Tampere University

Tri Thong Nguyen

DRIVER DISTRACTION ESTIMATOR

Real-time Embedded Implementation of Head Pose Classification and Warning System

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
October 2022

ABSTRACT

Tri Thong Nguyen: Driver Distraction Estimator
M. Sc. Thesis
Tampere University
Master's Degree Programme in Information Technology, Embedded Systems
October 2022

Nowadays, the necessity to attend to traffic is increasing tremendously, and as a consequence, the number of collisions is also climbing substantially. These are mainly caused by human error while driving, including texting, talking on the phone, eating and drinking, and so on. Drivers using cellular phones are nearly fourfold more likely to be involved in an accident than drivers who do not use a cell phone. In 2020, 3142 individuals were killed in automobile accidents caused by distracted drivers in the US. Distracted accidents are the main concern that this work seeks a solution to address.

In automotive systems, numerous methods to address driver distraction have been developed, including the use of biological testing, AI-based prediction, and image processing. However, the majority of these approaches were developed in laboratory settings that may not be ideal for actual automobile cabins. Some of the techniques are insufficient for use in automotive contexts due to platform constraints, or biological testing cannot adapt to real-time needs. Image processing may adapt to the two mentioned requirements, but the accuracy and flexibility may not be achievable.

This thesis presents an end-to-end system that combines two types of models - the Coral AI model and the Decision Tree - to certainly assist tackle this challenge. The outcomes of this system are the driver's status, and an external module is in charge of advising the driver to drive carefully.

This thesis focuses on a real-time AI-based monitoring system that categorizes a driver's distraction level. The system is separated into three major stages: head pose detection, distraction estimation, and warning module. The key joints in the top half of the human body, such as the head, neck, and shoulders, are extracted using pose detection., and the joints are used as input parameters by the distraction estimator to predict the status of the drivers. Finally, the warning assesses the situation and, if necessary, activates the warning system to alert the driver.

In a nutshell, the key contributions of this study included implementing the classifier model and performing it in an embedded environment. The PoseNet model was chosen as the pose estimator; it operates on Coral's Edge TPU and can deliver results in 2.6 ms. To identify the distraction, a fine-tuned decision tree is used. It can work on the Nvidia Jetson Nano, which is the primary environment for running the classified task. Also, the external warning module is constructed on a Raspberry Pi 2 and it interacts with the Jetson through Socket protocols to receive reports and notify the driver. The system performance can achieve from 15 to 20 frame rate, and the decision tree's prediction accuracy is more than 90%.

Keywords: AI application, Distracted Estimator, Embedded System, Real-time

PREFACE

My deepest gratitude goes to my supervisor and examiner, Jarno Vanne, for advice on this thesis and for allowing me to be a part of this work. Also, I would like to thank Marko Viitanen, Tero Partanen, and Teo Niemirepo for accompanying me with helpful advice and recommendations during my research.

Lastly, I want to thank my family for always being there for me. I am grateful to my late father for his unending love, support, and encouragement. This thesis was the last thing he wanted me to finish, and I hope the outcome would bring him joy in paradise. Also, special thanks to my mother, who has always believed in me and supported my decisions. This thesis is dedicated to both of my parents, without whom I would not be the person I am today.

Tampere, 31st October 2022

Tri Thong Nguyen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

3D	Three dimension
AI	Artifact intelligent
API	Application programming interface
ASIC	Application Specific Integrated Circuit
BSD	Berkeley Software Distribution
C4.5	A decision-tree-based algorithm created by Ross Quinlan.
CAN	Controller area network communication
CART	Classification and Regression Tree
CHAID	Chi-square automatic interaction detection
CNN	Convolutional neural network
CPU	entral processing unit
CUDA	Compute Unified Device Architecture
DSC	Depthwise separable convolutions
ECUs	Electronic control units
FIFO	First-In-First-Out queue
FN	False negative
FP	False positive
FPS	Frames per second
GPU	Graphics Processing Unit
ID3	Iterative Dichotomiser 3
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
mAP	mean average precision
MAR	Missing at Random
MARS	Multivariate adaptive regression splines
MCAR	Missing Completely At Random

MICA	micathermic
MNAR	Graphics Processing Unit
OLED	organic light-emitting diode
RAM	Random-access memory
RGB	Red, Green, and Blue; the color space for an image
SDK	Software development kit
SVM	Support-Vector Machine
TensorFlow	An end-to-end machine learning platform
TensorRT	A C++ library for high-performance inference on NVIDIA GPUs and deep learning accelerators.
TN	True negative
TOPS	trillion operations per second
TP	True positive
TPU	Tensor Processing Unit
UFF	Universal File Format
UI	user interface
USB	Universal Serial Bus

1. INTRODUCTION

In this modern world, traffic accidents are becoming more common, and most of them result in serious injuries or fatalities. According to studies, distracted drivers cause the majority of accidents in the US. These are mainly caused by human error while driving, including texting, talking on the phone, eating and drinking, and so on. The number of accidents caused by mobile phone use while driving is four times that of the other causes [1]. Based on the survey of the National Highway Transportation and Safety Administration, in 2020, 3 142 individuals were killed in fatal distracted driving accidents in the US, accounting for 8.7% of all collision deaths that year. In addition, roughly 280 000 people are wounded each year as a result of driving distractions [2]. The numbers show how disastrous the consequences may be when we drive while distracted.

A traditional vehicle was mostly composed of mechanical components until the 1960s. In the 1970s, the integration of electronic components began to improve. Modern automobiles may have 100 or more electronic control units (ECUs), which manage tasks ranging such as fundamental, comfort, security, and access tasks (such as door locks and keyless entry) [3]. The ECUs are connected with others by controller area network (CAN) communication [4]. They are specifically developed to satisfy the demands of the automotive industry and communicate between modules. Today, electronic technologies have mostly supplanted mechanical ones in autos. The automotive sector makes extensive use of embedded systems. Embedded systems have taken over the control of vehicles, from wiper controls to complicated anti-lock braking controls and airbags, so the techniques can secure safety for drivers and travelers. However, these solutions only have use in decreasing the damage to drivers, passengers, and cars. To address the problems caused by distracted behaviors, several high-technology modules have been added to vehicles to support the drivers, namely artificial intelligence (AI) or deep learning.

Since 2003, Tesla has been a leader in automotive AI adoption. One of the company's most notable developments is an AI-powered inside camera mounted above the rearview mirror to increase cabin safety. The camera identifies and analyzes drivers' eyes using AI innovation to identify tiredness and avert on-road accidents [5]. However, if only drowsiness is detected, that would be a massive loss since many activities can be classified as "distracted driving." Therefore, it's critical to understand the distinction between a legal distraction and a personal distraction before getting behind the wheel. Common distract-

ing behaviors include applying makeup, driving while texting, or having a lengthy phone conversation and turning to grab items from the back seat. Therefore, all of this behavior has to be taken into account. This thesis focuses on detecting the status: safe or distracted, as well as providing real-time notification to the driver directly in the cabin. This can be helpful in terms of addressing the need to reduce traffic accidents.

The designed system is separated into three major stages: head pose detection, distraction estimation, and warning interaction. To be more exact, the head pose detection is developed in a single-board NVIDIA Jetson Nano capable of performing AI tasks, with a Coral USB Accelerator [6] included to boost performance. Real-time needs could be successfully adapted thanks to the accelerator. The head poses estimator is a ready-made model from Coral [7] to extract the human stance and a decision tree to determine the driver's state. whereas the distraction estimation module is the main target of this thesis, it is a single decision tree to recognize the status of the drivers via inputs of the head pose detection. Raspberry Pi 2B main board is in charge of collecting status information and activating the buzzer to alert drivers. The Ethernet protocol is used to link the two main boards. Because these two boards lack CAN connection, Ethernet is employed as a replacement as well as an alternate means of connecting with other projects.

The thesis is organized as follows. Chapter 1 presents the background information regarding the study field. In Chapter 2, the basic principles of the work as well as several practical implications for the development of driver distraction detection systems are introduced. Chapter 3 discusses the target model such as PoseNet and Decision Tree and facts related to their benefits. In addition, the algorithms of both models, are completely covered. In Chapter 4, the evaluation approaches are described, which includes the presentation of all relevant material for the thesis and the research technique. Chapter 5 introduces all relevant background material concerning the implementation and experiments. Then, the testing outcomes and comparison with other works are then fully examined in Chapter 6. Finally, chapter 7 offers an overview of the system as well as a discussion of future work.

2. DRIVER MONITORING SYSTEM

2.1 Basic Principles

Driver behavior is commonly classified as either intentional or unintentional. The intended activity of the driver is an adaptation of his or her mental thinking, and it could be utilized to deduce the driver's inner mental state. Normal driving behaviors include head movement, eye-gaze mechanics, hand motions and gestures, body movement, and foot patterns. Driving tiredness, attention, and focus have all been successfully predicted using these driver behavior characteristics. As a result, recognizing the driver's distraction is critical in regular driving scenarios. Simple driver distraction monitoring comprises the following components in general.

First, a feature selection and assessment-based driving behavior need to be conducted, and pass the feature to the recognition system. Then, the driver's head and body joint positions can be now monitored. Several previous studies investigated the impact of driver body joints on distraction detection. Individually, the system evaluates the driver's head attitude, body posture, arm position, and depth information. Driver body position attributes are found to help with more accurate driver status classification.

When the driver is driving, the system determines the driver's status and the information of the driver sends to a warning module, which is used to inform distracted drivers. This module is important for communication between the proposed system and the real world. It's also worth considering the integration of the driver behavior detection system with its working environment. For example, how the system will interact with the other subsystems of the vehicle such as the Advanced Driver Assistance Systems (ADAS) [8], and how the system will work together with the existing embedded system in the vehicle to convey information to the driver.

2.2 Existing Driver Distraction Systems

Driver distraction detection entails initially gathering and then comprehending information about the behavior of drivers and items in the car cabin. As a result, there are three major types of distraction: physical distraction, cognitive distraction, and visual distraction. These three forms of driving distractions can occur independently, or simultaneously [9].

2.2.1 Manual Distraction

Manual distraction is driving behaviors using the hands and feet, such as feet from the pedals or removing hands from the steering wheel. In other words, whenever the driver becomes distracted, the pace and direction of the vehicle can be changed instantaneously. As a result, systems for tracking lanes or detecting velocity can be designated as a remedy for dealing with this form of distraction.

Hand detection is one of the methods for dealing with this manual distraction. According to the experiment of Cuong Tran and Mohan Manubhai Trivedi [10], There are three states of hands when driving vehicles: Two, one, or no hand on the steering wheel. Their technology presents a method for tracking hand position in order to identify how many hands are currently holding the steering wheel, which is an indicator of vehicle readiness. The movement of hand blobs may indicate whether the driver is at rest or performing certain tasks (which may or may not relate to the driving task). Because the driver normally sits in a fixed position, head blob tracking approximates the sitting posture, which is critical for smart airbag activation, for example.

Another solution to address the issue of manual distraction while driving is to classify hand actions. Yan et al. [11] developed a system that utilizes Convolutional Neural Networks (CNN) - a type of neural network architecture used primarily for image and video recognition tasks. The CNN-based system identifies driver postures by analyzing hand position data, enabling it to recognize when a driver is engaging in activities such as eating, smoking, or using a phone. By detecting these behaviors, the system can alert the driver to the potential danger and remind them to focus on the road, significantly reducing the risk of accidents caused by distracted driving. The use of CNN-based systems for recognizing driver postures has the potential to improve road safety by promoting responsible driving habits and reducing the number of accidents caused by manual distractions. Overall, this technology can help address the issue of manual distraction and contribute to a safer driving experience for all.

2.2.2 Cognitive Distraction

Cognitive distraction happens when a driver's thoughts are diverted from the task of driving, such as while conversing, experiencing strong emotions, feeling anxious, or daydreaming. In other words, mental issues produce cognitive distraction. As a result, the built-in system or computer must communicate with the human brain in order to evaluate if the driver is distracted. To evaluate the driver's state, biological tests such as electroencephalography (EEG) and electrocardiogram (ECG) are employed. Also, the system can be enhanced by integrating the driver's physiological data (heart rate, breathing rate, etc.) to detect the driver's stress level, fatigue, and drowsiness, which can be correlated to

distraction. However, cognitive distraction must be performed in real-time and without intrusion; subjective report measurements and driver biological measures are not adequate for a real-world context. [12].

Eye movements are the most often utilized driver physical evidence for driver cognitive distraction. [13], [14]. Azman et al. [15] discovered that when a person is thinking or intellectually distracted, his or her lips and eyes are associated and that they may be utilized to identify a driver's cognitive distraction. According to Victor and Trent, cognitive distraction forces drivers to focus their attention in the center of the driving scene, as characterized by the vertical and horizontal standard deviation of gaze distribution and reduces drivers' ability to recognize targets throughout the full driving scene. [16]. In a nutshell, visual distraction detection algorithms could be used to identify cognitive distraction whereas obtaining information such as eye gaze direction, eye closure, and blink detection, as well as head position.

2.2.3 Visual Distraction

Visual distraction refers to behaviors that cause the driver to lose sight of the road or the surrounding environment. Examples of visual distraction include shutting one's eyes or failing to keep their gaze on the road ahead. The purpose is to identify the direction in which an eye is directed because it is a significant indicator of a driver's field of view and present focus of attention. One of the most common ways is to track drowsiness drivers. In this approach, the state of the eyes and mouth will be tracked to determine whether the driver falls asleep during driving. The four factors to monitor drowsiness detection include Eye State Analysis, Eye Blinking Analysis, Mouth, and Yawning Analysis, and Facial Expression Analysis.

To begin with the eyes, in 2010, Amol M. Malla et al developed an affordable system in 2010 that uses a camera and near-infrared (NIR) illumination to observe the subject's facial indicators of drowsiness and Behavioral microsleeps without being intrusive, even in varied lighting conditions. They utilized signal processing and computer vision techniques to identify the location of the face, eyes, and eyelids within a video frame. The positions of the eyelids were then analyzed to determine the proportion of eyelid closure, which can serve as a measure for detecting drowsiness and Behavioral microsleeps [17].

The use of two cameras that operate in the visible and near-infrared spectra, respectively, has enabled safe driving during both day and night conditions. Each spectrum undergoes image processing through a series of two classifiers. The first classifier detects eye regions using specific eye models for each spectrum. The second classifier verifies the eyes based on the higher-order singular value decomposition of geometrically deformed versions of real eye prototypes, specific to each spectrum. [18]. Moreover, there are other solutions that detect distraction based on eyes like matching eyes template [19],

and tracking eye expressions [20].

The mentioned approaches mainly focus on gaze estimators and drowsiness. They are based on image processing, which can ensure a fast speed for real-time requirements. However, other solutions apply deep learning to discover the distracted status. In 2018, Bhakti Baheti et al. propose a detecting approach using a Convolutional Neural Network [21]. The aim of this system is to utilize the advantage of the Convolutional Neural Network to recognize the distracted driver, and also determine the reason for the distraction. This is also a classifier model since it has the ability to recognize seven activities of the driver, such as texting, talking, and drinking.

2.3 Head Pose Estimation

Nowadays, numerous computer vision sectors, including the interaction of human and robot, data analysis, and the automotive environment, benefit from head pose estimation.

Few studies have integrated Convolutional Neural Networks (CNN) and depth maps to directly estimate head poses from images: the authors in [22], [23] suggested utilizing a CNN with captured depth images from a Microsoft Azure Kinect camera. Other studies that utilize CNNs and 3D data focus on various tasks, such as estimating skeleton body poses [24], action detection [25], object pose detection [26], and human body joint estimation [27].

Fanelli et al. [28] suggested a real-time system based on Random Regression Forests for detecting the head orientation and position in in-depth pictures. Also, there are other approaches to optimize the head pose estimation task: in [29], Depth images are subjected to Particle Swarm Optimization and least-square minimization techniques. S. Malassiotis and M. G. Strintzis developed a method for detecting head poses using low-quality depth data [30]. RGB pictures with extremely low resolution are employed in [31] and despite the low input quality, outcomes that are near to state-of-the-art are obtained. [32] The problem of significant variations in head pose and partial occlusions is tackled by this method, which utilizes geometric features and nose tip identification to extract the head position in real time. This is achieved with the help of a specialized graphics unit. The angles of head rotation are predicted using the Histograms of Gradients (HOG) feature, which is derived from RGB images, in [33]. CNN and RGB pictures from a monocular camera are merged in [34]; Even with variations in lighting within input images, a deep architecture is used to regress with high accuracy. However, deep learning algorithms typically require a large amount of annotated data, and the use of synthetic RGB datasets is becoming more prevalent. [35].

The head posture alone gives a rough indicator of gaze even though eyes are not visible. As a result, there are systems that use position detection to evaluate the driver's

tiredness. For example, Erik Murphy-Chutorian et al. have investigated the driver's head movement [36]. They assess driver distraction using extracted head position information from a localized gradient histogram, eye-gaze information, and support vector regressors (SVRs). The system can perform quick face detection in a grayscale video by employing three cascaded-Adaboost face detectors [37]. Then, to identify the face region, a Localized Gradient Orientation (LGO) histogram is constructed to scale into a given size. These histograms serve as the dataset for training the two Support Vector Regressors (SVRs) that estimate head pitch and yaw. The drowsiness of the driver can be recognized by combining pitch and yaw.

3. HEAD POSE ESTIMATION AND CLASSIFICATION IN EMBEDDED ENVIRONMENT

The main object of this work is to design and implement a system equipped in front of the driver to track whether the driver is in a distracted situation as the main target. The system is made up of the head pose estimation and the classification Decision Tree. The head pose estimation is a ready-made model called PoseNet which is used to transfer the pose of the driver to the decision tree to determine the driver's status. This Chapter provides information about the two mentioned models. In section 3.1, the basic details as well as the structure of PoseNet are discussed. Next, section 3.2 introduces the status determination of the decision tree. Finally, section 3.3 gives information about the combination of these two models.

3.1 Head Pose Estimation

Pose estimation is a computer vision technique that identifies the human head, neck, eyes, or joints between the arms and shoulders in images or video frames [38], which means that the system would estimate the driver's upper half of the human body from the fixed camera in the cabin. This method uses the PoseNet model, which has the ability to locate important joints in the human body rather than to identify the individual in an image.

3.1.1 PoseNet

PoseNet uses a convolutional neural network (CNN) to estimate a human's stance from an input RGB image. In this work, the model is deployed using a MobileNet V1 architecture. The primary aim is to create a series of heatmaps and offset maps of the human body components rather than classifying. Furthermore, quantized and optimized PoseNet models are required for Coral's Edge TPU to operate well.

MobileNet V1 Architecture

The development of computer vision in the modern era has led to the creation of numerous models with varying degrees of accuracy. However, the resources of these devices

play a crucial role in choosing the best model for each type of device when developing AI applications for mobile devices such as smartphones, IoT devices, embedded computers, etc. The most popular models that can be used in this industry need to be highly accurate and computationally cheap. The model that meets the aforementioned characteristics is called MobileNet.

According to [39], the main component of MobileNet V1 architecture is the convolution layer. It identifies the output of neurons linked to local sections of the input by calculating the scalar product of their weights and the region related to the input volume [40]. Generally, the architecture consists of 30 layers with different characteristics, for example:

- Layer 1: Convolution layer with stride is equal to 2.
- Layer 2: Depthwise layer with stride is equal to 1.
- Layer 3: Pointwise layer.
- Layer 4: Depthwise layer with stride is equal to 2.
- Layer 30: Softmax.

