In a typical DCNN layout, the input image is passed through multiple layers of Convolutions to extract different high-level features from it [78].

Figure 4.7: A ResNet Architecture [78]

Additionally, in a ResNet, identity mapping is added between the connections of such Convolution layers, using Element-Wise Addition. This allows the algorithm to pass a state from one layer to another to promote the gradient propagation [78].

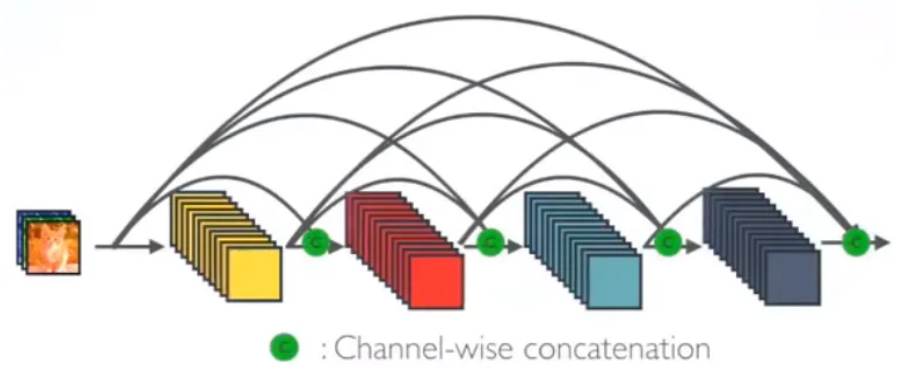


Figure 4.8: A DenseNet Architecture [78]

Alternatively in a DenseNet, each layer collects additional inputs from all of its preceding layers and feeds its own feature maps to all succeeding layers ahead. This utilizes a method known as Concatenation, which translates that each layer gains "collective knowledge" from all its previous layers [78].

Therefore due to this design, and in contrast to the Inception V3 architecture, the network may require a lesser amount of channels, and are thereby thinner and compact. In return, it will consume much less computational and memory resources for training. On top of that, the DenseNet offers a Strong Gradient Flow; error signals are propagated more seamlessly and directly to the preceding layers, which

allows direct supervision from the final classification layer too. Moreover, since each layer collects inputs from the previous layers, this yields more diversified features, thus more patterns could be potentially uncovered by the model. With regards to complexity, the DenseNet classifier adopts features of all complexity levels, unlike traditional DCNNs that only handles the most complex features. Thereby, this yields smoother decision boundaries, allowing it to perform well even if the training data fed into it isn't sufficient [78].

4.4.4 DenseNet 201 (Pre-Trained ImageNet)

The ImageNet Dataset is an open source image database that is based on the WordNet hierarchy, whereby each node consists of hundreds up to thousands of pictures [80]. Over the years, this public dataset had been widely known in the Computer Vision community, assisting individuals or even laboratory researchers with vast amounts of datasets for various Machine Learning tasks, such as Object Detection. Likewise, the database is continuously innovated to address common consumer's needs [81]. With regards to this project, an alternative DenseNet 201 architecture that was Pre-Trained on the ImageNet was also included. This Pre-Trained model carrying ImageNet weights will perform Transfer Learning on our dataset.

Figure 4.9: ImageNet Dataset [82]

4.4.5 Xception (2016)

The Xception is a Neural Network consisting of 71 layers with roughly 23 million parameters, which was inspired by the earlier Inception V3 architecture, both of

which are developed by Google [83]. In a nutshell, this newer architecture incorporates a mechanism known as Depthwise Separable Convolutions, which is technically the Inception model with "the largest maximum number of towers". As a result, the Xception was able to defeat the VGG-16, ResNet, as well as the Inception V3, which were amongst the most superior models prior to its invention, in image classification competitions [84].

	Top-1 accuracy	Top-5 accuracy
VGGNet – 1st Runner Up in ILSVRC 2014 — **VGG-16**	0.715	0.901
ResNet – Winner in ILSVRC 2015 — **ResNet-152**	0.770	0.933
Inception-v3 – 1st Runner Up in ILSVRC 2015 — **Inception V3**	0.782	0.941
Xception	**0.790**	**0.945**

Figure 4.10: Xception Outperforming Other Top Architectures [85]

As an alternative to traditional Convolutions, the Depthwise Separable Convolutions are expected to be much more efficient in terms of computation time. In a traditional Convolution of a coloured image, we respect the following formula [84]:

$$K^2 \times d^2 \times C \times N$$

Legends:

1. K: 2-Dimensions of Image Pixels
2. d: 2-Dimensions of each Convolution Kernel
3. C: 1 Dimension for the Image's color
4. N: The number of Convolution Kernels

Therefore, a traditional Convolution is typically a costly process. Alternatively, the Depthwise Separable Convolution mechanism was invented to address such issues. The method could be broken down into a 2 fold process; Depthwise and Pointwise Convolutions [84].

In Depthwise Convolution, instead of having to convolve the image on every color channel, we could do it for only one channel at a time. This transforms the Convolution formula to [84]:

$$K^2 \times d^2 \times 1$$

Therefore, the resulting Convolved feature will have a volume of K × K × C, as opposed to the K × K × N if a traditional Convolution method had been applied [84].

Next, a Pointwise Convolution, which is just a traditional Convolution with the size of 1 × 1 × N, is introduced to the previous K × K × C volume. This will result in the same outcome if a traditional Convolution had been implemented from the beginning. However, using the Depthwise Separable Convolution mechanism, we were able to minimize the number of operations to a factor of $\frac{1}{N}$ [84].

Therefore, the Xception architecture incorporates such a mechanism backwards; implementing Pointwise Convolutions first, followed by Depthwise Convolutions. This modification was inspired by a foundation in the earlier Inception V3 module, where the smaller Convolutions have to be operated first [85].

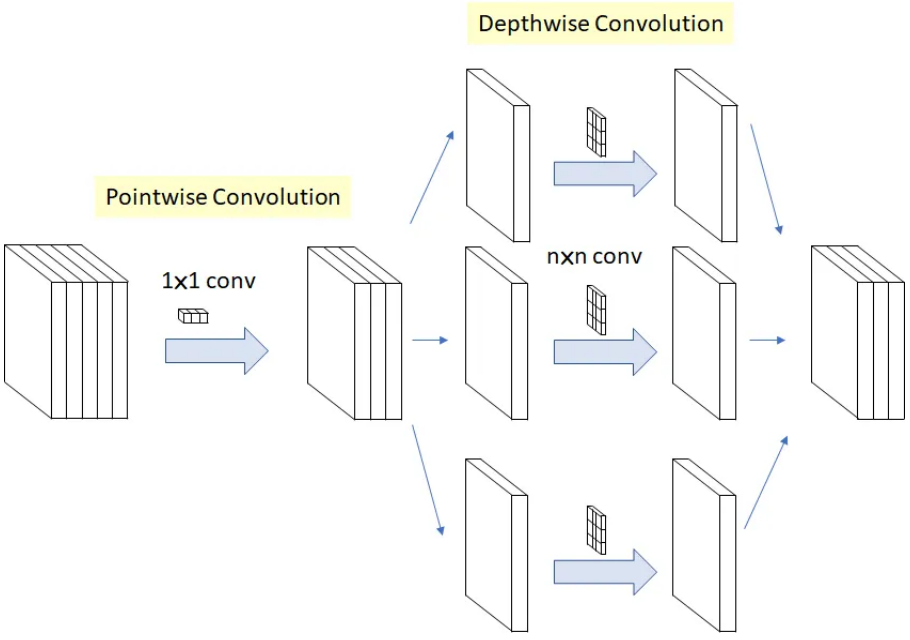


Figure 4.11: Pointwise & Depthwise Convolutions in an Xception Architecture [85]

Likewise, the following figure illustrates the overall Xception Architecture:

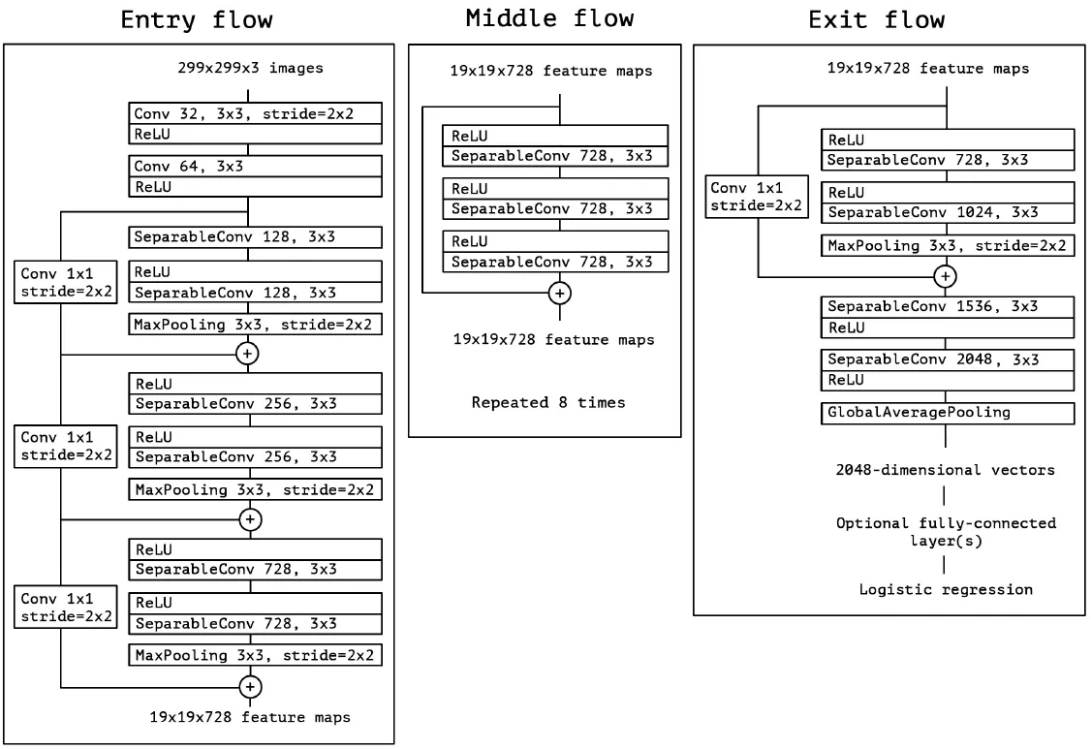


Figure 4.12: An Xception Architecture [85]

4.4.6 Custom-Built DCNN Architecture

On top of the 5 previously mentioned popular architectures used for this research, we have also constructed our own custom Deep Convolutional Neural Network architecture as a part of our comparison. This architecture comprises 19 of the following layers:

1. Convolutional Layer
    a. 3 ✕ 3 Kernels with 32 Filter
    b. ReLU Activation Function
2. Batch Normalization Layer
3. Convolutional Layer
    a. 3 ✕ 3 Kernels with 32 Filter
    b. ReLU Activation Function
4. Batch Normalization Layer
5. Max Pooling Layer
    a. Pool Size of 2 ✕ 2
6. Convolutional Layer:
    a. 3 ✕ 3 Kernels with 64 Filter
    b. ReLU Activation Function

7. Batch Normalization Layer
8. Convolutional Layer:
    a. 3 × 3 Kernels with 64 Filter
    b. ReLU Activation Function
9. Batch Normalization Layer
10. Max Pooling Layer:
    a. Pool Size of 2 × 2
11. Convolutional Layers:
    a. 3 × 3 Kernels with 128 Filter
    b. ReLU Activation Function
12. Batch Normalization Layer
13. Convolutional Layers:
    a. 3 × 3 Kernels with 128 Filter
    b. ReLU Activation Function
14. Batch Normalization Layer
15. Max Pooling Layer:
    a. Pool Size of 2 × 2
16. Flatten Layer
17. Dropout Layer:
    a. Fraction of 0.2
18. Dense Layer:
    a. ReLU Activation Function
19. Dense Layer (Output Layer):
    a. Softmax Activation Function

The intuition to this architecture is based solely on the preprocessed input car images' sizes, which are 100 × 100 × 3. To make it simpler to comprehend, this architecture is broken down into 4 sections. The first part functions to extract the low-level characteristics of cars; such as basic strokes, edge detection, and colors. Moving forward, the second part improvises and recognizes the mid-level patterns; for instance, their grille layout, rim sizes & designs, hood and trunk lengths, windshield angles, and many more. Next, the third part concentrates on its high-level elements; gaining knowledge of their overall body structure, which may contain feature components from the previous part. Finally, the last portion aims to compile

the weights learned by the model, removing insignificant Neurons, and thereby generating the final predictions. Likewise, the details of each part, including its layers will be discussed below.

