VEHICLE CRASH PREDICTION USING TRANSFORMER NETWORKS

Don Kalindu Sekarage

218541J

Master of Science in Artificial Intelligence

Department of Computation Mathematics
Faculty of Information Technology

University of Moratuwa
Sri Lanka

June 2024

VEHICLE CRASH PREDICTION USING TRANSFORMER NETWORKS

Don Kalindu Sekarage

218541J

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree

Master of Science in Artificial Intelligence

Department of Computation Mathematics

Faculty of Information Technology

University of Moratuwa

Sri Lanka

June 2024

DECLARATION

I declare that this is my own work and this thesis/dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 30-06-2024

The above candidate has carried out research for the Masters/MPhil/PhD thesis/dissertation under my supervision.

Name of Supervisor: Dr.A.L.A.R.R Thanuja

30/06/2024

Signature of the Supervisor: Date:

DEDICATION

This thesis is dedicated to the notable individuals who have offered unwavering support and encouragement throughout my academic pursuits. I acknowledge with profound gratitude the essential role they have played in my journey.

I extend my deepest appreciation to my beloved family, whose relentless support and faith in my potential have pushed me forward. Your sacrifices and consistent encouragement have laid the foundation for my success. I am eternally thankful for your unwavering presence in my life.

ACKNOWLEDGEMENT

I deeply appreciate the support and guidance from those who helped me complete this thesis. Their constant encouragement and valuable contributions have been essential, and I am sincerely thankful for their involvement.

At the forefront, I owe a debt of gratitude to my supervisor, Dr. Romesh Thanuja, whose invaluable guidance, encouragement, and support have been critical throughout my research journey. His profound insights and expertise significantly influenced the direction and enhanced the quality of this research.

Additionally, I thank the academic staff at the Faculty of Information Technology, University of Moratuwa. Their rigorous standards and passionate teaching have inspired me to improve my critical thinking and advance my skills as a Master's student. Their enthusiasm for their fields has continually motivated my academic endeavors.

ABSTRACT

Road accidents pose a significant threat to human life, causing numerous injuries, fatalities, and economic damage worldwide. Recently, there has been growing interest in leveraging Artificial Intelligence (AI) to create systems that can predict vehicle crashes. This thesis focuses on vehicle crash prediction and aims to develop a solution using pre-trained Convolutional Neural Networks (CNN) and transformer networks to mitigate the occurrence of such accidents. By leveraging advanced deep learning techniques, this research addresses the limitations of traditional crash analysis methods. The objectives of this study involve critically reviewing the evolution of vehicle crash prediction systems, studying modern technologies employed in vehicle crash prediction, designing and implementing a crash prediction system using pre-trained CNN and transformer networks, and evaluating the solution's effectiveness. The motivation behind this research stems from the urgency to reduce the growing frequency of vehicle crashes and their associated consequences. Current approaches to road safety primarily rely on reactive measures, prompting the need for proactive strategies to prevent accidents. The Car Learning to Act (CARLA) simulator was used for data gathering, in that an ego-vehicle attached to RGB and RGB-Depth cameras were used. Ego-vehicle traveled through a simulated environment with different weather conditions and maps and captured a sequence of RGB and RGB-Depth (RGB-D) images in safe and unsafe scenarios. To extract features from RGB and RGB-D images pre-trained CNN were utilized. Four pre-trained CNNs were used for feature extraction. With those extracted features, a transformer network was employed to train a model. After model training and testing, it was observed that the transformer model trained with VGG16-based feature extraction performs better than other methods. This research can be implemented in the real world with real-time predictions and addresses the shortcomings of visual-based systems. Ultimately, it aims to improve road safety by enabling proactive crash prevention and minimizing the human and economic costs associated with accidents.

Keywords: CNN, Transformer Network, CARLA, Feature Extraction

TABLE OF CONTENTS

Declaration i

Dedication ii

Abstract iv

List of Figures viii

List of Tables x

List of Abbreviations xi

List of Appendices xii

Chapter 1 1

Introduction 1

1.1 Prolegomena 1
1.2 Objectives 2
1.3 Background and Motivation 2
1.4 Problem in Brief 3
1.5 CrashGuard Solution for Vehicle Crash Prediction 4
1.6 Resource Requirements 4
1.7 Structure of the Thesis 4
1.8 Summary 5

Chapter 2 6

Evolution of Vehicle Crash Prediction Systems 6

2.1 Introduction 6
2.2 Gestation of vehicle crash prediction systems 6
2.3 Major developments in vehicle crash prediction systems 7
2.4 Attention-based autonomous driving 9
2.5 Use of RGB and RGB Depth Camera 9
2.6 Future Directions of The Vehicle Crash Prediction Systems 10
2.7 Summary of Challenges in the Vehicle Crash Prediction Systems 10
2.8 Problem Definition 11
2.9 Summary 12

Chapter 3 13

Technology adapted 13

3.1 Introduction 13
3.2 CARLA Simulator 13
3.2.1 Unreal Engine 14
3.2.2 Collision detector 14
3.2.3 RGB Camera 14
3.2.4 RGB Depth Camera 15

3.3 Programming Language 15
3.3.1 CARLA Python API 15
3.3.2 TensorFlow 15
3.4 Cloud Computing 16
3.5 CNN 16
3.5.1 Convolutional Layer 17
3.5.2 Pooling Layer 18
3.5.3 ReLU Activation Function 18
3.5.4 Fully Connected Layer 19
3.5.5 SoftMax Activation Function 19
3.6 Pre-trained CNN Models for Feature Extraction 19
3.6.1 VGG16 20
3.6.2 MobileNet 21
3.6.3 ResNet50 22
3.6.4 DenseNet121 23
3.7 Transformer Network 24
3.7.1 Encoder Block Input 25
3.7.2 Creating Positional Encoding 25
3.7.3 Multi-Head Attention Module 26
3.7.4 Add and Normalization 28
3.7.5 Decoder Block 30
3.7.6 Encoder-Only Models 30
3.8 Summary 31

Chapter 4 32

An Approach to Vehicle Crash Prediction 32
4.1 Introduction 32
4.2 Hypothesis 32
4.3 Input 32
4.4 Output 33
4.5 Process 33
4.6 Features 34
4.7 Users 34
4.8 Summary 35

Chapter 5 36

Design of CRASHGUARD 36
5.1 Introduction 36
5.2 Top-level Architecture 36
5.3 Data Collection From CARLA 36
5.4 Data Pre-Processing 38
5.5 Feature Extraction with Pre-Trained Network 38
5.6 Training Model with Transformer 39
5.7 Evaluation 41

5.8 Summary 41
Chapter 6 42
Implementation of CRASHGUARD 42
6.1 Introduction 42
6.2 Installing CARLA Simulator 42
6.2 Data Collection 42
6.3 Data Pre-processing 43
6.4 Feature Extraction and Concatenation 44
6.5 Data Training Using Transformer Network 44
6.6 Creating Demo Video 45
6.7 Summary 45
Chapter 7 46
Evaluation of CRASHGUARD 46
7.1 Introduction 46
7.2 Transformer Setup 46
7.3 Classification Metrics 47
7.3.1 Precision 47
7.3.2 Recall 48
7.3.3 F1-Score 48
7.4 Evaluation of Feature Extraction Methods 48
7.4.1 Transformer Network with VGG16 49
7.4.2 Transformer Network with MobileNet 50
7.4.3 Transformer Network with ResNet50 51
7.4.4 Transformer Network with DenseNet121 53
7.5 Result Comparison with Other Research 55
7.6 Summary 56
Chapter 8 57
Conclusion and Further Work 57
8.1 Conclusion 57
8.2 Limitations 58
8.3 Further work 58
References 59

LIST OF FIGURES

Figure	Description	Page
Fig. 3.1	CNN Architecture [33]	16
Fig. 3.2	Example of Convolution Operation [34]	17
Fig. 3.3	Example of Max Pooling and Average Pooling [34]	18
Fig. 3.4	Activation Function ReLU [35]	18
Fig. 3.5	Activation Function SoftMax [35]	19
Fig. 3.6	Architecture of VGG16 [37]	20
Fig. 3.7	Architecture of MobileNet [39]	22
Fig. 3.8	Architecture of ResNet50 [40]	23
Fig. 3.9	Architecture of DenseNet121 [41]	24
Fig. 3.10	Transformer Model Architecture [43]	24
Fig. 3.11	Input Embedding for Each Image	25
Fig. 3.12	Combining Input with Positional Encoding	26
Fig. 3.13	Creating, Q,K and V	26
Fig. 3.14	Scaled dot-product (on the left). Multi-Head Attention with Simultaneous Levels(on the right) [20]	27
Fig. 3.15	The Dot Product of Q and KT	27
Fig. 3.16	Final Stage of the Encoder Block	30
Fig. 4.1	Input Image Sequence	33
Fig. 5.1	Top-level Architecture of the Design	36
Fig. 5.2	Different Maps	37
Fig. 5.3	Different Weather Conditions	37
Fig. 5.4	Feature Extraction and Concatenation	39
Fig. 5.5	Encoder Block of the Transformer [43]	40
Fig. 7.1	Accuracy and Loss Graphs for Training vs Validation for VGG16 Based Feature Extraction	49
Fig. 7.2	Model Confusion Matrix for VGG16 Based Feature Extraction	50
Fig. 7.3	Accuracy and Loss Graphs for Training vs Validation for MobileNet Based Feature Extraction	51
Fig. 7.4	Model Confusion Matrix for MobileNet Based Feature Extraction	51
Fig. 7.5	Accuracy and Loss Graphs for Training vs Validation for ResNet50 Based Feature Extraction	52

Fig. 7.6	Model Confusion Matrix for ResNet50 Based Feature Extraction	53
Fig. 7.7	Accuracy and Loss Graphs for Training vs Validation for DenseNet121 Based Feature Extraction	53
Fig. 7.8	Model Confusion Matrix for DenseNet121 Based Feature Extraction	54

LIST OF TABLES

Figure	Description	Page
Table 5.1	CNN Models and No of Extracted Features	39
Table 7.1	Summary of Transformer Parameters	47
Table 7.2	Summary of Performance Metrics for Each Feature Extraction Method	55

LIST OF ABBREVIATIONS

Abbreviation	Description
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
RGB-D	RGB Depth
AI	Artificial Intelligence
SMOTE	Synthetic Minority Over-sampling Technique
RF	Random Forest
KNN	K-Nearest Neighbors
RNN	Recurrent Neural Network
SVM	Support Vector Machines
ReLU	Rectified Linear Unit
ML	Machine Learning
CARLA	Car Learning to Act]
GDP	Gross Domestic Production
CPN	Crash Prediction Network
NLP	Natural Language Processing

LIST OF APPENDICES

Figure	Description	Page
Appendix – A	Script for Importing Necessary Libraries, Setting Basic Configuration, and Connect with CARLA Simulator	61
Appendix – B	CarlaSession Class with add_vehicles Method	61
Appendix – C	Method to Change Weather Settings and Add Pedestrians	62
Appendix – D	Method to Spawn Ego-vehicle and Attach Sensors to It	62
Appendix – E	Method to Trigger Crash	63
Appendix – F	Methods to Run Simulation and Collet RGB and RGB-D Images	63
Appendix – G	RGB and RGB-D Image Preprocessing	64
Appendix – H	Importing Necessary Libraries and Defining CNN Model	64
Appendix – I	Script for Creating Extracted Feature Array	65
Appendix – J	Script for Transformer Positional Embedding	65
Appendix – K	Script for Transformer Encoder	66
Appendix – L	Utility Functions for Training	67
Appendix – M	Script for Creating a Demo Video	68

CHAPTER 1
INTRODUCTION

1.1 Prolegomena

In recent years, road accidents have emerged as a major contributor to injuries, fatalities, and property damage worldwide. According to the World Health Organization (WHO) around 1.35 million individuals annually lose their lives to automobile accidents. Furthermore, motor vehicle accidents are the foremost underlying cause of mortality among children and young adults aged 5-29 years [1]. In Sri Lanka, available statistics show that there are, on average 38,000 road traffic crashes each year. These incidents result in approximately 3,000 fatalities and 8,000 serious injuries [2]. In the modern world, people are more attracted to buying more and more vehicles for personal use, leading to higher traffic-related accidents. According to the U.S. Department of Transportation, 228,687 driving licenses were issued in 2020, an increase from the 190,625 licenses issued in 2010 [3]. In Sri Lanka, the economic cost of road accidents is estimated to exceed Rs 10,000 million annually, constituting approximately 1% of the country's Gross Domestic Product (GDP) [4]. Apart from causing damage to living persons, road accidents are also responsible for damage to property and infrastructure. The current trend among motor vehicle manufacturers is making vehicles fully autonomous. Autonomous vehicles have the potential to dramatically reduce traffic accidents by removing the human factor in traffic accidents. However, Autonomous Vehicles are still not able to prevent traffic accidents entirely. The California Department of Motor Vehicles reported 129 traffic accidents involving autonomous vehicles [5]. In this thesis, we consider this issue and find a solution that can be implemented to mitigate accidents. A CNN is a specific form of deep learning method that specializes in tasks related to image recognition and classification. A key advantage of a CNN is its ability to simultaneously learn feature extraction layers and the classification layer, resulting in a model output that is highly structured and strongly reliant on the extracted features [6]. Transformer networks have transformed Natural Language Processing (NLP) tasks with the introduction of the self-attention mechanism, significantly enhancing their capabilities [7]. This mechanism allows the network to obtain global interdependencies and effectively process sequences of data, making transformers suitable for analyzing sequential data

like image sequences. The integration of CNN and transformer networks in vehicle crash prediction systems holds great potential for improving prediction accuracy and timeliness. By leveraging CNN's ability to extract features from images without requiring manual feature engineering or domain knowledge and the sequence modeling capabilities of transformer networks, it becomes possible to detect specific patterns and features in image sequences that may indicate a potential vehicle crash. This allows for proactive measures to be taken, such as issuing warnings to drivers or triggering automatic safety mechanisms. In this thesis, we aim to explore and develop a vehicle crash prediction system utilizing pre-trained CNN and transformer networks. The primary objective is to design a system that can effectively analyze a sequence of multimodal images captured within a vehicle and provide a binary classification of whether the oncoming situation is safe or unsafe. By accurately predicting potential crashes in real-time, this system has the capacity to improve road safety and prevent accidents significantly. The rest of this chapter is organized to present the following: Objectives, background and motivation, problem in brief, CrashGuard solution, resource requirements, and structure of the thesis.

1.2 Objectives

The objectives for creating a Vehicle Crash Prediction can be summarized as follows:

- Critical review of the evolution of vehicle crash prediction systems
- In-depth study of technologies used in modern vehicle crash prediction systems
- Design and implement vehicle crash prediction systems with pre-trained CNN and transformer network
- Evaluate the solution using proper evaluation metrics

1.3 Background and Motivation

The motivation behind this research stems from the urgency to reduce the alarming rate of vehicle crashes and their associated consequences. Traditional crash analysis methods often lack the ability to capture complex relationships and dynamic patterns present in crash data. This thesis seeks to utilize AI and Machine Learning (ML) to overcome these limitations and provide a deeper, more accurate insight into the factors that contribute to crashes. The predictive capabilities of the proposed system can

empower policymakers, traffic authorities, and drivers to take proactive measures to avoid vehicle crashes. The current approach to road safety primarily relies on reactive measures, such as emergency response systems and post-crash investigations. While these measures are essential, they are insufficient in preventing crashes proactively. By utilizing AI techniques, this research aims to shift the paradigm from reactive to proactive strategies, enabling crash prediction in advance.

Current AI-based vehicle crash prediction systems utilize various AI technologies, including Artificial Neural Networks (ANN), Support Vector Machine (SVM), Decision Trees, Fuzzy Logic and Genetic Algorithms [8]. One research suggests that the ensemble of neural networks called Crash Prediction Network (CPN) can effectively monitor vehicular safety [9]. In that network makes use of vision (only RGB cameras) as input to predict the crash, another vision-based vehicle crash prediction is proposed, utilizing RetinaNet for detection, Kernelized Correlation filter for tracking, and Gaussian distribution approximation for trajectory prediction [10]. When considering Deep Learning techniques, like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) are effective for accident detection using spatiotemporal sequential data [11]. Furthermore, Real-time traffic, weather, and congestion status data can be used with LSTM models along with the Synthetic Minority Oversampling Technique (SMOTE) for balancing imbalanced datasets. Other than predicting vehicle crashes, A hybrid K-means, and Random Forest (RF) method is proposed for predicting road accident severity, showing improved performance over traditional methods like SVM or K-Nearest Neighbours (KNN) [12].

1.4 Problem in Brief

Annually, a number of people face vehicle-related accidents and succumb due to injuries or suffer permanent disability and incur huge costs to GDP. This kind of accident can occur in both conventional vehicles, where human factors are involved, and in autonomous vehicles. There is more research done on ML classification using historical data but less on real-time prediction. There are visual-based real-time crash prediction systems, but visual data used to train those models contain more accidents that happened outside the driving vehicles recorded with dashboard cameras without vehicle involvement or using traffic cameras. These systems do not give a warning to the driver behind the wheel. The proposed method is to address these shortcomings.