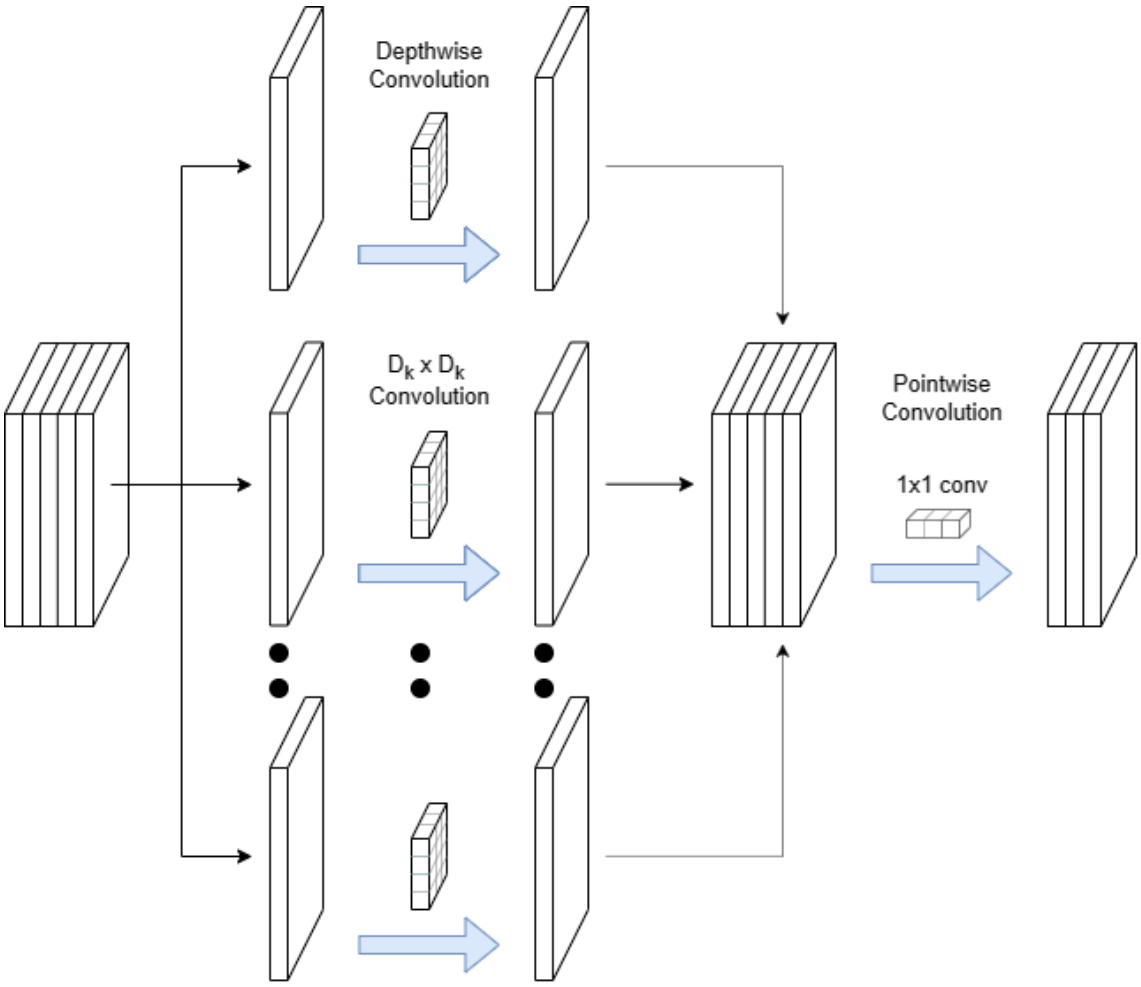


Figure 3.1. *The overview structure of a depthwise separable convolution.*

Overall, the depthwise separable convolutions (DSC) architecture is used to describe the

MobileNet model. DSC is used to minimize the amount and complexity of computations. It is therefore appropriate for use on embedded and mobile devices. Basically, figure 3.1 illustrates a depthwise convolution followed by a convolution, called a pointwise convolution, to create a depthwise separable convolution [41]. Each input channel for MobileNet receives a single k x k filter thanks to depthwise convolution. The pointwise convolution, which is a 1x1 convolution, then combines the outputs of the depthwise convolution. A conventional convolution is used to filter and produce a new aggregate of outputs from the inputs in a single step. This approach decreases computation and model size. [39]

Although MobileNet's performance may be viewed as adapting to applications on mobile devices, there are some specific circumstances, such as real-time requirements, where two extra factors, alpha, and rho, are added to the calculation cost of DSC. Also, the width multiplier parameter would affect the performance of the model. The main use of this is to regulate the number of channels, whereas the resolution multiplier parameter is used to change the resolution of the input pictures. [39]

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6 %	569	4.2
0.75 MobileNet-224	68.4 %	325	2.6
0.5 MobileNet-224	63.7 %	149	1.3
0.25 MobileNet-224	50.6 %	41	0.3

Table 3.1. *MobileNet width multiplier [39].*

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6 %	569	4.2
0.75 MobileNet-224	69.1 %	418	4.2
0.5 MobileNet-224	67.2%	290	4.2
0.25 MobileNet-224	64.4%	186	4.2

Table 3.2. *MobileNet resolution [39]*

Table 3.1 makes it evident that even while the net's accuracy marginally declines, it exhibits a large drop in computations when alpha is equal to 0.75 or 0.5. However, if alpha is 0.25, the model's accuracy, which fell to 50.6 % from 70 %, cannot be considered satisfactory. Furthermore, according to table 3.2, changing the value of the rho parameter simply affects the number of calculations; the parameter's actual quantity is unaffected.

Because the resolution of the photos is fairly low, there may not be specific features of objects in the images, which causes the accuracy percentage of ImageNet to fall. As a result, selecting the value of the alpha and rho parameters is crucial, and this technique benefits the model to run in the embedded environment.

3.1.2 Algorithm

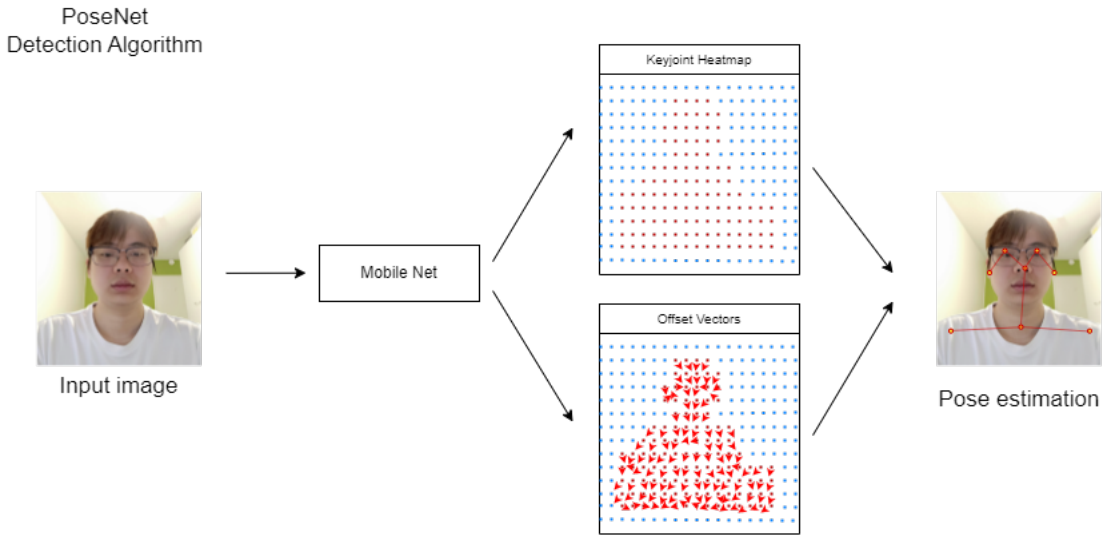


Figure 3.2. *Single person pose detector pipeline using PoseNet.*

Pose estimation is a computer vision technique used to detect human figures in photos and videos. It enables the estimation of a person's body joint positions, such as the neck, in an image. This technology does not identify individuals in a photo, and no personal information is linked to posture detection. The model in this study is designed to focus solely on the driver and perform single-pose detection on each input frame. Therefore, this section only covers the single-pose detection algorithm.

Figure 3.2 introduce the single-pose estimation algorithm at a high level. The single-person pose detector pipeline using PoseNet begins with capturing an image or video frame using a camera. The input frame is then preprocessed to convert it into a format suitable for the PoseNet model. The PoseNet model takes the preprocessed input frame as input and produces a heatmap that shows the likelihood of each key point being present at each location in the image. The model also produces an offset vector that predicts the distance between each key point and its associated joint. The final step in the pipeline is to use the heatmap and offset vector to estimate the 2D position of each key point in the image. This is done using a technique called decoding, which involves finding the maximum value in each keypoint heatmap and using the offset vector to adjust the keypoint position.

Model Input Processing: Output Strides

As described in [42], the PoseNet model has the advantage of being picture size invariant, which means it can forecast pose positions at the same scale as the original image regardless of whether it has been downscaled. Therefore, by altering the output stride during runtime, PoseNet may be tuned to have better accuracy. The output stride controls how much the output is scaled down in comparison to the size of the input picture. It has an impact on the model outputs and layer sizes. The resolution of layers in the network and the outputs, as well as their accuracy, decrease with increasing output stride. The output stride in this implementation can be either 8, 16, or 32. In other words, an output stride of 32 has best performance but the lowest accuracy, whereas an output stride of 8 produces the highest performance but the highest accuracy. Therefore, with a value equal to 16, the performance and accuracy are considerable.

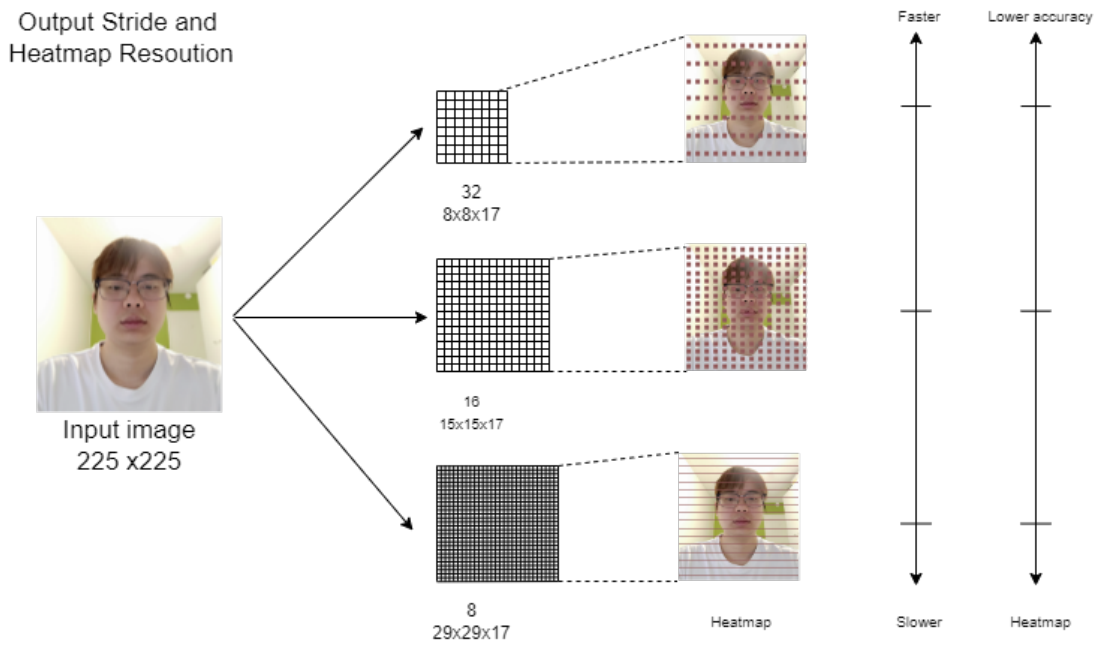


Figure 3.3. *Relationship between output stride's value and performance, accuracy.*

The amount of input striding in the layers is decreased to produce a higher output resolution when the output stride is 8 or 16. The next step is to utilize atrous convolution to give the convolution filters in the next layers a broader field of vision. Atrous convolution is helpful in term of explicitly controlling how densely to compute feature responses in fully convolutional networks. However, atrous convolution cannot be applied if the output stride is 32.

Model Outputs: Offset Vectors and Heatmaps

When PoseNet analyses an image, it returns a heatmap as well as offset vectors to locate high-confidence areas in the picture that match posture key points. These two types of outputs are 3D tensors with a resolution - height, and width. Besides, figure 3.4 below shows how each posture key point is connected with one heatmap tensor and one offset vector tensor.

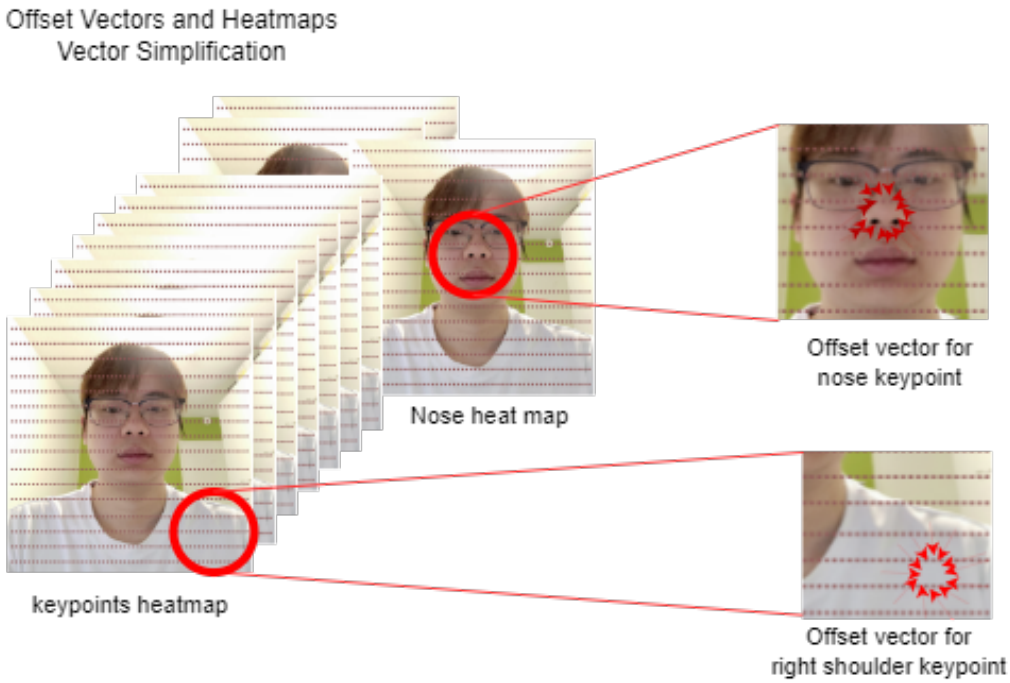


Figure 3.4. *Offset vectors and heatmaps.*

The resolution can be calculated from the input image size and the output stride as:

$$Resolution = \frac{InputImageSize - 1}{OutputStride} + 1 \tag{3.1}$$

Heatmap is a resolution x resolution x 17 3D tensors, where 17 is the number of key points observed by the PoseNet model. For instance, with a 255 pixels image and output stride of 16, the heatmap would be 15 x 15 x 17. In the third dimension, each slide represents a specific key point. Each position in the heatmap has a confidence score, which represents the likelihood that a component of that keypoint type occurs at that location. It can be considered as dividing the original image into a 15x15 grid, with the heatmap scores indicating how probable each key point is to present in each grid square.

The offset vector is also a 3D tensor with a different size: resolution x resolution x 34, where 34 is double the number of key points. Because heatmaps only give approximate locations of the key points, the offset vectors correlate to the heatmap points in location

and are used to anticipate the precise position of the key points by traveling along the vector from the corresponding heatmap point. The first 17 slices of the offset vector include the horizontal x value, whereas the final 17 contain the vertical y. The offset vector sizes are the same as the original picture scale.

Establishing Poses

After running the image through the model, a few computations are executed to estimate the posture based on the results. For example, the single-pose estimation technique outputs a pose confidence score, which comprises an array of key points (indexed by part ID). In this array, the necessary information of each keypoints like a confidence score and x, y location are stored.

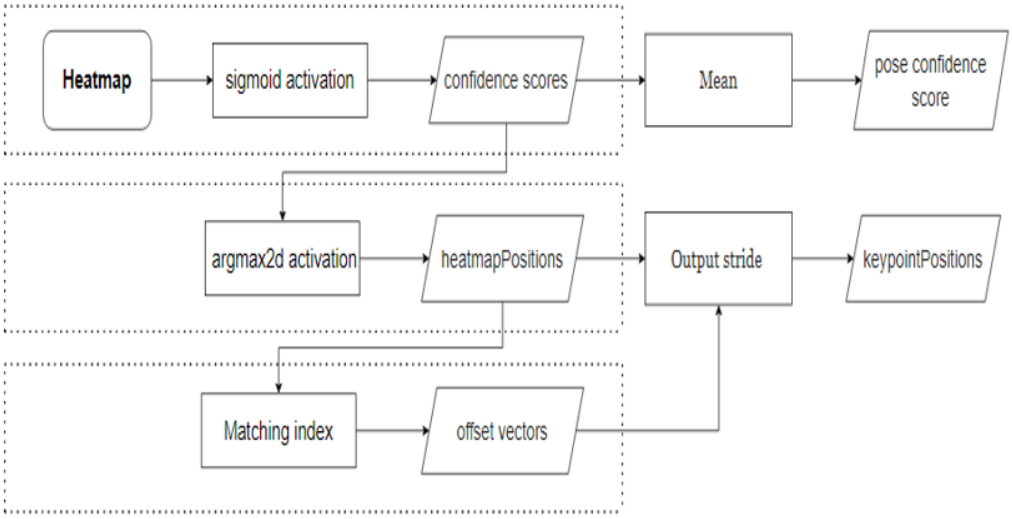


Figure 3.5. *Determining poses from model outputs.*

In ref [42], to obtain confidence scores, sigmoid activation is applied to the heatmap. Then, another layer called the argmax2d layer is applied to the keypoint confidence scores to generate the x and y index in the heatmap with the greatest score for each part, indicating where the part is most probable to appear. This produces a 17x2 tensor, with each row representing the y and x index in the heatmap with the highest score for each section. The offset vector for each body's part is obtained by collecting the x and y values from the offsets matching to the x and y indexes in the heatmap. This yields a 17x2 tensor, with each row representing the offset vector for the associated key point. In the next step, to acquire the key point, the x and y coordinates of each portion are multiplied by the output stride and then added to their corresponding offset vector, which has the same scale as the original picture. Lastly, by calculating the mean of the scores of the key points, the pose confidence score can be easily acquired.

3.2 Classification Decision Tree

Decision Trees are among the most widely used techniques for classifying data. Decision trees work well for tasks involving categorization and regression. They can easily comprehend the input data and produce the outputs by simulating the level of human thought. The difficulty of creating a decision tree from provided data has been solved by a number of specialized domains, including statistics, machine learning, and pattern recognition. Basic information on the major components that make up a decision tree is covered in this section.

3.2.1 Definition

Decision trees are a categorization method that may be thought of as a recursive division of the specific field [43]. Also, decision trees are a non-parametric approach since no assumptions are created about a set of classes. The structure or model of the tree is also unknown prior to the tree growth process [44]. Decision trees are implemented in predictive analysis, where the model is trained on a dataset and then utilized to forecast. Decision tree models must be trained on a dataset in order to learn from it. Thereafter, the models are validated against other data within the same type, which can either be from the same dataset (the data would have been separated into training and testing sets) or from a different source. This means that the model can now forecast new or unknown data and estimate which class the unknown data may belong to.

This type of classifying method is built based on the "tree" concept:

- Root node is the first node in decision trees, the essential component of a rooted tree. A node, that has no incoming edges, can be considered as the base point of the model [43].
- Internal node, also known as a decision node, is a node with outgoing edges. Each decision node divides the instance space into two or more sub-spaces [43]. This node normally connects to other internal nodes or leaf nodes, and they form the attribution of the model tree.
- Leaf node, also known as a terminal node. A class that represents a suitable target value assigned to a single leaf. Each leaf has a probability vector containing a certain value. The input data will be sorted as it is processed from root to leaf depending on testing results.

3.2.2 Working Principle

With the help of a given dataset, algorithms known as decision tree inducers may automatically build a decision tree. The usual objective is to minimize the generalization error

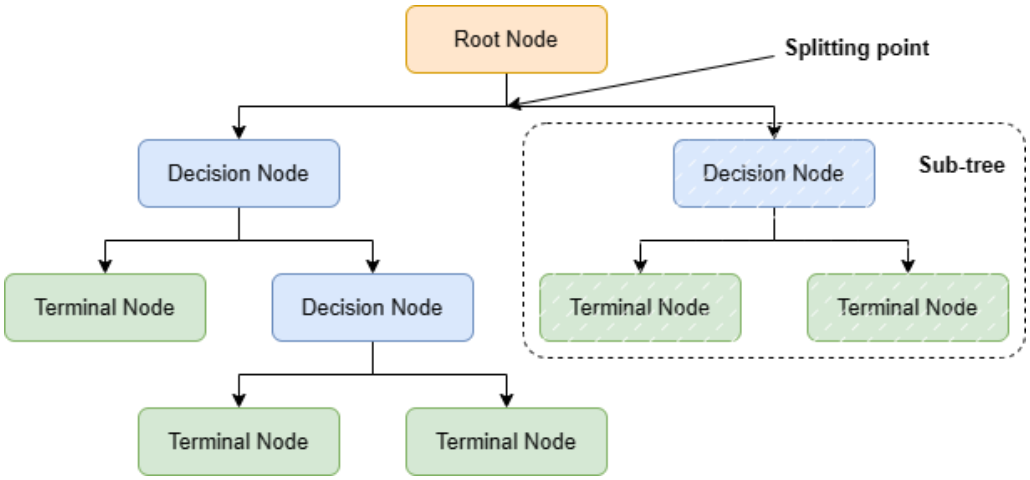


Figure 3.6. *The overview structure of decision tree.*

in order to get the best decision tree. Also, the number of nodes or average depth can be minimized. The accuracy of a tree is significantly impacted by the choice of strategic splits. For classification and regression trees, the decision criteria vary.

Decision trees employ diverse methods to decide when to split a node into several sub-nodes. The creation of sub-nodes enhances the uniformity of the groups obtained. Node purity also rises as the target variable increases. The decision tree splits the nodes using all the accessible factors and selects the division that leads to the most homogeneous sub-nodes.

The algorithm selection is also based on the type of target variables. There are several common algorithms frequently used in Decision Trees:

- Extension of D3: (ID3).
- Successor of ID3: (C4.5).
- Classification and Regression Tree: (CART).
- Chi-square automatic interaction detection when constructing classification trees does multi-level splits: (CHAID).
- Multivariate adaptive regression splines: (MARS).

With no backtracking, the top-down greedy search method of the ID3 algorithm, which is the Iterative Dichotomiser 3 is a decision tree algorithm created in 1986 by Ross Quinlan., creates decision trees by traversing the space of potential branches based on a given dataset. As the name implies, a greedy algorithm always selects the option that, at the time, appears to be the best. [45]

Furthermore, decision trees have several unique qualities that are similar to supervised learning approaches. The properties of decision trees are determined by several conditions:

- Sufficient data: A decision tree, like any other data mining approach, requires data. The number of training instances should be adequate to allow for a successful and sturdy tree building. The number of test cases is also critical in validating the decision tree's correctness [46]. Furthermore, each class should have a sufficient number of instances to appropriately represent that class [44].
- Erroneous values: Errors may happen in the training data, although decision trees may tolerate these flaws. The mistake might be in attribute values, classification labels, or continuous output values.
- Attribute-value pairs: A data instance to be studied must be in attribute format, with each attribute having its own set of values. These values may be categorized or numerical. In distinct data instances, the same property cannot have various value types. [46]
- Missing values: Although the training data may contain missing data instance values, decision trees may also accept these missing values. Attribute data, categorization labels, and continuous output values may also be absent.
- Predefined output expectations: Each data instance that will be trained or tested should be allocated a classification label or a numeric output value.

3.2.3 Types of Decision Tree

Depending on the type of target variable, different decision trees are used. There are two varieties:

- Categorical Variable Decision Tree (Classification trees): Categorical target variables that are separated into categories are included in a categorical variable decision tree. Attributes might be numerical or categorical in type. The primary goal of this kind of tree is to divide data into classification labels or classes using classification techniques [47]. Impurity measurements are used to evaluate and decide on tree splits or the quality of characteristics. The characteristic with the highest purity (or lowest impurity) is chosen as the node from which to branch. A purity measure's principal goal is to partition the attribute's values into pure distributions of the classes. The entropy value is one of the most often used impurity metrics [48].
- Continuous Variable Decision Tree (Regression trees): In this type of tree, labels or categories are not required, but the results are continuous values that are used for prediction. Furthermore, entropy or equivalent measures could be employed to determine impurity and mean squared error is utilized. Because regression trees and classification trees are similar, the same algorithm may be applied by simply substituting entropy measurements with mean squared error calculations and class

labels with averages. The production of leaf nodes is the only distinction in the building of a regression tree. These are constructed by averaging the scattered target values of the path traveled after all branching is completed until the leaf node. Furthermore, the resultant tree is binary since nodes are always divided into two partitions: some value larger than or equal to the provided value, and some value less than the stated value.

3.2.4 Attribute Selection Measures

Entropy

Entropy is a term used to describe the degree of disorder, randomness, or uncertainty present in a system. It is a scientific concept and a measurable physical property. Entropy attempts to compute the average amount of information included in each received message [49]. In the context of a decision tree, entropy is used to determine which feature to split on at each node. The goal is to find the feature that results in the most information gain, which is the difference between the entropy of the parent node and the weighted sum of the entropies of the child nodes. Based on ID3 algorithm, with S is the current state, c is the number of features, i is the feature and P is the probability of i, Entropy is defined as [50]:

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{3.2}$$

In general, the entropy-based approach to decision tree learning seeks to maximize the homogeneity of the resulting subsets at each level of the tree, leading to a more accurate and interpretable model.

Information Gain

One of the first and most widely used decision tree attribute selection criteria was information gain. With no backtracking, the top-down research method of the ID3 algorithm, which is Quinlan, the creator of the ID3 technique, was also the first to employ the information gain selection criterion in a decision tree. However, information gain would not have existed without Claude E. Shannon's discovery of entropy [51], creates decision trees by traversing the space of potential branches. The information gained quantifies how successfully a certain variable splits training instances into their intended classifications. Finding a characteristic that yields the most information gain and the lowest entropy is the key to building a decision tree. The arrangement of characteristics in nodes is determined in part by information gain. The information gain is the difference between the entropy of the parent node E_{parent} and the average entropy of the child nodes E_{child} [50]:

$$InformationGain = E_{parent} - E_{child} \tag{3.3}$$

As a result, information gain is computed for each conceivable attribute that may be branched on the test node to discover the attribute with the highest information gain among the other attributes. The attribute with the biggest gain is forked, and the procedure is repeated until the classification is finished.

3.3 Driver Distraction Detection Algorithms

After extracting the pose of the driver, the information of the pose is passed to a preprocessing stage.

3.3.1 Preprocessing

Since the pose of the driver is a set of coordinates of each part, they can be considered as fixed points. Each position of the driver is a set of fixed points, and this may require numerous frames to train the decision tree. Thus, a pre-processing stage is used to convert fixed points to distances between the neck and other parts. The model can recognize that position is in a particular pattern. To monitor drivers, the upper half of the body is only taken into account, which means only the eyes, ears, nose, neck, and shoulders are utilized to input the tree.

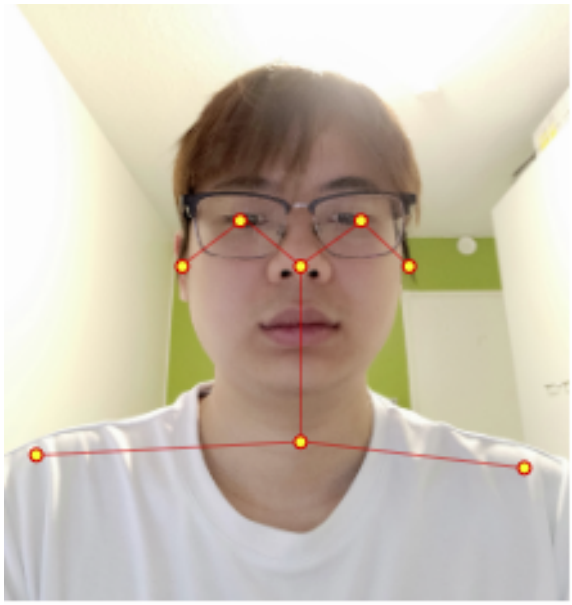


Figure 3.7. *Set of fix points – output of PoseNet.*

To calculate the distances between the neck to other parts, the distance formula is ap-

plied. The length of the line segment connecting any two points is the distance between them:

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.4}$$

An image can be considered as 2D coordination, and all the distances are converted by applying the distance formula:

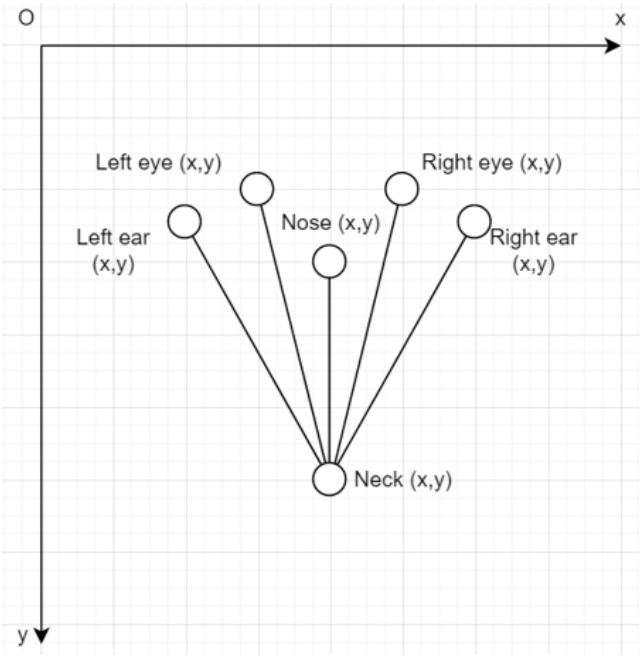


Figure 3.8. *Converting to distance based on distance formula.*

Moreover, this compression step can help to reduce the number of features of the decision tree from sixteen parameters to seven parameters. If the fixed points are used as the main features of the model, both x and y of the eight parts will be set as the input features of the model. However, after converting to distances, there are only seven parameters, such:

- From neck to left eye: denote as NeLey.
- From neck to right eye: denote as NeRey.
- From neck to left ear: denote as NeLea.
- From neck to right eye: denote as NeRea.
- From neck to left shoulder: denote as NeLs.
- From neck to right shoulder: denote as NeRs.
- From neck to nose: denote as NeNo.

Before putting it into the model, the seven features need to be normalized. The largest value in the package needs to be obtained, and divide the seven features by that value to get the normalized values.

3.3.2 Algorithm

Since the purpose of the model is to classify the status of drivers, the algorithm Classification and Regression Tree: CART is chosen to build the tree. Similar to C4.5, CART allows for numerical target variables but does not build rule sets (regression). By choosing the feature and threshold that maximizes information acquisition at each node, CART constructs binary trees.

Mathematical formulation

A decision tree divides the feature space recursively, grouping samples with similar labels or target values together. Consider each node to be a yes/no question; if the answer is yes, the samples will go to group A; otherwise, the samples will go to group B. More nodes are created if this method is performed recursively. Converting to mathematical formulation, denote that:

- $x_i \in R_n$ with $i = 1, ..., l$ is a training vector.
- $y \in R_l$ is a label vector.
- The data at node m be expressed as Q_m with n_m samples.
- Feature j and threshold t_m form candidate split $\theta(j, t_m)$
- Left and right data subset $Q_m^{left}(\theta)$, $Q_m^{right}(\theta)$

The sample set nm is divided into left and right subsets at node m based on the threshold tm. If the value is greater than or equal to tm, the sample is sent to $Q^{left}m(\theta)$, otherwise it is sent to $Q_m^{right}(\theta)$ [52]:

$$Q_m^{left}(\theta) = \{(x, y)|x_j \geq t_m\} \tag{3.5}$$

$$Q_m^{right}(\theta) = \frac{Q_m}{Q_m^{left}(\theta)} \tag{3.6}$$

After splitting the samples, the quality of a candidate split is calculated by using an impurity function or loss function $H()$, these functions are selected based on the task of the model which is classification or regression [52]:

$$G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta)) + \frac{n_m^{right}}{n_m} H(Q_m^{right}(\theta)) \tag{3.7}$$

The target of this step is to choose the θ value that can minimize the impurity $G(Q_m, \theta)$. From this point, the nodes are created until the depth of the decision tree is maximized, the number of samples in one subset is smaller than the minimum samples, or the total

samples n_m is only one sample left.

Classification criteria

If the goal is to produce a classification result with values of 1, 2,..., K, the only changes to the tree algorithm are the criteria for splitting nodes and pruning the tree, so the percentage of class k in node m is computed by (3.6), and because the model is built based on the Entropy criteria, the impurity is measured by (3.7) [53]:

$$p_{mk} = \frac{1}{n_m} \sum_{y \in Q_m} (I(y = k)) \tag{3.8}$$

$$H(Q_m) = -\sum_{k} p_{mk} \log(p_{mk}) \tag{3.9}$$

Additionally, the entropy criterion calculates the Shannon entropy of the feasible classes. This computation involves using the class frequencies of the training data points that have reached a particular leaf as their probability. The Shannon entropy serves as a means of minimizing the log loss, which is also known as cross-entropy and multinomial deviance, between the genuine correct labels and the probabilistic forecasts for class made by the tree model [54].

4. RESEARCH AND TEST METHODOLOGIES

This thesis seeks to answer three main questions: 1) Which aspects of the head pose affect the decision of the model; 2) How can the model classify the status of the driver from the head pose joints; 3) How the application can adapt the real-time requirement. To illustrate a general solution, the strategies utilized to address these questions are presented in this chapter. Section 4.1 introduce research methodologies about the related aspects and section 4.2 gives more information about verifying and testing the decision tree.

4.1 Reseach Methodology

To address the three questions, the research methodology in this thesis adopts a design science approach, which involves a cyclical process of design, implementation, and evaluation aimed at creating and evaluating a system that detects the driver's head orientation effectively and efficiently in real-time. This method encompasses constructing an artifact, such as a system or process, to address a specific problem and evaluate it to ensure that it meets the desired objectives and achieves the intended outcomes.

In this case, the artifact is a decision tree model trained to detect the driver's head orientation based on head movements. The heads' orientation contains a wealth of interpersonal information. For instance, a person turns his head in the direction of the intended focus of a conversation. Similarly, in a conversation, the head direction is a nonverbal indication that tells a listener when to swap roles and start speaking. In terms of driving behavior, the head attitude also is reserved for certain tasks in the cabin, such as looking and talking to passengers. As a result, the study focuses on the driver's head orientation, and the decision tree model is trained to detect the state based on head movement. The model is designed and evaluated iteratively to ensure that it meets the desired objectives and achieves the intended outcomes. Next, the workflow of the tree is researched to conduct a suitable configuration for the input features as well as the accuracy of the results. Utilizing the advantages of an embedded board such as I/O devices and communication ports, embedded boards, NVIDIA Jetson and Raspberry Pi in this case, are chosen to accomplish the work. Lastly, a parallelism technique is also considered to increase the performance of the procedure to respond to real-time needs. Overall, the adoption of

design science in this research methodology ensures a rigorous approach to problem-solving and aligns with the principles of creating effective and efficient artifacts to address specific problems.

4.1.1 Dataset

The selection of datasets is critical in the experiment since the data must be credible and appropriate. The data must be in the right format and ready for categorization in order to be considered applicable. Furthermore, not all forms of data are appropriate for decision tree learning or categorization. As a result, datasets must be carefully selected and, if not previously, preprocessed.

Pointing'04

This collection has 15 image sets that represent 15 different people. Each set covers two sets of 93 images of the same person in different poses. To be specific, 15 people in the database who wear glasses or not, have varied skin tones and have varying head and neck sizes. The head poses spanning from -90 to +90 degrees and determine posture or head position. [55]

Biwi Kinect Head Pose Database

There are about fifteen thousand photos of 20 persons (six females and fourteen males - four people were captured twice) in this database. A depth picture, the accompanying RGB image (both 640x480 pixels), and the annotation are supplied for each frame. Like the above dataset, the head position range is around +-75 degrees yaw and +-60 degrees pitch. The training set can cover all conceivable positions of the drivers whereas they execute certain fundamental behaviors such as glancing about, talking to other passengers, and checking the car's mirrors due to its wide range of head pose positions.[56]

4.1.2 Model Implementation

The model tree may be constructed after the dataset has been established. The aforementioned research topics and other aims influence the development of a realistic decision tree model. The development method includes a stage for building up tree properties as well as a validation step to effectively generate a high-accuracy model. Finally, the model is exported so that it may be executed and tested on an embedded platform.

The dataset used to build the model is a measure of the uncertainty of a random variable. Hence, entropy is used as the main criterion because it can indicate the impurity of an arbitrary set of cases. Another attribution that massively affects the model is the splitter.

In this case, the best splitter is applied to the model. Because the goal is to divide the data so that the resulting group of records at the new node has the least amount of impurity. Because the parent split did not successfully divide the data, a node with high impurity has a high population of numerous possible values of the target variable. The model is expected to be ready for training at this point, but a few minor post-training steps can be explored to increase the model's accuracy. There are various side attributions that may be fine-tuned to produce better outcomes, such as the maximum depth of the tree, the number of characteristics to examine when looking for the optimal split, and so on.

Each action while driving happened very quickly and the system needs to capture the activities in millisecond units, hence real-time requirements are demanded in this type of solution. To run efficiently on an embedded platform, a real-time structure program must be implemented to help the model respond quicker in real life. As a result, transforming the serial execution to parallel execution is an alternate technique to speed up the processing time.

4.1.3 Experimental Platform

NVidia Jetson Nano

The NVIDIA Jetson Nano Kit is a compact but powerful computer that allows users to run several neural networks in parallel for image classification, object recognition, segmentation, and voice processing. All in a simple platform that consumes as low as 5 watts. JetPack is interoperable with NVIDIA's world-leading AI platform for training and deploying AI applications, minimizing developer complexity and effort [57].

According to Erik Ostrowski and Stefan Kaufmann, [58], the processing time when performing with TensorFlow is three times faster than the other model. In terms of computing power, a speed-up of around 157% for both Mobilenet Version 1 models and the TensorRT optimized TensorFlow models, and a speed-up of 485% if we convert the model to Nvidia's UFF format. For the Mobilenet Version 2 model, there are even larger improvements. In other words, if the requirement is to deploy the Jetson Nano for application, TensorRT or TensorLite should be considered. [58]

Coral Accelerator USB

This device helps the system to increase the speed of the AI work. The onboard Edge TPU coprocessor can execute 4 trillion operations per second (TOPS), consuming 0.5 watts for each TOPS (2 TOPS per watt). It can, for example, perform cutting-edge mobile vision models like MobileNet v2 at around 400 FPS while remaining power efficient [59].

In [60], when experiencing the speed of the accelerator USB, the frame rate of Coral was around 6-7 times faster than its NVIDIA equivalent. This is especially noteworthy given that the Coral USB Accelerator consumes around one-fourth the power (2.5W) of the Jetson Nano (10W).

The explanation for the significant difference is most likely the performance of the specialized Edge TPU ASIC vs the broad GPU architecture of the Jetson Nano. Coral, for example, only employs 8-bit integer values in its models, and the Edge TPU is designed to take maximum advantage of this. Because of the decreased integer precision, detection accuracy is decreased. The bounding boxes of the discovered objects were collected and calculated the mean average precision (mAP) for all 80 classes in the COCO dataset was to measure the platforms' accuracy when conducting object detection. Erik Ostrowski and Stefan Kaufmann pointed out that with the help of Coral devices, the accuracy of the Nano is approximately 12.60 percentage points higher than the other and the processing time also has been improved tremendously [58]. Therefore, performing optimized TensorFlow Lite models with Coral Edge TPUs is selected to guarantee accurate results that can adapt to real-time requirements.