First Part

As shown in the overall architecture, the first layer is a 2-Dimensional Convolution with a 3 × 3 Kernel size, and consisting of 32 Filters. The premise of selecting this Kernel size is that upon research, it is among the most commonly used Kernel size for Convolutional Neural Networks. Therefore, this Kernel layout will be preserved throughout the whole architecture. Next, since we are extracting just low-level features, it is ideal to begin with a not too significant number of Filters; 32. Additionally, the Rectified Linear Unit (ReLU) Activation Function is chosen for this layer, returning a continuous value between 0 to infinity as its weights. Other parameters such as Padding are set to be the same, to make sure that the output dimensions remain the same as the input dimensions to prevent them from exponentially decreasing in sizes over the next Convolutional and Pooling layers. Similar to the Kernel sizes, this Activation Function and Padding configuration will be maintained throughout all the Convolutional layers in this architecture, except for the last layer which utilizes the Softmax Activation Function to return the probabilities of each category prediction. Nonetheless, the output size of this Convolution is 100 × 100 × 32.

Then, Batch Normalization layer is performed to standardize the output weights of the previous layer for each mini-batch, so that it could be used as the input for the next layer. This process is necessary to stabilize the overall network's training process so that it requires fewer training cycles to achieve similar results [86]. Again this step maintains an output size of 100 × 100 × 32.

Next, the two previously mentioned layers are repeated another time. Finally, the last layer in this part is a Max Pooling layer of Kernel sizes 2 × 2. As mentioned in earlier chapters, this downsizes the images by a factor of its Kernel dimensions, which in this scenario leads to an output size of 50 × 50 × 32.

Second and Third Part

Moving on, the second and third parts of the network each consist of the same 5 layers from the first part. The only differences are, since these parts attempt to extract the mid and high level features, this means that we would need to increment the number of Filters for their Convolution layer, having 64 and 128 Filters respectively. Hence, the resulting dimension for the second and third layers are of 25 × 25 × 64 and 12 × 12 × 128 respectively. Again this seems much smaller than the input sizes as each part is also adjourned with a Max Pooling layer.

Fourth Part

Finally, the last portion of the architecture comprises a Flatten layer, followed by a Dropout layer, Dense layer, another Dropout layer, and the last Dense layer to compute the class probabilities.

The Flatten layer aims to transform all 2-dimensional vectors of weights learned from the previous parts, into a single continuous linear array [87]. This transforms the output size into 18432 × 1. The functionality of a Flatten layer is depicted in the picture below:

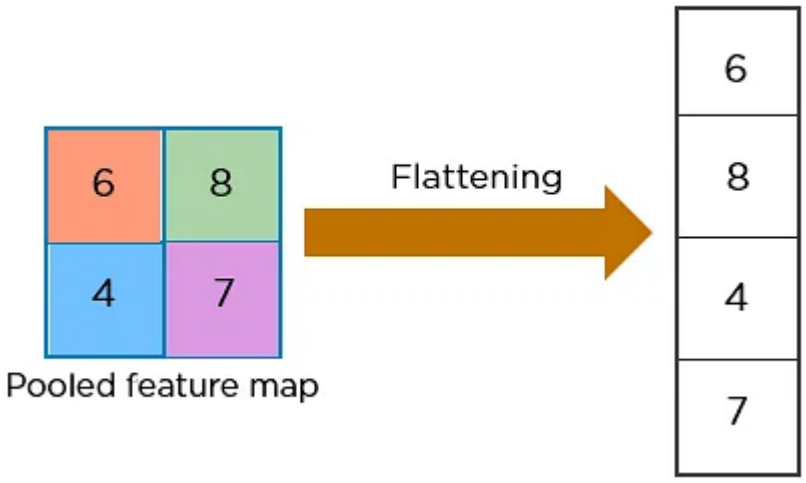


Figure 4.13: Flatten Layer in Neural Network [87]

Upon flattening the weights, a Dropout layer is succeeded to “kill” 20% of the insignificant Neurons, by replacing them with empty values and disconnecting them from the network. This layer is necessary to prevent overfitting on the training data [88]. Next, a Dense layer having 1024 units and ReLU Activation Function is added

to transform the output size into 1024 × 1. Then, another Dropout layer is implemented again to "execute" another 20% of the insignificant Neurons. Finally, the last Dense Layer is adopted to generate the probabilities for each category, using the Softmax Activation function. The class with the highest computed probability will be the prediction category of the input image.

Overall, the whole architecture could be modeled in the diagram below:

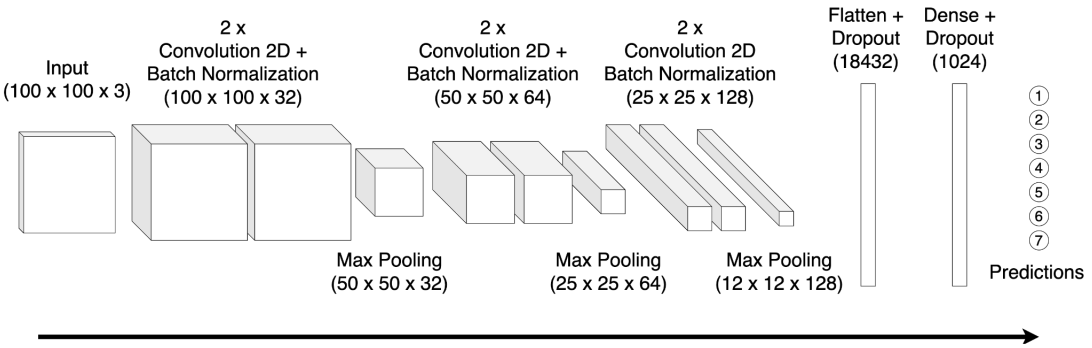


Figure 4.14: Custom-Built DCNN Architecture Overview

4.5 Model Training and Selection

Upon constructing each of the 6 previously-mentioned DCNN architectures, they are then trained & tuned on behalf of the training & validation datasets with 100 training cycles, or more commonly referred to as Epochs. Next, predictions are generated and evaluated for each model with regards to the test dataset. From here, we can compare the performances amongst different architectures, and thereby conclude the best performing model.

4.6 Model Evaluation Techniques

As discussed in depth in earlier chapters, since this experiment is a Multiclass Classification problem, and that the dataset used for training is imbalanced, the comparison in performance amongst the different models will be assessed through the Classification Report, consisting of the following metrics:

1. Accuracy Score
2. F1 Score
3. Precision Score
4. Recall Score

Moreover, the following trends will also be evaluated for the Training and Validation sets for each model:

1. Accuracy Score VS Epochs
2. Loss Value VS Epochs

Finally, efficiency is also a crucial measure in development, as it estimates the amount of resources that needs to be allocated in the process. Therefore, the time complexity for each architecture will also be assessed, by observing the Training Times per Epochs trends.

CHAPTER 5: RESULT ANALYSIS

This chapter presents the outcomes of the methodologies used in Chapter 4, interpreting the performances of every Deep Convolutional Neural Network architectures. Based on the results obtained, the best performing model among the architectures will be discussed in the next chapter, Discussion.

5.1 Classification Report

The following tables entail the Classification Reports for each model trained on the preprocessed dataset; listing the Accuracy, F1, Precision, and Recall Scores for each category:

Table 5.1: Classification Report for ResNet-50 Architecture

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.575221	0.416667	0.483271	
MPV	0.817204	0.703704	0.756219	
Sedan	0.697674	0.724638	0.7109	
Sport	0.778523	0.769912	0.774194	
SUV	0.756173	0.862676	0.805921	
Truck	0.803191	0.94375	0.867816	
Wagon	0.666667	0.277778	0.392157	
Overall	0.727808	0.671303	0.684354	0.742236

From the results displayed in table above, the ResNet-50 is able to achieve a merely decent score in predicting the test dataset; with an overall Accuracy Score of 74.2%. Quite noticeably, the F1, Precision, and Recall Scores differ from one category to another. This indicates that the model is able to distinguish one category better than another. Specifically, Trucks have the highest F1 Score, while Wagons have the worst. Both categories also receive the same conclusion for the Recall Score. However for the Precision Scores, MPVs seem to take the lead. Still, the Wagon class is left behind. Overall, the average of F1, Precision, and Recall scores across the predicted results are evaluated with the ‘macro avg’ row, having values of 0.68, 0.73, and 0.67 respectively. Hence, it could be finalized that the ResNet-50 architecture is able to attain a decent level of performance across the metrics.

Table 5.2: Classification Report for Inception V3 Architecture

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.526012	0.583333	0.553191	
MPV	0.584906	0.861111	0.696629	
Sedan	0.735376	0.637681	0.683053	
Sport	0.729831	0.860619	0.789848	
SUV	0.905213	0.672535	0.771717	
Truck	0.872611	0.85625	0.864353	
Wagon	0.5	0.25	0.333333	
Overall	0.693421	0.674504	0.670304	0.729193

In the Classification Report of the Inception V3 model above, it could be noted that it is able to achieve a global Accuracy Score of 72.9%, which is also a decent enough performance like the previously mentioned ResNet-50. As expected, it also yields a range of different F1, Precision, and Recall Scores for the different classes in the predicted dataset. Similarly, Trucks lead the F1 Score, whereas Wagons achieved the poorest performance. Although the MPV class again achieves the highest Recall Score, its Precision scores receive a very contrast performance, and are much poorer than most of the other classes. Regardless, the other categories, such as Hatchback, Sport, and Truck are able to maintain a similar level for their respective metrics. Nonetheless, the Inception V3 model is able to achieve a Precision, Recall, and F1 Scores of hardly 0.7 each. This is also considered to be decent; similar however still poorer than the ResNet-50 if compared side by side.

Table 5.3: Classification Report for DenseNet 201 Architecture

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.589404	0.570513	0.579805	
MPV	0.9	0.666667	0.765957	
Sedan	0.772472	0.664251	0.714286	
Sport	0.763265	0.827434	0.794055	
SUV	0.811075	0.876761	0.84264	
Truck	0.797872	0.9375	0.862069	

Wagon	0.421053	0.444444	0.432432	
Overall	0.722163	0.71251	0.713035	0.76087

As depicted in the figure above, the DenseNet 201 architecture is able to attain an Accuracy Score of 76%, which is higher than our previous models. Thus, having a score above 75% allows this description to begin with a good performance. Alternatively, it also has a constant overall F1, Precision, and Recall Scores; between 0.71 and 0.72, which are better than the previous architectures. Looking across the same three metrics across the different classes, most of them also achieved a much better performance compared to the other models. Again, the Wagon class attained the lowest F1, Precision, and Recall Scores. In contrast to that of the Inception V3 model, the MPV class is now superior to the Precision score. Overall, this model's performance can be considered to be good.

Table 5.4: Classification Report for DenseNet 201 (Pre-Trained ImageNet) Architecture

	Precision	**Recall**	**F1 Score**	**Accuracy**
Hatchback	0.407407	0.423077	0.415094	
MPV	0.666667	0.685185	0.675799	
Sedan	0.586517	0.630435	0.607683	
Sport	0.72028	0.683628	0.701476	
SUV	0.683502	0.714789	0.698795	
Truck	0.84507	0.75	0.794702	
Wagon	0.291667	0.194444	0.233333	
Overall	0.600158	0.58308	0.589555	0.645963

In comparison to the regular DenseNet 201 architecture, its Pre-Trained ImageNet model apparently attained a considerably poor performance upon Transfer Learning. This could be proven from its Accuracy and Precision Scores of just 64% and 60% respectively. Both its F1 and Recall Scores also barely reached 59%. Nevertheless, the ranking distribution of F1, Precision, and Recall Scores seemed to be maintained across the classes, although its performances are much worse than its raw architecture. Regardless of its terrible performance, the model did not fail on the test dataset since it is still able to achieve over 50% for all of these metrics' scores.

Table 5.5: Classification Report for Xception Architecture

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.640625	0.262821	0.372727	
MPV	0.796117	0.759259	0.777251	
Sedan	0.787425	0.635266	0.703209	
Sport	0.674457	0.893805	0.768792	
SUV	0.793333	0.838028	0.815068	
Truck	0.878788	0.90625	0.892308	
Wagon	0.355556	0.444444	0.395062	
Overall	0.703757	0.677125	0.674917	0.738509

Moving forward with the Classification Report for Xception model, we now have a runner up to the leading DenseNet 201 architecture in terms of Accuracy Score; this model attained an Accuracy of 73.8% for its prediction results. Peeking closely to the Precision, Recall, and F1 Scores however, there is a contrasting order between the classes as compared to previous models. For instance, the Hatchback class now has a much lower Recall score, even lower than the Wagon category (which was well known for having the worst Recall scores since the beginning). Otherwise, most of the other classes seem to still maintain a good to tremendous F1 Score. Overall, the model attained an average Precision, Recall, and F1 Scores of 0.70, 0.68, and 0.67 respectively, rewarding it with a decent performance like the ResNet-50 and Inception V3 architectures.

Table 5.6: Classification Report for Custom-Built DCNN Architecture

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.666667	0.371795	0.477366	
MPV	0.793814	0.712963	0.75122	
Sedan	0.622309	0.768116	0.687568	
Sport	0.787952	0.723451	0.754325	
SUV	0.778502	0.841549	0.808799	
Truck	0.814607	0.90625	0.857988	
Wagon	0.333333	0.138889	0.196078	

Overall	0.685312	0.637573	0.647621	0.726087

Lastly, the Classification Report for our Custom-Built DCNN architecture also reveals a decent performance in terms of its 72.6% Accuracy Score. However, its overall F1, Precision and Recall Scores are not very satisfactory; the macro average Precision is only 0.69, whereas that of its Recall and F1 Scores barely made it to 65%. An extreme observation could also be noticed when looking deeper into such metrics across the different categories. The Wagon class seems to have a very poor Recall and F1 Scores, even worse than that of our least performing model so far, the DenseNet 201 PreTrained architecture. Regardless, thanks to the Accuracy and Precision Scores for being above 70% and 65% respectively, this model's performance can still be classified to be decent.

5.2 Accuracy & Loss Trends

As mentioned in the final subchapter in Chapter 4, the Accuracy Score and Loss Values per Epoch will also be compared amongst the models, thus yielding the following charts for each model:

Accuracy Trends

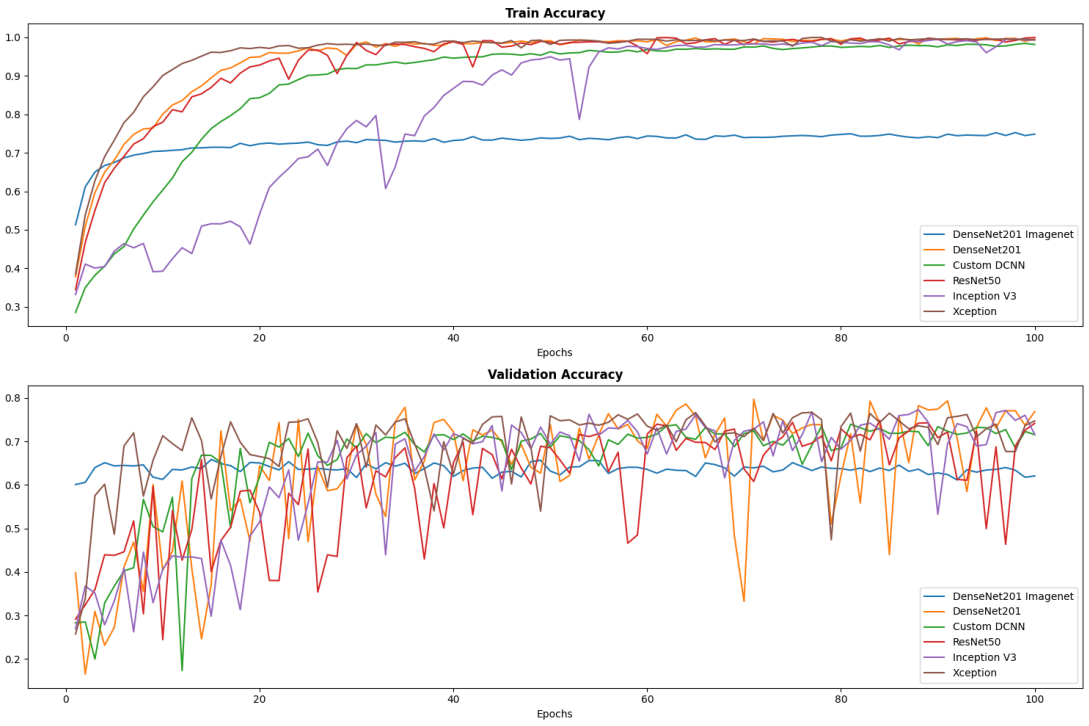


Figure 5.1: Accuracy Trend per Epoch for Each DCNN Architecture

The chart above clearly depicts that every DCNN architecture's Accuracy levels do converge similarly on behalf of their own Epochs iteration. Specifically during training, they all improve and eventually converge sideways as the number of Epochs increases during training. On the other hand, the validation Accuracy displays an unsteady outcome for nearly all the models. This is expected, as they are validation scores, and unlike training scores.

If compared against one another, it could be proven that indeed different architectures do also yield contrasting Accuracy level performances during training. For instance, the ResNet-50, DenseNet 201, and Xception architectures raced for the top, whereas others like the DenseNet 201 (Pre-Trained ImageNet) are left behind. Additionally, the Custom-Built DCNN architecture seems to accelerate in Accuracy level much faster than that of the Inception V3 model in the beginning. Fortunately, the Inception V3 was able to catch up to it upon the 60th Epoch, and ended up performing even better after that.

Loss Trend

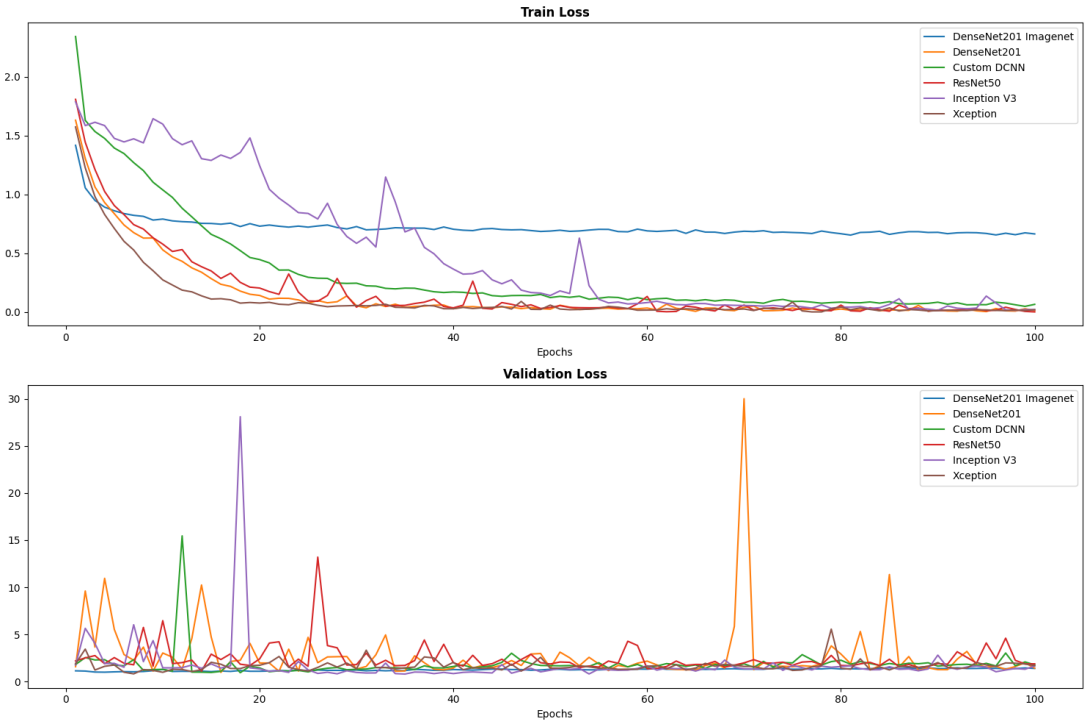


Figure 5.2: Loss Trend per Epoch for Each DCNN Architecture

Like the prior Accuracy charts, the visualization above also presents similar Loss values per Epoch trends for each model; overall converging downward during training. This proves that the architectures optimize themselves as the training iteration increases. For validation, it also follows an up-and-down trend like the Accuracy levels; some architectures, such as the Custom-Built DCNN and Inception V3 have large spikes in the beginning iterations, while others like the DenseNet 201 have them on later stages.

Moving forward with its comparison between different models, it could be seen that again the ResNet-50, DenseNet 201, and Xception architectures are in tight competition to be having the least, while the DenseNet 201 (Pre-Trained ImageNet) converges at a higher Loss value. Comprehensively, these Loss values seem to be the inverse of Accuracy values, and thereby also concludes that different training iteration yields different performance results for each respective model.

5.3 Training Time Efficiency

Since efficiency is crucial for estimating the amount of resources required for development, the following chart visualize the Training Times per Epoch for each of our DCNN architectures:

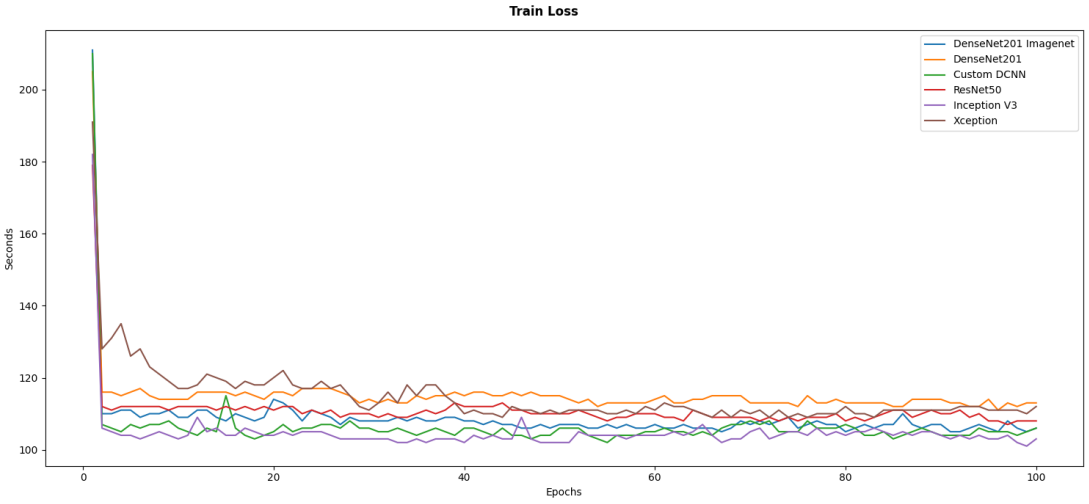


Figure 5.3: Training Time Trend per Epoch for Each DCNN Architecture

The plot above proves that different DCNN architectures built with different algorithmic ‘depths’, ‘widths’, and structures do indeed require contrasting training

times. Overall, the training times seem to peak for all the models during the first training iteration, and eventually stabilize themselves right after. Also, nearly all the models maintained a training time range of between 100 and 120 seconds.

When added up, the time it took to construct, train, predict, & evaluate each DCNN architecture are concluded in the total Execution Time table below:

Table 5.7: DCNN Architectures' Execution Times

DCNN Architecture	Total Execution Time (Seconds)	Total Execution Time (Hours)
ResNet-50	10654.07	2.96
Inception V3	10644.52	2.96
DenseNet 201	11721.9	3.26
DenseNet 201 (Pre-Trained ImageNet)	11561.97	3.21
Custom-Built DCNN Architecture	9826.47	2.73
Xception	11712.98	3.25

It could be seen that this time, the Custom-Built DCNN takes the lead for the most time efficient results, followed by the Inception V3, ResNet-50, and DenseNet 201 (Pre-Trained ImageNet) architectures. Alternatively, the Xception and DenseNet 201 architectures seem to be having longer total Execution Times.

5.4 Impact of Removing Background

As promised in earlier chapters, this research will also compare the experiments' results without their dataset image background removed during the preprocessing stage. The goal of this evaluation is to indicate whether or not removing background, or in other words noise from our raw data, has any impact on each of our Deep Convolutional Neural Network architectures' performances. The records to be compared are Classification Report, Accuracy & Loss trends, as well as the Time Efficiency of these models.

5.4.1 Classification Report

Table 5.8: Classification Report for ResNet-50 Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.442308	0.294872	0.353846	
MPV	0.677083	0.601852	0.637255	
Sedan	0.569444	0.594203	0.58156	
Sport	0.639839	0.726027	0.680214	
SUV	0.671233	0.690141	0.680556	
Truck	0.730061	0.74375	0.736842	
Wagon	0.222222	0.095238	0.133333	
Overall	0.564599	0.535155	0.543372	0.620474

Table 5.9: Classification Report for Inception V3 Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.12987	0.064103	0.085837	
MPV	0.604651	0.240741	0.344371	
Sedan	0.390282	0.601449	0.473384	
Sport	0.467925	0.56621	0.512397	
SUV	0.497585	0.362676	0.419552	
Truck	0.616822	0.4125	0.494382	
Wagon	0	0	0	
Overall	0.386734	0.321097	0.332846	0.438202

Table 5.10: Classification Report for DenseNet 201 Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.56383	0.339744	0.424	
MPV	0.702703	0.722222	0.712329	
Sedan	0.634529	0.683575	0.65814	
Sport	0.685771	0.792237	0.735169	
SUV	0.836207	0.683099	0.751938	
Truck	0.854305	0.80625	0.829582	
Wagon	0.258065	0.380952	0.307692	
Overall	0.647915	0.629726	0.631264	0.686642

Table 5.11: Classification Report for DenseNet 201 (Pre-Trained ImageNet) Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.417476	0.275641	0.332046	
MPV	0.457364	0.546296	0.49789	
Sedan	0.473469	0.560386	0.513274	
Sport	0.663067	0.700913	0.681465	
SUV	0.725191	0.334507	0.457831	
Truck	0.552036	0.7625	0.64042	
Wagon	0.184615	0.285714	0.224299	
Overall	0.496174	0.495137	0.478175	0.543071

Table 5.12: Classification Report for Xception Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.468085	0.282051	0.352	
MPV	0.67	0.62037	0.644231	
Sedan	0.529307	0.719807	0.610031	
Sport	0.68913	0.723744	0.706013	
SUV	0.777202	0.528169	0.628931	
Truck	0.722543	0.78125	0.750751	
Wagon	0.473684	0.214286	0.295082	
Overall	0.618565	0.552811	0.569577	0.630462

Table 5.13: Classification Report for Custom-Built DCNN Architecture (Without Background Removed)

	Precision	Recall	F1 Score	Accuracy
Hatchback	0.325581	0.179487	0.231405	
MPV	0.689189	0.472222	0.56044	
Sedan	0.502128	0.570048	0.533937	
Sport	0.541401	0.776256	0.637899	
SUV	0.690583	0.542254	0.607495	
Truck	0.813559	0.6	0.690647	

Wagon	0	0	0	
Overall	0.50892	0.44861	0.465975	0.564919

From the Classification Reports listed above, the changes in performance when the images' background were maintained on our collected dataset may not be visually interpretable. Thus the following tables & figures concludes the change of the two most crucial metrics in the Classification Reports; the Accuracy & F1 Scores:

Table 5.14: Accuracy Scores Comparison (With & Without Background Removed)

Model	Accuracy Score - With Background Removed	Accuracy Score - Without Background Removed	Change
ResNet-50	0.742236	0.620474	-12.1762
Inception V3	0.729193	0.438202	-29.0991
DenseNet 201	0.76087	0.686642	-7.4228
DenseNet 201 (Pre-Trained ImageNet)	0.645963	0.543071	-10.2892
Custom-Built DCNN Architecture	0.726087	0.564919	-16.1168
Xception	0.738509	0.630462	-10.8047

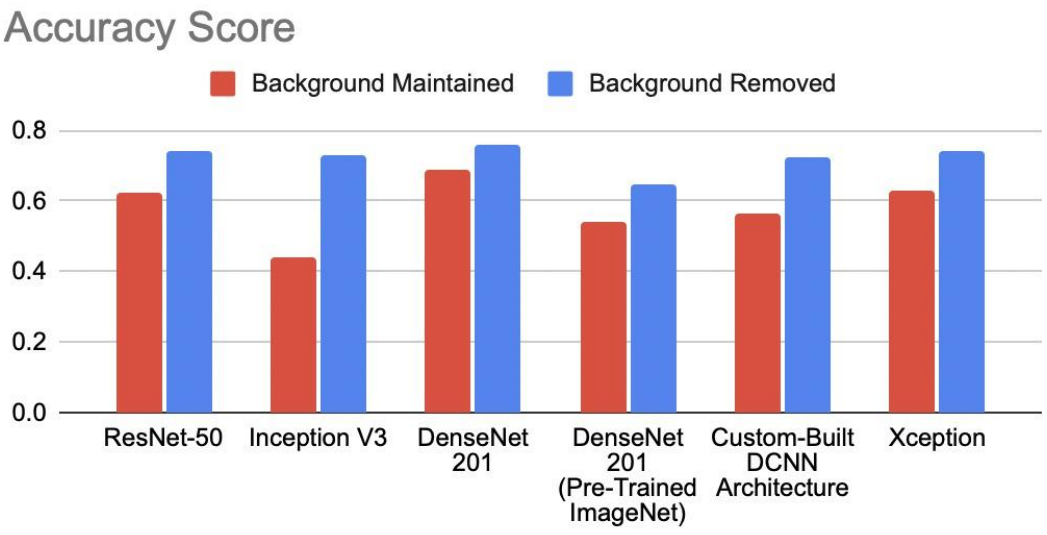


Figure 5.4: Accuracy Scores Comparison (With & Without Background Removed)

Table 5.15: F1 Scores Comparison (With & Without Background Removed)

Model	F1 Score - With Background Removed	F1 Score - Without Background Removed	Change
ResNet-50	0.684354	0.543372	-14.0982
Inception V3	0.670304	0.332846	-33.7458
DenseNet 201	0.713035	0.631264	-8.1771
DenseNet 201 (Pre-Trained ImageNet)	0.589555	0.478175	-11.138
Custom-Built DCNN Architecture	0.647621	0.465975	-18.1646
Xception	0.674917	0.569577	-10.534

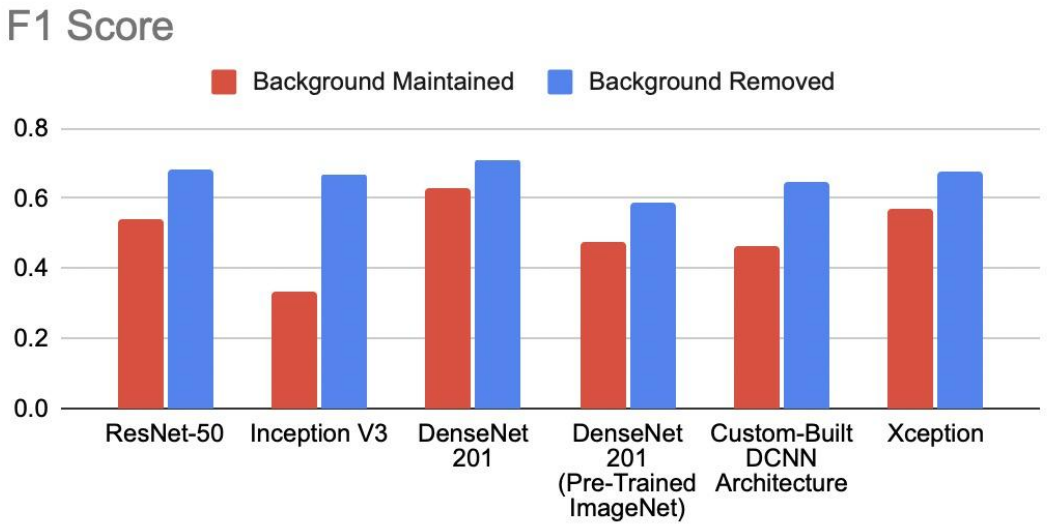


Figure 5.5: F1 Scores Comparison (With & Without Background Removed)

From the aggregated tables & charts above, we could clearly determine that there is a significant decrease in performance when the images background is maintained on our collected dataset. On average, the Accuracy and F1 Scores depleted by 14% and 16% respectively; which could be most notably depicted from the Inception V3 architecture. Even our best performing model, the DenseNet 201, which previously scored an Accuracy of 76%, only managed to retain a mere 68.7%. Therefore, from this comparison alone, we could conclude that neglecting irrelevant features from our dataset does indeed improve the performances of the model trained on it. Hence, this is a very crucial step to be considered for Computer Vision tasks.

5.4.2 Accuracy Trend per Epoch

In the records presented in Appendix A, we could depict that maintaining the images' backgrounds do have varying impacts on the training and validation performances of our DCNN architectures. For some, such as the DenseNet 201 (Pre-Trained ImageNet) model, the Accuracy level converges at a much lower value throughout its training process. Other models, like the Inception V3, seem to be taking more iterations to learn from the noisier dataset. Surprisingly, the other models seem to converge to a similar Accuracy level at an identical rate, regardless if the image background is removed or maintained.

Looking into the validation performance, however, they are a completely different story. All the models except the DenseNet 201 architecture seem to score a noticeably lower Accuracy level for nearly every Epoch iteration. Nonetheless, the DenseNet 201 architecture still forfeits a small percentage of Accuracy when the images' backgrounds are kept.

In conclusion, removing our dataset image backgrounds does not necessarily improve the model's training performance (depending on the architecture), however, it always increases their ability to predict accurately on the validation dataset. Therefore, removing such irrelevant noises from our dataset features is still the best move.

5.4.3 Loss Trend per Epoch

Based on the charts in Appendix B, we could again determine a very inversely proportional performance as to that depicted in the Accuracy score charts. During training, it could be seen that the DenseNet 201 (Pre-Trained ImageNet) model yielded significantly higher Loss values when the image backgrounds are maintained. On the other hand, the Inception V3 also takes a longer time to lower its Losses. The other architectures took a very coinciding amount of iteration to converge into their optimal Loss value; which are again very close regardless if the image backgrounds are removed or not.

For the validation performance, only the DenseNet 201 (Pre-Trained ImageNet) returns a very noticeable increase in Loss values throughout its training iteration. Surprisingly, the other architectures maintained similar Loss value results. Some models, like the DenseNet 201 and Custom DCNN architectures have Loss spikes for some Epoch iteration when the image background is removed, and surprisingly even more of those when the image background is maintained. Regardless, the Inception V3 and Xception networks have higher and more unstable Loss values without their dataset image background removed.

Hence, this concludes that depending on the architectures, removing background on our sample dataset does not fundamentally affect its Loss values while training nor validation.

5.4.4 Training Time Efficiency

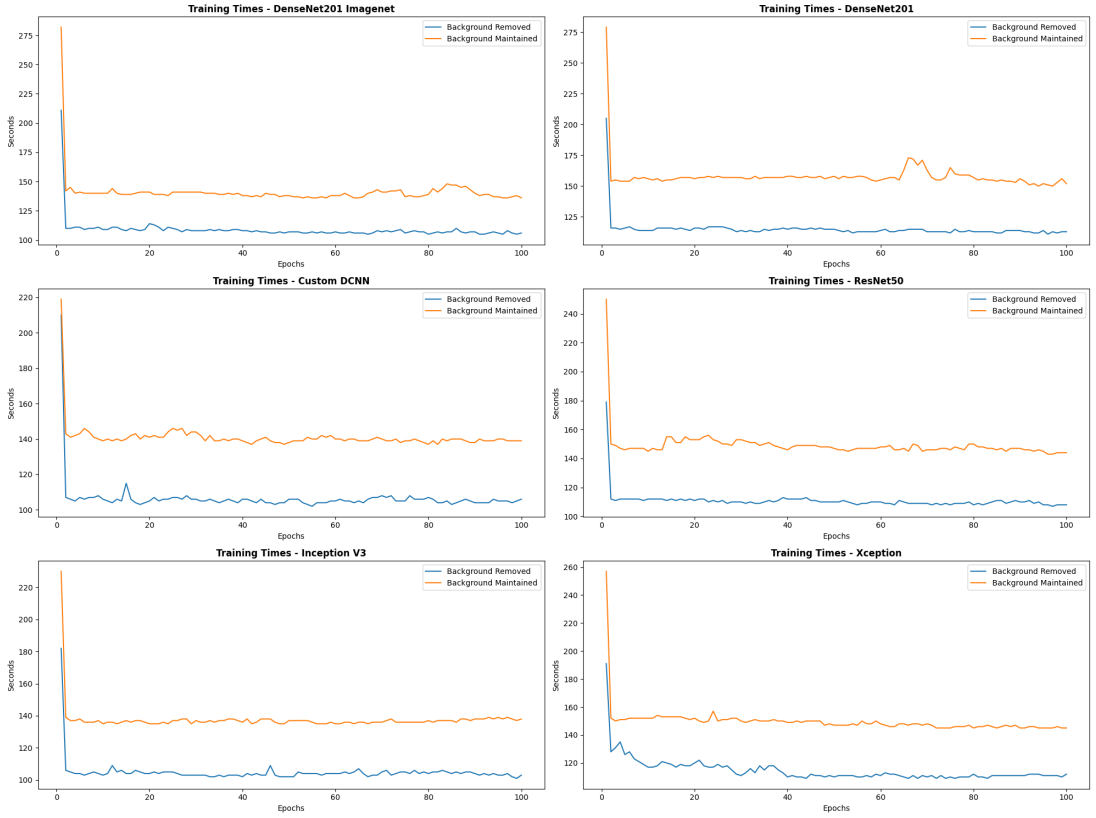


Figure 5.6: Training Time Trends per Epoch Comparison (With & Without Background Removed)

The diagrams above project the training time per Epoch records for each of our DCNN architectures, both with and without their dataset image backgrounds removed. Overall, it could be clearly depicted that undoubtedly all the models are

able to train much faster with the backgrounds removed. Looking closely, they are able to train less than 120 seconds without these unnecessary features, and would take up to over 150 seconds otherwise.

Therefore, the total Execution Times for each DCNN architecture, with & without its input features' backgrounds removed, could be compared through the following table and visualization:

Table 5.16: Execution Times Comparison (With & Without Background Removed)

Model	Training Time (Hours) - Without Background Removed	Training Time (Hours) - With Background Removed	Change (%)
ResNet-50	4.27	2.96	-44.26
Inception V3	3.84	2.96	-29.73
DenseNet 201	4.45	3.26	-36.50
DenseNet 201 (Pre-Trained ImageNet)	3.98	3.21	-23.99
Custom-Built DCNN Architecture	3.94	2.73	-44.32
Xception	4.25	3.25	-30.77

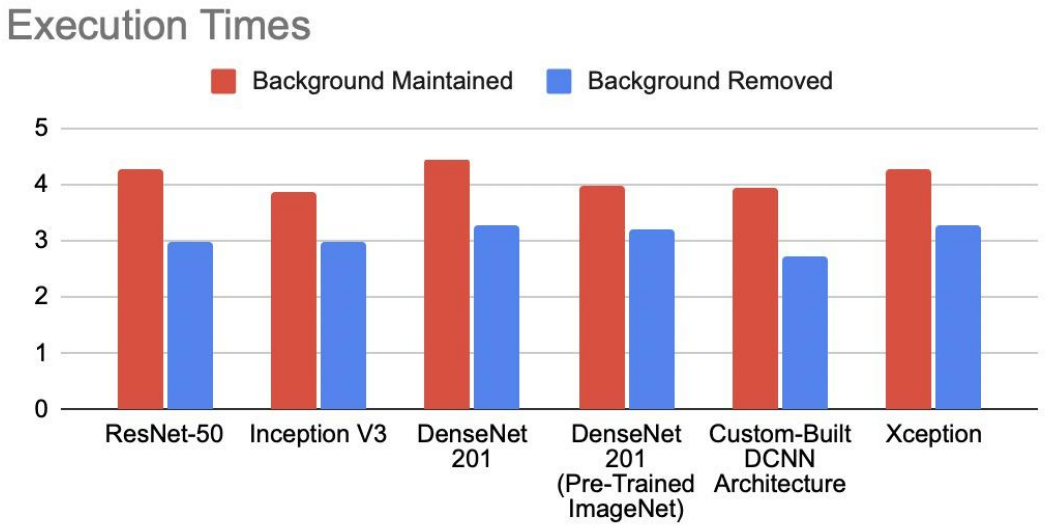


Figure 5.7: Execution Times Comparison (With & Without Background Removed)

Comparing the results summarized from table and figure above, we could clearly see a significant reduction in the total Execution Times for every single model when its

dataset image backgrounds were removed; decreasing by a range of 24 - 44% depending on the architecture. Therefore, just like the previous comparisons, this proves that removing the background, or in other words removing noise from our dataset features do have a positive impact on our Execution Times' efficiency as well.

CHAPTER 6: DISCUSSION

6.1 Final Results Evaluation

As foreshadowed and presented from the Classification Report tables, it could be noticed that indeed every single architecture is able to achieve different scores across the metrics, especially for different categories. As seen multiple times, the Wagon class had always been achieving the lowest scores. Coincidentally, this is also the class that has the least number of pictures in the imbalanced dataset. Hence, it is the confirmed culprit. On the other hand, Trucks seem to constantly achieve the best records for F1, Precision, and Recall Scores. From a logical and visual perspective, this may be due to the fact that they are usually much larger than other vehicles, or that they almost always have a very distinguishable "extended rear trunk" layout.

Likewise, to conclude which is the best performing model from just these metrics alone, we can begin by finding the top scoring model for each specific metric. The DenseNet 201 architecture takes the lead on behalf of the Accuracy, F1 and Recall Scores; having that of 76%, 0.71, and 0.71 respectively. For the Precision Score, the ResNet-50 only beats the DenseNet 201 by a tiny value of 0.005. Hence from this conclusion, the DenseNet 201 is crowned to be the best performing model through the Classification Reports.

From the Accuracy trend per Epoch during training however, the DenseNet 201 did not converge to its most optimal Accuracy level as quick as the Xception network, however, it still eventually ended up converging to a similar score after about 35 Epochs. On the validation chart, the DenseNet 201 may be a little more unstable than the competing Xception architecture, but it was able to achieve higher than all the other models on many later iterations. This allows the DenseNet 201 architecture to take the lead for this comparison.

Next, on the Loss trend per Epoch during training, it could be clearly seen that the Xception model is able to outperform all of the models in terms of converging speed and optimization level. Regardless, like the previous comparison, both Xception and DenseNet 201 architectures converge to a pretty similar value. For the validation

graph however, the Xception model clearly beats the DenseNet 201, alongside all the other models; with the least and less volatile Loss value. Thus, the triumph is now awarded to the Xception network.

Moving on to the Execution Times, it could be concluded directly that the most time efficient architecture goes to the Custom-Build DCNN; with just a total of 2.73 hours. Leaving the top performing models previously described, Xception and DenseNet 201, at the least; 3.25 and 3.26 hours respectively.

Overall, the DenseNet 201 architecture seems to attain the most promising results, gaining traction from most of our preceding comparison methodologies but Execution Times. The Xception network may seem comparable from the Accuracy and Loss trends, but it scores 4% less in its overall Accuracy score than that of the DenseNet 201, which is in fact a pretty hefty amount. Upon consideration, the DenseNet 201 network shall maintain its crown, as intuitively the final output evaluation metric scores means more than its training processes. Lastly, although the DenseNet 201 comes off as the least Time Efficient model during training, it is merely just 30 minutes less, or equivalent to just below 20 percent slower, than our fastest Custom-Build DCNN architecture. Nonetheless, such tradeoffs cannot justify an 8 percent decrease in performance for the latter model. Hence, the DenseNet 201 architecture is elected as the best performing model in this research.

As for the Impact of Removing Backgrounds of our image dataset prior to feeding them to our DCNN architectures, every comparison pointed to the fact that doing so indeed improves the performances as well as efficiency of all our models. Specifically on average, we have seen the Accuracy and F1 Scores surge by 14% and 16% respectively (most notably from the Inception V3 architecture). On the other hand, the Execution Times have decreased by a range of 24 - 44% depending on the model. Thus, Removing Backgrounds is a crucial and extremely advantageous preprocessing measure for our experiment.

Likewise, all the source codes for the experiments conducted in this research have been attached in Appendices C to I section of the paper.

6.2 Other Discussion

The previous chapters and sections have thoroughly identified our research problem, presented solution methodologies, as well as described their results. Thus, such comprehensive interpretations shall already conclude our research. However, there are also some additional experimentations that have been conducted during the development of this project, but have not been mentioned in earlier sections, hence will be discussed thoroughly in this subchapter.

During the development stage, multiple other techniques were considered as well, but have been archived out of this research. Specifically, there had been other architectures involved in the experiment, such as the MobileNet V3 Large, VGG-19, as well as three other Transfer Learning (Pre-Trained ImageNet) weighted networks, ResNet-50, Inception V3, and Xception. Every single architecture was given more than one attempt for training and prediction, for both with and without their image dataset background removed, in order to have a non-bias output. Upon result inspection and consideration, we have selected only 6 most appropriate models for this research, in terms of evaluation score and architecture layout.

Moreover, the Pooling hyperparameter for these Deep Learning architectures has also been investigated, between No Pooling and Max Pooling. The Average Pooling method is not included, as its shortcomings in feature generalization have been revealed in earlier theoretical chapters. After acknowledging the performance results, we have concluded that most of the models were able to perform better using Max Pooling. Hence, this configuration has been equipped to all our selected architectures.

Additionally, multiple target image dimensions have also been tested prior to feeding them into the Deep Convolutional Neural Networks; 50 × 50, 100 × 100, 150 × 150, and 250 × 250. Upon examining the balance between its evaluation performance and time efficiency records, we have identified that 100 × 100 is the sweet spot for our image target size.

In total, on top of the 12 that were selected and placed into this research (6 different DCNN architectures, each with & without their dataset image background removed), there had been over 50 additional experiments conducted during the development stage, interchangeably utilizing the combination of the above-mentioned configurations.

Finally, there are also computational limitations encountered while developing these advanced Deep Learning architectures. Due to the extensive amount of complexity to train each model, it needs to be carried out with a computer enhanced with GPU (Graphics Processing Unit), otherwise it would take an extremely long time to accomplish them all. Unfortunately, the author of this paper does not have such a powerful setup at home. Thankfully upon research, an online Machine Learning competition platform, Kaggle, provides a free-to-use & collaborative Jupyter Notebook to run Python codes, that has an additional option to be equipped with a GPU or TPU (Tensor Processing Unit) to boost its processing capabilities. Nonetheless, there are still some constraints with the service, that is only one Jupyter Notebook could be run at once, and that there is a weekly usage limit for the GPU & TPU resources. Even so, as shown in the results section of this research, it still takes 2.5 - 3.5 hours to execute each model's complete codes. Hence even with such privilege, it still took months to complete all the experimentations during the development stage of this project.

CHAPTER 7: CONCLUSION

7.1 Research Conclusion

This research incorporates Computer Vision and Deep Learning for multiclass image classification on different automobile types. As revealed in our analysis and discussion, among the 6 different Deep Convolutional Neural Network architectures tested, the DenseNet 201 model is able to achieve the best performance in terms of Accuracy and F1 Scores of 76% and 0.71 respectively. Additionally, this research has also concluded that removing backgrounds from the dataset images, or in other words removing unnecessary noise from our input features, does have a significant positive impact on the model's performances and Execution Times. On average, the Accuracy and F1 Scores surge by 14% and 16% respectively. Furthermore, the Execution Times have decreased by a range of 24 - 44% depending on the architecture. Hence, the results yielded from the conducted experiments has proven that Computer Vision had been successful in classifying different car types, and that Deep Learning could be helpful in automating the process of manually labeling cars, to assist alternative parking layout designs that improves the efficiency of space allocations in modern parking areas, and thereby indirectly contribute to reducing parking congestion.

7.2 Recommendation

Although the solution of this experiment has been proven functional and thereby successful in assisting our research problem, its top performance metrics still lies in the seventies, which is considered decent but not exceptionally satisfactory. Hence, the models in this project could still be optimized further in many ways to achieve better performances; such as training it on larger & balanced car datasets, more or newer vehicle models (car manufacturers typically release new models every few years), tuning their hyperparameters, and many more. Likewise, this project is simply just a proof of concept for the extent of Deep Learning and Computer Vision in solving existing general issues; in the case of this research, assisting alternative parking layouts to reduce parking congestion.

REFERENCES

[1] McCoy, K., & TODAY, U. (2017, July 12). *Drivers spend an average of 17 hours a year searching for parking spots*. USA TODAY. https://www.usatoday.com/story/money/2017/07/12/parking-pain-causes-financial-and-personal-strain/467637001/

[2] *Parking lot striping and traffic marking*. (2021, November 19). SealMaster. https://sealmaster.net/parking-lot-striping-traffic-marking/

[3] *What is computer vision?* IBM - United States. https://www.ibm.com/id-en/topics/computer-vision

[4] *What is deep learning?* IBM - United States. https://www.ibm.com/topics/deep-learning

[5] *Cars dataset*. Stanford Artificial Intelligence Laboratory. http://ai.stanford.edu/~jkrause/cars/car_dataset.html

[6] *Who invented the automobile?* (2019., November 19). The Library of Congress. https://www.loc.gov/everyday-mysteries/motor-vehicles-aeronautics-astronautics/item/who-invented-the-automobile/

[7] *The first car, Karl Benz's patent motor car, hits the road*. (2018, July 3). Automotive News. https://www.autonews.com/article/20180703/CCHISTORY/180709892/the-first-car-karl-benz-s-patent-motor-car-hits-the-road

[8] *2014 Volkswagen Golf GTD - Dimensions, car, HD wallpaper.* PeakPX https://www.peakpx.com/en/hd-wallpaper-desktop-fsatr

[9] *You can buy an almost-new Mercedes W124 for $60,000*. (2021, July 1). Team-BHP.com. https://www.team-bhp.com/news/you-can-buy-almost-new-mercedes-w124-60000

[10] *Honda all new city dimensions*. autoX.
https://www.autox.com/new-cars/honda/city/dimensions/

[11] *BMW 5 series dimensions*. autoX.
https://www.autox.com/new-cars/bmw/5-series/dimensions/

[12] *Mercedes-Benz S-class dimensions*. autoX.
https://www.autox.com/new-cars/mercedes-benz/s-class/dimensions/

[13] *Used 2019 FIAT 500 Abarth Specs & Features.* Edmunds.
https://www.edmunds.com/fiat/500/2019/abarth/features-specs/

[14] *2021 Volkswagen Golf Specs & Features.* Edmunds.
https://www.edmunds.com/volkswagen/golf/2021/features-specs/

[15] *2011 Chevrolet HHR review, pricing and specs*. (2019, May 14). Car and Driver.
https://www.caranddriver.com/chevrolet/hhr/specs

[16] *Mazda 6 dimensions 2020*. CarsGuide.
https://www.carsguide.com.au/mazda/6/car-dimensions/2020

[17] *Maserati Levante dimensions*. autoX.
https://www.autox.com/new-cars/maserati/levante/dimensions/

[18] *BMW X7 dimensions*. autoX.
https://www.autox.com/new-cars/bmw/x7/dimensions/

[19] *2022 Cadillac Escalade ESV*. Media Cadillac.
https://media.cadillac.com/media/us/en/cadillac/vehicles/escalade-escalade-esv/2022.tab1.html

[20] *Mazda MX-5 dimensions 2021*. CarsGuide.
https://www.carsguide.com.au/mazda/mx-5/car-dimensions/2021

[21] *Ferrari Portofino dimensions*. autoX.
https://www.autox.com/new-cars/ferrari/portofino/dimensions/

[22] *McLaren 720S dimensions*. autoX.
https://www.autox.com/new-cars/mclaren/720s/dimensions/

[23] *Seperti APA dimensi Toyota Kijang Innova?* Oto.
https://www.oto.com/mobil-baru/toyota/kijang-innova/faq/seperti-apa-dimensi-toyota-kijang-innova

[24] *Mercedes-Benz V-class dimensions 2020*. CarsGuide.
https://www.carsguide.com.au/mercedes-benz/v-class/car-dimensions/2020

[25] *Mercedes-Benz sprinter dimensions 2021*. CarsGuide.
https://www.carsguide.com.au/mercedes-benz/sprinter/car-dimensions/2021

[26] *Used 2015 Nissan Frontier Specs & Features.* Edmunds.
https://www.edmunds.com/nissan/frontier/2015/features-specs/

[27] *2022 Ford F-450 Super Duty Specs & Features.* Edmunds.
https://www.edmunds.com/ford/f-450-super-duty/2022/features-specs/

[28] *What is machine learning and why is it important?* (2021, March 30).
SearchEnterpriseAI.
https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML

[29] Firican, G. (2022, January 7). *The history of machine learning*. LightsOnData.
https://www.lightsondata.com/the-history-of-machine-learning/

[30] *When was machine learning invented?* (2021, February 9). Pandio.
https://pandio.com/when-was-machine-learning-invented/

[31] *What is supervised learning?* (2021, March 26). SearchEnterpriseAI.

https://www.techtarget.com/searchenterpriseai/definition/supervised-learning

[32] Kurama, V. (2019, September 4). *Regression in machine learning: What it is and examples of different models*. Built In. https://builtin.com/data-science/regression-machine-learning

[33] Banoula, M. (2021, February 2). *Classification in machine learning: What it is and classification models*. Simplilearn.com. https://www.simplilearn.com/tutorials/machine-learning-tutorial/classification-in-machine-learning

[34] *What is unsupervised learning?* (2020, September 21). IBM - United States. https://www.ibm.com/cloud/learn/unsupervised-learning

[35] *Supervised vs. unsupervised learning: What's the difference?* (2021, March 12). IBM - United States. https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

[36] Bhatt, S. (2019, April 19). *Reinforcement learning 101*. Medium. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

[37] *Computer vision vs. machine vision. What's the difference?* (2022, September 9). Appen. https://appen.com/blog/computer-vision-vs-machine-vision/

[38] *Data quality explored. STS* https://www3.tuhh.de/sts/hoou/data-quality-explored/2-1-1-image-representation.html

[39] *Deep learning vs machine learning -what's the difference?* (2022, July 8). ProjectPro. https://www.projectpro.io/article/deep-learning-vs-machine-learning-whats-the-difference/414

[40] Parmar, R. (2018, September 11). *Training deep neural networks*. Medium. https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

[41] Pramoditha, R. (2021, December 29). *The concept of artificial neurons (Perceptrons) in neural networks*. Medium. https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc

[42] Sanjay.M. (2019, December 15). *Neural net from scratch (using Numpy)*. Medium. https://towardsdatascience.com/neural-net-from-scratch-using-numpy-71a31f6e3675

[43] *What are neurons in neural networks / how do they work?* Cross Validated. https://stats.stackexchange.com/questions/241888/what-are-neurons-in-neural-networks-how-do-they-work

[44] Babs, T. (2022, June 24). *The mathematics of neural networks*. Medium. https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05

[45] *Effect of bias in neural network*. (2018, September 25). GeeksforGeeks. https://www.geeksforgeeks.org/effect-of-bias-in-neural-network/

[46] *Kotakode.com | Komunitas developer Indonesia*. https://kotakode.com/blogs/3468/10-Konsep-Deep-Learning-yang-Harus-Kamu-Ketahui-untuk-Interview

[47] *Vanishing gradient problem, explained*. KDnuggets. https://www.kdnuggets.com/2022/02/vanishing-gradient-problem.html

[48] Shankar297. (2022, July 12). *Understanding loss function in deep learning*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/06/understanding-loss-function-in-deep-learning/

[49] Kwiatkowski, R. (2022, July 13). *Gradient descent Algorithm — a deep dive*. Medium. https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21

[50] Menon, A. (2018, September 19). *Linear regression using gradient descent*. Medium. https://towardsdatascience.com/linear-regression-using-gradient-descent-97a6c8700931

[51] *Small learning rate vs big learning rate*. Stack Overflow. https://stackoverflow.com/questions/62690725/small-learning-rate-vs-big-learning-rate

[52] Doshi, S. (2020, August 3). *Various optimization algorithms for training neural network*. Medium. https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

[53] Gupta, A. (2022, December 2). *A comprehensive guide on deep learning optimizers*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers

[54] *Intuition of Adam optimizer*. (2020, October 24). GeeksforGeeks. https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

[55] *Understanding deep Convolutional neural networks*. (2023, February 6). Run:ai - AI Optimization and Orchestration. https://www.run.ai/guides/deep-learning-for-computer-vision/deep-convolutional-neural-networks

[56] *Why convolution is required, or what is the philosophy behind convolution?* Signal Processing Stack Exchange.

https://dsp.stackexchange.com/questions/9751/why-convolution-is-required-or-what-is-the-philosophy-behind-convolution

[57] Sinha, U. *Convolutions: Image convolution examples - AI shack*. AI Shack. https://aishack.in/tutorials/image-convolution-examples/

[58] *Introduction to Convolutional neural networks CNNs*. AIgents - Data Science Career Community. https://aigents.co/data-science-blog/publication/introduction-to-convolutional-neural-networks-cnns

[59] Unzueta, D. (2022, March 15). *Convolutional layers vs fully connected layers*. Medium. https://towardsdatascience.com/convolutional-layers-vs-fully-connected-layers-364f05ab460b

[60] Jason Brownlee. (2017, December 20). *A Gentle Introduction to Transfer Learning for Deep Learning.* Machine Learning Mastery. https://machinelearningmastery.com/transfer-learning-for-deep-learning/

[61]
Bajaj, A. (2023, February 17). *Performance metrics in machine learning [Complete guide]*. neptune.ai. https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide

[62] *F-beta score*. Hasty.ai. https://hasty.ai/docs/mp-wiki/metrics/f-beta-score

[63] *What is a good F1 score? Simply explained (2022)*. (2022, December 8). Stephen Allwright. https://stephenallwright.com/good-f1-score/

[64] Kohli, S. (2019, November 18). *Understanding a classification report for your machine learning model*. Medium.

https://medium.com/@kohlishivam5522/understanding-a-classification-report-for-your-machine-learning-model-88815e2ce397

[65] Md. Belal Hossain, S.M. Hasan Sazzad Iqbal, Md. Monirul Islam, Md. Nasim Akhtar, & Iqbal H. Sarker. (2022). *Transfer learning with fine-tuned deep CNN ResNet50 model for classifying COVID-19 from chest X-ray images*. Informatics in Medicine Unlocked.
https://www.sciencedirect.com/science/article/pii/S235291482200065X

[66] *Application of a modified inception-v3 model in the dynasty-based classification of ancient murals*. (2021, July 27). SpringerOpen.
https://asp-eurasipjournals.springeropen.com/articles/10.1186/s13634-021-00740-8

[67] Faisal Dharma Adhinata, Diovianto Putra Rakhmadani, Merlinda Wibowo, & Akhmad Jayadi. (2021, May 1). *A Deep Learning Using DenseNet201 to Detect Masked or Non-masked Face*. Neliti.
https://media.neliti.com/media/publications/441926-a-deep-learning-using-densenet201-to-det-cdfb0c17.pdf

[68] *Maharashtra government to form committee to ease parking, traffic issues in Mumbai*. (2018, September 6). DNA India.
https://www.dnaindia.com/mumbai/report-maharashtra-government-to-form-committee-to-ease-parking-traffic-issues-in-mumbai-2659746

[69] *Real-time vehicle classification using CNN*. (2020, October 15). IEEE Xplore.
https://ieeexplore.ieee.org/document/9225623

[70] *Convolutional neural network based vehicle classification in adverse Illuminous conditions for intelligent transportation systems*. (2021, February 13). Publishing Open Access research journals & papers | Hindawi.
https://www.hindawi.com/journals/complexity/2021/6644861/

[71] Donny Avianto, Agus Harjoko, & Afia Hayati. (2022, October 22). *CNN-Based Classification for Highly Similar Vehicle Model Using Multi-Task Learning*. MDPI - Publisher of Open Access Journals.
https://www.mdpi.com/2313-433X/8/11/293/pdf

[72] *Keras image Preprocessing: Scaling image pixels for training*. (2017, February 16). LinkedIn.
https://www.linkedin.com/pulse/keras-image-preprocessing-scaling-pixels-training-adwin-jahn

[73] *You might be resizing your images incorrectly*. (2020, December 27). Roboflow Blog.
https://blog.roboflow.com/you-might-be-resizing-your-images-incorrectly/

[74]
Karim, R. (2022, December 31). *Illustrated: 10 CNN architectures*. Medium.
https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

[75]
Deep Residual Networks (ResNet, ResNet50) - 2023 Guide (2023). Viso.ai.
https://viso.ai/deep-learning/resnet-residual-neural-network/

[76]
Resnet-50: The basics and a quick tutorial. (2023, February 9). Datagen.
https://datagen.tech/guides/computer-vision/resnet-50/

[77]
Inception V3 model architecture. (2021, October 8). OpenGenus IQ: Computing Expertise & Legacy.
https://iq.opengenus.org/inception-v3-model-architecture/

[78]
Tsang, S. (2019, March 20). *Review: DenseNet — Dense Convolutional network (Image classification)*. Medium.

https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803

[79]
Yesilscience. (2022, March 8). *Transfer learning using DenseNet201*. Yesil Science.
https://www.yesilscience.com/transfer-learning-densenet201/

[80] *ImageNet Home.* (2021, March 11). ImageNet.
https://www.image-net.org/

[81] *An Update to the ImageNet Website and Dataset.* (2021, March 11). ImageNet.
https://www.image-net.org/update-mar-11-2021.php

[82] *Prepare the ImageNet dataset — gluoncv 0.11.0 documentation*. GluonCV Toolkit.
https://cv.gluon.ai/build/examples_datasets/imagenet.html

[83]
CNN architectures timeline (1998-2019). (2019, October 24). AISmartz.
https://www.aismartz.com/blog/cnn-architectures/

[84]
XCeption model and Depthwise separable convolutions. (2019, March 20). Mael Fabien
https://maelfabien.github.io/deeplearning/xception/#pointwise-convolution

[85]
Tsang, S. (2019, March 20). *Review: Xception — With Depthwise separable convolution, better than inception-v3*. Medium.
https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568

[86] Brownlee, J. (2019, December 3). *A gentle introduction to batch normalization for deep neural networks*. Machine Learning Mastery.

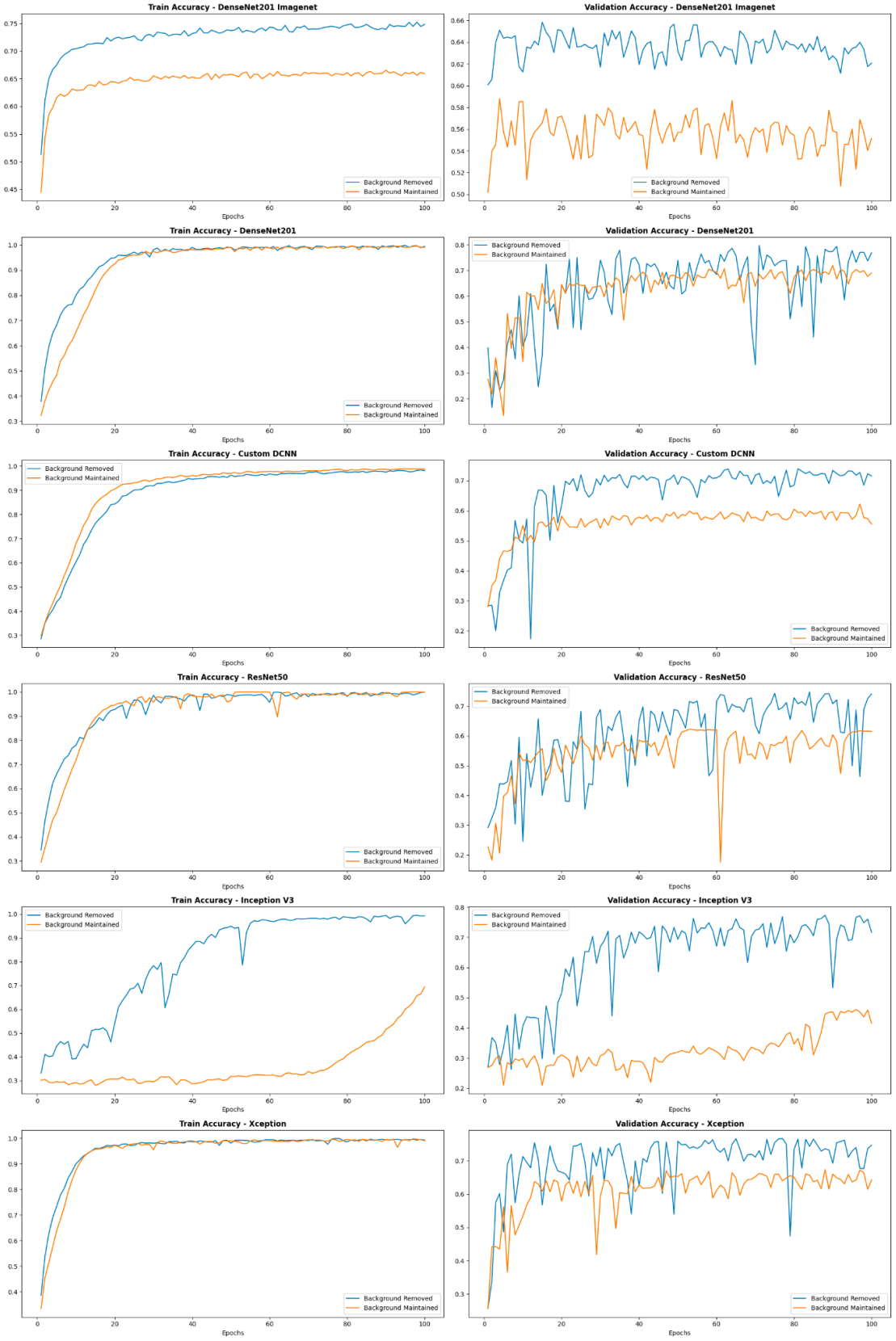
https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

[87] Biswal, A. (2020, April 23). *Convolutional neural network tutorial [Update]*. Simplilearn.com. https://www.simplilearn.com/tutorials/deep-learning-tutorial/convolutional-neural-network

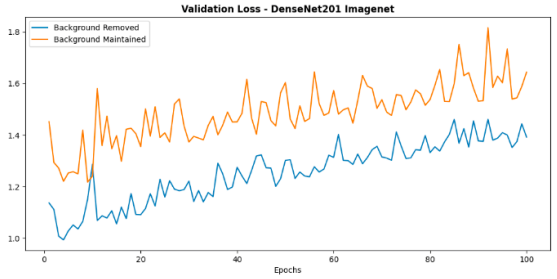
[88] *How ReLU and Dropout Layers Work in CNNs.* (2023, March 16). Baeldung. https://www.baeldung.com/cs/ml-relu-dropout-layers

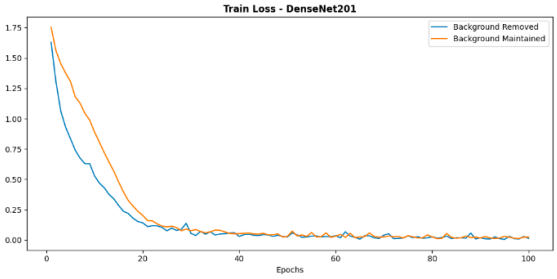
APPENDICES

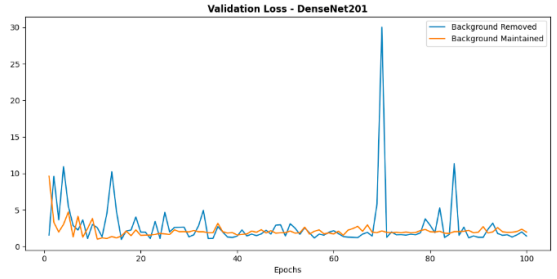
Appendix A: Accuracy Trend per Epoch

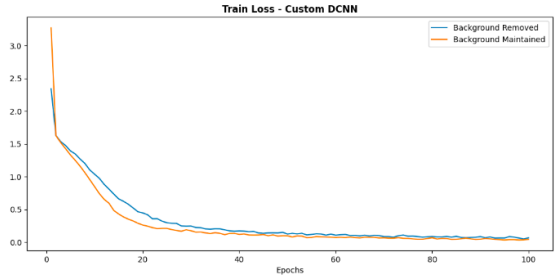


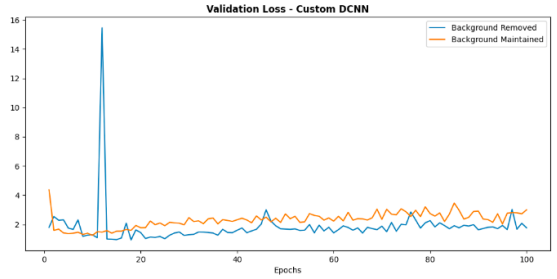
Appendix B: Loss Trend per Epoch

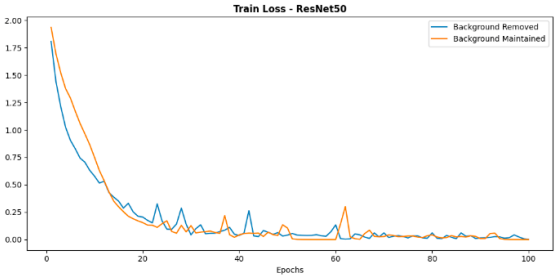


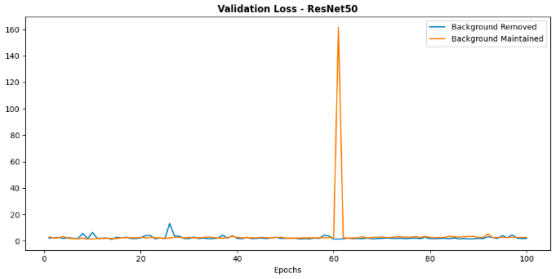


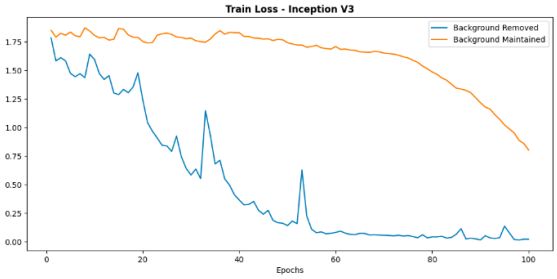


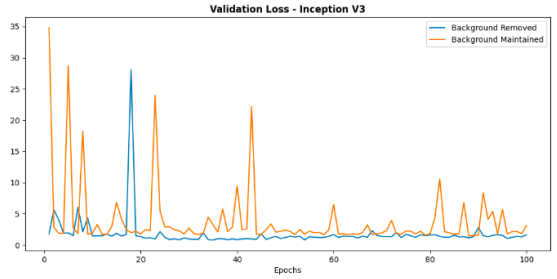


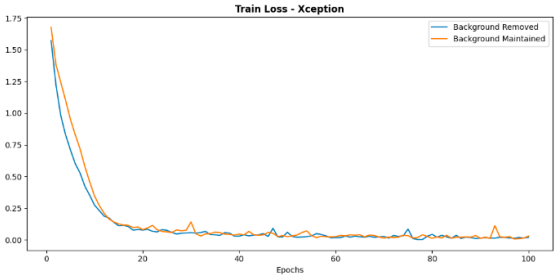


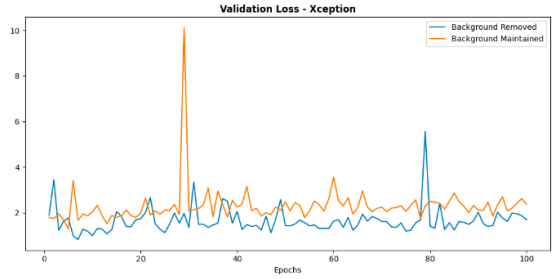












Appendix C: Source Code for Removing Image Backgrounds

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import os, io, math
import pandas as pd
from rembg.bg import remove
import numpy as np
from multiprocessing import Pool
from PIL import Image, ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
PATH_DATA_TRAIN = 'data/data-raw/train'
PATH_DATA_TEST = 'data/data-raw/test'
PATH_DATA_PREPROCESSED_TRAIN = 'data/data-preprocessed/train'
PATH_DATA_PREPROCESSED_TEST = 'data/data-preprocessed/test'
```

```
def generate_paths(path_data, path_data_remove_bg):

    df = pd.DataFrame()

    files = [i for i in os.listdir(path_data) if '.DS_Store' not in i]

    for file in files:
        df_file = pd.DataFrame()
        images = [i for i in os.listdir(f'{path_data}/{file}') if
'.DS_Store' not in i]
        df_file['path_original'] = [f'{path_data}/{file}/{i}' for i in
images]
        df_file['path_remove_bg'] = [f'{path_data_remove_bg}/{file}/{i}' for
i in images]
        df = pd.concat([df, df_file])

    return df
```

```
df_path_train = generate_paths(PATH_DATA_TRAIN,
PATH_DATA_PREPROCESSED_TRAIN)
df_path_train
```

```
df_path_test = generate_paths(PATH_DATA_TEST, PATH_DATA_PREPROCESSED_TEST)
df_path_test
```

```
def remove_bg(df):

    list_path_original = list(df['path_original'])
    list_path_remove_bg = list(df['path_remove_bg'])

    for i in range(len(df)):

        # display progress log
        if (i+1)%50==0:
            print('Completed:', i+1, '/', len(df))

        # load image
        image = Image.open(list_path_original[i])

        # remove background
        with io.BytesIO() as buf:
            image.save(buf, 'jpeg')
            image = buf.getvalue()
            image = remove(image)

        # convert into icon
        image = Image.open(io.BytesIO(image))
        image = image.convert("RGBA")

        # change background to white
        background = Image.new(image.mode[:-1], image.size, (255, 255, 255))
        background.paste(image, image.split()[-1])
        image = background

        # convert back to jpeg
        image = image.convert('RGB')
        image.save(list_path_remove_bg[i], 'JPEG')
```

```
remove_bg(df_path_train)
remove_bg(df_path_test)
```

Appendix D: Source Code for ResNet-50 Architecture

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications import ResNet50

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'resnet50.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
model = ResNet50(
   input_shape=input_shape,
   classes=output_shape,
   weights=None,
   classifier_activation='softmax'
)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
```

```
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

Appendix E: Source Code for Inception V3 Architecture

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications import InceptionV3

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'inception-v3.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
model = InceptionV3(
   input_shape=input_shape,
   classes=output_shape,
   weights=None,
   classifier_activation='softmax'
)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
```

```
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

Appendix F: Source Code for DenseNet 201 Architecture

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications import DenseNet201

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'densenet201.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
model = DenseNet201(
   input_shape=input_shape,
   classes=output_shape,
   weights=None,
#      classifier_activation='softmax'
)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
```

```
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

Appendix G: Source Code for DenseNet 201 (Pre-Trained ImageNet)

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications import DenseNet201

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'densenet201-imagenet.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
model = DenseNet201(
   input_shape=input_shape,
   weights="imagenet",
   include_top=False,
   pooling='max'
)

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

i = Input(shape=input_shape)
x = model(i)
x = Flatten()(x)
x = Dense(output_shape, activation='softmax')(x)

model = Model(inputs=i, outputs=x)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
```

```
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
```

```
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

Appendix H: Source Code for Xception Architecture

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications import Xception

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'xception.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
model = Xception(
   input_shape=input_shape,
   classes=output_shape,
   weights=None,
   classifier_activation='softmax'
)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
```

```
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

Appendix I: Source Code for Custom-Built DCNN Architecture

Note: All the Python codes utilized for the experiments this research are compiled and run with Jupyter Notebook. Therefore, any gap between the codes represents a separation between cells.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout,
GlobalMaxPooling2D, MaxPooling2D, BatchNormalization
from tensorflow.keras.models import Model

import time
start_time = time.time()
```

```
PATH_DATA_TRAIN =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/train'
PATH_DATA_TEST =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/test'
PATH_DATA_VALIDATION =
'../input/cars-data-ttv-80-10-10/data-preprocessed/data-preprocessed/valida
tion'
PATH_SAVED_MODEL = 'mario.h5'
```

```
generator = ImageDataGenerator(rescale = 1./255.)

target_size = (100, 100)
batch_size = 32

train_batches = generator.flow_from_directory(
   PATH_DATA_TRAIN, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)

validation_batches = generator.flow_from_directory(
   PATH_DATA_VALIDATION, class_mode='categorical',
   batch_size=batch_size, target_size=target_size
)
```

```
test_batches = generator.flow_from_directory(
   PATH_DATA_TEST, class_mode='categorical', shuffle=False,
   batch_size=batch_size, target_size=target_size
)
```

```
# preview data (batch size, width, height, colors)
print('x_train shape:', train_batches[0][0].shape)
print('x_validation shape:', validation_batches[0][0].shape)
print('x_test shape:', test_batches[0][0].shape)
print('y_train shape:', train_batches[0][1].shape)
print('y_validation shape:', validation_batches[0][1].shape)
print('y_test shape:', test_batches[0][1].shape)
```

```
# input & output shape
input_shape = train_batches[0][0][0].shape
output_shape = len(train_batches[0][1][0])

# display input & output shape
print('Input Shape:', input_shape)
print('Output Shape:', output_shape)
```

```
i = Input(shape=input_shape)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dropout(0.2)(x)
```

```
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(output_shape, activation='softmax')(x)

model = Model(i, x)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# train model
r = model.fit(train_batches, validation_data=validation_batches,
epochs=100)

# save model
model.save(PATH_SAVED_MODEL)

# load model
# model = load_model(PATH_SAVED_MODEL)
```

```
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.legend()
```

```
y_pred = model.predict(test_batches).argmax(axis=1)
```

```
model.evaluate(test_batches)
```

```
# generate classification report
class_labels = list(test_batches.class_indices.keys())
report = classification_report(test_batches.classes, y_pred,
target_names=class_labels, output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
df_classification_report
```

```
# compute confusion matrix
matrix = confusion_matrix(test_batches.classes, y_pred)

# normalize confusion matrix
```

```
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
matrix = np.around(matrix, decimals=2)

# generate confusion matrix heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(matrix, annot=True, xticklabels=class_labels,
yticklabels=class_labels)
plt.ylabel('Test Values')
plt.xlabel('Predicted Values')
```

```
execution_time_s = round(time.time() - start_time, 2)
execution_time_m = round(execution_time_s/60, 2)
execution_time_h = round(execution_time_m/60, 2)

print('Execution Time (Hours):', execution_time_h)
print('Execution Time (Minutes):', execution_time_m)
print('Execution Time (Seconds):', execution_time_s)
```

CURRICULUM VITAE

As of July 2023

Personal Information

Name:	Christensen Mario Frans
Place of Birth:	Jakarta, Indonesia
Date of Birth:	5 October 2000
Gender:	Male
Address:	Taman Ratu Indah Blk. BB-2/17, RT/RW 006/011, Kedoya Utara, Kebon Jeruk, Jakarta Barat, DKI Jakarta, Indonesia
Telephone:	+6281280007101

Education & Training

BINUS International University

- Bachelor's Degree in Computer Science
- Date: November 2019 - Present

BINUS School Simprug

- International Baccalaureate Diploma Programme
- Date: July 2017 - May 2019

Work Experience

Data Scientist

- Company: Pintu
- Industry: Financial Technology & Cryptocurrency
- Employment: Full Time
- Date: August 2022 - Present

Data Scientist

- Company: Fiverr & Upwork
- Industry: Online Marketplace for Freelance Services
- Employment: Freelance (Seller)
- Date: February 2022 - Present

Data Engineer

- Company: Telkomsel
- Industry: Infrastructure & Telecommunications
- Employment: Internship
- Date: February 2022 - July 2022

Unity Game Developer & Founder

- Company: CMF Games
- Industry: Gaming & Entertainment
- Employment: Self-Employed
- Date: January 2018 - December 2018