CHALMERS
UNIVERSITY OF TECHNOLOGY

Control and Camera-based State Estimation using Machine Vision and Machine Learning

A Comparison Study in IMU-replacing Neural Networks on a Wheeled Inverted Pendulum System

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

JONATHAN ALMGREN, LASSE KÖTZ

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Control and camera-based state estimation using machine vision and machine learning

A comparison study in IMU-replacing Neural Networks on a Wheeled Inverted Pendulum system

JONATHAN ALMGREN, LASSE KÖTZ

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2023

Control and camera-based state estimation using machine vision and machine learning
A comparison study in IMU-replacing Neural Networks on a Wheeled Inverted Pendulum system
JONATHAN ALMGREN, LASSE KÖTZ

© JONATHAN ALMGREN, LASSE KÖTZ, 2023.

Supervisor: Carl-Henrik Hult, Knowit Connectivity
Examiner: Jonas Sjöberg, Mechatronics

Master's Thesis 2023
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Control and camera-based state estimation using machine vision and machine learning
A comparison study in IMU-replacing Neural Networks on a Wheeled Inverted Pendulum system

JONATHAN ALMGREN, LASSE KÖTZ
Department of Electrical Engineering
Chalmers University of Technology

Abstract

State estimation is an important aspect in a large number of robotic applications. With recent advancements within the field of Machine Learning (ML), it has become increasingly interesting to study how Neural Networks (NNs) can be applied to overcome this problem.

This master thesis consists of implementing, training and comparing three different NN architectures that, given a video stream as input, estimate the current leaning angle of a Wheeled Inverted Pendulum (WIP) system. This is done to such precision and inference speed that it can be used to control an unstable real time system. To deploy and demonstrate the real time performance of these ML-models, a self-balancing robot was constructed. The demonstration robot consists of a custom designed platform with 3D-printed components mounted together by threaded rods and actuated by two brushed DC-motor equipped with encoders. The core processing is performed on a Raspberry Pi 3B boosted by a Tensor Processing Unit (TPU) that helps with processing the incoming camera data through the NNs.

Control of the system is performed using two cascade Proportional–Integral–Derivative controllers (PID), where one outer loop controls the horizontal speed of the robot while the inner loop controls the leaning angle of the robot. Design of the control system is performed using classic control methods and its' biggest challenge is handling the slower sampling rate of camera data compared to the alternative solution of using an IMU for angle estimation.

The models showed results with Mean Absolute Errors (MAE) reaching as far down as 0.8255° and a standard deviation of 0.4072 in ideal cases. Through signal processing, this could be reduced further under certain conditions. When running on the Raspberry Pi 3B hardware, the deployed NN reached a sampling rate of 60 Hz, which was too slow to get accurate performance in controlling the system. Simplified test runs were conducted on upgraded hardware which allowed it to reach adequate sampling rates for stability but could not be deployed on the physical robot due to project limitations.

Analysis of test run data shows that ML-models have a tendency to predict conservatively for leaning angles of higher magnitude. Through signal processing methods the prediction error can be reduced slightly for certain cases at the cost of reduced performance in other cases. Due to the nature of the demonstration platform, which should balance around low leaning angles, the processing is optimised around these cases.

Keywords: State estimation, Machine Learning, Neural Networks, Inverted Pendulum

Acknowledgements

During the realization of this report and project we have received plenty of help and advice and would like to express our gratitude. Firstly to our examiner Professor Jonas Sjöberg who has given us advice and recommendations for how to proceed during challenging technical obstacles and through his feedback allowed us to keep up a sufficient quality of the work. Furthermore, we would like to thank the team at Knowit Connectivity for their support both in funding the project but also for all the helpful advice in how to proceed. We would like to dedicate a special thanks to our supervisor Carl-Henrik Hult who has shown genuine interest in the project and helped us move forward throughout the whole process.

Jonathan Almgren, Lasse Kötz, Gothenburg, June 2023

List of Acronyms

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

ANN, NN	Artificial Neural Network, Neural Network
CNN	Convolutional Neural Network
COM	Center Of Mass
DNN	Deep Neural Network
DMP	Digital Motion Processor
FBD	Free-Body Diagram
GPIO	General Purpose Input-Output
IMU	Inertial Measurement Unit
LR	Linear Regression
MAE	Mean Absolute Error
ML	Machine Learning
MPN	McCulloch-Pitts Neuron
MPU	Memory Protection Unit
MSE	Mean Squared Error
PCB	Printed Circuit Board
PID	Proportional–Integral–Derivative
PLA	Polyactic Acid
PTQ	Post-Training Quantization
RGB	Red Green Blue
SGD	Stochastic Gradient Descent
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
WIP	Wheeled Inverted Pendulum

Nomenclature

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

Indices

i,j	Indices for Neural Network weights
t	Index for time step
l	Index for layer in Neural Network

Parameters

α	Lower quantization bound, Filter constant
θ	Leaning angle
η	Learning rate
β	Upper quantization bound
b	Neural Network bias(es)
L	Pendulum length
m_c, m_p	Cart and pendulum masses
k	Convolutional Kernel
f_c	Cutoff frequency
f_t	Sampling rate
g	Gravitational constant
t_c	Pendulum correction time
D	Wheel diameter
R_a	Internal Resistance
L_a	Inductance

Variables

F	Thrust
z	Weighted input, zero point
L_{MAE}	Mean Absolute Error Loss
L_{MSE}	Mean Squared Error Loss
w	Neural Network weights
$\boldsymbol{x}$	Input vector
$\boldsymbol{x}_q$	Quantized input
X_{filt}	Filtered signal
$\hat{y}$	Predicted states
y	Ground-truth target states
Φ	Activation function

Contents

List of Figures

List of Tables

1
Introduction

This chapter presents an overview of the thesis, including subject background, motivation, proposed solution as well as the scientific contribution.

1.1 Background

State estimation is an underlying problem and task within robotics and computer vision that involves estimating the state of a system based on various sensor data. With increasing availability of cameras and growing interest in visual sensing, camera-based state estimation has grown to become a well important topic for researches and companies. Compared to many other sensors, a camera offers the possibility of collecting much more information from the environment and can be used to also include features like object detection to reach higher levels of intelligence in robotics.

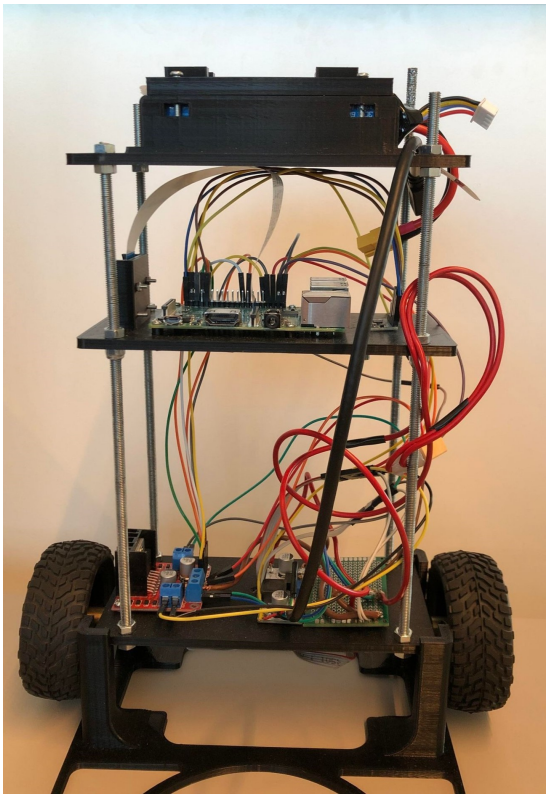
When controlling the desired angles of something like a drone or an inverted pendulum the most used method has been to use an Inertial Measurement Unit (IMU) to observe the relevant states. A complementary filter can be applied to get good estimates for the states in most cases, but a camera can be argued to be much more powerful since it also allows for much more data to be collected. Great success in controlling things like synchronised drone swarms have been reached by using cameras for state estimation [2]. Companies like Tesla have also decided to use machine vision to estimate many states for their cars instead of more traditional sensors. For example the ultrasonic sensors for parking assistance were removed in favour of machine vision.

Enabling efficient inference algorithms on edge devices and other resource-poor hardware has become increasingly important during recent years, as a consequence of a growing industry within Internet of Things (IoT) applications and embedded systems. This puts emphasis on development of algorithms that are both computationally efficient and operate at low latency and high inference speeds.

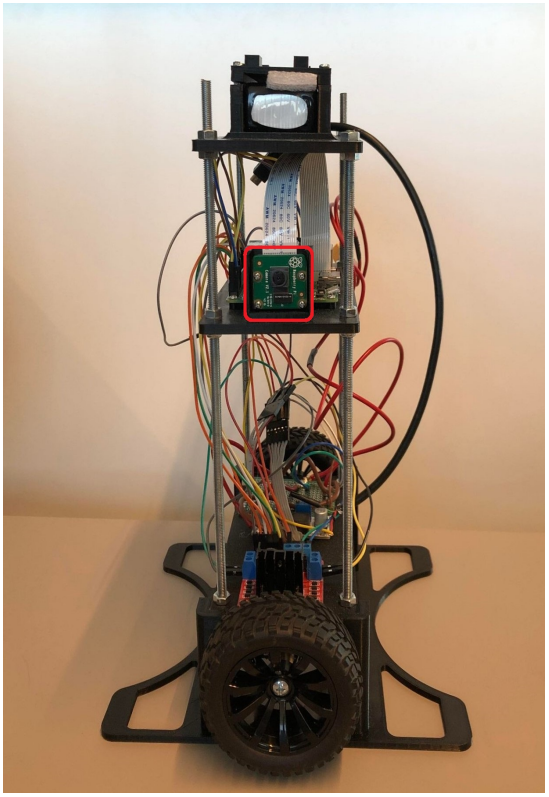
1.2 Overview

During this Master's thesis, a vision-based state estimator that detects leaning angles in real time on an embedded system is implemented, with the use of Machine Learning (ML) and Artificial Neural Networks (ANNs). The Neural Networks estimate the states given live image data from a Raspberry Pi camera module mounted on the system. The

estimated states are further validated using an IMU to compare accuracy using the Mean Absolute Error loss function (MAE). The specific investigated models are Linear Regression (LR) and Convolutional Neural Networks (CNNs) of VGG16 and MobileNetV2 architectures. Training is done with three different image resolutions (64×64, 96×96 and 128×128) for each model in order to examine how inference speed and prediction accuracy varies. Following its' training, the most accurate ML-model is deployed on a Wheeled, Inverted Pendulum (WIP) robotic system prototype to be tested in real-time.



(a) Front view with camera module to the left in image, pointing towards the side.



(b) Side view with the camera module marked in red on the second platform.

Figure 1.1: Robotic system prototype.

The machine vision works together with two cascade Proportional–Integral–Derivative (PID) controllers to make up a system that balances a custom built physical two-wheeled robot. In Figure 1.2 the system can be seen in the form of a flowchart.

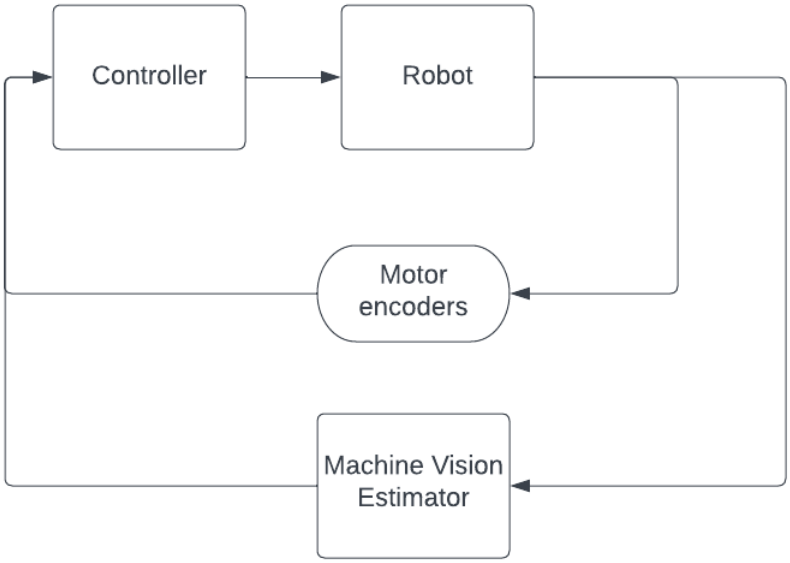


Figure 1.2: Robotic system flowchart.

1.3 Scientific Contribution

The primary scientific contributions by the work done during this thesis are the following:

- A novel Neural Network state estimator based on MobileNetV2 architecture which estimates the leaning angles given only image data.
- A real-time application for neural networks running inference on resource-constrained edge devices and robotic systems.
- A physical construction of a wheeled inverted pendulum robotic system showcasing the above mentioned results in a prototype application.
- Quantitative comparison of state estimation accuracy between three different architectures.

2

Mathematical Models

In this chapter, the mathematical model of the dynamical system is described and the governing equations are derived. For simplicity's sake, they are derived from via Newtonian dynamics. The equations are primarily used in Chapter 5 for designing the controller and in Section 3.1.3.1 used to choose adequate DC-motors.

2.1 The Wheeled Inverted Pendulum

A Wheeled Inverted Pendulum (WIP) is defined as a pendulum with its' center of mass position above its' pivot point, which in turn is placed on a horizontally moving base. The WIP is a classic problem within dynamics and control theory, and often is used as a benchmark to demonstrate various control strategies and theories. In 2D, it can be simplified as an inverted pendulum on a cart as shown in Figure 2.1. The pendulum is fixed on top of the cart, and can freely rotate about the z-axis.

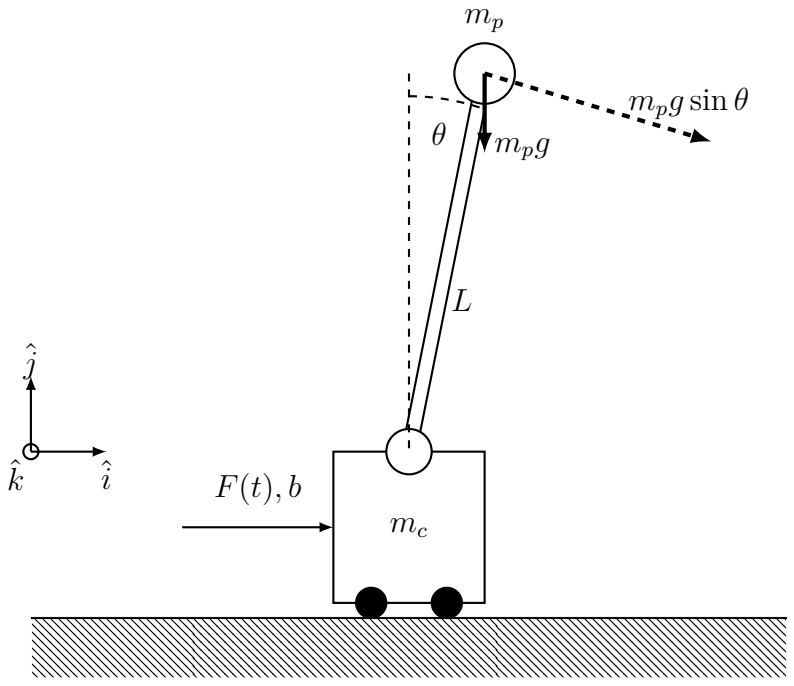


Figure 2.1: Wheeled, inverted pendulum system with the associated forces and physical properties.