4.2 Test Methodology

Assuming a model is developed using a specific algorithm and trained on a given dataset, achieving a high accuracy of 95% or more on the training data does not necessarily mean that the model is suitable for prediction. The reason is that the model has become familiar with the training data and has learned to generalize effectively over it. However, the model's accuracy on new and unseen data is likely to be low, as it has not been exposed to this data during training and therefore fails to generalize well over it. This phenomenon is known as overfitting, which can be a major challenge in machine learning and can result in poor performance when applying the model to real-world scenarios. Therefore, it is crucial to evaluate the model's accuracy on unseen data to ensure that it can generalize well and make accurate predictions in practice. In this case, cross-validation provides a useful approach for estimating the performance of a model. Specifically, cross-validation involves dividing the available data into two distinct sections: a training set, which is used to develop the model, and a test set, which is used to evaluate its performance. By repeating this process multiple times with different splits of the data, cross-validation provides a more reliable estimate of a model's accuracy on new, unseen data.

To that end, the training set has been divided into 2 sets: 60% for training and 40% for testing. The prediction is tested and determined to evaluate how effectively the model performs on the testing set. If the model performs well on test data, it implies that it has not overfitted the training data and can be trusted to make predictions. However, if it performs poorly, the model cannot be trusted and further tweaking steps are required to

enhance it.

To observe the performance of the program - in terms of speed, profiling code is an alternate approach for tracking the program. It is a technique for assessing how much time is spent on a program. It is easier to verify the delay time that occurs at whatever step if the program structure is broken down into little sections. There are three major sections in this work: interact with the driver's stance through the camera, identify the driver's state, and alert stage. Brief footage is used as a testing sample to see how the software has been enhanced to adapt to the real-time requirement. This can aid in fine-tuning the procedure to accommodate any real-time restrictions. The frame rate is the metric to observe the speed of the program. The running time of each major state is measured and exported as a data file type, for example, excel or CVS file. At this stage, a line chart can be exported from the data to conduct an overview of the performance.

5. DESIGNED HEAD POSE ESTIMATION AND CLASSIFICATION SYSTEM

In this chapter, the design scheme as well as the software structure is be introduced. Section 5.1 demonstrates the overall architecture of the work, while Section 5.2 depicts the software structure and related information of the head pose estimator and the decision tree. Lastly, the parallelization scheme also is presented.

5.1 Overall Architecture

Figure 5.1 illustrates the program architecture which includes two main parts: Server and Client. The Client detects the status of the driver while the Server is used as an alarm device. The purpose of separating it into two parts is to reduce the number of tasks for one device as well as save more time to adapt the real-time requirements. The server connects to the Client via a Socket protocol, which is utilized to transmit and receive data between different network systems [61]. In this work, the socket Transmission Control Protocol, generally known as TCP/IP, is utilized as a data transfer technique because it is faster and guarantees data delivery to the destination router, making it reliable.

In the Client, the PyCoral API is an external library that performs a small set of utility functions such as establishing the TensorFlow Lite Interpreter with the Edge TPU delegation and performing other inference chores for typical models such as interpreting a labels file, pre-processing input tensors, and post-processing output tensors. In other words, the API simplifies the process of loading and using the Edge TPU model.

The 'Control' section contains the majority of the management instructions. All data structures, such as pictures, frames, data frames, or decision tree results, are gathered the information at this section for transmission to the Server. This is also where the PoseNet and trained decision tree are loaded via the PyCoral API. Furthermore, 'Control' instructs the daemon thread 'Webcam' to process in order to communicate with the real world.

The queue-structure thread, which is located in the Thread block in figure 5.1, is employed by the application to conduct parallel tasks. It is responsible for scheduling tasks based on established dependencies and performing them in different threads. These threads are generated in 'Control' and placed in a pool, where they wait for execution authorization.

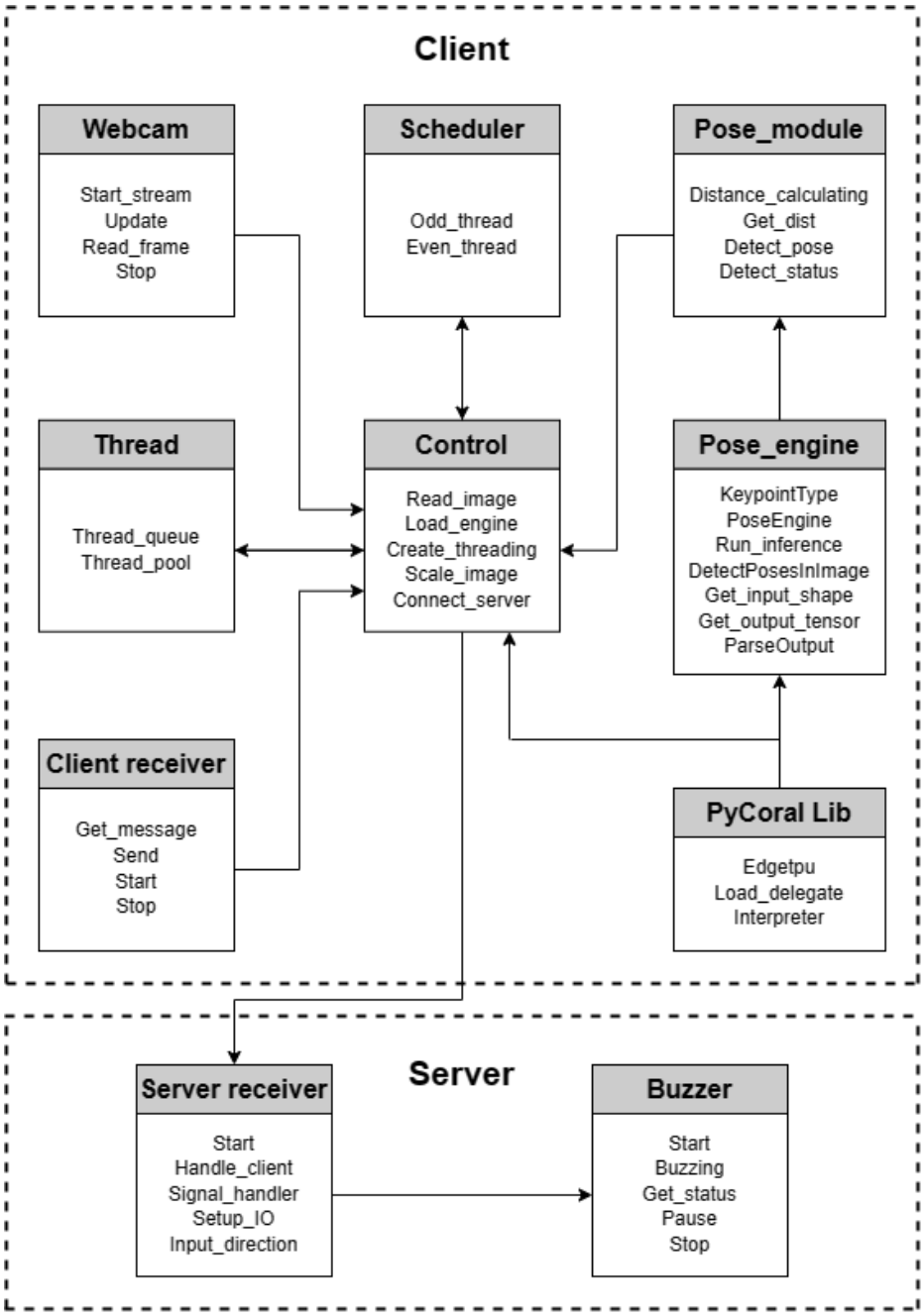


Figure 5.1. *Driver's distraction estimator architecture*

The major thread functionality is gathered under 'Scheduler', and the tasks are defined under 'Pose modules' and 'Pose engine'. The 'Pose engine' contains all of the essential stages from preprocessing frames to exporting driver poses, while the compressed data frame and normalized data are stored in the 'Pose modules'. From there, the result is transported back to 'Control', and sent to the Server by using the communication functions in 'Client receiver'.

The Server takes care of listening to the clients and package analysis. The tasks are executed under 'handle client', which is also a thread structure that is useful in terms of collecting the package from the client on time. Because the buzzer is a peripheral device,

I/O communications are necessary to drive it. As a result, the 'setup I/O' is produced to initialize the ports for the buzzer. Furthermore, the 'Buzzer' is a daemon thread that is designed to adjust alarm requirements, such as delay time, of the Buzzer thread.

5.2 Software Architecture

Figure 5.2 illustrates the workflow of the program. There are two main components in the diagram: Server and Client.

The Client's function is wrapped up with 'Webcam,' 'Control,' and 'Scheduler,' as shown in the figure 5.2. The 'Webcam' layer basically is a daemon thread which is the thread that constantly operates in the background and gives assistance to main or non-daemon threads. The Daemon Thread runs in the background and does not prevent the main thread from quitting [62]. To begin, the 'Webcam' checks for accessible webcams and begins reading a single frame from the video capture stream. To increase the precision of the pose estimator, the input frame is scaled to meet the Coral model's required input size and transmitted to the main thread 'Control'. Also, if no more frames are requested, the stream is closed and the webcam is released.

In the 'Control' layer, the models are loaded in the main thread at the same time as the 'Checking webcam' step. Then threads are created, and the operation of each thread is fairly straightforward. Firstly, in the main loop of the 'Control', the threads are initialized based on the 'Scheduler' layer. An already resized frame is returned from the 'Webcam' and written to the designated thread. Following that, a resized frame is fetched to the designated threads. At the completion of each loop, the generated package is released from each thread and transmitted to the server. This main loop is continued running until a 'stop' signal is detected or there are no more frames to analyze. Finally, after exiting the main loop, the main thread releases all assigned resources and sends a disconnected signal to the Server.

The 'Scheduler' layer generates the thread configuration. This setup is then utilized to prepare the threads for processing by initializing them. The input frame is read and fetched into the PoseNet model at the beginning, which is the responsibility of 'Get pose location.' A 'Request new thread' step generates parallelism architecture for the application. It instructs the main thread 'Control' to generate a new thread to perform alongside the current thread. A more detailed explanation will be provided in a forthcoming section. The data from the pose position is then provided to the 'Get driver's status' stage, which extracts the driver's status and other necessary information. These values are returned to 'Control,' and the thread terminates itself to complete the procedure.

The 'Server' module must continuously be active in order to prepare for the upcoming request from the 'Client.' The first step is to determine whether or not the connection

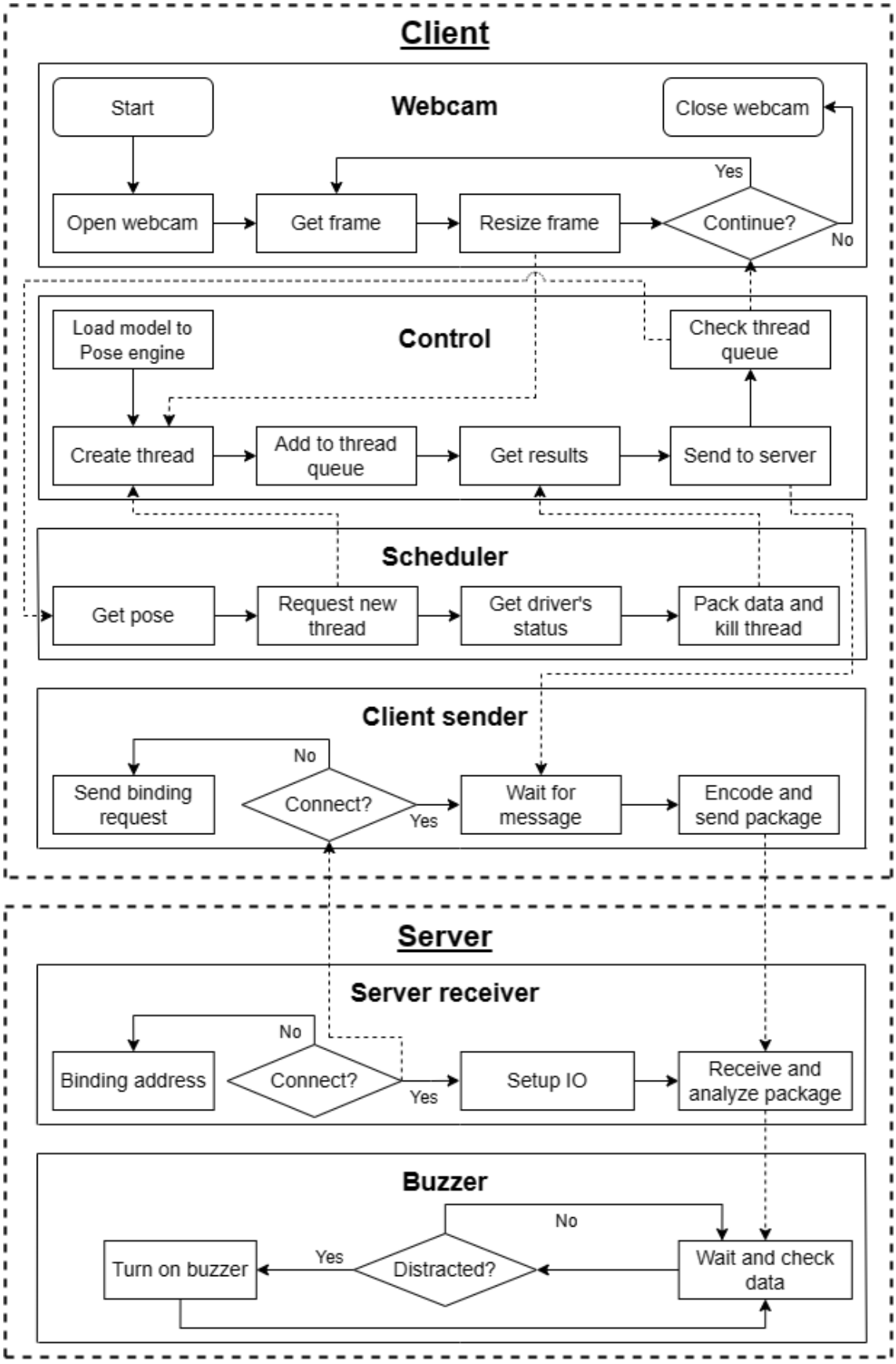


Figure 5.2. *Workflow of program.*

has been established. If the server is linked to the client, the 'Client' proceeds to retrieve and encode the data package while the server configures the I/O ports for the buzzer. Following the completion of these processes, the package is transmitted to the server through a socket connection. The data is then analyzed by the server. Depending on the results of the decision tree, the server activates the buzzer to inform the driver. A supplementary 'Delay time' step is added, however, to provide a sufficient delay between each time the buzzer rings. Furthermore, to meet the real-time need, the 'Client sender' and 'Buzzer' are implemented using the daemon thread approach.

5.3 Workflow

5.3.1 Client Working Principle

The Client component contains three parts: Webcam, Control, and Client Sender. It controls the webcam-capturing process as well as the classified procedure. As described in Figure 5.2, there is a Scheduler part which is the based workflow of each thread to process a frame received from the Webcam.

Webcam

The video processing normally is divided into two sections: retrieving the next available frame from a camera and applying further processing on the frame. Reading and processing the next frame could be executed sequentially in a program without multi-threading. The program pauses for the next frame to become available before doing the necessary processing on it. The time it takes to read the frame is mostly related to the time it takes to request, wait for, and transmit the next video frame from the camera to memory. As a result, the time it takes to conduct computations on the video frame, whether on the CPU or GPU, accounts for the majority of the time spent on video processing.

However, reading the next frame and processing can be parallelized with multi-threading. While one thread reads the next frame, the main thread uses the CPU or GPU to analyze the previous read frame. By overlaying the two activities, the overall time for reading and processing the frames is lowered. Therefore, the webcam capture is developed always active, and whenever the main thread requests a new frame, it is ready to submit the frame to the 'Control'. With the Threading library in Python, a daemon thread is utilized for handling webcams. When starting the stream, the available webcam has to be checked before processing further. Only if the webcam is accessed, the stream can be continued to grab the frame. Then the frame is scaled to match with PostNet input. As described in the Coral API, each ready-made model has a specific input size of the image. Therefore, to obtain perfect results, the frame has to be scaled down to fit the chosen model.

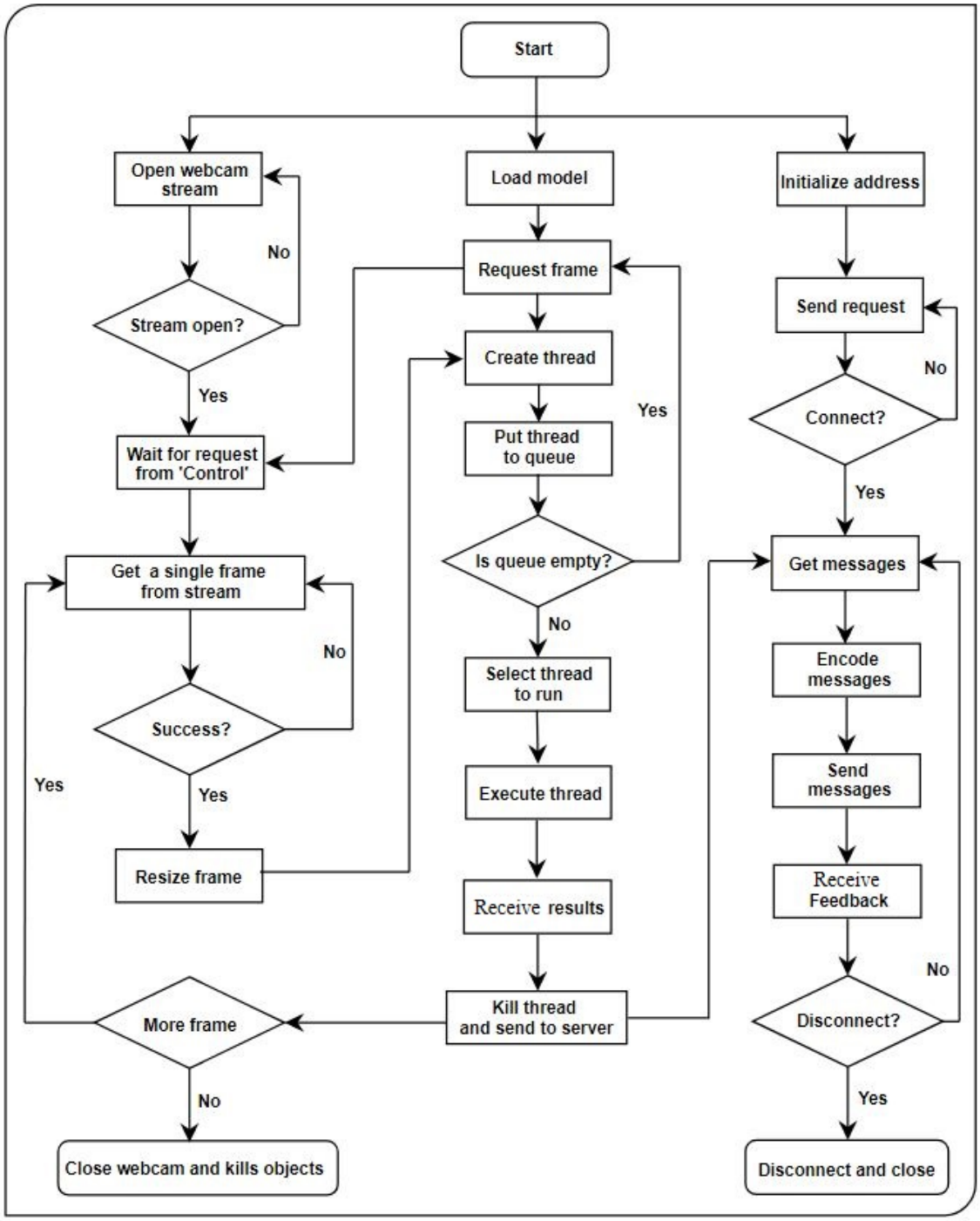


Figure 5.3. *Client workflow process.*

Control

At the main thread, the models are loaded to perform. The program uses a First-In-First-Out queue (FIFO), which is notably helpful in threaded programming where information must be passed among different threads in the correct order. In this solution, a thread obtains a frame and resources to run the model. However, to avoid conflict in using resources, threads are pushed to a queue to wait. When the resources are available, the

threads are executed as a first-come-first-server approach.

Scheduler

Figure 5.4 describes the way that two small tasks run concurrently in each thread: get the status of the pose and the head direction. In 'Get Pose Joints', the PoseNet is used to parse interpreter output tensors and return decoded poses. Each pose contains seven parts of the upper half of a human including the shoulders, neck, nose, eyes, and ears. Each component has its own x and y coordinates, which represent the pixel value in the picture. Then, by applying the method introduced in Section 3.2.1, the sixteen features are compressed into seven for the decision tree in the 'Calculate distance'. Before importing to the decision tree, the data has to be normalized. Data normalization seeks to eliminate data redundancy, which exists when many fields contain redundant information. The database may be more adaptable by applying normalization.

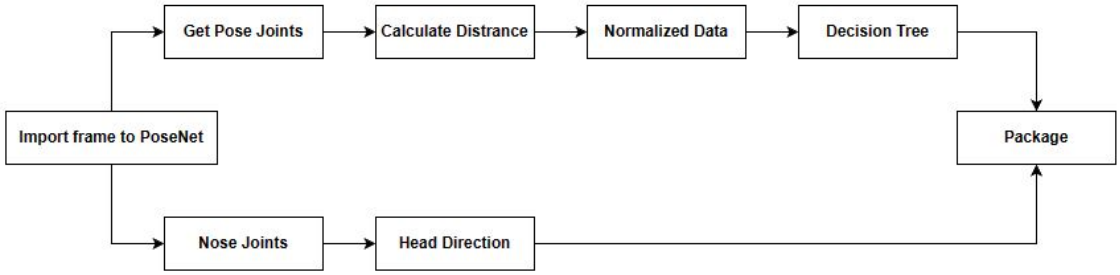


Figure 5.4. *Scheduler workflow.*

Client Sender

In this Client, another daemon thread is employed to establish a connection to the server and work in the background to transmit data on time. When the program starts, this thread is responsible to collect the address and port to interface with the server. The communication that is utilized is called Socket which began with ARPANET in 1971 and eventually evolved into Berkeley sockets, an API in the Berkeley Software Distribution (BSD) operating system launched in 1983 [61]. Client-server applications are the most popular form of socket applications, in which one side operates as the server and waits for connections from clients. The socket module in Python provides access to the Berkeley sockets API.

Once the binding is established, this thread waits for the packages and encodes them in the 'utf-8' format. A socket can only transport bytes but a string is a series of characters rather than bytes. To transport a string across a socket, it must first be encoded as a series of bytes and then decoded. Hence, 'utf-8' format is selected in this case. In the 'Receive feedback' process, this stage is used to check that whenever the server wants

to disconnect and inform into another thread to stop processing.

5.3.2 Server Working Principle

In this thesis, the server functions as an additional control unit to manage the car's alarm system. In the real model, the control units communicate through CAN. Due to the limitations of the testing boards, however, Ethernet is utilized to imitate CAN communication. The procedure that occurs within the server is detailed below. Figure 5.5 illustrates how the server works in order to alarm the driver on time. Furthermore, the warning module also is introduced.

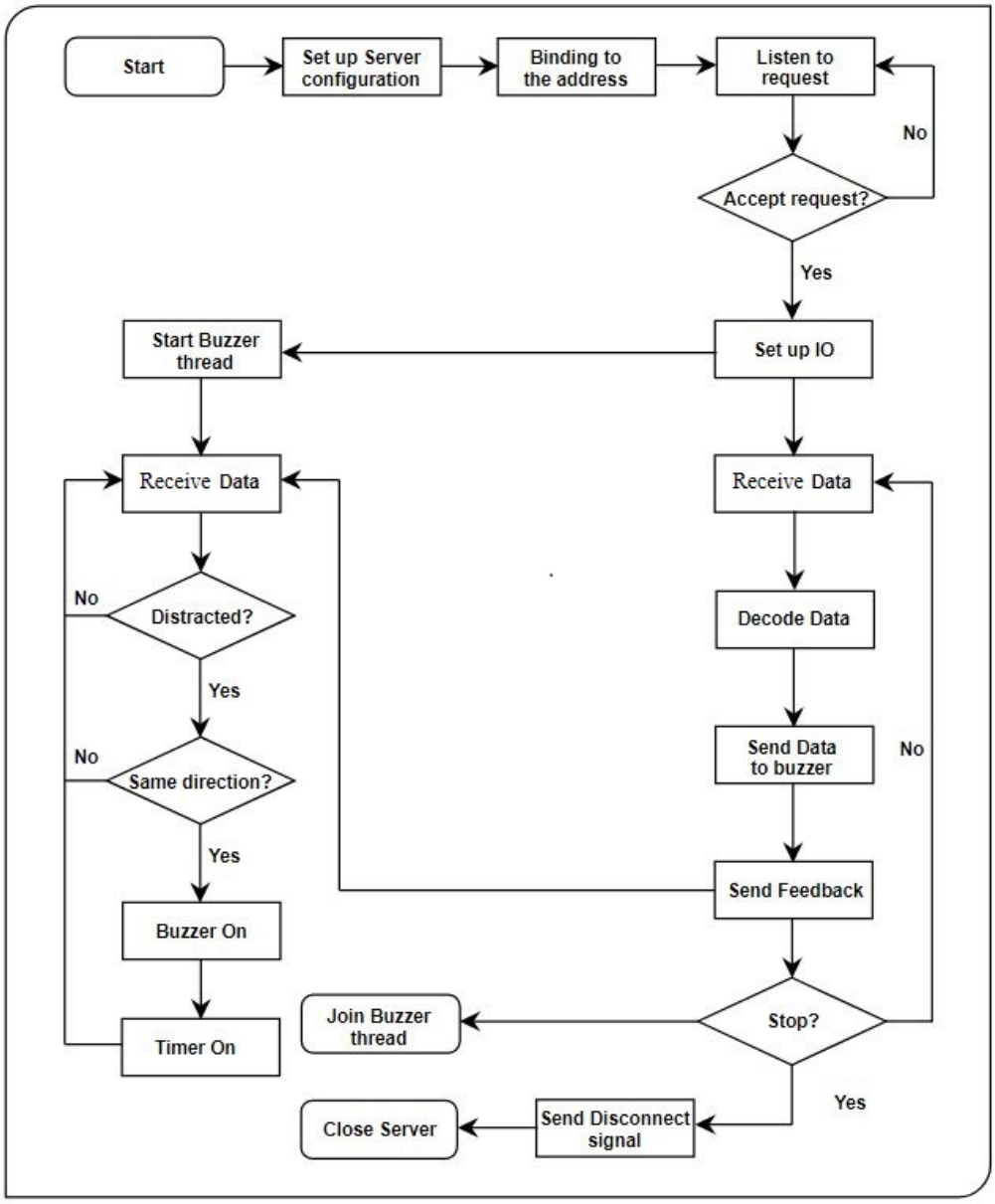


Figure 5.5. *Server working principle.*

Software design

The configuration of the server must be defined clearly. In this case, the socket was set to SOCK_STREAM and the address belong to the AF_INET address family. A pair (host, port) was included, where the host is a string indicating a hostname in internet domain format or an IPv4 address such as '100.50.195.5', and the port is an integer. Then, the server is bound to the static IP of the testing board, and this address must not be bound already. At this point, the server is able to listen to the client and connect to the request.

Once the connection is established, the input and output ports must be configured to operate the buzzer. Another daemon thread handles the buzz, which is launched when the connection between the devices is established. When the data arrives at the server, it is encoded into a string message that includes two pieces of information: the status and the heading. These data are subsequently sent to the Buzzer thread for further processing.

In a modern car, a control unit monitors the turning signals to determine when the driver wants to turn the vehicle. By incorporating this function into the server, the server can now estimate when to switch on. For example, if the driver's head must turn left to check in the left-side mirror and put on the left signal, the decision tree recognizes that the driver is now distracted and the head is turning to the left. To determine whether to turn on the buzzer, the buzzer compares the turning signal and the direction of the head. If the direction of the head matches the turn signal of the car, the buzzer does not be triggered to alert. Alternatively, a delay time component is added to prevent the buzzer from alarming the driver continuously.

Model design

Figure 5.6 depicts an external module that interfaces with the real world via the buzzer which means alerts the driver when he or she is in distracting status. The warning module was designed in 3D via the SolidWorks program. The material of the box is 4mm-anti-static-black MICA panels.

This module also performs like a car simulation. There are buttons that are used as the main functions of a car, such as turning left or right, running, or stopping. The central heart of this module is the Raspberry Pi 3 which is responsible for all the I/O operations. This board collects the status and head direction and then releases the decision to turn on or off the buzzer. This module has two floors, the first floor contains an Arduino board which is used for reading analog signals and transferring the data to the Raspberry board since the Raspberry board is not equipped with analog I/O. The second floor is for the main function. The hole right above the ethernet spot is the place for the buzzer. Moreover, for a better UI experience, an OLED screen is set up to show the connected status of two

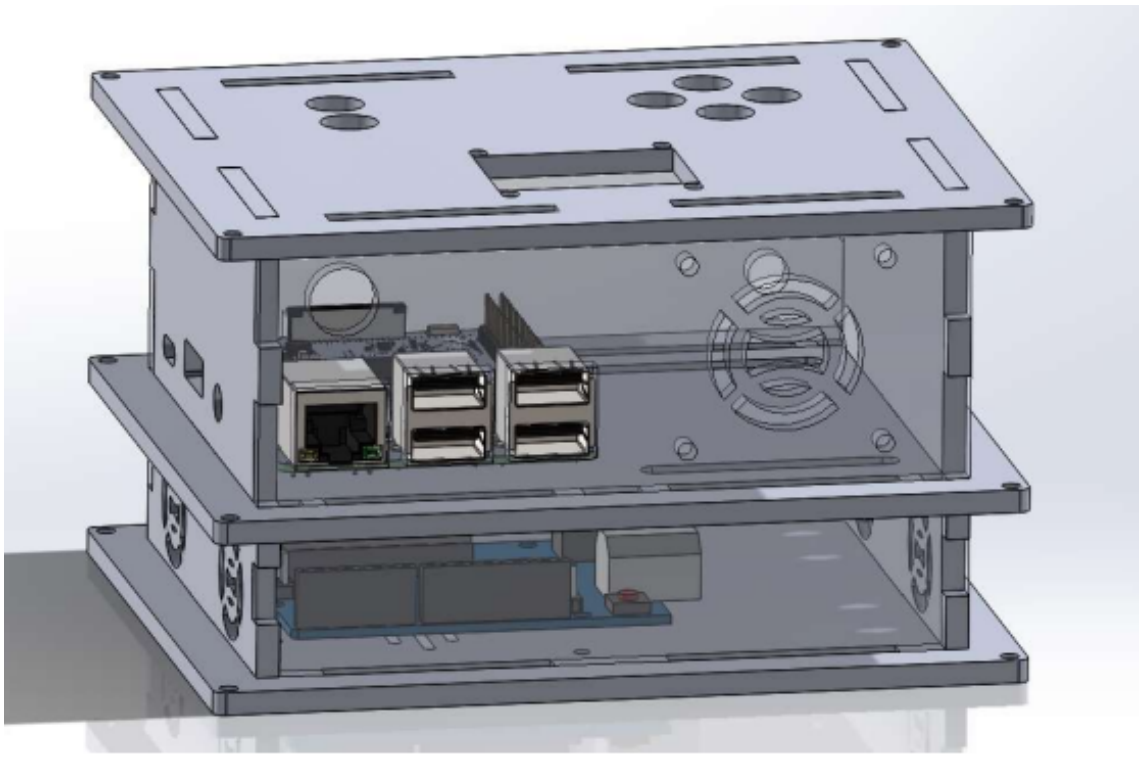


Figure 5.6. *3D design of warning module.*

main boards, the IP address, the status of the driver, and the car. Figure 5.7 is how the module look like after assembling all components.

Figure 5.7. *Warning module.*

5.3.3 Head Direction Estimator

The head direction estimator identifies whether a distracted situation is a purposeful or unintended behavior in a distinct distracted circumstance. An intentional action occurs when the driver desires to maneuver the automobile to the left or right, which necessitates moving the head to check the side mirrors. Unintended behavior appears when drivers move their heads to hear the phone, converse with other passengers, or succumb to drowsiness. Based on these particular points, the algorithm to detect the direction of the head can be described in Figure 5.8.

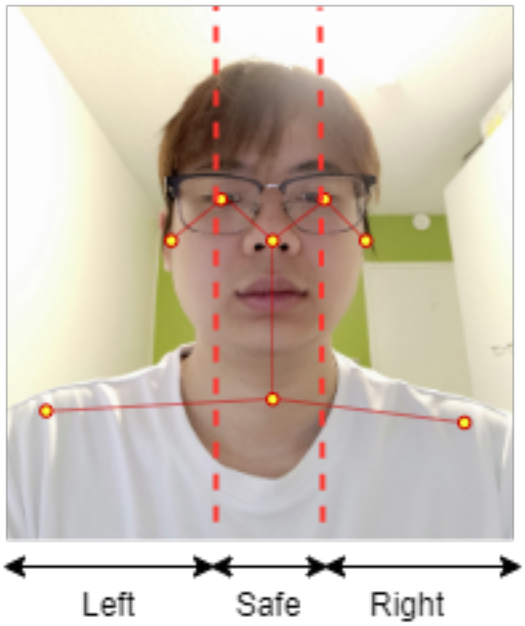


Figure 5.8. *Analyzing head's direction.*

Each captured frame could be thought of as the driver's working space in the cabin. This working space must have a clear view of the driver, which implies that all parts of the driver's upper half body in a safe scenario must be visible in the picture. The image's width is separated into three sections: Left, Safe, and Right. The Safe gap was adjusted so that drivers may turn their heads in an appropriate space. If the x coordinate of the nose passes through one of two red lines, the direction is recorded and formed into a package with the status results. This package is sent back to the main 'Control' thread.

5.3.4 Decision Tree and Processing

Cameras operate as eyes to notice changes and motions in the actual world. The motion of the objects is captured in different frames by the camera. These frames are transferred into the models for further processing. Data preparation and model deployment is made

easier with the help of Python libraries. The model specifics and how the camera function works are provided below.

Prepare Data

A set of around 10000 pictures have been selected from three main datasets: Pointing'04 dataset [55], Biwi Kinect Head Pose Database [56] to extract the features. In order to accurately analyze the poses of individuals in a given set of images, it is necessary to carefully examine the position of each individual's head. This set of images is then processed using the PoseNet model, which is executed on the Coral Acceleration USB. The model is able to extract the position of eight different body parts, including the left and right shoulders, eyes, and ears, as well as the neck and nose.

However, in some circumstances, missing values may occur because the model's posture is difficult to identify. All data were evaluated one by one to see whether any attribute in the dataset had missing values and whether they were relevant to the dataset or not. There were missing attribute values after extracting the feature from PoseNet, and some of them with nonsensical properties. The reason for this is that the number of missing values may have a significant influence on the attributes, turning those attributes meaningless to the dataset. Missing values must be handled in these circumstances to avoid inaccurate Distraction detector model findings. Missing values are classified into three types: Missing At Random (MAR), Missing Completely At Random (MCAR), and Missing Not At Random (MNAR). The possibility of missing data is the same for all observations in MCAR and MAR, and it has a link with other monitored data, but MNAR is dependent on unobserved data. Because the majority of the missing values in this situation are MAR, the whole columns are erased to handle these data.

Attribute	Missing values	Percentage
Left shoulder	20	3.6%
Right shoulder	26	4.34%
Left eye	5	0.47%
Right eye	6	1.83%
Left ear	63	12.87%
Right ear	75	14.64%
Nose	3	0.34%
Neck	0	0%

Table 5.1. *Missing values of prepared data.*

Building Decision Tree

Before importing the features into the decision tree model, it is important to normalize them. This is done by dividing each feature value of the pose by the highest distance value.

Dataset								
No	class	NeNo	NeLs	NeRs	NeLey	NeRey	NeLea	NeRea
0	0	0.86464	0.6538	0.64035	0.97931	1	0.86054	0.89468
1	0	0.86256	0.66209	0.64845	0.97875	1	0.87201	0.89429
2	0	0.86464	0.66726	0.64035	0.97931	1	0.86054	0.89468
3	0	0.86408	0.6777	0.63664	0.98273	1	0.86571	0.89101
...	...	...	...	...	...	...	...	...
9106	1	0.7964	0.90364	0.90364	1	0.94641	0.98585	0.94234
9107	1	0.79403	0.89498	0.89498	1	0.94572	0.98735	0.92336
9108	1	0.78768	0.91862	0.91862	1	0.95583	0.98675	0.97668
9109	1	0.78767	0.93012	0.93012	1	0.965	0.9832	0.966

Table 5.2. *Normalized featured.*

In table 5.2, the class column contains only two values: 0 and 1 stand for safe and distracted respectively. The other columns are values of the distance between the neck to other parts. Those values were normalized by dividing by the longest distance in each row. By doing so, the range of feature values is standardized, allowing for a fair comparison and preventing any one feature from dominating the others during the decision-making process. This normalization step helps to ensure that the decision tree is built based on unbiased feature values and improves the accuracy of the model.

After this step, the processed data is divided into two parts: 60% for training and 40% for testing. The training data is randomized to ensure that each data point causes an "independent" change in the model without being influenced by previous data points. As described in earlier chapters, the decision tree's attributions must be configured as an entropy criterion with the appropriate splitter. Because seven features are imported into the model, the most appropriate maximum number of features is seven, and the appropriate depth of the tree is six.

The table 5.3 provides insights into the crucial features that can be used to develop and enhance the accuracy level of the decision tree. By incorporating these attributes into the model, the decision tree can achieve better performance and produce accurate results while avoiding overfitting. At this point, the testing data is used for checking the performance of the model and determining whether the results of the model are trustable able or not.

Specification	
Parameters	Value
criterion	entropy
splitter	best
max features	7
max depth	6

Table 5.3. *Specification of the decision tree.*

The following code snippet demonstrates how to configure the decision tree's parameters using the Python programming language and the Scikit-learn (sklearn) library. With this method, it is possible to customize the decision tree's maximum depth, the minimum number of samples required to split an internal node, the minimum number of samples required to be at a leaf node, and the criterion used to measure the quality of a split. These parameters can have a significant impact on the decision tree's accuracy and overfitting tendencies.

```
# Import the necessary library for building the tree
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

# Setting up the configuration of the tree
my_tree = DecisionTreeClassifier (
                                    criterion='entropy',
                                    splitter='best',
                                    min_samples_split=5,
                                    max_features=7,
                                    max_depth=6
                                )
my_tree.fit(x_train, y_train)
```

Code Listing 5.1. *Set up decision tree*

After building the decision tree, it is important to test its accuracy using a separate test set. This helps to evaluate the performance of the tree on new and unseen data. By comparing the predicted values of the decision tree with the actual values in the test set, metrics such as accuracy, precision, and recall can be calculated. These metrics provide insights into how well the decision tree model generalizes to new data and can be used to fine-tune the model for better performance.

```
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import confusion_matrix

#Pass the test set to the tree model
y_pred = my_tree.predict(X_test)

# Model Accuracy: how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
# Model Precision: what percentage of positive tuples are labeled as such?
print("Precision:",metrics.precision_score(y_test, y_pred))
# Model Recall: what percentage of positive tuples are labeled as such?
print("Recall:",metrics.recall_score(y_test, y_pred))
print(f"F1 score: {metrics.f1_score(y_test, y_pred)}")
plot_confusion_matrix(my_tree, x_test, y_test)
```

Code Listing 5.2. *Testing performance of the model*

5.4 Parallelization

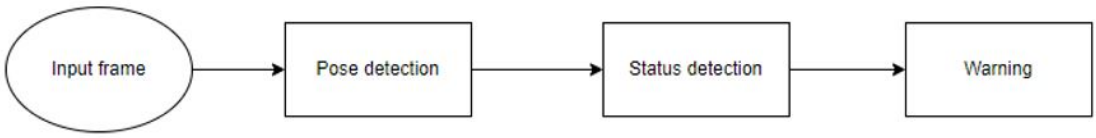


Figure 5.9. *Sequential process.*

A single thread can be summed up in Figure 5.9. Each operation in a pipeline may serve as a producer and a consumer. Each pipeline stage may ingest a data package from the previous pipeline stage, perform the processing, and then generate another data package for the following pipeline stage. However, in the case of running thread by thread, the speed of the program can be considered very slow and cannot adapt to the real-time requirement. Therefore, after analyzing the dependencies between each stage of the pipeline, the gap between 'Pose detection' and 'Status detection' can be utilized to call another thread to work.

The parallelism can be implemented as shown in Figure 5.10. Stage A is for posture detection, stage B is for status detection, and stage C is for warning. A thread is employed as a frame loader in the daemon threading scheme. This thread is executed in the background. It allows the software to save time in stage A while loading the frame.

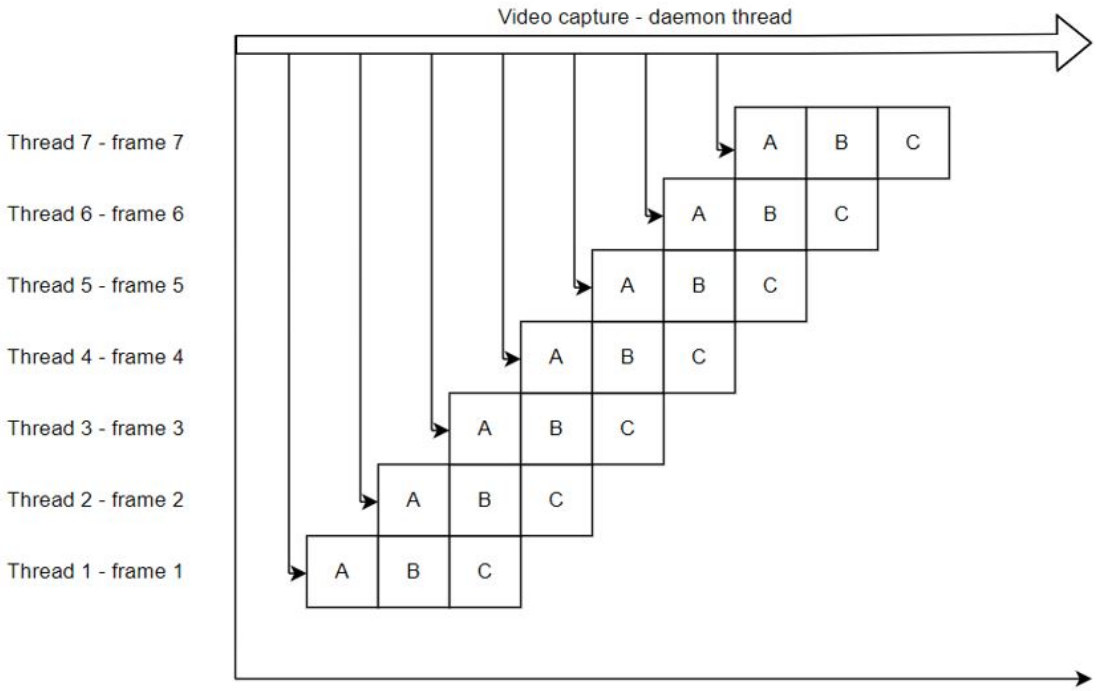


Figure 5.10. *Parallelism process.*

Pipelining in this fashion ensures that subsequent image loads can be conducted as fast as possible without incurring an unacceptable delay. Furthermore, rather than waiting for the processes of stages B and C to complete, each frame may be processed immediately after stage A of the preceding thread has been completed. Once step C is finished, the results are now transferred to the server, and the thread is destroyed.

Moreover, according to Figure 5.10, stage A of thread 1 does not run at the same time as stage A of thread 2, similar to threads B and C. This approach prevents the system from going to a deadlock situation and resource conflict. For example, at the beginning of thread one, the thread provided the resources to run stage A. After receiving the result from stage A, the thread must release the resources before asking for a new one to run stage B. The aim is to prevent a process from retaining one or more resources while simultaneously demanding additional resources.

6. PERFORMANCE RESULTS AND ANALYSIS

This chapter offers a statistical evaluation of the tree decision and the program's speed. Firstly, section 6.3 introduces the important tests that need to be performed. Next, section 6.2 compares post-trained vadiation versus practical results, whereas Section 6.2 presents a speed test of the promoted parallel approach.

6.1 Testing Module

Because this system is being carried out to ensure human safety, testing is essential. The testing module is divided into three components that must be thoroughly examined: accuracy testing, speed testing, and memory consumption testing. Section 6.1.1 illustrates the tool that is used for checking the model's correctness, Section 6.1.2 proposes a technique for checking the frame rate, and Section 6.1.3 will present a method for measuring the memory consumption.

6.1.1 Accuracy Testing

To verify that the model is appropriate to be used in a real-time environment, the accuracy is validated using prepared data from the dataset described in Chapter 5. The metric to determine the accuracy is the number of correct results released from the model - Safe/Distracted status. Moreover, by checking the confusion matrix, a general analysis of the model can be constructed. The relevant information about the model may be thoroughly examined using this matrix. The Python module Metrics from sklearn [52] is used to generate this matrix. This module has ready-made functions for extracting the accuracy level, the degree of precision and recall, and the confusion matrix.

6.1.2 Frame Rate Testing

In this test case, there are two circumstances are checked to see the improvement from serial processing to parallel processing. The figure 6.1 below describes how the process is organized in both fashions:

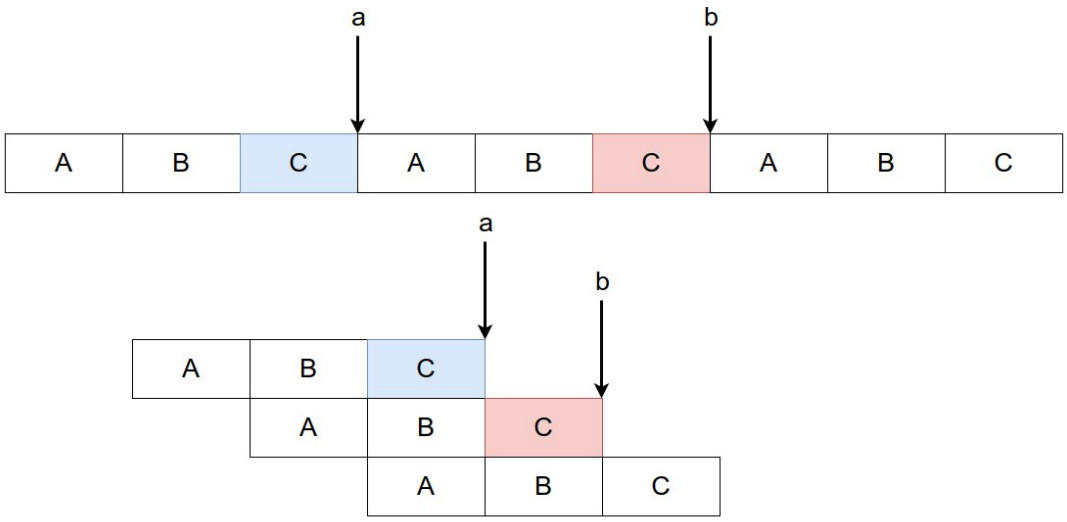


Figure 6.1. *Inference time in a sequential and parallel process.*

In serial processing, the frame rate can be easily calculated by applying the formula (FPS = 1/(b - a)), which means that the inference time of 1 frame, in this case, is the time between point a - starting stage of the process and point b - when the processing frame is completed.

The frame rate of the parallel process, on the other hand, must be closely monitored. In the ideal situation which is the running time of each thread is constant time, the frame rate can be calculated as 1 second divided by the time between two completed frames, point 'a' of the blue C and point 'b' of the red C. However, the ideal situation is hard to reach in the real world. Let's consider red C to be the current frame and blue C to be the previous frame that has been entirely processed. In this case, the formula 6.1 cannot be used since the second thread begins to execute when the first thread is executing. Also, due to the fact that the memory and other elements affect the performance of the board, such as resource usage, and power mode, the frame rate can be calculated by measuring the number of finished threads in a one-second interval.

6.1.3 Memory Consumption Testing

Memory consumption refers to the quantity of memory employed by a program during its operation. The greater the number of variables, the greater the memory usage.

To optimize the system, the memory consumption of the decision tree is monitored using a program called Memory Profiler. This is a Python module for monitoring a process's memory use on Python scripts. Along with the tree model, other models are tested to evaluate performance, and therefore an overall explanation for selecting the decision tree can be presented.

6.2 Performance Result

6.2.1 Statistical Validating

This section presents the performance results from the prepared test set described in Section 5.3.4. Results are exported from the tree decision to validate the predicted performance of the model after being trained with the training set.

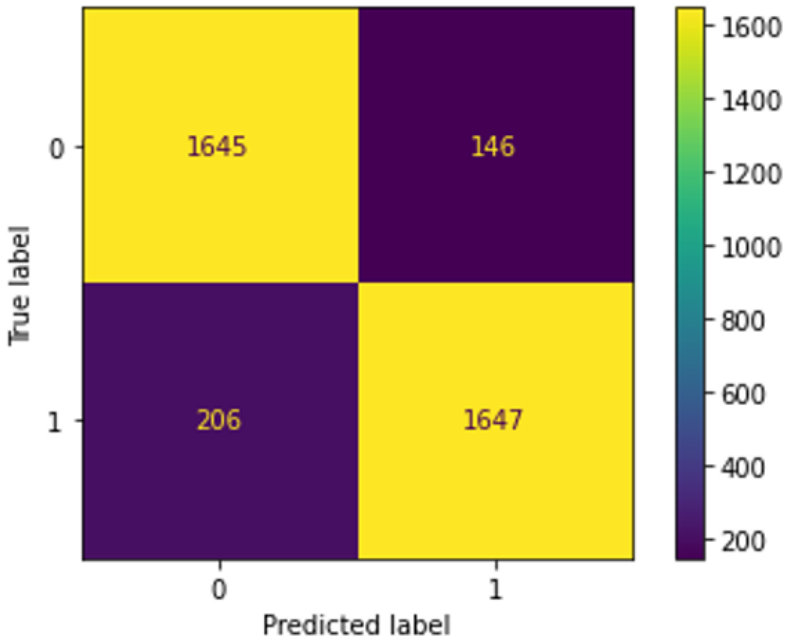


Figure 6.2. *Confusion matrix.*

Using the testing data, the model gave an accuracy of 90.34%, and the following confusion matrix. A confusion matrix is a two-dimensional matrix in which the rows represent the real labels and the columns represent the classifier's predicted labels. In this methodology, for evaluating a classification algorithm's performance, when there is an unbalanced amount of observations in each class or more than two classes in the dataset, classification accuracy may be incorrect. A confusion matrix can assist in comprehending what classification model actually delivers and what sorts of errors it produces. From the matrix, an overall evaluation of the performance of the model can be given.

The diagram shows that the number of true positive (TP) and true negative (TN) cases is significantly high with 1645 and 1647 respectively. In contrast, the figures for false positive (FP) and False Negative (FN) are quite low at 146 and 206. To be more specific, in the first column, '0' stands for safe cases, the model predicts that there are 1645 cases safe while 206 cases are incorrect. Regarding the second column, '1' is the distracted case, there are 1853 cases marked as 'Distracted' but 146 cases were misjudged from 'Safe' to 'Distracted'. This can consider quite an acceptable result.

Furthermore, for a general explanation of why the decision tree was chosen, the table below depicts the performance of each different machine learning model: Support-Vector Machine (SVM), XGboost, Logistic Regression, Logistic Regression, K Neighbors, Random Forrest, and Gaussian Naive Bayes. These models are classifiers that are adequate for the system's objectives. Each model's confusion matrix can be utilized to provide an overview evaluation of its performance and accuracy. All models used the same prepared set of training data and are validated using the same test set.

Model	TP	FP	FN	TN	Accuracy
Decision tree	1645	146	206	1648	90.36%
Support-Vector Machine	1639	152	187	1666	90.69%
XGboost	1683	108	156	1697	92.75%
Logistic Regression	1647	144	401	1452	85.04%
K Neighbors	1696	95	150	1703	93.27%
Random Forrest	1696	95	124	1729	90.99%
Gaussian Naive Bayes	1652	139	855	998	72.72%

Table 6.1. *Models' Performance with prepared test set.*

As can be seen from the table 6.1, there are five models that satisfy the necessary accuracy with above 90%, such as Decision tree, SVM, XGBoost, K Neighbors, and Random Forrest. The number of the true positive and true negative of these three models is significantly higher than the others. In the next subsection, these five models are used to test with some random sample data to give a conclusion about why the Decision tree is one of the best options for the system.

6.2.2 Real Data Testing

In this part, a self-recording video was prepared for checking the level of accuracy of each model. The video with a length of ninety seconds included 2155 frames. Each model was tested with the video to extract the confusion matrix. The table below illustrates the figures about the accuracy and the difference between test-validated data and some random real data.

Overall, it can be observed that the highest accuracy belongs to Decision tree models, while the figures for Gaussian Naive Bayes, and Logistic Regression are the worst accuracy in this situation.

In this case, the Decision tree increased the accuracy to 93.31%, higher by around 3 percentage points than the last round. XGBoost also remains as one of the best models to use alongside K Neigbores models, 92.75% and 93.27% respectively. The amount of False Negative and False positive cases of these three models can be acceptable, with

Model	TP	FP	FN	TN	Accuracy
Decision tree	969	62	82	1042	93.31%
Support-Vector Machine	954	77	227	897	85.89%
XGboost	914	117	90	1034	90.39%
Logistic Regression	977	54	455	669	76.38%
K Neighbors	920	111	180	944	86.49%
Random Forrest	905	126	82	1042	90.34%
Gaussian Naive Bayes	931	100	638	486	65.75%

Table 6.2. *Models' performance with real data.*

only 62 and 82 for Decision tree models. Regarding the poorest performance Gaussian Naive Bayes, Logistic Regression had a very high number of False Negatives 638 and 455 cases. Therefore, the lowest accuracy model is eliminated, but only above 90% taken into account, so the support-Vector machine and K Neighbors are also removed.

However, since the solution is deployed for automotive, another condition that needs to be taken into account is the running speed in the embedded environment. In the next sections, brief information about the speed tests is provided. Furthermore, appendix A sections contain figures visualizing the confusion matrix results on both validated and random video data — used to evaluate the performance of each model. These figures provide a more detailed picture of the results in color.

6.3 Testing Performance and Analysis

The frame rate is used to measure the running speed of an image-processing application. Since most of the models are deployed under different circumstances like hardware, the performance of each model has a distinct frame rate and memory consumption. Section 6.3.1 will introduce how the frame rate has been improved under parallel situations and how the decision tree can adapt to the embedded environment.

6.3.1 Frame Rate

The decision tree was evaluated in 60 seconds using the technique described in section 5.4.2 to compute the frame rate under sequential and parallel execution. When comparing the two scenarios, it is clear that the parallel environment produces a greater frame rate. The difference in frame rate is due to the fact that the software does not need to wait until the end of each frame, but the new frame loads immediately once the driver's stance is identified, as explained in section 5.4.2.

As can be seen from the diagram 6.3, when the program reaches the stable stage, the

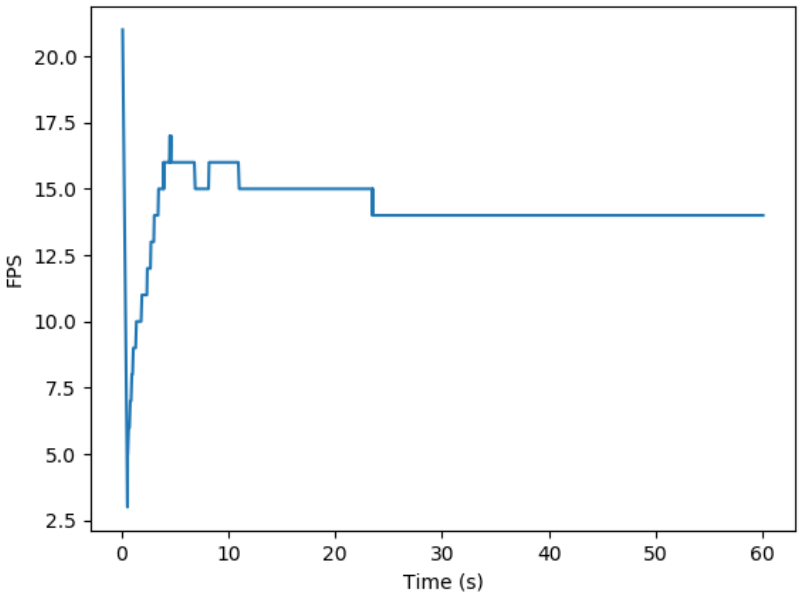


Figure 6.3. *Frame rate of the decision tree model in parallel approach.*

frame rate was around 14 FPS. This rate was acceptable because the gap between each frame is around 0.04 seconds. In the sequential process - diagram 6.4, the frame rate was decreased to 6 FPS at the stable stage. To obtain the result, the program took approximately 1 second to estimate the status of the driver, which means the possible loss of data can happen and it cannot cover every action of the drivers. Therefore, the sequential process is considered as not adapting to the real-time requirements.

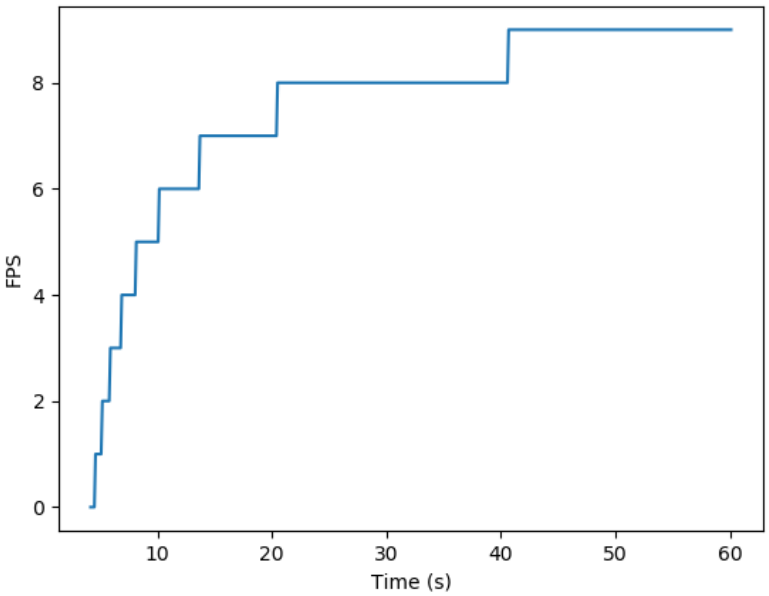


Figure 6.4. *Frame rate of the decision tree model in a sequential approach.*

The frame rate of the Decision tree is considered one of the categories to choose the model as the main part of the system. From the two diagrams in Appendix B, the frame rate of XGBoost and Random Forest model is around 13 FPS (Figure B.1) and 7 FPS (Figure B.2) respectively. Despite the accuracy of Random Forest being as good as Decision Tree, its implementation is not suitable for embedded systems because the optimization for running in embedded devices of this model type is not quite good. According to the sklearn library, [52], the implementation of all the models is deployed for CPU, not GPU. Therefore the structure of each model is the factor that affects the performance. Appendix B contains diagrams that represent the frame rate of the other models. These figures provide a more detailed performance of the program when running with different models.

6.3.2 Memory Consumption

The measurements of memory consumption were conducted using the Memory Profiler module, which is an alternate method for tracking the memory usage of the models.

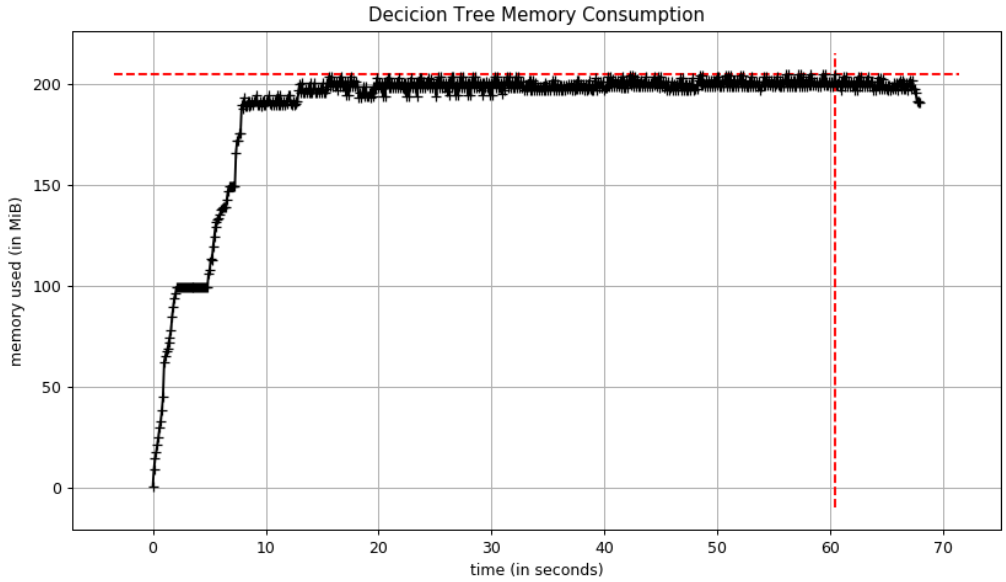


Figure 6.5. *Memory consumption diagram of the decision tree model.*

The maximum memory usage of the software was around 205 MiB and was constant at around 198 MiB. The XGBoost and Random Forest models, on the other hand, require approximately 210 and 230 MiB (Figure B.3 and Figure B.4), respectively, which is slightly more than the decision tree. Differences in implementation can explain the discrepancy between models. In other words, due to its higher frame rate and lower memory utilization, the decision tree has been shown to be suitable for the work.

6.3.3 Latency

The latency is measured from the time that a frame is imported to the detecting pose stage to the time that the system alerts the driver. The chart 6.6 gives information about the latency of the program. The peak of the latency is approximately 15ms. The reason for this sudden increase is that camera needs more time to adjust the resolution to view the driver clearly. Overall, to run as a real-time application, the response of each thread performance can be considered acceptable.

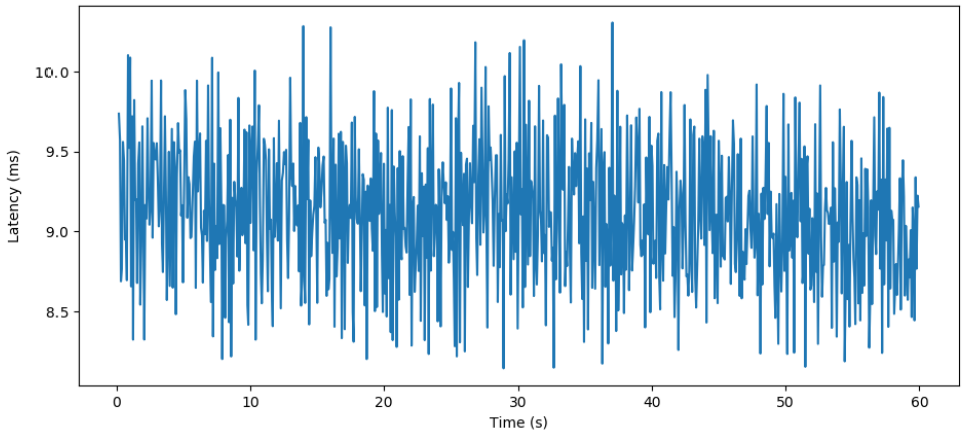


Figure 6.6. *Latency diagram of the program.*

6.4 Comparison

As discussed in Chapter 2, the solutions based on eye expressions, [18], [19], and [20], only focus on drowsiness and this appears to be insufficient to cover the entire cabin of the car, where many actions of the driver can occur. The Convolutional Neural Network system of Bhakti Baheti et al. [21] can process at 42 FPS on an NVIDIA P5000 GPU with 16GB RAM. However, this solution is restricted to specific categories, it is unreasonable to equip a large device in the car system which can lead to a hard change in hardware; this may be seen as a disadvantage since if the drivers perform any activity for which the model has not been trained, a false alarm may arise. Furthermore, despite the fact that Erik Murphy-Chutorian et al. [36] have a method for estimating head posture, the authors focus solely on napping habits. In these kinds of circumstances, the developed approach alerts the driver by triggering an alert if he or she shuts one or both of his or her eyes for a long period of time. However, these works only focus on one aspect of distraction, drowsiness in particular, and this seems not enough to cover the whole cabin of the car, where a lot of actions of the driver can happen.

In conclusion, this thesis has the potential to cover possible poses of the driver in most of the circumstances that can happen in the car cabin because the system's goal is to

extract the status from the position and head direction, not the other categories. With an accuracy of more than 90%, the model may ensure that the driver is notified if he or she is becoming distracted. Another aspect worth mentioning is that the other initiatives listed have not been tested in the embedded system, which is the working environment of most contemporary cars. The thesis provides the opportunity to run a machine learning model in this type of environment to adapt to real-world constraints such as real-time and mobility. Currently, the model can attain 14 to 17 FPS and consume around 200 MiB on NVIDIA Jetson Nano, the lowest model in Jetson Family specifications [63]. As a result, the model can be considered adequate for use in an embedded setting.

6.5 Functional Testing in Virtual Environment

CiThruS2 (See-Through Sight, version 2) is a simulation framework that is a digital 3D city twin of Hervanta, Finland. The CiThruS simulation framework offers a new and realistic virtual environment for developing vision-based ADAS. It includes a 3D urban scene that has authentic traffic infrastructure and driving conditions, with various time-of-day, weather, and lighting effects. Users can generate different traffic scenarios with a large number of autonomous vehicles and pedestrians who can follow specific traffic regulations. The CiThruS Traffic Simulator was a component of the CiThruS system. It is being developed with the Unreal engine. This portable simulator is intended for 360-degree traffic imagery at various locations throughout the city. [64].

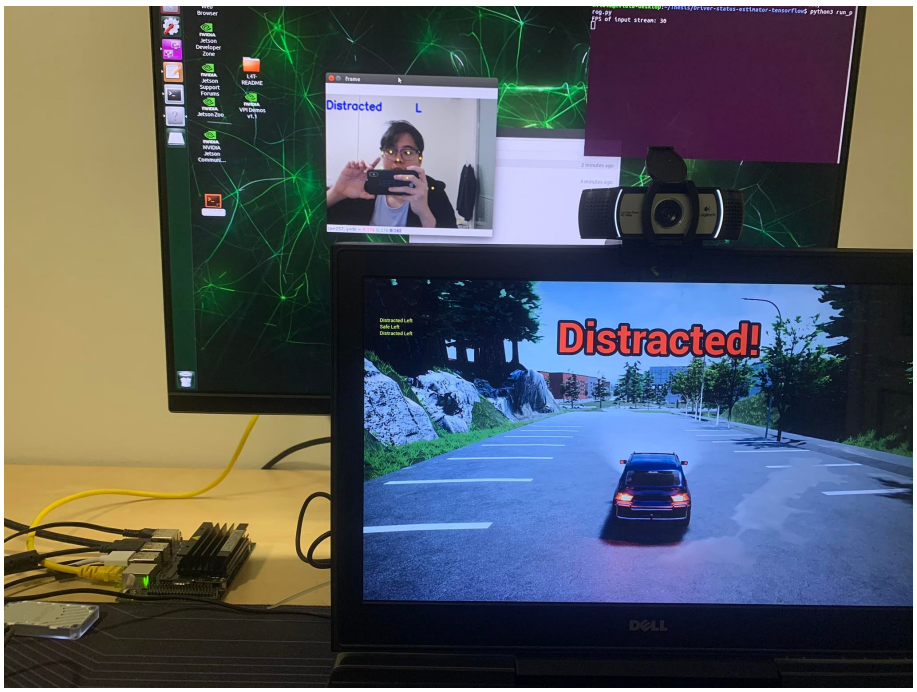


Figure 6.7. *Interface with CiThruS Traffic Simulator. [64]*

As an extension of the application, the distracted estimator connects to the virtual world by using an Ethernet cable. Instead of connecting to the warning module, the Nvidia Jetson

board connects directly to the laptop or PC running CiThruS. If the driver is distracted, the warning screen shows up. This application can benefit new drivers that want to learn how to manage a car before controlling a real one.

7. CONCLUSION

This thesis provided an outline of the real-time warning system which is employed when drivers are involved in traffic and get distracted. First, the fundamental principles of the distracted estimator, notably the decision tree, were introduced, followed by the real-time implementation of parallelism and the warning system. Basic information regarding the PoseNet, which is a key component of this end-to-end model, was also provided. The thesis then examined the model's performance as well as its accuracy, briefly introducing the testing procedure and analyzing the decision tree's advantage over other machine learning models. Section 7.1 of this last chapter discusses the main contribution of this study and summarizes the main achievement as well as the thesis's weaknesses, and Section 7.2 discusses future work and alternative development options for the solution.

7.1 Main Contribution

Overall, the following components were successfully developed for this thesis:

1. Pose prediction pipeline.
2. Implementation of a parallelized pipeline.
3. A decision tree is used to assess the distracted state of the drivers.
4. A socket interface is used to broadcast the status.
5. Module warning box model.

The main contributions of this work involved implementing the quality of the classifier model and its performance in an embedded environment. Decision trees are essential in this system since they are in charge of identifying whenever the driver is in an unsafe position. This thesis also demonstrated how to prepare the data and build up the model's characteristics. Furthermore, assessment methods were extremely significant because they are strongly linked to decision trees. The process of developing a decision tree model was guided by evaluation methodologies. Moreover, the speed of the system was the main concern because it is an extension device that must be placed inside the embedded system of a car, so it must be fast enough to adapt to the real-time requirement.

The decision tree was evaluated using verified and actual data in the experiment part to see whether the model is ready or not. As a result, the major purpose was to investi-

gate the impacts of assessment methods on decision tree accuracy and undertake model tweaking. To begin with, cross-validation approached outperformed the other models. Second, the tree's split and depth might have a substantial impact on its performance. Furthermore, a testing section concerning the accuracy, speed, and memory consumption was structured to present an overall picture of each model's performance to demonstrate clearly why the decision tree was chosen to be a part of this system. The decision tree has shown that it has the best performance to run in the embedded system - the NVIDIA Jetson Nano board.

Another main point of this thesis is optimizing the workflow of the program. By utilizing the threading concept, the program pipelined according to the "Customer-Producer" procedure. In this way, the dependencies of each thread can be recognized and performed in a parallel fashion. Compared to the serial process, the program can process approximately 14 FPS to 17 FPS while serial fashion can only produce 6 FPS to 7 FPS.

7.2 Future Work

Despite the fact that the frame rate of this version is adequate, the model tree could perform faster if the API module is constructed from the source in GPU and delivered to the embedded architecture. Currently, the module's API is integrated into the CPU, which is not very fast. Moreover, Implementing the model with CUDA support might also be beneficial in terms of reducing processing time. [65].

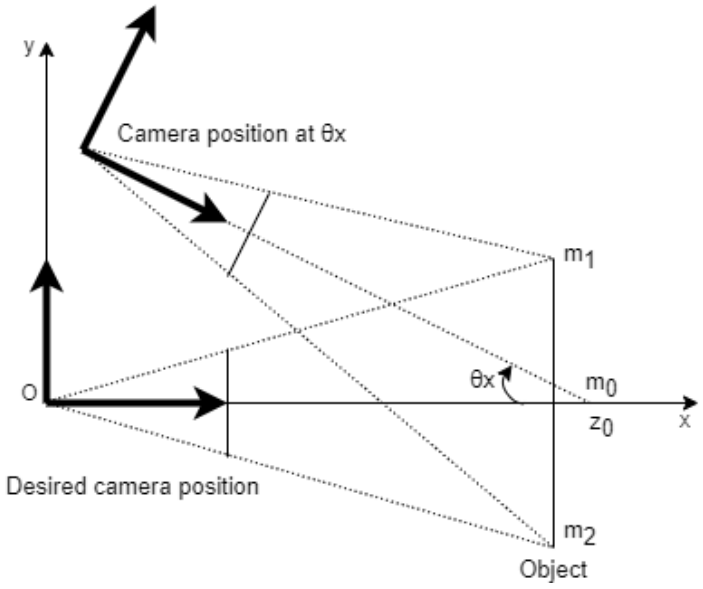


Figure 7.1. *The impact of the camera's position.*

Regarding the solution's weakness, the percentage of accuracy level is highest only when the camera is in the correct position, directly facing the driver. If the camera is positioned too high or too low, the location of each component on the picture is offset, resulting in a change in pixel value.

Figure 7.1 illustrate the position of the camera relative to an object can affect the way the object is projected in an image or video. This is because the camera's position and angle can distort the perspective of the object, leading to differences in shape, size, and position in the final image. According to Christophe Collewet and François Chaumette in 2009 [66], the object's location will shift by a θ degree, which represents the orientation error with regard to the target position. As a result, the distance between sections affects the model tree's accuracy level. Improving the richness of the dataset might be one solution to this challenge. Because the dataset only comprises face-to-face images, integrating additional various camera angles might aid in the resolution of the problem.

Finally, while the proposal is simply a functioning prototype, the concept might be expanded to be installed in a real automobile. The system might become a black box that records the path of the car as well as the behaviors of the driver. This may enhance the driving experience.

REFERENCES

[1] WHO. *Road traffic injuries*. URL: https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries.

[2] T. Stewart. *Overview of motor vehicle crashes in 2020 (Report No. DOT HS 813 266)*. National Highway Traffic Safety Administration. 2022.

[3] M. INSIDER. *What is an Electronic Control Unit?* Apr. 2020.

[4] A. Nagaruru. *Understanding Controlled Area Network (CAN) for Automobile Industry needs*. [Online; accessed 2022-08-11]. June 2020.

[5] W. Gonzalez. *Council Post: Three Ways AI Is Impacting The Automobile Industry*. [Online; accessed 2022-08-09]. Apr. 2022.

[6] Coral. *Dev Board*. URL: https://coral.ai/products/accelerator.

[7] M. Tyka, S. Main, J. Wong, and N. Vu. *GitHub - google-coral/project-posenet: Human Pose Detection on EdgeTPU*. 2021. URL: https://github.com/google-coral/project-posenet.

[8] E. Commission. *Advanced driver assistance systems*. Nov. 2016. URL: https://road-safety.transport.ec.europa.eu/system/files/2021-07/ersosynthesis2016-adas15_en.pdf.

[9] A. Kashevnik, R. Shchedrin, C. Kaiser, and A. Stocker. "Driver Distraction Detection Methods: A Literature Review and Framework". In: *IEEE Access* 9 (2021), pp. 60063–60076. DOI: 10.1109/ACCESS.2021.3073599.

[10] C. Tran and M. M. Trivedi. "Driver assistance for "Keeping Hands on the Wheel and Eyes on the Road"". In: (2009), pp. 97–101. DOI: 10.1109/ICVES.2009.5400235.

[11] C. Yan, B. Zhang, and F. Coenen. "Driving posture recognition by convolutional neural networks". In: *2015 11th International Conference on Natural Computation (ICNC)*. 2015, pp. 680–685. DOI: 10.1109/ICNC.2015.7378072.

[12] L. Jin, Q. Niu, H. Hou, H. Xian, Y. Wang, and D. Shi. "Driver Cognitive Distraction Detection Using Driving Performance Measures". In: *Discrete Dynamics in Nature and Society* 2012 (Nov. 2012). DOI: 10.1155/2012/432634.

[13] Y. Liang, M. L. Reyes, and J. D. Lee. "Real-Time Detection of Driver Cognitive Distraction Using Support Vector Machines". In: *IEEE Transactions on Intelligent Transportation Systems* 8.2 (2007), pp. 340–350. DOI: 10.1109/TITS.2007.895298.

[14] M. Hayhoe. "Advances in Relating Eye Movements and Cognition". In: *Infancy* 6 (Sept. 2004), pp. 267–274. DOI: 10.1207/s15327078in0602_7.

[15] A. Azman, Q. Meng, and E. Edirisinghe. "Non intrusive physiological measurement for driver cognitive distraction detection: Eye and mouth movements". In:

2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE). Vol. 3. 2010, pp. V3-595-V3–599. DOI: 10.1109/ICACTE.2010.5579547.

[16] T. Victor. “Keeping Eye and Mind on the Road”. In: (Jan. 2005).

[17] A. M. Malla, P. R. Davidson, P. J. Bones, R. Green, and R. D. Jones. “Automated video-based measurement of eye closure for detecting behavioral microsleep”. In: *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*. 2010, pp. 6741–6744. DOI: 10.1109/IEMBS.2010.5626013.

[18] B. Cyganek and S. Gruszczyński. “Hybrid computer vision system for drivers’ eye recognition and fatigue monitoring”. In: *Neurocomputing* 126 (2014). Recent trends in Intelligent Data Analysis Online Data Processing, pp. 78–94. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2013.01.048. URL: https://www.sciencedirect.com/science/article/pii/S0925231213006863.

[19] A. B. Albu, B. Widsten, T. Wang, J. Lan, and J. Mah. “A computer vision-based system for real-time detection of sleep onset in fatigued drivers”. In: *2008 IEEE Intelligent Vehicles Symposium*. 2008, pp. 25–30. DOI: 10.1109/IVS.2008.4621133.

[20] X. Sun, L. Xu, and J. Yang. “Driver fatigue alarm based on eye detection and gaze estimation”. In: *MIPPR 2007: Automatic Target Recognition and Image Analysis; and Multispectral Image Acquisition*. Ed. by T. Zhang, C. A. Nardell, D. D. Smith, H. Lu, T. Zhang, C. A. Nardell, and H. Lu. Vol. 6786. International Society for Optics and Photonics. SPIE, 2007, p. 678612. DOI: 10.1117/12.747671. URL: https://doi.org/10.1117/12.747671.

[21] B. Baheti, S. Gajre, and S. Talbar. “Detection of Distracted Driver Using Convolutional Neural Network”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2018, pp. 1145–11456. DOI: 10.1109/CVPRW.2018.00150.

[22] M. Venturelli, G. Borghi, R. Vezzani, and R. Cucchiara. “Deep Head Pose Estimation from Depth Data for In-Car Automotive Applications”. In: May 2018, pp. 74–85. ISBN: 978-3-319-91862-4. DOI: 10.1007/978-3-319-91863-1_6.

[23] M. Venturelli, G. Borghi, R. Vezzani, and R. Cucchiara. “From Depth Data to Head Pose Estimation: A Siamese Approach”. In: Jan. 2017, pp. 194–201. DOI: 10.5220/0006104501940201.

[24] B. Crabbe, A. Paiement, S. Hannuna, and M. Mirmehdi. “Skeleton-Free Body Pose Estimation from Depth Images for Movement Analysis”. In: *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*. 2015, pp. 312–320. DOI: 10.1109/ICCVW.2015.49.

[25] S. Ji, W. Xu, M. Yang, and K. Yu. “3D Convolutional Neural Networks for Human Action Recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.1 (2013), pp. 221–231. DOI: 10.1109/TPAMI.2012.59.

[26] A. Doumanoglou, V. Balntas, R. Kouskouridas, and T.-K. Kim. “Siamese Regression Networks with Efficient mid-level Feature Extraction for 3D Object Pose Estimation”. In: (July 2016).

[27] Y. Sun, X. Wang, and X. Tang. “Deep Learning Face Representation by Joint Identification-Verification”. In: *Proc. NIPS* 27 (June 2014).

[28] G. Fanelli, J. Gall, and L. Van Gool. “Real time head pose estimation with random regression forests”. In: *CVPR 2011*. 2011, pp. 617–624. DOI: 10.1109/CVPR.2011.5995458.

[29] P. Padeleris, X. Zabulis, and A. A. Argyros. “Head pose estimation on depth data based on Particle Swarm Optimization”. In: *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 2012, pp. 42–49. DOI: 10.1109/CVPRW.2012.6239236.

[30] S. Malassiotis and M. G. Strintzis. “Robust real-time 3D head pose estimation from range data”. In: *Pattern Recognition* 38.8 (2005), pp. 1153–1165. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2004.11.020. URL: https://www.sciencedirect.com/science/article/pii/S0031320305000063.

[31] J. Chen, J. Wu, K. Richter, J. Konrad, and P. Ishwar. “Estimating head pose orientation using extremely low resolution images”. In: *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*. 2016, pp. 65–68. DOI: 10.1109/SSIAI.2016.7459176.

[32] M. D. Breitenstein, D. Kuettel, T. Weise, L. van Gool, and H. Pfister. “Real-time face pose estimation from single range images”. In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587807.

[33] V. Drouard, S. Ba, G. Evangelidis, A. Deleforge, and R. Horaud. “Head pose estimation via probabilistic high-dimensional regression”. In: *2015 IEEE International Conference on Image Processing (ICIP)*. 2015, pp. 4624–4628. DOI: 10.1109/ICIP.2015.7351683.

[34] B. Ahn, J. Park, and I. S. Kweon. “Real-Time Head Orientation from a Monocular Camera Using Deep Neural Network”. In: *Computer Vision – ACCV 2014*. Ed. by D. Cremers, I. Reid, H. Saito, and M.-H. Yang. Cham: Springer International Publishing, 2015, pp. 82–96.

[35] X. Liu, W. Liang, Y. Wang, S. Li, and M. Pei. “3D head pose estimation with convolutional neural network trained on synthetic images”. In: *2016 IEEE International Conference on Image Processing (ICIP)*. 2016, pp. 1289–1293. DOI: 10.1109/ICIP.2016.7532566.

[36] E. Murphy-Chutorian, A. Doshi, and M. M. Trivedi. “Head Pose Estimation for Driver Assistance Systems: A Robust Algorithm and Experimental Evaluation”. In: *2007 IEEE Intelligent Transportation Systems Conference*. 2007, pp. 709–714. DOI: 10.1109/ITSC.2007.4357803.

[37] M. Jones and P. Viola. “Fast multi-view face detection”. In: (Dec. 2003).

[38] A. Toshev and C. Szegedy. *DeepPose: Human pose estimation via Deep Neural Networks*. Aug. 2014. URL: https://arxiv.org/abs/1312.4659.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. *MobileNets: Efficient convolutional neural networks for Mobile Vision Applications*. Apr. 2017. URL: https://arxiv.org/abs/1704.04861.

[40] K. O'Shea and R. Nash. *An introduction to Convolutional Neural Networks*. Dec. 2015. URL: https://arxiv.org/abs/1511.08458.

[41] L. Sifre. *Rigid-motion scattering for Image classification*. Oct. 2014. URL: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf.

[42] D. Oved, I. Alvarado, and A. Gallo. *Real-time human pose estimation in the browser with tensorflow.js*. Sept. 2018. URL: https://medium.com/tensorflow/real-time-human-pose-estimation-in-the-browser-with-tensorflow-js-7dd0bc881cd5.

[43] L. Rokach and O. Maimon. "Decision Trees". In: vol. 6. Jan. 2005, pp. 165–192. DOI: 10.1007/0-387-25465-X_9.

[44] E. Alpaydin. *Introduction to machine learning*. The MIT Press, 2020.

[45] N. S. Chauhan. *Decision tree algorithm, explained*. Feb. 2022. URL: https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html.

[46] M. Kantardzic. *Data Mining: Concepts, models, methods, and algorithms*. John Wiley; Sons Inc., 2020.

[47] W. Y. Loh. *Classification and regression trees*. 2011. URL: https://wires.onlinelibrary.wiley.com/doi/10.1002/widm.8.

[48] J. R. Quinlan. In: *Induction of Decision Trees*. New South Wales Institute of Technology, School of Computing Sciences, 1985, pp. 81–106.

[49] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2011.

[50] N. Yang, T. Li, and J. Song. "Construction of Decision Trees based Entropy and Rough Sets under Tolerance Relation". In: *International Journal of Computational Intelligence Systems* (Oct. 2007). DOI: 10.2991/iske.2007.258.

[51] C. E. Shannon. In: *Prediction and entropy of printed English*. Bell Telephone Laboratories, 1951, pp. 50–64.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[53] T. Hastie, J. Friedman, and R. Tisbshirani. *The elements of Statistical Learning: Data Mining, Inference, and prediction*. Springer, 2017.

[54] *Understanding Shannon's Entropy metric for Information*. Mar. 2014. URL: https://arxiv.org/ftp/arxiv/papers/1405/1405.2061.pdf.

[55] N. Gourier, D. Hall, and J. L. Crowley. "Estimating Face Orientation from Robust Detection of Salient Facial Features". In: *Proceedings of Pointing 2004, ICPR, In-*

ternational Workshop on Visual Observation of Deictic Gestures. Cambridge, UK, 2004.

[56] G. Fanelli, M. Dantone, J. Gall, A. Fossati, and L. Van Gool. “Random Forests for Real Time 3D Face Analysis”. In: *Int. J. Comput. Vision* 101.3 (Feb. 2013), pp. 437–458.

[57] N. Developer. *Jetson Nano Developer Kit*. Sept. 2022. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit?cv=1.

[58] E. Ostrowski and S. Kaufmann. “Survey of alternative hardware for Neural Network computation in the context of Computer Vision”. In: (May 2020).

[59] Coral. *Dev Board*. URL: https://coral.ai/products/dev-board?cv=1.

[60] . Weiss. *Nvidia Jetson Nano and google coral edge TPU-a comparison*. Sept. 2019. URL: https://medium.com/3dvisionlabs/nvidia-jetson-nano-and-google-coral-edge-tpu-a-comparison-b244a176d0e0.

[61] Z. West. *Network sockets: The bridges between end systems*. July 2021. URL: https://www.alpharithms.com/network-sockets-250910/.

[62] P. S. Foundation. *threading — Thread-based parallelism*. Aug. 2021. URL: https://docs.python.org/3/library/threading.html.

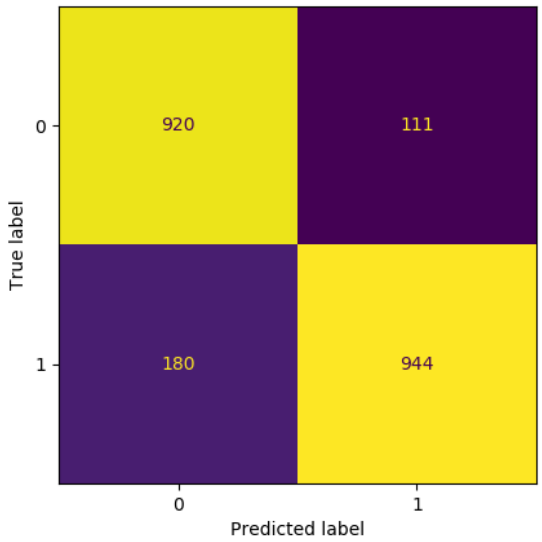
[63] Forecr. *Jetson nbsp;AGX Xavier™ Carrier Board - DSBOARD XV2*. URL: https://www.forecr.io/blogs/all/comparison-of-nvidia-jetson-modules.

[64] U. V. Group. *CiThruS*. URL: http://ultravideo.fi/#cithrus.

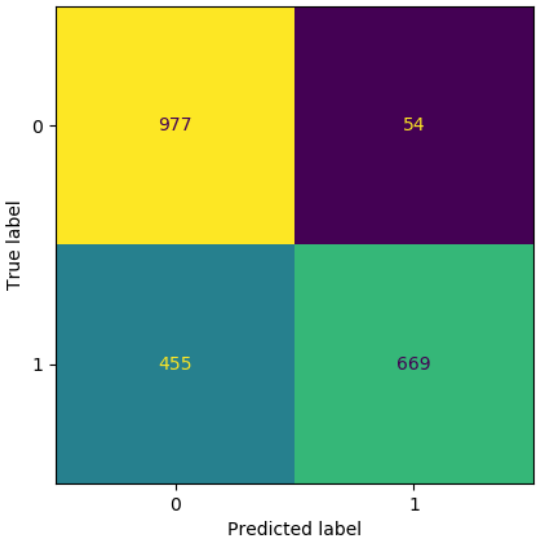
[65] C.-C. Chiu, G.-H. Luo, and S.-M. Yuan. “A decision tree using CUDA GPUs”. In: Dec. 2011, pp. 399–402. DOI: 10.1145/2095536.2095615.

[66] C. Collewet and F. Chaumette. “Positioning a camera with respect to planar objects of unknown shape by coupling 2-D visual servoing and 3-D estimations”. In: *IEEE Trans. Robotics Autom.* 18 (2002), pp. 322–333.

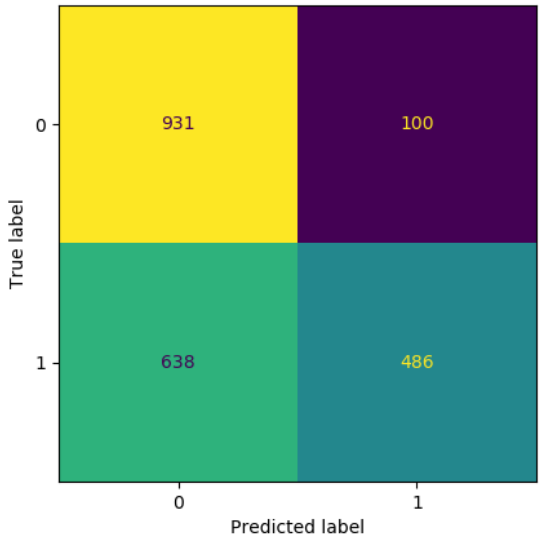
APPENDIX A: ACCURACY - CONFUSION MATRIX



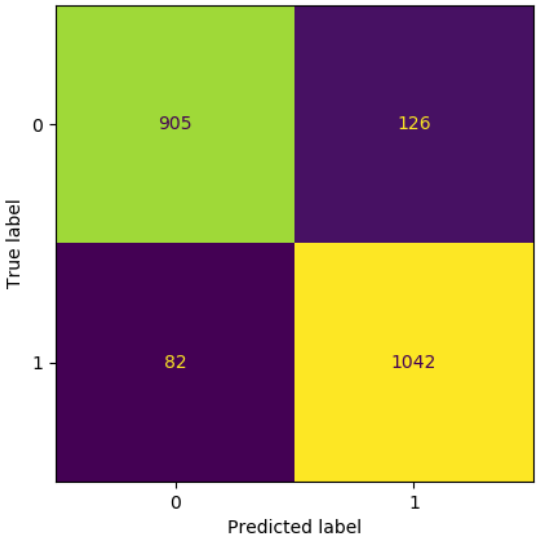
(a) K Neighbors



(b) Logistic Regression



(c) Gaussian Naive Bayes



(d) Random Forrest

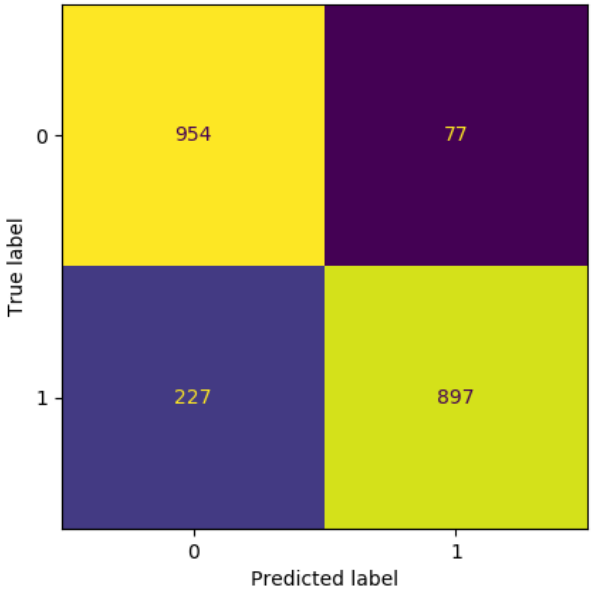


Figure A.1. *Support-Vector Machine model.*

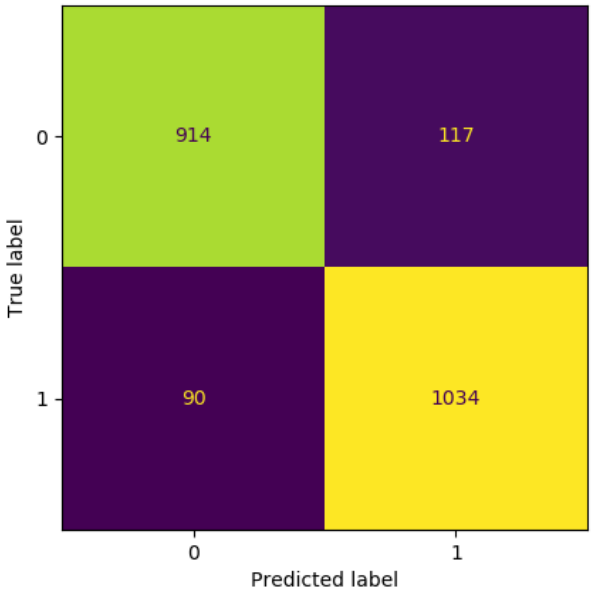


Figure A.2. *XBoost model.*

APPENDIX B: PERFORMANCE ON NVIDIA JETSON

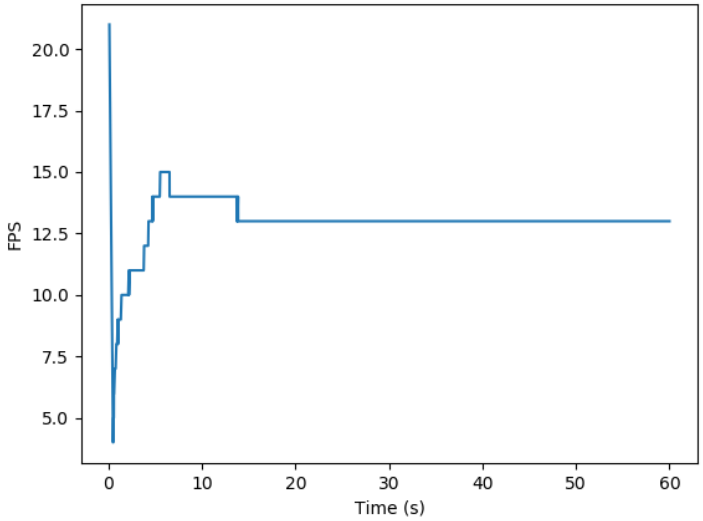


Figure B.1. *Frame per second of the XGBoost model.*

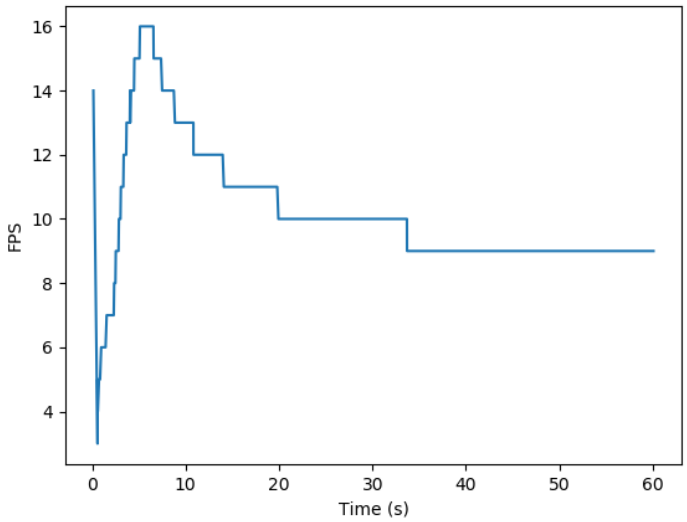


Figure B.2. *Frame per second of the Random Forest model.*

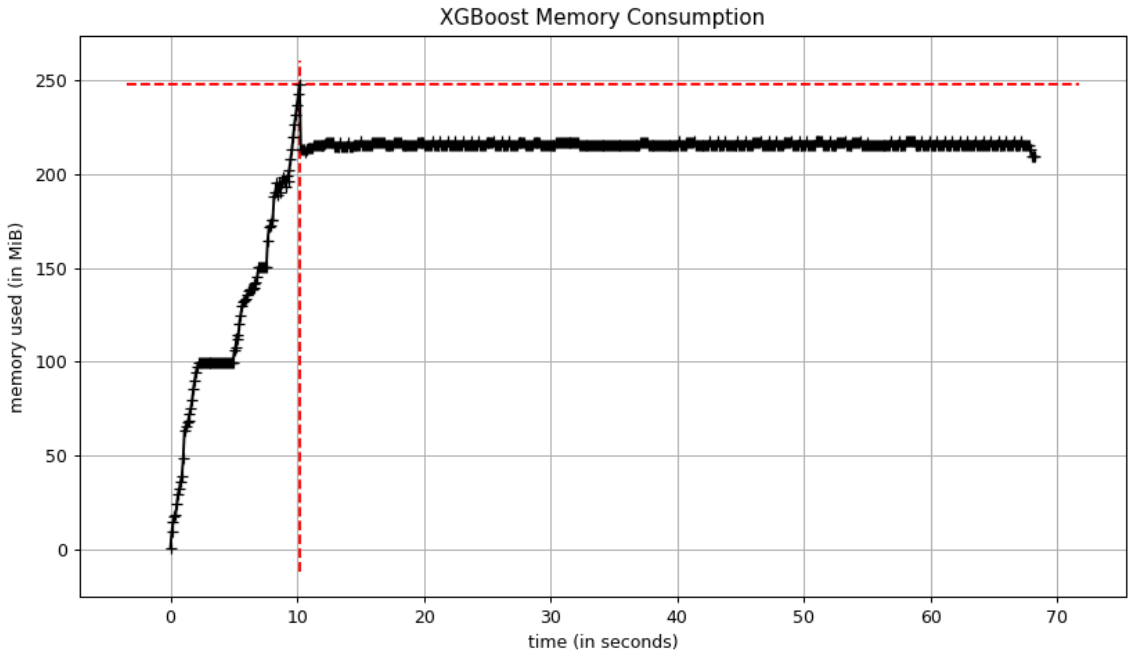


Figure B.3. *Memory consumption diagram of the XGBoost model.*

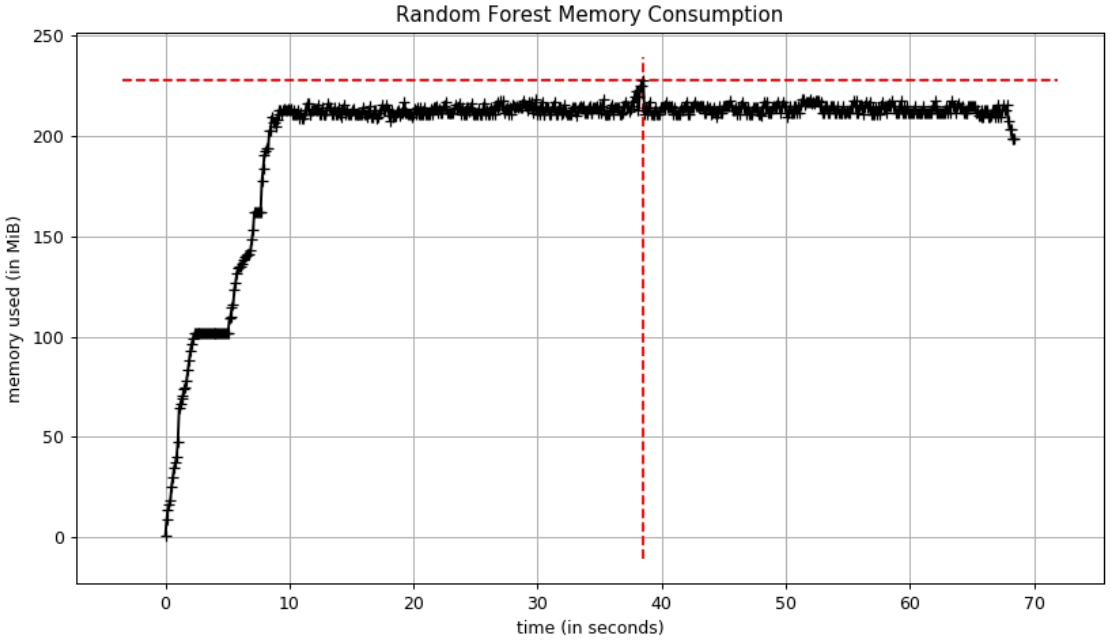


Figure B.4. *Memory consumption diagram of the Random Forest model.*