1.5 CrashGuard Solution for Vehicle Crash Prediction

This proposal introduces CrashGuard, an innovative vehicle crash prediction system that leverages the capabilities of a pre-trained CNN and transformer network. With the continuous advancements in deep learning, novel neural network architectures have emerged, offering powerful solutions to address complex challenges. The primary objective of this system is to leverage the unique strengths of both the pre-trained CNN and transformer networks, enabling accurate predictions in the context of vehicle crashes. To evaluate and validate the effectiveness of CrashGuard, extensive data gathering is conducted using the CARLA simulator. The input data, consisting of a sequence of multimodal images (RGB and RGB-Depth), is subjected to preprocessing before inputting into a pre-trained CNN. This network effectively extracts the spatial information from the input sequence of multimodal images and generates an encoding representing the relevant features. Subsequently, the output encoding from the pre-trained CNN is fed into the transformer network, which incorporates the temporal features of the data. By considering spatial and temporal information, the transformer network produces a binary classification output, indicating whether the provided input sequence of multimodal images corresponds to a safe or unsafe driving scenario by combining the strengths of the CNN and transformer network.

1.6 Resource Requirements

The following equipment, resources and software will be utilized in the project:

- CARLA simulator
- Unreal Engine
- Computer with at least 6 GB NVIDIA GPU and 16 GB RAM.
- Python 3.7+
- CARLA, TensorFlow python libraries

1.7 Structure of the Thesis

Our thesis on vehicle crash prediction using a transformer network is structured into eight chapters. Subsequent chapters will explore the assessment of vehicle crash prediction systems, covering aspects such as technology, methodology, design, execution, evaluation, conclusions, and future work.

1.8 Summary

This thesis presented a novel system to identify potential vehicle crashes and give early warning using a pre-trained CNN and transformer network. Here, pre-trained CNN is used for feature extraction of RGB and RGB-D images, and those collected features were used to train the transformer model. Data gathering for the proposed solution is done using the CARLA simulator. This solution will overcome some of the drawbacks of prevailing vehicle crash prediction methods.

CHAPTER 2
EVOLUTION OF VEHICLE CRASH PREDICTION SYSTEMS

2.1 Introduction

In Chapter 1, we have presented the overall picture of this thesis, covering the research problem and the essentials of the solution. This chapter gives a comprehensive literature review in the area of vehicle crash prediction systems. The literature review has been structured to cover the gestation of vehicle crash prediction systems, their major developments, and future directions. Here we also summarized the research challenges in vehicle crash prediction systems and defined our research problem.

2.2 Gestation of vehicle crash prediction systems

Vehicle crash prediction is an essential part of road safety that enables predicting oncoming accidents and gives warnings to evade them. Initially, early warning systems to avoid collisions were made up with basically radar technology. The first documented passenger automobile equipped with a smart Collision Warning System was the Cadillac Cyclone XP-74, developed by General Motors in 1959 [13]. The use of radar-like technology alone is not intelligent decision-making, so it has its own pitfalls, which may lead to catastrophic outcomes. Some of the issues are limited perception of Non-Metallic objects, sensitivity to environmental conditions, radar signals that can be susceptible to interference, and higher cost for implementation. Dominique et al. [14] discuss the Generalized Estimating Equation procedure in developing Accident Prediction Models (APMs) for network safety at four-legged signalized intersections. APMs are utilized to estimate the expected frequency of accidents across various entities, such as intersections and road sections. These estimations play a crucial role in identifying potential areas for safety interventions and evaluating the effectiveness of implemented treatments. Overall, this method is used as a statistical analysis model, not an intelligence system, compared with modern machine learning techniques. During the early stages of development, it has been observed that the majority of vehicle crash prediction systems lacked the integration of intelligent decision-making components. The earliest adaptation of a neural network

for autonomous driving is known as the Autonomous Land Vehicle in a Neural Network (ALVINN) [15]. ALVINN utilizes images from a camera and data from a laser range finder as inputs to determine the optimal direction for a vehicle to follow the road. It employs a three-layer backpropagation network tailored specifically for road-following tasks.

2.3 Major developments in vehicle crash prediction systems

AI technologies play a vital role in predicting vehicle accidents based on vehicle and driving features. Some of the AI techniques used for crash prediction include Neural Networks, SVM, Decision Trees, Fuzzy Logic, and Genetic Algorithms [8]. The paper [9] addresses the issue of safety concerns surrounding autonomous vehicles and proposes a CPN as a potential solution. The goal of the text is to propose a solution for improving safety in autonomous vehicles by using an ensemble of neural networks called CPN to supervise decision-making modules. Experiments were carried out using the CARLA 0.9.6 simulator, with an emphasis on preventing locally avoidable catastrophes. Both models, Simple CPN and Spatio-Temporal (ST-CPN), were further extended for additional experiments involving uncertainty by implementing Monte Carlo Dropout. Researchers were able to find that an ensemble of neural networks called CPN can be used to monitor various aspects of vehicular safety. Further Bayesian deep learning can be used to combat over-fitting issues but is difficult to train. The issue of predicting accidents in dashcam videos was explored in the paper [16]. The objective of the paper is to propose a method that utilizes a Dynamic-Spatial-Attention Recurrent Neural Network (DSA-RNN) to predict road vehicle accidents in dashcam videos. The outcome of the research is cutting-edge object detector, full-frame, and object-based appearance/motion features are incorporated into the model, and the proposed method surpasses other baseline models that do not incorporate attention mechanisms or RNN on challenging datasets with 80% recall at an average time-to-event of 1.8559 seconds before it occurs with 56.14% precision. This research has implications for improving vehicle safety by developing AI agents that can drive safely alongside human drivers while scaling up their learning process through observing corner cases at scale using dashboard cameras (dashcams). The goal of the paper [11], is to discuss the use of deep learning techniques for accident detection in Chicago using spatiotemporal sequential data. Here, researchers have proposed a wide

range of techniques to detect the occurrence and estimate the duration of accidents. Researchers have found that deep learning techniques like LSTM and GRUs are effective for predicting accidents using time series data. Furthermore, real-time traffic, weather, and congestion status data can be used with LSTM models along with SMOTE technique for balancing imbalanced datasets. When comparing LSTM and GRU, the GRU model achieved slightly better results than the LSTM model. It was demonstrated that accessing real-time data combined with advanced modeling can help accurately detect traffic incidents such as car crashes, which will lead to safer roads. The objective is to design and construct a vehicle collision detection system by utilizing a combined deep learning model that utilizes various types of data from dashboard cameras. [17]. The field of study addressed in this text is vehicle accident detection systems, which are becoming increasingly vital in response to the rising number of motor vehicle accidents. Vehicle collision detection has achieved new standards with the use of ensemble deep learning models that incorporate multimodal inputs, surpassing the performance of existing models. The study demonstrated that auditory characteristics and spectrogram images are more crucial for identifying automobile collisions compared to visual data captured by dashboard cameras. Therefore, combining both video and audio data significantly enhances performance compared to using only one type of data. A proposed vision-based vehicle crash prediction pipeline is outlined in [10]; it uses RetinaNet for detection, a Kernelized Correlation filter for tracking, and a Gaussian distribution approximation for trajectory prediction. The paper presents a framework aimed at predicting the trajectory of vehicles by detecting and tracking them on roads using CCTV frames. Vehicle detection is executed with high accuracy using the RetinaNet algorithm. A probabilistic algorithm is used to forecast the vehicles' trajectories, enhancing the detection and prediction of vehicle crashes. Yassin et al. [12] proposed a hybrid K-means and RF approach for predicting road accident severity. K-Means clustering was used to categorize the road accident dataset by similarity, and RF was applied to classify the data into different severity labels. These hybrid approaches have demonstrated improved prediction performance compared to traditional methods such as SVM or KNN, especially when handling large datasets. Zhao et al. [18] proposed a traffic accident prediction algorithm that leverages CNN and utilizes data collected from a Vehicular Ad-hoc Network (VANET). There, CNNs were used for feature extraction, with various convolutional kernels responsible for extracting different

features. It was discovered that using Rectified Linear Unit (ReLU) as the activation function can mitigate the problem of gradient disappearance and offers advantages over the Sigmoid function in terms of computational cost and processing speed.

2.4 Attention-based autonomous driving

RNN, with specialized architectures like LSTM [19] and GRU [20] have demonstrated themselves as advanced methodologies for tackling challenges in sequence modeling and transduction. These architectures exhibit outstanding performance in applications like language modeling and machine translation, demonstrating their efficacy in capturing long-range dependencies and effectively processing sequential data. The "Attention is All You Need" paper introduces the Transformer model architecture, which discards recurrence and relies solely on an attention mechanism to capture global interdependence between input and output. This transformer model is able to address the limitation in RNN. In the paper [21] introduces an attention-based hierarchical deep reinforcement learning approach for modeling lane change behaviors in autonomous driving. They incorporated temporal and spatial attention mechanisms into the deep reinforcement learning architecture, which improved the vehicle's ability to focus on nearby vehicles, resulting in smoother and more efficient lane change behavior.

2.5 Use of RGB and RGB Depth Camera

To address the limitations of relying on single-modal data inputs, employing a multimodal approach has emerged as a solution in ML research [17], [22], [23], [24]. Using only RGB information for object recognition has several drawbacks in real-world applications. This occurs because the process of transforming a 3D environment into 2D images inevitably leads to a loss of information [23]. RGB Depth (RGB-D) images record Red, Green, Blue, and Depth data, offering per-pixel depth information that aligns with the matching image pixels. This alignment improves the accuracy and density of the data.

Geo et al [23], in their paper, describe techniques that can be used to fuse RGB and RGB-D images. There are two main fusion strategies: early-level fusion and late-level fusion. In the early-level fusion approach, features from RGB and RGB -Depth data

are extracted using two separate CNNs and then fused early in the process. This fusion is performed to enhance the depth of comprehension of the acquired information, meeting the requirements for more complex analysis. Late-level fusion is a method that combines the outputs of many classifiers, each trained on separate modalities, in order to improve the accuracy and dependability of object identification. SanchezRiera et al. [25] argued that in the context of RGB-D based hand gestures and object recognition, early fusion showed better performance than late fusion.

2.6 Future Directions of The Vehicle Crash Prediction Systems

All the techniques used to predict vehicle crash predictions have their own pros and cons, which leads to some future developments. Future research will need to concentrate on bridging the gap between 2D image data and 3D data like Light Detection and Ranging (LiDAR) to enhance the accuracy and robustness of systems that rely on both data types [26]. According to paper [9], some of the future developments are longer history duration and expanding the CPN model to support additional sensory data as input and beyond only ‘safe’ and ‘unsafe’ labels to add more granularity. Furthermore, the ensemble networks can be expanded beyond sensory inputs to incorporate geographical region data or specific rules. The accuracy of the prediction can be improved using data from cellphones’ GPS and accelerometer data, as suggested in [11].

2.7 Summary of Challenges in the Vehicle Crash Prediction Systems

Predicting vehicle crashes poses significant challenges due to the intricate nature of traffic systems and the multitude of factors that can influence accidents. In this section, we are discussing certain challenges faced when creating a vehicle crash prediction system. One of the challenges is data quality and preprocessing; the accuracy of accident prediction models is highly dependent on the data's quality. It's essential to make sure the data is clean and devoid of missing or noisy entries, as these can significantly impact the performance of the models. To enhance data quality, preprocessing techniques are employed, such as filling in missing data with the mean for numerical variables and the mode for categorical variables. These techniques are crucial for enhancing the reliability of the data used in crash prediction. Further inaccurate or incomplete crash records can affect both injury severity analysis and

crash frequency models [27]. Another one is model selection and complexity. The process of selecting an appropriate prediction model and determining the optimal level of complexity is a challenging task. In crash prediction, researchers have investigated various ML and deep learning algorithms such as RF, SVM. It's important to recognize that there is no one-size-fits-all approach, and researchers need to carefully evaluate their specific data and problem to choose the most appropriate model. Another identified challenge is feature selection, which is one of the most important steps in building an accurate machine-learning model. Identifying the most key features for crash prediction is essential for building an accurate model. Researchers have used various feature selection techniques, including clustering algorithms like K-means, to extract hidden information from road accident data and create new features [12]. However, identifying the most relevant features for crash prediction continues to be challenging. Another issue is assessing the performance of crash prediction models, which is crucial for gauging their efficacy. Commonly used performance metrics such as accuracy, precision, recall, and F1-score are employed to compare different models. Adaptability to different datasets also remains a challenge in building models. Developing models that can be easily adapted to different datasets and traffic problems is important for the practical application of crash prediction models. Another key aspect of vehicle crash prediction is real-time prediction, which is also challenging. An essential assumption for real-time risk evaluation is that the margin of error in the reported crash time is minimal [28].

2.8 Problem Definition

Through extensive literature review, it has been identified that a significant number of individuals experience vehicle-related accidents annually, resulting in injuries, fatalities, and permanent disabilities. These accidents also impose a substantial economic burden on the GDP. Notably, such accidents can occur in both conventional vehicles, where human factors play a role, and autonomous vehicles. While there have been numerous research efforts focusing on machine learning classification using historical data, there is a notable gap in real-time prediction capabilities. Existing real-time crash prediction systems primarily rely on visual data, but they often include accidents occurring outside the vehicle, captured by dashcams or traffic cameras, without providing warnings to the driver behind the wheel.

To address these limitations, this method aims to develop a novel approach for real-time vehicle crash prediction. The objective is to bridge the gap between historical data-based classification and proactive real-time prediction. By leveraging advanced techniques such as transformer networks, the system will enhance the accuracy and timeliness of crash prediction by analyzing the sequence of images captured within the vehicle. This method is designed to overcome the shortcomings of existing systems by focusing on in-vehicle data and providing real-time warnings to the driver. By leveraging the power of CNN and transformer networks, the system aims to capture complex relationships and dynamic patterns present in the image sequences, enabling the proactive detection of potential crash scenarios. By addressing these research gaps and limitations, his approach has the capacity to greatly improve the level of safety on roads by providing timely and accurate predictions, empowering drivers to take necessary precautions, and potentially preventing accidents.

2.9 Summary

This chapter presented our literature in the area of vehicle crash prediction by covering research papers with very high citation records. Here, we carried out the identification of vehicle crash prediction techniques, technology adopted in existing solutions, future developments, and challenges. We have also defined our research problem as a real-time crash prediction system that remains challenging. This chapter also identified machine learning as a potential technology for addressing this issue in vehicle crash prediction. In chapter 3 presents our in-depth study of ML techniques by highlighting their relevance for vehicle crash prediction.

CHAPTER 3
TECHNOLOGY ADAPTED

3.1 Introduction

In this chapter, the focus is on various technologies and tools that have been utilized for the development of vehicle crash prediction systems. Specifically, we will investigate the simulator adopted for data collection, pre-trained CNN models used for feature extraction, the transformer model used for training and inferencing, the programming language, supporting Python libraries, and cloud computing used for training the model. Each technology was adopted based on its effectiveness, ease of use, and suitability to prove the hypothesis. Each technology is concisely described and their usage in this research.

3.2 CARLA Simulator

CARLA is a freely available simulator designed for research in autonomous driving. It is developed for enhancing autonomous driving technologies, it addresses core tasks like trajectory prediction, motion planning, and 3D tracking [22] [29]. Utilizing Unreal Engine for simulations and adhering to the OpenDRIVE standard for road and urban environment definitions, CARLA offers Python and C++ APIs for simulation control. OpenDRIVE, initiated in 2006, is a de facto standard for describing road networks, focusing on driving simulation applications [30]. It offers a dynamic environment and includes a simple interface for an agent to engage with its surroundings. Structured as a server-client system, CARLA's server component manages the simulation and scene rendering. The simulation environment includes three-dimensional (3D) representations of both still objects like buildings, vegetation, and traffic signs, and dynamic objects such as vehicles and pedestrians, supporting comprehensive development of urban driving systems. CARLA further provides a number of sensors to collect data from the environment with ego-vehicle, which refers to a vehicle that equipped with sensors that perceive the surrounding environment. Some of the built-in sensors in CARLA are collision detectors, depth cameras, Global Navigation Satellite System (GNSS) sensors, Lane invasion detectors, LIDAR sensors, Radar

sensors, RGB cameras, RGB-D, etc. Out of those sensors for this research, we have created an ego vehicle with a collision detector, RGB camera, and RGB-D camera.

3.2.1 Unreal Engine

In order to run CARLA, it first needs to install Unreal Engine. Unreal Engine, developed by Epic Games, is a sophisticated game engine known for its powerful graphics rendering capabilities and extensive toolset, making it suitable for various applications beyond gaming, including cinematic content creation and real-time 3D simulations. It fulfills a key function of running the CARLA Simulator, an open-source autonomous driving research platform, by providing highly realistic urban environments, accurate physics simulation, and extensive sensor simulation capabilities. These features enable the detailed and realistic testing of autonomous driving algorithms, ensuring they can be developed, trained, and validated in a controlled yet complex and scalable virtual environment. Through Unreal Engine, CARLA offers a rich, versatile platform that bridges the gap between theoretical algorithms and their practical, real-world application, making a significant contribution to the field of autonomous driving research.

3.2.2 Collision detector

This sensor records an event each time its associated vehicle collides with something in the world. Collisions can occur with moving objects, such as pedestrians or vehicles, or with static objects in the environment. This sensor is useful for collecting data whenever a crash occurs.

3.2.3 RGB Camera

The "RGB" camera functions like a standard camera, capturing RGB images of the scene. RGB camera is one of the two sensors used for data gathering in research. An RGB camera sensor is used to capture sequences of images when the ego-vehicle drives around in simulated environments. It provides a three-channel RGB image.

3.2.4 RGB Depth Camera

The camera records raw scene data, encoding the distance from each pixel to the camera lens in a format known as a depth buffer or z-buffer, which creates a depth map of the scene elements [31]. Each pixel's depth is encoded using the three channels of the RGB color space, with the red channel encoding the least significant byte and the blue channel the most significant byte. The actual distance in meters can be decoded from these values.

$$normalized = \frac{R + G * 256 + B * 256 * 256}{256 * 256 * 256 - 1}$$

$$in\_meters = 1000 * normalize$$

3.3 Programming Language

Python is the most popular programming language with an easier learning curve. It is suitable for both academic and industrial research, especially AI and ML projects. Python offers abundant libraries for data manipulation, scientific computing, and ML. It has libraries capable of handling CARLA, getting data, manipulating collected data, training, and evaluating processed data.

3.3.1 CARLA Python API

The CARLA Python API is a Python package designed for controlling and communicating with CARLA. This package enables users to manage the CARLA simulator and access simulation data via the Python API. For instance, users can control any actor (such as vehicles, pedestrians, and traffic lights) within the simulation, attach sensors to vehicles, and retrieve sensor data.

3.3.2 TensorFlow

TensorFlow is a freely available software library designed for efficient and powerful numerical calculations. The development of this technology was initially undertaken by researchers and engineers from the Google Brain team, which is part of Google's AI division. TensorFlow is particularly suited for deep learning; it allows the efficient

design, training, and deployment of deep neural networks that can process the image related data. With TensorFlow, we easily develop CNN and transformer networks and then train and test with data.

3.4 Cloud Computing

Since training data contains a large number of images, to expedite training and evaluation, we have used the Google Colab platform, which provides a notebook with Graphics Processing Units (GPU)s and Tensor Processing Units (TPU)s. For Google, Colab Pro Plus subscription offers powerful GPUs for a considerably low price. The use of GPU can greatly expedite the training of deep learning models, including transformers. GPU-enabled hardware is particularly useful for handling large amounts of data. When it comes to RAM storage, it provides higher capacity RAM, which is crucial for handling large datasets. Also, Colab comes with most of the popular data science and machine learning libraries that have been pre-installed, including TensorFlow.

3.5 CNN

Pre-trained CNNs are utilized for extracting features from RGB and RGB-D pictures. Mainly, the transformer model was trained feature extracted with several pre-trained CNN models, which are MobileNet, VGG16, ResNet50, and DenseNet121. All models are based on CNN architecture. The concept of Convolutional neural networks was derived from the hierarchical organization of neurons seen by Hubel and Wiesel in 1962. The theoretical notion of convolutional neural networks was initially introduced by LeCun [32]. In the next subsections, we will discuss CNN's architecture and components. The following image shows CNN's architecture. The CNN model's architecture is depicted in Figure 3.1.

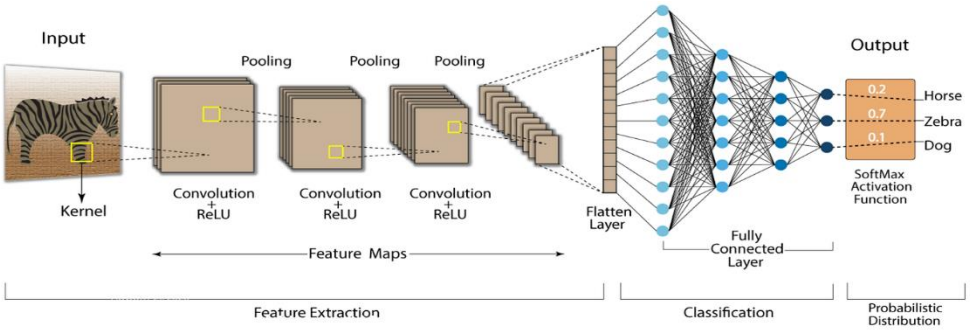

Fig. 3.1: CNN Architecture [33]

3.5.1 Convolutional Layer

The primary goal of this layer is to identify and recognize features such as edges, textures, and complex patterns within the images. This layer applies filters (kernels) to the input image through a mathematical operation called convolution. The filter moves over the picture in both vertical and horizontal directions, computing the dot product between the filter and small sections of the input image [34]. This procedure produces a feature map that emphasizes the existence or nonexistence of specific characteristics at various positions in the image. Filters are learnable matrices that capture features (e.g., vertical edges, horizontal edges). The number of pixels by which the filter travels can be adjusted across the image with a stride parameter. A smaller stride means the filter moves in smaller steps, producing a large feature map. Using padding, we can add zeros around the input image so the convolution operation can be applied at the borders, allowing the output features map to maintain a desired size.

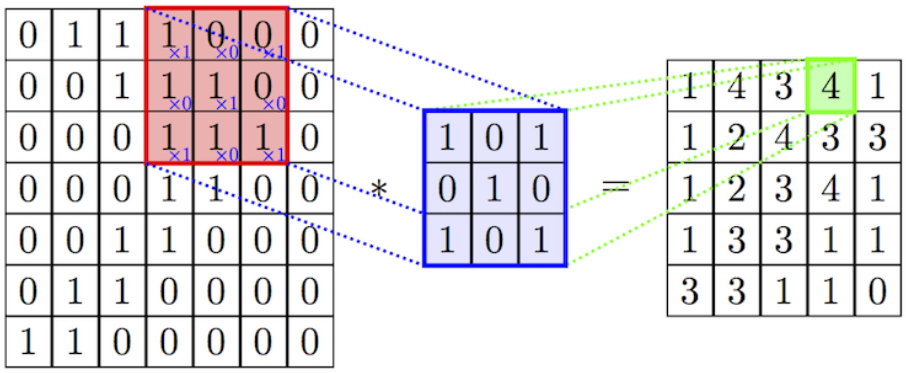

Fig. 3.2: Example of Convolution Operation [34]

3.5.2 Pooling Layer

To decrease the dimensions of the input volume for the next convolutional layer, CNN uses a pooling layer. It helps to reduce the number of parameters and computation in the network. This layer employs a process of aggregating the outputs of neuron clusters from one layer into a single neuron in the subsequent layer, resulting in a reduction in the size of the feature maps. Max Pooling and Average Pooling are the most often used approaches.

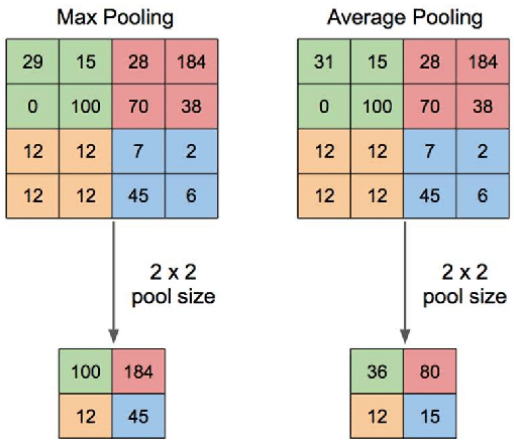

Fig. 3.3: Example of Max Pooling and Average Pooling [34]

3.5.3 ReLU Activation Function

The purpose of utilizing the ReLU is to inject non-linearity into the network. Since images and real-world data exhibit non-linear patterns, it is necessary to employ a non-linear function in order to effectively understand and represent these non-linear relationships. The ReLU function sets all negative pixel values in the feature map to zero. This function is $f(x) = \max(0, x)$, which is computationally efficient and allows the network to converge faster.

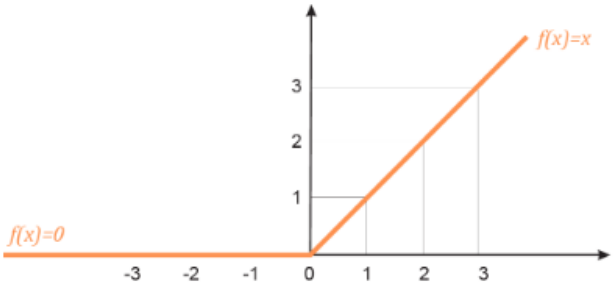

Fig. 3.4: Activation Function ReLU [35]

3.5.4 Fully Connected Layer

To categorize the picture into labels based on the detected high-level features from the convolutional and pooling layers, a fully connected layer is employed. The neurons in a completely connected layer establish connections with all the activations in the previous layer, as is the case in conventional neural networks. Their outputs are determined by the process of matrix multiplication, which is then adjusted by a bias offset. These outputs are then utilized to categorize pictures into different groups, depending on the distinctive characteristics identified by the convolutional layers.

3.5.5 SoftMax Activation Function

In order to allocate decimal probabilities to each class in a multi-class issue, where the sum of the probabilities should be 1, a SoftMax activation function is employed. The SoftMax activation function is used in the output layer of the network to standardize the output into a probability distribution across the expected output classes. SoftMax function denoted as,

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

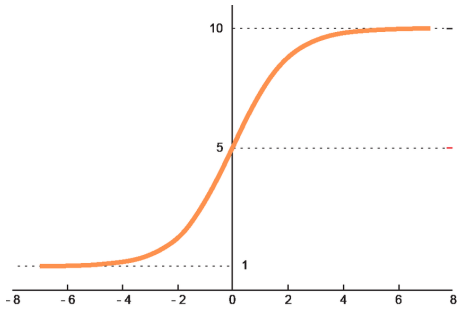

Fig. 3.5: Activation Function SoftMax [35]

3.6 Pre-trained CNN Models for Feature Extraction

We utilized four pre-trained CNN models, namely VGG16, MobileNet, ResNet50, and DenseNet121, to extract features from both images captured in RGB and RGB-D format. Each selection was made based on criteria such as model size, accuracy, and inference speed. Training a CNN from the beginning necessitates substantial computer resources and effort. However, utilizing pre-trained models can conserve both. Another

benefit of utilizing pre-trained models is their exposure to a wide range of pictures, enabling them to be flexible and adaptive across different domains.

3.6.1 VGG16

VGG16 is a CNN architecture developed by the Visual Geometry Group at the University of Oxford [32]. It is highly accurate, with a top-1 accuracy of 71.30% and a top-5 accuracy of 90.1% [36]. The VGG16 architecture is renowned for its straightforwardness and efficiency, and it has been extensively employed for many computer vision applications, particularly for picture categorization.

VGG16 is distinguished by its profound construction, including 16 layers of weights, which serves as the foundation for its model name. To obtain depth, the model utilizes a technique called stacking, which involves using many convolutional layers with tiny 3x3 filters and max-pooling layers. The VGG16 model employs convolutional layers that utilize compact 3x3 convolutional filters with a stride of 1. The VGG16 architecture incorporates max-pooling layers that utilize a 2x2 window and a stride of two. The ReLU activation function is applied to each layer of the network following both the convolutional and fully connected layers. After the convolutional layers, there are three completely linked layers. The last layers are comprised of 1000 completely linked neurons. The model was trained using the ImageNet dataset, which consists of 1000 classes. The last layer is provided with a SoftMax activation function, which is frequently used in classification tasks. It transforms the network's unprocessed output into scores that represent probabilities.

The model was trained using RGB photos of a fixed size of 224 by 224 pixels. The batch size is configured as 256, while the momentum is set as 0.9. The initial learning rate was set at 10^{-2} and subsequently reduced by a factor of 10 until the accuracy of the validation set ceased to improve. The following Fig. 3.6 depicts the architectural design of the VGG16 model.

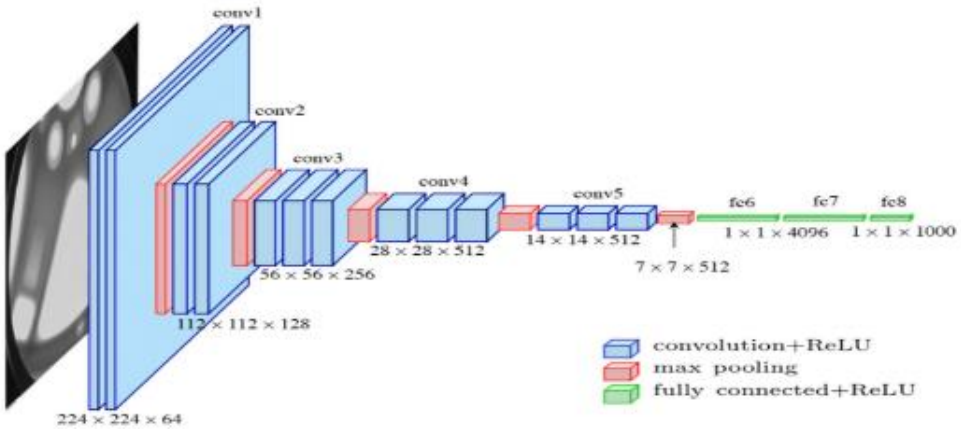

Fig. 3.6: Architecture of VGG16 [37]

3.6.2 MobileNet

MobileNet is a type of CNN structure designed primarily for mobile and embedded vision jobs. MobileNet is characterized by its compact model size and exceptional inference speed, making it the smallest model accessible with the fastest performance on both CPU (22.6ms) and GPU (3.4ms) [36]. The model has commendable precision, with a top-1 accuracy of 70.4% and a top-5 accuracy of 89.5%, considering its model size. These characteristics render this model appropriate for devices with limited resources or applications that require real-time processing.

MobileNet presents an efficient design that employs depth-wise separable convolutions to minimize the computational burden and model size while still achieving competitive performance [38]. This method breaks down regular convolutions into depth-wise and pointwise convolutions, resulting in a substantial decrease in the number of parameters and computational resources needed. This approach divides the convolution process into two components: a depth-wise convolution that applies one filter for each input channel and a pointwise convolution that merges the output channels from the depth-wise convolution. As a consequence, there is a significant decrease in the computational complexity and size of the model. MobileNet, like other CNN systems, utilizes pooling layers. MobileNet commonly employs the ReLU, and it uses the activation function to incorporate non-linearity. MobileNet generally incorporates a fully connected layer that is followed by a SoftMax activation function for the purpose of classification jobs.

MobileNet can handle variable input sizes, but common dimensions include 128x128, 160x160, 192x192, and 224x224 pixels for different versions and applications. MobileNet training adapts to its efficient architecture, utilizing techniques such as batch normalization and dropout to prevent overfitting. The training process is optimized to work with lower precision to increase efficiency further. Fig. 3.7 shows the architecture of MobileNet.

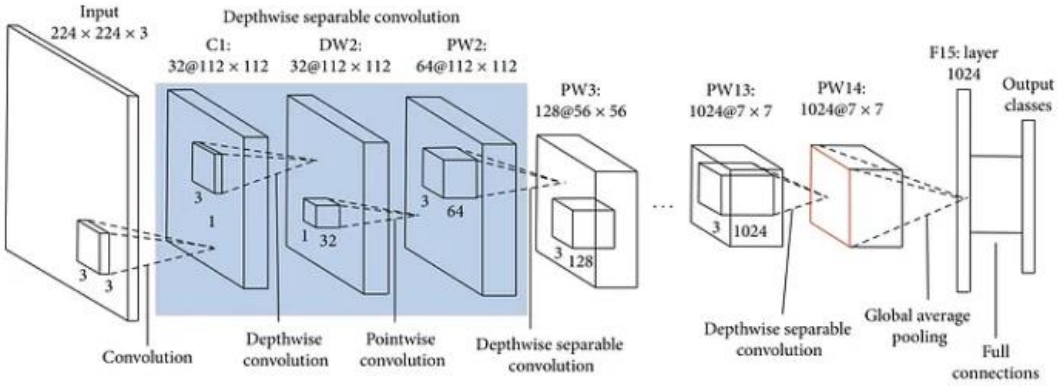

Fig. 3.7: Architecture of MobileNet [39]

3.6.3 ResNet50

The Residual Network 50 is a convolutional neural network model that was proposed in reference [32]. The ResNet50 design is distinguished by its use of residual learning, which aims to enable the training of far deeper networks by solving the issue of disappearing gradients. Despite its relatively tiny model size of 528MB, this model has commendable accuracy with a top-1 accuracy of 74.9% and a top-5 accuracy of 92.1% [36]. ResNet50 is a neural network model with a complex structure consisting of 50 levels. These layers include convolutional layers, identity shortcut connections, and fully connected layers. The fundamental concept of ResNet50 is the use of residual blocks. Every block is equipped with a bypass link that allows for the omission of one or more layers. The shortcut links achieve identity mapping. ResNet50 employs a 7x7 convolutional layer at the start, succeeded by 1x1, 3x3, and 1x1 convolutions in its residual blocks. The 1x1 layers have the task of decreasing and then raising (restoring) dimensions, while the 3x3 layer serves as the constriction point. The first 7x7 convolutional layer is followed by an initial max pooling operation. Before the final fully linked layer, average pooling is implemented. The network utilizes the ReLU activation function consistently. The last layer of the network utilizes a SoftMax

activation function to convert the network's output into probability scores for the classification job. Fig. 3.8 depicts the architectural structure of ResNet50.

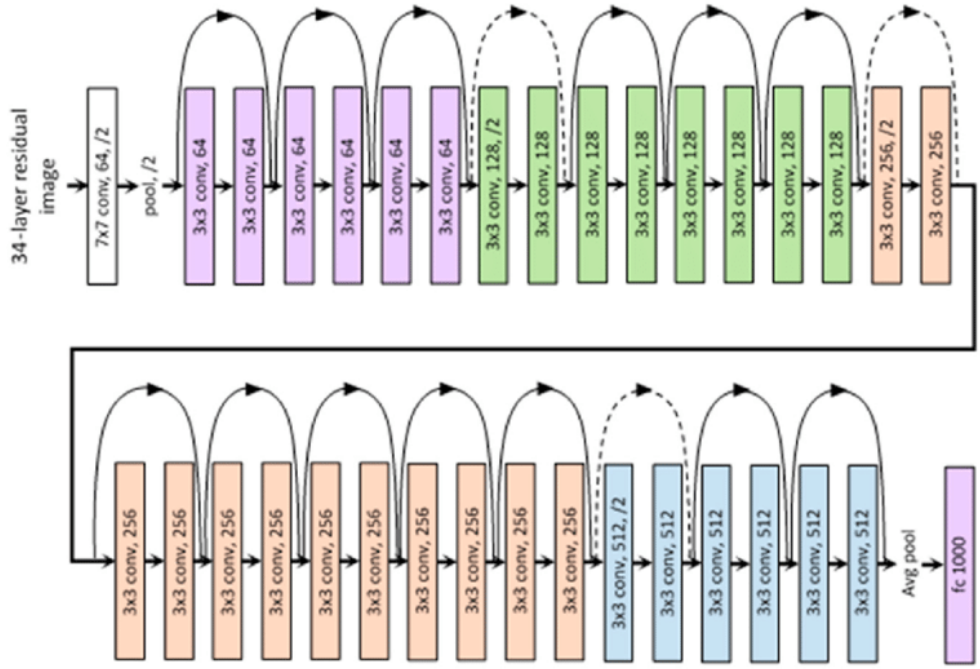

Fig. 3.8: Architecture of ResNet50 [40]

3.6.4 DenseNet121

DenseNet121 is a unique convolutional neural network (CNN) architecture that was first shown in [41]. The DenseNet121 model has a top-1 accuracy of 75% and a top-5 accuracy of 92.3% [36].Thanks to its extensive connection, it exhibits efficient memory utilization. DenseNet121 employs a novel kind of connection where each layer is interconnected with to every other layer in a feed-forward fashion. This architecture was specifically developed to mitigate the vanishing gradient problem.

The model consists of 121 weight layers. The architecture of the system is structured in a way that consists of closely linked blocks. These blocks are divided by transition layers that carry out convolution and pooling operations to decrease the dimensionality of the feature maps. Each dense block has layers that follow a certain sequence: batch normalization, ReLU activation function, 1x1 convolution for reducing feature map dimensionality, another batch normalization, ReLU, and ultimately a 3x3 convolution. The arrangement is commonly referred to as the BN-ReLU-Conv (Bottleneck Layer)

setup. The transition layers that separate the dense blocks consist of convolutional layers followed by 2x2 average pooling layers. The network employs the Rectified Linear Unit (ReLU) activation function. The architecture of DenseNet121 is illustrated in Fig. 3.9.

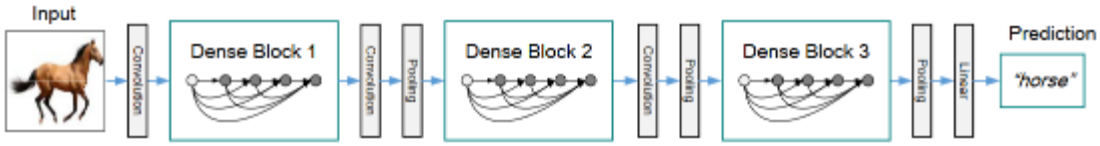

Fig. 3.9: Architecture of DenseNet121 [41]

3.7 Transformer Network

Transformer networks, initially shown in the influential publication [42], have significantly transformed the domain of NLP. The Transformer architecture consists of an encoder and a decoder network. The encoder is positioned on the left side of the Fig. 3.10, whilst the decoder is placed on the right side on the figure.

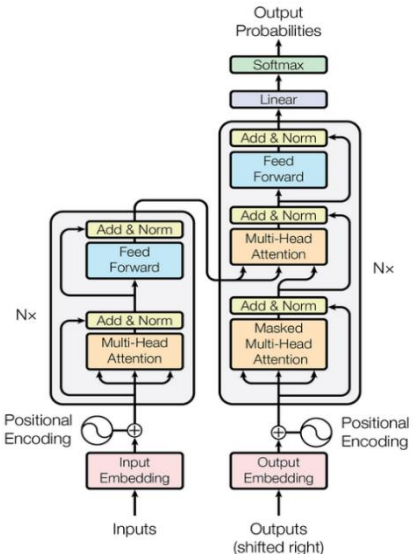

Fig. 3.10: Transformer Model Architecture [43]

The self-attention mechanism is at the heart of the transformer design, allowing the model to assign varying degrees of priority to different portions of the incoming data. The transformer method enables the recording of intricate dependencies and linkages within the data, irrespective of their spatial separation in the input sequence. Transformers, unlike RNN and LSTM, do not analyze data in a sequential manner,

which leads to a notable enhancement in parallelization. We shall explain the operational mechanism of a transformer in the following sub-sections.

3.7.1 Encoder Block Input

The input to the encoder block is a sequence of embedding; in our case, it is the RGB and RGB-D image features extracted through pretrained CNN, or if it is an NLP task, it will be word token embeddings.

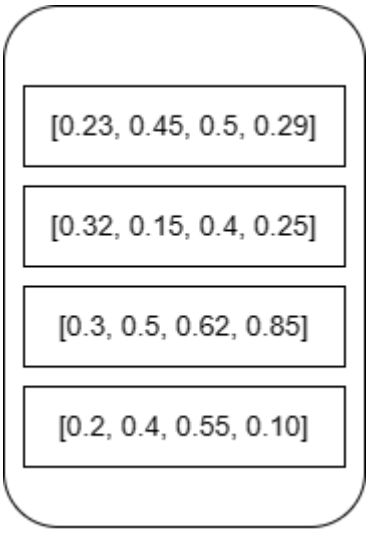

Fig. 3.11: Input Embedding for Each Image

3.7.2 Creating Positional Encoding

Positional encoding involves creating a vector through a specific function that varies according to a condition. For example, for odd-numbered input embedding, a cosine function is used to produce the positional encoding vector, and for even-numbered input embedding, a sine function is applied to generate the positional encoding vector. This method ensures each position in the input sequence has a unique, conditionally generated vector, enabling the model's ability to understand sequence order. Following are the sine and cosine functions used to calculate positional embedding.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Here, pos is the position in the sequence, ranging from 0 to the sequence length, i represents the dimension within the positional encoding vector. The variable d represents the dimensionality of the model's embedding. In the next step, previously created positional encoding vectors will be added to the input vector.

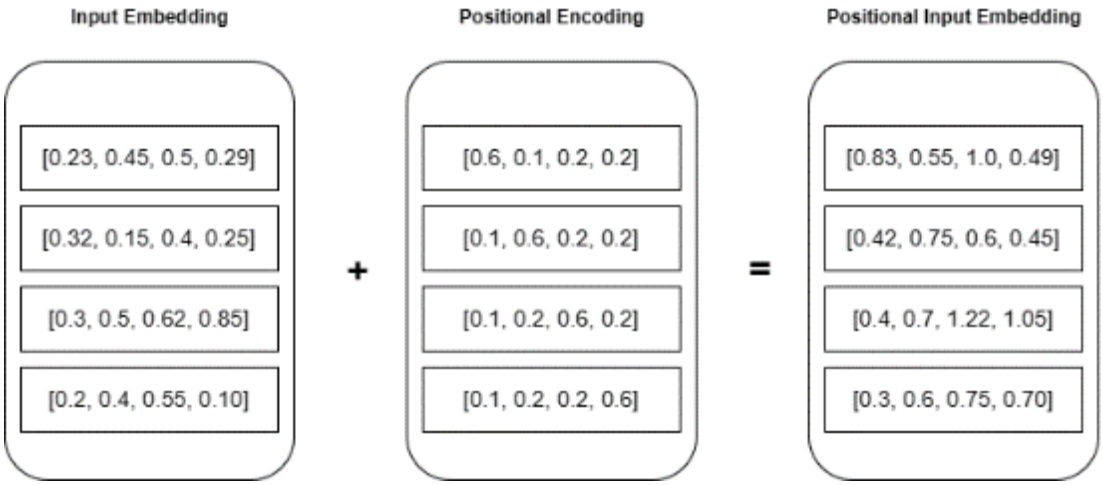

Fig. 3.12: Combining Input with Positional Encoding

3.7.3 Multi-Head Attention Module

In the previous sub-section, we learned how to create positional input embedding by merging input embedding with positional encoding. This embedding produces a set of three vectors, Query (Q), Key (K), and Value (V), for every word or picture in our case. The positional input embedding is sent via a linear neural layer to minimize its dimensionality, as seen in the image below.

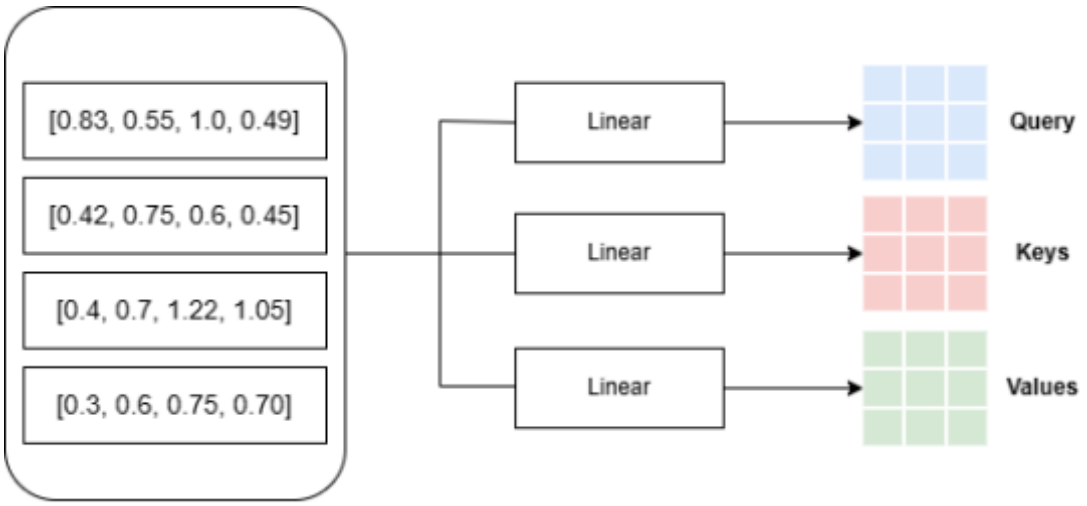

Fig. 3.13: Creating, Q,K and V

The encoder's multi-headed attention utilizes a particular type of attention mechanism known as self-attention. This approach enables the models to create an association between each word/image in the input and the other word/image.

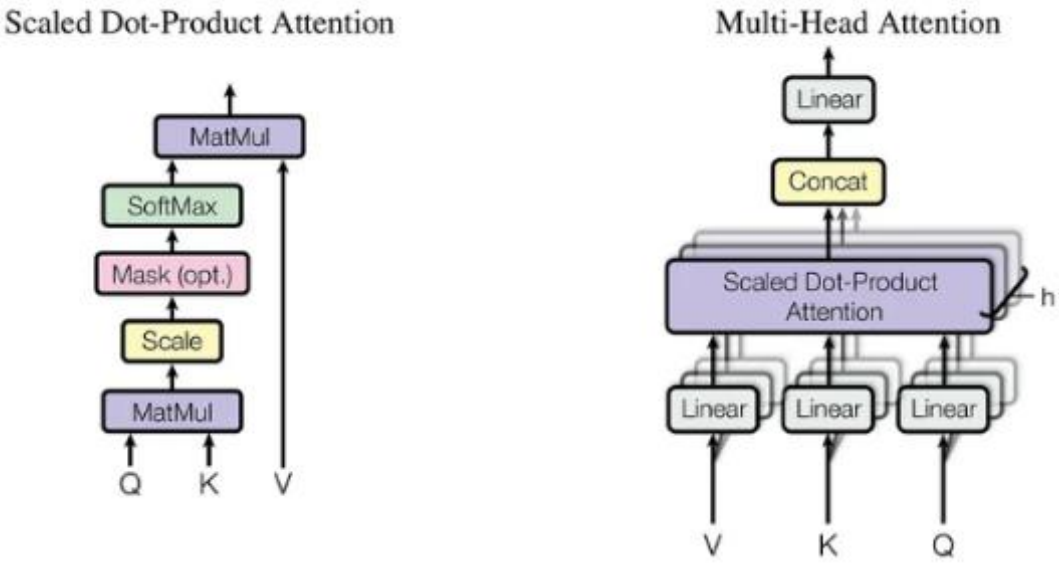

Fig. 3.14: Scaled dot-product (on the left). Multi-Head Attention with Simultaneous Levels(on the right) [20]

The multiplication of the query (Q) matrix by transpose of the key (K) matrix plays a pivotal role in computing attention scores. This operation produces a score matrix, which indicates how much attention or focus a word/image should place on another word/image within the input sequence. This step is essential for enabling the model to dynamically adjust its focus on different parts of the sequence, thereby capturing complex relationships and producing context-aware representations of each word or image. Fig. 3.15 represents the dot product between matrices Q and KT.

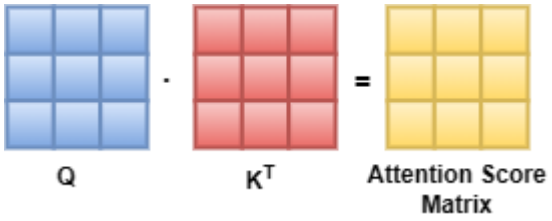

Fig. 3.15: The Dot Product of Q and K^T

Scaling down the score matrix in the attention mechanism is essential for stabilizing the gradient during training. Without scaling, the dot product tends to have a high magnitude. This is achieved by performing a division operation on each member of

the score matrix, where the divisor is the square root of the dimension of the key vectors, denoted as ($\sqrt{d_k}$). After scaling the score matrix in the transformer encoder's attention mechanism, the next step is to apply a SoftMax function to the scaled scores. This operation converts the scaled scores into a probability distribution, ensuring that all attention weights are positive and sum up to 1 for each word/image. Next, the attention weights are multiplied by the value (V) matrix. The multiplication of the attention weights and the value matrix yields the output of the attention mechanism. To calculate the matrix of outputs:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Rather than using a single attention mechanism with keys, values, and queries of dimension d_{model}, authors [42] have discovered an advantage in transforming these elements h times through different learned linear transformations. Each transformation changes the dimensionality of the queries, keys, and values to d_q, d_k and d_v respectively. After transforming, the attention mechanism is applied to each set of transformed queries, keys, and values simultaneously, producing output that are d_v-dimensional. This approach allows the capture of different aspects of the information in parallel, improving the ability of the model to understand and represent the data. Let's say we have H heads indexed by $h = 1, \ldots, H$ of the form:

$$H_h = Attention(Q_h, K_{h,} V_h)$$

At first, the heads are merged together into a unified matrix. Subsequently, this outcome is subjected to a linear transformation using a matrix $W^{(o)}$, resulting in a consolidated output in the specified format:

$$Y(X) = Concat[H_1, \ldots, H_H]W^{(o)}$$

3.7.4 Add and Normalization

Following the completion of the self-attention mechanism, the resultant output vectors are amalgamated and subsequently integrated with a residual connection that stems from the input layer. This residual connection plays a critical role in enhancing the flow of gradients throughout the network, thus promoting more effective learning and backpropagation. By incorporating this residual link, the model effectively leverages

both the initial input information and the insights garnered from the self-attention process. After this integration, the combined output undergoes layer normalization. This normalization step is pivotal as it standardizes the data across the features, which not only stabilizes the learning process but also tends to reduce the overall training time slightly. This series of operations ensures that the model can adaptively refine its internal representations, leading to more robust and accurate learning outcomes.

After the self-attention mechanism concludes, the outputs are processed through a point-wise feedforward network, which aims to cultivate a more intricate representation of the input data. This network is structured with two linear (dense) layers, which are interspaced by a non-linear activation function, typically ReLU. This configuration enables the model to develop a more profound comprehension of the interrelationships among the input elements. By integrating this layering, the model can enhance its ability to distinguish and interpret complex patterns and dependencies within the data, thereby achieving a richer and more detailed understanding. This step is crucial for enabling the transformer to handle various complexities and nuances in the dataset.

Following the feedforward network, the output is subjected to layer normalization, a process that ensures consistency in the distribution of the data features across different instances in the batch. Concurrently, residual connections are established from the preceding layers to this output. These connections are pivotal as they allow the model to incorporate both the insights from earlier layers and the newly processed information from the feedforward network. This integration of residual connections plays a critical role in preserving the continuity of information throughout the network, which significantly enhances the model's ability to comprehend complex relationships within the input data.

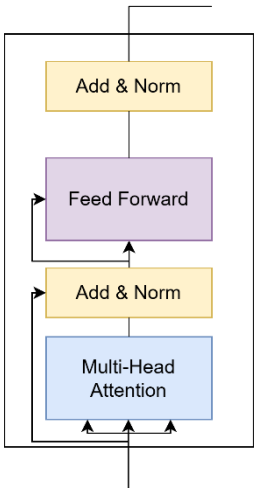

Fig. 3.16: Final Stage of the Encoder Block

3.7.5 Decoder Block

A decoder block combines multiple layers to transform sequences in a context-aware manner, incorporating both previous outputs and encoder information. Initially, it employs a masked self-attention mechanism, where the mask ensures causality by preventing tokens from attending to subsequent tokens, thus preserving the autoregressive property which is essential for generating sequences. This layer calculates attention over the input of the decoder while guaranteeing that each location can only focus on the preceding places in the sequence. Subsequently, the block employs a cross-attention layer in which the queries (Q) are derived from the output of the masked self-attention, while the keys (K) and values (V) are obtained from the output of the encoder. This allows the decoder to selectively concentrate on pertinent sections of the input sequence. Residual connections and layer normalization are used to both attention methods. The last element is a position-wise Fully Connected Feed-forward Neural Network (FCNN), which is used in the same manner as in an encoder.

3.7.6 Encoder-Only Models

In applications, transformers can be used in three architectures, which are encoder-only, decoder-only, and encoder-decoder. For this research, we will use an encoder-only architecture model, which is useful for tasks primarily including sequence classification or language understanding. This architecture primarily relies on self-attention mechanisms, does not utilize causal masking, and employs bidirectional context for tokens. Bidirectional context refers to the model's ability to consider both

the preceding and subsequent tokens in a sequence when understanding or processing any given token.

3.8 Summary

In this chapter, we have discussed the technology adopted for this research, including the CARLA simulator and its components, the main programming language and its libraries, basic CNN architecture, and all four pre-trained CNN models used for testing, and transformer architecture specially giving more attention to encoder block since we going to utilize in this research, which is the main training and inferencing model. In chapter 4, we will discuss the approach to vehicle crash prediction.

CHAPTER 4
AN APPROACH TO VEHICLE CRASH PREDICTION

4.1 Introduction

In Chapter 3, we have justified the suitability of deep learning as the technology adapted for developing a vehicle crash prediction system. This chapter presents our deep-learning approach to a vehicle crash prediction system named CrashGuard. This chapter presents an approach to our solution covering our hypothesis, input, output, process, features, and users. Among others, the process specifically identified the task within CrashGuard, which provides insight into the design of the solution.

4.2 Hypothesis

We have hypothesized that a combination of CNN and transformer networks can predict vehicle crashes, providing multimodal visual input. While driving the vehicle, visually (RGB and RGB-D) monitor the oncoming traffic and environment and give early warning if the vehicle is going to crash into another vehicle or object on the road. The inspiration behind the development of vehicle crash prediction system stems from the desire to improve road safety and reduced the occurrence of accidents. Leveraging advancements in ML, especially in deep learning, aims to create intelligent systems that can accurately predict the likelihood of a crash.

4.3 Input

A comprehensive dataset will be collected using the CARLA simulator; data will be collected as a sequence of RGB and RGB-D images captured by RGB and RGB-D cameras fixed in the car. The Car will be driven all over the simulated environment provided by CARLA. Further, the environment is simulated with various weather conditions and different maps. Images will be captured under two categories which are prior to safely driven condition and prior to some crash occurred. Then, captured data will be preprocessed and transformed into a suitable format. Fig. 4.1 shows the sample of data taken out from CARLA. It contains a sequence of RGB and its respective RGB-D images, where each sequence contains 8 images.

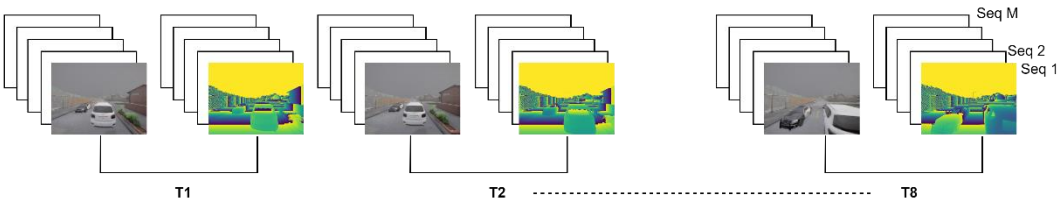

Fig. 4.1: Input Image Sequence

4.4 Output

Based on its analysis, the system generates a binary classification output, which signifies whether the observed driving scenario poses a risk of a vehicle crash or not. The output serves as a valuable indicator for drivers, policymakers, and traffic authorities, allowing them to make informed decisions and take proactive measures to prevent accidents.

4.5 Process

The process begins with captured data preprocessed and transformed into a suitable format. Deep learning architecture is comprised of two networks, pre-trained CNN and transformer. Data will first be input into a pre-trained CNN network. A pre-trained CNN network can extract a wide range of features from images, capturing both low-level and high-level representations. Low-level features such as edges and contours, textures, and colors are extracted in the initial layers of a CNN. Specific arrangements of edges or textures and parts of objects are extracted as mid-level features by pre-trained CNN. For high-level features, object detection, scene recognition, and action recognition can be considered. The extracted features serve as a rich, compressed representation of the visual information in images in the dataset.

Then, the output encoding provided by the CNN network is seamlessly fed into the final classification stage of the transformer network. The revolutionary nature of transformer networks lies in their ability to effectively model and capture long-range dependencies, which has significantly advanced the field of NLP. In our specific project, we aim to leverage the power of the transformer architecture to capture not only temporal dependencies but also contextual information inherent in the driving data. By incorporating the outputs from the CNN network into the transformer

network, we can effectively learn and understand the intricate temporal relationships and patterns that play a crucial role in influencing crash occurrences. This combination of the CNN and transformer networks allows us to uncover hidden insights and make accurate predictions regarding road safety, thereby enabling us to implement proactive measures for accident prevention. Output will be classification whether it is safe from crash or not.

4.6 Features

Here are the primary features of this system:

- Multimodal image-based detection: This system is able to analyze a sequence of multimodal images captured by RGB and RGB-D cameras installed in the vehicle and detect objects, vehicles, pedestrians, and other relevant elements in the surrounding environment.
- Feature extraction module: Using pre-trained CNN to extract relevant features from multimodal images and reduce dimensionality while preserving critical information for crash prediction.
- Temporal modeling: Transformer networks can capture the temporal relationships and patterns present in a succession of pictures. By considering the temporal information, the system can examine the changes over time in the scene and detect possible signs of a crash.
- Real-time prediction: This system provides real-time predictions and warnings to the driver or autonomous driving system to take preventive actions. This allows for timely intervention and the avoidance of potential accidents.

4.7 Users

Vehicle manufacturers could be interested in integrating such a system into their vehicles as an advanced safety feature. By predicting potential crashes, they can develop proactive measures to enhance vehicle safety and reduce the risk of accidents. Subsequently, this will help drivers drive carefully with an added early warning system.

4.8 Summary

This chapter discusses the methodology for designing a car collision prediction system utilizing pre-trained CNN and transformer networks. We have addressed the hypothesis, input, output, method, features, and users. We have identified four primary tasks specifically data collection, data pre-processing, feature extraction, and model training and forecasting. The next chapter provides a description of the CrashGuard, with specific reference to the process specified in this section.

CHAPTER 5

DESIGN OF CRASHGUARD

5.1 Introduction

This chapter will provide an explanation of the vehicle crash prediction design. The goal of this system is to develop an advanced vehicle crash prediction model using a combination of pre-trained CNN and Transformer networks. By analyzing a sequence of images captured by dashcam videos, the system aims to accurately predict potential crash events, enabling proactive measures to enhance road safety.

5.2 Top-level Architecture

Data collection is done using the CARLA simulator, where RGB and RGB-D cameras are attached to the ego-vehicle and drive around the city to capture safe and unsafe situations. Then, captured images will be resized, normalized, and augmented to enhance the model's generalization capabilities. Preprocessed data is subsequently divided into training and testing sets. Then, the training set is used to train the pre-trained CNN and transformer network. Pre-trained CNN for feature extraction and transformer network for training and classification. Classification is binary whether it is safe or unsafe. Fig. 5.1 shows the top-level architecture diagram of the design.

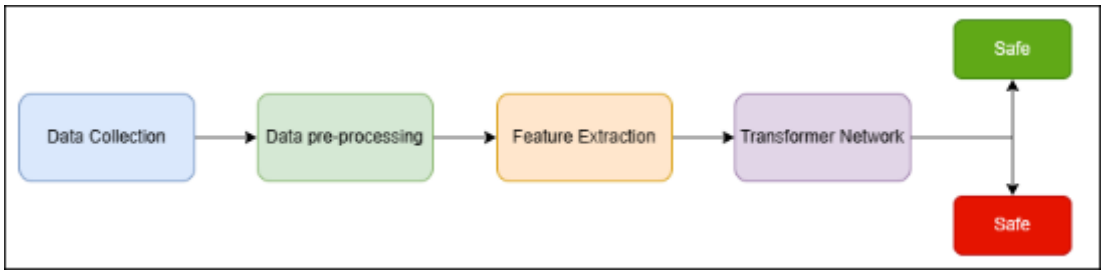

Fig. 5.1: Top-level Architecture of the Design

5.3 Data Collection From CARLA

In order to gather a diverse and comprehensive dataset for training the transformer model, a simulation environment was meticulously created CARLA simulator. This virtual setting featured a fleet of 100 autonomous vehicles, ensuring a wide array of vehicles to be captured by the model and their driving behaviors and scenarios. To

make the environment more challenging, another 20 pedestrians were integrated into the simulation. This enhances the realism and complexity of the traffic conditions that are captured by RGB and RGB-D cameras.

This simulation did not just focus on one default map, a total of four maps were selected, which are provided by CARLA. The range of maps represents different urban layouts and road networks, each with its unique challenges. Fig. 5.2 represents the different maps used in simulations.

Fig. 5.2: Different Maps

To ensure the model's robustness in varied operational conditions, different weather conditions were created in addition to different maps. This includes changes in lighting, precipitation, and fog density, among other atmospheric conditions. To change the weather conditions, CARLA's already built-in presets, such as wet, cloudy sunset, hard rainy noon and hard rainy sunset were used. Fig. 5.3 depicts those weather conditions.

Fig. 5.3: Different Weather Conditions

From each scenario, safe and unsafe 8000 sequences were captured, for a total of 16000 sequences. One sequence consists of RGB and RGB-D images, and each format contains 8 images. During data collection, we collected 15 images per sequence and later reduced to 8 by removing 7 images at the end; this reduction is intended to enhance the model's ability to predict an impending collision before it occurs. Of these, 11200 were reserved for training, and the remaining 4800 sequences were used for testing. When separating data for training and testing, it was done without any class imbalance and distributed different maps and weather conditions with balance, and when labeling, the safe class was labeled as 0 and the unsafe class was labeled as 1.

5.4 Data Pre-Processing

Collected RGB and RGB-D images cannot be directly input into a pre-trained CNN model; before that, we need to do some pre-processing. First, images must be read with the Python “cv2” library and then converted into RGB format because “cv2” will read images in BGR format by default. As the next step, again using the “cv2” library, the image is resized into a 224 x 224 dimension. Resizing is necessary because it is a requirement of the architecture of the pre-trained CNN model. Resizing also gives some added benefits, such as training on a smaller, uniform image size, which can lead to faster computations and less memory usage during training. After resizing, we get an array of 224 x 224 x 3; 3 is the number of channels in the image. That array needs to be normalized before feeding into pre-trained CNN by dividing it 255; this will make values in the array between 0 and 1. The benefits of doing so are neural networks often perform better and cover faster when the input features are scaled to a smaller consistent range, it gives uniformity in Pixel Values, and during backpropagation, having smaller normalized values helps in maintaining a more stable gradient flow.

5.5 Feature Extraction with Pre-Trained Network

To extract features, RGB and RGB-D were separately fed into pre-trained CNN models, and the extracted features were then concatenated. This technique is concatenated fusion, which comes under early-level fusion [23], and Fig. 5.3 shows the diagram of it. For pre-trained models, we have used VGG16, MobileNet, ResNet50, and DenseNet121. The output dimension depends on the selected pre-

trained model. Table 5.1 represents the number of features extracted for each pre-trained CNN model.

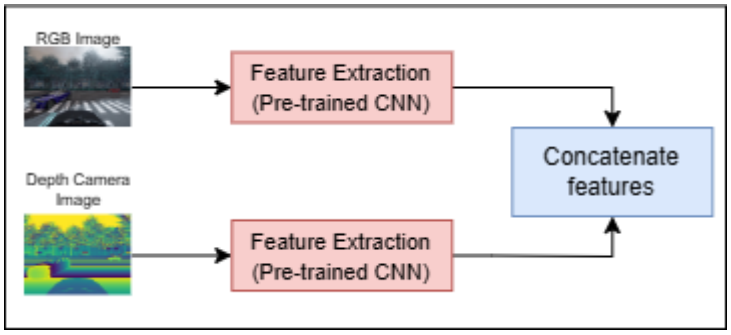

Fig. 5.4: Feature Extraction and Concatenation

TABLE 5.1: CNN MODELS AND NO OF EXTRACTED FEATURES

CNN Model	**No of Extracted Features**	**Concatenated Features**
VGG 16	512	1024
MobileNet	1024	2048
ResNet50	2048	4096
DenseNet121	1024	2048

Once the features from each pair of RGB and RGB-D images in one sequence are concatenated, these features are assembled into an array with the shape of 8 x [output dimension] of the model; for example, in VGG16 model array shape is 8 x 1024. This results in a structured form where each row in the array corresponds to the feature vector one RGB/RGB-D image pair ready for subsequent processing steps. In this array, rows represent RGB/RGB-D image pairs, and columns represent concatenated features. Finally, we create four arrays, features for training and testing, and their corresponding labels. As final arrays (for VGG16), we get (11200 x 8 x 1024) (11200,) size for train feature input and labels, and (4800 x 8 x 1024) (4800,) for testing feature input and labels.

5.6 Training Model with Transformer

In our research, we will use only the encoder block of the transformer network since it is a classification task given a sequence of inputs. Our training input feature array

and label array will be fed in. In modeling the temporal dependencies among the extracted features from an image sequence, a transformer architecture can be highly effective. This architecture utilizes self-attention mechanisms between different frames in the sequence. Unlike traditional models that may only capture local interactions, self-attention allows the model to weigh the importance of all frames, regardless of their position in the sequence. Additionally, positional encoding is integrated into the transformer network to encode the temporal order of the images. After coding the transformer architecture with Keras Fig. 5.4 shows the transformer architecture used in our model training.

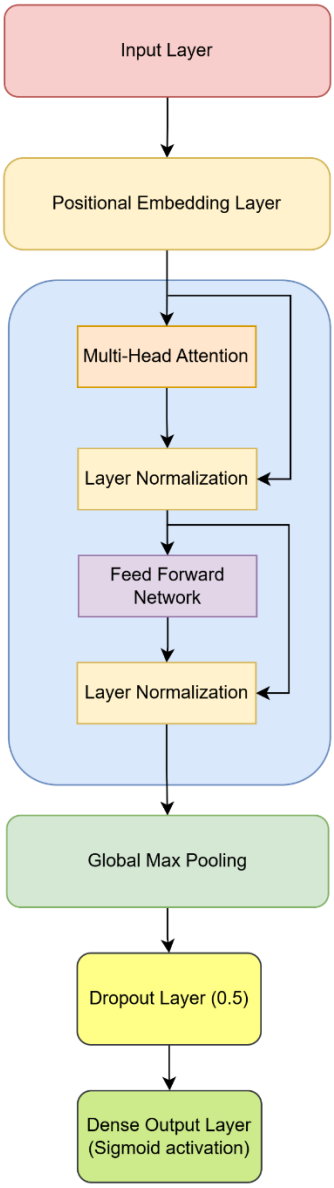

Fig. 5.5: Encoder Block of the Transformer [43]

5.7 Evaluation

Once the model has been trained, it is necessary to assess its performance. We have maintained a distinct dataset that was not utilized throughout the training phase. To assess the success of our model, we may include quantitative indicators such as accuracy, F1-score, precision, and recall. And also Training Vs. Validation Loss and Accuracy graphs and confusion metrics are used for evaluation. Additionally, we will compare our findings with existing research that shares the same purpose but employs different methodologies.

5.8 Summary

In this chapter, we discussed the high-level architecture of the proposed solution, which includes main processes such as feature extraction with a pre-trained CNN network, training with a transformer network, and additionally evaluation. In the next chapter, we describe the implementation of the proposed solution.

CHAPTER 6

IMPLEMENTATION OF CRASHGUARD

6.1 Introduction

This chapter will focus on the practical implementation of the suggested approach for predicting automobile crashes. The procedure we outlined in the approach chapter allows for the implementation to be divided into primary portions. The practical execution of our project consisted of five main steps: data collecting, data preprocessing, feature extraction, training using a transformer network and evaluation.

6.2 Installing CARLA Simulator

As we explained earlier, the data needed for this research is collected through the CARLA simulator. We need to begin by setting up the necessary development environment to install the CARLA simulator. First, we need to download and install Visual Studio 2019 and ensure to include the necessary workloads for game development with C++, which are critical for Unreal Engine integration. After Visual Studio is set up, download and install Unreal Engine, selecting a version that is compatible with CARLA. If these prerequisites are in order, we can then install the CARLA simulator. This is done by cloning the CARLA repository from GitHub and then building the simulator using Unreal Engine. The build process compiles the CARLA source code and integrates it with Unreal Engine. Once compiled, we can launch CARLA from within the Unreal Engine environment. To communicate with CARLA, we need to install the Python Carla package.

6.2 Data Collection

Data collection was completely conducted through the CARLA simulator with the help of the Python CARLA API library. Data collection was carried out on two scenarios: ego-vehicle drive in the environment without any collision and ego-vehicle met with a collision with objects in the environment.

Appendix - A shows the script used for importing necessary Python libraries, setting up basic configuration and connecting with the CARLA simulator. We need to import mainly the Carla library to connect with the CARLA server and the numpy and cv2

libraries for image manipulation. As basic configuration, we are providing capturing image width and height (640 x 480), RGB and RGB-D image saving folders, sequence length (15), and the number of autonomous vehicles (100) and pedestrians (20) needed to create within the simulated environment. Then, we create the Carla client object and the Carla blueprint library, which is used to create objects in the CARLA simulator.

The part of the script created for data collections (see Appendix – B), where we defined the "CarlaSession" python class with several methods under that. As the first method, we have created "add_vehicles", which is used to populate simulation environments with autonomous vehicles as per the number of vehicles we defined in Appendix - A.

To create more dynamic weather conditions and add pedestrians to simulation we are using "add_pedestraians" methods (see Appendix – C).

In order to collect data from the simulated environment we need to spawn ego-vehicle and attach RGB, RGB-D camera sensors and a collision sensor. Appendix - D shows the script used for that. In this script, the "Mustang" car is created as an ego-vehicle and spawned in a random location on the map. We have also given some attributes to RGB and RGB-D cameras, such as image size in width and height, shutter speed (200) and field of view (100).

Our spawned ego vehicle will drive autonomously through the simulated environment, following the guidelines set by the simulator. To trigger a crash, the "get_directions" method is scripted, see Appendix - E. This method adjusts the ego vehicle's speed and trajectory, creating specific collision scenarios.

To initiate the "CarlaSession" class, the "start_new_seq" method is explicitly defined. Once instantiation of this class, this method is invoked to generate an ego-vehicle equipped with sensors. Additionally, two helper methods are implemented to facilitate the saving of RGB and RGB-D images. The implementation details for these methods are depicted in Appendix - F

6.3 Data Pre-processing

The images must undergo preprocessing before using a pre-trained CNN for feature extraction. Initially, the images are loaded using the "cv2" library in Python, which reads them in BGR format by default. These images need to be converted to RGB

format for compatibility with the CNN. Subsequently, the images are resized to dimensions of 224x224 pixels, which are the accepted dimensions of the CNN model. The final step involves normalizing the image arrays. The Python script for these preprocessing steps is illustrated in Appendix - G.

6.4 Feature Extraction and Concatenation

The feature extraction process from RGB and RGB-D images was conducted through pre-trained CNN models. We can import the necessary pre-trained models easily using “Keras” Python library. There is a function defined to return an instance of the pre-trained model. For the implementation of the VGG16 model, see Appendix - H.

The "create_model" function was used to develop a script that extracts features from RGB and RGB-D photos. These features are then combined and returned as a feature rray, see Appendix - I.

6.5 Data Training Using Transformer Network

The inclusion of positional embedding within the transformer encoder block is essential for the functioning of the transformer model. It enables the model to capture the token order in sequences, enhancing its capacity to comprehend temporal or sequential links in the data. The inputs consist of the sequence length, which is 8 in our example, and the dimension of the array, which varies based on the pre-trained CNN model. The code for positional embedding may be seen in Appendix - J.

Our transformer model only utilizes the encoder block since it is only going to predict binarily classified features; for that, the script in Appendix - K is used to create an encoder block. The inputs for that are the dimension of extracted features, the size of the inner layer within the transformer encoder and the number of attention heads.

In addition, we have developed utility functions, as seen in Appendix - L, specifically for the purpose of training. There exists a method to compile the model with the following values: The model uses a dropout value of 0.5, the output activation function is SoftMax, the optimizer is Adam, and the loss function is sparse categorical cross-entropy. When invoking the "run_experiment" function, we provide the batch size as a hyperparameter.

6.6 Creating Demo Video

For creating a demo video of how the system works, a helper function was written, see Appendix - M. Here, we can input the sequence of RGB and RGB-D and the images captured by the ego-vehicle while driving around in the environment. The function will use 8 images from both modal and do feature extraction with another function, and predict the situation with the provided extracted features. The function will also create a video using only RGB images and annotate the videos according to the prediction given by the model.

6.7 Summary

In this chapter, we have discussed how to implement our research, starting with installing the CARLA simulator in our system. After running the simulator, we need to collect data for that "CarlaSession" class, which was implemented with several methods. These methods helped to capture data in both safe and unsafe situations. Captured data cannot be fed directly into the transformer network; we need to perform some pre-processing. For that, a separate Python script was developed, which includes converting images in RGB format and normalizing. The next step for training the transformer model was built mainly using the Keras library. Since we need binary classification after feeding the sequence of data, only an encoder block was developed. Apart from the main transformer model, a couple of utility functions were developed to compile and run the model. In the next chapter, the evaluation of the transformer model with different pre-trained CNN models and their accuracy will be discussed.

CHAPTER 7

EVALUATION OF CRASHGUARD

7.1 Introduction

The purpose of this assessment chapter is to determine the precision of the model that was trained using a transformer network for predicting car crashes. The model utilizes characteristics that were retrieved using several pre-trained CNN models. The tests were conducted on the test dataset, comprising 4,800 sequences. The main goal of this model is to utilize the spatial characteristics of individual frames and the temporal connections within a sequence in order to attain high predicted accuracy. This chapter presents the approaches used to assess the success of our strategy and examines the findings gained from these methods.

7.2 Transformer Setup

The training configuration for the transformer model intended to handle sequences of pictures comprises many essential elements and parameters that are crucial for maximizing its performance. The architecture of our approach includes 10 multi-head attention layers. This arrangement enables the model to attend to information at several points in the input sequences concurrently. This is especially advantageous for comprehending the intricate spatial-temporal connection in image data.

The training of this model was conducted with a batch size of 16. This particular size achieves a good trade-off between computational efficiency and model performance. It ensures that there is enough data for each iteration without using excessive memory. The training was scheduled to run for 100 epochs. In order to mitigate overfitting and enhance the model's ability to perform well on new, unknown data, early stopping was incorporated into the training process. The early stopping mechanism was set with a patience of 20 epochs. This implies that the training process would stop if there is no improvement in the validation loss over a continuous period of 20 epochs. A dropout rate of 0.5 was implemented during the training phase. Dropout is a regularization method employed to mitigate overfitting by randomly deactivating a portion of the neurons to zero during the training process. Within this configuration, a random selection of half of the units is eliminated after each training cycle.

The Adam optimizer was chosen due to its adjustable learning rate capabilities, which enhance its efficiency in converging inside intricate landscapes commonly encountered in deep-learning models that process high-dimensional input, such as photographs. The utilized loss function was sparse categorical cross-entropy. Table 7.1 presents a concise overview of the parameters included in the transformer model.

TABLE 7.1: SUMMARY OF TRANSFORMER PARAMETERS

Parameter	**Value**
Number of multi-head attention	10
Batch size	16
Epochs	100
Early stopping	True (Patience 20)
Optimizer	Adam
Loss function	Sparse categorical cross-entropy
Dropout	0.5

7.3 Classification Metrics

Comprehending the concepts of accuracy, precision**,** recall, and F1-score is of utmost importance since these metrics are extensively employed for evaluating the performance of classification models. We have employed all the specified models to assess the performance of our trained model.

7.3.1 Precision

Precision quantifies the degree of correctness in optimistic predictions. Formally, it is defined as the quotient of the number of correctly anticipated positive outcomes divided by the total number of expected positive outcomes. Mathematically, precision is measured:

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

Precision answers the question, “Of all the instances classified as positive, how many are actually positives ?”. It quantifies the level of accuracy attained in making

optimistic predictions. Accuracy is especially crucial in scenarios where the occurrence of False Positives is of greater significance than that of False Negatives. For instance, in the context of email spam detection, a high precision indicates that a smaller number of non-spam emails are mistakenly identified as spam, thereby preventing inconvenience to consumers.

7.3.2 Recall

Recall, or sensitivity, quantifies the model's capacity to identify all pertinent instances. The term refers to the ratio of true positives to the sum of true positives and false negatives. In terms of mathematics:

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

Recall addresses the question, "Of all the actual positives, how many were correctly identified? It focuses on the completeness of the positive predictions. Recall is critical in our research because missing a positive case (Unsafe situation) could be fatal, high recall ensures that most positive cases are caught.

7.3.3 F1-Score

The F1-score is the harmonic mean of precision and recall. This means it is a balance between precision and recall. It is defined as:

$$F1 - Score = 2 \times \frac{Precison \times Recall}{Precision + Recall}$$

The F1-score conveys the balance between precision and recall. It is particularly useful when we need a single metric to compare models where the trade-offs between precision and recall need to be balanced.

7.4 Evaluation of Feature Extraction Methods

This section investigates the comparative analysis of different feature extraction techniques employed within our transformer-based model framework to assess their impact on performance. Specifically, we have utilized four distinct pre-trained CNNs, VGG16, MobileNet, ResNet50, and DenseNet121, as the feature extractors. Each of

these CNN architectures offers unique characteristics and capabilities in terms of parameter efficiency, depth, and architectural innovations, which are hypothesized to influence the overall effectiveness of the transformer model in processing image sequences. This evaluation aims to uncover which pre-trained CNN architecture best matches the transformer network, enhancing its ability to capture and utilize the temporal and spatial dependencies inherent in the image data. By systematically testing each pre-trained CNN, we provide insights into how different feature representations affect the learning dynamics and predictive accuracy of the model, thereby guiding future model optimization and application strategies.

7.4.1 Transformer Network with VGG16

Both training and validation accuracy remain relatively high and close together through the training process, left graph of Fig. 7.1. This suggests that the model generalizes well and is learning features that are relevant to unseen data. The slight fluctuations in validation accuracy could be due to the complexity of the model, the nature of the dataset.

When comparing the training loss and validation loss, it is regularly observed that the training loss is generally smaller than the validation loss. The graph shows some volatility in validation loss, which could be common in smaller datasets or with certain optimization algorithms. The overall trend, however, does not indicate overfitting since validation loss does not increase as the training progresses.

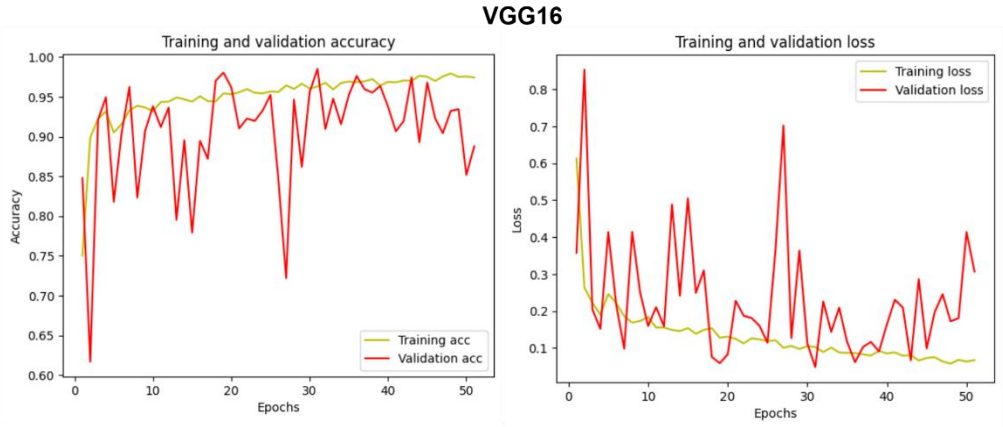

Fig. 7.1: Accuracy and Loss Graphs for Training vs Validation for VGG16 Based Feature Extraction

We got testing accuracy and an F1-score of 0.95, which is quite high, indicating that the model performed well on the task. The recall is 0.98 is especially good, meaning that the model is able to identify 98% of relevant instances. The precision of our model is 0.92, indicating a good level of accuracy in correctly predicting positive classes. The confusion matrix for the model trained with VGG16-based feature extraction is shown in Fig. 7.2.

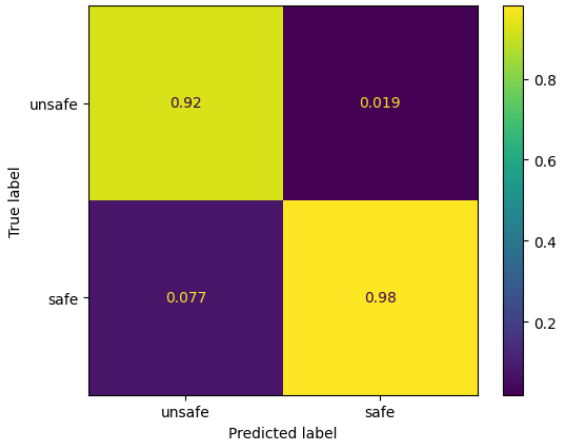

Fig. 7.2: Model Confusion Matrix for VGG16 Based Feature Extraction

7.4.2 Transformer Network with MobileNet

Training and validation accuracy are quite close throughout the training process, with validation accuracy tracking slightly below training accuracy, which is expected (left graph of Fig. 7.3). There's a slight downward trend in validation accuracy toward the end of the epochs, which might be a sign of beginning overfitting, which was avoided due to early stopping or could be due to the variability of the validation set.

The training loss exhibits a consistent and gradual decline, indicating that the model is acquiring knowledge effectively. The validation loss exhibits variations, characterized by significant spikes, suggesting that the model has difficulties in generalizing to the validation set at certain instances. However, since the last value is lower, the model may not be overfitting (right graph of Fig. 7.3). The gap between training and validation loss is small towards the end, suggesting a decent generalization.

Fig. 7.3: Accuracy and Loss Graphs for Training vs Validation for MobileNet Based Feature Extraction

We got an accuracy of 0.92 and an F1-score of 0.92, which are strong indicators of a high-performing model. For precision, we go 0.88, which is slightly lower compared to the other metrics, that indicates that when the trained model predicts a positive result, it is correct 88% of the time. The most valuable metric in our research, recall, got 0.97, which is excellent, indicating the model is identifying the relevant instances almost all the time. Fig. 7.4 shows the confusion matrix for the transformer model trained with MobileNet based feature extraction.

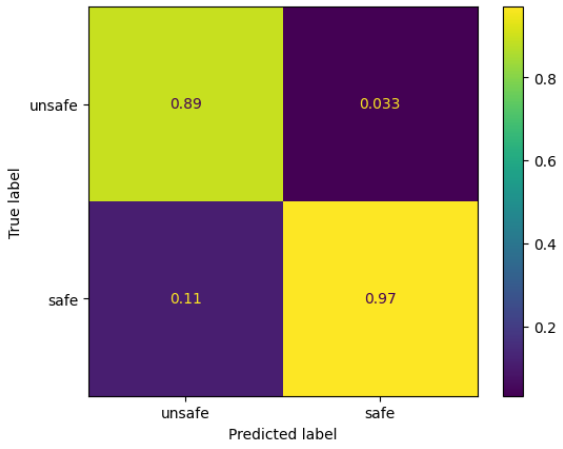

Fig. 7.4: Model Confusion Matrix for MobileNet Based Feature Extraction

7.4.3 Transformer Network with ResNet50

The training accuracy is quite stable but low, around 0.5, which is the accuracy of a random guess in binary classification tasks. The validation accuracy is extremely

volatile, with perfect accuracy scores at certain epochs followed by a drop to 0. This pattern is not typical and suggests severe overfitting on a subset of the data or potential issues with the validation set or data processing (left graph of Fig. 7.5).

The training loss decreases, which is typical and indicates learning. However, the validation loss exhibits high variance, with very noticeable spikes. This erratic behavior in validation loss could indicate a problem with the extracted features (right graph of Fig. 7.5).

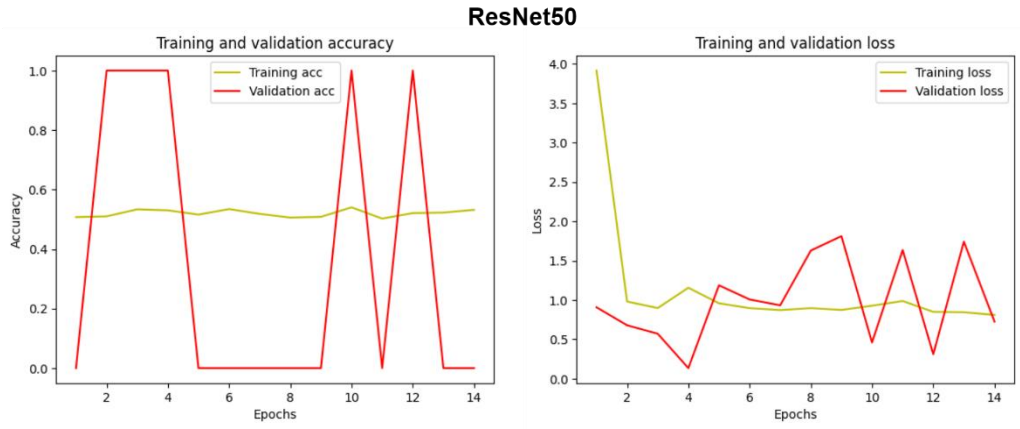

Fig. 7.5: Accuracy and Loss Graphs for Training vs Validation for ResNet50 Based Feature Extraction

When it comes to performance metrics, we got an accuracy and precision of 0.5, which indicates that the model is not better than random guessing. This is not a good sign for model performance, and the model is only able to predict one class. A recall of 1 suggests that the model is classifying all positive instances correctly, but given the low precision, it is likely also classifying many negative instances as positives, which is typical for models that predict only one class. An F1-score of 0.66 is not particularly high and suggests an imbalance between precision and recall in this case skewed towards recall. Fig. 7.6 shows the confusion matrix for the transformer model trained with features extracted from ResNet50.

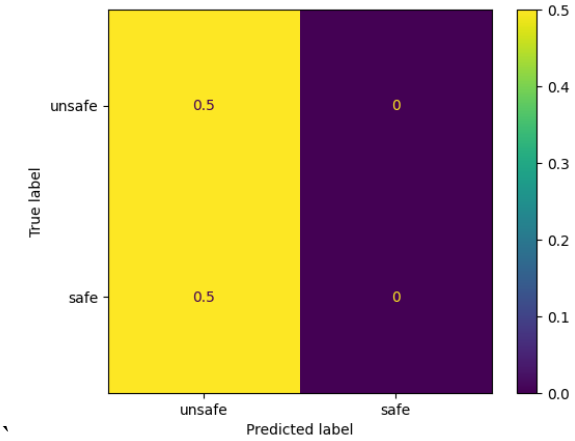

Fig. 7.6: Model Confusion Matrix for ResNet50 Based Feature Extraction

7.4.4 Transformer Network with DenseNet121

The training accuracy consistently exhibits a high and steady performance, indicating that the model has effectively acquired knowledge from the training data. The validation accuracy has a consistent rising trajectory, remaining in close proximity to the training accuracy. However, there is a noticeable decline that aligns with the sudden increase in validation loss, as seen in the left graph of Fig. 7.7.

The training loss shows a general downward trend, which is good. The validation loss decreases overall but has a significant spike around epoch 14. This might suggest a batch of particularly challenging validation data or possible overfitting at that point (right graph of Fig. 7.7).

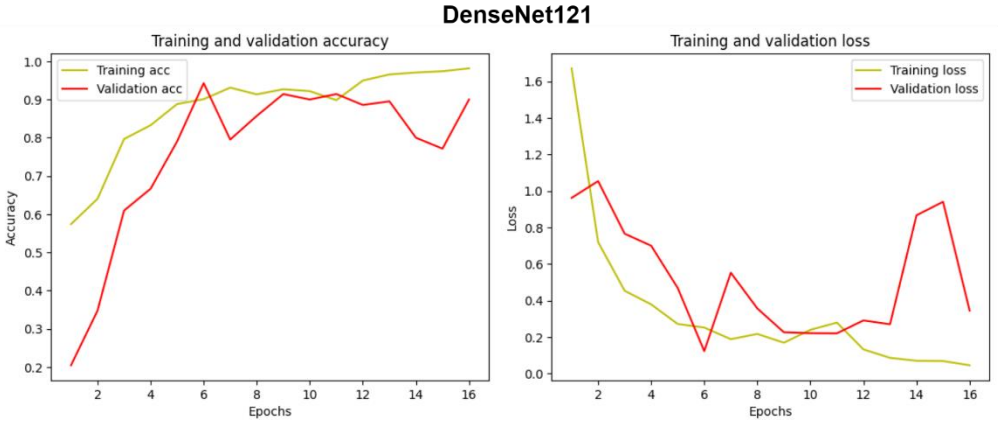

Fig. 7.7: Accuracy and Loss Graphs for Training vs Validation for DenseNet121 Based Feature Extraction

After evaluating the model, an accuracy of 0.93 and an F1-score of 0.93 were obtained, indicating that the model's predictions are generally reliable. For precision and recall we got 0.91 and 0.95 respectively, which both are considerably high.

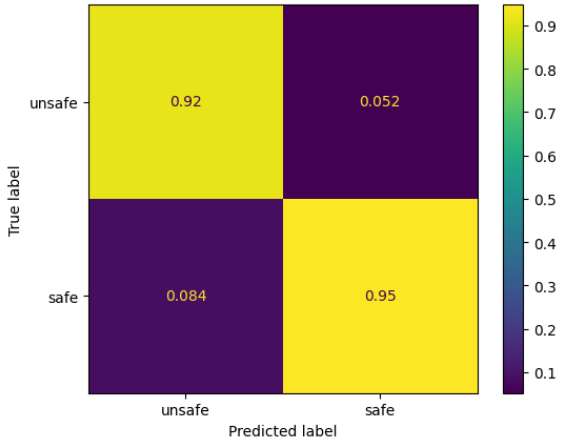

Fig. 7.8: Model Confusion Matrix for DenseNet121 Based Feature Extraction

7.5 Evaluation and Selection of Model for Crash Detection

Out of all pre-trained CNN models used for feature extraction, VGG16 has the highest accuracy (0.95) and very high precision (0.92), and recall (0.98). This suggests that it is effective in both identifying collisions (high recall) and ensuring that detected collisions are likely real (high precision).

MobileNet shows a good balance but slightly lower performance metrics compared to VGG16 in all aspects. While still effective, it is a more lightweight model, which could be beneficial if computational efficiency is a priority.

ResNet50 shows significantly lower performance in accuracy and precision but a perfect recall. The high recall indicates it detects all collisions, but the low precision suggests many false positives. The very low accuracy hints at a potential feature extracted from ResNet50 that is not good enough for this task.

DensNet121 shows good accuracy and precision, slightly lower than VGG16 but better than MobileNet. It has a relatively high recall, indicating it also effectively identifies collision scenarios.

TABLE 7.2: SUMMARY OF PERFORMANCE METRICS IN PERCENTAGE FOR EACH FEATURE EXTRACTION METHOD

Model	**Test Accuracy**	**Precision**	**Recall**	**F1-Score**
VGG16	95%	92%	98%	95%
MobileNet	92%	88%	97%	92%
RestNet50	50%	50%	100%	66%
DenseNet121	93%	91%	95%	93%

7.5.1 Consideration Beyond Raw Metrics

- **Computational Efficiency**: MobileNet is designed to be more efficient than VGG16 and DesnseNet121, which might be beneficial in real-time applications where resources are constrained.
- **Model Complexity:** Although VGG16 performs well, it is quite heavy in terms of parameters and computation. DenseNet121 also shares this trait but is more efficient in parameter usage due to its dense connectivity patterns.
- **Application-Specific Needs:** Depending on the application, the trade-off between recall and precision can be critical. In our research, missing a collision scenario is more problematic than false positives, so a model with higher recall is preferred.

7.5 Result Comparison with Other Research

In [8], researchers have collected only RGB camera images from the CARLA simulator and trained with the ensemble of the network. They have primarily used Convolutional-LSTM for their architecture. As a result, their best model has an accuracy of 0.94, precision of 0.93, and recall of 0.9. Compared with our VGG16-based model, our model has slightly higher accuracy (0.95) and recall (0.98). Our research on higher recall would be advantageous, especially in proactive crash avoidance systems where missing an actual scenario could have dire consequences.

The use of RGB-D data in our research could be seen as an advantage since depth information can be crucial for accurately understanding scene context, which might explain the improved recall rate. Furthermore, the transformer attention mechanism

focuses on relevant parts of the sequence for prediction, potentially handling long-range dependencies better than LSTMs.

7.6 Summary

In this chapter, we have discussed the transformer configuration used for model training, performance metrics, and evaluation of four models using all those metrics. Given the high accuracy, precision, recall, and F1 score, VGG16 appears to be the best choice for implementation based on the current performance metrics. It strikes an excellent balance between all metrics, suggesting that it can correctly identify most collision scenarios with few false positives or missed detections. However, if computational resources are a concern, MobileNet may be considered for its efficiency, though with a slight compromise on performance. In the next chapter, we will discuss the conclusion and future works.

CHAPTER 8
CONCLUSION AND FURTHER WORK

8.1 Conclusion

This research focuses on enhancing vehicular safety through the integration of advanced warning systems capable of predicting imminent collision with static or dynamic objects in the environment. Utilizing the capabilities of the transformer model, originally renowned for its exceptional performance in NLP, we extended their application to the domain of images for real-time crash prediction. One of the primary objectives of this project is to augment conventional vehicles with an intelligence warning system. This system leverages both RGB and RGB-D cameras as input devices, enabling the perception of the vehicle's surroundings through color imagery and depth data. By processing these inputs through the transformer model, the system can accurately detect potential collision scenarios and alert the driver thereby significantly enhancing road safety.

The performance of the transformer model depends on the pre-trained CNN model used for feature extraction. This study focused on evaluating the performance of the transformer model based on four prominent pre-trained CNN models used for feature extraction: VGG16, MobileNet, ResNet50, and DenseNet121. Features extracted from those models were used for collision detection in a simulated autonomous driving environment using the CARLA simulator.

Our findings indicate that VGG16 emerged as the effective model for this specific task. It demonstrated the highest accuracy and F1-score coupled with robust precision and recall. This performance underscores VGG16's capability to reliably identify collision scenarios while maintaining a low rate of false positives, which is crucial for the safety-critical application of autonomous driving.

MobileNet, while slightly inferior in terms of raw performance metrics compared to VGG16, offered a balance between efficiency and effectiveness, making it a potential candidate for scenarios where computational resources are limited. DenseNet121 also performed well, presenting itself as a viable alternative to VGG16 with slightly less effectiveness but similar strengths. ResNet50, however, showed considerable

limitations in this context, notably in precision and overall accuracy, which could be attributed to model-specific characteristics or the nature of the training data.

8.2 Limitations

Although the utilization of a simulated environment such as CARLA for data generation is beneficial, it may not encompass all the intricacies of actual driving circumstances. To use this in practical situations, the model must undergo training using a dataset that has been collected from real-world scenarios.

The study employed four specific pre-trained CNN architectures for image feature extraction without modification or optimization tailored to the specific task of collision detection. The performance of these models might be enhanced significantly through customization or by employing models specifically designed for the task.

8.3 Further work

In this research, we tried to develop an ML model that is capable of giving warning to drivers about impending danger while driving, in a classification manner, safe or unsafe. In the real world, we can implement this using RGB and RGB-D cameras as the input sensor fixed on the optimal position in the vehicle and connect these inputs to the model running on an edge device.

In the current implementation, features extracted separately from RGB and RGB-D images are concatenated to form a comprehensive feature set for collision prediction. While effective, this method primarily relies on simple concatenation, which may not fully capitalize on the potential synergies between the two types of data. For further work, a more sophisticated fusion method could be explored to enhance the integration of features derived from RGB and RGB-D inputs.

As autonomous driving systems must be transparent and understandable, further work could also focus on enhancing the explainability of these models. Techniques such as visualizing feature maps and attention mechanisms could provide insights into model decisions, increasing trust and reliability.

REFERENCES

[1] World Health Organization, *Global status report on road safety 2018*. Geneva: World Health Organization, 2018. Accessed: Jun. 03, 2023. [Online]. Available: https://apps.who.int/iris/handle/10665/276462

[2] W. Bank, "Delivering Road Safety in Sri Lanka," Feb. 2020, doi: 10.1596/33341.

[3] M. Aljaban, "Analysis of Car Accidents Causes in the USA".

[4] R. Kumarage, S. M. Wickramasinghe, and M. D. R. P. Jayaratne, "Analysis Of Road Accidents in Sri Lanka." 2003.

[5] Đ. Petrović, R. Mijailović, and D. Pešić, "Traffic Accidents with Autonomous Vehicles: Type of Collisions, Manoeuvres and Errors of Conventional Vehicles' Drivers," *Transportation Research Procedia*, vol. 45, pp. 161–168, 2020, doi: 10.1016/j.trpro.2020.03.003.

[6] L. Alzubaidi *et al.*, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 1, p. 53, Mar. 2021, doi: 10.1186/s40537-021-00444-8.

[7] A. Cheok and E. Zhang, *From Turing to Transformers: A Comprehensive Review and Tutorial on the Evolution and Applications of Generative Transformer Models*. 2023. doi: 10.32388/3NTOLQ.2.

[8] Z. Halim, R. Kalsoom, S. Bashir, and G. Abbas, "Artificial intelligence techniques for driving safety and vehicle crash prediction," *Artif. Intell. Rev.*, vol. 46, no. 3, pp. 351–387, Oct. 2016, doi: 10.1007/s10462-016-9467-9.

[9] A. P. Staff *et al.*, "An Evaluation of ``Crash Prediction Networks'' (CPN) for Autonomous Driving Scenarios in CARLA Simulator".

[10] M. Haris, M. A. Moin, F. A. Rehman, and M. Farhan, "Vehicle Crash Prediction using Vision," in *2021 55th Annual Conference on Information Sciences and Systems (CISS)*, Baltimore, MD, USA: IEEE, Mar. 2021, pp. 1–5. doi: 10.1109/CISS50987.2021.9400257.

[11] A. B. Parsa, R. S. Chauhan, H. Taghipour, and S. Derrible, "Applying Deep Learning to Detect Traffic Accidents in Real Time Using Spatiotemporal Sequential Data," 2019.

[12] S. S. Yassin and Pooja, "Road accident prediction and model interpretation using a hybrid K-means and random forest algorithm approach," *SN Appl. Sci.*, vol. 2, no. 9, p. 1576, Aug. 2020, doi: 10.1007/s42452-020-3125-1.

[13] E. J. Rodríguez-Seda, "Collision Avoidance Systems, Automobiles," in *International Encyclopedia of Transportation*, Elsevier, 2021, pp. 173–179. doi: 10.1016/B978-0-08-102671-7.10121-6.

[14] D. Lord and B. N. Persaud, "Accident Prediction Models With and Without Trend: Application of the Generalized Estimating Equations Procedure," *Transportation Research Record*, vol. 1717, no. 1, pp. 102–108, Jan. 2000, doi: 10.3141/1717-13.

[15] D. A. Pomerleau, "ALVINN: An Autonomous Land Vehicle in a Neural Network," in *Advances in Neural Information Processing Systems*, Morgan-Kaufmann, 1988. Accessed: Sep. 10, 2023. [Online]. Available: https://proceedings.neurips.cc/paper/1988/hash/812b4ba287f5ee0bc9d43bbf5bbe87fb-Abstract.html

[16] F.-H. Chan, Y.-T. Chen, Y. Xiang, and M. Sun, "Anticipating Accidents in Dashcam Videos," in *Computer Vision – ACCV 2016*, vol. 10114, S.-H. Lai, V. Lepetit, K. Nishino, and Y. Sato, Eds., in Lecture Notes in Computer Science, vol.

10114. , Cham: Springer International Publishing, 2017, pp. 136–153. doi: 10.1007/978-3-319-54190-7_9.

[17] J. G. Choi, C. W. Kong, G. Kim, and S. Lim, “Car crash detection using ensemble deep learning and multimodal data from dashboard cameras,” *Expert Systems with Applications*, vol. 183, p. 115400, Nov. 2021, doi: 10.1016/j.eswa.2021.115400.

[18] H. Zhao, H. Cheng, T. Mao, and C. He, “Research on Traffic Accident Prediction Model Based on Convolutional Neural Networks in VANET,” in *2019 2nd International Conference on Artificial Intelligence and Big Data (ICAIBD)*, Chengdu, China: IEEE, May 2019, pp. 79–84. doi: 10.1109/ICAIBD.2019.8837020.

[19] S. Hochreiter, “LONG SHORT-TERM MEMORY,” presented at the Neural Computation, Nov. 1997, pp. 1735–1780. doi: https://doi.org/10.1162/neco.1997.9.8.1735.

[20] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” arXiv, Dec. 11, 2014. doi: 10.48550/arXiv.1412.3555.

[21] Y. Chen, C. Dong, P. Palanisamy, P. Mudalige, K. Muelling, and J. M. Dolan, “Attention-Based Hierarchical Deep Reinforcement Learning for Lane Change Behaviors in Autonomous Driving,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Long Beach, CA, USA: IEEE, Jun. 2019, pp. 1326–1334. doi: 10.1109/CVPRW.2019.00172.

[22] F. Marchetti, F. Becattini, L. Seidenari, and A. D. Bimbo, “Multimodal trajectory prediction in the CARLA simulator”.

[23] M. Gao, J. Jiang, G. Zou, V. John, and Z. Liu, “RGB-D-Based Object Recognition Using Multimodal Convolutional Neural Networks: A Survey,” *IEEE Access*, vol. 7, pp. 43110–43136, 2019, doi: 10.1109/ACCESS.2019.2907071.

[24] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. M. López, “Multimodal End-to-End Autonomous Driving,” *IEEE Trans. Intell. Transport. Syst.*, vol. 23, no. 1, pp. 537–547, Jan. 2022, doi: 10.1109/TITS.2020.3013234.

[25] J. Sanchez-Riera, K.-L. Hua, Y.-S. Hsiao, T. Lim, S. C. Hidayati, and W.-H. Cheng, “A comparative study of data fusion for RGB-D based visual recognition,” *Pattern Recognition Letters*, vol. 73, pp. 1–6, Apr. 2016, doi: 10.1016/j.patrec.2015.12.006.

[26] A. J. M. Muzahid, S. F. Kamarulzaman, M. A. Rahman, S. A. Murad, M. A. S. Kamal, and A. H. Alenezi, “Multiple vehicle cooperation and collision avoidance in automated vehicles: survey and an AI-enabled conceptual framework,” *Sci Rep*, vol. 13, no. 1, Art. no. 1, Jan. 2023, doi: 10.1038/s41598-022-27026-9.

[27] M. Imprialou and M. Quddus, “Crash data quality for road safety research: Current state and future directions,” *Accident Analysis & Prevention*, vol. 130, pp. 84–90, Sep. 2019, doi: 10.1016/j.aap.2017.02.022.

[28] B. Dong, X. Ma, F. Chen, and S. Chen, “Investigating the Differences of Single-Vehicle and Multivehicle Accident Probability Using Mixed Logit Model,” *Journal of Advanced Transportation*, vol. 2018, p. e2702360, Jan. 2018, doi: 10.1155/2018/2702360.

[29] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An Open Urban Driving Simulator,” in *Proceedings of the 1st Annual Conference on Robot Learning*, PMLR, Oct. 2017, pp. 1–16. Accessed: May 15, 2023. [Online]. Available: https://proceedings.mlr.press/v78/dosovitskiy17a.html

[30] M. Dupuis, M. Strobl, and H. Grezlikowski, “OpenDRIVE 2010 and Beyond – Status and Future of the de facto Standard for the Description of Road Networks,” 2010.
[31] “Sensors reference - CARLA Simulator.” Accessed: Apr. 24, 2024. [Online]. Available: https://carla.readthedocs.io/en/latest/ref_sensors/
[32] M. Jogin, Mohana, M. S. Madhulika, G. D. Divya, R. K. Meghana, and S. Apoorva, “Feature Extraction using Convolution Neural Networks (CNN) and Deep Learning,” in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, Bangalore, India: IEEE, May 2018, pp. 2319–2323. doi: 10.1109/RTEICT42901.2018.9012507.
[33] N. Shahriar, “What is Convolutional Neural Network — CNN (Deep Learning),” Medium. Accessed: Apr. 23, 2024. [Online]. Available: https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bdd82d5
[34] I. S. Mohamed, “Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques,” 2017. doi: 10.13140/RG.2.2.30795.69926.
[35] F. Es-sabery, A. Hair, J. Qadir, B. Sainz de Abajo, B. Garcia-Zapirain, and I. De la Torre Díez, “Sentence-Level Classification Using Parallel Fuzzy Deep Learning Classifier,” *IEEE Access*, vol. PP, pp. 1–1, Jan. 2021, doi: 10.1109/ACCESS.2021.3053917.
[36] K. Team, “Keras documentation: Keras Applications.” Accessed: Apr. 23, 2024. [Online]. Available: https://keras.io/api/applications/
[37] K. Le, “An overview of VGG16 and NiN models,” Medium. Accessed: Apr. 24, 2024. [Online]. Available: https://lekhuyen.medium.com/an-overview-of-vgg16-and-nin-models-96e4bf398484
[38] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” arXiv, Apr. 16, 2017. Accessed: Feb. 03, 2024. [Online]. Available: http://arxiv.org/abs/1704.04861
[39] “K_04 Understanding of MobileNet - EN,” 위키독스. Accessed: Apr. 24, 2024. [Online]. Available: https://wikidocs.net/165429
[40] S. Bangar, “Resnet Architecture Explained,” Medium. Accessed: Apr. 24, 2024. [Online]. Available: https://medium.com/@siddheshb008/resnet-architecture-explained-47309ea9283d
[41] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks.” arXiv, Jan. 28, 2018. Accessed: Feb. 03, 2024. [Online]. Available: http://arxiv.org/abs/1608.06993
[42] A. Vaswani *et al.*, “Attention is All you Need”.
[43] S. Cristina, “The Transformer Model,” MachineLearningMastery.com. Accessed: Apr. 24, 2024. [Online]. Available: https://machinelearningmastery.com/the-transformer-model/

Appendix – A: Script for Importing Necessary Libraries, Setting Basic Configuration, And Connect With Carla Simulator

```
# Import necessary libraries
import os, random, time
import carla
import numpy as np
import cv2

# Basic configurations
IM_W, IM_H = (640, 480)
image_save_path = 'data/safe_data'
depth_save_path = 'data/safe_data'
seq_len = 15
number_env_vehicles = 100
num_pedestrians = 20

# Randomly select map
client = carla.Client('localhost', 2000)
client.set_timeout(5)
maps = ['Town01', 'Town04', 'Town03', 'Town10HD']
map_name = random.choice(maps)

# Create main carla objects
client = carla.Client('localhost', 2000)
client.set_timeout(5)
world = client.load_world(map_name)
blueprint_library = world.get_blueprint_library()
```

Appendix – B: Carlasession Class with Add_Vehicles Method

```
class CarlaSession:

    def __init__(self):
        self.actors = []
        self.counter = 0
        self.counter_d = 0
        self.n_seq = len(os.listdir(image_save_path))
        self.n_seq_d = len(os.listdir(depth_save_path))
        self.collision_flag = False
        self.episode_images = []
        self.track_cleanup =[]
        self.env_actors = []

    def add_vehicles(self):
        env_vehicles_bp = blueprint_library.filter('vehicle.*')
        env_vehicles_bp = [x for x in env_vehicles_bp if int(x.get_attribute('number_of_wheels')) == 4]
        unwanted_veh = ['isetta', 'carlacola', 'fusorosa', 'firetruck']
        env_vehicles_bp = [x for x in env_vehicles_bp if not any(x.id.endswith(ending) for ending in unwanted_veh)]
        spawn_points = world.get_map().get_spawn_points()
        self.env_actors = []
        for i in range(number_env_vehicles):
            vehicle_bp = random.choice(env_vehicles_bp)
            while True:
                try:
                    npc = world.spawn_actor(vehicle_bp, random.choice(spawn_points))
                    self.env_actors.append(npc)
                    break
                except:
                    pass
        for v in world.get_actors().filter('*vehicle*'):
            v.set_autopilot(True)
```

Appendix – C: Method to Change Weather Settings and Add Pedestrians

```
@staticmethod
def add_pedestrians(self):
    # Set weather in environment and add pedestrians to environment
    world.set_weather(carla.WeatherParameters.HardRainSunset)
    pedestrians = []
    spawn_points = world.get_map().get_spawn_points()
    random.shuffle(spawn_points)
    pedestrian_bp = random.choice(world.get_blueprint_library().filter('walker.*'))

    for i in range(num_pedestrians):
        transform = spawn_points[i]
        pedestrian = world.try_spawn_actor(pedestrian_bp, transform)
        if pedestrian is not None:
            pedestrians.append(pedestrian)

    for pedestrian in pedestrians:
        pedestrian_controller = carla.WalkerControl()
        pedestrian_controller.speed = 1.0
        pedestrian_controller.direction.y = 1
        try:
            pedestrian.apply_control(pedestrian_controller)
        except AttributeError:
            pass
```

Appendix – D: Method to Spawn Ego-Vehicle and Attach Sensors to It

```
def add_actors(self):
    # Setup our ego vehicle to collect data
    start_points = world.get_map().get_spawn_points()
    # set vehicle
    vehicle_bp = blueprint_library.find('vehicle.ford.mustang')
    while True:
        try:
            self.vehicle = world.spawn_actor(vehicle_bp, random.choice(start_points))
            break
        except:
            pass

    # Setup collision sensor, RGB and RGB-D cameras
    collision_sensor_bp = blueprint_library.find('sensor.other.collision')
    camera_sensor_bp = blueprint_library.find('sensor.camera.rgb')
    camera_sensor_bp.set_attribute('image_size_x', str(IM_W))
    camera_sensor_bp.set_attribute('image_size_y', str(IM_H))
    camera_sensor_bp.set_attribute('shutter_speed', '200')
    camera_sensor_bp.set_attribute('fov', str(100))

    depth_camera_bp = blueprint_library.find('sensor.camera.depth')
    depth_camera_bp.set_attribute('image_size_x', str(IM_W))
    depth_camera_bp.set_attribute('image_size_y', str(IM_H))
    depth_camera_bp.set_attribute('fov', str(100))

    # Attach created RGB and RGB-D cameras to ego-vehicle
    sensor_location = carla.Transform(carla.Location(x=4, y=0, z=2.5))
    self.camera = world.spawn_actor(camera_sensor_bp, sensor_location, attach_to=self.vehicle)
    self.depth_camera = world.spawn_actor(depth_camera_bp, sensor_location, attach_to=self.vehicle)
    self.collision_sensor = world.spawn_actor(collision_sensor_bp, sensor_location, attach_to=self.vehicle)
    self.actors.extend([self.vehicle, self.camera, self.collision_sensor, self.depth_camera])
    self.camera.listen(lambda image: self.add_image(image))
    self.depth_camera.listen(self.process_image)
    self.collision_sensor.listen(lambda collision: self.end_seq(collision, 'collision'))
```

Appendix – E: Method to Trigger Crash

```
@staticmethod
def get_directions(self):
    thr = random.choice([0.8,0.7,0.6])
    steer = random.choice([-0.3,0.0,0.0,0.0,0.3,0.1,-0.1])
    return carla.VehicleControl(thr,steer)
```

Appendix – F: Methods to Run Simulation and Collet RGB And RGB-D Images

```
def start_new_seq(self):

    self.add_actors()
    self.collision_flag = False
    self.counter = 0
    self.counter_d = 0
    self.n_seq += 1
    self.n_seq_d += 1
    self.track_cleanup.append(self.n_seq)
    if not os.path.exists(os.path.join(image_save_path, str(self.n_seq))):
        os.makedirs(os.path.join(image_save_path, str(self.n_seq)))

    if not os.path.exists(os.path.join(depth_save_path, str(self.n_seq_d))):
        os.makedirs(os.path.join(depth_save_path, str(self.n_seq_d)))

def add_image(self, image):
    self.counter += 1
    img = np.reshape(image.raw_data, (IM_H, IM_W, 4))
    img = img[:, :, :3][:]

    cv2.imwrite(os.path.join(image_save_path, str(self.n_seq), '{}.png'.format(self.counter)), img)

def process_image(self, image):
    # Convert the raw data into an array
    self.counter_d += 1
    array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
    array = np.reshape(array, (image.height, image.width, 4))

    cv2.imwrite(os.path.join(depth_save_path, str(self.n_seq_d), '{}.png'.format(self.counter_d)), array)
```

Appendix – G: RGB and RGB-D Image Preprocessing

```
def image_preprocessor(rgb_path: str, rgb_images: List[str], depth_path: str, depth_images: List(str)):
    rgb_list = []
    depth_list = []

    for rgb_image in rgb_images:
        rgb_full_path = os.path.join(rgb_path, rgb_image)
        rgb_image = cv2.imread(rgb_full_path)
        rgb_image = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2RGB)  # Convert to RGB
        rgb_image = cv2.resize(rgb_image, (224, 224))  # Resize to match model input size
        rgb_image = rgb_image.astype('float32') / 255.0
        data = np.expand_dims(rgb_image, axis=0)
        rgb_list.append(data)

    for depth_image in depth_images:
        depth_full_path = os.path.join(depth_path, depth_image)
        rgb_imaged = cv2.imread(depth_full_path)
        rgb_imaged = cv2.cvtColor(rgb_imaged, cv2.COLOR_BGR2RGB)  # Convert to RGB
        rgb_imaged = cv2.resize(rgb_imaged, (224, 224))  # Resize to match model input size
        rgbd_image = rgb_imaged.astype('float32') / 255.0
        rgbd_image = np.expand_dims(rgbd_image, axis=0)
        depth_list.append(rgbd_image)

    return rgb_list, depth_list
```

Appendix – H: Importing Necessary Libraries and Defining CNN Model

```
import tensorflow as tf
from tensorflow.keras.applications import VGG16, MobileNet, ResNet50, DenseNet121
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Concatenate, GlobalAveragePooling2D, Input

NUM_FEATURES = 1024

# Define function to create pre-trained CNN models for RGB and RGB-D images
def create_model(input_shape, name=None):
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)
    base_model.trainable = False

    inputs = Input(shape=input_shape)
    x = base_model(inputs)
    outputs = GlobalAveragePooling2D()(x)

    return Model(inputs=inputs, outputs=outputs, name=name)
```

Appendix – I: Script for Creating Extracted Feature Array

```
def create_features_array(rgb_images, rgbd_images):
    MAX_SEQ_LENGTH = 8
    NUM_FEATURES = 1024

    num_samples = len(rgb_images)

    features = np.zeros(
        shape=(num_samples, MAX_SEQ_LENGTH, NUM_FEATURES), dtype="float32"
    )
    result_array = np.zeros((8, NUM_FEATURES))

    # Create CNN models for RGB and RGB-D images
    rgb_shape, rgbd_shape = (224, 224, 3), (224, 224, 3)
    model_rgb = create_vgg_model(rgb_shape, name='vgg16_rgb')
    model_rgbd = create_vgg_model(rgbd_shape, name='vgg16_rgbd')
    # Create input layers for RGB and RGB-D images
    input_rgb = Input(shape=rgb_shape)
    input_rgbd = Input(shape=rgbd_shape)

    # Extract features from RGB and RGB-D images using respective models
    features_rgb = model_rgb(input_rgb)
    features_rgbd = model_rgbd(input_rgbd)

    # Concatenate the features from both networks
    concatenated_features = Concatenate()([features_rgb, features_rgbd])

    # Define a new model to get the fused features and reshape to (1, 512)
    fused_model = Model(inputs=[input_rgb, input_rgbd], outputs=concatenated_features)
    output_shape = (1, NUM_FEATURES)

    for idx, (r, d) in enumerate(zip(rgb_images,rgbd_images)):
            fused_features = fused_model.predict([r, d],verbose=0)
            fused_features_reshaped = tf.reshape(fused_features, output_shape)
            result_array[idx] = fused_features_reshaped.numpy()

    features[index,] = result_array
    return features
```

Appendix – J: Script for Transformer Positional Embedding

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim
        )
        self.sequence_length = sequence_length
        self.output_dim = output_dim

    def call(self, inputs):
        # The inputs are of shape: `(batch_size, frames, num_features)`
        length = tf.shape(inputs)[1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_positions = self.position_embeddings(positions)
        return inputs + embedded_positions

    def compute_mask(self, inputs, mask=None):
        mask = tf.reduce_any(tf.cast(inputs, "bool"), axis=-1)
        return mask
```

Appendix – K: Script for Transformer Encoder

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim, dropout=0.3
        )
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation=tf.nn.gelu), layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]

        attention_output = self.attention(inputs, inputs, attention_mask=mask)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)
```

Appendix – L: Utility Functions for Training

```
def get_compiled_model():
    sequence_length = MAX_SEQ_LENGTH
    embed_dim = NUM_FEATURES
    dense_dim = 2
    num_heads = 10
    classes = len(label_processor.get_vocabulary())
    inputs = keras.Input(shape=(None, None))
    x = PositionalEmbedding(sequence_length, embed_dim, name="frame_position_embedding")(inputs)
    x = TransformerEncoder(embed_dim, dense_dim, num_heads, name="transformer_layer")(x)
    x = layers.GlobalMaxPooling1D()(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(classes, activation="softmax")(x)
    model = keras.Model(inputs, outputs)

    model.compile(
        optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"]
    )
    return model
def run_experiment():
    filepath = "./tmp/classifier"
    checkpoint = keras.callbacks.ModelCheckpoint(
        filepath, save_weights_only=True, save_best_only=True, verbose=1
    )
    model = get_compiled_model()
    history = model.fit(
        train_data,
        train_labels,
        batch_size= BATCH_SIZE,
        validation_split=0.15,
        epochs=EPOCHS,
        callbacks=[checkpoint,tensorboard_callback],
    )
    model.load_weights(filepath)
    _, accuracy = model.evaluate(test_data, test_labels)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    tf.keras.backend.clear_session()
    return model, history
```

Appendix – M: Script for Creating a Demo Video

```
def inference(model, paths, path_d):
    image_folder = os.listdir(paths)
    image_folder_d = os.listdir(path_d)
    image_folder.sort(key=lambda x: int(re.sub('\D','',x)))
    image_folder_d.sort(key=lambda x: int(re.sub('\D','',x)))
    image_seq_rgb = deque([],8)
    image_seq_rgb_d = deque([],8)
    counter = 0
    stat = 'safe'
    for image, image_d in zip(image_folder, image_folder_d):
        try:
            _frame = cv2.imread(os.path.join(paths, image))
            _frame_d = cv2.imread(os.path.join(path_d, image_d))
            image_seq_rgb.append(_frame)
            image_seq_rgb_d.append(_frame_d)
            if counter%2 == 0:
                if len(image_seq_rgb)==8:
                    extracted_features = prepare_all_videos(image_seq_rgb, image_seq_rgb_d)
                    r = model.predict(extracted_features)
                    stat = ['safe', 'unsafe'][np.argmax(r,1)[0]]

            cv2.putText(_frame,stat, (230,230), cv2.FONT_HERSHEY_SIMPLEX, 3, (0,255,0),3)
            out.write(_frame)
            counter+=1
            print (counter)
        except Exception as e:
            print(e)
    out.release()
    cv2.destroyAllWindows()
```