In order to balance the pendulum towards the equilibrium point ($|\theta| > 0$), a horizontal

force $F(t)$, needs to be applied to the cart. The parameters of the system are explained in Table 2.1.

Table 2.1: System variables

Variable	Explanation	Unit
$F(t)$	Applied thrust from motors	$[N]$
b	Damping coefficient	$[kg/s]$
m_c	Mass of cart	$[kg]$
m_p	Mass of pendulum	$[kg]$
L	Distance to pendulum center of mass	$[m]$

2.1.1 Cart

By isolating the cart system into a seperate free body diagram, Figure 2.2. The cart is accelerated by a force, F, which stems from the control system, as well as reaction forces R_x and R_y that stem from the pendulum. The cart is also dampened by a damping force $b\dot{x}$.

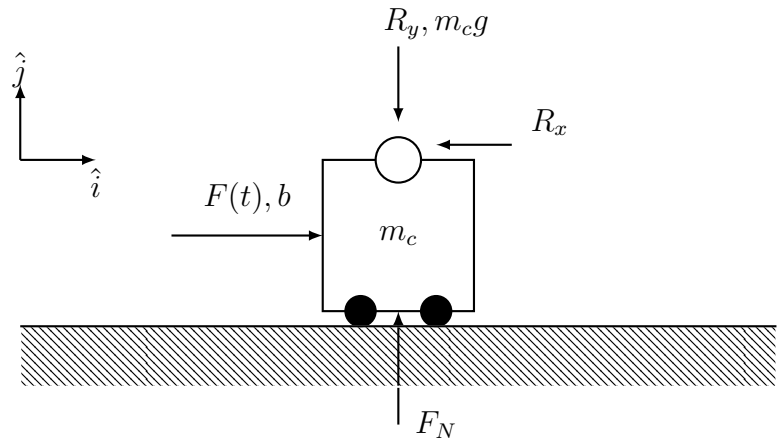


Figure 2.2: FBD of cart system.

This isolated subsystem yields the following Newtonian dynamics in 2D:

$$\begin{aligned} \hat{i} &: F(t) - b\dot{x} - R_x = m_c\ddot{x} \\ \hat{j} &: F_N = R_y + m_c g, \end{aligned} \tag{2.1}$$

where x is the horizontal-coordinate of the carts' center of mass.

2.1.2 Pendulum

Repeating the process of isolating the pendulum into a free-body diagram, Figure 2.3.

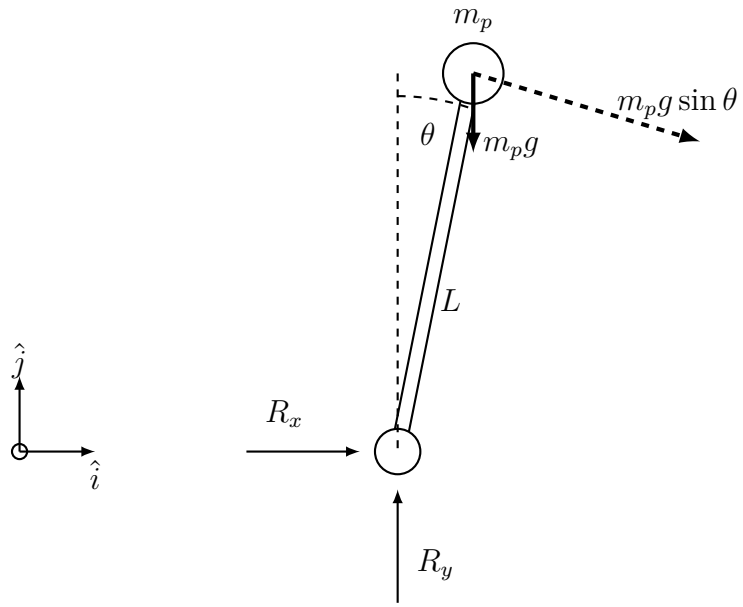


Figure 2.3: Isolated inverted pendulum FBD with massless rod of length L.

The corresponding dynamics become:

$$\begin{aligned} \hat{i} &: R_x = m_p \cdot \ddot{x}_p \\ \hat{j} &: R_y - m_p g = m_p \cdot \ddot{y}_p \end{aligned} \tag{2.2}$$

The system dynamics need to be expressed in terms of the system states where x_p and y_p are the coordinates of the load such that

$$\begin{aligned} x_p &= L\sin\theta + x \\ y_p &= L\cos\theta \end{aligned} \tag{2.3}$$

By differentiating (2.3) and substituting the states into (2.2), we receive

$$\begin{aligned} R_x &= m_p \left[L(-\sin\theta \cdot \dot{\theta}^2 + \cos\theta \cdot \ddot{\theta}) + \ddot{x} \right] \\ R_y &= m_p \left[g - L(\cos\theta \cdot \dot{\theta}^2 + \sin\theta \cdot \ddot{\theta}) \right] \end{aligned} \tag{2.4}$$

Substituting the expression for R_x and R_y into (2.1) yields the governing equation:

$$\begin{cases} F(t) = M\ddot{x} + b\dot{x} + m_p L \left[-\sin\theta \cdot \dot{\theta}^2 + \cos\theta \cdot \ddot{\theta} \right] \\ M = m_c + m_p \end{cases} \tag{2.5}$$

These non-linear equations are then used in primarily Chapter 5 to design a controller that can balance the wheeled inverted pendulum around its equilibrium point.

3
Hardware

In this chapter, we present the hardware used to design a robotic, wheeled inverted pendulum system used to build the state estimator. The various components and selections thereof are described thoroughly guided by previously introduced mathematical models.

3.1 Hardware Design

The electrical components used to replicate a wheeled, inverted pendulum system used throughout this thesis are shown in Table 3.1 and are explained more thoroughly in the following sections.

Table 3.1: List of used electric components.

Type	Component
Platform	Raspberry Pi 3
Motors	Planetary gear DC-motors 350 RPM 3-12V
Camera module	Raspberry Pi camera v2.1
Power Supply	Li-Po 3S 11,1V 30C 2200mAh
Motor driver	L298N
Tensor Processing Unit	Google Coral USB accelerator @2TOPS/Watt [7]
Inertial Measurement Unit	MPU-6050
Voltage regulator	M2596S

3.1.1 Frame and platforms

The robot used to showcase the state estimator, consist of three parallel, horizontally oriented platforms that are layered vertically with offsets, as well as four threaded rods that connect the platforms. The platforms are fixed along the rods to reduce play and thereby shifting of the robot's Center Of Mass (COM). The three platforms consists of Polyactic acid (PLA) and are 3D-designed before being additively manufactured using 3D-printing. In Figure 1.1, the finished constructed and assembled prototype is shown, including electric components.

3.1.2 Electrical circuit

The electrical circuit consists of several different components with varying desired voltages. The circuit in its entirety can be seen in Figure 3.1. The nature of a self-balancing

robot requires constant change of input voltages and current to the motors which puts a high demand of supplied current from the battery. A Li-Po battery is capable of outputting high amounts of current quickly while also having a very high weight to power ratio which made it an ideal choice for this robot. The nominal voltage of a Li-Po cell is 3.7V and to comply with the voltage demand of the DC-motors a battery with three serial connected cells is used which comes out to a nominal voltage of 11.1V which packs a maximum discharge rate of 132A burst and 66A continuous.

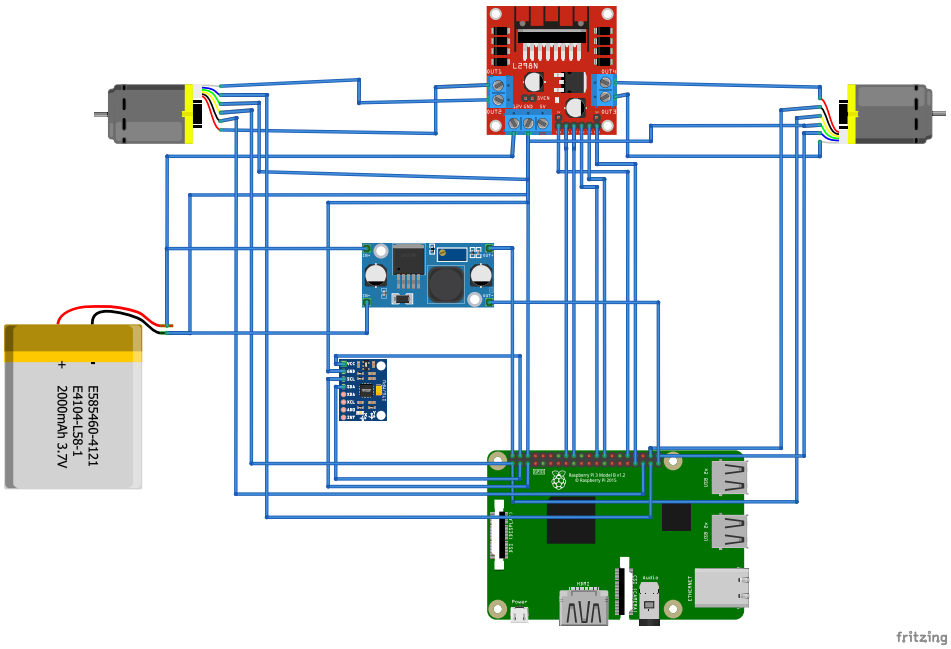


Figure 3.1: Circuit diagram

Following the demands described in Chapter 3.1.3 a pair of 12V dc-motors with a rated load current of 1A [4] was chosen to handle the actuation of the robot. They are driven by a driver module based on the L298 H-bridge which was chosen because of it's cheap price and being widely available. The module specifies a voltage drop of 1.4V which during testing gives a output voltage of just over 12V to the motor when using the fully charged Li-Po battery.

The Raspberry Pi requires a voltage of 5V which means that the input Li-Po battery's 11.1V needs to be stepped down with a voltage regulator to safely be used together. A module based on the LM2596S voltage regulator is included and connected in parallel to the L298N module which steps down the input voltage to the Raspberry Pi to 5V.

The encoders on the motors which tracks their current speed accepts a wide spectrum of voltages ranging from 2.4V to 26V which can be satisfied directly from the Raspberry Pi's GPIO pins. This is true for the MPU-6050 as well. To connect a common ground for all component a simple PCB is used to simplify wiring. This is not shown on the circuit diagram but exists on the physical hardware.

3.1.3 DC-motors

The robot uses two brushed DC-motors equipped with a planetary gearbox that uses a rated voltage of 12V and rated current of 1A. They offer a maximum speed of 350 rpm and can each output 0.16 Nm of torque. To keep track of the their angular velocity the are equipped with hall effect rotary encoders with a resolution of 3 ppr before the gearbox which gives a total resolution of 1050 ppr for the wheels. The were chosen to fulfil the requirements set up in Section 3.1.3.1

3.1.3.1 Requirements for DC-motors

The choice of DC-motors is dependent on the system's design requirements. In order for the robot to balance towards the equilibrium point, there should exist a forward thrust, $F(t)$, for any angular perturbation, $\delta_\theta = \theta(t)$, powerful enough to cause the pendulum to accelerate towards its operating point. Assuming a maximum deviation of $|\delta_\theta^{max}| = \frac{\pi}{9}$ radians from the operating point and a desired correction time of $t_c = 0.5$ seconds, we receive the following initial conditions at the turning point:

$$\begin{cases} \theta(0) = \frac{2\pi}{18} = \frac{\pi}{9} \\ \dot{\theta}(0) = 0 \\ \ddot{\theta}(0) = -\frac{2\pi}{9} \end{cases} \tag{3.1}$$

Solving for $F(t)$ and assuming $b = 0$, $\ddot{x}_{max} = 1$ in eqs. (2.5) gives

$$\begin{aligned} F(0) &= M + m_p L \left[-\sin\theta(0) \cdot \dot{\theta}(0) + \cos\theta(0) \cdot \ddot{\theta}(0) \right] \\ &= M - m_p L \cos\left(\frac{\pi}{9}\right) \cdot \frac{2\pi}{9} \\ \implies a_c &\approx 2.34 \text{m/s}^2 \end{aligned} \tag{3.2}$$

Acceleration a_c is constant from $v_0 = v(0)$ to $v_t = v(\frac{1}{2})$ and a minimum required angular velocity ω_{RPM} can be determined such that

$$\begin{aligned} v_t &= \frac{a_c}{2} + v_b = 1.17 + v_0 \\ \omega_{RPM} &= \frac{60}{\pi D}(1.17 + v_0) \end{aligned} \tag{3.3}$$

The wheel diameter D is measured as 7.7mm and assuming an initial velocity $v_0 = 0$ gives

$$\omega_{RPM} \approx 290 \,\text{rpm}, \tag{3.4}$$

which is sufficed by the chosen Planetary gear 350RPM DC-motors.

3.1.3.2 Motor identification

To properly design a robust controller the relationship between input voltages and output speed of the motors is needed. A mathematical description of a DC-motor can be derived using Kirchhoff's voltage law leading to the following differential equation:

$$-u(t) + R_a i_a(t) + L_a \frac{d}{dt} i_a(t) + u_m = 0 \tag{3.5}$$

As can be seen from the equations the DC-motor has three traits of interest. It's internal resistance R_a, inductance L_a and it's back emf u_m. The inductance of the motor will be relatively very small when compared to its internal resistance. This means that to further simplify the equation we can remove that too. The output torque and as a consequence the output speed will depend on the current passing through the motors with a relationship that can be estimated as a linear function according to:

$$\tau = K_m i_a(t) \tag{3.6}$$

The output torque will then lead to a rotation of the wheels whose speed will depend on moment of inertia and friction which in turn will depend on the rotational speed of the wheel according to:

$$J\frac{d}{dt}\omega(t) = T_d(t) - b\omega(t) \tag{3.7}$$

By performing a Laplace transform on the simplified equations and rearranging it leads to the following transfer function:

$$G(s) = \frac{\Omega(s)}{U(s)} = \frac{K_m}{K_u K_m + R_a(b + Js)} \tag{3.8}$$

From this we see the important result that the transfer function from voltage input to wheel rotation speed can be estimated as a first order transfer function with one pole and no zeros.

To estimate the constant parameters of equation 3.8 white Gaussian noise inputs are sent to the motors and feedback from rotational velocity feedback from the encoders are noted. Using Matlab's System Identification toolbox [12] the inputs and outputs can be used, in combination with the knowledge that the transfer function can be estimated as a first order transfer function, to estimate the transfer function in 3.8 and leads to the result

$$G(s) = \frac{0.004124}{1 + 5.428s} \tag{3.9}$$

This result is used in Chapter 5 to design the both controller that syncs the motors and the main balancing controller.

3.2 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) is a component that measures the forces acting on an object as well as it's angular velocity. Internally it usually consists of an accelerometer and a gyroscope. The accelerometer measures forces in 3 directions which, when the object is standing still, corresponds to how the gravity is affecting the object. By examining which direction gravity is pointing one can calculate the current orientation of the object. The gyroscope on the other hand measures the objects angular velocity which when integrated can estimate the current orientation. The accelerometer is reliable when the object is in rest but inaccurate when currently moving while a gyroscope on the other hand is reliable while moving but the integration will cause it to drift over time.

A complementary filter can be used to fuse these two sensor values together by applying a low-pass filter to the accelerometer and a high-pass filter on the gyroscope to get accurate orientation readings over time. An alternative is to use a Kalman filter to fuse the two values together which, assuming certain conditions hold, can give even more accurate estimations at the cost of a higher complexity.

In this project an IMU in the form of a MPU-6050 module is used to partly tune the regulator of the robot and partly to label collected camera data for training. The MPU-6050 contains a 3-axis acceleromter, a 3-axis gyroscope as well as Digital Motion Processor (DMP) which can process the collected data. In this project the DMP is used to apply a low-pass filter with a cutoff frequency of 40 Hz on the incoming data to filter out sensor noise.

By performing these calculations directly on the DMP, valuable processing power on the Raspberry Pi can be saved for the resource-intensive neural network. The communication between the MPU-6050 registers and Raspberry Pi is done by i2c with a maximum speed of 400kHz. The speed is however limited by the accelerometer readings which update at a frequency of 1000 Hz. When using the DMP this is lowered further to 200 Hz. These sampling rates are important when deciding the parameters of the controller and affect the robustness of the controller.

3.3 Tensor Processing Unit

A Tensor Processing Unit (TPU) is a specialized processor designed specifically for accelerating machine learning workloads. It is optimized for processing large-scale, highly parallel computations that are common in deep learning algorithms. The Raspberry Pi 3b used in the projects has limited processing power and the TPU is used to offload some of high demanding processes of the angle predictor.

At a high level, a TPU consists of a large number of processing cores organized into clusters, with each core capable of performing a large number of multiply-accumulate operations per second. These cores are connected by a high-speed mesh network that allows for efficient communication between them.

The TPU is designed to handle large matrices of data, which are fundamental to many machine learning algorithms. The specific TPU used during this thesis' robotic system is the Coral USB accelerator, developed by Google. It can perform matrix multiplication and other linear algebra operations at a speed of 4 Trillion Operations Per Second (TOPS), allowing it to process large amounts of data in parallel.

In addition to the processing cores, a TPU also includes a number of on-chip memory banks that can be accessed with very low latency. This allows for efficient data movement between the processing cores and the memory, which is critical for achieving high performance.

Overall, the TPU is designed to provide highly efficient and scalable processing for deep learning workloads. By optimizing for the specific requirements of these workloads, it can provide significant speedups over a resource-constrained edge device, such as the Raspberry Pi 3.

4

State Estimator

In this chapter, the state-estimator of the pendulum is presented in further detail, using both a Linear Regression model, as well as both untrained and pre-trained Convolutional Neural Networks, including VGG16 [20] and MobileNetV2 [19]. The state-estimators are formulated such that the difference in leaning angles is minimized. It is also presented how quantization is applied in post-processing, in order to mitigate latency and computation when controlling the wheeled, inverted pendulum with said state estimator in real-time.

4.1 Introduction to Neural Networks

Artificial Neural networks (ANNs) or simply Neural Networks (NNs) are types of machine learning models that can learn to perform complex tasks by training on large amounts of data. They are inspired by the structure and function of the human brain, which consists of interconnected neurons that communicate with each other to process information. The flow of information through a NN is heavily dependent on the specific architecture that is being used. In the below sections, three major types of NNs are introduced and described in further detail, namely Linear Regression (LR) and two different types of Convolutional Neural Networks (CNNs).

4.1.1 Linear Regression

Linear Regression in the domain of Machine Learning, is a type of supervised learning algorithm [3] used to model the relationship between a dependent variable, y, and one or more independent variables, $\boldsymbol{x}$ also known as regressors [14]. A model predicts a value by using a function, f, such that

$$f : \mathbb{R}^n \to \mathbb{R}. \tag{4.1}$$

This mapping is often referred to as making a prediction. In simple LR, which is the most basic form of LR ($n = 1$), a prediction $\hat{y}$ can be computed as

$$\begin{aligned} \hat{y} &= \Phi(z) \\ z &= f(w, b, x) \\ &= wx + b, \end{aligned} \tag{4.2}$$

where x is an independent input variable and Φ is some function, known as the activation function. The role of the activation function is to provide the network with nonlinearity, by taking the weighted sum for a neuron as argument. The bias b and weight w

are trainable parameters which are tuned during training of the model. A schematic representation of simle LR in a Neural Network is shown in Figure 4.1.

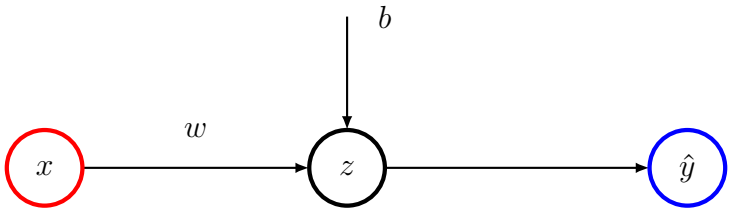


Figure 4.1: Simple LR neuron.

In multiple LR, a vector $\boldsymbol{x}$ of independent variables is instead fed to the network. Expanding eq. (4.2) into multiple LR, a prediction is computed as the weighted sum

$$\begin{aligned} \hat{y} = f(\boldsymbol{w}, b, \boldsymbol{x}) &= w_1 x_1 + w_2 x_2 + ... + w_n x_n + b \\ \hat{y} &= \Phi\left[\sum_{i=1}^{n}(w_i x_i) + b\right] \\ &= \Phi(z) \end{aligned} \quad , \tag{4.3}$$

where $w_1, ..., w_n$ are the tunable weights corresponding to inputs $x_1, ..., x_n$. In Neural Networks, this is often referred to as the McCulloch-Pitts Neuron (MPN) [13]. A schematic representation of the MPN is shown in Figure 4.1.

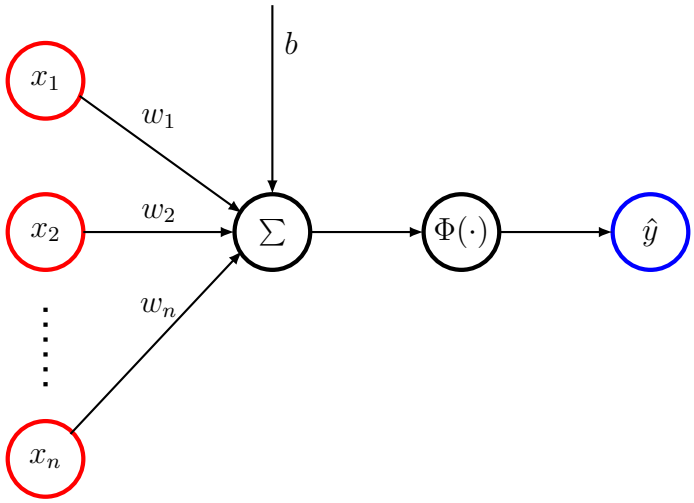


Figure 4.2: McCulloch-Pitts Neuron illustrated with corresponding inputs, weights and bias.

In the case of predicting $m > 1$ outputs, the weighted sum from eq (4.3) is repeated for each output j, such that

$$\hat{y}_j = \Phi\left[\sum_{i=1}^{n}(w_{ij} x_i) + b_j\right] = \begin{bmatrix} w_{11} & \dots & w_{n1} \\ \vdots & \ddots & \vdots \\ w_{1m} & \dots & w_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_m, \end{bmatrix} \tag{4.4}$$

where Φ is the chosen activation function of the layer. The activation function is a nonlinear function that provides the model with complexity.

4.1.2 Convolutional Neural Networks

A convolutional Neural Network (CNN) is a network architecture which is designed to utilize spatial dependencies, also known as features, within an input-array during a forward-pass [15]. In its' simplest form, this is done by making use of the discrete 2D convolution operation [6]

$$o[m,n] = \sum_{i=-s}^{s} \sum_{j=-s}^{s} \boldsymbol{x}[i,j] * k[m-i, n-j], \tag{4.5}$$

where s is the size of the input array. Here, k is the so-called kernel of the convolutional layer with dimensions $[m,n]$. The kernel is trainable and consists of a set of weights which are tuned during backpropagation. The product $\boldsymbol{x}[i,j] * k[m-i, n-j]$ for any given indices i,j in the input, is also commonly referred to as the Receptive Field (RF). CNNs consist of multiple such convolutional layers built on top of each other, which provides the model with nonlinearity and abstraction. This proves useful when it comes to feature recognition in the input image. In the case of using RGB imagery as inputs, i.e. a 3D-array, the 2D convolution operator will be repeated over each channel in the image, resulting in an output with the same number of dimensions as the input. In Figure 4.3, the convolution is represented visually, with a dummy 2D input.

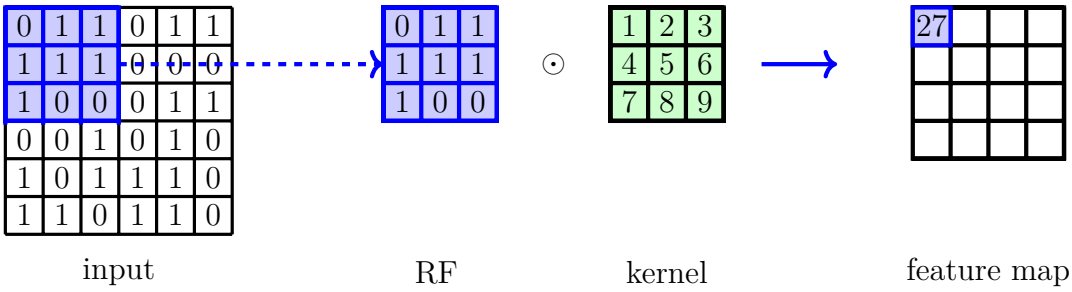


Figure 4.3: Example of a convolutional layer and its' forward pass on a greyscale image. The Receptive Field (RF) consist of a 3x3 square and is element-wise multiplied by the kernel to yield the output. The local receptive field is later convolved through the entire array according to (4.5)

4.1.2.1 VGG16

The Visual Geometry Group 16 (VGG16) model [20] is a type of CNN which is 16 layers of trainable parameters deep, consisting of 13, 5 and 3 convolutional, max-pooling and dense layers, respectively. It was introduced in 2014 and is pre-trained on the ImageNet classification challenge [5]. The depth of VGG16 corresponds to around 138 million parameters which contributes to a slow forward-pass, relative to MobileNetV2.

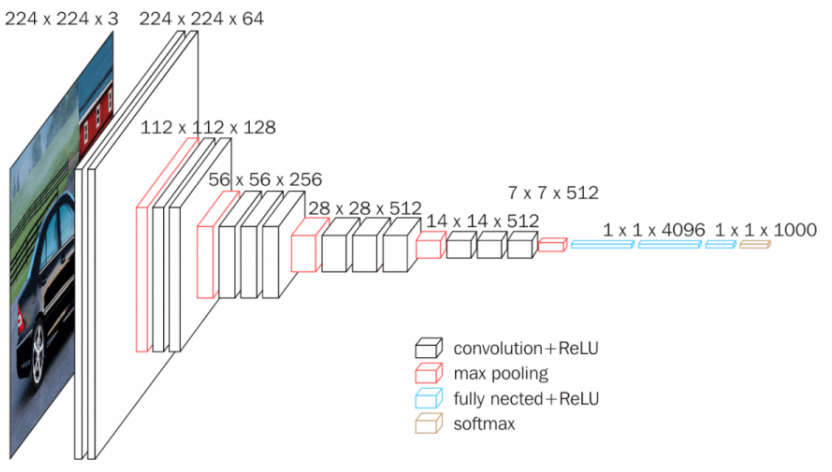


Figure 4.4: VGG16 architecture. Available via [21] under Creative Commons license (CC BY 3.0).

4.1.2.2 MobileNetV2

MobileNetV2 is a convolutional-type NN which gained traction following its' publish in 2018 [19]. It consists of 19 so-called residual bottleneck layers, which were also introduced in the paper. Just like VGG16, MobileNetV2 is pretrained on the ImageNet classification task [5], with input resolution 224x224. The architecture is optimized for inference speed, while maintaining a high accuracy compared to its' predecessor MobileNetV1 [8], making it a suitable choice for on-device DNNs, especially within the realm of image classification and image segmentation. The increase in inference speed relative to vanilla CNNs is mainly due to the implementation of depthwise seperable convolutions, linear bottlenecks and inverted residuals. The default, pretrained MobileNetV2 architecture consists of a total of 2,259,265 parameters, making it significantly smaller than VGG16.

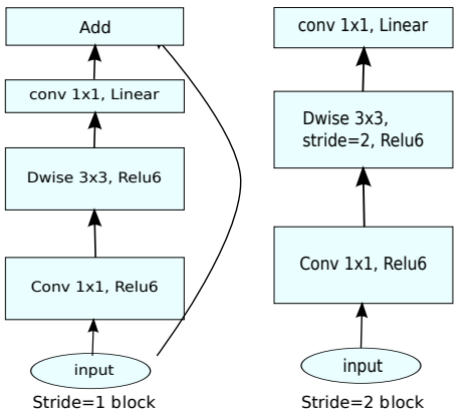


Figure 4.5: Convolutional blocks in MobileNetV2

Depthwise seperable convolution functions similar to standard convolution, with the exception that it is performed in two seperate layers; depthwise and pointwise convolution. It works by seperating each channel in an input image and performing 2D convolutions

(4.5) over each channel independently. The resulting output layers are restacked on top of each other, before a pointwise convolution (1×1 kernel) is applied. As shown in Figure 4.3, the required number of computations for an image i with resolution $h_i \times w_i \times d_i$ in a vanilla convolutional layer is

$$C_i = k^2 h_i w_i d_i o_i, \tag{4.6}$$

where o_i is the size of the output. In depthwise seperable convolutional layers, the computations are

$$C_i^{'} = k^2 h_i w_i d_i + o_i h_i w_i d_i = h_i w_i d_i (k^2 + o_i), \tag{4.7}$$

which corresponds to a reduction of almost $\frac{C_i}{C_i^{'}} \approx k^2$ per layer, where k is the kernel size. In MobileNetV2, the kernels are of sizes $k = 3$, which leads to a cost reduction of a factor 8 to 9 per depthwise convolutional layer.

A necessary modification to the default MobileNetV2-architecture, is to resize the final layers of the network from the default 1000 output neurons to 1, in order to fit it to a regression task with one prediction, rather than a classification task with one prediction per class.

4.1.3 Activation functions

The activation function Φ introduces non-linearity into a Neural Network, allowing for more complex patterns between input- and output data to be learned. Furthermore, they mitigate the famous vanishing or exploding gradient problem [16] by avoiding saturation of neurons. Every ML-model during this thesis is constructed using the Rectified Linear Unit (ReLU) activation function, which is defined as

$$\Phi_{ReLU} = max\{0, x\}. \tag{4.8}$$

A visual representation of ReLU is shown in 4.6.

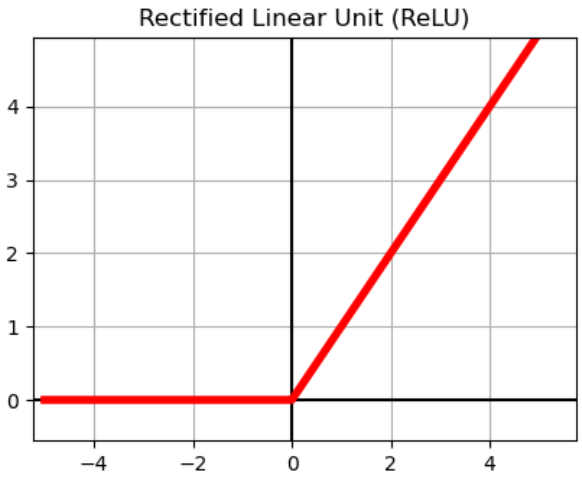


Figure 4.6: ReLU activation function.

4.1.4 Loss function

A model is, during the training phase, continuously evaluated on its' performance in order to tune its' parameters and improve over time. This is commonly done by using a loss-function, which is an arbitrary penalty metric indicating how poorly the model is performing. Depending on the specific type of problem, the most suitable loss-function may vary. In regression however, common choices are to use the Mean Absolute Error (MAE) [23] which is measured by averaging over the absolute differences between predictions, $\hat{\boldsymbol{y}}$, and ground-truth labels, $\boldsymbol{y}$, of all samples, n, according to

$$L_{\text{MAE}} = \frac{1}{n}\sum_{i=1}^{n}|\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)}|, \tag{4.9}$$

or the Mean Squared Error (MSE) [10] which squares the differences instead, such that

$$L_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2. \tag{4.10}$$

Comparing (4.10) and (4.9), it can be said that MSE penalizes mispredictions greater than 1 heavier, but MAE penalizes mispredictions less than 1 heavier. This becomes increasingly clear when plotting the two different loss functions together, as shown in Figure 4.7. In the scope of this project, a higher precision model is preferable, while errors with high errors can be filtered away. MAE is therefore deemed to be a more suitable choice.

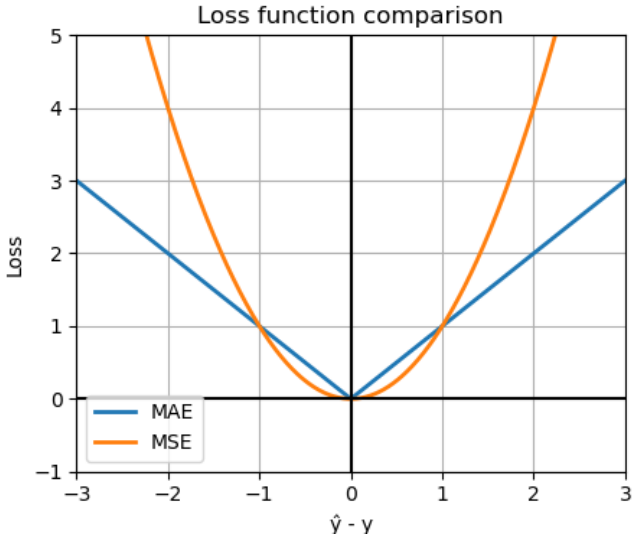


Figure 4.7: Comparison between losses MAE and MSE as functions of the prediction difference, $\hat{y} - y$.

4.1.5 Stochastic Gradient Descent and backpropagation

Backpropagation is a widely-used algorithm for training Artificial Neural Networks (ANNs) in a supervised learning setting. The goal of supervised learning is to train a neural network to produce output values that closely match the target values for a given set of input

data, i.e minimizing for the selected loss function (4.9). Backpropagation achieves this goal by iteratively updating the weights and biases of the network based on the gradient of given loss function with respect to these parameters [18].
The backpropagation algorithm consists of two phases: forward propagation and backward propagation. During the forward propagation phase, the input data is fed through the network and the output is calculated using the current values of the weights and biases. Specifically, for each layer l in the network, the activation a^l of each neuron is calculated as follows:

$$a^l = \Phi(z^l) \tag{4.11}$$

where Φ is the activation function, and z^l is the weighted input to the l-th layer.
During the backward propagation phase, the error of each neuron in the network is computed by backpropagating the error from the output layer to the input layer. The error of a neuron j in layer l is defined as follows:

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} \tag{4.12}$$

where L is the loss function. The error is then used to compute the gradient of the loss function with respect to the weights and biases of the network. Specifically, for each weight w_{jk}^l and bias b_j^l in the network, the gradient is computed as follows:

$$\frac{\partial L}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{4.13}$$

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \tag{4.14}$$

The weights and biases are then updated using an optimization algorithm such as gradient descent:

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \frac{\partial L}{\partial w_{jk}^l} \tag{4.15}$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial L}{\partial b_j^l} \tag{4.16}$$

where η is the learning rate.

4.2 Data

The dataset consists of 40,000 unique images of various scenery such as buildings and other places and architecture and was first introduced in scene recognition tasks [24]. The images are mapped to their respective ground-truth labels, in the form of leaning angles as positive (clockwise) or negative (counterclockwise) deviations from the equilibrium point. The rotations are randomly rotated in the range $[-30°, 30°]$, with a uniform probability distribution as made available in [22]. During training of the models, a train-, validation- and test-split ratio of $[80\%, 10\%, 10\%]$ is used. The label distribution is shown in Figure 6.1.

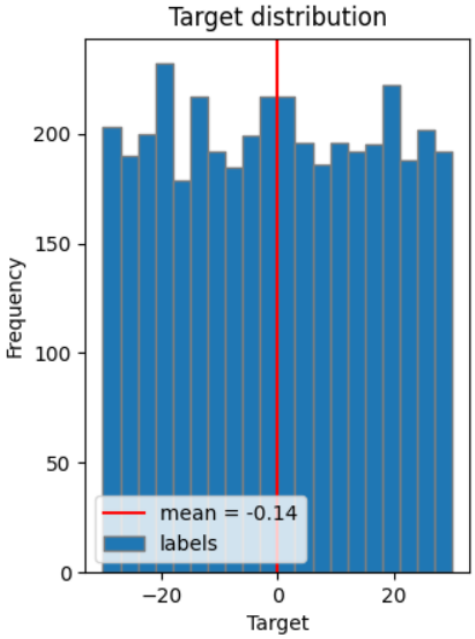


Figure 4.8: Distribution of 4,000 randomly sampled labels in the dataset.

4.2.1 Prep-rocessing

All data is pre-processed before being forward-propagated through the network, wether during training or deployment inference. The pre-processing consists of:

- Reshaping images into 128×128, 96×96 and 64×64 resolution, respectively. The raspberry pi camera module is ought to have at least the same update frequency as the state estimator, so that the estimator never casts a prediction on two identical frames in sequence. Furthermore, the images of the dataset are of 128×128, meaning that upscaling would increase compute without providing more information, unless using a lossless algorithm such as more sophisticated ML-based upscaling techniques.
- Normalizing inputs into the range [0, 1].

This type of pre-processing, and especially the rescaling part, is common practice within ML applications and regression in particular. The main reason for this is to make the model regard each feature equally relative to the weights which prevents single neurons

in the network to saturate and get stuck by suffering from the vanishing gradient problem [16]. A sample datapoint is shown in Figure 4.9.

Figure 4.9: Sample 128×128 image mapped to the ground-truth target 12.27°.

4.3 Post-processing

In this section, the post training processing methods are described further. Specifically, the method of Post-Training Quantization is explained and how it mitigates latency during a forward-pass in the network, by sacrificing some prediction accuracy.

4.3.1 Post-Training Quantization

When deploying a Neural Network on a mobile but resource-constrained device, such as the Raspberry Pi 3, one can make use of Post-Training Quantization (PTQ) in order to mitigate computational requirements. PTQ is a optimization technique, in which the parameters of a model are converted from the commonly used 32-bit default floating point values $x \in [\alpha, \beta]$ to a lower b-bit representation, such that

$$x_q \in [\alpha_q, \beta_q] = [-2^{b-1}, 2^{b-1} - 1], \tag{4.17}$$

where b is the number of bits used to represent the parameters. This can be done by rescaling a parameter such that

$$x_q = round(\frac{x}{s} + z). \tag{4.18}$$

In the case of $b = 8$, which is a common quantization choice, the consequence is that each of the model weights' memory footprint is reduced to a fraction of $\frac{8}{32} = \frac{1}{4}$. Such a mapping, from 32-bit floating points to 8-bit INT8 integers is shown in Figure (4.10).

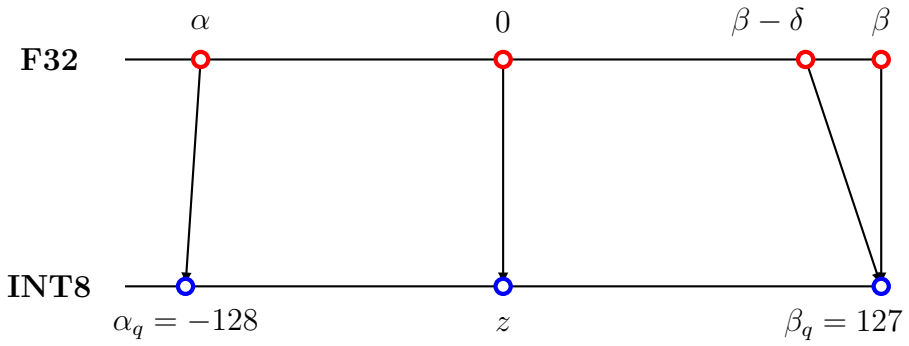


Figure 4.10: Quantization from floating point $x \in [\alpha, \beta]$ to a lower bit representation, $x_q \in [\alpha_q, \beta_q]$. Here, INT8-quantization is used on four different weights.

Additionally, in the case of quantizing all of the models parameters to 8-bit integer representation, the model can be deployed on integer-only hardware, such as the Coral USB accelerator [7].

However, as shown in Figure 4.10, quantization comes to a prize of unrecoverable precision loss due to its' usage of rounding (eq. (4.17)). Hence, two different weights β, $\beta - \delta$ which are close to each other might become mapped to the same quantized output, β_q. Thus, when dequantizing back to F32 representation again, there is no way to distinguish between $\beta - \delta$ and β from a pre-quantized state, signifying a loss in accuracy during the quantization process.

4.4 State estimation

The state estimation is designed by training a variety of Deep Neural Networks, both pre-trained models, by the use of transfer learning, as well as untrained models. A Linear Regression (LR) model is also implemented. The models are trained to predict the leaning angles given a set of input images. The label values y in the datasets are distributed continuously such that $y \in (-30, 30)$, making it a regression task.

4.4.1 Network architecture and deployment

The models are implemented using the open-source PyTorch library [17]. In order for the models to be compilable with the Google Coral USB accelerator TPU (3.3), they have to be converted in a series of steps, from the default pytorch .pt format to a TPU compileable .tflite format. This is done by the following series of conversions:

$$\text{.pt (NCHW)} \rightarrow \text{.onnx (NCHW)} \rightarrow \text{OpenVINO (NCHW)} \rightarrow \text{.tflite (NHWC)}$$

When initially implementing the models in the PyTorch framework, the data is formatted in the NCHW (batch **N**, channel **C**, height **H**, width **W**) format, while the TensorFlow framework which is used on the coral accelerator TPU is optimized for NHWC. Therefore, when compiling a NCHW model on the TPU, the tensors are transformed back and forth

between the two formats, unnecessarily taking up significant computational resources. This problem can be omitted by using the openvino2tensorflow library [9] which supports the conversion.

4.4.2 Training

Training was carried out with a batch-size of 32 and using Stochastic Gradient Descent (SGD) as optimizer. The learning rate was set to $\eta = 5 \cdot 10^{-3}$ and the models were trained to minimize the Mean Absolute Error (MAE) loss function, Equation 4.9. All of the models were trained with a patience set to 5 epochs. Training parameters are shown in Table 4.1. The patience parameter determines how many epochs are allowed to pass without the model improving. If the validation loss decreases, the patience counter is reset. If not, the training process is stopped and the model parameters corresponding to the lowest previous validation loss is saved. The last layer in each model is modified to fit the selected image resolution, if needed and the number of output neurons is set to 1, representing the predicted angle. The existing softmax activation function is also removed in the VGG16-based and MobileNetV2-based state estimators. The training is carried out on a NVIDIA RTX A2000 Laptop Graphics Processing Unit (GPU).

Table 4.1: Training parameters

Setting	Quantity
Learning rate	$5 \cdot 10^{-3}$
Batch size	32
Patience	5 epochs
Optimizer	SGD

4.5 Signal Processing

The angular movement around the wheel axis for a two-wheeled inverted pendulum can be expected to be of relatively low frequency while much of the unwanted noise can be expected to be of a more high frequent nature. The noise can be counteracted by filtering the signal. Two primary methods of filtering is used to remove noise in this project, both of them digitally.

4.5.1 Low-pass filtering

The most simple way to process the signal is a standard low-pass filter that will filter out the high frequent noise but allow the low frequent signal to pass. Since computational speed is of constant essence a simple implementation of a low-pass filter will be used in the following way:

$$X_{filt}(t) = X_{filt}(t-1) + \alpha(X(t) - X_{filt}(t-1)) \tag{4.19}$$

Where $X_{filt}(t)$ is the filtered signal and $X(t)$ is the input signal. The parameter α is dependent on the desired cutoff frequency as well as the sampling rate. It will be

calculated according to

$$\alpha = \frac{1}{2\pi RC f_t} \tag{4.20}$$

where f_t corresponds to the sampling rate and RC is calculated according to:

$$RC = \frac{1}{2\pi f_c} \tag{4.21}$$

where f_c corresponds to the desired cutoff frequency.

4.5.2 Kalman Filter

The possibility of fusing the gyroscope data with the predicted angles from the camera data, and thus omitting the accelerometer, is examined with the use of the more sophisticated Kalman filter. A Kalman filter can produce a statistically optimal estimate under certain conditions [11]. By using the known characteristics of the noise for each input it will fuse values together. To be optimal it requires the noise to be normal distributed with zero average. For the IMU data the noise is quite close to this but the existing bias described in Section 6.6 hints that it might not be true for the ML model.

4.5.3 Implementation

The methods are evaluated and tested both in real time on the robot through a implementation in C-code and also by analysing data from tested runs in Matlab. The implementation of a low-pass filter is accomplished by following the exact formula showed in equation 4.19. While Matlab allows for more advanced implementation the same equation will be used there to make sure that the filter performance remains the same when run in real time. Matlab will then allow for many different parameters to be tested and evaluated.

5
Control System

In this chapter, the control system of the wheeled, inverted pendulum is described further. Theory behind the control system is explained as well as the design processes, including simulation and implementation of the controller.

5.1 Control Theory

The inverted pendulum is an inherently unstable system that needs active control to keep itself balanced. The one dimensional wheeled, inverted pendulum has two states, speed and angle, and one input. If the system states are decoupled and treated as separate they can be controlled by two separate cascade connected Proportional–Integral–Derivative controllers (PID).

In this case the first controller will handle the speed of the robot. It will receive closed loop feedback from the rotary encoders and output a desired leaning angle for the next PID controller which in turn will speed up or slow down the robot. The next PID controller will then receive closed loop feedback from the current leaning angle and determine the motor voltages needed to correct the leaning angle.

The computer vision leaning angle input can be expected to contain quite high amounts of noise compared to the IMU when faced with suboptimal conditions for the camera. Therefore the controller will needed to include enough robustness to handle this. Since the system will be controlled with two cascade siso PID controllers the robustness can be adjusted in the frequency domain using standard classic control tools.

Another important factor with regards to robustness is the sampling rate. Too slow of a sampling rate will make the system to slow to actively balance itself. The time constant of the DC-motors which is seen in equation 3.9 can be used to approximate the needed sampling time to maximise their use. As a value for the sampling rate the time constant times 20 is used which gives a desired sampling rate of around 100Hz. This will be used as a requirement on the speed of the neural network.

5.1.1 Simulation

To help with the controller design a Simulink simulation is used to see how different noise models will affect the balancing performance. The Simulink model uses Matlabs built in PID controller block with activated features for low-pass filtering on the derivative part

and anti-windup on the integral part. A snapshot of the Simulink model can be seen in Figure 5.1. Two primary noise models was used to test the controller performance. One of the noise models tested is white noise which it can handle relatively large amounts of without loosing stability. From the test run shown in Figure 6.13b the difference in the IMUs calculated angle and the ML models predicted value can be used to estimate the characteristics of the noise. The ML model limitation specifies a maximum angle of 20° so the values outside this region are omitted when designing the controller. This gives a mean error of 0.8255 degrees with a variance of 0.4072. Worth to note is that this noise is in comparison with the IMU which itself is affected by noise to a certain degree but it still provides a reference point. The simulated model corrects this without problems and stability remains intact. The second noise model tested is constant disturbance which can also be expected on the model. As described in Section 6.6 the runtime tests shows that the model has a bias predict angles closer to zero than the IMU and for non zero leaning this can be estimated as a constant disturbance. This is also handled well in simulation with the chosen controller parameters.

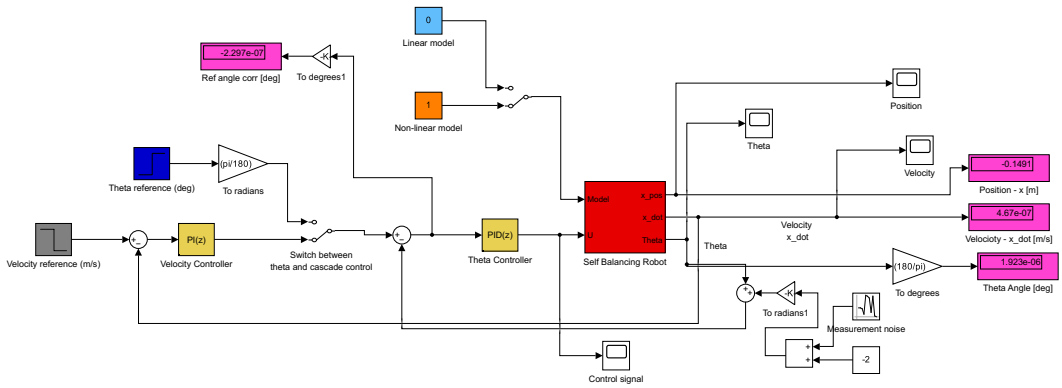


Figure 5.1: Simulink model of control system.

5.1.2 Implementation

The controller is implemented on the robot in the C programming language using the standard PID-algorithm. The code includes a simple implementation of a low-pass filter for the derivative part and a simple function to prevent over-saturation of the integral error. It takes parameter values as argument and then loops while reading current angle

values. An important difference for the implementation compared to simulation is that the real system will introduce non-linear traits that need to be compensated for. For example the motors will have a friction to overcome before they start spinning. This can either be compensated for by setting a higher gain in the controller or by adding the required voltage to overcome the friction to the calculated voltage outputted from the controller. Both methods were tried where the second option performed slightly better and is therefore used.

Since the ML model is written in Python while the controller runs in C on a separate program a shared memory is allocated from the Python script and a pointer to the memory address is read in each loop of the C program to update the current angle value. The benefit of this is that the controller can continuously loop without being locked by the machine learning algorithm, that can run on it's own individual core. This means that the syncing of the motors and the outer control loop can still adjust the reference angle. The downside is that the two CPU-cores are not synced and an angle update risks being delayed by one control loop. This is solved by having a higher sampling rate for the controller to reduce the consequences of this delay. The method proved to be efficient enough to run the controller without issues on the desired sampling rate.

6
Results

In the following sections, we present various results stemming from the state estimator and control system, including model training, performance distribution and real-time runs.

6.1 Data Collection

The collected dataset consists of 40,000 unique images of various scenery and places which are randomly rotated with a uniform probability distribution, such that their rotation lies in the range $[-30, 30]$. All of the images are RGB with a shape of 128x128x3. In Figure 6.1, the labels of 4000 randomly sampled datapoints are shown to be approximately uniformly distributed.

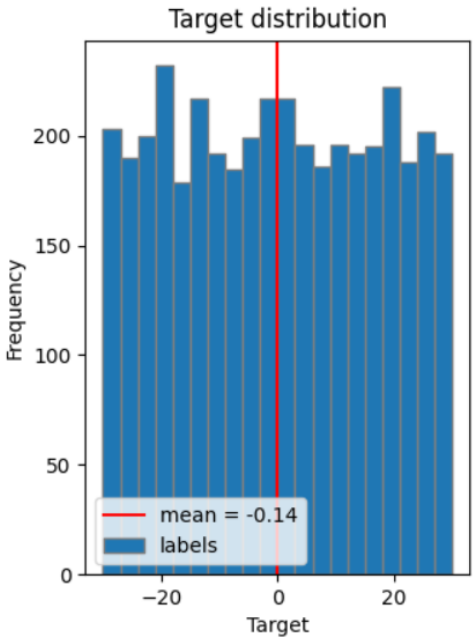


Figure 6.1: Distribution of 4,000 randomly sampled labels in the dataset.

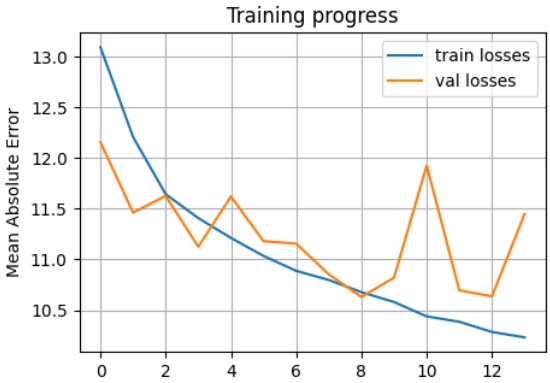
6.2 Training progress

In this section, training progresses are presented to illustrate convergence in the models.

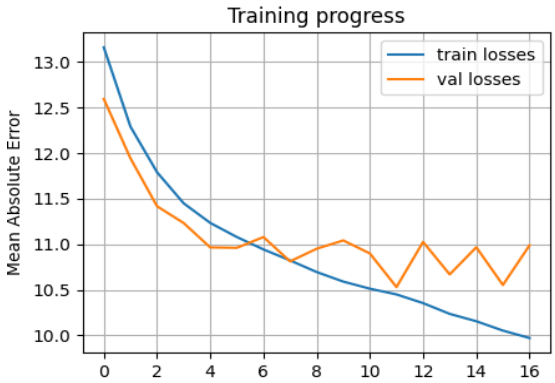
6.2.1 Linear Model

The validation losses of the LR models do not maintain a smoothly declining loss-profile. Instead, they oscillate heavily between almost every epoch while the training loss is strictly declining. In the 96×96 resolution and 64×64 resolution trainings, the training makes significant progress before converging at around the 10th epoch. The 128×128 resolution training keeps making slight progress every few epochs and thereby does not pass the patience criteria until 46 epochs. The smaller resolution training runs are locally optimal just above $L_{MAE} = 10.5$. The 128×128 resolution training run achieves a local optimum at $L_{MAE} = 9.99$.

(a) 128×128 resolution.



(b) 96×96 resolution.

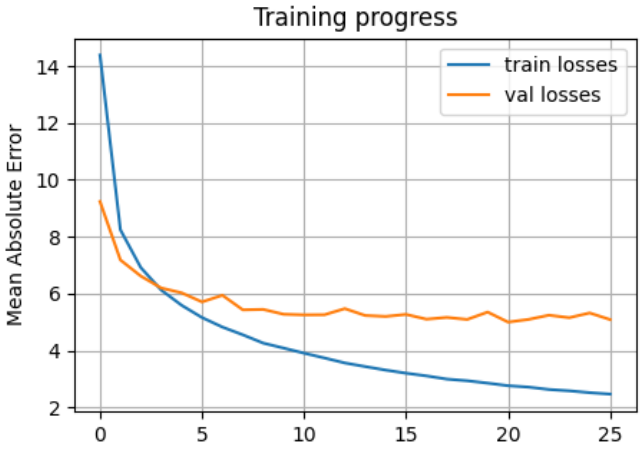


(c) 64×64 resolution.

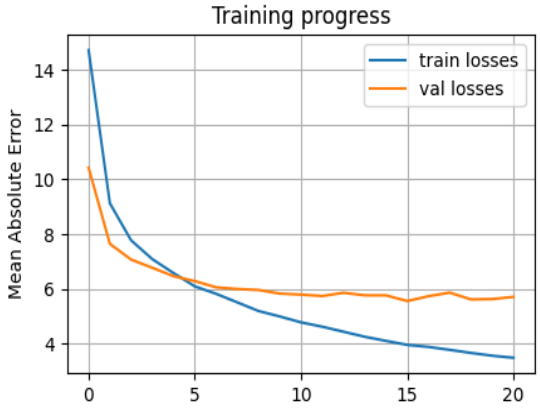
Figure 6.2: Training progress of LR-based state estimator. Patience set to 5 epochs on validation loss.

6.2.2 VGG16

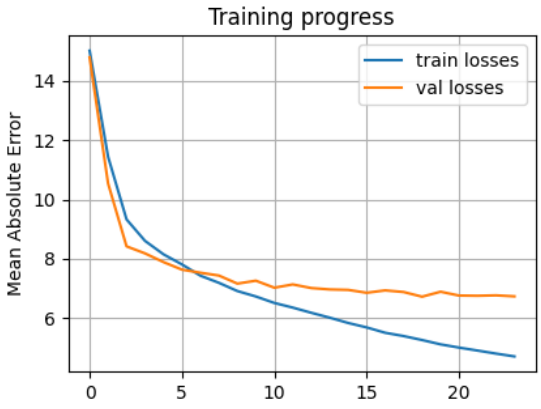
In Figure 6.3, the VGG16 model can be seen to drastically improve during the first 5 epochs in every resolution configuration. Thereafter, the validation loss begins to stagnate and improvement slows down, while training loss keeps strictly decreasing, signifying overfitting of the models.. Validation loss is at its' lowest on the 20th, 15th and 18th epoch at a loss of $L_{MAE} = 5.12$, $L_{MAE} = 5.64$ and $L_{MAE} = 6.39$ for 128×128, 96×96 and 64×64 resolution runs, respectively.



(a) 128×128 resolution.



(b) 96×96 resolution.

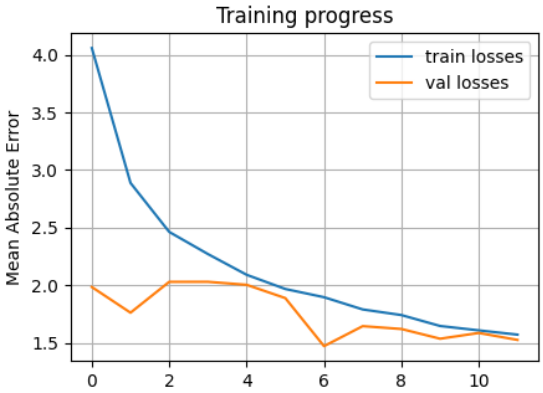


(c) 64×64 resolution.

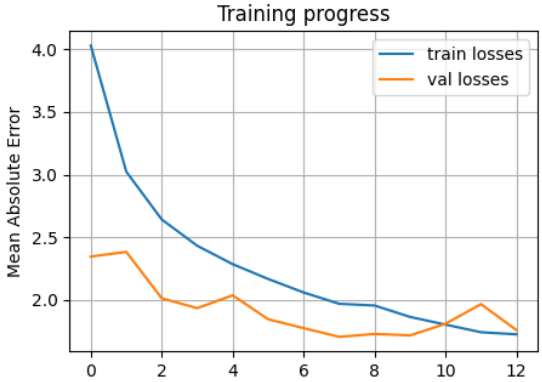
Figure 6.3: Training progress of VGG16-based state estimator as a function of elapsed epochs with varying image resolutions. Patience set to 5 epochs on validation loss.

6.2.3 MobileNetV2

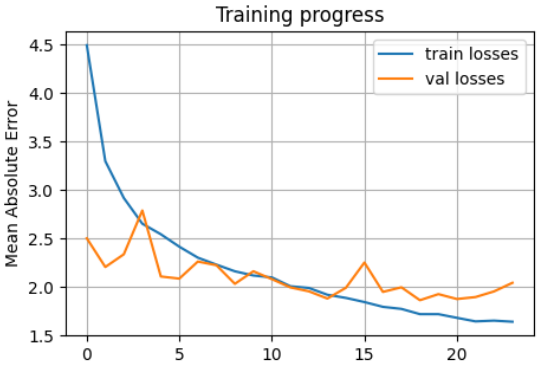
Training the vision-based state estimator built on MobileNetV2 converges quicker than remaining models, and reaches local optimum at the 6th, 7th and 18th epoch, as shown in Figure 6.4, whereafter they fail to further decrease and reach patience criteria. Validation losses can be observed to reach comparatively low values already in the first training epoch, after which the models become tuned and reach local minima at $L_{MAE} = 1.49$, $L_{MAE} = 1.71$ and $L_{MAE} = 1.88$ for resolutions in deacreasing order, respectively. The training loss can be observed to maintain a steeper decline over the late training-stage, which would likely overfit the model if training is continued. In general, the validation loss follows the training loss well and even slightly outperforms the training loss in the end, meaning that it performs as good on untrained images as on trained images. This implies good generalizability in the model, which is desirable for model robustness.



(a) 128 × 128 resolution.



(b) 96 × 96 resolution.



(c) 64 × 64 resolution.

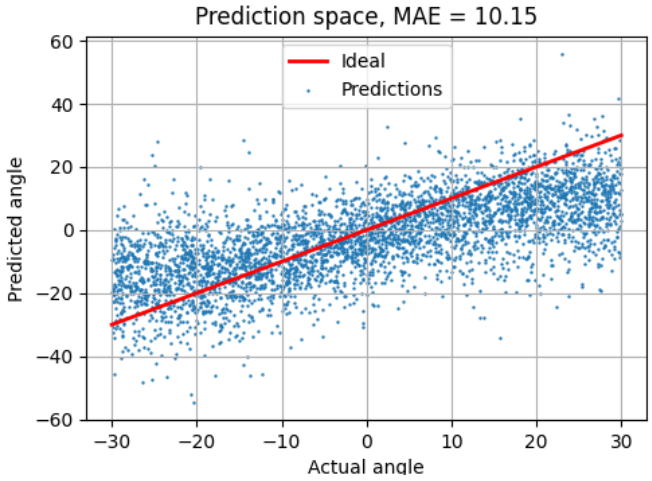
Figure 6.4: Training progress of MobileNetV2-based state estimator with varying image resolutions. Patience set to 5 epochs on validation loss.

6.3 Model evaluation

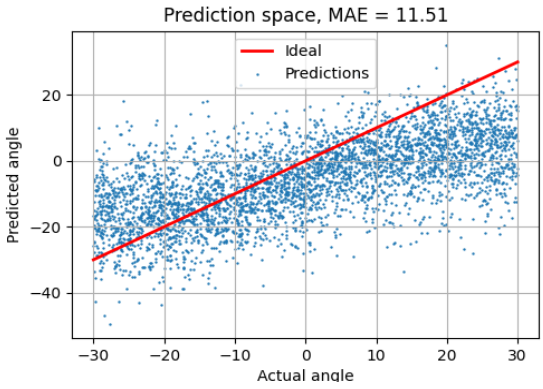
The accuracy of the camera-based state estimator, in the form of leaning angles, is evaluated by running the test set, i.e 10% of the dataset on which the models are not trained, through the trained model. Predictions which the model casts are then compared to the ground-truth labels.

6.3.1 Linear Regression

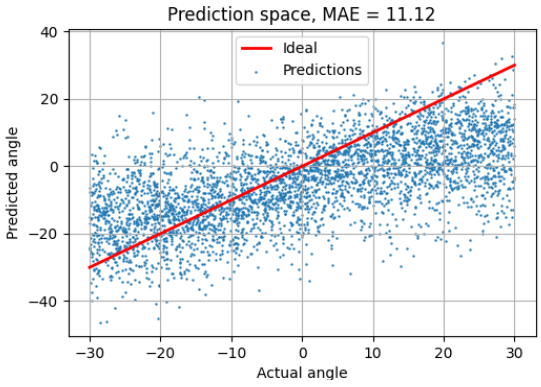
When running the LR-model on the test-data, it becomes evident that the model makes predictions with high, seemingly Gaussian noise, as shown in Figure 6.5. Furthermore, the estimated states are distributed towards lower magnitudes, both in the clockwise- and counterclockwise direction, hinting at semi-converted model weights. There is also a bias deviating from the equilibrium point in the prediction which is confirmed by the diverging error mean $\mu > 0$ in the probability distributions in Figure 6.6.



(a) 128 × 128 resolution.



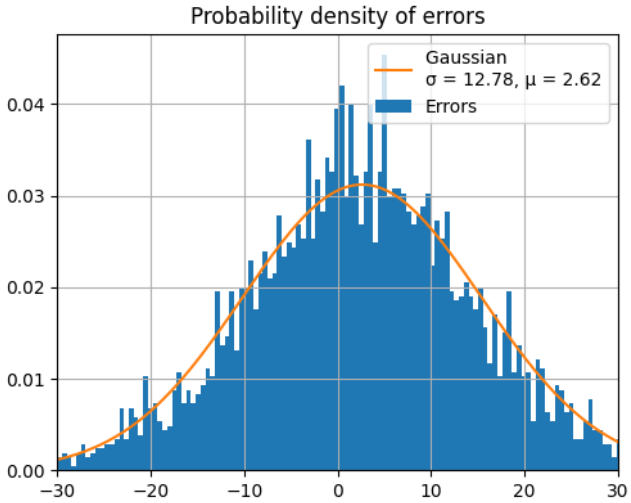
(b) 96 × 96 resolution.



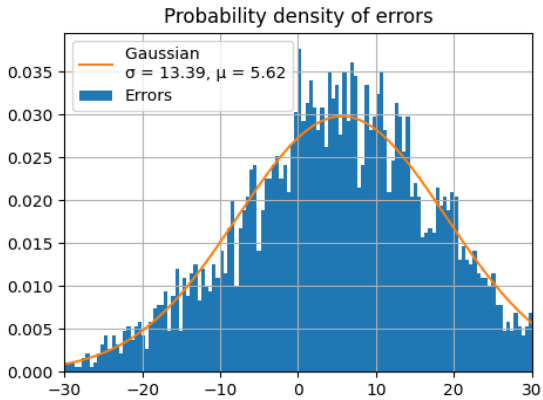
(c) 64 × 64 resolution.

Figure 6.5: Prediction space of LR-based state estimator with varying image resolutions. Each datapoint represents a predicted angle $\hat{y}$ and the corresponding ground-truth label y.

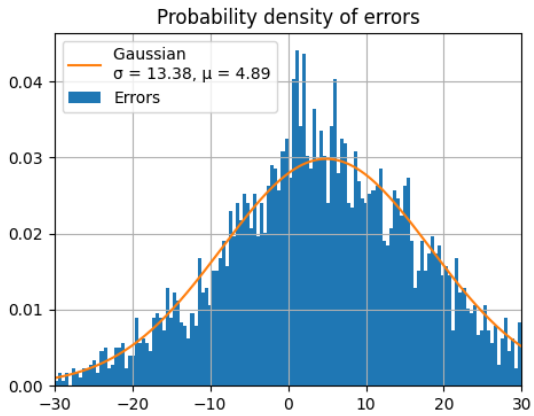
When analyzing the probability distribution of the prediction difference $y - \hat{y}$, as shown in Figure 6.6, it can be confirmed that the trained LR-based state estimators are biased with mean prediction differences $\mu = \overline{y - \hat{y}} = 2.62$ for the most accurate variant, Figure 6.6a. This means that the predicted state $\hat{y}$ on average is skewed towards the counterclockwise direction, relative to the ground-truth target y. The noise is also high, approaching a Gaussian distribution, with standard deviation $\sigma = 12.78$.



(a) 128 × 128 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.



(b) 96 × 96 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.

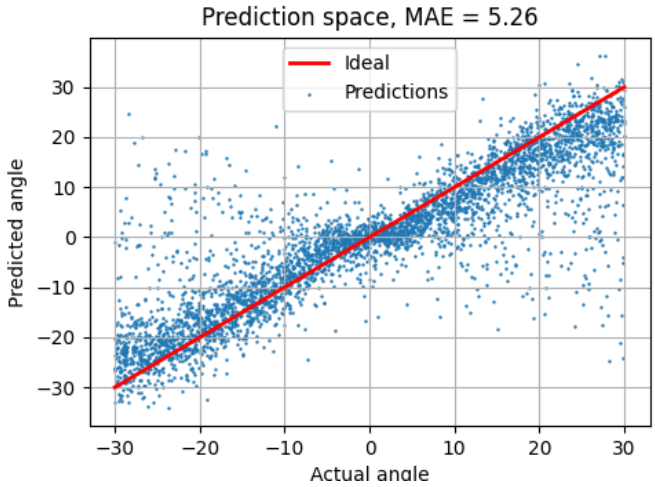


(c) 64 × 64 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.

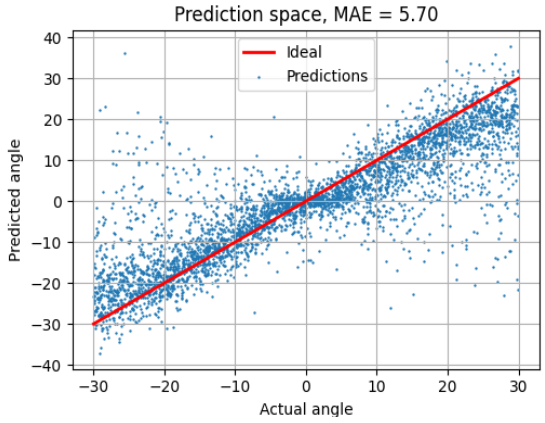
Figure 6.6: Probability density distributions on LR-based state estimator.

6.3.2 VGG16

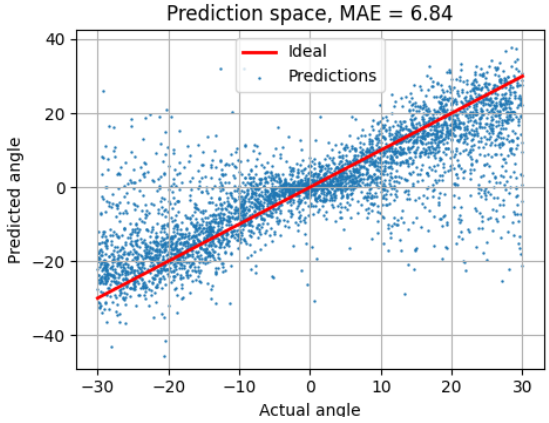
As shown in Figures 6.3, 6.4, the VGG16-based state estimator performs at a worse level than the MobileNetV2-based equivalent. This is also reflected in the prediction space on unseen images, shown in Figure 6.7. Most datapoints are distributed along the ideal, but with a high variance and noisy signal.



(a) 128 × 128 resolution.



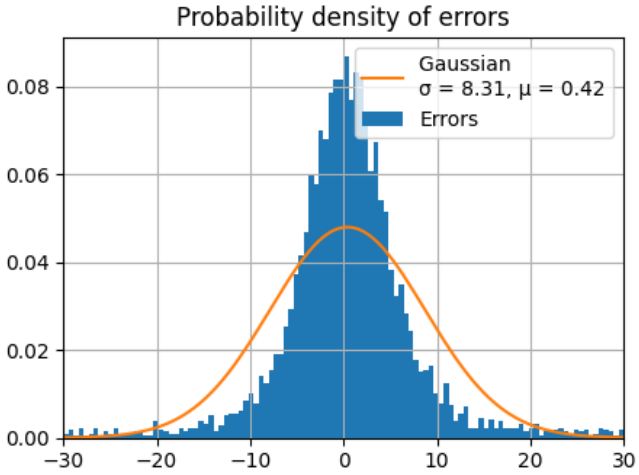
(b) 96 × 96 resolution.



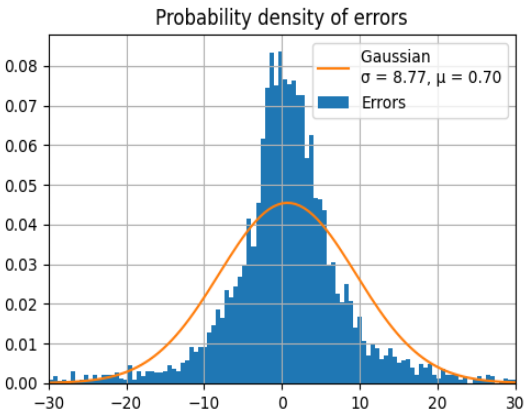
(c) 64 × 64 resolution.

Figure 6.7: Prediction space of VGG16-based state estimator with varying image resolutions. Each datapoint represents a predicted angle $\hat{y}$ and the corresponding ground-truth label y.

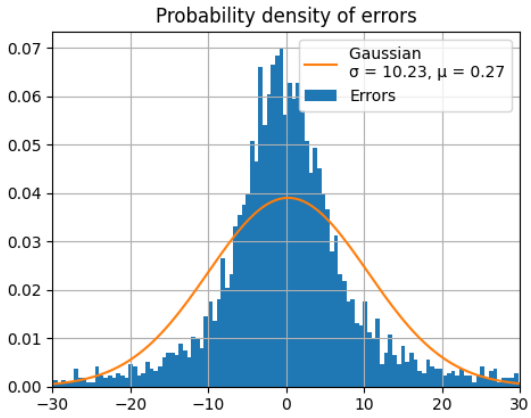
Compared to the LR-based state estimator, the VGG-based counterpart is significantly closer centered around the equilibrium point, with comparatively smaller bias and standard deviation in every image resolution. The average prediction difference lies at $\mu = \overline{y - \hat{y}} = 0.27$ for the least skewed variant (64 × 64), Figure 6.8c. The variant with the most narrow noise distribution is the 128 × 128 model, at $\sigma = 8.31$ and $\mu = 0.42$.



(a) 128 × 128 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.



(b) 96 × 96 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.

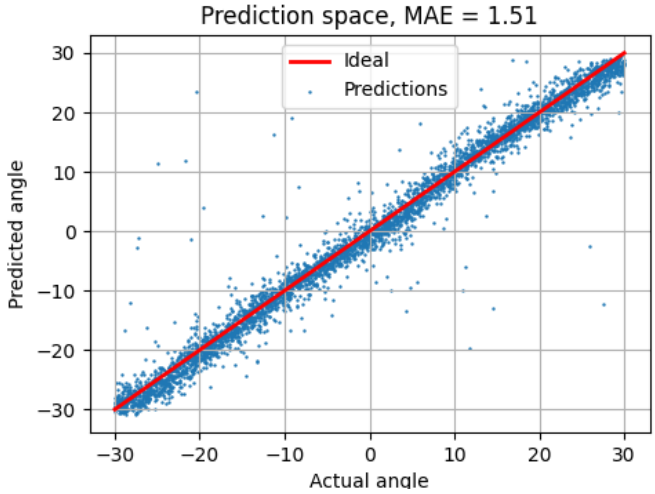


(c) 64 × 64 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.

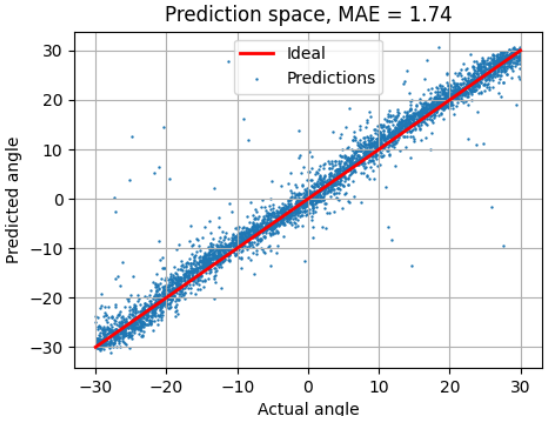
Figure 6.8: Probability density distributions on VGG16-based state estimator.

6.3.3 MobileNetV2

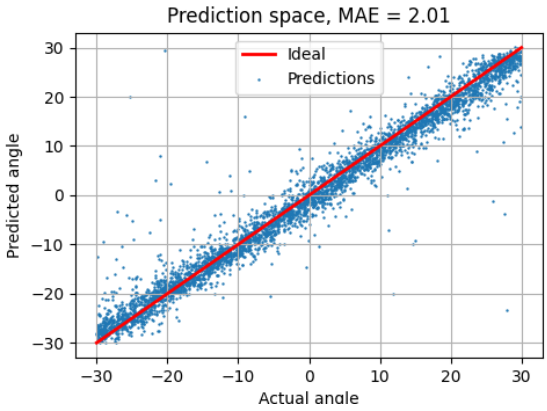
In Figure 6.9, the distribution of 4,000 predictions on three different image resolutions is shown in comparison to each respective ground-truth leaning angle. As can be seen, the MAE increases as in image resolution decreases. This can be explained by the loss of detail in the images. Furthermore, the predictions are for the most part scattered in close vicinity to the ideal, namely $y = x; x \in [-30, 30]$. This results in a Mean Average Error of between 1.46 and 2.00 depending on the resolution. There are some cases of outliers, however, that lie far outside of what could be considered acceptable performance. Similar to the VGG16-based state estimator, Figure 6.7, datapoints with ground-truth labels close to the interval boundaries are affected more strongly by noise than points close to the equilibrium point, $\theta = 0$. Thus, this implies a problem when the WIP enters states of steep leaning angles because measurements will be noisy and self-balancing becomes more difficult.



(a) 128 × 128 resolution.



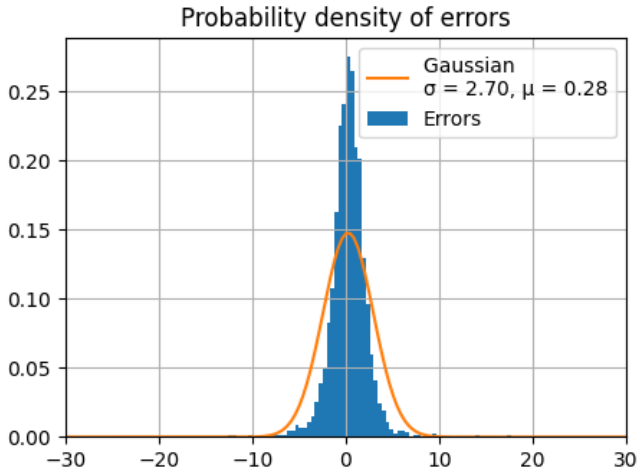
(b) 96 × 96 resolution.



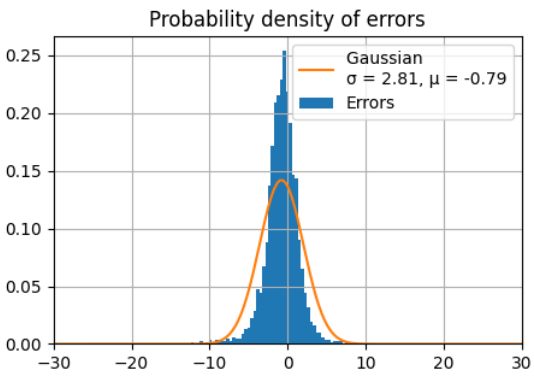
(c) 64 × 64 resolution.

Figure 6.9: Prediction space of MobileNetV2-based state estimator with varying image resolutions. Each datapoint represents a predicted angle $\hat{y}$ and the corresponding ground-truth label y.

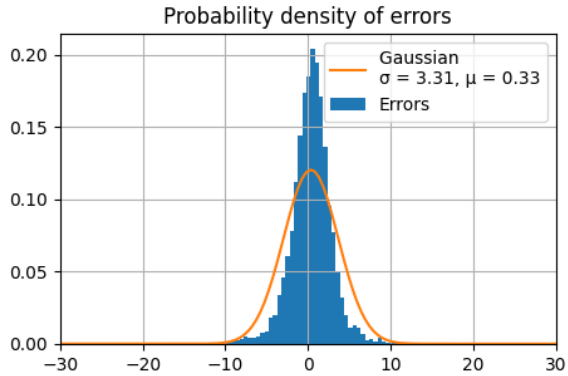
In Figure 6.10 the errors are considered as the difference between ground-truth target and prediction, $y - \hat{y}$. As might be expected given the prediction space in Figure 6.9, the error distribution appears symmetrical around the mean. The mean is in all cases positioned at $\mu > 0$, implying that the model is slightly biased to predict smaller angles, $\hat{y} < y$, i.e. counter-clockwise perturbations from the ground-truth targets. Furthermore, the standard deviation and noise increases as image resolution decreases, which implies less robust models with decreased image quality.



(a) 128 × 128 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.



(b) 96 × 96 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.



(c) 64 × 64 resolution. Horizontal axis corresponds to the prediction difference $y - \hat{y}$.

Figure 6.10: Probability density distributions on MobileNetV2-based state estimator.

6.4 Model comparison

Training has been carried out with a variety of hyperparameters, optimizers, and resolutions in order to compare the tradeoff between prediction accuracy and sampling frequency. The results are shown in Table 6.1.

Table 6.1: Model comparison on different training and test configurations

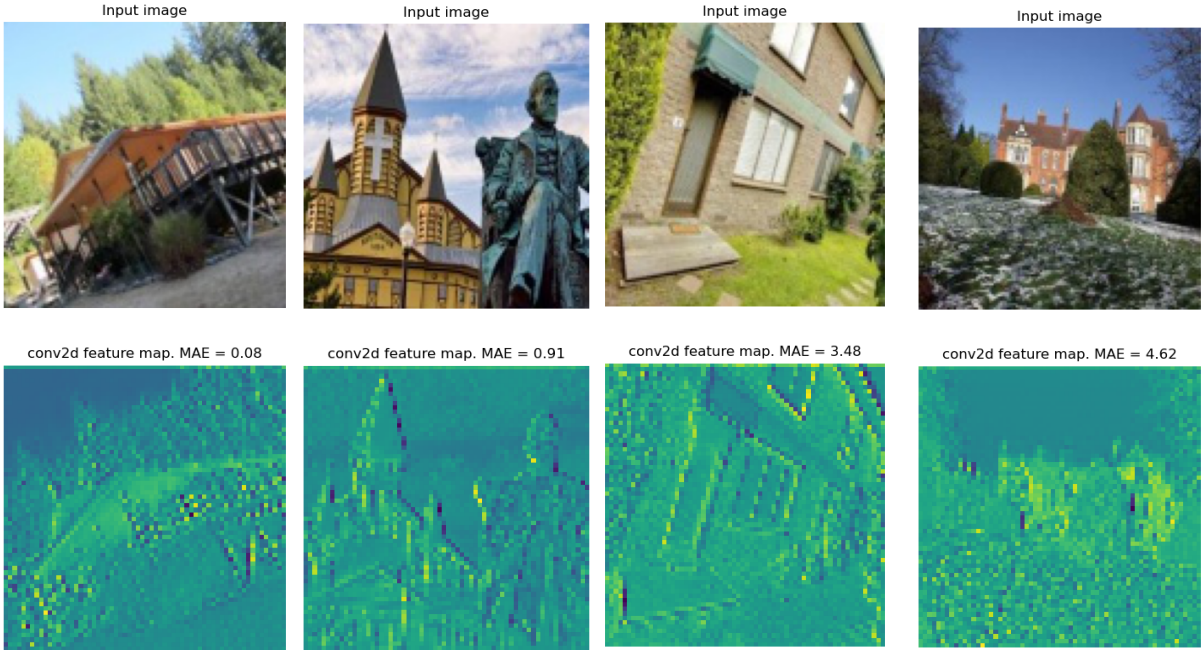
Model	Resolution	σ	μ	Optimizer	Learning Rate	Accuracy (MAE)
LR	64×64	13.38	4.89	SGD	$5 \cdot 10^{-3}$	11.12
	96×96	13.39	5.62			11.51
	128×128	12.78	2.62			10.51
VGG16	64×64	10.23	0.27	SGD	$5 \cdot 10^{-3}$	6.84
	96×96	8.77	0.70			5.70
	128×128	8.31	0.42			5.26
MobileNetV2	64×64	3.31	0.33	SGD	$5 \cdot 10^{-3}$	2.01
	96×96	2.81	-0.79			1.74
	128×128	2.70	0.28			1.51

As shown, the state estimator based on MobileNetV2 is the most accurate, while the LR-based model is the most inaccurate. Less expected is that the VGG16-based state estimator is significantly less accurate than the MobileNetV2-based equivalent, besides being a larger Neural Network with more trainable parameters. It should be noted however, that this might change with different training configurations or parameter settings.

From this comparison, it can be reasoned that the MobileNetV2-based state estimator would be the most qualified candidate for deployment on the robotic system.

6.5 Feature extraction

When analyzing the behaviour of a trained network, one can make use of feature maps and filters to display what parts of an image are attended to during a forward pass. In the following subsections, the feature maps and receptive fields of the various models are presented as well as the computed loss in leaning angles in the end of each respective forward pass.



(a) Very good accuracy **(b)** Good accuracy **(c)** Poor accuracy **(d)** Very poor accuracy

Figure 6.11: Input images and their corresponding feature maps of the first convolutional layer in the fine-tuned MobileNetV2 model. Brighter neurons reflect a higher activation in the network during a forward pass. MAE corresponds to the absolute prediction error between prediction and label mapped to each input image, respectively.

In Figure 6.11 the feature maps from the first convolutional layer of four randomly selected images and four different prediction errors are displayed. Since the input images are of sizes 128x128 and the convolutional layer has a 2-by-2 stride setting, the output arrays are of sizes 64x64, hence the decrease in image quality. Due to Neural Networks' common property of being hard to interpret, it is impossible to determine the causes of good or bad performance in the predictions, judging by only one convolutional layer, but what can be said is that the model seems to correctly identify lines and edges in the input images, such as the outlines of the church in Figure 6.11d.

6.6 Sensor comparison

In Figure 6.12 a comparison between the three sensors can be seen. The ML model follows the gyroscopes calculated angle even when the swift movements occurs after 5 seconds of runtime and does not suffer from the inaccuracy of the accelerometer, described further in Chapter 3.2, under quick direction changes.

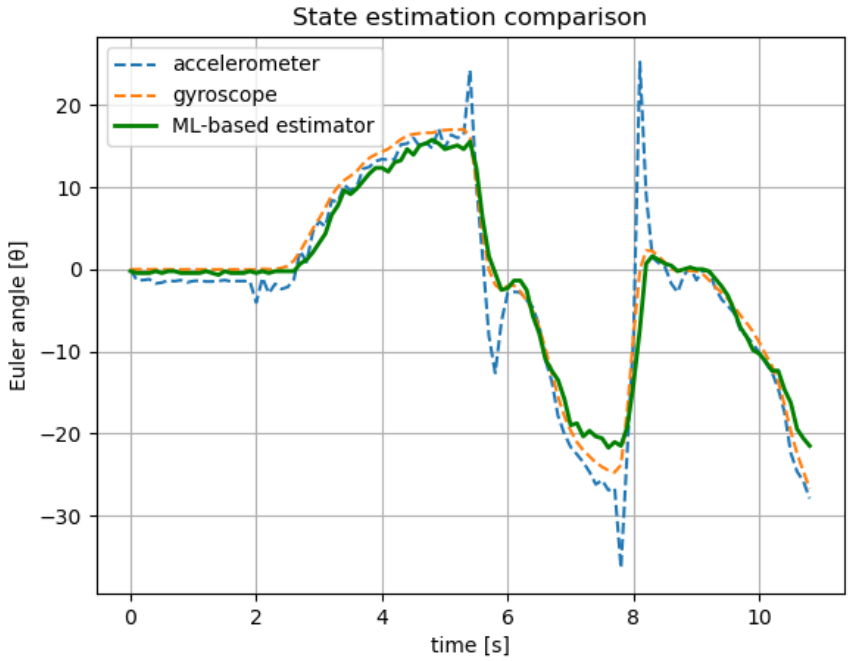
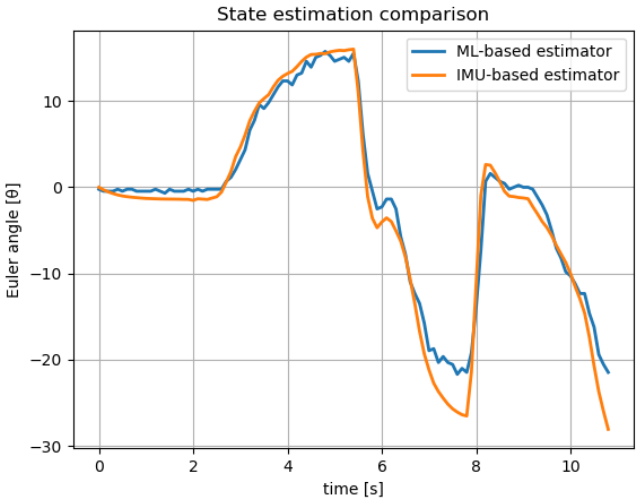


Figure 6.12: Comparison between sensors during testing. Testing is done by manually rotating the pendulum and observing the sensor readings.

When run in real time the gyroscope and accelerometer are fused with a complementary filter to provide a more accurate angle estimation during ideal conditions. In Figure 6.13b this is compared with the ML model. From the graph it can be seen that the camera follows angles up to around ±23° which is the limit of how far the model can estimate the angles with the current training data set. A slight bias in the ML angles where it tends to predict closer to zero can also be seen. The test run gave mean absolute error a 0.8255° and a standard deviation of 0.4072 compared to the IMU. The camera view for this test run is shown in Figure 6.13a and the vertical lines of the curtains proved to be beneficial for the ML based angle estimator. More test runs in different environments can be found in Appendix A.

(a) Point of view for test run

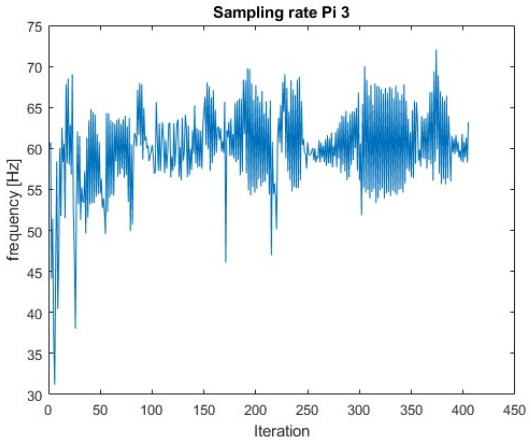


(b) Fused IMU and Camera comparison

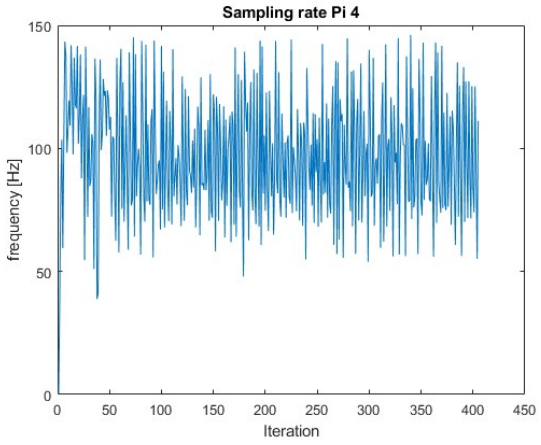
Figure 6.13: Comparison between IMU and ML-based state estimator built on MobileNetV2 during test run. The ML-based state estimator reacts with low latency and high accuracy, but makes conservative predictions compared to the IMU.

6.7 Real time performance

The requirements on the sampling rate during runtime for the ML model is as specified in Section 5.1 set to 100Hz. In Figure 6.14a it is shown that the model is able to run on with with a mean frequency of just over 60 Hz, but note that it contains a very high variance of the sampling rate. This means that the requirement for sampling time is not able to be satisfied with the used hardware of a Raspberry Pi 3. For reference the model was also tested on a Raspberry Pi 4 to see how improved hardware is able to perform. This is shown in Figure 6.14b and as seen the model is able to run with a mean frequency of 97 Hz in the test, which is close to the requirement and potentially enough to satisfy the controller requirements for stability. This hints that the angle prediction and robot performance might be improved more powerful hardware.



(a) Sampling rate for Raspberry Pi 3B



(b) Sampling rate for Raspberry Pi 4

Figure 6.14: Sampling frequencies for the deployed MobileNetV2 model on different processors.

While performing test runs of the robot system it was able to balance itself for a few seconds, without the need of an IMU, under certain close to ideal conditions. Unfortunately the limitations in precision and speed of the angle prediction ultimately lead to the robot not being able to balance itself reliably and robustly in most environments.

7
Conclusion

In this chapter, conclusions are drawn from the presented results. More specifically, we discuss the performances in the respective models, with regards to MAE, as well as causes of errors and suboptimalities. Lastly, further improvements are mentioned, such as alternative training methods, hardware improvements and system modelling approaches.

7.1 State estimator

The camera-based state estimators performed vastly different with regards to Mean Absolute Error, where the MobileNetV2-based state estimator reaches the highest accuracy out of the implemented models. Remaining models are deemed to be insufficiently accurate to qualify for deployment on the robotic system prototype. As shown in Figure 6.13b, the MobileNetV2-based state estimator reacts quickly to change in the leaning angle and the errors rarely differ more than a few degrees within the interval of interest. The main cause of tipping over, is because the deployed MobileNetV2-based state estimator has a relatively conservative prediction of the leaning angles' magnitudes, compared to the IMU.

As mentioned in Section 6.7, the state estimator performed well enough under ideal conditions to balance the robot for a few seconds which shows that the method of estimating angles from camera data might be feasible but more studies would have to be done on how improvements can be made to make it work reliably in all operating conditions. Limiting factors may be hardware, described more in Section 7.2 as well as training data.

While the camera-based angle estimator did not fulfil all the requirements for the balancing of a robot it still provided quite accurate predictions of leaning angles and can potentially be used, in its current state, for other purposes where a leaning angle is desired but without as strict accuracy and real-time demands.

7.2 Hardware

A limiting factor when performing inference on a Raspberry Pi 3, is that the USB 2.0 ports bottlenecks the transfer between Tensor Processing Unit and the CPU of the Raspberry Pi. This slows the inference algorithm down significantly and consequentially reduces sampling rates, as seen when comparing Figures 6.14a and 6.14b. An additional reason to why the sampling-rates differed so strongly in the tests is that the Raspberry Pi model 4 has larger Random Access Memory (RAM) and was running on a 64-bit operating system, which generally are faster than their 32-bit counterparts. To improve the stability for the robot it could theoretically help to add more weight to the top of the robot which could be enough to reach reliable balancing but it would need to be supplemented by a stronger battery to counteract the higher torque demand while correcting angles.

7.3 Future work

There are several potential improvements to be made on many aspects of the conducted solution. As mentioned in 7.2, the processing power has been a limiting factor in maintaining satisfactory sampling frequencies. This could be omitted by using a more powerful CPU or an interface which supports higher transfer speeds of image-data to the TPU, such as USB 3.0 ports instead of the installed USB 2.0 equivalent.

Furthermore, optimizations to the software could be done, both by choosing an alternative operating system, or by making improvements to the source-code that is executed during runtime. This includes utilizing the C/C++ API for model inference, instead of reading and writing from a shared memory address using two parallel running scripts.

With regards to the ML-models, the most imminent modification is the sophistication of the implemented training algorithms. This mostly consists of hyperparameter tuning. Performing grid-searches for the training parameters, where training is done on vastly more combinations of hyperparameters, is one of many ways to establish a result closer to the global optimum of a state estimator model [1].

To further enhance the ML models functionality it could be expanded to keep track of all three degrees of freedom described by the Euler angles. This could introduce many more use cases for the state estimator.

Bibliography

[1] Daniel Mesafint Belete and Manjaiah D Huchaiah. "Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results". In: *International Journal of Computers and Applications* 44.9 (2022), pp. 875–886.

[2] CGTN. *Controlling swarms of drones with machine vision*. URL: https://www.youtube.com/watch?v=MlFtHuXPbv4.

[3] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. "Supervised Learning". In: *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*. Ed. by Matthieu Cord and Pádraig Cunningham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–49. ISBN: 978-3-540-75171-7. DOI: 10.1007/978-3-540-75171-7_2. URL: https://doi.org/10.1007/978-3-540-75171-7_2.

[4] *DC Planetary Gear Brush Motor*. 638366. Robotzone. URL: https://www.electrokit.com/uploads/productfile/41016/motor-ds.pdf.

[5] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Google. *USB Accelerator | Coral*. URL: https://coral.ai/products/accelerator/.

[8] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[9] Katsuya Hyodo. *openvino2tensorflow*. URL: https://github.com/PINTO0309/openvino2tensorflow/pkgs/container/openvino2tensorflow.

[10] Elise K Jackson et al. "Introductory overview: Error metrics for hydrologic modelling–A review of common practices and an open source library to facilitate use and adoption". In: *Environmental Modelling & Software* 119 (2019), pp. 32–48.

[11] Kristian Lauszus. *A practical approach to Kalman filter and how to implement it*. URL: http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/.

[12] Mathworks. *Matlab System Identification toolbox documentation*. URL: https://se.mathworks.com/help/ident/.

[13] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[14] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. 6th ed. Hoboken, NJ, USA: John Wiley & Sons, 2021.

[15] Keiron O’Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[16] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].

[17] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[18] Raúl Rojas. “The Backpropagation Algorithm”. In: *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 149–182. ISBN: 978-3-642-61068-4. DOI: 10.1007/978-3-642-61068-4_7. URL: https://doi.org/10.1007/978-3-642-61068-4_7.

[19] Mark Sandler et al. “Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation”. In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

[20] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[21] T Sugata and C Yang. “Leaf App: Leaf recognition with deep convolutional neural networks”. In: *IOP Conference Series: Materials Science and Engineering* 273 (Nov. 2017), p. 012004. DOI: 10.1088/1757-899X/273/1/012004.

[22] Shiva Verma. *Rotated-Images*. https://www.kaggle.com/datasets/shivajbd/imagerotation. [Online; accessed 17-February-2023]. 2021.

[23] Cort J Willmott and Kenji Matsuura. “Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance”. In: *Climate research* 30.1 (2005), pp. 79–82.

[24] Bolei Zhou et al. “Learning Deep Features for Scene Recognition using Places Database”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper_files/paper/2014/file/3fe94a002317b5f9259f82690aeea4cd-Paper.pdf.

A

Appendix 1

A.1 Test runs

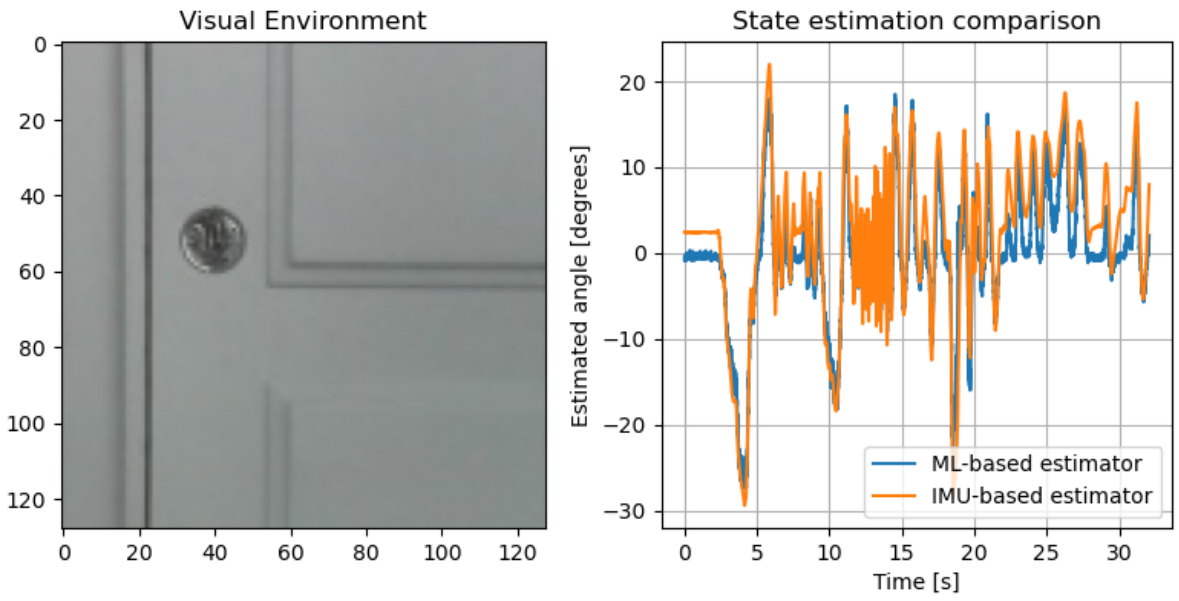


Figure A.1: Comparison for test run 2.

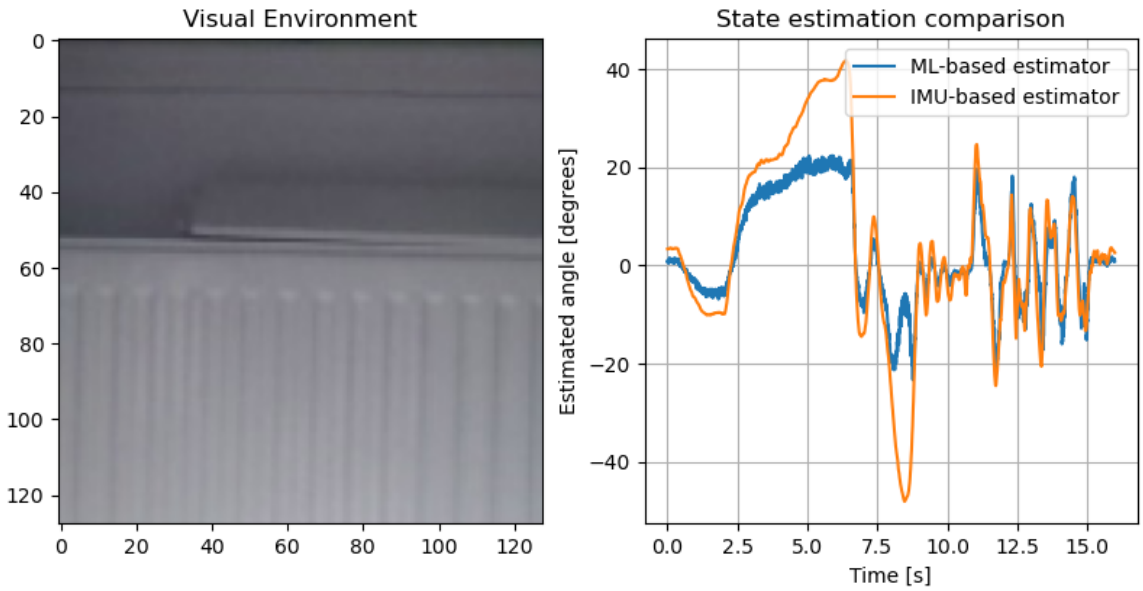


Figure A.2: Comparison for test run 3.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY