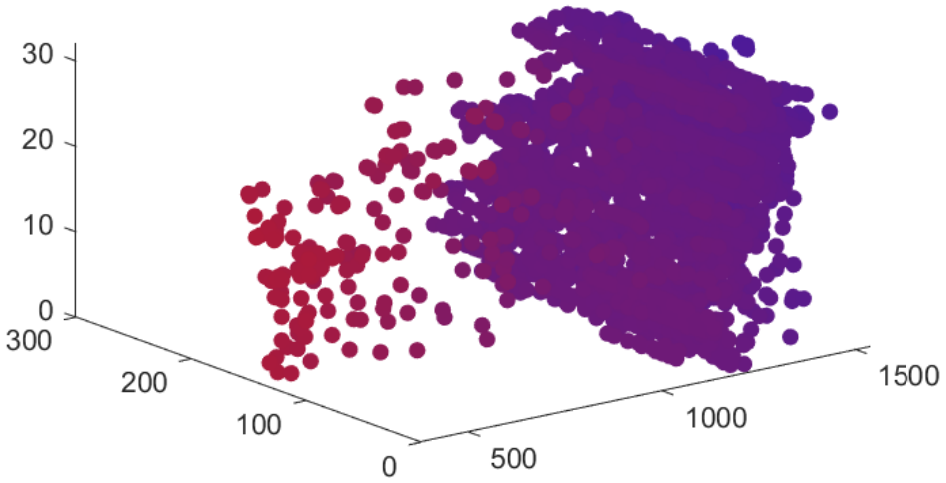


Exploring the feasibility of using ultrasonic sensors and cameras for human gesture recognition to activate trunk opening in vehicles

Master's thesis in Complex Adaptive Systems and Systems, Control and Mechatronics

Tim Johansson and Krister Mattsson

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Exploring the feasibility of using ultrasonic sensors and cameras for human gesture recognition to activate trunk opening in vehicles

Tim Johansson, Krister Mattsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Some Subject or Technology
Division of Division name
Name of research group (if applicable)
Chalmers University of Technology
Gothenburg, Sweden 2024

Exploring the feasibility of using ultrasonic sensors and cameras for human gesture recognition to activate trunk opening in vehicles
Tim Johansson, Krister Mattsson

Supervisor: Pratish Ray, Volvo Cars Exterior Vision & Ultrasonics
Supervisor: Jonas Fredriksson, Department of Electrical Engineering
Examiner: Jonas Fredriksson, Department of Electrical Engineering

Master's Thesis 2024
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of the general simplified logic where all networks are shown as boxes with an input and output signal. The yellow arrow illustrates the initiation of the time window used for the USS model. Both the vision model and the USS model classification outputs are weighed using a factor α to determine the total classification output.

Typeset in LaTeX
Gothenburg, Sweden 2024

Exploring the feasibility of using ultrasonic sensors and cameras for human gesture recognition to activate trunk opening in vehicles
Tim Johansson, Krister Mattsson
Department of Electrical Engineering
Chalmers University of Technology

Abstract

The integration of new advanced technologies plays a crucial role in the industrial market. The automotive industry is no different. With the introduction of ultrasonic parking sensors and high-resolution cameras in new vehicles combined with the integration of high-performance computing power, it is possible to implement machine learning and classical methods to process real-time sensor information. This thesis focuses on recognizing human gestures using the combined information from the ultrasonic sensors and visual camera data for functional actuation. In particular, the thesis serves as a feasibility study for using gesture recognition as an input for activating the automatic opening of the trunk.

Several approaches to this problem have been investigated through literature studies, and the most suitable method has been determined to be a combination of machine learning neural networks and sensor fusion from classical methods. Two different machine learning methods are implemented and analyzed for the visual input. One model that classifies static images and one model that classifies a series of images to capture information from dynamic movement. Another model is built for the parking sensory input, which, similarly to the previous model, utilized a series of measurements in time for the classification. Together, these models form a logical pipeline that utilizes classical ultrasonic sensory input as an indicator for activating the models. These models are evaluated for both binary outputs, meaning classifying gesture or no gesture, and multi-class gestures, meaning several different gesture classifications.

Separately, the vision models achieved close to perfect test accuracy for both the binary and the multi-class implementations, while the model for the ultrasonic sensors achieved a test accuracy of around 70 %. Using sensor fusion, the combined model achieved perfect test accuracy for both the static implementation and the dynamic, proving the proposed solution's feasibility. However, one should note that the results are all based on a small data pool collected during the thesis. Furthermore, the data lacks diversity. Implementing the solution on a greater scale would likely yield some changes in the results. In conclusion, it is possible to reliably use human gesture recognition for functional actuation from ultrasonic and visual data.

Keywords: Human gesture recognition, machine learning, neural networks and sensor fusion.

Acknowledgements

This thesis was conducted in collaboration with Volvo Cars in Torslanda, Göteborg, within the department of Safe Vehicle Automation. We would like to extend our deepest gratitude to the team members of *USS Enterprise*.

A special thank you goes to Pratish Ray, our supervisor at Volvo, for his unwavering support throughout the project. We are also grateful to Venu Gopal Puripanda and Simon Rudh for their technical support related to test vehicles. We would like to express our appreciation to Khadija Dallah and Srinath Shanmugam for their guidance in decoding recorded data. Finally, we thank Jonas Fredriksson for taking on the roles of supervisor and examiner at Chalmers University of Technology. Your collective expertise, guidance, and support have been invaluable to the success of this project.

Tim Johansson, Krister Mattsson, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

HMR	Human Motion Recognition
PoC	Proof of Concept
USS	Ultra Sonic Sensors
ANN	Artificial Neural Network
NN	Neural Network
CNN	Convolutional Neural Network
ML	Machine Learning
SGCM	Static Gesture Classification Model
TP	True Positives
TN	True Negatives
FP	False Positives
FN	False Negatives

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

i,j,k	Indices in tensors/matrices
t	Index for time step

Parameters

η	Learning rate
λ	Scaling parameter
n_h	Number of hidden layers
n_s	Number of samples in training set
n_i	Number of input neurons
n_o	Number of output neurons

Variables

O_i	Outputs
$g(x)$	General activation function
w_{ij}	Weights
x_j	Nodes
θ_i	Biases
$Q(w)$	Loss function
Δt	Time step (time interval)

Contents

List of Figures

List of Tables

1
Introduction

This chapter presents the project and its background, delimitations, and outline.

1.1 Background

In the automotive industry, the integration of advanced technologies plays a pivotal role in enhancing safety and user experience. With the introduction of autonomous drive and driving aid features, the industry has significantly augmented the deployment of sensors in their vehicles, allowing for an increased perception of their surroundings. Furthermore, advancements in machine learning have paved the way for novel solutions that not only enhance the efficiency of features but also reduce costs for manufacturers as they allow for the possibility of removing previously required sensors. For instance, Tesla replaced their front-facing radar with vision [1].

A feature that has gained recognition across the industry, adding convenience and innovation to the overall user experience, is the radar-based contactless control of the trunk. The radar system is centered underneath the rear bumper and requires a person to approach the center of the rear bumper and perform a 'kick' gesture to activate and open the trunk. The detection range of the radar is limited by its placement, which consequently requires the person to stand close to the rear to activate the trunk, and depending on the model of the vehicle, it can be necessary for the person to inconveniently take a step back to not be in the way of the trunks path. In addition to the user experience challenges, the current implementation of the radar-based system incurs significant costs. The cost associated with the current solution for contactless trunk control through a single-purpose radar system is extensive, considering the intricate integration projects and expenses associated with suppliers-, production-, logistics- and service contracts. According to the function owner, this model has an accuracy of around 96 %.

The automotive industry continuously strives for cost-efficient and innovative solutions. This project focuses on leveraging the existing ultrasonic sensors (USS) and a rear-view fish-eye camera to replace the radar-based system. Such a solution would eliminate the need for these radar sensors, saving all costs associated with material and logistics, which would, in return, reduce the environmental impact.

1.2 Objectives

The main objectives of the project are as follows:

- Determine a suitable approach to detect and classify patterns of human gestures in real-time from echo and vision data.
- Translate meaningful human gestures into inputs for functional actuation.
- Compare model accuracy between the proposed system from the captured data set recorded in this thesis project and the radar-based system.

1.3 Delimitations

This section outlines the boundaries and limitations of the project, ensuring clarity and managing expectations regarding the outcomes.

- In this project, only the available ultrasonic sensors and the camera positioned at the rear of the car are utilized without exploring additional sensor options.
- In terms of the number of gestures considered for the project, the project focuses on a specific set of gestures rather than a comprehensive range to ensure a large dataset due to the limited availability of test cars. Consequently, there is a limited distribution of the performed gestures concerning the number of people performing the gesture and environmental factors like weather.
- The recording sessions are conducted on the company's premises leading to further constraints.
- The project does not address constraints related to car system integration, such as computational load, storage capacity, or system architecture.
- The thesis project exclusively considers the user intention of opening the trunk by one person, without exploring other potential user intentions.
- All development is performed on local workstations to retain confidential and sensitive information and proprietary knowledge. This approach was essential to comply with company policies and ensure data security.

1.4 Ethical and Sustainability aspects

This project utilizes sensors already implemented on the car and will, therefore, have a minimal impact on sustainability and not increase the risk of privacy intrusion any further. The thesis work is purely software-oriented, and the technology is aimed at being used for comfort, accessibility, and simplicity, aimed at functional actuation, such as opening the trunk. No personal information such as name, age, or gender is recorded, ensuring the privacy of the persons participating in the recording session.

1.5 Outline of thesis

Advancements in machine learning have paved the way for novel solutions that enhance feature efficiency and reduce manufacturer costs by potentially eliminating previously required sensors.

In this thesis report, the underlying theory used in this project is presented in the theory chapter, and the methods for classifying gestures are presented in the method chapter. After this, the results obtained using the presented methods are stated and illustrated. Furthermore, the next chapter presents the discussion, results, and potential error sources. Here, the created models for the project are also compared to the current radar-based system. After this, the conclusion is presented, followed by some ideas for future work.

2
Theory

This section introduces the underlying theory used in the project to motivate the method and analyze the results.

2.1 Human motion recognition

Human motion recognition (HMR) involves the processes of identification, classification, and characterization of human movements [2]. In the context of computer vision, HMR is a multidisciplinary field composed of biomechanics, machine vision, image processing, data analytics, nonlinear modeling, and pattern recognition [3]. The development of an efficient HMR system requires it to handle a vast diversity of human features like body size, postures, and appearances, as well as environmental factors like illumination, viewing angles, and disturbances. The complexity of human motion and the variability of recording conditions make HMR challenging, but extensive research has gone into HMR due to its wide range of applications [2]. Each of the applications faces similar primary challenges: interpreting ambiguous poses and actions; varying interpretations of classification; potential partial occlusion of bodies or objects; poor video quality, including blurring and noisy data from low-quality sensors; significant time differences between actions; inadequate or excessive lighting; and difficulty in acquiring large-scale datasets [3]. These challenges necessitate advanced methods to accurately capture and analyze human motion.

HMR can be broadly divided into two categories: vision-based and sensor-based recognition [4]. The vision-based method relies on one or more cameras; the method of approach for reaching motion predictions, therefore, varies significantly depending on the techniques employed and continues to be a field of interest for studies within the topic of HMR. On the other hand, the sensor-based method is a more standard approach and an extensively researched area given the feasibility of attaching sensors or using mobile devices [4].

Recent studies, such as those reviewed in [5], have explored a variety of HMR methods, covering traditional approaches to manually designed motion features extracted from RGB and depth data, as well as modern deep learning-based approaches for motion feature representation, techniques for recognizing human-object interactions, and methods for action detection. Unlike image classification, which primarily focuses on spatial information, vision-based HMR requires the integration of temporal information to accurately capture and analyze motion sequences. The review in [5]

concludes that deep learning-based methods exhibit superior performance in motion feature learning problems as they leverage advanced neural network architectures to learn complex patterns and relationships within the data. In addition, the nature of deep learning-based methods is that they are much more resource-efficient compared to traditional computer vision approaches, [4].

2.2 Deep learning

Deep learning is a subset of machine learning in Artificial Neural Networks (ANN) where hidden layers are introduced to capture complex and intricate patterns in data. As a problem aimed to be solved using artificial neural networks cannot be solved by linear separation only, deep learning models or deep neural networks can approximate more complex patterns of information and have the ability to classify non-linear problems [6]. As mentioned earlier, a deep neural network consists of one or more hidden layers in addition to the input and output layers, see figure 2.1. The hidden layers are pivotal in an ANN's capture of complex classification patterns. For nonlinear classification or complex data patterns, the ability to handle these types of intricate data patterns by separating information becomes necessary. Each hidden layer in the network will contribute to and make a more complex classification possible, but it will also add more parameters to tune. The extra size and parameters also mean that a deep learning model often requires large datasets for all weights and biases to be tuned in a desirable way [7]. A common approximation measure

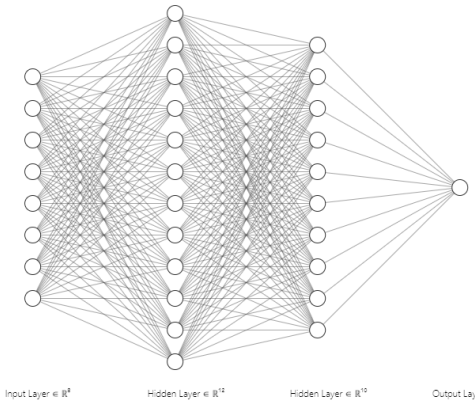


Figure 2.1: Illustration of a deep neural network consisting of an input layer, two hidden layers and a singular output.

for determining a reasonable amount of hidden layers according to [8] is:

$$n_h = \frac{n_s}{\lambda(n_i + n_o)}, \tag{2.1}$$

where n_h is the number of hidden layers, n_s the number of samples in the training set, n_i number of input neurons, n_o number of output neurons, and λ is a constant which is usually in the range of 2-10.

2.2.1 Activation functions

An activation function within the field of ANNs is a mathematical function that converts the output of each network layer to some binary value type, ranging from positive and negative numbers to specific integers, depending on the network specifications. Activation functions can be of different forms. Two commonly used functions are $\tanh(b)$ and $\text{sgn}(b)$, where b is the neuron states, weights and biases of the current layer. There is one distinct difference: $\tanh(b)$ is continuous while $\text{sgn}(b)$ is not. This detail becomes important when the networks are trained, as it is relatively common to use training algorithms, such as backpropagation, which utilizes the activation function's derivative. It is also important to note that when the activation function is continuous, the states of the neurons also become continuous. Another activation function that is commonly used in image classification networks and CNNs is the Rectified Linear Unit (ReLU) function. ReLU is a linear non-negative activation function. One of the key advantages of ReLU is its non-saturating property, which further mitigates the phenomenon of vanishing gradient [9].

2.2.2 Convolutional neural networks

A convolutional neural network (CNN) is a deep learning model designed and mainly used for processing visual data. More specifically, CNNs are well suited for tasks such as image classification and object detection within images or videos. CNNs include convolutional layers, where each layer applies filters or kernels to the input data. The kernels or filters are used to detect features and patterns within the visual data. Pooling layers are a common way to downsize the spatial dimensions after a convolution layer. This reduces the computational resources necessary for training and using the network. At the end of the network, after the convolutions and pooling layers, CNNs typically have one or more fully connected layers connecting the last layer with the output. This part performs a high-level feature extraction from the last convolution and connects it to the output.

CNNs use supervised learning, or in other words, they need labeled datasets for training, which furthers the need for good-quality datasets. Backpropagation is commonly used for training. The network weights and thresholds are adjusted to minimize the difference between the labeled targets and the network output [10].

2.2.3 CNN-architecture, Residual Network

Over the past decade, extensive research of CNN architectures has taken place, leading to the successive development of AlexNet, GoogleNet, ResNet, and DensNet, to name a few. Each of these architectures has significant contributions to the development and performance of deep learning models, particularly in the field of image recognition [11], with unique approaches to address some of the common issues in deep learning like for instance vanishing gradients, etc.

To address the commonly encountered vanishing gradient problem, Residual Net-

works (ResNets) are purposely designed architectures to counter the issues with the use of so-called skip connections [9]. As neural networks become deeper, the gradients used in backpropagation can become very small as a consequence of both the chain rule and the selection of activation functions of saturating nature, such as $\tanh(b)$, leading to slower and even stalled learning during the training process. The key element in ResNets is the Residual block, shown in figure 2.2.

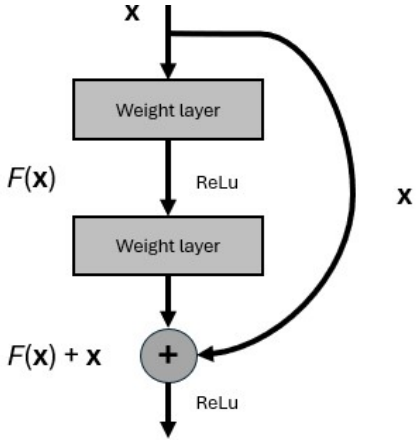


Figure 2.2: This figure illustrates the Residual block.

By introducing skip connections, where the input to a layer is added directly to the output of a subsequent layer, the gradients are less likely to diminish to insignificantly small values as they pass through each layer of the network [9]. If the desired underlying mapping is denoted as $H(\mathbf{x})$, ResNets reformulate the learning task to instead model the residual function $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$ and subsequently the original function becomes $H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$. Residual blocks will commonly include two or more convolution layers, batch normalization, and ReLU activation functions [9]. ResNets have been shown to achieve remarkable performance and significantly outperform traditional CNN architectures in terms of both accuracy and depth on various image recognition tasks [11]. Using residual blocks effectively allows the network to preserve the essential features learned in earlier layers.

2.2.4 Spatial-temporal data and deep learning models

The temporal dimension is crucial in capturing the dynamics of motion over time, adding complexity to the task of HMR and making it more informative. In HMR, the focus on deep learning techniques and the processing of RGB video data has greatly increased since 2015 [3]. Various methods, including deep learning architectures based on CNN, Recurrent Neural Networks (RNN), and hybrid approaches, have undergone comprehensive analysis of their advantages and limitations [5, 12, 3, 4, 13].

Different architectures for handling spatial-temporal data in HMR have been explored. One approach is the 3D CNNs, where the third dimension can be viewed as the time axis. These networks build upon the architecture of 2D CNNs by adding an extra dimension to the input, allowing for the processing of temporal information

for several frames in a video sequence. Another approach is the hybrid method, which combines different types of neural networks to handle both spatial and temporal features. For example, CNN-RNN architectures utilize ResNet to extract spatial features and RNNs to extract temporal features. While 2D CNN-based architectures excel in spatial data handling, they cannot capture temporal features effectively. The limitation can be addressed by including algorithms such as optical flow, Long Short-Term Memory (LSTM), which handle sequential data and capture temporal dependencies effectively [4], and temporal grouping [3]. An alternative strategy is that of stream networks, meaning that types of inputs are handled on different networks. For instance, processing RGB frames in the first stream and optical flow in the second stream [3]. This approach allows for the capture of both spatial and temporal information.

Interestingly, despite the disadvantage of ordinary 2D CNNs being applied to individual frames and therefore cannot model temporal information, they perform remarkably well in some instances, such as the Sport-1M benchmark [13]. Nevertheless, 3D CNNs are still vastly outperforming 2D CNNs on large datasets [13]. A more specific example, [3] evaluates a 3D ResNet of depth 50 and a 2D vision transformer (ViT) with a long short-term memory network (LSTM) on the human motion database (HMDB51). It was shown that the 3D ResNet outperformed the ViT with LSTM, reaching accuracy scores in the train and test phases of 96.7 $\pm$ 0.35% and 41.0 $\pm$ 0.27%, respectively.

3D CNNs continue to be an explored topic within HMR [13]. An attractive feature of 3D CNNs, compared to the two-stream method, is that the architecture creates the hierarchy and relationship between spatial and temporal features without the need for other information like optical flows [13]. Furthermore, 3D CNNs are known as end-to-end networks as the input processing and generation of output do not require any additional step sequences. However, a significant disadvantage of 3D CNNs compared to 2D CNNs is their high parameter count, which is an order of magnitude greater, leading to a higher risk of overfitting, thereby requiring a large volume of data like Kinetics [3].

In [13], several spatial-temporal architectural models based on 3D CNNs, two-stream networks, and ResNets are studied with regard to their performance on HMR. In particular, architectures such as 2D convolutions over frames, 2D convolutions over video clips, alternating 3D-2D convolutions, and factorization of 3D convolution into a 2D spatial convolution followed by 1D temporal convolution have been investigated. The residual 2D plus 1D CNN architecture R(2+1)D, stems from the factorization of the N_i 3D spatiotemporal convolution of size $N_{i-1} \times t \times d \times t$ into M_i 2D spatial convolution filters of size $M_{i-1} \times 1 \times d \times t$ and N_i 1D temporal convolution filters of size $M_i \times t \times 1 \times 1$. The hyperparameter M_i defines the number of dimensions in the intermediate spaces where the signal is mapped during the transition between spatial and temporal convolutions [13]. To effectively maintains a similar number of parameters as a 3D convolution block [13], M_i is chosen according to:

$$M_i = \frac{td^2Ni - 1N_i}{d^2Ni - 1 + tN_i} \tag{2.2}$$

The study in [13] concluded that the R(2+1)D, which is closely related to Pseudo-3D, outperforms the other models and even achieves comparable or superior results of the benchmarking models like Iterative Dichotomiser 3 on datasets of Sports-1M, Kinteics, UCF101, and HMDB51 [13]. The performance gain of the R(2+1)D model can be attributed to the factorization of each spatiotemporal block, leading to consecutive spatial and temporal convolutions across the network with the following positive effects. Firstly, an additional nonlinear rectification is incorporated between the two operations, which effectively doubles the number of nonlinearities with the same number of parameters as in 3D convolutions. Secondly, yielding a lower training and testing loss at the factorization facilitates optimization [13].

2.2.5 Balance of data

In a classification approximation problem, as well as other problems of similar characteristics, when implementing a neural networks approach, it is relevant to look into possible local optima while training. If the majority of the training data for the model is of one type or class, one such local optima can be for the model to classify only one class. The loss will seem rather low, but in reality, the model has just approximated the problem to a constant output from only the data types. To combat this problem, one can balance the dataset so that there are roughly the same amount of data samples for each class or data type. In this way, the network is forced to fit another pattern within the data. Regardless of how often the data types or classes normally occur outside the test environment, the network still needs to be trained on a balanced dataset to avoid an unwanted bias [10].

It is common to split the data into a training set, a validation set, and a test set to avoid training biases in evaluation processes. By using different parts of the dataset, the evaluation will simulate the network in use since it has to handle data that is completely new to it.

2.2.6 Overfitting

When training a neural network such as a CNN, over-training or, in other words, training too much may result in unwanted pattern findings in the training dataset. This also depends on the number of hidden layers within the network, which allows for more complex information classification. The network will adapt to the specific training set trends and patterns, which may be unique for this set. If this happens, the accuracy against the validation set is decreased. Since the network has not been trained on the data from the validation set, its unique unwanted features will not be integrated. Therefore, the overall accuracy will decrease against the validation set. However, a network can reach a local peak in accuracy. It is not certain that

the network is overfitting if the validation accuracy is lowered temporarily, see e.g., [10] and [14].

2.2.7 Transfer learning

In neural networks and machine learning, it can sometimes be useful to use information from pre-trained weights and biases in a smaller scope than the original model. By using a pre-trained model with several classification outputs, one can use these outputs as inputs to a new layer or model where the problem dimensionality is significantly reduced. Essentially, one transfers one network model's knowledge of the domain or area it is trained on to another targeted domain, [15]. This domain could, for instance, be a subset of the original one with a more specific classification. This also means that less data is necessary for training the specific model since the complexity of the problem is already decreased by the pre-trained model [15].

2.2.8 Cross entropy loss

Cross entropy loss is a metric for measuring the performance of classification model networks. Cross-entropy loss quantifies how well the predicted probabilities match the actual class labels. For networks with multiple output classes, the cross entropy loss CE_L can be calculated as

$$CE_L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\ln(p_{ij}), \tag{2.3}$$

where N is the number of data points, C is the number of classes, p_{ij} is the predicted probability of data point i belongs to class j and y_{ij} is a boolean value (either 0 or 1) that indicates if j is the correct class for data point i. y_{ij} is 1 if this is true and 0 if not [10].

2.2.9 Stochastic gradient descent

Stochastic gradient descent (SGD) is a mathematical method for optimizing parameters. The goal is to minimize a loss function. The network model's parameters are updated for each training iteration following an update rule dictated by SGD. First, the dataset is shuffled randomly, then the data is passed through the network. The gradient of the loss function is calculated, and the parameters are updated using:

$$w_k = w_{k-1} - \eta\nabla Q(w_{k-1}), \tag{2.4}$$

where w are the weights, η is the learning rate (a constant that scales the change), and $Q(w)$ is the loss function. The updated form for the thresholds is updated in the same way. This is repeated through the dataset and for every parameter [6].

2.3 Containing information in descaled images

A standard format image, such as .jpg and .png, uses pixels to store information, where the resolution describes the number of pixels used. Descaling such an image,

therefore, means approximating the same image using fewer pixels. It is apparent that an image of high resolution contains a lot of information. The greater the resolution of the image, the more details can be shown and the clearer the image becomes. However, due to hardware limitations and/or runtime optimization, keeping a low image resolution is often preferable. Sometimes, information from the most important details can remain even if the image is scaled down, [16]. How much an image resolution can be descaled to contain relevant information still depends on the content of the image and the purpose of the downsizing. In the case of hardware limitations and computing speed for neural networks, it depends on the network size and the used GPU [17]. A common way to determine this is through iterative testing with different image resolutions.

2.4 Evaluating network models

There are many ways to analyze and evaluate neural network models, and what results are relevant depends on the problem the model is aimed to solve or approximate. In machine learning classification models, measures such as accuracy, precision, and recall are commonly used to help evaluate the quality of the classifications [18].

Accuracy is a measure of how often a model can predict the correct class or outcome. It is calculated using the following equation

$$A = \frac{p_c}{p}, \tag{2.5}$$

where A is the accuracy, p_c the number of correct predictions and p the total number of predictions. In a classification problem that is binary or has only two classes, one can split the prediction outcomes into four possible categories. If we imagine one class being positive and the other negative then,

- True Positive (TP), the model correctly classified positive.
- True Negative (TN), the model correctly classified negative.
- False Positive (FP), the model incorrectly classified positive.
- False Negative (FN), the model incorrectly classified negative.

Using this terminology, accuracy can be written as

$$A = \frac{TP + TN}{TP + TN + FP + FN}, \tag{2.6}$$

i.e., $TP + TN = p_c$ and $TP + TN + FP + FN = p$.

Precision measures how reliably the model classifies true positives. Or, in other words, how often the positive classifications are correct. This measure is calculated using the following equation

$$P = \frac{TP}{TP + FP}, \tag{2.7}$$

where P is the precision, TP is the number of true positive predictions and FP is the number of false predictions.

Recall measures how well the model can classify one class correctly. In other words, recall will measure if the model finds all instances of this class in a given data set. Recall R is calculated as [18]:

$$R = \frac{TP}{TP + FN}, \tag{2.8}$$

To manage a trade-off between P and R, the $F1$ score is used as a harmonic mean of these two metrics, giving a single measure of accuracy. Balancing the two measurements is crucial as the FP and FN should be minimized. The $F1$ score is calculated as

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}, \tag{2.9}$$

which ensures that the score is high only if both P and R are high, making it a robust metric for evaluating the effectiveness of our classification models.

2.4.1 Network certainty

Network certainty measures how decisive the network model acts on each classification. If an ANN classification model has m output nodes, where each node corresponds to a class, and the node with the highest value indicates the predicted class, the absolute difference between the node values can be used to estimate a model certainty. In this thesis, the network certainty, Γ, is defined as

$$\Gamma = \frac{kn_{\max} - \sum_k ||n_k||}{k}, \tag{2.10}$$

where n_k are the node values and $k \in \mathbb{Z}+, k \in [1, m]$. For a combined model that uses several network models, the model certainty is defined as the sum of the model's certainties.

2.5 Ultrasonic sensor systems

Many new cars use ultrasonic sensors (USS) to detect objects in proximity. Ultrasonic sensors can measure distances with low power consumption. The sensors send a sound wave with a frequency outside the human hearing spectrum, making them appear quiet. If the sound wave hits an object, it will be reflected, and the sensor will then listen for the echo to measure the distance to the object. The reflection amplitude and general direction depend on the object's material and shape, but since the sound wave has a spherical propagation, it is very likely for some sound to reflect back regardless of the shape or material. As sound travels at $v_s \approx 343$ m/s in ground level air [19], the distance to the object can be calculated using

$$d = \frac{v_s \Delta t}{2}, \tag{2.11}$$

where Δt is the difference in time between the emission and detection of the sound wave. These sensors are commonly used for parking and object detection in both the front and rear of the car. New car models have several USSs in the rear and the front, which can all triangulate and listen to each other's echos and their own. Therefore, a specific firing sequence is used to map and measure the objects.

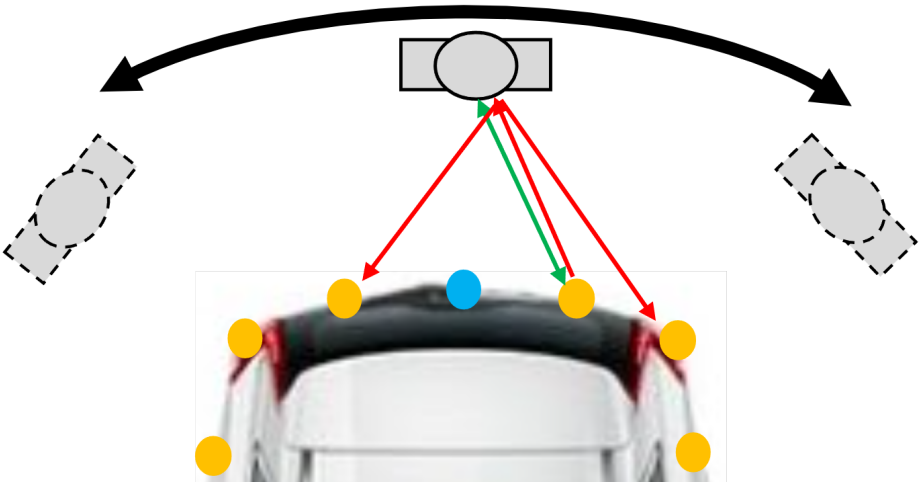


Figure 2.3: Illustration of the firing sequence and how neighboring sensors listen to their own and each other's echos. The yellow circles indicate the positions of the ultrasonic sensors, and the blue indicates where the camera is located.

The sensors in the rear of the vehicles have the following notation:

- ROR - Rear Outer Right
- RIR - Rear Inner Right
- RIL - Rear Inner Left
- ROL - Rear Outer Left

In figure 2.3, RIL fires a signal and listens to its own echo, and the neighboring sensors, RIR and ROL, listen to the same echo. There are two sequences of sensor firing where a sensor either only listens to neighboring sensors or emits a signal and listens to itself. These modes define the firing sequences and are swapped for each sequence. The sensors that listen to other sensors can distinguish which sensor signal it receives by utilizing small sound signal frequency differences that make each signal unique.

Data could be obtained from the following signal ways:

- Direct Signal way - when the receiving sensor detects its transmitted burst (RIL-RIL & RIR-RIR).
- Indirect Signal way - when the receiving sensor detects a burst from its neighbor sensor (RIL-ROL).

3
Method

In this chapter, the method is presented together with the investigated gestures.

3.1 Gestures representation

Two distinct gestures were chosen for this project, a 'kick' gesture and a 'hand' swipe, to function for trunk actuation activation. The gestures were selected based on their distinctiveness and ease of detection for both ultrasonic and visual sensor perspectives. All gestures were, therefore, conducted around a one-meter distance away from the trunk.

The 'kick' gesture is a well-established gesture that is sometimes used in combination with a radar sensor. The 'kick' was specifically chosen as users already know it for trunk actuation activation, as illustrated in figure 3.1. The other gesture investigated is the 'hand' swipe gesture due to its simplicity and natural association with symbolizing opening, see figure 3.2.

Figure 3.1: Illustration of the leg 'kick' gesture. Note that the distance between the starting position and the vehicle was roughly one meter.

Figure 3.2: Illustration of the 'hand' swipe gesture.

3.2 Approach and general idea

It is relatively easy to realize that there is not only one solution to the formulated problem in the project. The created method was influenced by other gesture recognition projects. The main idea from the method is to use all the available information, meaning both the data from the USS and the visual input from the rearview camera on the vehicles, and use the combined information from these sensors to create a robust model for classifying the information. The classification problem becomes more complex as false positives, meaning the model classifies a non-intended gesture as a gesture, which is considered a non-intended gesture that should not activate the actuation. The model needs to know that the gesture was intended, and, at the same time, it should be able to distinguish the same gestures for all people.

Considering the idea of a combined model, combining the information from the USS and vision data, external information, and logic from the vehicle can be utilized. For instance, the model should first determine if the key to the car is near the vehicle. If it is, the model should check for nearby objects and the change of object distances using the USS. The camera system is activated if an object moves close and this logic is satisfied. Now, the vision model is initiated and uses visual information of the nearby object to classify whether the given object is human or not. The next step can be initiated if the object is classified as a human. After this, the vision model will classify whether a gesture is made. The first time the vision model classifies a gesture, the USS model classifier is initiated to verify the classification. Since this model requires temporal input, a time window is created where the most recent USS measurement replaces the earliest. The outputs of each classifier, vision and USS, should now be combined. This is done using a weight function that utilizes the network certainty of each model/classifier together with a parameter α that scales each signal, creating an adjustable bias towards one of the

models. This parameter is tuned by iterative testing. The vision data is used in combination with the USS data to acquire as much relevant information as possible such that the model where this information acts as a fail-safe for an incorrect classification. In this way, the power consumption is reduced by using the passive USS before activating the vision system.

The problem was split into smaller parts, and information was handled separately to achieve this logical structure. Three networks were created: a USS-based model, a static vision human detection model, and a dynamic vision model. These models should then work together, following the logic presented in figure 3.3.

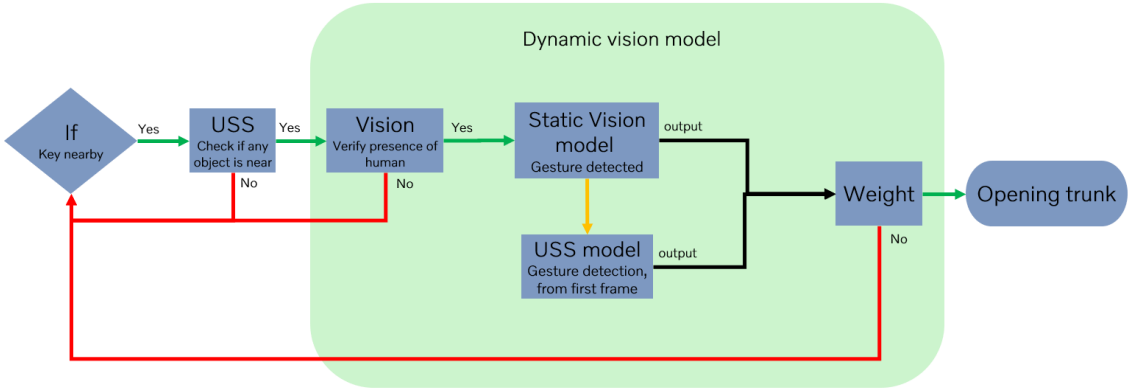


Figure 3.3: Illustration of the general simplified logic, where all networks are shown as boxes with an input and output signal. The yellow arrow illustrates the initiation of the time window used for the USS model. The vision and USS model classification outputs are weighed using a factor α to determine the total classification output.

3.3 Combined model

The USS and vision models can be used in combination with each other. Using some external logic to fuse the output classifications, a combined model was created using both the USS and visual inputs. This logic can be tuned to potentially achieve an increased results performance compared to the USS and vision models separately. The overall logic used in this combined model is illustrated in figure 3.3. In the figure, there are some external functions and information, such as key detection and human detection; these functions are already implemented and are, therefore, assumed to work flawlessly for this project.

As an object is classified as a human, the static vision network will be triggered to classify for any gestures. At the same time, the USS model will collect data points until the length of the time window is satisfied and then classify the measured distance pattern over time. This is triggered by the static vision model when it first classifies a gesture. The collected measurements for the USS before this instance are used to fill the time window in the USS model. All new measure data is then inserted, and the oldest data point is removed so that the time window is moved. The outputs of both models are weighted by a weight function that takes

the network certainty into consideration and a tuning parameter that the user can adjust. In this way, the classifications are fused and can easily be tuned to compensate and to rule out false positives, etc. This logic and utilization of the USS and vision models is defined as the combined model.

To evaluate the model, recorded data was fed to a Python script which simulated the combined model. Randomly selected USS data and vision data that belong to the same classification were fed to the model. External factors such as key proximity and human detection are assumed to always be triggered for these cases.

3.4 Data acquisition

For the network models to be able to identify certain gestures, data has to be collected for all classification tasks involved in the project. In this project, it was necessary to generate new data for the specific gestures and the sensor setup of the provided test vehicles. The test vehicles have systems created for data collection in all instruments, which were saved on a portable solid-state hard drive. The data the USS and the rear-view fish-eye camera generated were synchronized in time. Each measurement for both types of sensors was also initiated simultaneously. Each measurement could be extracted and saved into a folder containing the separate data for each type of sensor in the predetermined mf4 format.

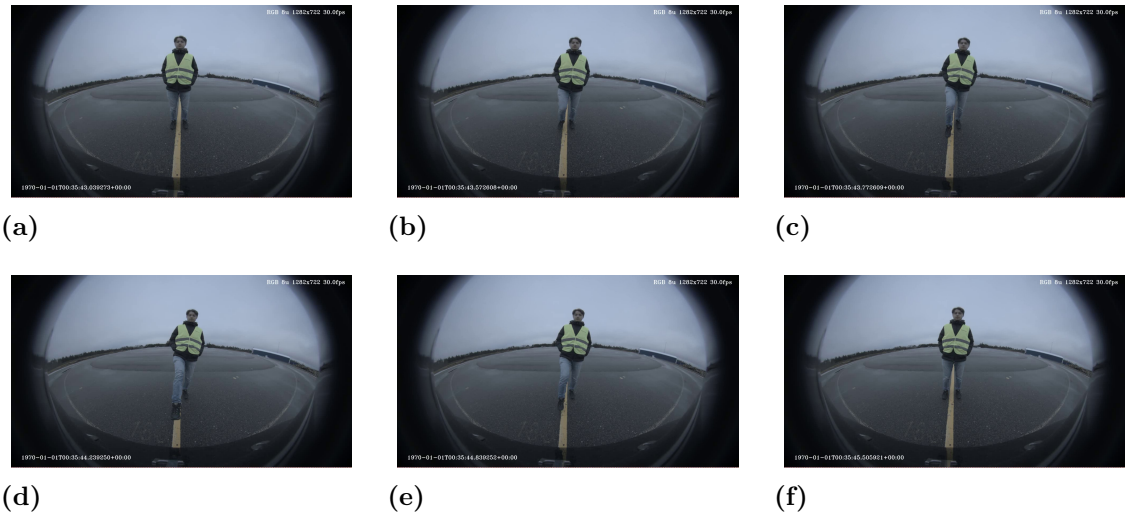
The measurements were conducted as follows in the following order:

1. Discuss and determine what gesture and motion should be recorded and in what position.
2. Find an appropriate area to record the measurements, free from obstructions, to ensure unimpeded movement and accurate gesture capture. The chosen environment replicates typical parking scenarios encountered in urban settings.
3. Set up logging equipment and designate one team member to operate the recording equipment from within the vehicle, starting and stopping each session and monitoring real-time data stream to the logger, to its hard drive, and the capture through the rear camera system.
4. One person makes the agreed upon gesture communicating with the person starting/stopping the recordings when to initiate each measurement.

In total, 231 recordings were acquired. A snippet of the measurement data logbook is illustrated in figure 3.4. The recordings included three different people making gestures in different situations, with various backgrounds and weather conditions. Figure 3.5 illustrates a few frames from one snippet.

Nr.	Timestamp	Filename	Size (GB)	Gesture	Attribute	Position	Distance (m)	Number of measurements	Enviroment	Temperature	Light intesity (lux	Whether	Person	Vehicle
43	2024-03-25	015042	0,5	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
44	2024-03-25	015053	0,6	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
45	2024-03-25	015105	0,5	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
46	2024-03-25	015114	0,5	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
47	2024-03-25	015125	0,4	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
48	2024-03-25	015134	0,5	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
49	2024-03-25	015146	0,5	Kick	Right leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
50	2024-03-25	015701	0,5	Kick	Left leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
51	2024-03-25	015712	0,5	Kick	Left leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
52	2024-03-25	015722	0,4	Kick	Left leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
53	2024-03-25	015731	0,4	Kick	Left leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519
54	2024-03-25	015741	0,4	Kick	Left leg	Rear middle	1	1	Garage	17	5000	Indoors	Person 3	P519

Figure 3.4: A snippet of the measurement data logbook. This file connects the data files to the measurements and was used to label the datasets.



(a) **(b)** **(c)**

(d) **(e)** **(f)**

Figure 3.5: Illustration of a frame sequence from a single recording, depicting an individual standing in an open area without performing any gestures.

3.4.1 Collected USS data

The data from the USS contains measures, such as distance and signal amplitude, for each sensor's own and neighboring echoes. The sampling frequency of the USS is 50 Hz. Each USS recording is 262 samples long, roughly corresponding to a data recording of 5.2 seconds. The measurements are low-pass filtered to remove extreme points and noise. From the kick motion gesture, a typical measurement would look like what is illustrated in figure 3.6. For this gesture, a human would stand roughly one meter away from the car trunk and make the gesture. In the figure, one can distinguish seven data points where the measured distance is significantly closer, which indicates the kick.

Some sampled data points were more similar to figure 3.7, where extreme value measurements are illustrated. Several points of valuable information were lost due to noise, in some instances, all of the distance readings during the gesture were lost to noise, which resulted in readings that only indicated the presence of an object roughly one meter away. At other times, all the important data points are captured, and a clear motion signature can be detected, which is crucial for the model in clas-

sifying this information. The data illustrated by the figure 3.6 and 3.7 is filtered such that the points that are considered noise are removed.

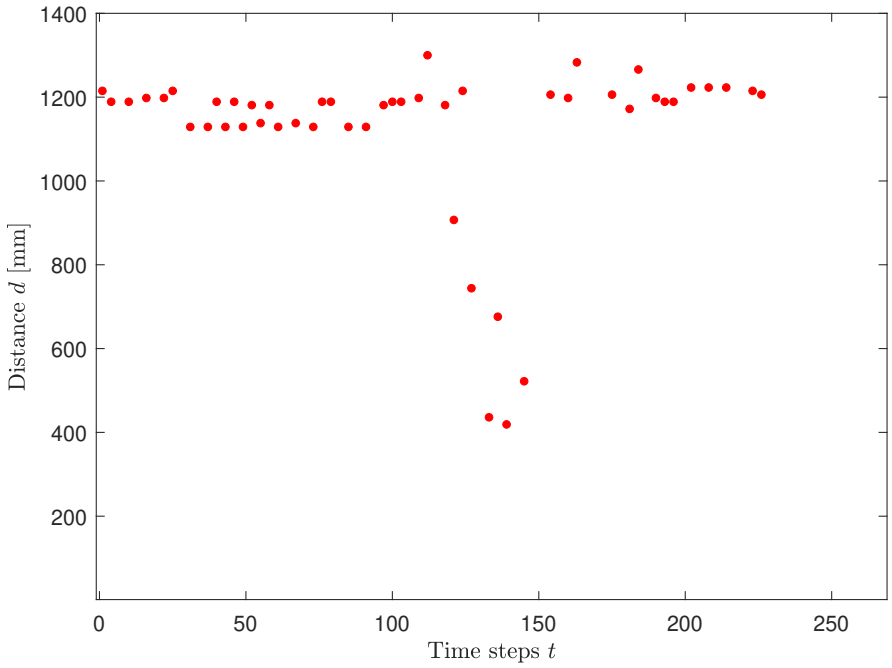


Figure 3.6: Illustration of the general distance measured over a time span around 5 seconds. Note how the detected distance is closer around time step 150. This is the indication of the kick gesture.

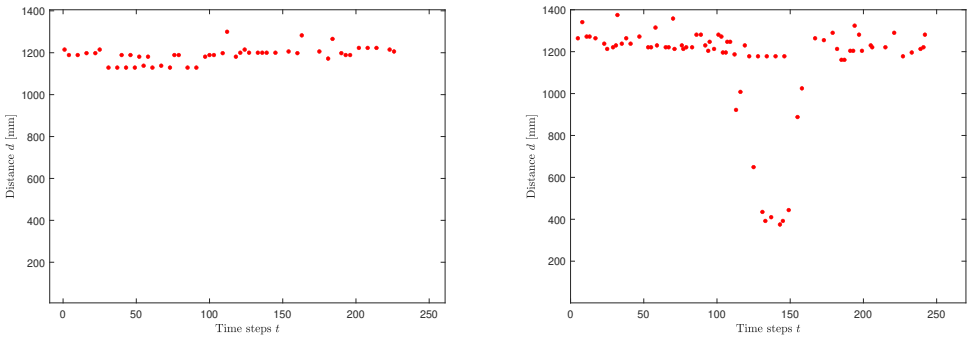


Figure 3.7: In the figure to the left one can see an example of a measurement series where all points of interest were lost by noise such that no kick gesture could be distinguished. The right figure shows an example of a clear gesture profile. The green points are the preprocessed and merged points, as explained in the methods chapter. The red and blue points are from `echo1` in `RIL` and `RIR` respectively.

By combining 30 measurements of data points, a refined set of data points can be obtained, see figure 3.8.

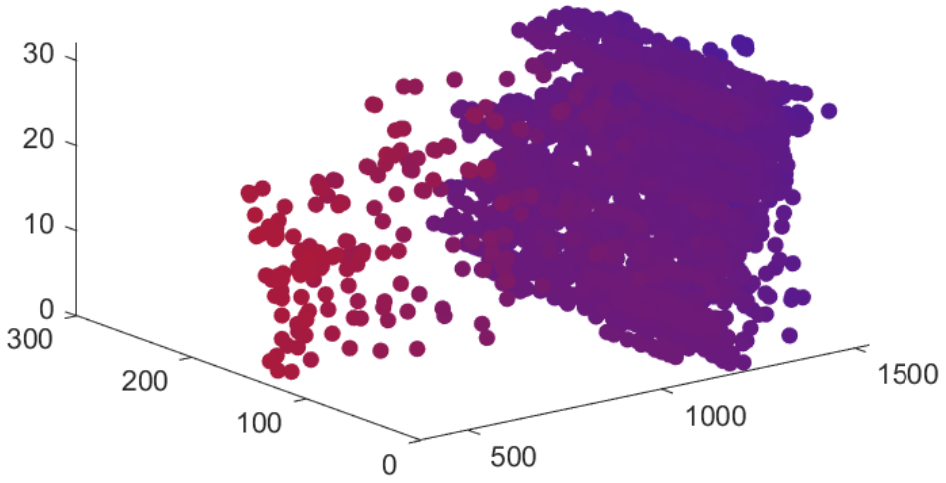


Figure 3.8: Illustration of several (30) measurement sequences put together.

3.5 Preprocessing: Decoding

The initial preprocessing stage consisted of decoding the acquired recordings. The process of decoding involved setting up required environments and employing decoding utilities according to the following algorithm 1, described in further detail in appendix A.1.

Algorithm 1 Decoding of acquired USS and Vision recordings.

1: Ensure execution environment:
2: - Linux environment, using `WSL2` with `Ubuntu` for this project.
3: - Deploy `CUDA` extension and set up `Singularity` container.
4: Extract decoding utilities to local workstation.
5: Run script `decode_logg.py` (see appendix A.1) to streamline decoding:
6: **for all** USS recordings in the input directory **do**
7: - Recreate output directory structure based on input directory structure.
8: - Construct Singularity, employing decoding utilities, for conversion.
9: - Execute the conversion command.
10: **end for**
11: **for all** Vision recordings in the input directory **do**
12: - Recreate output directory structure based on input directory structure.
13: - Construct Python command, employing decoding utilities, for conversion.
14: - Execute the conversion command.
15: **end for**

3.6 USS model network

The ultrasonic sensor-based model network was built upon the acquired data and the merged pre-processing of USS inputs. Since the sensors used in the project only measure the distance away from itself in one dimension, the idea is to capture a pattern of distance changes over time. The model essentially uses the information from the distance changes over a given time. Several approaches were considered, and in the end, a time window approach was selected, covering the 262 measurements corresponding to the time window size. This was done based on the collected data as one gesture took roughly this time to complete. The vehicle that was used to collect and record the USS data had four rear sensors, where each sensor listened to its own echo and its neighboring echo. The car's mainframe manages the triangulation of these echoes to calculate distances, which restricts direct access to processed data. This limitation necessitates the model network to use direct sensory data and excludes the possibility of data points in more dimensions. Therefore, time windows containing distance measurements were used to train the network for these kinds of data structures. The time window for sampling is triggered by an external signal linked to the car's security and proximity alert system, allowing for precise data capture when a gesture is likely to occur. This method enhances model accuracy by focusing on relevant data periods when a gesture is possible, see figure 3.3.

3.6.1 Preprocessing USS classification data

After the raw data extraction and conversion to the format .hd5f, it was possible to extract the direct USS distance data using a Python script, see Appendix A.2. The rear sensor echo distances are extracted from the format, noise is filtered out, and time windows are created where the data from each recorded gesture is merged together separately for each time window. Since the recordings vary in length, measurements over 5.2 seconds are cut and measurements under are padded. A comma-separated value file, csv-file, was created with all the measurements of the same class or gesture. The data was plotted over the given time steps to visualize the time window. Since the data is saved as separate files if it succeeds at a given storage size, the Python script combines all similar files in each directory, where each directory contains one measurement sample.

The noise filtering as discussed earlier, works by removing the data points that are outside the range of 60000 mm and 2 mm since these are the limits of the hardware and all points above or below are considered to be noise. Since not all sensors have guaranteed disturbances or noise at the same time stamps, the script also checks each step to see if the neighboring sensor picked up a non-noise measurement and adds that value to a new vector containing the merged values from the sensors. In that way, more information of interest can be saved in a singular measurement vector, which can later be fed to the neural network. These vectors were later combined with the script and saved as a csv-file.

3.6.2 Build USS classification model

Based on the data pattern complexity of the input data classification problem, the network was initiated with three hidden layers and a substantial number of nodes as stated in the theory section. By taking a given set of measurement frames or measurements over given time steps, an input window for a short time sequence can act as the input to the network. The model would use the differences in detected distance within this time window over the time steps to detect patterns from the performed gesture. Only the' kick' gesture was classified to narrow down the complexity of the initial data. The network had two output nodes. The idea of having two output nodes was also to be able to integrate a network certainty as discussed in the theory section 2.4.1. These two nodes corresponded to either whether a 'kick' was detected or no 'kick' was detected. Furthermore, two outputs is useful for both the evaluation of the network and the combined model, which will be discussed later. The node with the maximum output is used as the chosen output.

3.6.3 Training and validation

The USS classification model is a linear dense NN using three hidden layers utilizing the tanh activation function. The network was built for two network classes, 'kick' or 'no kick', with 262 input nodes, one for each time step in the given time window. See figure 3.9 or A.3 for a detailed description of the network architecture. The time window corresponds to roughly 5.2 seconds in recorded time, which was deemed sufficient to capture all the kick data in the collected samples. A dataloader class was defined where the labeled data is loaded from the csv files and then easily extracted by a function. The dataset was then split into a training set, a validation set, and a test set with the respective separation, 80 %, 10%, and 10%. A training loop was defined where data from the training set was loaded together with their respective labels. The `Adam` optimizer was used with a learning rate of 0.001, and a mean square error function was used as a loss function.

Each epoch was monitored while running the loop to roughly evaluate the network model. By observing the trend of the validation accuracy and loss, one can monitor overfitting and evaluate the model as stated in the theory section. The validation accuracy was saved for each epoch, and after training, the model was saved.

3.7 Static vision model networks

Several projects in gesture recognition and static image analysis use a Convolutional Neural Networks (CNN:s) approach, which was also chosen for this project. After training, it is quite compact and requires little computational resources compared to other alternative models, such as the vision transformer (ViT) network. For these networks, greyscale imagery was used in this project.

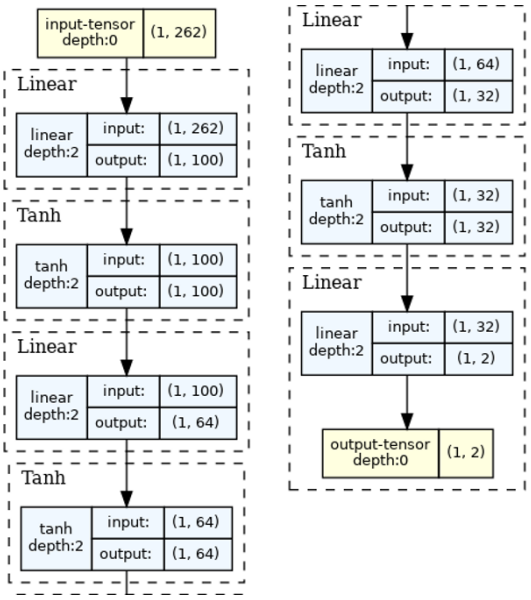
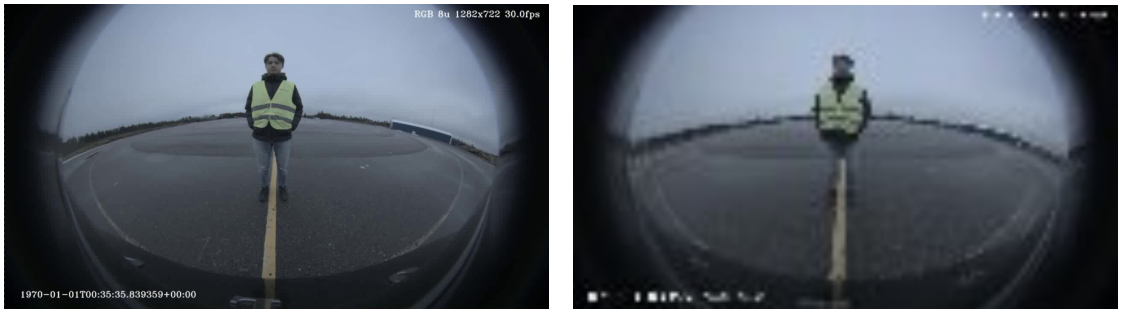


Figure 3.9: Illustration of the USS network architecture.

3.7.1 Preprocessing static vision classification data

From the data collection, the recorded visual data was saved in mf4-format, similar to the raw data from the USS. However, a different decoding method was used due to the difference in size and encoding. A script, see Appendix A.5, was created for saving each frame as a jpg file and downsizing the image resolution by a factor γ. The lesser resolution will result in a smaller network as the input dimensions directly correspond to the number of pixels in the images. If the colors of the pixels are included, each pixel contributes with three color channel inputs. The scale factor, γ, was tested iteratively as a parameter using the same network model with a scaled-down input size and comparing the results with the model using the maximal resolution until a satisfactory scale factor could be determined. See 2.3 in the theory section.



(a) **(b)**

Figure 3.10: Raw frame extracted from one of the mp4-files to the left and the down-scaled version of the same frame to the right using the scale factor $\gamma = 0.1$.

All images were manually labeled as either the given gesture or no gesture for each

measurement and saved in separate folders for each gesture, counting no gesture as a class. The dataset was split as equally as possible to balance the data.

3.7.2 Static Vision classification model

Using (2.1), a starting point for the initial linear neural network part was made. Several different network structures were tested and implemented. As a base point, three convolutions were used as this is rather common in facial expression recognition networks of similar characteristics [10]. The network size, pooling layers, and linear deep neural network are all parameters that were shifted and implemented in several different ways, yielding different results. Since gestures can contain much information, the network needs to be able to capture a vast amount of feature information. Therefore, the overall size and channels of the network were set up to be able to capture this, and several transformations to the images can be applied, [20] and [21]. A base channel size of 64 was used and varied slightly between the layers. The static vision classification network was based on the CNN structure with three convolutional layers and three linear layers to allow for the possibility of complex data classification which is expressed in the given dataset problem while still being rather compact.

For the linear network part, the layers were also varied in terms of size and amount. As a starting point, one hidden layer was implemented, which connected to the seven outputs, one for each emotion classification. See figure 3.11 for the full architecture and A.6.

The CNN-model was built using `pytorch`. Before the data is fed to the network, each image is resized to 100x80 pixel and converted to greyscale such that the input dimension is reduced for the network. The images are also normalized using the standard normalization for greyscale imagery. After this, the data set is split into a training set, a validation set, and a test set. This is then fed to the training and validation functions with a batch that is selected randomly from each set. For this work a batch size of 32 was used. A cross-entropy loss function was used together with `optims SGD` optimizer. The learning rate was set to 0.005. Using the preprocessed data, a training and validation function could be defined, see 3.7.3.

3.7.3 Training and validation

A validation function was defined, where the network model is fed data from the validation set, and its output is compared to the labeled targets. In this function, each correct classification is counted and then divided by the total number of validations. This is done using the dataloader for the validation set. Also, for each image or item in the dataloader, the validation loss is calculated using the mean square distance between the network output and the validation targets. This is done in the same way as for the training set.

To combat overfitting, each epoch is monitored while training. When the valida-

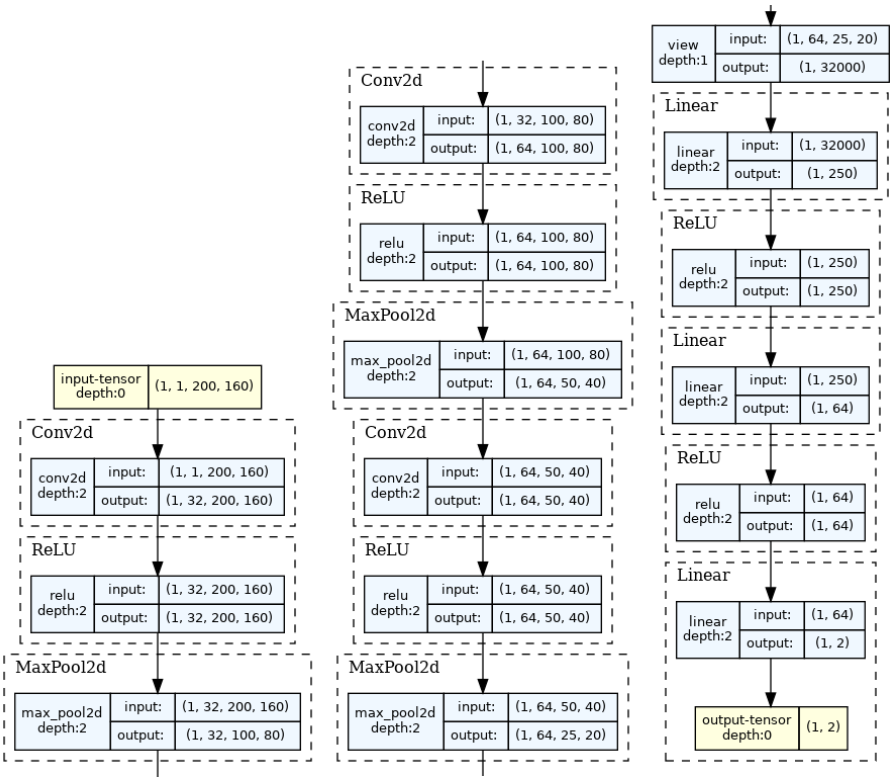


Figure 3.11: Illustration of the static vision network architecture.

tion accuracy has reached a maximum peak, and then the accuracy is decreased for several iterations, the network is assumed to be overfitting. Also, the training loss was calculated using cross-entropy which was monitored in the same way. As the training loss continues to decrease while the validation accuracy is not increasing, it can also be a sign of overfitting. The validation accuracy was saved for each epoch above a set limit of validation accuracy, and if this accuracy was better than the previous one, the current network was saved. In this way, further training would not negatively impact the saved network.

3.8 Dynamic vision model network

The dynamic vision model network was developed to model spatial-temporal features. Compared to the static vision model network, which only models the spatial features. The literature study in 2.2.4 suggests that the ResNet 3D CNN, and especially the R(2+1)D classification model, were a suitable choice as a basis for the dynamic vision model.

Due to time limitations, it was not feasible to develop the R(2+1)D model from scratch, and it was therefore retrieved from the `PyTorch` library [22]. The model is based on the work described in [13]. To meet the project's classification requirements, adjustments were made to the model's architecture as stated in section 3.8.2. Training the model from scratch on a small dataset, which consisted of 231 videos of multiple classes, posed a significant risk of overfitting due to the high number

of trainable parameters, approximately 31.5 million. A large volume of data was required to minimize the risk of overfitting [3]. Fortunately, `PyTorch` provided pre-trained models, trained on the large benchmark dataset `Kinetics-400`, widely used for human action recognition [22]. Transfer learning was, therefore, possible, [23], where the acquired dataset was used for fine-tuning. As the pre-trained model does not accept videos with arbitrary size and length, preprocessing was required.

3.8.1 Preprocessing dynamic vision classification data

As stated in section 3.5, the directory structure consists of folders for each separate measurement, containing one or several mp4-files, depending on the length of the recording. This required the creation of the script, `merge_videos.py` presented in appendix A.9, which finds and merges mp4-files by concatenation. The fish-eye camera captures clips of 30 frames per second (fps), and through iterative testing, it became clear that 2 fps was sufficient for the dynamic model to achieve high accuracy.

Subsequent steps addressed the labeling process. The script `video_Labeling.py`, presented in appendix A.10, labels merged video files according to logbook entries shown in the figure 3.4. It executes three types of labeling functions: binary class labeling, multiclass labeling, and extended multi-class labeling. The binary labeling function assigns labels 'kick' - 'no_kick' or 'hand' - 'no_hand'. The multiclass labeling function assigns three labels 'kick', 'hand' and 'no_gesture' and the extended multi-class labeling function uses combined gestures and attributes reaching a total of nine labels, seen in table 3.1. All three functions generate a csv-file with video paths and labels, and a csv-file with label mapping, creating a dataframe.

Table 3.1: Overview of labeling functions and their corresponding labels

Binary class labeling		**Multi-class labeling**	**Extended multi-class labeling**
'kick'	'hand'	'kick'	'kick right leg'
'no_kick'	'no_hand'	'hand'	'kick left leg'
		'no_gesture'	'kick right leg and 2 bags'
			'walk pass perpendicular forward and back'
			'walk approach and depart gull wing'
			'walk approach and depart straight'
			'walk approach and depart straight 2 bags'
			'hand motion right hand'
			'hand motion left hand'

The remaining preprocessing was deployed within the principal script named `dynamicVisionNetwork.py` presented in appendix A.11. The script was initiated by reading video paths, labels, and mapping from previously created data. The video data and labels are split into training, validation, and test sets using the `train_test_split` function from the `scikitlearn` library in two steps. The initial split separates the data into training and test sets with an 80/20 ratio. A subsequent split further divides the training data into training and validation sets, also with an 80/20 ratio. Stratified sampling was employed for all the datasets to address the

imbalance in the distribution of classes. Followed by data loading, transformation parameters are defined using `PyTorch`'s `transforms.Compose` to ensure that the input data was consistent with size and normalized according to the requirements. To effectively leverage the learned features of the pre-trained model, the configuration parameters and preprocessing operations, such as the frame size, are preferred to be consistent with that of the pre-trained model. These are presented below in table 3.2 below [24].

Table 3.2: Configuration parameters and preprocessing operations for the pre-trained R(2+1)D model [24].

Parameter	**Configuration**
Frame rate	15
Clips per video	5
Clip length	16
Resize size	[128, 171]
Crop size	[112, 112]
RGB $\bar{x}$	[0.43216, 0.394666, 0.37645]
RGB σ	[0.22803, 0.22145, 0.216989]

Video datasets are created using a custom `VideoDataset` class, presented in appendix A.11, which handles the loading and processing of video data according to the defined transformations. In the `VideoDataset` class, the video frames are read using `read_video` method from `PyTorch`, which returns the frames in a tensor format (C, T, H, W) where T stands for temporal dimension, C stands for channels, which is three in the case of RGB-video, H stands for the height of each frame in pixels and W stands for the width of each frame in pixels. Due to time limitations, the configuration parameters in table 3.2, with regard to length, were not considered. Instead, the change of fps from 30 to 2 fps gives the longest video 70 frames. To ensure that the length of the videos in the datasets are concise, the elementary method of padding the last frame to 70 frames. Transformations to each frame are provided using the `apply_transform` method. This method converts each frame from a tensor to Python Imaging Library image format, normalizes it to the range [0, 1], and applies the predefined transformations.

These transformations ensure that the input data is consistent with size and normalized according to the specified mean and standard deviation values. As the R(2+1)D model accepts video frames in batched format (B, C, T, H W) where B stands for the number of video samples in a batch, the `apply_transform` method finally converts the transformed frames back into a tensor and reorders the dimensions to match the expected format (C, T, H, W) [24]. In figure 3.12, it is shown how a video frame of original resolution $1282 \times 722 = 925,604$ pixels was resized to $128 \times 171 = 21,888$ pixels, cropped to $112 \times 112 = 12,544$ pixels. The final cropped image has about 1.36 % of the original number of pixels.

(a) 1282 × 722

(b) 112 × 112

Figure 3.12: In figure 3.12a one can see the original resolution of a mp4-file, and in figure 3.12b one can see the down-scaled version of the same mp4-file, the scale factor is approximately $\gamma = 0.1$.

3.8.2 Dynamic Vision classification model

As mentioned earlier, the dynamic vision classification model is based on the R(2+1)D model of 18 layers from `PyTorch` [24], which is in turn based on [13]. `PyTorch` provides R(2+1)D-18 model which has been pretrained on Kinetics-400 dataset, the model has 40.52 GFLOPS and a file size of 120.3 MB. In `dynamicVisionNetwork.py`, presented in appendix A.11, the class `DynamicVisionNN` defines a custom neural network model. It was initiated by loading the pre-trained weights' configuration `R2Plus1D_18_Weights.DEFAULT` into the model `r2plus1d_18`. Furthermore, the final fully connected layer of the pre-trained model was replaced with a new linear fully connected layer modified for the specified number of output classes required by the dynamic vision classification model. To prevent the weights of the parameters of the pre-trained model from being updated during training, they were all frozen except for the newly added fully connected layer, ensuring that only this layer's weights would be updated during training.

In figure 3.13, a visual representation is seen of the modules, module hierarchy, tensor operations, shapes, and tensors involved during the forward pass of the model. Similarly, figure 3.14 highlights how gradients are computed and how they pass through the model during backpropagation. The nodes are color-coded and represent different types of tensors and functions: gray indicates backward functions, blue indicates reachable tensors requiring gradients, and green indicates the output tensor. A detailed description of the R(2+1) architecture is presented in A.12.

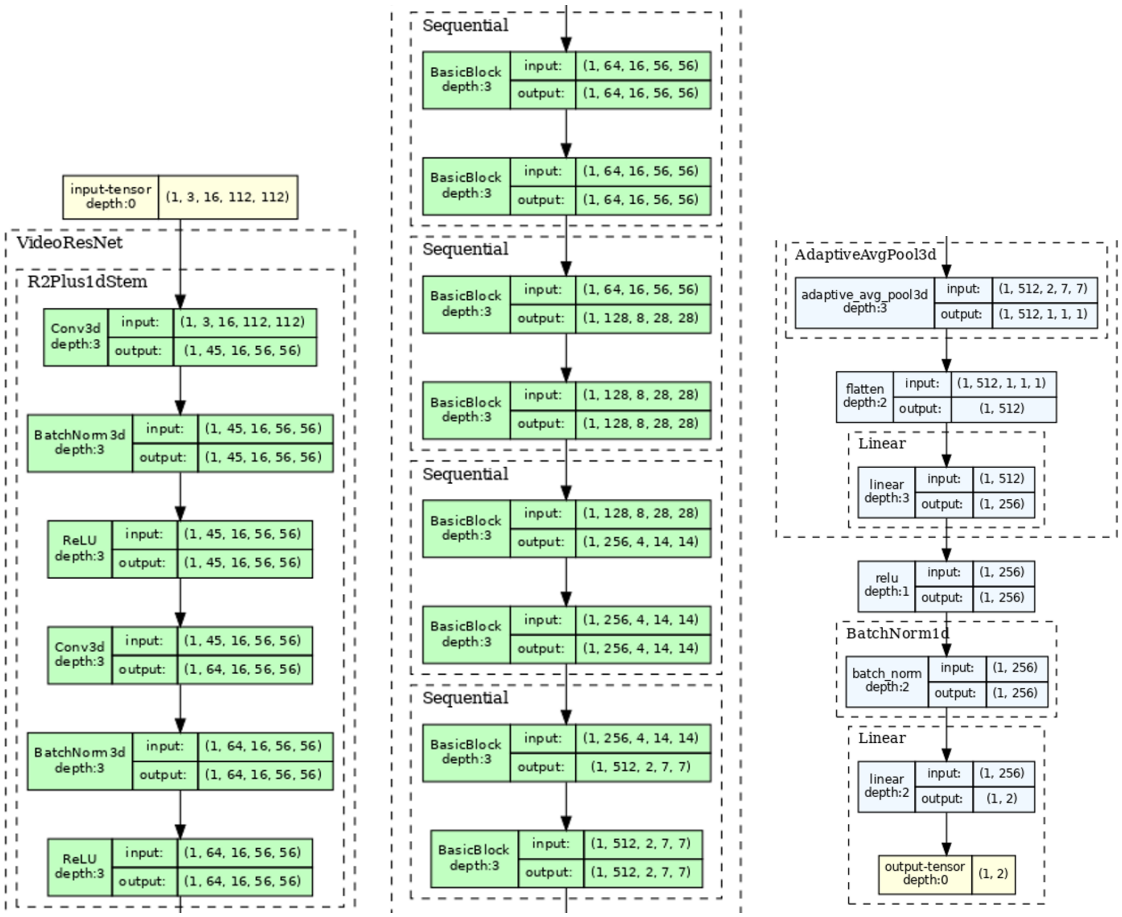


Figure 3.13: Illustration of the dynamic vision network architecture based on [24]. The final fully connected layer is adjusted for binary classification.

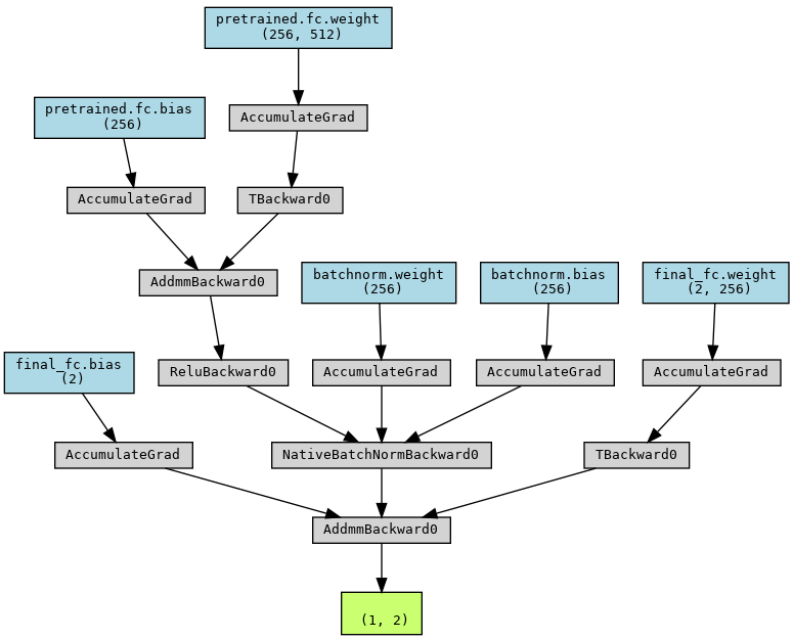


Figure 3.14: Overview of the entire model during the backward pass [24].

3.8.3 Training and validation

Building, training, and validating the dynamic vision model was done by a produced Python script seen in appendix A.11. The necessary components for training the model include data loaders, the model itself, an optimizer, and a loss function. A training function was defined, containing a loop that runs for a specified number of epochs. During each epoch, the training data is processed in batches, and for each batch, the model performs a forward pass to compute the output. The loss was calculated using the cross-entropy loss function defined in section 2.2.8. Backpropagation was then performed, and the Adam optimizer updated the model's parameters. A batch size of 4 and the Adam optimizer are chosen based on recommendations from the paper [13] and memory capacity. The learning rate was initiated as $\eta = 0.001$.

After the training loop for an epoch, the model's performance was validated. A validation function was defined, and provided a validation dataset. The model returned the validation loss, F1 score, and accuracy. To further detect or monitor overfitting, the model's performance was validated and printed at the end of each epoch. When the validation F1 score reached a peak and then decreases for several iterations, it was seen as an indication of overfitting. Similarly, if the training loss continued to decrease while the validation accuracy did not improve, this was also seen as a sign of overfitting. To retain the model with the best generalized performance, the model with the best validation F1 score was saved.

4

Results

In this section, the results for the different models and the combined model is presented.

4.1 USS model

The USS-based NN model architecture consists of 262 input nodes which are then connected to a linear hidden layer of 100 nodes. These are then connected to a second hidden linear layer with 64 nodes. After this, a third hidden layer consisting of 32 nodes and biases is connected. These are then connected to two output nodes which are used to indicate the classification. All layers have biases and use tanh as an activation function.

4.1.1 Model evaluation

The USS model achieved a validation accuracy of 75 percent after 26 epochs of training using a batch size of 10 samples while training on a refined dataset containing only kicks and no kicks. The model's performance varies rather distinctively between randomized training loops, but the loss becomes rather stable using the stated parameters, see figure 4.1.

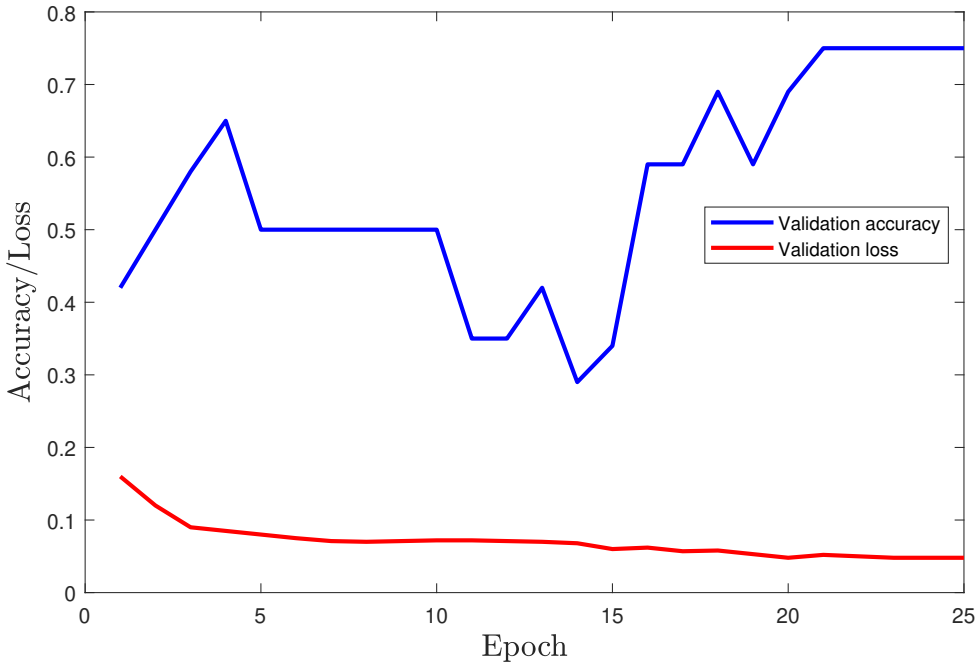


Figure 4.1: Illustration of the validation accuracy over each epoch using a batch size of 12 (blue) together with the validation loss scaled up by a factor of three (orange).

While evaluating the model using all data points in the set, a classification accuracy of 70.59 percent was achieved. The model precision was calculated as 72.50 percent and the recall as 76.32 percent. See table 4.1.

Table 4.1: The number of true positives, true negatives, false positives, and false negatives and their relative mean certainty are shown in this table for the USS model.

	Percentage of data	Quantity	Mean certainty
TP	42.65	29	0.06476634
TN	27.94	19	0.04153539
FP	16.18	11	0.051356044
FN	13.26	9	0.06391618

4.2 Static vision models

The static vision model NN architecture consists of three convolutional layers, three max-pooling layers, and three linear layers. The structure can be seen in appendix A.6. The rear-end fish-eye camera captured clips of 30 fps with a resolution of 1282x722 pixels. This resolution was scaled down to 128x72 pixels.

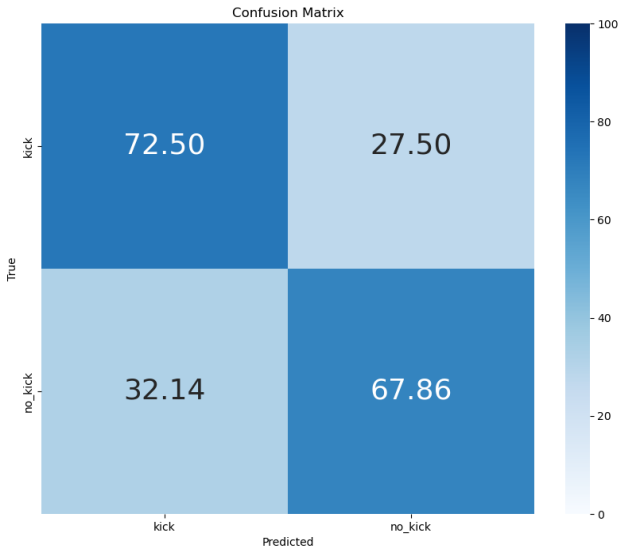


Figure 4.2: Illustration of the amount of TP:s and TN:s together with the FP:s and FN:s.

4.2.1 Binary gesture classification

The binary version of the static gesture model network, classifying the binary action of the kick motion as stated in section 3.1 and no kick, yielded a validation accuracy of 100 percent over a dataset containing over a thousand data points. The network was iterated for 10 epochs with a batch size of 15. Evaluating the network model over the test dataset, an accuracy of 99.906 percent was obtained with a precision score of 99.825 percent and a recall of 100 percent. This gives an F1 score of 99.912 percent, see table 4.2. The accuracy and loss trend is illustrated in fig 4.3 and a confusion matrix for the model is shown in fig 4.4.

Table 4.2: This table shows the number of true positives, true negatives, false positives, and false negatives and their relative mean certainty.

	Percentage of data	Quantity	Mean certainty
TP	53.61	572	9.971248
TN	46.29	494	6.7518287
FP	0.094	1	1.3280579
FN	0.000	0	-

4.2.2 Multiclass gesture classification

Training the static vision network on several gestures or classifications, 'kick' and 'hand' gestures, as explained previously, the model achieved a total validation accuracy of 100 percent using a batch size of 30 while training for 20 epochs. As seen in figure 4.5, the validation accuracy for each gesture is shown together with the total

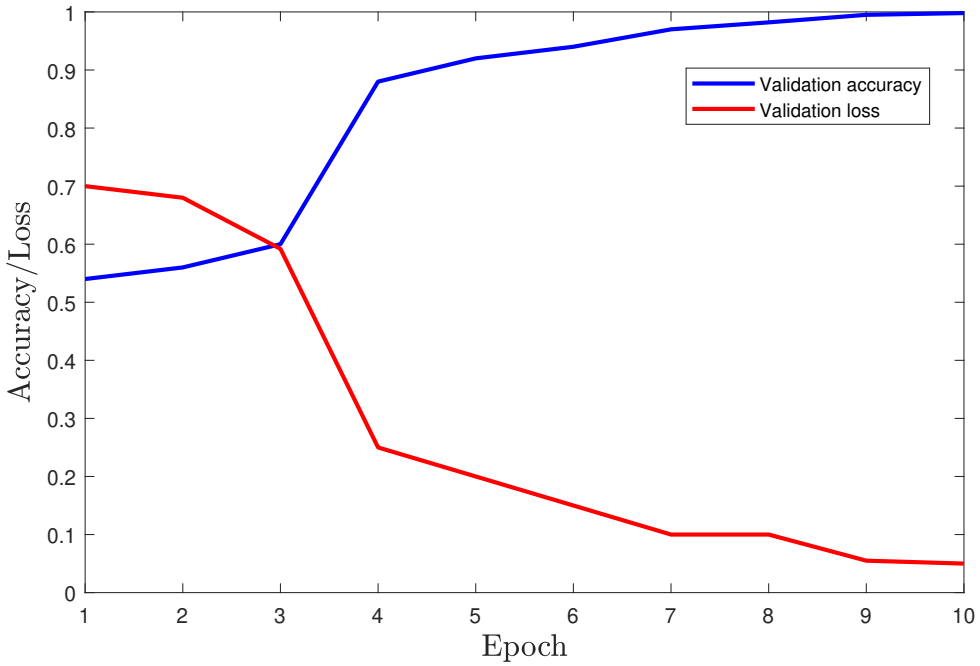


Figure 4.3: Illustration of the validation accuracy (blue line) over epochs and the validation loss scaled by a factor of three (orange).

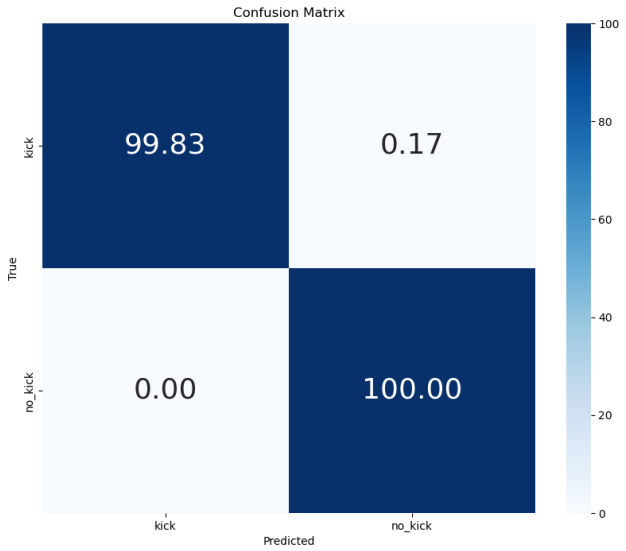


Figure 4.4: Illustration of the confusion matrix for the binary static vision model is presented.

validation accuracy and validation loss. The validation loss is scaled up by a factor of 30 for visibility.

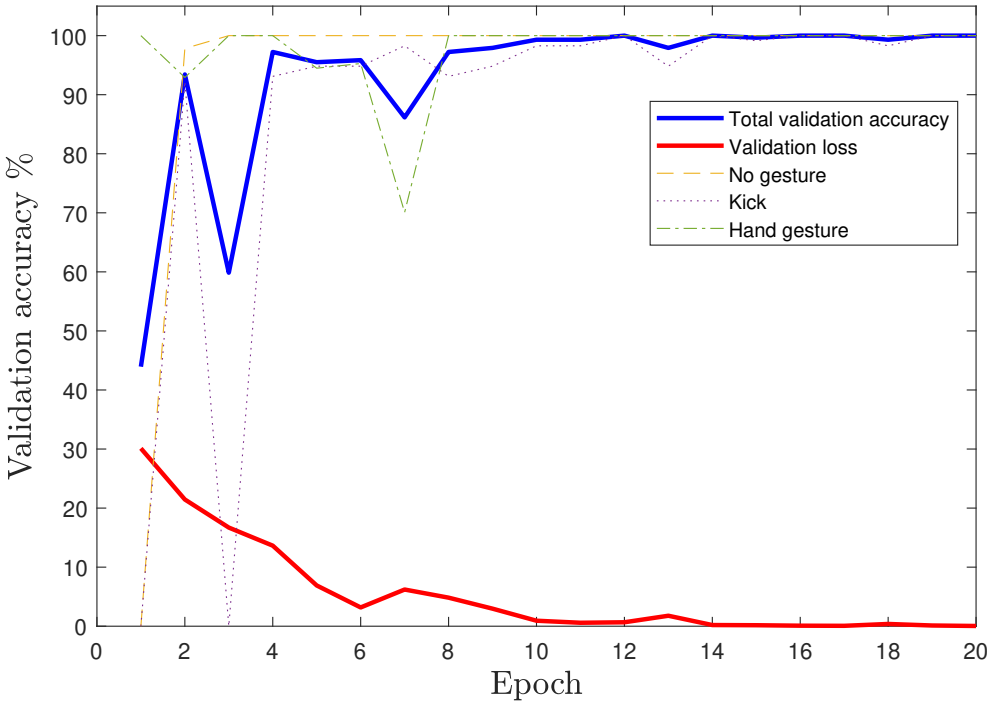


Figure 4.5: Illustration of the total validation accuracy over all gestures together with the separate validation accuracies for each gesture and the validation loss.

4.3 Dynamic vision models

The dynamic vision models underwent training, validation and testing on three distinct classification tasks. The first model was designed for binary classification. The second one handled multi-class classification with three classes. The third model extended the multi-classification model to a nine-class multi-classification task. The networks were iterated for 20 epochs with a batch size of four.

4.3.1 Binary gesture classification

The first iteration of the dynamic vision model network, classifying only binary action of the 'kick' - 'no kick' gesture, yielded a validation accuracy of 100 percent over a dataset containing 37 videos. The second iteration of the dynamic vision model network, classifying only binary action of the 'hand' - 'no hand' gesture, yielded a validation accuracy of 100 percent over a dataset containing 47 videos.

4.3.1.1 Collected and preprocessed data

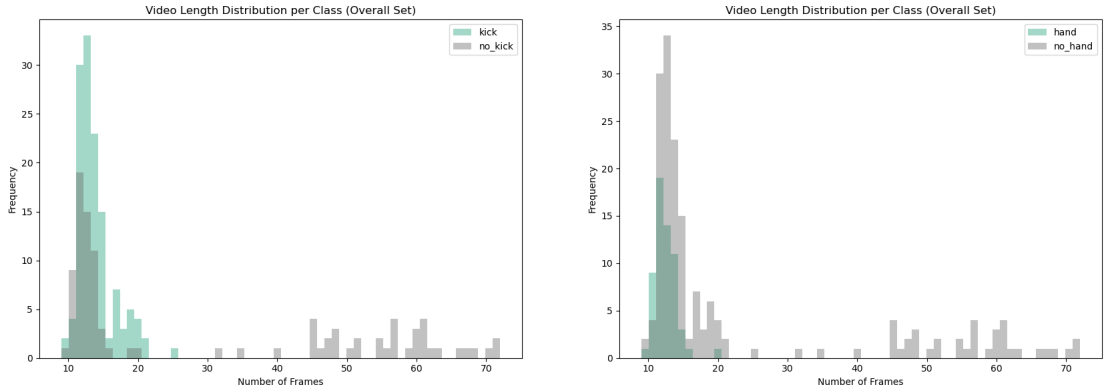
In the tables 4.3 and 4.4, the distribution of the binary classes over each of the datasets is presented. As mentioned in section 3.8.1, the distribution was achieved by using **train_test_split** and stratified sampling. In figure 4.6, it can be observed that a great number of videos at 2 fps are in the range of 10 to 15 frames long. The resulting maximum length of the videos in the dataset was 70 frames.

Table 4.3: Number of videos per 'kick' - 'no kick' gesture from the acquired dateset.

Dataset	Class	
	'kick'	'no Kick'
Overall set	131	100
Training set	84	63
Validation set	22	15
Test set	25	22

Table 4.4: Number of videos for the 'hand' - 'no hand' gesture from the acquired dateset.

Dataset	Class	
	'hand'	'no Hand'
Overall set	59	172
Training set	38	109
Validation set	9	28
Test set	12	35

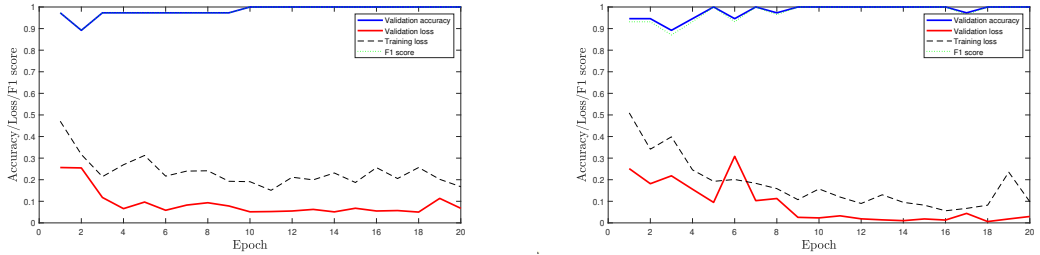


(a) 'kick' - ' No kick' gesture **(b)** 'hand' - 'no hand' gesture

Figure 4.6: Illustration of the video lengths, in number of frames, per class in the case of a binary classification task.

4.3.1.2 Model evaluation

For the binary classification task, the dynamic vision model recorded validation accuracies for both the 'kick' - 'no kick' and 'hand' - 'no hand' classification reached 100 %. The model achieved a perfect score for the test dataset, as seen in tables 4.7 and 4.8. This can be further visualized in the confusion matrices, figure 4.8a and 4.8b for this model, as the diagonal is 100 %.



(a) 'kick' - ' No kick' gesture metrics **(b)** 'hand' - 'no hand' gesture

Figure 4.7: Illustration of the training and validation loss together with validation accuracy and F1 score over 20 epochs

Table 4.5: The mean certainty and count for true positives, true negatives, false positives, and false negatives for the 'kick' - 'no kick' gesture.

	Percentage of data	Quantity	Mean certainty
TP	57.45	27	6.260
TN	42.55	20	3.776
FP	0.0	0	-
FN	0.0	0	-

Table 4.6: The mean certainty and count for true positives, true negatives, false positives, and false negatives for the 'hand' - 'no hand' gesture.

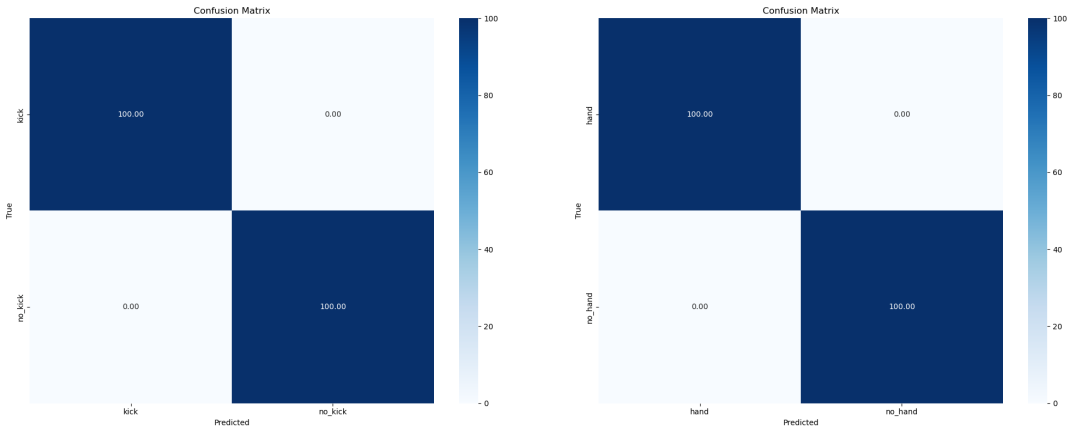
	Percentage of data	Quantity	Mean certainty
TP	25.53	12	5.808
TN	74.46	35	5.340
FP	0.0	0	-
FN	0.0	0	-

Table 4.7: Test metrics of dynamic vision model for binary classification of 'kick' - 'no kick' gesture.

	Loss	**Accuracy**	**F1**
Test set	0.025	1	1

Table 4.8: Test metrics of dynamic vision model for binary classification of 'hand' - 'no hand' gesture.

	Loss	**Accuracy**	**F1**
Test set	0.034	1	1



(a) 'kick' - ' No kick' gesture matrix **(b)** 'hand' - 'no hand' gesture matrix

Figure 4.8: Illustration of the confusion matrix from the test evaluation

4.3.2 Multi-class gesture classification

The multi-class gesture classification setup of the dynamic vision model classifies, as previously mentioned three distinct classes, 'kick', 'hand' and 'no gesture'. The

model yielded again a validation accuracy of 100 percent over a dataset containing 30 videos. In table 4.9, the distribution of the three classes over each of the datasets is presented.

Table 4.9: This table shows the number of videos per class in each of the datasets

Dataset	Class		
	'kick'	'hand'	'No gesture'
Overall set	131	59	41
Training set	83	38	26
Validation set	21	9	7
Test set	27	12	8

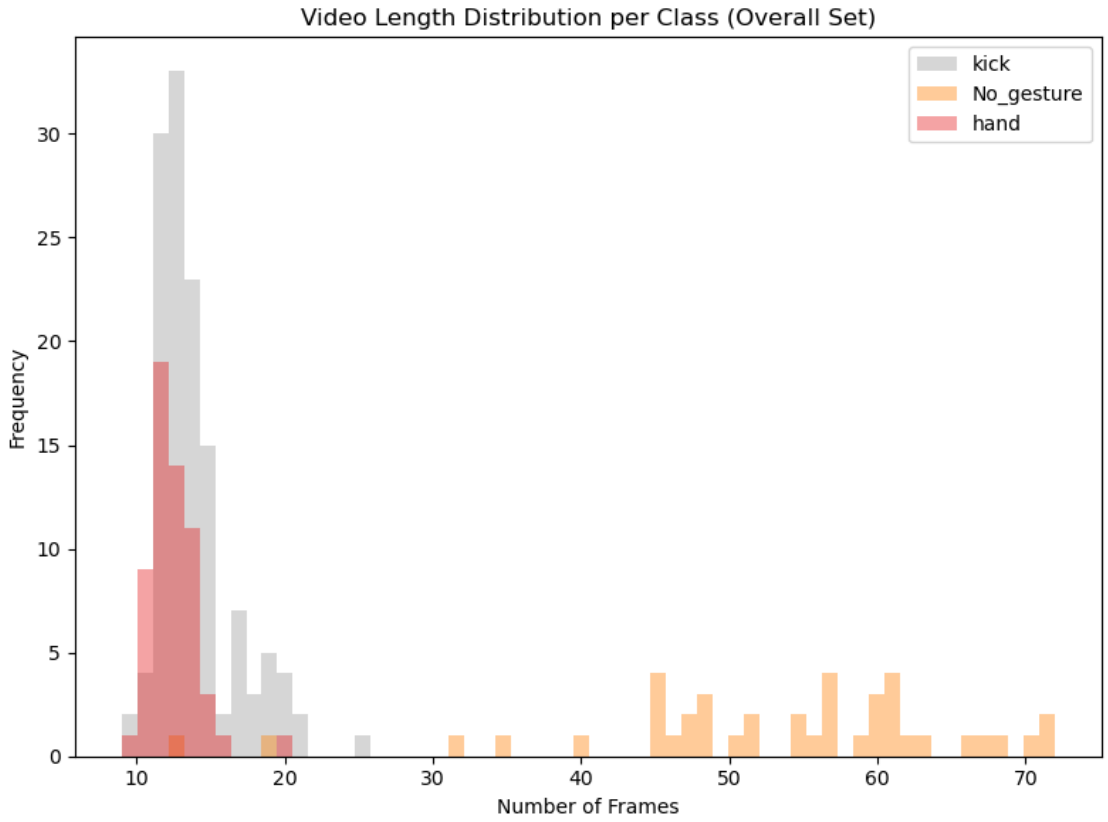


Figure 4.9: Illustration of the video lengths, in number of frames, per class in the case of the multi-class classification task.

4.3.2.1 Model evaluation

Recorded validation accuracy for both the 'kick', 'hand' and 'no gesture' classifications reached 100 %. The model achieved a perfect score for the test dataset, as seen in table 4.11. This can be further visualized in the confusion matrix 4.11 for this model, as the diagonal is 100 %.

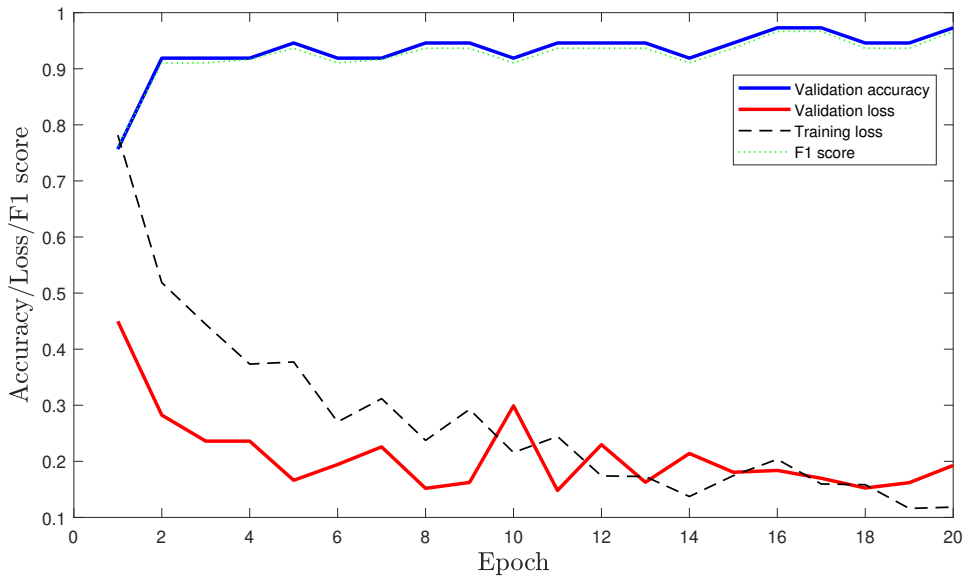


Figure 4.10: Illustration of the training and validation loss together with validation accuracy and F1 score over 20 epochs

Table 4.10: The mean certainty and count for true positives, true negatives, false positives, and false negatives.

	Percentage of data	Quantity	Mean certainty
TP	77.14	27	6.060
TN	22.86	8	6.328
FP	0.0	0	-
FN	0.0	0	-

Table 4.11: Test metrics of dynamic vision model for multi-classification.

	Loss	**Accuracy**	**F1**
Test set	0.019	1	1

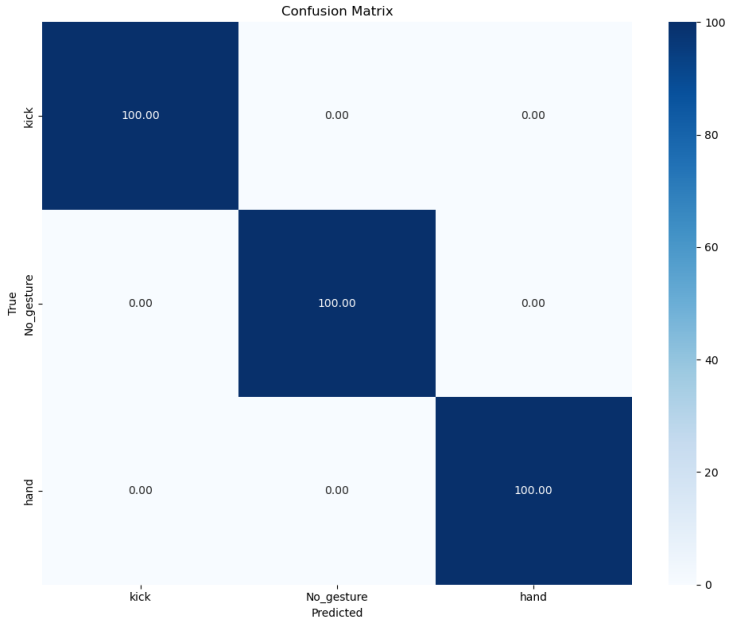


Figure 4.11: Illustration of the confusion matrix from the test evaluation

4.3.3 Extended multi-class gesture classification

The extended multi-class version includes several gestures as presented in the table 4.12. The distribution of the nine classes over each of the datasets is presented. As mentioned in section 3.8.1, the distribution was achieved by using **train_test_split** and stratified sampling. In figure 4.12, it can be observed that 'kick' and 'hand' classes have a higher concentration of videos with a shorter number of frames in the range of 10 to 20 frames long, while the 'walk' classes had a wider range of video lengths. The maximum length of the videos of 2 fps in the datasets is 70 frames.

Table 4.12: This table shows the number of videos per class in each of the datasets for extended multi-class classification.

Class	Dataset			
	Overall set	**Training set**	**Validation set**	**Test set**
'kick right leg'	62	35	15	12
'kick left leg'	50	34	7	10
'kick right leg and 2 bags'	38	15	1	3
'walk pass perpendicular forward and back'	13	8	3	2
'walk approach and depart gull wing'	12	8	1	3
'walk approach and depart straight'	10	6	2	2
'walk approach and depart straight 2 bags'	4	2	1	1
'hand motion right hand'	38	25	7	6
'hand motion left hand'	23	13	3	7

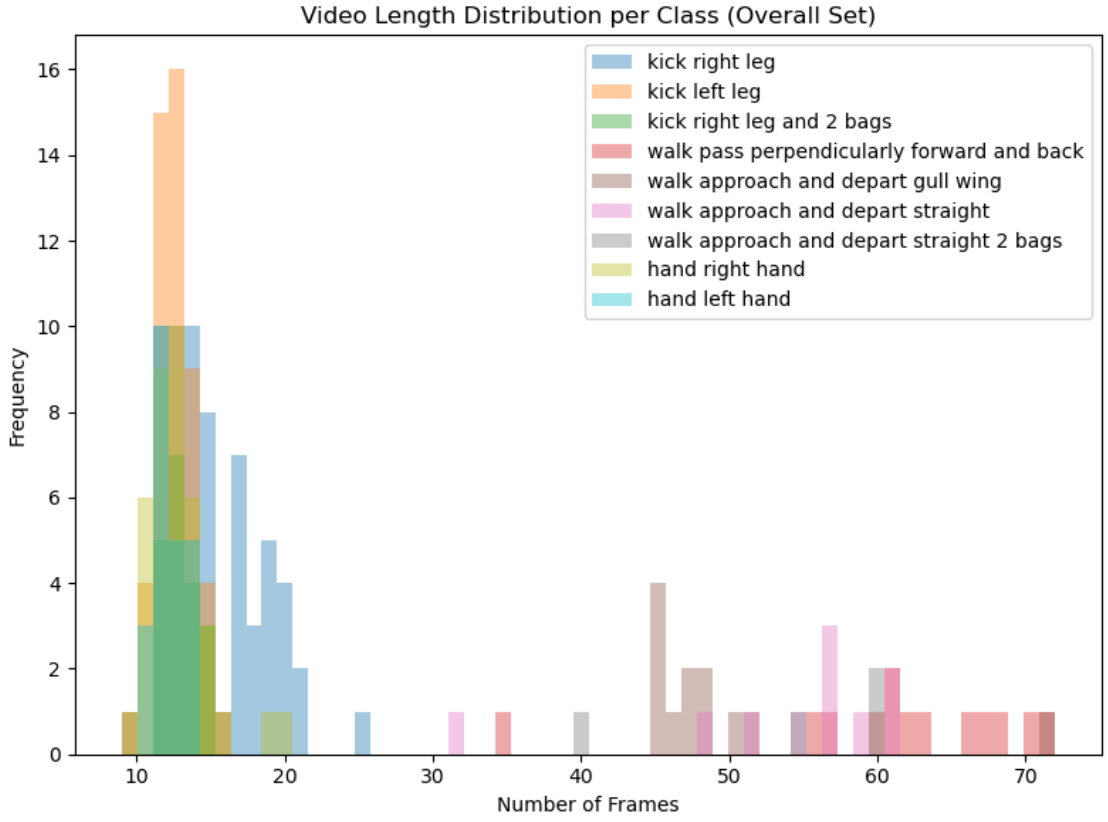


Figure 4.12: Illustration of the video lengths, in number of frames, per class in the case of the extended multi-class classification task.

4.3.3.1 Model evaluation

The dynamic model with extended multi-class classification achieved an accuracy of 85 %, as shown in table 4.14. This can be further visualized in the confusion matrix 4.14, as the diagonal is approximately 100 %.

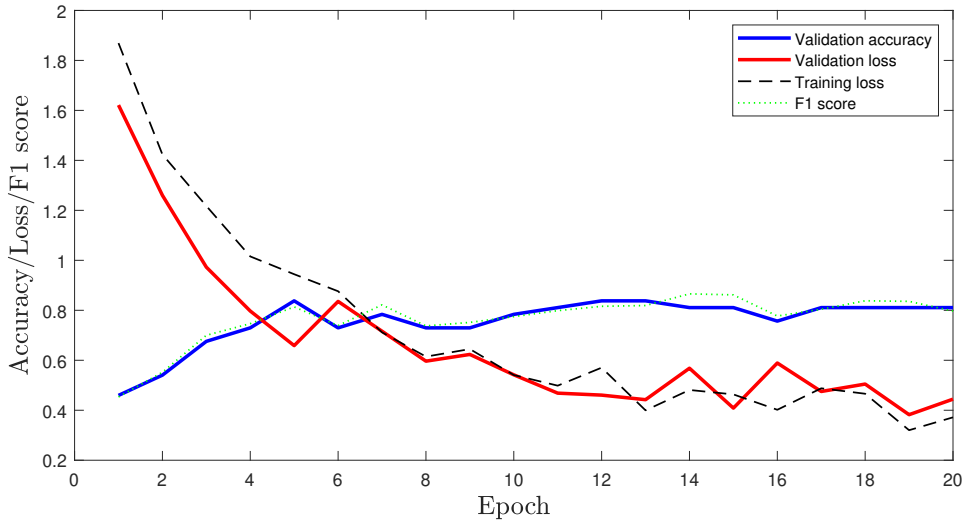


Figure 4.13: Illustration of the training and validation loss together with validation accuracy and F1 score over epochs

Table 4.13: The mean certainty and count for true positives, true negatives, false positives, and false negatives.

	Percentage of data	Quantity	Mean certainty
TP	33.33	7	2.886
TN	47.62	10	3.590
FP	4.76	1	1.013
FN	14.28	3	1.953

Table 4.14: Test metrics of dynamic vision model for extended multi-classification.

	Loss	**Accuracy**	**F1**
Test set	0.511	0.851	0.896

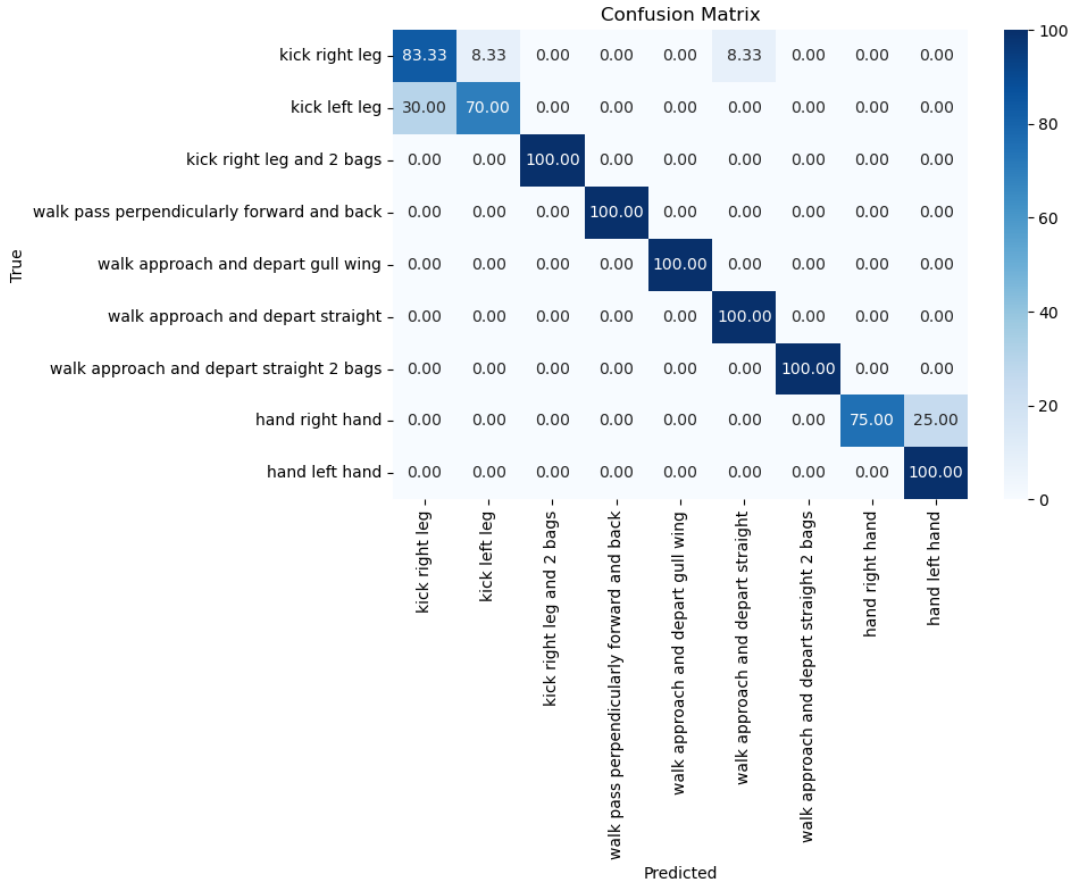


Figure 4.14: Illustration of the confusion matrix from the test evaluation

4.4 Combined model

The data fed into the combined model consisted of all gesture data of the kick motion and a similar amount of gesture data containing no kick for both models, i.e., image data and USS data.

4.4.1 Model evaluation

Combining the network models for static vision and USS and feeding randomly selected samples of gesture data and data with no gesture, the results in table 4.15 were obtained after 2000 iterations (to likely capture most of the unique data in the respective data sets).

Table 4.15: This table shows measures of the combined network model evaluation. Note that the network certainty is defined in a different way for the combined model.

	Percentage of data	Quantity	Mean certainty
TP	50	1000	11.279448
TN	50	1000	11.277735
FP	0.000	0	-
FN	0.000	0	-

The combined model's accuracy was 1.0. This accuracy can be compared to the models presented for the static vision network model, 1.0, and the USS network, 0.773.

4.4.2 Combined model using dynamic vision model

In tables 4.16 and 4.16, the results of processing the full dataset with the binary dynamic vision model for classifying the 'kick' - 'no kick' gesture are presented.

Table 4.16: Test metrics for dynamic vision model for multi-classification.

	Loss	**Accuracy**	**F1**
Test set	0.03134	0.995	0.995

Table 4.17: The mean certainty and count for true positives, true negatives, false positives, and false negatives.

	Percentage of data	Quantity	Mean certainty
TP	56.71	131	5.971
TN	42.86	99	3.938
FP	0.43	1	2.059
FN	0	0	-

5
Discussion

This chapter discusses the results and potential error sources and compares the tested models and the current radar-based systems.

5.1 Non neural network based approach

It is possible to use classical measures, such as the radar-based system, to activate the requested actuation using information from other sources. Still, there are some drawbacks as well. Without using a neural network approach to classify the movement signatures of intended gestures, it could be hard to determine whether a human made a gesture intending to open the trunk or not. More accurate real-time analysis would require significant computational resources compared to the ANN solution. Perhaps someone is walking by, or an object, for instance, an animal or plastic bag, moves close. It would not be optimal if such scenarios triggered the actuation. Therefore, one would need more information to avoid activating false positives, which would be quite an unpleasant or dangerous situation for customers. To combat this issue, one could use the key position as an indicator of where the driver is. If the person holding the key is standing behind the car and the classical method determines that the trunk should open, this information could potentially be matched and used to cause an intended actuation more reliably. However, that also means that only the person wearing the key can use this feature. This removes the possibility of, for instance, a family member using the feature without the key. Perhaps it would be enough to use only the ultrasonic distance measured, but there could still be false positives, such as if the driver walks by the car too close to the vehicle with the key. Then, this method would classify this as a gesture even though it is not. Of course, there are workarounds for this as well, for instance, if the driver has to stand a given time span at the right position for the trunk to open. However, this might be a bit unpractical.

5.2 USS model and data

The obtained data from the ultrasonic sensors only captured distance in one dimension, making the human gesture pattern hard to distinguish from a potential object over time. As seen in figures 3.7 and 3.6, each gesture has no clear, unique pattern. Due to measurement noise and errors, not all sampled measurement points in the data collection gave reasonable results, and the data density in time after the noise filtration was insufficient for the classification to be done reliably. For most

recorded measurements, one could see a curve pattern as the foot or hand came closer to the sensors for each time step, but sometimes these points were all noise or errors, making the measurement invalid or unsuitable for the training. This would also mean that this can occur in live data readings. There is also the concern that too many false positives could trigger inaccurate actuation using only this network model even if it would have classified with high accuracy. It is hard for even a human to interpret what is considered a kick or an unintentional movement from a human or object from this data. As the results suggest, one would most likely need more information to accurately determine what is a human gesture and what is not. Our ANN model could classify slightly more accurately than what just random classifications would yield. As mentioned, the lack of information is most likely the reason for these results.

Perhaps a higher sampling rate would show a more detailed positional change or movement, which, in turn, could be used to classify the gesture accurately. A more prominent curve would perhaps show a pattern in movement that could be identified. As of the current sampling rate, 50 Hz, with a kick duration that is roughly 1 second long, there should be close to 50 measurements. However with errors and noise, it was common to only get a few distance points in the interval of the kick. This is not always enough information to determine how the gesture signature. As seen in figure 4.1, the validation accuracy alternates up and down before reaching a somewhat stable state. This was also the case for several different types of training, which might be an indication of bad data being used, meaning data samples that do not belong to any class. The loss curve, however, has a relatively stable decent, but training the model more than shown in the figure would not yield any better results, and the loss evens out at around 25 epochs.

The results suggest that the network model seems to have difficulty classifying the USS data. The mean certainty is far below the certainty of the static vision model. There is also much less data on the USS model because it needs the whole time series for the classification and cannot use individual time point data as a separate input, such as for the static vision model. If, however, it would be possible to obtain one more spatial dimension of data from the USS, the method could be applied since there would be a clear signature for each frame or time point of data. So, if wanting to use only USS to classify the gesture, it is crucial to extract more information.

5.3 Static vision model

The static vision gesture model has several strengths but also some drawbacks. Since the network classifies static images from a video feed, no previous information is fed to the model, and each classification is made only from that instant in time. Information from several instances could be highly useful in detecting what is intended as a kick. This was even more apparent while labeling the data for this model. Sometimes, it was hard for a human to determine what was intended as a kick and what was not. While labeling, there is also the concern that one might influence the model by labeling frames too early or too late in the gesture motion. If frames

are labeled too early in the motion, there is a risk that the model will adapt to unintentional gestures or similar motions. As a human labeling the frames, one has the information that there is a more prominent gesture in the next frame, which would influence what can be considered a gesture, which this model cannot do. If, instead, frames are labeled too late, the model would adapt to the most extreme points in the recorded data, meaning that it would only classify extreme gestures as gestures. If this is not in sync with the sampling or if someone makes a lesser intended gesture, the model could have problems classifying this. Also, more data points would have to be collected for the same amount of data since each frame in each video feed can be used in the static vision dataset. Then at what frame of the video does the gesture start? Looking at a sequence of frames makes it easier. Since false positives are considered worse than false negatives for the project, the window in between the frame at which the gesture is assumed to start and the frame when it is assumed to stop was narrowed down such that the frames in the interval only contain clear indications of gestures. In turn, the model is less likely to trigger when for instance, a person is walking by. However, this might also mean that a person using this technology, such as a customer, has to exaggerate the gestures for it to work more reliably.

As seen in the figure 4.3, the model finds a good validation accuracy rather fast. After only 4 epochs, the model reaches a validation accuracy of around 90 %, which is then further increased but with a lower slope. The model showed signs of stability in between runs, meaning that it converges in a similar way, similar curvatures as shown in figure 4.3 would be obtained for different training iterations. The validation loss is also shown to decrease gradually, and after around 10 epochs, it stabilizes as the validation accuracy is 100% for two epochs in a row.

The model is rather small and does not require much computational resources, and it can classify at a high speed, making it possible to trigger actuation close to instantaneously. These aspects could be considered a great advantage of this model, especially with the great results shown by the model evaluation scores 4.2.

One should also note that the data that was used to train the network only contains three people and three different backgrounds. The results may vary from what is shown if applied to vastly different data, but the overall results indicate a very promising proof of concept for future development and implementation. With more diverse data to train on, this solution would have the possibility of yielding good results for all potential customers regardless of environment and appearance.

5.4 Dynamic vision model

As seen in table 4.16, the dynamic vision model achieved a good accuracy on the overall dataset. The results showed an exceptional performance that could compete with the combined model, although more computationally heavy due to being a much bigger network. The dynamic model achieved a loss of 0.03, an accuracy of 99.50%, and an F1 score of 0.99. These metrics emphasize the model's robustness

and precision in distinguishing between 'kick' and 'no kick' gestures. In table 4.17, it can be observed that only one instance was incorrectly classified as 'kick', which constitutes 0.43 % of the data. During the recording of the misclassified video, the person was intended to remain still without performing any gesture. However, the video started prematurely, capturing the person taking a small step to the left to adjust the position relative to the car. Moreover, another car passing by alongside the car added further disturbance. The model likely labels this as a 'kick' because it is the only video where both a sidestep and a moving car are within the field of view.

The complement of the temporal and spatial dimensions reinforces the capability of feature representation. The R(2+1)D architecture proved capable and requires further exploration compared to other architectures in terms of a more diverse dataset, computational load, memory capacity, and system integration in the final product. Furthermore, the transfer learning technique and fine-tuning of the model with the acquired dataset demonstrated its effectiveness in performance and implementation time. As stated in section 3.8.1, iterative testing showed that the number of fps was lowered from 30 to 2 without compromising the models' performance to any greater extent. The lowest limit of fps is dictated by how swiftly the gesture is performed and how efficiently the model captures the spatial and temporal features. Unlike the sampling of USS data, where most data frames/points are lost to noise, the video frames are assumed to be unaffected. Given the significant reduction in fps and the strong performance of the static vision model, the importance of the temporal dimension in binary classification tasks is questionable.

The dynamic model has many strengths but it also comes with some drawbacks. The model requires videos of fixed size and length, which may result in the loss of valuable spatial and temporal information, i.e., it lacks long-range temporal and wide-range spatial modeling. The video length is limited and in this thesis project that problem was solved through padding of the last frame in the video up to 70 frames. A probable step in development would have seen the employment of a sliding window. The window size would have been enough to fit the gestures intended for recognition but not much bigger to minimize the risk of capturing several gestures and also avoid delayed functional actuation. The required video size is achieved by resizing to (128, 171) and center cropping to (112, 112). Gestures performed in the outer extremities of the frame are consequently not processed by the model and can lead to an increase in FN.

As seen in figures 4.7a and 4.7b, the model performed well in training and validation. The F1 score and validation accuracy were high and stable from the start, and the slight fluctuations in the validation loss did not seem to impact the overall performance. This could indicate that the binary classification task might be too simplistic, and there is more potential to harness from the model. The implementation of additional intermediate hidden layers to the architecture of the dynamic vision network was therefore not further explored. In section 4.3.3.1, one can observe the confusion matrix 4.14 for the extended multi-class gesture classification model. It is clear that the model struggled slightly to distinguish some gestures,

for instance, between the left and right leg. Since the model is trained on an already small dataset, extending the number of classes further increases the risk of FN and FP. Extending the overall dataset is, therefore, the highest priority to further investigate the capability and performance of the dynamic model. Nevertheless, preprocessing configuration depends on the intended gesture to be classified, and the requirements of the NN might need to be revised.

5.5 Combined model

The combined network model uses a combination of the USS and Static vision models. Since there is a weight function based on the network certainty, the combined model depends on this function to work. The weight can also be tuned and scaled to various degrees, which makes it possible to increase or decrease the combined accuracy. The tuning of weight parameter and use of network certainty, can possibly lead to further increase in the overall model accuracy and limit the false positives. One should also note that the results for this project show 100 percent accuracy, but this is only for the data that was collected during this project. By definition, errors are more likely to occur for people who are not similar in appearance to the collected data. If trained on a larger, more varied pool of people, it is possible to decrease these potential errors. However, this might also give results that differ from ours. In this case, perhaps larger networks would fit the data better and make a more complex classification approximation.

Of course, several factors can contribute to errors. In fact, each step within the combined model introduces its own error source. The results show that 100 percent accuracy was achieved from this random sampling method, which would indicate that it works rather well. However, this is partial data that the network is trained on, and the data involves only three different people making the gestures.

Regarding the overall structure, it can be wise to consider internal loops to combat the vision system turning on and off constantly. This could be problematic if the startup time is too long. For instance, maybe after the vision system is started for the first time, it will stay on and loop for a given time frame before it turns off. That way, the system does not have to reboot each time a gesture is not detected.

The weight function can be tuned with a parameter, which in turn can be used to optimize the behavior of the combined model manually, but it also has the potential to be adjusted by some other external sensors or logic. For instance, when it is dark outside, the weight function could be tuned such that it would lean more on the USS model since the ultrasound is not affected by light, or when it is snowing, adjust the parameter so that the model will prioritize the vision model.

5.6 Compared to current solution

According to the function owner, the currently integrated solution has an accuracy of around 96 %. Compared to the best models, the vision-based and combined models show the potential to outperform the current solution. However, as stated before, the data and test sets used in the thesis project are small and lack diversity, which may not have been the case for the current model. The results are, therefore, not directly comparable. However, as the proposed models obtain 100% accuracy, this is an indicator that the approach has great potential, especially since it, for this scale, outperforms the current radar-based system.

5.7 Data distribution

Since the recording sessions were confined to a specific geographical location, namely the company's premises, they may limit the proposed models' ability to be generalized. In addition, the project was conducted over six months and does not account for seasonal variations or changes over longer periods. As the diversity of the participants was limited, the models might perform worse given a wider demographic range and behavioral patterns.

6
Conclusion

In conclusion, three neural network-based models have been developed based on the vision data and ultrasonic sensor data in a modern vehicle. All models show promising results for future implementation and further development as a gesture recognition system in cars. This thesis shows that it is possible to use a static vision model that analyses static frames in a video feed to achieve high performance and precision classification for gestures, especially in combination with ultrasonic sensor data.

A dynamic model approach is also shown to achieve great precision for the specified and more specific gestures. The model requires a higher computational cost but can be used successfully on small datasets because of its nature.
The given results of this thesis project show that the developed models have the potential to outperform the currently integrated radar-based system, which has an accuracy of 96 %.

7
Future work

7.1 Dataset expansion

The dataset created for this thesis is considered small in several aspects, which limits the potential of the models developed. To further investigate the capability and enhance the performance and robustness of the models presented, it is crucial to extended the dataset used for training, validation, testing, and evaluation [25].This expansion should address two primary aspects. Firstly, the number of classes should be increased to represent a broader range of real-world scenarios that can occur in the surroundings of a vehicle. For instance, include the leg swipe motion seen in figure 7.1. Secondly, each class requires a wide diversity in both human features like body size, postures, appearances, and environmental factors like illumination, and disturbances, such as background noise. By expanding the dataset, it is expected to enhance the generalization of the model.

Figure 7.1: Illustration of the leg swipe motion.

7.2 USS model

Continuing with the USS model, one could investigate the possibility to triangulate USS data to gain more spacial dimensions using for instance methods from computer

vision camera triangulation. Using this approach one could also potentially reduce the amount of noise in the data which would in turn also help a NN model to classify the data. With more spacial dimensions in the data the network structure would have to be different. Perhaps some sort of CNN approach would be suitable where as if three spacial dimensions can be obtained, the third dimension could be integrated in the model as a form of color gradient and then one could evaluate static USS taking only one instance in time into consideration and a dynamic USS just as for the vision models investigated in this thesis project. Perhaps this could be preferable to the vision based systems when taking power consumption into consideration.

7.3 Static vision models

The static vision models for two or three classes could both be expanded upon using a more diverse and bigger dataset. This would potentially also mean that the networks need to be expanded in size to be able to handle the introduced diversity in data. This evaluation is something that would be very relevant for continuing on this project.

7.4 Improved performance of ResNet

To further enhance the performance of the dynamic model, [25] proposes improvements, consisting of training methods, regularization methods and architecture changes, that are suitable to explore. In [25], the application of these improvements to a 3D ResNet-50 model yielded in an accuracy increase of nearly 5% compared to the baseline of 73.4% on the Kinetics-400 dataset. The improvements to the 3D ResNet-50 model consisted of extending the number of epochs during training and implementing regularization methods such as dropout on fully connected layers, label smoothing, stochastic depth, exponential moving average of weights, decreased weight decay and scale jittering.

Additionally, two architectural modifications were applied to all bottleneck blocks: Squeeze-and-Ecitation and ResNet-D. The Squeeze-and-Ecitation module re-calibrates channel weights through cross-channel communication achieved by averaging pooled signals across the entire feature map. The ResNet-D architecture replaces the 7×7 convolution in the stem with three 3×3 convolutions, adjusting stride sizes in downsampling blocks, substituting stride-2 1×1 convolution with stride-2 2×2 average pooling followed by non-strided 1×1 convolution, and removing the stride-2 3×3 max pool layer. These modifications aim to enhance feature representation and model efficiency.

7.5 Improved approach for videos with arbitrary size and length

The (2+1)D network deployed in this thesis requires videos of fixed size and length, which may result in the loss of valuable spatial and temporal information, i.e. it lacks the long range temporal and wide range spacial modeling. The video length is limited, realized in this thesis through padding of the last frame. The required video size is achieved by resizing to (128, 171) and centre cropping to (112, 112). These fixed input sizes are necessary due to the fully connected layers' fixed size requirement. However, required transformation of size can lead to the object of interest, such as an arm or a leg, being partially or entirely outside the resulting region. In addition, the already warped image produced by the fish-eye camera is warped again, which might lead to undesirable geometric distortions. The required transformation of the length can lead to reduced recognition performance as it affects motion completeness by dividing the video into clips.

Multiple studies exist which explore suitable approaches for dealing with data with arbitrary size and length by combining CNN-based and RNN-based modeling[26]. For instance, [26], presents a novel end-to-end approach named "Two-stream 3D ConvNet", which incorporates a spatial pyramid pooling method, a multi-level pooling technique to extract equal dimension descriptors, into a general 3D CNN and then adopts a LSTM or a convolutional layer embedding. This approach, according to [26], should improve all CNN-based video analysis algorithms. It is therefore deemed suitable to explore this approach with R(2+1)D with the aim of further increasing accuracy and allowing for a larger variety of motions to be classified.

7.6 Technological Scope

In this project, only the available ultrasonic sensors and the camera positioned at the rear of the car were utilized. Since the car is able to detect and track the position of the key fob and establish a user intention profile, it is possible to add an additional input to the models to further increase their robustness. Given the possibility to utilize the smartphone as a car key it is also suitable to explore the integration of the extensively researched sensor based HMR method [4] to improve the model performance.

7.7 Combined model

Regarding the overall logic structure as seen in 3.3, it can be wise to consider the idea of internal loops to combat the potential scenario where the vision system is turned on and off all the time. This could be problematic if the startup time for this system is much longer that the sample frequency of the sensors. For instance maybe after the vision system is started for the first time it will stay on and loop for a given time frame before it turns off. That way the system does not have to

reboot each time a gesture is not detected.

/

Bibliography

[1] Lambert F. *Tesla engineers tried to convince Elon Musk not to give up radar for self-driving*. 2023. URL: https://electrek.co/2023/03/21/tesla-engineer-convince-elon-musk-not-give-up-radar-self-driving/#:~:text=The%20automaker%20decided%20to%20remove,cameras%20being%20the%20only%20sensors..

[2] Liangliang Zhang. "Applying Deep Learning-Based Human Motion Recognition System in Sports Competition". In: *Frontiers in Neurorobotics* 16 (2022). ISSN: 1662-5218. DOI: 10.3389/fnbot.2022.860981. URL: https://www.frontiersin.org/articles/10.3389/fnbot.2022.860981.

[3] Guilherme Augusto Silva Surek et al. "Video-Based Human Activity Recognition Using Deep Learning Approaches". In: *Sensors* 23.14 (2023). ISSN: 1424-8220. DOI: 10.3390/s23146384. URL: https://www.mdpi.com/1424-8220/23/14/6384.

[4] Wei Zhong Tee et al. *A Close Look into Human Activity Recognition Models using Deep Learning*. 2022. arXiv: 2204.13589 [cs.CV].

[5] Hong-Bo Zhang et al. "A Comprehensive Survey of Vision-Based Human Action Recognition Methods". In: *Sensors* 19.5 (2019). ISSN: 1424-8220. DOI: 10.3390/s19051005. URL: https://www.mdpi.com/1424-8220/19/5/1005.

[6] Bernhard Mehlig. *MACHINE LEARNING WITH NEURAL NETWORKS*. Cambridge University Press, 2022. DOI: 10.1017/9781108860604.

[7] Sarker I. H. "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions." In: *SN computer science* (2021).

[8] Howard B. Demuth Martin T. Hagan, Mark H. Beale, and Orlando D. Jesús. *Neural Network Design*. URL: https://hagan.okstate.edu/NNDesign.pdf. 2024-04-18.

[9] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[10] Tim Johansson Roman Erkens Arthur von Kaenel and Robin Hanik. "Face deformation based on face detection and emotion recognition". In: (2023).

[11] Abdulazeez Alsajri and Vugar Abdullayev. "Review of deep learning: Convolutional Neural Network Algorithm". In: *Babylonian Journal of Machine Learning* 2023 (Apr. 2023), pp. 19–25. DOI: 10.58496/BJML/2023/004.

[12] Md. Milon Islam et al. "Human activity recognition using tools of convolutional neural networks: A state of the art review, data sets, challenges, and future prospects". In: *Computers in Biology and Medicine* 149 (2022), p. 106060. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2022.

106060. URL: https://www.sciencedirect.com/science/article/pii/S0010482522007739.

[13] Du Tran et al. "A Closer Look at Spatiotemporal Convolutions for Action Recognition". In: *CoRR* abs/1711.11248 (2017). arXiv: 1711.11248. URL: http://arxiv.org/abs/1711.11248.

[14] IBM. *What is overfitting?* URL: https://www.ibm.com/topics/overfitting.

[15] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. "What is being transferred in transfer learning?" In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 512–523. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/0607f4c705595b911a4f3e7a127b44e0-Paper.pdf.

[16] *How do you integrate image transformation with convolutional neural networks into your workflow and pipeline?* URL: https://www.linkedin.com/advice/0/how-do-you-integrate-image-transformation-convolutional.

[17] *To what resolution should I resize my images to use as training dataset for deep learning?* 2017. URL: https://www.quora.com/To-what-resolution-should-I-resize-my-images-to-use-as-training-dataset-for-deep-learning.

[18] *Accuracy vs. precision vs. recall in machine learning: what's the difference?* 2024. URL: https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall.

[19] Iowa State University. *Temperature and the speed of sound*. Accessed: 2024-03-01. URL: https://www.nde-ed.org/Physics/Sound/tempandspeed.xhtml.

[20] Hsien-I Lin, Ming-Hsiang Hsu, and Wei-Kai Chen. "Human hand gesture recognition using a convolution neural network". In: *2014 IEEE International Conference on Automation Science and Engineering (CASE)*. 2014, pp. 1038–1043. DOI: 10.1109/CoASE.2014.6899454.

[21] Li G et al. "Hand gesture recognition based on convolution neural network". In: *Springer* (2017).

[22] *Pre-trained video classification models*. Accessed: 2024-03-01. URL: %https://pytorch.org/vision/stable/models.html#video-classification.

[23] Olena Pavliuk, Myroslav Mishchuk, and Christine Strauss. "Transfer Learning Approach for Human Activity Recognition Based on Continuous Wavelet Transform". In: (2023). URL: https://www.mdpi.com/1999-4893/16/2/77.

[24] *Torchvision pre-trained residual 2D plus 1D CNN architecture R(2+1)D of 18 layers*. Accessed: 2024-03-01. URL: https://pytorch.org/vision/stable/models/generated/torchvision.models.video.r2plus1d_18.html#torchvision.models.video.R2Plus1D_18_Weights.

[25] Irwan Bello et al. "Revisiting ResNets: Improved Training and Scaling Strategies". In: *CoRR* abs/2103.07579 (2021). arXiv: 2103.07579. URL: https://arxiv.org/abs/2103.07579.

[26] Xuanhan Wang et al. "Two-Stream 3-D convNet Fusion for Action Recognition in Videos With Arbitrary Size and Length". In: *IEEE Transactions on Multimedia* 20.3 (2018), pp. 634–644. DOI: 10.1109/TMM.2017.2749159.

A

Appendix 1

A.1 Preprocessing: Decoding

The preprocessing stage involving decoding of the acquired mf4-files, necessitated the use of already established decoding utilities provided by the company's cloud-based computer services. Extracting the utilities to a local workstation ensured uninterrupted access, thus mitigating the risk of interfering with concurrent projects requiring the same resources. To streamline the decoding process, the script `decode_logg.py`, in Appendix A.1, was created. This script, iteratively retrieved the data from the designated input directory, called the required decoding utility to process the data, and stored the resultant data in a specified output directory. The directory structure consisted of folders for each separate measurement, containing one or more mf4-files, depending on the length of the recording. The choice of decoding utility was contingent upon the origin of the data: USS data was converted into hdf5-format, while the vision data was converted into mp4-format.

Execution of the script `decode_logg.py` and the decoding utilities necessitated a Linux operating system, as detailed in Figure A.1. The tool that decoded the vision data, in particular, must operate within an Nvidia CUDA environment to leverage the GPU's computational capabilities for processing large video files efficiently. The tool that decoded the USS data required the Singularity container.

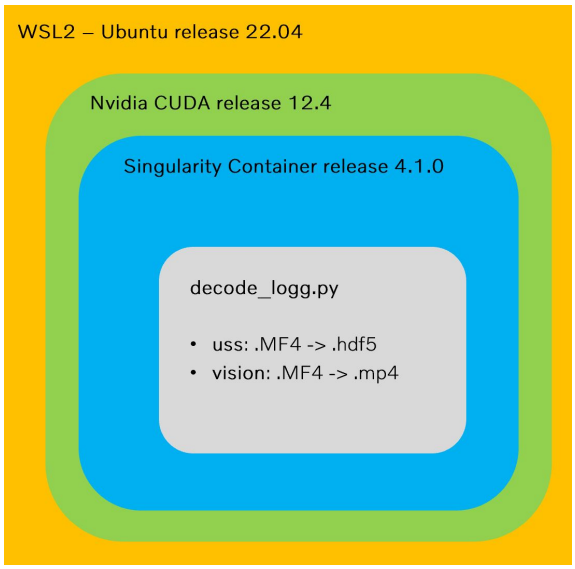


Figure A.1: Overview of the system environment configuration for preprocessing of USS and vision data

A.1.1 Decoding recorded files

Listing A.1: Script for decoding of USS and vision recordings

```
#-----------------------------------------
#Decoding MF4 files to MP4 and hdf5
#Version 1.0
#-----------------------------------------
#Authors: Tim Johansson and Krister Mattsson
#-----------------------------------------
import os
import glob
import subprocess

# conda activate dml
# Run under directory: ~/decoding
USER = os.getenv('USER')
default_path = f"/home/{USER}/master_thesis/decoding"
os.chdir(default_path)

# Define input and output directories
input_base = f"/home/{USER}/master_thesis/data/input/"
#logg_name= "Logg_HPG_2024-02-19"
logg_name= "Logg_HPG_2024-03-25"
input_dir_srcb = f"{input_base}{logg_name}/*/srcb/"
input_dir_uss = f"{input_base}{logg_name}/*/uss/*.MF4"
output_base = f"/home/{USER}/master_thesis/data/output"


# Loop through input directoriesof uss
for dir_path in glob.glob(input_dir_uss):
        # Get the folder name from the input path
```

```
        copy_folder_name = os.path.basename(os.path.dirname(os.path
            .dirname(dir_path)))

        # Create the corresponding output directory with the
            correct structure
        output_dir = os.path.join(output_base, f"{logg_name}/{
            copy_folder_name}/uss/")

        # Create the output directory if it doesn't exist
        os.makedirs(output_dir, exist_ok=True)

        # Run the command to convert .MF4 files to videos
        # Singularity: https://docs.sylabs.io/guides/4.1/user-guide
            /quick_start.html
        command = [
            "singularity",
            "run",
            f"/home/{USER}/master_thesis/decoding/vccd.sif",
            "decode",
            dir_path,
            "-i",
            f"LPCETHGW,MRCCAN,src_ip=xxx.xx.xx.xx+src_udp=2990+
                dst_ip=xxx.xx.xx.xx+dst_udp=2990,/home/{USER}/
                master_thesis/decoding/LPCSystem_LpcEthToCanLpcDp.
                dbc",
            "-gen_hdf5",
            "-out_dir",
            output_dir
        ]

        try:
            subprocess.run(command, check=True)
            print(f"Converted {dir_path} to HDF5 in {output_dir}")
        except subprocess.CalledProcessError as e:
            print(f"Conversion failed for {dir_path}. Error: {e}")

# Loop through input directoriesof srcb
for dir_path in glob.glob(input_dir_srcb):
    if os.path.isdir(dir_path):
        # Get the folder name from the input path
        copy_folder_name = os.path.basename(os.path.dirname(os.path
            .dirname(dir_path)))

        # Create the corresponding output directory with the
            correct structure
        output_dir = os.path.join(output_base, f"{logg_name}/{
            copy_folder_name}/srcb/")

        # Create the output directory if it doesn't exist
        os.makedirs(output_dir, exist_ok=True)

        # Run the command to convert .MF4 files to videos
        command = [
            "python",
            f"/home/{USER}/master_thesis/decoding/
                image_log_to_video.py",
```

```
            dir_path,
            "-o",
            output_dir
        ]

        try:
            subprocess.run(command, check=True)
            print(f"Converted {dir_path} to videos in {output_dir
                }")
        except subprocess.CalledProcessError as e:
            print(f"Conversion failed for {dir_path}. Error: {e}")
```

A.2 Python code

A.2.1 USS model

Listing A.2: Script for exporting USS data to time windows in csv format.

```
from cmath import nan
import numpy as np
import h5py
import copy
import glob
import matplotlib.pyplot as plt

def saveCSV(echoDists):
    np.savetxt('kickDataMergedFilter.csv', echoDists, delimiter
        =",")
    print('Successfully saved')

#rootDir = 'XXX'
picked_signals = {}
delta_ulsTimestamp = {}
signals = [
    "BlindDist",
    "MaxDetnDst",
    "ulsTimestamp",
    "Echo1Dist",
    "Echo2Dist",
    "Echo1Amp",
    "Echo2Amp",
    "Echo3Dist"
    ]

echoDistTot = []

for folder in glob.glob('C:\XXX\\Desktop\\ultrasonicsensor2.0\\data
    \\decoded\\Logg_HPG_2024-02-19\\'+'/*/'):
    #print(folder) C:\XXX\\Desktop\ultrasonicsensor2.0\data\decoded
        \Logg_HPG_2024-02-19\003535
    rootDir = folder + 'uss\\'

    for files in glob.iglob('*.hdf5', root_dir=rootDir):
```

```
    with h5py.File(rootDir+files,'r') as hf:
        #print('List of arrays in this file: \n', hf.keys())
        # Get the first key in the list that is the group:
            LPCETHGW || UMF
            f_key = list(hf.keys())[0]
            # print(f"The key of the file are {f_key=}")

            # Get the elements in LPCETHGW || UMF
            interface_name = hf.get(f_key)
            interface = dict(interface_name.items())

            # picked frames:
            picked_frame = []
            for frN, frSg in interface.items():
                for sgN in frSg.keys():
                    if sgN.startswith("NFSEchoMeas") and sgN
                       [12:13] != "C" and frN not in
                       picked_frame:
                        picked_frame.append(frN)

            # Create a dictionarie of the signals to be
               compared
            for frame in picked_frame:
                for signal in interface[frame].keys():
                    for sg_name in signals:
                        if signal.endswith(sg_name) and signal
                           [12:13] != "C":
                            if signal[11:13] not in
                               picked_signals.keys():
                                picked_signals[signal[11:13]] =
                                     {sg_name: list(interface[
                                    frame][signal])}
                            else:
                                picked_signals[signal[11:13]][
                                    sg_name] = list(interface[
                                    frame][signal])
                # if signal == "Timestamp" and frame[15:16] in
                   picked_signals.keys():
                #     picked_signals[signal[11:13]]["Timestamp
                   "] = list(interface[frame][signal])
            if f_key == 'XXX':
                uss = copy.deepcopy(picked_signals)
            if f_key == 'XXX':
                umx = copy.deepcopy(picked_signals)

#Extract information
#R1Echo = np.array(uss['1R']['Echo1Dist'])

#Echo1 RIR and RIL
R4EchoDist1 = [float(arr[0]) for arr in uss['4R']['Echo1Dist']]
R5EchoDist1 = [float(arr[0]) for arr in uss['5R']['Echo1Dist']]

mergedEcho = np.zeros(262) #MAX LENGTH
   ----------------------------------------

for i in range(len(R4EchoDist1)-1):
```

```
        #Filtration/noice reduction
        if R4EchoDist1[i] > 60000 or R4EchoDist1[i] < 2:
            R4EchoDist1[i] = 0
        if R5EchoDist1[i] > 60000 or R5EchoDist1[i] < 2:
            R5EchoDist1[i] = 0

        #Merge logic
        mergedEcho = R5EchoDist1
        if R4EchoDist1[i] == 0:
            if R5EchoDist1[i] == 0:
                mergedEcho[i] = 0
            else:
                mergedEcho[i] = R5EchoDist1[i]
        elif R5EchoDist1[i] == 0:
            if R4EchoDist1[i] == 0:
                mergedEcho[i] = 0
            else:
                mergedEcho[i] = R4EchoDist1[i]


    plt.plot(R4EchoDist1,'r.')
    plt.plot(R5EchoDist1,'b.')
    plt.plot(mergedEcho,'g.')
    plt.ylabel('detected distance')
    plt.xlabel('time step')
    plt.show()

    #Append to save Matrix if resonable
    counter = 0
    for dataPoint in mergedEcho:
        if dataPoint > 0 and dataPoint < 1000:
            counter = counter + 1

    if counter > 1:
        echoDistTot.append(mergedEcho)




#plt.show()

#Find the length of the largest array
max_length = 262 #max(len(array) for array in echoDistTot)

#Initialize an empty list to store padded arrays
padded_arrays = []

# Pad shorter arrays with NaN values
for array in echoDistTot:
    padded_array = np.pad(array, (0, max_length - len(array)), mode
       ='constant', constant_values=0)
    padded_arrays.append(padded_array)

processedEchoDists = np.array(padded_arrays)
```

```
saveCSV(processedEchoDists)
```

Listing A.3: USS network.

```
#-------------------------------------------
#Neural network USS v1.0
#-------------------------------------------
#Authors: Tim Johansson and Krister Mattsson
#-------------------------------------------

import torch
import torch.nn as nn

class USSNetwork(nn.Module):
    def __init__(self, num_inputs, num_classes):
        super(USSNetwork, self).__init__()
        #Fully connected layers
        self.fc1 = nn.Linear(num_inputs, 100, bias = True)
        self.af1 = nn.Tanh()
        self.fc3 = nn.Linear(100,64, bias = True)
        self.fc4 = nn.Linear(64,32, bias = True)
        self.fc5 = nn.Linear(32, num_classes, bias = True)
        self.af3 = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.af1(x)
        x = self.fc3(x)
        x = self.af1(x)
        x = self.fc4(x)
        x = self.af1(x)
        x = self.fc5(x)
        return x


#Test network
num_classes = 2
num_inputs = 10
model = USSNetwork(num_inputs, num_classes)

# Print the model architecture
print(model)
```

Listing A.4: Script for building, training and validate USS network model.

```
#-------------------------------------------
#Main file for USS network kick classification
#Version 2.3
#Refined data and several trainings
#-------------------------------------------
#Authors: Tim Johansson and Krister Mattson
#-------------------------------------------

import os
import torch
from torchvision import datasets, transforms
```

```
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import csv
import torch.nn.functional as Functional
import pandas as pd
from sklearn.model_selection import train_test_split

#Import netowrk script
import USSnet

USER = os.getenv('USER')

seed = 100
#torch.manual_seed(seed)
g = torch.Generator()
g.manual_seed(seed)
np.random.seed(seed)

#Load CSV as DataLoader
class CSVDataset(Dataset):
    def __init__(self, csv_file1, csv_file2):
        self.data1 = pd.read_csv(csv_file1)
        self.data2 = pd.read_csv(csv_file2)

    def __len__(self):
        return len(self.data1)

    def __getitem__(self, idx):
        # Get data samples by index
        sample1 = torch.tensor(self.data1.iloc[idx,:].values, dtype
           =torch.float32)
        sample2 = torch.tensor(self.data2.iloc[idx,:].values, dtype
           =torch.float32)

        return sample1, sample2


#Function to save the trained model
def save_model(model, filepath):
    torch.save(model.state_dict(), filepath)
    print(f"Model saved to {filepath}")

#Parameters---------------------------------
num_epochs = 26 #26
validation_accs = np.zeros(num_epochs)
batchSize = 10 #10

#Network specifications
num_classes = 2
num_inputs =  262 #shape[1] #Length of csv data
USSmodel = USSnet.USSNetwork(num_inputs, num_classes)
optimizer = optim.Adam(USSmodel.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss()
val_accuracy = 0.0
```

```
#Print the model architecture
print(USSmodel)


#Training
def train(num_epochs):
    USSmodel.train()
    running_loss = 0.0
    for epoch in range(num_epochs):
        for data1, data2 in train_loader:

            #Define inputs and labels. Note that labels are defined
                as either 1 or 0 where data1 => 0 which would
               corralate to file_path1 => 0
            inputs = torch.cat((data1, data2), dim=0)
            labels = torch.cat((torch.zeros(data1.size(0)), torch.
               ones(data2.size(0))), dim=0).long()

            optimizer.zero_grad()

            outputs = USSmodel(inputs)
            labels = labels.unsqueeze(1)
            loss = Functional.mse_loss(outputs, labels.float())

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        #Print training loss for this epoch
        print(f"Training Loss: {running_loss / (2*len(train_loader)
           )}")
        return outputs

#Validation
def validate():
    USSmodel.eval()
    val_loss = 0.0
    correct_predictions = 0

    with torch.no_grad():
        for data1, data2 in val_loader:

            #Define inputs and labels. Note that labels are defined
                as either 1 or 0 where data1 => 0 which would
               corralate to file_path1 => 0
            inputs = torch.cat((data1, data2), dim=0)
            labels = torch.cat((torch.zeros(data1.size(0)), torch.
               ones(data2.size(0))), dim=0).long()

            outputs = USSmodel(inputs)
            labelsT = labels.unsqueeze(1)
            loss = Functional.mse_loss(outputs, labelsT.float())

            #Validation calculation
```

```
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            #predicted = torch.transpose(predicted,0,1)
            correct_predictions += torch.eq(predicted, labels).sum
               ()

    val_accuracy = 100.0 * correct_predictions / (2*len(val_dataset
       ))
    print(f"Validation Loss: {val_loss / (2*len(val_loader))},
       Validation Accuracy: {val_accuracy}%")
    return val_accuracy, predicted, loss, labels



#Merge data
file_path1 = 'C:\\{USER}\\Desktop\\ultrasonicsensor2.0\\KickData\\
   emptyData.csv'
file_path2 = 'C:\\{USER}\\Desktop\\ultrasonicsensor2.0\\KickData\\
   kickDataMergedFilter.csv'

dataset = CSVDataset(file_path1, file_path2)
#dataLoader = DataLoader(dataset, batch_size=batchSize, shuffle=
   True)

#Splitting dataset into Training 80%, Validation 10%, and Test 10%
train_dataset, val_dataset = train_test_split(dataset, test_size
   =0.2)
test_dataset, val_dataset = train_test_split(val_dataset, test_size
   =0.5)

train_loader=DataLoader(train_dataset, batch_size=batchSize,
   shuffle=True, generator=g)
val_loader=DataLoader(val_dataset, batch_size=batchSize, shuffle=
   True, generator=g)
#test_loader=DataLoader(train_dataset, batch_size=batchSize)

#TRAIN SEVERAL TIMES
for training in range(10):

    #Train and validate for each epoch
    losses = np.zeros(num_epochs)
    for i in range(num_epochs):
        outputs = train(1)
        valAcc, predicted, lossTmp, targets = validate()
        valAcc = torch.Tensor.numpy(valAcc)
        lossTmp = torch.Tensor.numpy(lossTmp)
        validation_accs[i] = valAcc
        losses[i] = lossTmp
        print(f"Epoch {i+1}/{num_epochs}")

    predicted = torch.Tensor.numpy(predicted)
    targets = torch.Tensor.numpy(targets)

    print("Predictions: ",predicted)
    print("Targets:     ",targets.T)
```

```
    plt.plot(validation_accs,'b')
    plt.plot(losses*30,'r') #Scaled by 30
    plt.ylabel('Validation accuracy')
    plt.xlabel('Epoch')

plt.show()

save_model(USSmodel, "USS_Network\\models\\model3.pth")
```

A.2.2 Static vision model

Listing A.5: Script for pre processing vision data for the static vision model.

```
from logging import root
import numpy as np
import os
import cv2
import glob

USER = os.getenv('USER')

#Parameters
scale = 0.1 #Factor which determines the resolution

def reSize(frame, scale):
    height, width, _ = frame.shape

    new_width = int(width*scale)
    new_height = int(height*scale)

    #Set new size
    resized_frame = cv2.resize(frame, (new_width, new_height),
       interpolation=cv2.INTER_LINEAR)

    return resized_frame

def extract_frames(video_path, output_folder, frame_count):
    #Open the video file
    video_capture = cv2.VideoCapture(video_path)

    #Check if the video file is opened successfully
    if not video_capture.isOpened():
        print("Error: Unable to open video file.")
        return

    while True:
        ret, frame = video_capture.read()

        #If the frame is not read properly, the video is over
        if not ret:
            break

        #Resize the frame with lower resolution
        rFrame = reSize(frame,scale)
```

```
        #Save frame as image
        frame_filename = f"{output_folder}/frame_{frame_count:04d}.
            jpg"
        cv2.imwrite(frame_filename, rFrame)

        frame_count += 1

    #Release the video capture object and close all windows
    video_capture.release()
    cv2.destroyAllWindows()

    return frame_count


frame_count = 0

for folder in glob.glob('{USER}\\data\\'+'/*/'):
    root_dir = folder + 'srcb\\'
    #data\003535\srcb\spa2adas_turtle_xatXXX_srcb.MF4.mp4
    for files in glob.iglob('*.mp4', root_dir=root_dir):
        #Provide the path to the video file
        video_path = root_dir + files

        #Specify the folder where frames will be saved
        output_folder = "frames\\" + folder

        #Create the output folder
        os.makedirs(output_folder, exist_ok=True)

        #Call the function to extract frames
        frame_count = extract_frames(video_path, output_folder,
            frame_count)
        print('success!')
```

Listing A.6: Static vision network model

```
#-----------------------------------------
#Neural network Static Vision v2.0
#-----------------------------------------
#Authors: Tim Johansson and Krister Mattsson
#-----------------------------------------

import torch
import torch.nn as nn

class VisionNet(nn.Module):
    def __init__(self, num_classes):
        super(VisionNet, self).__init__()

        #Convolutional layers -> In channles for RGB, Greyscale ->
            1
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32,
            kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
```

```
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
           kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64,
           kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        #Fully connected layers
        self.fc1 = nn.Linear(32000, 5*50)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(5*50,64)
        self.relu3 = nn.ReLU()
        self.fc3 = nn.Linear(64, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        x = x.view(x.size(0), -1)  #Flatten output for the fully
           connected layers
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        x = self.relu3(x)
        x = self.fc3(x)

        return x

#Test network
num_classes = 2
model = VisionNet(num_classes)

# Print the model architecture
print(model)
```

Listing A.7: Script for building, training and validating the static vision model.

```
#-----------------------------------------
# Main Static Vision motion identification network
# Version 2.3
# --Implemented: Kick, human, Save model
#-----------------------------------------
# Authors: Tim Johansson and Krister Mattson
#-----------------------------------------

import os
```

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import Vision_Network
import glob

torch.manual_seed(100)
USER = os.getenv('USER')

#Generate network
NN = Vision_Network.VisionNet(2)
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(NN.parameters(), lr=0.005, momentum=0.9)
val_accuracy = 0.0

def countData():
    kickCount = 0
    nonKickCount = 0
    for files1 in glob.iglob('*.jpg',root_dir='{USER}\\KickData\\
       kick\\'):
        kickCount = kickCount + 1
    for files2 in glob.iglob('*.jpg',root_dir='{USER}\\KickData\\
       noKick\\'):
        nonKickCount = nonKickCount + 1

    print("########################################")
    print("Nuber of kicks: ", kickCount )
    print("Number of nonkickdata: ", nonKickCount )
    print("########################################")

#Function to save the trained model
def save_model(model, filepath):
    torch.save(model.state_dict(), filepath)
    print(f"Model saved to {filepath}")

#Training
def train(num_epochs):
    NN.train()
    running_loss = 0.0
    for epoch in range(num_epochs):

        for images, labels in train_loader:
            optimizer.zero_grad()

            outputs = NN(images)

            loss = criterion(outputs, labels)
            loss.backward()

            optimizer.step()
            running_loss += loss.item()

        # Print training loss for this epoch
```

```
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss /
            len(train_loader)}")

def validate():
    NN.eval()
    val_loss = 0.0
    correct_predictions = 0

    with torch.no_grad():
        for images, labels in val_loader:
            outputs = NN(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct_predictions += (predicted == labels).sum().item
                ()

    val_accuracy = 100.0 * correct_predictions / len(val_dataset)
    print(f"Validation Loss: {val_loss / len(val_loader)},
        Validation Accuracy: {val_accuracy}%")
    return val_accuracy, val_loss/len(val_loader)


#Define data transformations
transform = transforms.Compose([
    transforms.Resize((100, 80)),   #Resize images to desired size
    transforms.Grayscale(num_output_channels=1),  # Convert images
        to grayscale
    transforms.ToTensor(),           #Convert images to PyTorch
        tensors
    transforms.Normalize((0.5,),(0.5,))
])

#Data set
train_dataset = datasets.ImageFolder(root='KickData\\', transform=
    transform)

#Define the size of the validation set
validation_split = 0.2
dataset_size = len(train_dataset)
val_size = int(validation_split * dataset_size)
train_size = dataset_size - val_size

#Split the dataset into training and validation sets
train_dataset, val_dataset = torch.utils.data.random_split(
    train_dataset, [train_size, val_size])


#Create DataLoader instances for training and validation datasets
batch_size = 15 #5
train_loader = DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

#NUMBER OF EPOCHS--------------------------
num_epochs = 10
```

```
#Train and validate for each epoch
losses = np.zeros(num_epochs)
validation_accs = np.zeros(num_epochs)

#CountData
countData()

for i in range(num_epochs):
    train(1)
    valAcc, lossTmp = validate()
    validation_accs[i] = valAcc
    losses[i] = lossTmp
    print(f"Epoch {i+1}/{num_epochs}")

plt.plot(np.array(validation_accs))
plt.plot(np.array(losses)*100) #Scaled by 100
plt.ylabel('Validation accuracy')
plt.xlabel('Epoch')
plt.show()

save_model(NN, "models\\model2.pth")
```

Listing A.8: Multiclass static vision model.

```
#-----------------------------------------
# Main Static Vision motion identification network
# Version 3.0
# Full range of gestures included ***
#-----------------------------------------
# Authors: Tim Johansson and Krister Mattson
#-----------------------------------------

import os
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import Vision_Network
import glob

torch.manual_seed(100)
USER = os.getenv('USER')

#Generate network
NN = Vision_Network.VisionNet(3) #3 classes: kick, hand, no gesture
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(NN.parameters(), lr=0.005, momentum=0.9)
val_accuracy = 0.0

def countNHK(labels, predicted, i):
    mask1 = predicted == i
    mask2 = labels == i
    both = mask1 & mask2
```

```
    return torch.sum(both).item()

def countData():
    kickCount = 0
    nonKickCount = 0
    handCount = 0
    for files in glob.iglob('*.jpg',root_dir='{USER}\\KickData\\
       kick\\'):
        kickCount = kickCount + 1
    for files in glob.iglob('*.jpg',root_dir='{USER}\\KickData\\
       noKick\\'):
        nonKickCount = nonKickCount + 1
    for files in glob.iglob('*.jpg',root_dir='{USER}\\KickData\\
       hand\\'):
        handCount = handCount + 1

    print("########################################")
    print("Number of kicks:         ", kickCount )
    print("Number of hand gestures: ", handCount)
    print("Number of nongestures:   ", nonKickCount )
    print("########################################")

#Function to save the trained model
def save_model(model, filepath):
    torch.save(model.state_dict(), filepath)
    print(f"Model saved to {filepath}")

#Training
def train(num_epochs):
    NN.train()
    running_loss = 0.0
    for epoch in range(num_epochs):

        for images, labels in train_loader:
            optimizer.zero_grad()

            outputs = NN(images)

            loss = criterion(outputs, labels)
            loss.backward()

            optimizer.step()
            running_loss += loss.item()

        # Print training loss for this epoch
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss /
            len(train_loader)}")

def validate():
    NN.eval()
    val_loss = 0.0
    correct_predictions = 0
    kick_count = 0
    hand_count = 0
    no_count = 0
    correct_kick = 0
```

```
    correct_hand = 0
    correct_no = 0

    with torch.no_grad():
        for images, labels in val_loader:
            outputs = NN(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct_predictions += (predicted == labels).sum().item
                ()

            #Save data
            correct_kick += countNHK(labels, predicted, 0)
            correct_hand += countNHK(labels, predicted, 1)
            correct_no += countNHK(labels, predicted, 2)

            kick_count += torch.sum(labels == 0).item()
            hand_count += torch.sum(labels == 1).item()
            no_count += torch.sum(labels == 2).item()

    val_accuracy = 100.0 * correct_predictions / len(val_dataset)
    kickAcc = 100.0 * correct_kick / kick_count
    handAcc = 100.0 * correct_hand / hand_count
    noAcc = 100.0 * correct_no / no_count

    print(f"Validation Loss: {val_loss / len(val_loader)},
        Validation Accuracy: {val_accuracy}%")
    return val_accuracy, val_loss/len(val_loader), kickAcc, handAcc
        , noAcc


#Define data transformations
transform = transforms.Compose([
    transforms.Resize((100, 80)),   #Resize images to desired size
    transforms.Grayscale(num_output_channels=1),  # Convert images
        to grayscale
    transforms.ToTensor(),           #Convert images to PyTorch
        tensors
    transforms.Normalize((0.5,),(0.5,))
])

#Data set
train_dataset = datasets.ImageFolder(root='KickData\\', transform=
    transform)

#Define the size of the validation set
validation_split = 0.2
dataset_size = len(train_dataset)
val_size = int(validation_split * dataset_size)
train_size = dataset_size - val_size

#Split the dataset into training and validation sets
train_dataset, val_dataset = torch.utils.data.random_split(
    train_dataset, [train_size, val_size])
```

```
#Create DataLoader instances for training and validation datasets
batch_size = 32 #5
train_loader = DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

#NUMBER OF EPOCHS---------------------------
num_epochs = 20

#Train and validate for each epoch
losses = np.zeros(num_epochs)
validation_accs = np.zeros(num_epochs)
kickAccs = np.zeros(num_epochs)
handAccs = np.zeros(num_epochs)
noAccs = np.zeros(num_epochs)

#CountData
countData()

#Main training and validation loop
for i in range(num_epochs):
    train(1)
    valAcc, lossTmp, kickAcc, handAcc, noAcc = validate()
    validation_accs[i] = valAcc
    losses[i] = lossTmp
    kickAccs[i] = kickAcc
    handAccs[i] = handAcc
    noAccs[i] = noAcc
    print(f"Epoch {i+1}/{num_epochs}")

plt.plot(np.array(validation_accs))
plt.plot(np.array(losses)*10) #Scaled by 100
plt.show()
plt.plot(np.array(kickAccs))
plt.show()
plt.plot(np.array(handAccs))
plt.show()
plt.plot(np.array(noAccs))
plt.ylabel('Validation accuracy')
plt.xlabel('Epoch')
plt.show()

print(validation_accs)
print(losses)
print(kickAccs)
print(handAccs)
print(noAccs)

save_model(NN, "models\\model1Full.pth")
```

A.3 Preprocessing dynamic vision model

Listing A.9: `merge_videos.py`, script for merging mp4-files by concatenation and reduces the frame rate.

```
#-----------------------------------------------------
#Merges mp4-files by concatenation and
#reduces the frame rate.
#Version 1.0
#-----------------------------------------------------
#Authors: Tim Johansson and Krister Mattsson
#-----------------------------------------------------

import glob
import os
import shutil
from moviepy.editor import VideoFileClip, concatenate_videoclips

def find_videos(directory):
    # Use glob to find all MP4 files in the specified directory and
        subdirectories
    video_files = glob.glob(os.path.join(directory, '**', '*.mp4'),
        recursive=True)
    return video_files

def merge_videos(video_files, output_path, fps):
    # Load all video files as VideoFileClip objects
    clips = [VideoFileClip(video) for video in video_files]
    # Concatenate all video clips into one
    final_clip = concatenate_videoclips(clips)
    # Write the result to a file with the specified framerate
    final_clip.write_videofile(output_path, codec="libx264",
       audio_codec="aac", fps=fps)

def main():
    USER = os.getenv('USER')
    input_base = f"/home/{USER}/master_thesis/data/output/"
    #logg_name= "Logg_HPG_2024-03-25"
    logg_name= "Logg_HPG_2024-02-19"
    input_dir_mp4_merge = f"{input_base}{logg_name}/*/srcb"
    output_base = f"/home/{USER}/master_thesis/data/output/{
       logg_name}/"

    frame_rate_adjustment = 2
    #Merge videos
    for dir_path in glob.glob(input_dir_mp4_merge):
        video_files = find_videos(dir_path)

        # Merge videos if any found
        if video_files:
            copy_folder_name = os.path.basename(os.path.dirname(
               dir_path))
            output_dir_merge = os.path.join(output_base, f"{
               copy_folder_name}/srcb/merged")
            if os.path.exists(output_dir_merge):
                shutil.rmtree(output_dir_merge)
            # Create the output directory as it doesn't exist
```

```
            os.makedirs(output_dir_merge, exist_ok=True)

            merge_videos(video_files, os.path.join(output_dir_merge
                , "video.mp4"),frame_rate_adjustment)


if __name__ == "__main__":
    main()
```

Listing A.10: `video_Labeling.py`, script for labeling merged mp4-files according to logbook entries shown in the figure 3.4.

```
#-----------------------------------------------------
#Datalabeing of merged videos according to Logbook entries.
#Version 1.0
#-----------------------------------------------------------------------------

#Authors: Tim Johansson and Krister Mattsson
#-----------------------------------------------------



import pandas as pd
import glob
import os


def find_videos(directory):
    # Use glob to find all MP4 files in the specified directory and
        subdirectories
    video_files = glob.glob(os.path.join(directory, '**', '*.mp4'),
        recursive=True)
    return video_files


def video_labeling_Binary_Class_kick(directory, logbook_df,
   output_base):
    # List to store the final data

    csv_path = f'{output_base}/videolabelBi_kick.csv'
    label_mapping_path = f'{output_base}/
       gesture_to_label_Binary_kick.csv'

    # Remove the CSV files if they already exist
    if os.path.exists(csv_path):
        os.remove(csv_path)
    if os.path.exists(label_mapping_path):
        os.remove(label_mapping_path)

    # Create a binary gesture to label mapping
    gesture_to_label = {'kick': 1, 'no_kick': 0}

    video_data = []
    for index, row in logbook_df.iterrows():
        copy_folder_name = str(row['Filename']).zfill(6)
        gesture = row['Gesture']
```

```
        # Find video files in the specific directory
        video_files = find_videos(os.path.join(directory,
            copy_folder_name, 'srcb/merged'))
        for video_file in video_files:
            if os.path.exists(video_file):
                # Determine the label based on the gesture
                label = 1 if 'kick' in gesture.lower() else 0
                # Append the data to the list
                video_data.append([index, video_file, label])

    # Create a DataFrame from the list
    video_df = pd.DataFrame(video_data, columns=['index', 'path', '
        label'])
    gesture_df = pd.DataFrame(list(gesture_to_label.items()),
        columns=['Gesture', 'Label'])

    # Save the DataFrame to a CSV file
    gesture_df.to_csv(label_mapping_path, index=False)
    video_df.to_csv(csv_path, index=False)


def video_labeling_Binary_Class_hand(directory, logbook_df,
    output_base):
    # List to store the final data

    csv_path = f'{output_base}/videolabelBi_hand.csv'
    label_mapping_path = f'{output_base}/
        gesture_to_label_Binary_hand.csv'

    # Remove the CSV files if they already exist
    if os.path.exists(csv_path):
        os.remove(csv_path)
    if os.path.exists(label_mapping_path):
        os.remove(label_mapping_path)

    # Create a binary gesture to label mapping
    gesture_to_label = {'hand': 1, 'no_hand': 0}

    video_data = []
    for index, row in logbook_df.iterrows():
        copy_folder_name = str(row['Filename']).zfill(6)
        gesture = row['Gesture']

        # Find video files in the specific directory
        video_files = find_videos(os.path.join(directory,
            copy_folder_name, 'srcb/merged'))
        for video_file in video_files:
            if os.path.exists(video_file):
                # Determine the label based on the gesture
                label = 1 if 'hand' in gesture.lower() else 0
                # Append the data to the list
                video_data.append([index, video_file, label])

    # Create a DataFrame from the list
    video_df = pd.DataFrame(video_data, columns=['index', 'path', '
```

```
        label'])
    gesture_df = pd.DataFrame(list(gesture_to_label.items()),
        columns=['Gesture', 'Label'])

    # Save the DataFrame to a CSV file
    gesture_df.to_csv(label_mapping_path, index=False)
    video_df.to_csv(csv_path, index=False)


def video_labeling_Mult_Class(directory, logbook_df,output_base):
    # List to store the final data

    csv_path = f'{output_base}/videolabelMult3.csv'
    label_mapping_path = f'{output_base}/gesture_to_label_Mult3.csv
        '

    # Remove the CSV files if they already exist
    if os.path.exists(csv_path):
        os.remove(csv_path)
    if os.path.exists(label_mapping_path):
        os.remove(label_mapping_path)

    # Create a binary gesture to label mapping
    gesture_to_label = {'kick': 1, 'No_gesture': 0, 'hand': 2}

    video_data = []
    for index, row in logbook_df.iterrows():
        copy_folder_name = str(row['Filename']).zfill(6)
        gesture = row['Gesture']

        # Find video files in the specific directory
        video_files = find_videos(os.path.join(directory,
            copy_folder_name, 'srcb/merged'))
        for video_file in video_files:
            if os.path.exists(video_file):
                # Determine the label based on the gesture
                label = gesture_to_label.get(gesture.lower(), 0)
                #label = 1 if 'kick' in gesture.lower() else 0
                # Append the data to the list
                video_data.append([index, video_file, label])

    # Create a DataFrame from the list
    video_df = pd.DataFrame(video_data, columns=['index', 'path', '
        label'])
    gesture_df = pd.DataFrame(list(gesture_to_label.items()),
        columns=['Gesture', 'Label'])


    # Save the DataFrame to a CSV file
    gesture_df.to_csv(label_mapping_path, index=False)
    video_df.to_csv(csv_path, index=False)

def video_labeling_Mult_Class2(directory, logbook_df,output_base):

    # Save the DataFrame to a CSV file
    csv_path = f'{output_base}/videolabelMult3_2.csv'
```

```
    label_mapping_path = f'{output_base}/gesture_to_label_Mult3_2.
        csv'

    # Extract unique gestures and create a mapping to unique labels
    unique_gestures = logbook_df['Gesture'].str.lower().unique()
    gesture_to_label = {gesture: idx for idx, gesture in enumerate(
        unique_gestures)}

    # List to store the final data
    video_data = []
    for index, row in logbook_df.iterrows():
        copy_folder_name = str(row['Filename']).zfill(6)
        gesture = row['Gesture']
        if gesture != 'standstill':
            # Find video files in the specific directory
            video_files = find_videos(os.path.join(directory,
                copy_folder_name, 'srcb/merged'))
            for video_file in video_files:
                if os.path.exists(video_file):
                    # Determine the label based on the gesture
                    label = gesture_to_label.get(gesture.lower())
                    # Append the data to the list
                    video_data.append([index, video_file, label])

    # Create a DataFrame from the list
    video_df = pd.DataFrame(video_data, columns=['index', 'path', '
        label'])
    gesture_df = pd.DataFrame(list(gesture_to_label.items()),
        columns=['Gesture', 'Label'])



    # Remove the CSV files if they already exist
    if os.path.exists(csv_path):
        os.remove(csv_path)
    if os.path.exists(label_mapping_path):
        os.remove(label_mapping_path)


    # Save to a CSV file

    gesture_df.to_csv(label_mapping_path, index=False)
    video_df.to_csv(csv_path, index=False)

def video_labeling_Mult_Class_ext(directory, logbook_df,output_base
    ):
    # Save the DataFrame to a CSV file
    csv_path = f'{output_base}/videolabelMult.csv'
    label_mapping_path = f'{output_base}/gesture_to_label_Mult.csv'

    logbook_df['Combined_Label'] = logbook_df['Gesture'].str.lower
        () + ' ' + logbook_df['Attribute'].str.lower()
    unique_combined_labels = logbook_df['Combined_Label'].unique()
    combined_label_to_index = {label: idx for idx, label in
        enumerate(unique_combined_labels)}
```

```
    # List to store the final data
    video_data = []
    for index, row in logbook_df.iterrows():
        copy_folder_name = str(row['Filename']).zfill(6)
        combined_label = row['Combined_Label']
        # Find video files in the specific directory
        video_files = find_videos(os.path.join(directory,
           copy_folder_name, 'srcb/merged'))
        for video_file in video_files:
            if os.path.exists(video_file):
                # Determine the label based on the combined label
                label = combined_label_to_index.get(combined_label)
                # Append the data to the list
                video_data.append([index, video_file, label])

    # Create a DataFrame from the list
    video_df = pd.DataFrame(video_data, columns=['index', 'path', '
       label'])
    combined_label_df = pd.DataFrame(list(combined_label_to_index.
       items()), columns=['Gesture', 'Label'])

    # Remove the CSV files if they already exist
    if os.path.exists(csv_path):
        os.remove(csv_path)
    if os.path.exists(label_mapping_path):
        os.remove(label_mapping_path)

    # Save to a CSV file
    combined_label_df.to_csv(label_mapping_path, index=False)
    video_df.to_csv(csv_path, index=False)


def main():
    USER = os.getenv('USER')
        # Path to the logbook file
    logbook_path = f"/home/{USER}/master_thesis/data/Logbook.xlsx"
    # Directory to search for video files
    directory = f"/home/{USER}/master_thesis/data/output/*"
    output_base = f"/home/{USER}/master_thesis/data/output/"

    # Read the logbook file
    logbook_df = pd.read_excel(logbook_path)
    video_labeling_Binary_Class_hand(directory, logbook_df,
       output_base)
    video_labeling_Binary_Class_kick(directory, logbook_df,
       output_base)
    video_labeling_Mult_Class(directory, logbook_df,output_base)
    video_labeling_Mult_Class_ext(directory, logbook_df,output_base
       )


if __name__ == "__main__":
    main()
```

A.4 Dynamic Vision Model

Listing A.11: `dynamicVisionNetwork.py`, script for building, training and validating the dynamic vision model.

```
#---------------------------------------------
#Dynamic Vision Model v1.0
#---------------------------------------------
#Authors: Tim Johansson and Krister Mattsson
#---------------------------------------------

import os
import time
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
import matplotlib

from glob import glob
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, accuracy_score,
   classification_report, confusion_matrix

import seaborn as sns

import torch
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from torchvision.transforms import ToPILImage

from tqdm import tqdm

from torch import nn
from torch.nn.functional import relu

import torchvision.transforms as transforms
from torchvision.transforms import functional as F
from torchvision.transforms import InterpolationMode

from torchvision.io.video import read_video
from torchvision.models.video import r2plus1d_18,
   R2Plus1D_18_Weights

import warnings
warnings.filterwarnings(action='ignore')
matplotlib.use('Agg')  # Use a non-interactive backend

def seed_everything(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
```

```
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = True

class VideoDataset(Dataset):
    def __init__(self, video_paths, labels, transform=None,
       num_frames=70):
        self.video_paths = video_paths
        self.labels = labels
        self.transform = transform
        self.num_frames = num_frames
        self.to_pil = ToPILImage()
        self.device = torch.device("cuda:0" if torch.cuda.
           is_available() else "cpu")


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

    def __getitem__(self, idx):
        video_path = self.video_paths[idx]
        video_frames, _, _ = read_video(video_path, pts_unit='sec')

        video_frames = self.pad_or_truncate(video_frames)

        if self.transform:
            video_frames = self.apply_transform(video_frames)

        label = self.labels[idx]
        return video_frames.to(self.device), torch.tensor(label).to
           (self.device), video_path

    def pad_or_truncate(self, video_frames):
        num_frames = video_frames.shape[0]
        if num_frames < self.num_frames:
            # Pad with the last frame
            pad_size = self.num_frames - num_frames
            padding = video_frames[-1].unsqueeze(0).repeat(pad_size
               , 1, 1, 1)
            video_frames = torch.cat((video_frames, padding), dim
               =0)
        elif num_frames > self.num_frames:
            # Truncate the frames
            video_frames = video_frames[:self.num_frames]
        return video_frames

    def apply_transform(self, video_frames):
        transformed_frames = [self.transform(self.to_pil(frame.
           permute(2, 0, 1) / 255.0)).to(self.device) for frame in
           video_frames.to(self.device)] # Convert to (C, H, W)
           format Normalize to [0, 1] and apply transform
        return torch.stack(transformed_frames).permute(1, 0, 2, 3).
           to(self.device)   # Convert to (C, T, H, W)

class DynamicVisionNN(torch.nn.Module):
    def __init__(self, num_classes):
        super(DynamicVisionNN, self).__init__()
```

```
        # Load weights
        weights = R2Plus1D_18_Weights.DEFAULT
        # Load and initialize the pretrained model
        pretrained_model = r2plus1d_18(weights=weights)

        self.pretrained = pretrained_model
        # Update the fully connected layer for binary
           classification
        self.pretrained.fc = nn.Linear(in_features=pretrained_model
           .fc.in_features, out_features=256)
        self.batchnorm = nn.BatchNorm1d(256)
        self.final_fc = nn.Linear(in_features=256, out_features=
           num_classes)

        # Freeze layers except the fully connected layers
        for param in self.pretrained.parameters():
            param.requires_grad = False
        for param in self.pretrained.fc.parameters():
            param.requires_grad = True
        for param in self.final_fc.parameters():
            param.requires_grad = True

    def forward(self, x):
        x = self.pretrained(x)
        x = torch.relu(x)
        x = self.batchnorm(x)
        x = self.final_fc(x)
        return x

def get_video_lengths(video_paths):
    lengths = []
    for video_path in video_paths:
        video_frames, _, _ = read_video(video_path, pts_unit='sec')
        lengths.append(len(video_frames))
    return lengths

def visualize_data_distribution(video_paths, labels, label_dict,
   output_csv_path, dataset_name):
    # Reverse the label dictionary to map numeric labels back to
       class names
    reverse_label_dict = {v: k for k, v in label_dict.items()}
    label_names = [reverse_label_dict[label] for label in labels]

    # Count the number of videos per label
    label_counts = pd.Series(label_names).value_counts()

    # Plotting the bar chart for the number of videos per label
    fig_1, ax1 = plt.subplots(figsize=(8, 6))
    bars = label_counts.plot(kind='bar', color='skyblue', ax=ax1)
    ax1.set_xlabel('Motion')
    ax1.set_ylabel('Number of Videos')
    ax1.set_title(f'Number of Videos per Label ({dataset_name} Set)
       ')
    ax1.set_xticklabels(ax1.get_xticklabels(), rotation=0)
    ax1.bar_label(bars.containers[0], label_type='edge', padding=3)
```

```
    ax1.tick_params(axis='x', rotation=90)
    fig_1.tight_layout()
    fig_1.savefig(f'{output_csv_path}/NumofVideosPerClass_{
       dataset_name}.png')
    plt.close(fig_1)

    # Get video lengths
    video_lengths = get_video_lengths(video_paths)

    # Separate video lengths by class
    class_lengths = {label: [] for label in label_dict.keys()}
    for length, label in zip(video_lengths, labels):
        class_lengths[reverse_label_dict[label]].append(length)

    # Plotting the distribution of video lengths per class
    fig_2, ax2 = plt.subplots(figsize=(8, 6))
    colors = plt.cm.get_cmap('tab10', len(class_lengths)).colors  #
        Get a color map with enough colors
    for i, (class_name, lengths) in enumerate(class_lengths.items()
       ):
        ax2.hist(lengths, bins=30, alpha=0.7, label=class_name,
           color=colors[i])
    ax2.set_xlabel('Number of Frames')
    ax2.set_ylabel('Frequency')
    ax2.set_title(f'Video Length Distribution per Class ({
       dataset_name} Set)')
    ax2.legend()
    fig_2.tight_layout()
    fig_2.savefig(f'{output_csv_path}/NumofFrames_{dataset_name}.
       png')
    plt.close(fig_2)

def train(model, optimizer, train_loader, val_loader, device,
   epochs, criterion, output_csv_path,label_dict,scheduler=None):
    print(f'---Training commenced---')
    best_val_score = 0
    best_model = None

    train_losses = []
    val_losses = []
    f1_scores = []
    val_accuracies = []
    label_accuracies = []

    for epoch in range(1, epochs+1):
        model.train()
        train_loss = []
        for videos, labels, video_paths in tqdm(iter(train_loader))
           :
            videos = videos.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            output = model(videos)
            loss = criterion(output, labels)
```

```
            loss.backward()
            optimizer.step()

            train_loss.append(loss.item())

        _val_loss, _val_score, _val_accuracy, label_accuracy_dict
           = validation(model, criterion, val_loader, device,
           label_dict)
        _train_loss = np.mean(train_loss)
        print(f'Epoch [{epoch}], Train Loss : [{_train_loss:.5f}]
           Val Loss : [{_val_loss:.5f}] Val F1 : [{_val_score:.5f}]
            Val Accuracy: [{_val_accuracy:.5f}]')

        train_losses.append(_train_loss)
        val_losses.append(_val_loss)
        f1_scores.append(_val_score)
        val_accuracies.append(_val_accuracy)

        label_accuracies.append(label_accuracy_dict)

        if scheduler is not None:
            scheduler.step()

        if best_val_score < _val_score:
            best_val_score = _val_score
            best_model = model

    # Create a DataFrame to save the results
    results_df = pd.DataFrame({
        'Epoch': range(1, epochs + 1),
        'Train Loss': train_losses,
        'Validation Loss': val_losses,
        'F1 Score': f1_scores,
        'Validation Accuracy': val_accuracies
    })

    label_accuracies_df = pd.DataFrame(label_accuracies)

    # Save the DataFrame to a CSV file
    results_df.to_csv(f'{output_csv_path}/training_results.csv',
       index=False)
    label_accuracies_df.to_csv(f'{output_csv_path}/label_accuracies
       .csv', index=False)

    # Plot training loss, validation loss, F1 score, and validation
        accuracy
    evalPlot(epochs,train_losses,val_losses, f1_scores,
       val_accuracies, output_csv_path)

    return best_model

def evalPlot(epochs,train_losses,val_losses, f1_scores,
   val_accuracies, output_csv_path):
    epochs_range = range(1, epochs + 1)
    plt.figure(figsize=(12, 6))
```

```
    plt.plot(epochs_range, train_losses, label='Train Loss')
    plt.plot(epochs_range, val_losses, label='Validation Loss')
    plt.plot(epochs_range, f1_scores, label='F1 Score')
    plt.plot(epochs_range, val_accuracies, label='Validation
       Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Loss/F1 Score/Accuracy')
    plt.title('Training and Validation Metrics')
    plt.legend()
    plt.savefig(f'{output_csv_path}/Training_Validation_Metrics.png
       ')
    plt.show()

def validation(model,criterion,val_loader,device,label_dict):
    model.eval()
    val_loss = []
    preds, trues = [], []

    with torch.no_grad():
        for videos, labels, video_paths in tqdm(iter(val_loader)):
            videos = videos.to(device)
            labels = labels.to(device)

            output = model(videos)

            loss = criterion(output, labels)
            val_loss.append(loss.item())
            _, classification = torch.max(output, 1)

            trues.extend(labels.tolist())
            preds.extend(classification.tolist())


        _val_loss = np.mean(val_loss)
    _val_score = f1_score(trues, preds, average='macro')
    _val_accuracy = accuracy_score(trues, preds)

    # Calculate accuracy per label
    label_accuracy_dict = {}
    trues = np.array(trues)
    preds = np.array(preds)

    for label_name, label_index in label_dict.items():
        label_indices = trues == label_index
        label_trues = trues[label_indices]
        label_preds = preds[label_indices]
        label_accuracy = accuracy_score(label_trues, label_preds)
        label_accuracy_dict[f'{label_name}_Accuracy'] =
           label_accuracy

    return _val_loss, _val_score, _val_accuracy,
       label_accuracy_dict

def test(test_loader, model, device, criterion, output_csv_path):
    print(f'---Test commenced---')
    model.eval()
```

```
    test_loss = []
    preds, trues, paths = [], [], []
    tp, tn, fp, fn = [], [], [], []

    with torch.no_grad():
        for videos, labels, video_paths  in tqdm(iter(test_loader))
            :
            videos = videos.to(device)
            labels = labels.to(device)

            output = model(videos)

            _, classification = torch.max(output, 1)
            alpha = torch.abs(output[:, 0] - output[:, 1])
            loss = criterion(output, labels)

            test_loss.append(loss.item())
            paths.extend(video_paths)
            trues.extend(labels.tolist())
            preds.extend(classification.tolist())

            for i in range(len(labels)):
                true_label = labels[i].item()
                pred_label = classification[i].item()
                certainty = alpha[i].item()

                if true_label == 1 and pred_label == 1:
                    tp.append(certainty)
                elif true_label == 0 and pred_label == 0:
                    tn.append(certainty)
                elif true_label == 0 and pred_label == 1:
                    fp.append(certainty)
                elif true_label == 1 and pred_label == 0:
                    fn.append(certainty)

        _test_loss = np.mean(test_loss)
    _test_score = f1_score(trues, preds, average='macro')
    _test_accuracy = accuracy_score(trues, preds)
    print(f'Test Loss: {_test_loss:.5f}, Test F1 Score: {
        _test_score:.5f}, Test Accuracy: {_test_accuracy:.5f}')

    # Create a DataFrame to save the results
    results_df = pd.DataFrame({
        'Test Loss': [_test_loss],
        'F1 Score': [_test_score],
        'Test Accuracy': [_test_accuracy]
    })
    results_df.to_csv(f'{output_csv_path}/test_results.csv', index=
        False)

    # Identify False Positives and False Negatives
    false_pred = [(path, true, pred) for path, true, pred in zip(
        paths, trues, preds) if true != pred]

    # Save false positives and false negatives to CSV
```

```
    f_df = pd.DataFrame(false_pred, columns=['Path', 'True Label',
        'Predicted Label'])
    f_df.to_csv(f'{output_csv_path}/false_pred.csv', index=False)

    evaluate_certainties(tp,tn,fp,fn,output_csv_path)
    return trues, preds

def plot_confusion_matrix(trues, preds, class_names, label_dict,
    output_csv_path):
    # Map numeric labels to class names
    reverse_label_dict = {v: k for k, v in label_dict.items()}
    trues_named = [reverse_label_dict[label] for label in trues]
    preds_named = [reverse_label_dict[label] for label in preds]

    cm = confusion_matrix(trues_named, preds_named, labels=
        class_names)
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.
        newaxis]

    plt.figure(figsize=(10, 8))
    #sns.heatmap(cm_normalized*100, annot=True, fmt='.2f', cmap='
        Blues', xticklabels=class_names, yticklabels=class_names,
        vmin=0, vmax=100, annot_kws={"size": 26})
    sns.heatmap(cm_normalized*100, annot=True, fmt='.2f', cmap='
        Blues', xticklabels=class_names, yticklabels=class_names,
        vmin=0, vmax=100)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.tight_layout()
    plt.savefig(f'{output_csv_path}/Confusion_Matrix.png')
    plt.show()

def plot_classification_report(trues, preds, class_names,
    output_csv_path):
    report = classification_report(trues, preds, target_names=
        class_names, output_dict=True)
    df = pd.DataFrame(report).transpose()
    df.iloc[:-1, :-1] = df.iloc[:-1, :-1] * 100
    plt.figure(figsize=(10, 8))
    #sns.heatmap(df.iloc[:-1, :-1].T, annot=True, fmt=".1f", cmap='
        Blues', annot_kws={"size": 20})
    sns.heatmap(df.iloc[:-1, :-1].T, annot=True, fmt=".1f", cmap='
        Blues')
    plt.title('Classification Report')
    plt.tight_layout()
    plt.savefig(f'{output_csv_path}/Classification_Report.png')
    plt.show()

def plot_bar_chart(metrics, class_names, metric_name,
    output_csv_path):
    plt.figure(figsize=(8, 6))
    ax = sns.barplot(x=class_names, y=metrics, palette='Blues_d')
    plt.xlabel('Class')
    plt.ylabel(metric_name)
    plt.title(f'{metric_name} by Class')
```

```
    for container in ax.containers:
        ax.bar_label(container, label_type='edge', padding=3)
    plt.xticks(rotation=90)
    plt.tight_layout()
    plt.savefig(f'{output_csv_path}/{metric_name}_by_Class.png')
    plt.show()

def evaluate_certainties(tp,tn,fp,fn, output_csv_path):

    tp = torch.tensor(tp)
    tn = torch.tensor(tn)
    fp = torch.tensor(fp)
    fn = torch.tensor(fn)

    # Calculate mean certainties
    mean_certainty_tp = tp.mean().item() if len(tp) > 0 else 0
    mean_certainty_tn = tn.mean().item() if len(tn) > 0 else 0
    mean_certainty_fp = fp.mean().item() if len(fp) > 0 else 0
    mean_certainty_fn = fn.mean().item() if len(fn) > 0 else 0

    total = len(tp) + len(tn) + len(fp) + len(fn)

    # Calculate percentages
    percentage_tp = (len(tp) / total) * 100 if total > 0 else 0
    percentage_tn = (len(tn) / total) * 100 if total > 0 else 0
    percentage_fp = (len(fp) / total) * 100 if total > 0 else 0
    percentage_fn = (len(fn) / total) * 100 if total > 0 else 0

    print(f"Mean Certainty TP: {mean_certainty_tp}, Count: {len(tp)
       }, Percentage: {percentage_tp:.2f}%")
    print(f"Mean Certainty TN: {mean_certainty_tn}, Count: {len(tn)
       }, Percentage: {percentage_tn:.2f}%")
    print(f"Mean Certainty FP: {mean_certainty_fp}, Count: {len(fp)
       }, Percentage: {percentage_fp:.2f}%")
    print(f"Mean Certainty FN: {mean_certainty_fn}, Count: {len(fn)
       }, Percentage: {percentage_fn:.2f}%")

    data = {
        'Category': ['TP', 'TN', 'FP', 'FN'],
        'Mean Certainty': [mean_certainty_tp, mean_certainty_tn,
           mean_certainty_fp, mean_certainty_fn],
        'Count': [len(tp), len(tn), len(fp), len(fn)],
        'Percentage': [percentage_tp, percentage_tn, percentage_fp,
            percentage_fn]
    }
    df = pd.DataFrame(data)
    df.to_csv(f'{output_csv_path}/evaluation_certainties.csv',
       index=False)

def main():
    # Use GPU if available else revert to CPU
    USER = os.getenv('USER')
    device = torch.device("cuda" if torch.cuda.is_available() else
       "cpu")
    print("Device being used:", device)
```

```
# Configuration parameters
CFG = {
    'fps': 2,
    'seed': 42,
    'batch_size': 4,
    'num_epochs':20,
    'learning_rate': 3e-4,
    'num_classes': 9, # Binary: 2, MultiClass: 3, ExtMultiClass
       : 9
    'Resolution': 1 # 100% by default
}

tqdm.pandas()
# Sets seeds for various random number generators
seed_everything(CFG['seed'])

# Define directories
#Classification 'no_kick':0, 'kick':1. Dont forget to change '
   num_classes'
video_label_df = pd.read_csv(f"/home/{USER}/master_thesis/data/
   output/videolabelBi_kick.csv")
gesture_to_label_df = pd.read_csv(f"/home/{USER}/master_thesis/
   data/output/gesture_to_label_Binary_kick.csv")
output_csv_path = f"/home/{USER}/master_thesis/motion/Bi/
   twoLayer_4_kick"
"""
#Classification 'no_kick':0, 'kick':1. Dont forget to change '
   num_classes'
video_label_df = pd.read_csv(f"/home/{USER}/master_thesis/data/
   output/videolabelBi_hand.csv")
gesture_to_label_df = pd.read_csv(f"/home/{USER}/master_thesis/
   data/output/gesture_to_label_Binary_hand.csv")
output_csv_path = f"/home/{USER}/master_thesis/motion/Bi/
   twoLayer_4_hand"

#Classification of multiple classes, 3. Dont forget to change '
   num_classes'
video_label_df = pd.read_csv(f"/home/{USER}/master_thesis/data/
   output/videolabelMult3.csv")
gesture_to_label_df = pd.read_csv(f"/home/{USER}/master_thesis/
   data/output/gesture_to_label_Mult3.csv")
#output_csv_path = f"/home/{USER}/master_thesis/motion/Mult/
   twoLayer/class3_b4"

#Classification of multiple classes, 9. Dont forget to change '
   num_classes'
video_label_df = pd.read_csv(f"/home/{USER}/master_thesis/data/
   output/videolabelMult9.csv")
gesture_to_label_df = pd.read_csv(f"/home/{USER}/master_thesis/
   data/output/gesture_to_label_Mult9.csv")
output_csv_path = f"/home/{USER}/master_thesis/motion/Mult/
   twoLayer/class9_b4"

"""

model_save_path = f'{output_csv_path}/best_model_Bi.pth'
```

```
# Convert labels to integer
video_label_df['label'] = video_label_df['label'].astype(int)
# Create the label mapping dictionary
label_dict = dict(zip(gesture_to_label_df['Gesture'],
    gesture_to_label_df['Label']))
class_names = list(label_dict.keys())

# Extract video paths and labels
video_paths = video_label_df['path'].tolist()
video_labels = video_label_df['label'].tolist()

# First split into train and test, stratifying by labels
train_paths, test_paths, train_labels, test_labels =
    train_test_split(video_paths, video_labels, test_size=0.2,
    random_state=CFG['seed'], stratify=video_labels)
train_paths, val_paths, train_labels, val_labels =
    train_test_split(train_paths, train_labels, test_size=0.2,
    random_state=CFG['seed'], stratify=train_labels)

# Visualization
visualize_data_distribution(train_paths, train_labels,
    label_dict, output_csv_path, "Train")
visualize_data_distribution(val_paths, val_labels, label_dict,
    output_csv_path, "Validation")
visualize_data_distribution(test_paths, test_labels, label_dict
    , output_csv_path, "Test")
visualize_data_distribution(video_paths, video_labels,
    label_dict, output_csv_path, "Overall")

#Transformation parameters
transform = transforms.Compose([
            transforms.Resize([128 * CFG['Resolution'], 171*
                CFG['Resolution']], interpolation=
                InterpolationMode.BILINEAR),
            transforms.CenterCrop([112* CFG['Resolution'], 112*
                 CFG['Resolution']]),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.43216, 0.394666,
                0.37645], std=[0.22803, 0.22145, 0.216989])
        ])

# Create datasets
train_dataset = VideoDataset(train_paths, train_labels,
    transform=transform)
val_dataset = VideoDataset(val_paths, val_labels, transform=
    transform)
test_dataset = VideoDataset(test_paths, test_labels, transform=
    transform)
all_data_dataset = VideoDataset(video_paths, video_labels,
    transform=transform)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=CFG['
    batch_size'], shuffle=True)
```

```
    val_loader = DataLoader(val_dataset, batch_size=CFG['batch_size
        '], shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=CFG['
        batch_size'])
    all_data_loader = DataLoader(all_data_dataset, batch_size=CFG['
        batch_size'], shuffle=True)

    model = DynamicVisionNN(CFG['num_classes']).to(device)

    optimizer = Adam(model.parameters(), lr=CFG['learning_rate']) #
         hyperparameters as given in paper https://doi.org/10.48550/
        arXiv.1711.11248 sec 4.1
    criterion = torch.nn.CrossEntropyLoss().to(device)

    # saves the time the process was started, to compute total time
         at the end
    start = time.time()

    #trained_best_model  = train(model, optimizer, train_loader,
        val_loader, device, CFG['num_epochs'], criterion,
        output_csv_path,label_dict)

    # print the total time needed, HH:MM:SS format
    time_elapsed = time.time() - start
    print(f"Training completed in {time_elapsed // 3600}h {(
        time_elapsed % 3600) // 60}m {time_elapsed % 60}s")

    # Save the best model
    #torch.save(trained_best_model.state_dict(), model_save_path)

    # Load the state dictionary into a new model instance for
        testing
    best_model = DynamicVisionNN(CFG['num_classes']).to(device)
    best_model.load_state_dict(torch.load(model_save_path))

    #Test the model
    trues, preds =  test(test_loader, best_model, device, criterion
        , output_csv_path)
    #Test the model on all data
    #trues, preds = test(all_data_loader, best_model, device,
        criterion, output_csv_path)

    # Plot confusion matrix
    plot_confusion_matrix(trues, preds, class_names, label_dict,
        output_csv_path)

    # Plot classification report
    plot_classification_report(trues, preds, class_names,
        output_csv_path)


if __name__ == "__main__":
    main()
```

A.5 R(2+1)D Architecture

Listing A.12: R(2+1)D Pre-Trained Model Architecture [24], [13].

```
Total params: 31,301,151
Trainable params: 1,026
Non-trainable params: 31,300,125

R2Plus1DPreTrainedModel(
  (pretrained): VideoResNet(
    (stem): R2Plus1dStem(
      (0): Conv3d(3, 45, kernel_size=(1, 7, 7), stride=(1, 2, 2),
         padding=(0, 3, 3), bias=False)
      (1): BatchNorm3d(45, eps=1e-05, momentum=0.1, affine=True,
         track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv3d(45, 64, kernel_size=(3, 1, 1), stride=(1, 1, 1),
         padding=(1, 0, 0), bias=False)
      (4): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True,
         track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(64, 144, kernel_size=(1, 3, 3), stride=(1,
               1, 1), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(144, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): Conv3d(144, 64, kernel_size=(3, 1, 1), stride=(1,
               1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True
             , track_running_stats=True)
          (2): ReLU(inplace=True)
        )
        (conv2): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(64, 144, kernel_size=(1, 3, 3), stride=(1,
               1, 1), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(144, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): Conv3d(144, 64, kernel_size=(3, 1, 1), stride=(1,
               1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True
             , track_running_stats=True)
        )
        (relu): ReLU(inplace=True)
      )
      (1): BasicBlock(
```

```
      (conv1): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(64, 144, kernel_size=(1, 3, 3), stride=(1,
             1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(144, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(144, 64, kernel_size=(3, 1, 1), stride=(1,
             1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True
           , track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (conv2): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(64, 144, kernel_size=(1, 3, 3), stride=(1,
             1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(144, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(144, 64, kernel_size=(3, 1, 1), stride=(1,
             1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True
           , track_running_stats=True)
      )
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(64, 230, kernel_size=(1, 3, 3), stride=(1,
             2, 2), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(230, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(230, 128, kernel_size=(3, 1, 1), stride=(2,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (conv2): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(128, 230, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(230, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(230, 128, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
```

```
        (1): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv3d(64, 128, kernel_size=(1, 1, 1), stride=(2, 2,
            2), bias=False)
        (1): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(128, 288, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(288, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(288, 128, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (conv2): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(128, 288, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(288, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(288, 128, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(128, 460, kernel_size=(1, 3, 3), stride=(1,
              2, 2), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(460, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(460, 256, kernel_size=(3, 1, 1), stride=(2,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
```

```
        (2): ReLU(inplace=True)
      )
      (conv2): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(256, 460, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(460, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(460, 256, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv3d(128, 256, kernel_size=(1, 1, 1), stride=(2,
           2, 2), bias=False)
        (1): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(256, 576, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(576, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(576, 256, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (conv2): Sequential(
        (0): Conv2Plus1D(
          (0): Conv3d(256, 576, kernel_size=(1, 3, 3), stride=(1,
              1, 1), padding=(0, 1, 1), bias=False)
          (1): BatchNorm3d(576, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv3d(576, 256, kernel_size=(3, 1, 1), stride=(1,
              1, 1), padding=(1, 0, 0), bias=False)
        )
        (1): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=
           True, track_running_stats=True)
      )
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
```

```
        (conv1): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(256, 921, kernel_size=(1, 3, 3), stride=(1,
                2, 2), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(921, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): Conv3d(921, 512, kernel_size=(3, 1, 1), stride=(2,
                1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(512, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
        )
        (conv2): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(512, 921, kernel_size=(1, 3, 3), stride=(1,
                1, 1), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(921, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): Conv3d(921, 512, kernel_size=(3, 1, 1), stride=(1,
                1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(512, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
        )
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv3d(256, 512, kernel_size=(1, 1, 1), stride=(2,
             2, 2), bias=False)
          (1): BatchNorm3d(512, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(512, 1152, kernel_size=(1, 3, 3), stride
               =(1, 1, 1), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(1152, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): Conv3d(1152, 512, kernel_size=(3, 1, 1), stride
               =(1, 1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(512, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
          (2): ReLU(inplace=True)
        )
        (conv2): Sequential(
          (0): Conv2Plus1D(
            (0): Conv3d(512, 1152, kernel_size=(1, 3, 3), stride
               =(1, 1, 1), padding=(0, 1, 1), bias=False)
            (1): BatchNorm3d(1152, eps=1e-05, momentum=0.1, affine=
               True, track_running_stats=True)
```

```
            (2): ReLU(inplace=True)
            (3): Conv3d(1152, 512, kernel_size=(3, 1, 1), stride
               =(1, 1, 1), padding=(1, 0, 0), bias=False)
          )
          (1): BatchNorm3d(512, eps=1e-05, momentum=0.1, affine=
             True, track_running_stats=True)
        )
        (relu): ReLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool3d(output_size=(1, 1, 1))
    (fc): Linear(in_features=512, out_features=2, bias=True)
  )
)
```

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY