Østfold University College

Maneuver Autonomous Vehicles with Arm Gestures via a Smartphone

Generate steering signals from human poses

William Svea-Lochert

Master's thesis
Faculty of Computer Science
Østfold University Collage
Halden, Norway
May 26, 2022

Maneuver Autonomous Vehicles with Arm Gestures via a Smartphone

Generate steering signals from human poses

Master's Thesis

William Svea-Lochert

Faculty of Computer Sciences
Østfold University College
Halden
May 26, 2022

Abstract

Forsvarets Forskningsinstitutt (FFI) is currently looking into the development of a system for steering autonomous vehicles with arm gestures from a smartphone camera. A system like this would impact everyday life within the military by saving time and resources. This thesis continues the work from the previously written bachelor thesis Maneuver Autonomous Vehicles with Arm Gestures (BO20G38)[31] by looking into their findings for improving their prototype and implementing it into a smartphone application.

Developing a system like this required researching the topic of human pose estimation and Android development and implementation of machine learning models to Android applications. The system developed for this project serves as a prototype and foundational work for a fully developed system.

To see how a human pose estimation model could be developed, we researched previous work that has been conducted. After evaluating the research, the process of creating data labeling tools and augmentation pipelines was started to create custom keypoint datasets.

After creating datasets, architecture searches and hyperparameter tuning were conducted to find four different model architectures to try and find an optimal machine learning model architecture. The best-performing model is our MobileNetV2-based architecture. In testing with our testset 1, the model performs with a 17.930 mean overall pixel distance from ground truth on 15 keypoint values on 224x224 pixels images, outperforming MoveNet.SinglePose Thunder in testing.

To run the machine learning models, we created a smartphone application that opens an image stream and runs inference on the machine learning models to get a pose which is used to generate steering signals by looking at the angle between the hips, elbows, and wrists, to signal: forwards, reverse, left, right, and stop. The poses and current steering signal are drawn to the screen for the vehicle operator to get a visual cue of what the system is seeing. When running inference with our MobileNetV2-based model, the inference time is at 50 ms while running on a Huawei P30 Pro CPU.

The reliability testing of the system shows a massive improvement over the previous work done by BO20G38, where the system can generate correct steering signals at a much higher speed than previously. The research done throughout the project period confirms that a system for maneuvering vehicles with arm gestures from a smartphone is a realistic opportunity to pursue. This thesis lays the foundation for developing a fully operational system implementation and can be used as a reference point.

Acknowledgments

I want to thank Roland Olsson, my thesis advisor, for his technical expertise, guidance, and support. I would also like to thank Idar Dyrdal, my contact and advisor at FFI, for his invaluable assistance during this period and allow me to work on such an exciting project. I would also like to thank Simon Myhre for volunteering in the data-gathering process of this project. Finally, I would like to thank my family and friends for their patience and continuous support.

Prerequisites

This master thesis builds upon the previously written bachelor thesis Maneuver Autonomous Vehicles with Arm Gestures written by Ralf S. Sjøvold Leistad, Sander Lade Hellesø, Sverre Stafsegen Broen and William Svea-Lochert [31].

Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

In today's society, technology is quickly evolving, leading to a more modernized society as we live in today. By utilizing automation, one can be able to reduce human resources and at the same time have an increase in productivity which can benefit organizations and corporations[1]. Automation creates opportunities and benefits for the civilian population and governmental agencies like the military. It can increase productivity, consistency, as well as costs. Forsvarests forskningsinstitutt (FFI) is already developing and has already developed systems for autonomous vehicles that can benefit the military sector and society[2]. The boundaries of automation are constantly evolving and being pushed.

This thesis will primarily focus on the development stages of the steering/control system to maneuver an autonomous vehicle, such as an unmanned ground vehicle (UGV) or a Spot robot dog via a smartphone. The motivation behind the project is to make it easier for a soldier to keep their focus on the situation they are in and control the vehicle simultaneously. For the system to be viable, it has to run on a device like a smartphone as close to real-time as possible. The smartphone ought to capture an image stream, process and interpret the data with the help of a neural network and give the correct output to maneuver the vehicle.

FFI wants to continue with the research that was conducted in the previously written bachelor thesis Maneuver Autonomous Vehicles with Arm Gestures (B20G38)[31], by improving the system with the findings from that thesis as well as implementing it to run on a smartphone. The findings from B20G38 will be utilized as well as a new literature study will be conducted as new papers has been published and techniques and implementations will differ from B20G38.

This project proposes multiple machine learning models for human pose estimation developed through architecture searches and hyperparameter tuning that lays the foundation for further developing a fully functional maneuvering system for autonomous vehicles. In Addition to the machine learning models, we propose a method of creating datasets for human pose estimation with custom labeling tools and a data augmentation pipeline for generating large datasets from a small amount of base data. Finally, we have implemented the machine learning models into a smartphone application that runs inference with the models and generates steering signals by computing the angle between a set of joint locations.

[1] https://en.wikipedia.org/wiki/Automation#Advantages,_disadvantages,_and_limitations
[2] https://www.ffi.no/publikasjoner/arkiv/den-autonome-framtid

1.1 Background and motivation

Forsvarets Forskningsinstitutt (FFI)

FFI was established in 1946 after the second world war[3]. It was commonly recognized that technology played a big part in the war's outcome. The Norwegian army had to be rebuilt with modern weaponry. FFI was given three main goals; to help modernize the army, modernize the industry, and, lastly, to modernize the scientific environment in Norway.

Today FFI is the chief advisor for science and technology in the defense-related matter for the Ministry of Defence and the Norwegian Armed Force's military organization. The institute is also responsible for defense-related research in Norway. They consist of five research divisions: Defence systems, Strategic Analyses and Joint Systems, Sensor and Surveillance Systems, Total Defence, Innovation, and Industrial Development. As the official research institution for the Norwegian Defence Department, they have high-tech expertise, military, and political insight. They work in close conjunction with the Norwegian Armed Forces, The Norwegian Defence Department, other political sectors, national and international research institutions, and other countries through partnerships.

Motivation

As previously mentioned, FFI has been researching autonomous vehicle solutions for a long time. Previously, a bachelor thesis created a proof of concept for maneuvering autonomous vehicles with arm gestures[31]. FFI wants to continue the research and improve the systems and implement it on a smartphone. The reason for the project is to let soldiers focus on the situation they are currently in rather than having to give all their attention to the vehicle they are maneuvering.

1.1.1 Research questions

This study will attempt to create a system for maneuvering autonomous vehicles with arm gestures via a smartphone. The smartphone will utilize a deep-learning model to detect key-point on an image stream. From the key-point detection, we can calculate body joint angles, which can be used as steering signals for the vehicle.

My research question(RQ) consists of 3 parts. First I wish to shed some light on the performance of the neural networks that are going to be developed:

RQ 1 What type neural network architecture yields the best keypoint detection accuracy for human pose estimation?

RQ 1.1 *Which neural network architecture yields the best performance in terms of accuracy and speed?*

Secondly, I wish to look at the performance numbers of the architectures on a smartphone:

RQ 2 How close to real-time can a keypoint detection model run on a smartphone?

Thirdly, I wish to research:

RQ 3 What methods can be used for sending steering signals from a smartphone to a vehicle?

[3] https://www.ffi.no/en/about-ffi

1.1.2 Method

In order to answer the research question, an introduction to the required technologies and research areas; Human Pose Estimation(HPE), Machine learning(ML), and fundamental Android development. A literature review will have to be conducted, and the findings will be analyzed and summarized to understand the final system implementation better. Furthermore, the approach and methodology will be explained to help solve the problem at hand.

1.1.3 Deliverables

This study will have two main deliverables:

- Thesis paper.
- Smartphone application for generating vehicle steering input.

The project will also contain some helper code for creating our deep learning models. If time allows it, the application will be tested with a vehicle such as a Spot robot dog[4].

1.2 Report Outline

Chapter 2 begins with a description of the research topic of this thesis, followed by an in-depth project description. The following section gives an overview of the thesis research topic Human Pose Estimation, its different approaches, methods, and concepts. The next section looks at related works, and the following section looks into related technologies. Chapter 2 ends with an overview of the tools and hardware that can be used for this project.

Chapter 3 covers the planning of the project. The first section covers the project's initial plans: datasets, testsets, machine learning model, and Android applications development. The chapter concludes with how the proper framework and plans were selected.

Chapter 4 covers the implementation phase of the project. The first section explains how the data and testing -sets were created. The following section covers the development of the HPE models, which will contain the basic architectures of the models and their performance data in training, validation, and testing. The last section of chapter 4 describes the development and implementation of our HPE model into an Android application.

Chapter 5 presents and evaluates the results of chapter 4. It covers the initial tests and their results and reliability testing of the system.

Chapter 6 discusses the implementations and results, and the chapter concludes with suggestions for improving the system.

Chapter 7 concludes the project, where we will come to a conclusion of the project.

[4] https://www.bostondynamics.com/spot

Chapter 2

Analysis

This chapter starts with a brief explanation of the research topic and an in-depth project description in section 2.1. Section 2.2 gives an overview of the different approaches, methods, and concepts of HPE. Section 2.3 takes a look at related works. To finish the chapter section 2.5 Gives an overview of the tools that are relevant to this project.

2.1 Research topic: Human Pose Estimation

Human pose estimation (HPE) is the task of obtaining the posture of the human body from the given sensor inputs. One of the most common vision-based approaches is using cameras[29]. In simpler terms, HPE can be described as the task of localizing human joints[8]. With the recent rapid development in computer vision, we can go further than the classic tasks like bounding boxes and get a more detailed understanding of people in unconstrained environments[22].

2.1.1 An in-depth project description

As previously mentioned, FFI has been working on autonomous vehicles for quite some time. In 2020 they gave out the bachelor's assignment; Maneuver Autonomous Vehicles with Arm Gestures (BO20G38) [31], which laid the grounds for this project with proof of concept. For this assignment, FFI wants to take a step forward from the proof of concept that was previously created, implement the HPE software on a smartphone, and explore the improvements for future works discussed in BO20G38. An example of a vehicle that the system would be used on is shown in Figure 2.1, which is currently controlled by remote control.

Due to the vehicle being maneuvered with remote control, the soldier is bound to that task, making this person a nonactive team member. This problem can be solved with a HPE system, which makes it simpler to maneuver the vehicle, thus making the soldier a more active member of the team.

Figure 2.1: The tracked vehicle (UGV) FFI wants to use HPE for steering, image captured by: FFI, Espen Hofoss

2.2 HPE approaches, methods and concepts

HPE can be classified in to the following categories: generative, discriminative, top-down, bottom-up, regression-based, detection-based, one-/single- stage, multi-stage, skeleton-based model, contour-based model and finally volume-based model[29].

When one is talking about multi-person pose estimation, we can classify the approaches into; top-down or bottom up[22], [29].

Top-down

The top-down approach begins with a rough localization and identification of a person or persons instance by using techniques such as bounding box object detection (high-level abstraction), which in turn is followed by a single-person pose estimation[22], [29]. This pose estimation process is then repeated for each person that was localized in the first stage.

Bottom-up

Bottom-up differs from top-down in the way that it will start by predicting all joints/body parts for each person that is in the sensor input[29]. It will group them using either a human body model fitting or some other algorithm.

Multi-stage

Multi-stage is a method that tries to produce a increasingly refined estimation[26]. They can be both bottom-up (2.2) and top-down(2.2).

Multi-stage with bottom-up starts like the typical bottom-up approach with predicting the individual joints and then stitching them together [26]. The bottom-up multi-stage approach can be used as a 3D HPE method in the way that it starts with predicting the joint location on the 2D surface as a typical bottom-up approach, but where it differs is

that it then extends them into 3-dimensional space[29]. The results of this can, for example, be heat maps[26].

The top-down multi-stage approach starts with the location of the persons of interest, using classic methods like high-level abstraction (bounding box object detection)[22], [26], [29]. Then it utilizes a single pose estimation to predict the keypoint locations. The authors of Rethinking on Multi-Stage Networks for Human Pose Estimation propose a solution where they use the top-down approach [26]. The first stage utilizes what they call an off-the-shelf human detector. In the second step, a Multi-Stage Pose Network is applied to all the human detection to produce a pose result.

Single-/one- stage

A single-stage approach tends to use an end-to-end network for mapping the input image for human poses[29]. According to Rethink[26] all single-stage methods are top-down approaches.

Depending on the problem formulation, deep learning HPE methods can be put into regression-based, or detection-based approaches[29].

Regression-based

Regression-based HPE is when mapping is done directly onto the input to coordinates that match the position of the body joints[29].

Classification/Detection -based

A detection-based HPE approach looks at joints/body parts as the detection targets[29]. The targets are based on image patches and heatmaps of joint locations.

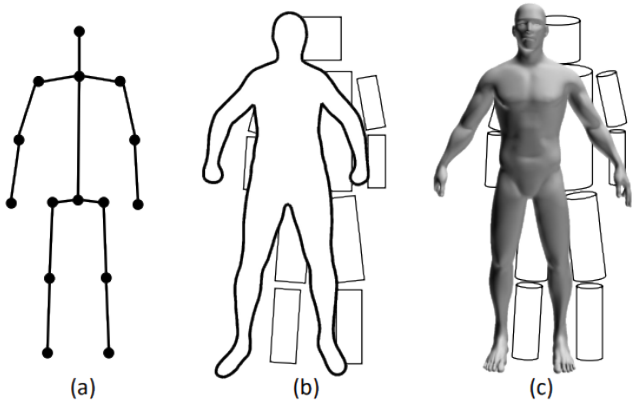


Figure 2.2: Common human body models (a) skeleton-based model; (b) contour-based models; (c) volume-based models. Obtained from Monocular Human Pose Estimation: A Survey of Deep Learning-based Methods[29]

Skeleton-based model

Skeleton-based model or kinematic model is represented as a set of joint locations with a corresponding limb orientation that follows the human body skeletal structure[29]. Another way to describe a skeleton-based model is to compare it to a graph where all vertices are specifying joints or prior connections of joints within a skeleton structure[2]. Commonly used human body models can be seen in Figure 2.2.

Contour-based model

Contoure-based models was a highly used method in early HPE methods[29]. It uses the rough width and contour information of limbs and torso.

Volume-based model

In general, with volume-based models, body shapes and poses are represented in 3D with meshes or geometric shapes[29]. Modern volume-based models are typically represented as a mesh generally captured with 3D scans.

Generative and Discriminative

What mainly sets generative and discriminative methods apart is whether the method uses a human body model[29]. A discriminative method takes the mapping from an input source and learns directly from it, and this is called learning-based. Alternatively, they can learn by searching in pre-existing examples without a human body model.

2.3 Related work

The following section gives an overview of previous work that has been done that relates closely to this project.

2.3.1 PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model

The authors of PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model[22], propose a bottom-up (section 2.2) approach. Their method starts with a set of keypoint predictions for all people found in the image, which is done in a fully convolutional way. They also propose a method of greatly improving accuracy for long-range predictions with a novel recurrent scheme; adding to this, they managed to predict the relative displacement between each keypoint pair. In addition, their keypoint predictions approach can create a dense instance segmentation mask for all the detected people in the input image.

To train their model, they used a subset of the standard COCO keypoint 2017 training dataset (64115 images). For validation, they used the COCO keypoint 2017 validation set (5000 images). The datasets contain images of people, single or multiple, with 12 body keypoints, five facial keypoints, and a segmentation mask.

The authors utilized the standard COCO keypoint task and COCO instance segmentation for the person class to evaluate their experiment. To set the initial weights for their model, they utilized ImageNet.

Their reported experimental results are with models using either Resnet-101 or ResNet-152 CNN as the backbone of the model, and these were pre-trained on the ImageNet classification task. They also removed the last layer of the ImageNet classification layer to then add a 1x1 convolutional layer for each of their model-specific layers.

The reported results of the experiment show that their model is outperforming other bottom-up and top-down methods. However, their average precision of 0.687 is lacking behind the COCO 2017 keypoint challenge winners. They obtained a COCO test-dev keypoint average precision of 0.665 using single-scale inference.

The authors concluded that their models address the problem of person detection, HPE, and instance segmentation.

2.3.2 A Top-down Approach of articulate Human Pose Estimation and Tracking

The authors of A Top-down Approach of articulate Human Pose Estimation and Tracking propose a top-down (section 2.2) approach to the HPE problem[21]. Their approach starts with performing a human candidate detection, followed by a single-person pose estimation, and finally, a pose tracking step by step.

To perform the human candidate detection they adopted a state-of-the-art object detector that was trained with ImageNet[1] and COCO datasets[2], more specifically they used pre-trained models from deformable ConvNets[17].

The single pose estimation module the authors used was an adoption of Cascade Pyramid Networks that was somewhat modified. A cascade Pyramid network is based on the Stacked Hourglass approach 2.3.4 where instead of eight stacked hourglasses, it is reduced to two[20]. The model was trained with a combined dataset of PoseTrack 2018[3] and a COCO dataset. They also describe their implementation performs a non-maximum suppression[4] in the detection phase on the bounding boxes that were set in the first module. Then pose estimation is performed on all the possible person detections. For all the candidates, they perform a post-process on the heatmaps from the prediction with cross-heatmap pose non-maximum suppression to get a more accurate keypoint position.

The last module is their Pose tracking module. The authors chose to use a flow-based pose tracker[24]. By associating poses that indicate that the same person is still in the frame, pose flows are built. The tracking process is started in the first frame, where a human candidate is detected, and each detection is assigned an ID. To prevent the IDs of people who have left the frame, the IDs are only kept for a limited number of frames; eventually, they are discarded.

Their approach resulted in a total Average precision of 69.4 on single-frame pose estimation on the PoseTrack 2018 test set.

[1] https://en.wikipedia.org/wiki/ImageNet
[2] https://cocodataset.org/#home
[3] https://posetrack.net/
[4] https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

2.3.3 DeepPose: Human Pose Estimation via Deep Neural Networks

Alexander Toshev and his colleague Christian Szegedy propose a holistic view of the human pose estimation method that is based on deep neural networks[8]. Their pose estimation problem is formulated to be a joint regression problem and shows how to correctly cast it in deep neural network settings. To find each location of the body joints in a person of interest, they utilize a 7-layered generic convolutional deep neural network. Furthermore, they explain two advantages of their formulation/approach; a deep neural network can see the full context of each body joint, meaning that each regressor uses a full-size image as a signal. Secondly, the approach is much simpler to formulate than other methods such as graphical models, as there is no need to explicitly design representations and detectors, meaning no need to design a distinct model topology and interactions between each joint. Lastly, the authors propose a cascade of deep neural network-based pose predictors and explain that it allows for increased precision of joint localization.

The authors conclude their paper by stating that their approach can get state-of-the-art or better results on several challenging academic datasets and further explains that they are showing that a generic convolutional neural network that was designed for classification tasks can be applied to localization tasks.

2.3.4 Stacked Hourglass Networks for Human Pose Estimation

The authors of Stacked Hourglass Networks for Human Pose Estimation present a novel convolutional neural network architecture to tackle the task of human pose estimation[15]. Their network design approach for pose estimation is made up of several stacked hourglass modules, and by stacking them this way allows for repeated bottom-up, top-down(bottom-up and top-down explained in section 2.2) inference. The network can capture and consolidate information across all scales of an image. The hourglass design refers to the visualization of the pooling and subsequent up-sampling steps used to get their network's final output. Like other convolutional networks that give pixel-wise outputs, the stacked hourglass network pools down to a meager resolution and afterward up-samples and then combines features across multiple resolutions. The hourglass network stands out from previous approaches primarily in the symmetric topology.

The network expands from a single hourglass by placing multiple hourglass modules together end-to-end. As previously mentioned this allows for repeated bottom-up(section 2.2), top-down(section 2.2) inference across all the scales of an image. The authors used Percentage of Correct Keypoints (PCK) to evaluate their results, which reports the percentage of keypoint detection's within a normalized distance of the ground truth keypoint. The authors state that the network can show a significant improvement on the state-of-the-art for two standard pose estimation benchmarks such as MPII[6] where they achieve over 2% average accuracy improvements on all body joints.

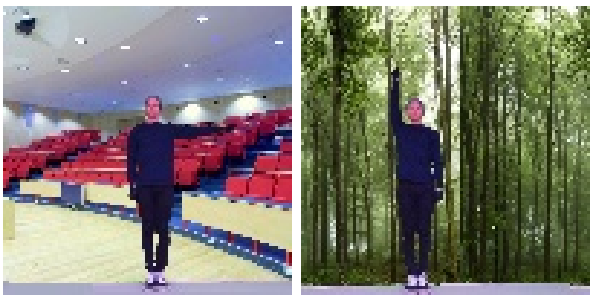
To conclude their paper, the authors state that they have demonstrated the stacked hourglass network design's effectiveness and robustness.

2.3.5 Deep High-Resolution Representation Learning for Human Pose Estimation

Ke Sun and colleagues explain that most human pose estimation methods use a high-to-low resolution network, meaning that the methods get high-resolution representations from a low-resolution representation[28]. The paper's authors propose a different approach; their solution HRNet keeps the high-resolution representation throughout the whole process. They start with a high-resolution subnetwork as the first stage. Step by step, they add high to low-resolution subnetworks for more stages and connect the multi-resolution subnetworks in parallel. The authors conclude that they have presented a high-resolution network for human pose estimation that predicts accurate and spatially precise keypoint heatmaps. They argue that the success comes from the following two reasons; By maintaining the high resolution through the whole process, there is no need for recovering the high resolution. Secondly, fusing multi-resolution representations repeatedly gives reliable representations with a high resolution.

2.3.6 Maneuver autonomous vehicles with arm gestures

Ralf S. Sjøvold Leistad and colleagues paved the way for this project with their proof of concept[31]. The project's goal was to make a proof of concept for maneuvering autonomous vehicles with arm gestures. The authors' approach was a classic image classification task with a convolutional neural network (CNN). To begin with, the group collected small datasets but quickly discovered that they needed more training data. Their model was trained on 3 937 800 100x100 pixel images spanning over six categories and tested with testsets totaling 984 450 images. The group collected the images, and they utilized a greenscreen and data augmentation to scale the production of their datasets. A base set of images had some basic transformations done to them and changed their background to create the datasets. Examples of their data can be seen in Figure 2.3. The transformations done to the data move the person around in the image and scale them up and down.



(a) Left (b) Forwards

Figure 2.3: Example data from BO20G38. Obtained from Maneuver autonomous vehicles with arm gestures

As previously mentioned, the model that the authors used was a CNN model. The model went through 4 iterations, and they also experimented with a dense neural network, eventually reaching a final CNN architecture. When testing with their test data, the model performed remarkably with an average accuracy of 99.9%. Later in their thesis, they evaluated the datasets that were created, and the evaluation resulted in finding flaws that could have implications later. One of their main concerns was the lack of diversity. Only one person was used for the data, which led to problems when another person tried to use the system. An even bigger issue was the strictness of the data, with how precise one has to be when handling the different steering signals. Both issues were later found to be a problem in real-world reliability testing leading to the author's conclusion that a proof of concept was established, yet to further develop the system, an HPE model or a combination of model implementations such as Guanghan Ning and colleagues[21] proposes should yield superior results compared to the Leistad and colleagues implementation.

2.3.7 Related works summary

The research that has been conducted shows excellent work from many talented developers. Alexander Toshev and his colleague propose a solution that relates to the approach we are using in this project by utilizing a generic convolutional deep neural network[8]. At the same time, the approach of others, such as the authors of Stacked Hourglass Networks for Human Pose Estimation where they, are stacking modules. Ralf S. Sjøvold Leistad and his colleagues made a simple CNN approach to classify poses as steering signals for a vehicle. However, they also introduced an efficient way of mass-producing custom datasets. Their greenscreen technique is one of the methods that also will be utilized in this project. The other articles explored for this project also give good explanations of the human pose estimation research field and the different methods and approaches that can be used in a project such as this one.

2.4 Related Technologies

This section gives an overview of some of the related technologies to this project.

2.4.1 Machine learning

The study of computer algorithms that can learn and improve automatically from data and experience is called machine learning or ML for short [5]. ML algorithms build models; these models are made to make predictions and decisions. For an ML model to make a good prediction, it has to be trained with sample data, also known as training data. Machine learning is commonly considered a part of artificial intelligence.

Machine learning has several approaches: Supervised learning, Unsupervised learning, Semi-supervised learning, reinforcement learning, dimensionality reduction as well as others. The two most commonly discussed methods are supervised and unsupervised learning. Supervised learning is when an input pattern has a known label[4]. A feature can, for example, be continuous, categorical, or binary, such as an image of a dog can be labeled as a dog while an image of a cat is labeled as a cat. Unsupervised learning is when a given feature is unlabeled.

Neural networks

Artificial neural networks, also called neural networks, are classified as a model approach within machine learning and a central part of deep learning algorithms[6][5]. Both the name and structure of neural networks are inspired by biological brains like the human brain. Neural networks have drawn inspiration from our understanding of the brain; it is essential to emphasize that neural networks are not models of our brain[16]. Neural networks are designed to mimic how a biological neuron in a brain sends signals to another neuron. In neural networks these are called artificial neurons[7]. A neural network is built up of these artificial neurons. They receive a signal, then process it and send it to adjacent connected neurons. The connection between these neurons is referred to as an edge. Each of the

[5] https://en.wikipedia.org/wiki/Machine_learning
[6] https://www.ibm.com/cloud/learn/neural-networks
[7] https://en.wikipedia.org/wiki/Artificial_neural_network

neurons and edges typically has a weight associated with them. The weights are being adjusted during the network training; by doing this, the accuracy of the network increases.

Each signal between the connected neurons is an actual number. However, the output of a neuron is the calculated sum of a non-linear function that has been predetermined, commonly referred to as an activation function. In a conventional neural network, the neurons are collected into different layers. The signal starts in the first layer of the network, commonly known as the input layer. The signal crosses through the network's hidden layers from the input layer until it reaches the output layer. The signal traveling through the network can be passed through in many different ways, depending on the architecture design of the network. A simple example of a neural network can be seen in Figure 2.4

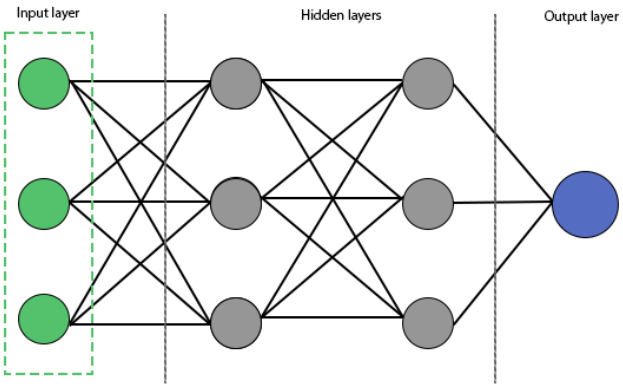


Figure 2.4: Common structure for neural networks.

Deep learning

Deep learning is classified as a family member of machine learning methods that utilize neural networks[8]. There are several different deep-learning architectures, such as Convolutional Neural networks which are covered in section 2.4.1. The meaning of "deep" in deep learning refers to the idea of successive/continuous layers[16]. The number of layers in a deep learning model is referred to as the depth of the model. A modern model can consist of tens and, in some cases, even hundreds of layers. All the layers in a deep learning model learn automatically from exposure to training data. Deep learning differs from other machine learning approaches with the number of layers that are being trained. In contrast, other approaches concentrate on training one or two layers.

Convolutional Neural networks (CNN or ConvNet)

Convolutional neural networks, or CNN for short, are a class of deep learning neural networks[9]. ConvNets process data in a grid-like fashion[13]. CNNs are most commonly used to process visual input such as video or image data. One of the main advantages with ConvNets is that they can learn features from the input, rather than engineers/developers

[8] https://en.wikipedia.org/wiki/Deep_learning
[9] https://en.wikipedia.org/wiki/Deep_learning

having to hand-make these filters/features[10]. A significant difference between ConvNets and Dense networks is that convolutional layers learn local patterns, while dense networks learn global patterns[16]. An example would be an image of a car; a CNN would, for example, see the mirrors, while a dense network would have to look at all the pixels of the car. An example of this can be seen in Figure 2.5.

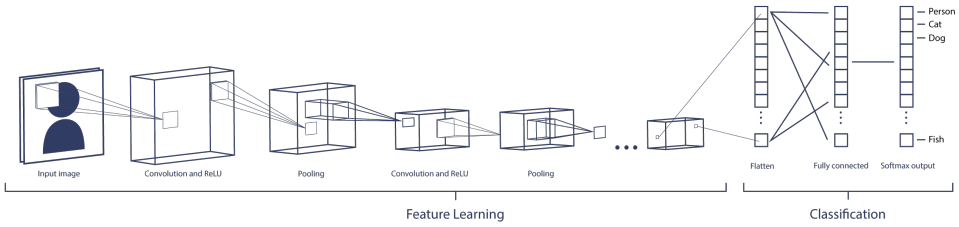


Figure 2.5: Common structure for CNN.

CNNs use the same structure as many other typical neural networks, with an input layer followed by hidden layers and concluding with an output layer. A visual example of a typical neural network structure can be seen in Figure 2.4. The hidden layers of a ConvNet generally consist of convolutional layers, which are normally used with an activation function such as ReLU, pooling, batch normalization, dropout, and dense -layers.

SeparableConv2D layers

A separableConv2D layer is a type of convolutional layer that can used in TensorFlow(subsection 2.5.2). Where a traditional convolutional layer uses kernels that convolve over the height and width of the image[11], a separableConv2D layer firstly performs a depth-wise spatial convolution that works on each input channel individually, and a point-wise convolution then follows this. The point-wise convolution mixes the consequential output channels[12].

2.4.2 Pooling layers

A significant part of convolutional neural networks is pooling. Pooling is described as a non-linear form of down-sampling, where max pooling is the most common. Max pooling works so that the input image is partitioned into groups of rectangles. For each sub-region, the maximum is outputted. An example of max pooling with a 2x2 filter and stride set to 2, see Figure 2.6.

[10] https://en.wikipedia.org/wiki/Convolutional_neural_network
[11] https://en.wikipedia.org/wiki/Convolutional_neural_network#Convolutional_layers
[12] https://www.tensorflow.org/api_docs/python/tf/keras/layers/SeparableConv2D

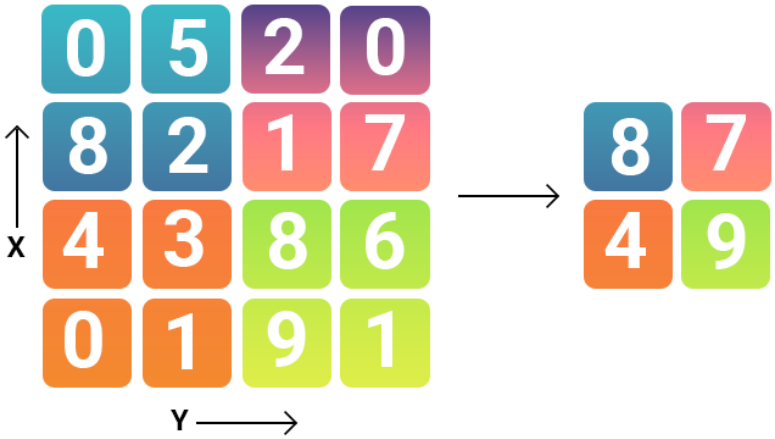


Figure 2.6: A simple example of max pooling with a filter of 2x2 and stride set to 2.

Residual blocks

As earlier mentioned in Section 2.4.1 each layer feeds data to the next layer until it reaches the output layer. Networks like ResNet[34] introduce the concept of residual blocks. In short, a residual block feeds the data into the next layer as well as directly into layers further down in the network; this is called skip connections or shortcuts[23]. An example of a residual neural network can be seen in Figure 2.7.

Figure 2.7: A simple example of residual neural network where layer 2 is skipped by layer 1.

The main reasons why we use residual blocks are the vanishing gradient problem or the degradation problem[34].

In each epoch of training a neural network, all the weights in the network are updated[35]. The updated value is calculated proportionally from the partial derivative of the used error function relative to the current weight of that given epoch. The vanishing gradient problem

occurs when the gradient becomes too small, meaning that the value will be so small that the weights are prevented from changing their values, which can cause the network not to be able to learn anymore.

As noted, the other main issue that residual blocks tackle is the degradation problem; this is when adding more layers to a deep model gives a higher training error[34].

2.4.3 Batch normalization (Batch norm)

Sergey Loffe and Christian Szegedy introduced batch normalization as a method of normalizing a layer input by re-centering and re-scaling; the method is used to speed up and stabilize artificial neural networks by making normalization a part of the model architecture[10]. Batch normalization is applied for each training mini-batch and allows for a higher learning rate and less careful initialization.

Transfer learning

Transfer learning is a commonly used technique in machine learning[13]. At its core, transfer learning is when we can use a previously trained model to gain an advantage in a new but similar task. By using a pre-trained model, we can significantly reduce the time required with model training[14]. To apply transfer learning, we first have to select a model to use; the model is trained on a large dataset that has similarities to the new task that its going to be applied to. When training the model, a good practice is to do some fine-tuning, meaning that we can train some of the top layers of the frozen model[15]. Usually, when using transfer learning, the model we are applying to our new model is frozen. However, the model can usually benefit from training some of the top layers of the network, especially convolutional neural networks, as the layers that are higher up in the network are more specialized than the lower layers.

2.5 Tools

This section gives an overview of the tools that are used or considered for use for this project.

2.5.1 Data labeling tool

To train a CNN, one first has to start with gathering sample data. There is a need for a tool for this project that makes it easy to set keypoint positions in an image. There are several good options; use a pre-made tool such as `www.makesense.ai` or LabelImg. There is an option to use a pre-made dataset such as the COCO dataset. The final option is to create our own custom data and labeling tool.

makesense.ai

makesense.ai is a free to use online labeling tool[16]. The tool supports many label types, such as keypoints which is vital for this project. With makesense.ai, one still has to collect

[13] https://research.aimultiple.com/transfer-learning/
[14] https://deepai.org/machine-learning-glossary-and-terms/transfer-learning
[15] https://www.tensorflow.org/tutorials/images/transfer_learning
[16] https://github.com/SkalskiP/make-sense

the images independently since there is no pre-made data with the tool. This makes makesense.ai a viable tool for this project.

LabelImg

LabelImg as makesense.ai is a free-to-use tool, but rather than being an online tool, it runs locally[17]. LabelImg lacks its functionality; it only supports labeling bounding boxes, rendering it not viable for this project.

COCO

COCO is a widely used dataset in the community and is a viable option for this project. One of the significant downsides with COCO is the lack of flexibility; the data is already labeled, which is fantastic for saving time, yet we have no control in altering the data to our needs without making a custom tool for labeling.

Gathering our own data and creating a labeling tool

Using a tool like makesense.ai or COCO is undoubtedly a good option. As previously noted, with a tool like makesense.ai, we still have to collect our own data. However, the data formatting is something we do not have control over. So if we already have to collect our own data, another good option is to create a custom labeling tool that can handle all our needs. Since we are using BO20G38's data gathering methods with a greenscreen will be utilized, data augmentation can be built into the labeling tool pipeline. This makes gathering our data and creating a labeling tool a viable option and will be used for this project.

2.5.2 TensorFlow (TF) and TensorFlow Lite (TFLite)

TensorFlow is a free open-source machine learning and artificial intelligence library created by the Google Brain team[18] [19][12]. TensorFlow has a big and flexible ecosystem of tools for creating state-of-the-art machine learning models. TensorFlow also has a lighter framework TensorFlow Lite for running on mobile and IoT devices, making it a perfect option for this project.

2.5.3 KerasTuner

KerasTuner is a hyperparameter optimization framework that makes it easy to search for the correct hyperparameters when tuning TensorFlow/Keras model[27]. To use the package, the user defines a search space, and KerasTuner will run through the search space and find the best models by the process of elimination.

As KerasTuner is such an easy package to use, it makes a perfect candidate for this project when we will be conducting architecture searches and hyperparameter tuning.

[17] https://github.com/tzutalin/labelImg
[18] https://www.tensorflow.org/
[19] https://en.wikipedia.org/wiki/TensorFlow

2.5.4 imgaug

imgaug is an image augmentation library for machine learning [30]. The library contains a wide range of augmentation techniques. It also supports augmenting images with keypoints, bounding boxes, and more. These features are essential for when we are later going to augment images with keypoints when we are creating our datasets; this is further discussed in section 3.1.3.

2.5.5 Smartphone/Android

Android is a mobile operating system and is developed by the Open Handset Alliance that is commercially sponsored by Google[20]. Android is the OS that runs on the smartphone that will be used for this project (Huawei p30 pro). TensorFlow and TensorFlow Lite integrate nicely with Android devices, boosting the confidence in choosing TensorFlow and TensorFlow Lite for this project.

2.5.6 MoveNet.SinglePose.Thunder

MoveNet is a Human pose estimation model built on MobileNetV2, with a Feature Pyramid Network decoder and CenterNet[36]. A Feature Pyramid Network is a feature extractor that outputs proportionally sized feature maps from a single scale image at multiple levels[19]. CenterNet is a object detector that strays from the norm by using triplets, instead of a pair of keypoints[25].

MoveNet was built to run in a browser using TensorFlow.js[21] or using TF Lite on a mobile device. MoveNet outputs 17 keypoints with a confidence score for each keypoint, a list of the keypoints the model outputs can be seen in Table 5.1. For this project MoveNet can be used for Model assisted labeling (section 4.2.3), as well as comparing it to our own models for precision testing (section 5.3).

[20] https://en.wikipedia.org/wiki/Android_(operating_system)
[21] https://www.tensorflow.org/js

Chapter 3

Design and Planning

This chapter covers the planning and designing phase of the project. Section 3.1 gives an overview of the different options for data collection and creating datasets. Section 3.2 Looks into the planning of how the models for this project will be created and tested. The final section 3.3 concludes the chapter with the planning of the implementation of our model on the smartphone.

3.1 Datasets

This section covers the planning regarding collecting, labeling, and techniques for the datasets needed for this project.

With the experience gained from BO20G38[31], the plan is to implement some of the key findings in data generation with greenscreen and data augmentation techniques.

3.1.1 Collecting images

When planning started for the datasets, the first thing to be decided was how the images would be collected. Several options can be viable.

Option 1 - Using a pre-made dataset

As earlier mentioned in section 2.5.1, there are pre-made datasets out there, such as the COCO datasets. However, using a pre-made dataset brought a concern about the flexibility, the images, and future training of HPE models.

The concern with flexibility in using a dataset like COCO is that the keypoints they are using might not be optimal for use in this project.

The images in COCO and other pre-made datasets bring apprehension about how the images look. All camera sensors are a bit different; thus, the images from one camera will look different from another. This makes it so that we are less in control of the data and what our model learns.

The organization behind COCO state that there are 250 000 instances of people with keypoints in their dataset [1]. A concern with this is whether or not this will be enough data for further training of our models.

[1] https://cocodataset.org/#home

Because of these concerns using a pre-made dataset was deemed not viable for this project.

Option 2 - Scraping the web for images

Since using a pre-made dataset is out of the picture, we must create a custom dataset. An option to collect images is to scrape the web for images of people and then label those images.

However, scraping the web for images brings up a major concern that was previously discussed in section 3.1.1 (option 1) about the lack of control over the images.

Because of this Option 2 was deemed not viable for this project.

Option 3 - Capturing our own images

The method that works around all the previously discussed issues is to capture our own images. This can be accomplished in multiple ways. A high-quality camera can be used, or we can use the same device that we are later using to implement our model. For this project Option 3 was deemed the best approach for collecting image data for our datasets.

3.1.2 Data labeling

After the image collection process, they have to be labeled/marked with the coordinates of the joints needed to form a pose. As discussed in section 2.5.1 makesense.ai, there are already made tools for labeling images and setting keypoints. Makesense.ai is a viable tool for this project. However, after some testing, it turned out to be a complicated tool to use, making labeling images more complicated and time-consuming than it needs to be.

Therefore, we decided to custom make a tool for labeling images. By making our tool, we can have more control over the data and how it is formatted. One of the most significant upsides to making our tool is that it can be tailored to our workflow.

3.1.3 Creating datasets

To create the datasets, we selected to capture our images on the device that we are later implementing our HPE model; this is to control how the training data looks to be as close as possible to the real-world data when the software is in use. A key feature of capturing our datasets is the use of a greenscreen. By having a person pose with a greenscreen background, we can create a vast amount of data from a small set of images.

Custom image labeling tools were created with our tools; we have total control over the workflow, data, and formatting.

When a set of greenscreen images have been labeled, we can change their backgrounds. From the images that have then been created with the new backgrounds, we can then use them and create an even more extensive set of labeled images by using data augmentation, with the help of frameworks like imgaug (2.5.4). Data augmentation is a crucial part of how the datasets for this project are made. The augmentation techniques that are used in this project are further explained in Chapter 4

When all the images have been captured, labeled, background swapped and augmented the data- set/sets are ready to be used to train an HPE model. These steps make the dataset vast in size, and we can adjust the complexity of the data.

3.2 Models

This section gives an overview of the planning phase of the HPE models training, and testing.

The first step in planning how to make the HPE models for this project was to research how to create keypoint detectors. Luckily there are many good examples online, such as Keras's guide: Keypoint Detection with Transfer Learning [2]. The research quickly discovered that models such as MobileNet and ResNet are commonly used for keypoint detection tasks. However, other model architectures have to be considered to answer the set of research questions.

3.2.1 Initial plan

To answer **RQ 1**, **RQ 1.1**, **RQ 2**, and **RQ 2.1** a range of pre-made models such as MobilNet, ResNet50, shall be trained and tested with the data made for this project. In addition to the pre-defined models, architecture searches and hyperparameter tuning shall be conducted to check if a different model architecture can yield better performance in terms of accuracy and speed. The search and Tuning shall utilize the Keras Tuner tool [27].

Since the models are intended to run on a smartphone, they shall be converted to a TensorFlow Lite model to run on this device, and in turn, they will be tested on how close to real-time they can run on this device. All models shall be tested on test data and real-life testing to find the best-performing model. The model that performs the best overall will be selected for the final build.

3.3 Smartphone

This section gives an overview of the planning regarding the implementations of image capture, and HPE model on the smartphone.

3.3.1 Selection of device

There is a wide range of Android smartphone devices on the market[3]. This project utilizes a Huawei P30 Pro, running Android 10. The device choice is somewhat simple as that was the phone we had at hand.

3.3.2 Image capture for datasets

After looking into the different image gathering/capture methods, the smartphone was selected to capture the dataset images. The primary camera sensor on the phone captures images in a resolution of 2736x3648, which is too large. An app was created to capture images and down-sample them to a lower resolution locally on the device for data capture. However, this was a slow process as the phone needed to process all the images locally and then write them to disk. Therefore the standard camera app was used with its burst capture, which captures 100 images in the standard 2736x3648 resolution. After the images are captured, they can be scaled down to a more reasonable resolution on a computer.

[2] https://keras.io/examples/vision/keypoint_detection/
[3] https://en.wikipedia.org/wiki/List_of_Android_smartphones

3.3.3 HPE model implementation

An app must be created to implement the HPE model on the smartphone. The must app handles the input stream of images, which are then fed to the model. For the app to interpret the poses returned by the model predictions, the app shall compute the angle for a set of selected joints. The specified joint angles are then interpreted as steering signals such as left, right, and so on. This application will also be used for performance testing on the smartphone to find what models yield the best results in terms of speed.

3.3.4 Sending steering signal to vehicle

For this project, our primary focus is on developing our Human Pose estimation models and implementing them into a smartphone application. If time allows, we will look into the connection between the smartphone and a vehicle; however, this topic will be explored in Section 4.4.5.

3.4 Framework, and plan selection

For the project to succeed, we have to select what frameworks and which plans to use moving forward. As mentioned in subsection 3.1.3, we chose Option 3 3.1.1 to capture and label the images ourselves for maximum control over the data.

After deciding to gather our data, we also chose to create a custom data labeling tool to implement all the tools that we feel necessary. Ultimately we ended up with two labeling tools which are described in subsections 4.2.2 and 4.5.

To train our HPE models, we used the machine learning framework TensorFlow 2.5.2 as we have previous experience with the framework.

As discussed in section 3.3.1, the smartphone selection was a simple process as we only had one at hand.

For a complete list of the primary packages and tools used for this project, see Table 3.1.

Tools	**Use case**
TF & TFLite[12]	Training and using ML models
imgaug[30]	Augmenting training data
KerasTuner[27]	Architecture search and hyperparameter tuning
Matplotlib[3]	Visualizing images and keypoints
TKinter[1]	Build custom labeling tool for Python
Numpy**harris2020array**	Data handling
Pillow[38]	Converting array to image
Android[37]	smartphone development
Open-cv[11][7]	Reading images from disk, and simple image transformation
MoveNet[36]	Model assisted labeling, and precision testing

Table 3.1: The packages used for this project

Chapter 4

Implementation

This chapter describes how the group found what seems to be the best solution for creating datasets, labeling tools, and Human Pose Estimation models by doing experiments with multiple model architectures, hyperparameter tuning, datasets, and labeling tools. The chapter starts with section 4.1 describing the framework implemented for pre-processing data, dataset handling, architecture search, hyperparameter search, training-, tuning-, converting-, and testing- TensorFlow models, and interpreting model predictions. Section 4.2 gives an in-depth description of how the datasets and data labeling tools were made. Section 4.3 covers how the different HPE models were found with architecture searches, and hyperparameter tuning, as well as training the models. The final section of the chapter 4.4 describes how the smartphone application was built, interpreting pose predictions from our models, and a look at different options for sending steering signals to a vehicle.

4.1 Creating a framework/package (Quantium)

We chose to develop a framework/package that contains all the logic for data pre-processing, model- training, testing, converting, architecture search/hyperparameter tuning, and visualization for this project. The package was created to have easy and quick access to functionality when needed to increase productivity. The package contains sub-packages that handle their respective parts of the project; the following subsection covers the package modules.

4.1.1 Firebolt - dataset pre-processing

Firebolt is our data pre-processing module, it handles everything needed for getting our dataset ready for training: background swapping (section 4.2.4), data augmentation (described in Section 4.2.5), auto dataset balancing, image resizing, and dataset splitting (used in Section 4.2.6). To see the code for Quantium's Firebolt module see Appendix E.2, E.5, E.3, E.4, E.6, and E.1.

4.1.2 Filch - Utility's functionality

Filch is our utility's module that handles all functionality used by more than one Quantium module, such as reading JSON files, parsing pose coordinates from datasets, loading trained models for use, and renaming files. To see the code for Quantium's Filch module, see Appendix D.1.

4.1.3 Albus - Architecture search and hyperparameter tuning

Albus is our architecture search and hyperparameter tuning module, the module utilizes Keras Tuner (section 2.5.3) to run architecture searches as well as hyperparameter tuning. Albus is used to find all the model architectures and hyperparameter tuning for all the models used in this project. To see the code for Quantium's Albus module see Appendix G.1, G.2, G.6, G.4, and G.5.

4.1.4 Dobby - dataset handler and single model training

The Dobby module has two purposes; it is responsible for training a single model and loading and delivering data to that model training.

For handling our dataset, Dobby has a class, DobbyDataset (Appendix F.2), that inherits from the *Tensorflow.Keras.utils.Sequence* module. *Sequence is a base object used to fit a sequence of data, such as a dataset*[1][12][9]. Using this way of delivering data to the training session, we can augment the data further if we desire, and have a reliable method of data delivery. The module also uses utility from Filch (section 4.1.2) to parse the data correctly. DobbyDataset also use imgaug[30] for augmentation.

DobbyAugmentation (Appendix F.1) is responsible for returning augmentation parameters for training and validation data; the module uses imgaug[30] to augment the data.

To deliver a DobbyDataset to training, we use a DobbyDelivery (Appendix F.3), which is responsible for reading all the data from the disk and creating a DobbyDataset. The module also uses DobbyAugmentation to collect augmentation parameters to augment images and keypoints before the data is delivered to the training session.

DobbyTrainer (Appendix F.4) is responsible for creating all the callbacks used during training; the callbacks used can be seen in section 4.3.2, as well as compiling the model and train that model as well as saving it when the training is finished.

4.1.5 Alastor - multiple model training session

Alastor is a module created to train all our models without stopping. The module utilizes Dobby (section 4.1.4) for training each model and Filch (section 4.1.2) for loading each model to make them ready for training.

To train all the models, Alastor loops through our dataset splits and trains each of the models; when all the models have been trained on one split, it loads the next split, and models are trained on the prior split/s trains them again. To see the code for Quantium's Alastor module, see Appendix H.1.

4.1.6 ModelBuilder - creating model architectures

The ModelBuilder module is a simple module that is responsible for building our finished model architectures to prepare them for the initial training session. To see the code for Quantium's ModelBuilder module see Appendix I.1, I.4, I.2, and I.3.

[1] https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence

4.1.7 Sirius - TFlite converter

Sirius (Appendix J.1) is responsible for converting our models from a TensorFlow/Keras model into a TensorFlow Lite model, in order to run our models on a smartphone. The module utilizes functionality from Filch (section 4.1.2) to load our TensorFlow model.

4.1.8 Lupin - model testing

Lupin is our model testing module; the module loads our testset with images and ground truth keypoints. To test our models, we use our models to predict on the testset; from the predictions, we can measure how precise the models are by looking at the distance from the predicted keypoints to the ground truth keypoints. Lupin uses functionality from Filch (section 4.1.2) for loading models, and visualizing keypoints. To see the code for Quantium's Lupin module, see Appendix K.1.

4.1.9 James - Interpreting poses

James is the signal interpretation prototype; the module looks at the angle from three keypoints and compares that to a set of predetermined angle ranges for generating a steering signal, as well as determining if the subject is facing the camera or not. James uses functionality from Filch to load models, visualize keypoints, and loading images for predictions. To see the code for Quantium's James module see Appendix L.2 and L.1.

4.2 Dataset

This section describes how the process of manufacturing the custom HPE datasets used for this project and building the tools needed. To create a dataset, there are five steps until they are ready for training and testing an ML model. The following subsections describe these four steps:

4.2.1 Image capture

Since we are not using a pre-made dataset for this project, we had to capture the images of people ourselves. The smartphone camera from our Huawei P30 Pro was utilized for image capture. To create unique datasets, the subject was placed in front of a greenscreen so that we could later change the background of the images and further augment them. The subject performed random poses while the camera on the phone used its burst capture function to capture 100 images over a period of approximately 15-20 seconds. This step is repeated until the desired amount of images is captured. After all the images have been captured, they are then transferred to a computer, ready for the next step. An example of a raw image from the smartphone camera can be seen in Figure 4.1

Figure 4.1: Raw dataset image example (2736x3648 pixels).

4.2.2 Image Labeling tool (C3P0-JavaScript)

To label images, we needed a tool. As we were not satisfied with other tools on the market, we chose to build our own. We chose to build it as a web-based solution so that image labeling can be done anywhere. The code and installation guide can be seen in Appendix A. To store the images and their keypoint we chose to use Firebase because of its ease of use and good implementation of a real-time store option than can be exported, which is key for the later steps. The Firebase console was set up with *authentication* for security, *Storage* to hold our image files, and their *Realtime Database* to store each individual image's keypoints (label).

Firebase's Realtime Database stores the data in a JSON format that can easily be exported. An example of how we store our data can be seen in Figure 4.2.

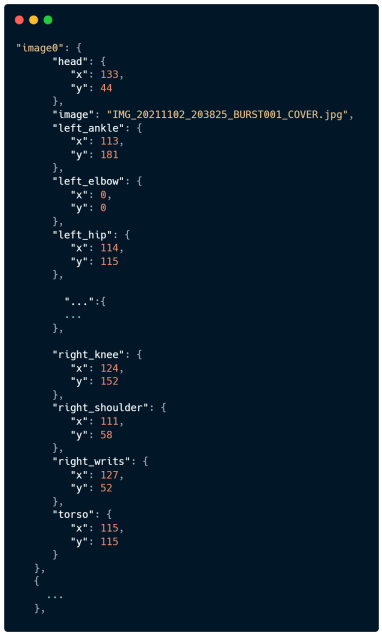
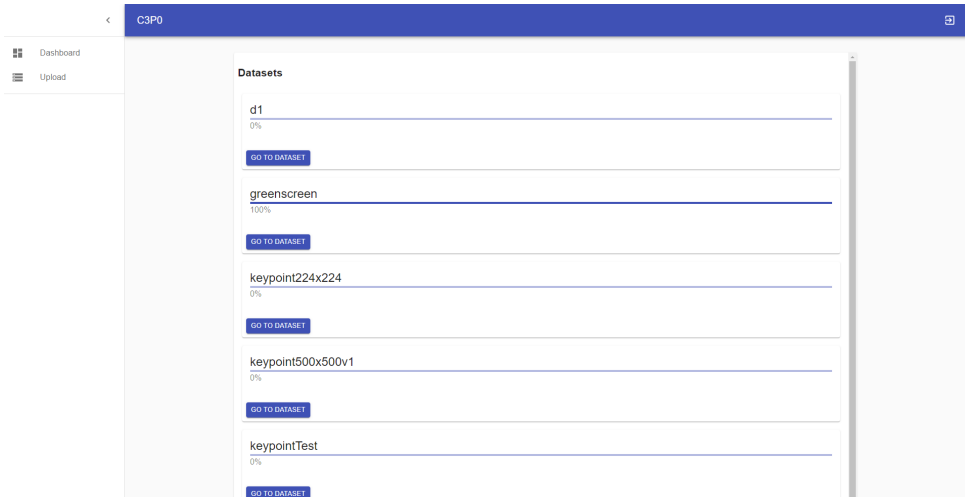


Figure 4.2: Example of how the datasets are stored from Firebase Realtime Database or C3P0-python 4.2.3.

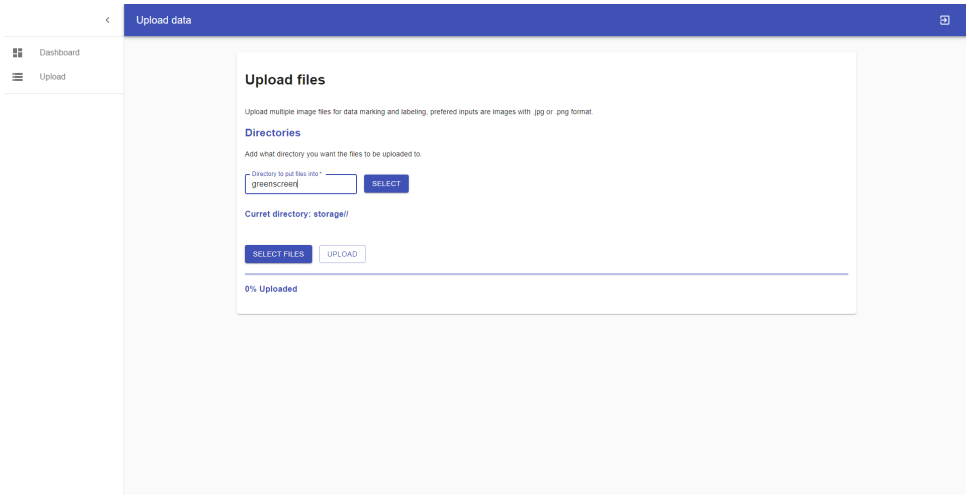
Upload module

So with the Firebase tools in place, the React part of the system was created. Firstly the secure login with Firebase Authentication was implemented. After the login module was finished, we had to create an upload module to upload the images captured in the image capture process to be labeled.

The upload logic was created so that each image in the dataset is uploaded to Firebase Storage so that we can render those to the web page, but at the same time, each dataset gets a record in Firebase Realtime storage; this is where the keypoints are stored. A screenshot of the upload module can be seen in Figure 4.3(b).



(a) Dashboard



(b) Upload

Figure 4.3: Screenshot of the dashboard and upload module in C3P0

Dashboard module

Next, the dashboard module was created. The dashboard is where all the datasets in the systems are listed and can be accessed and see the labeling progress of each dataset. A screenshot of the dashboard module can be seen in Figure 4.3(a).

Label module

Next, the labeling module was created. The label module renders the image to the screen. The image is placed underneath a canvas module, to which we will come back later in this section. The label module contains a set of buttons for each joint we want to place. An important thing we had to keep in mind here is that joints belonging on the left side will always have to be placed on the true left side of the subject in the picture, and the same with the joints for the right side of the body. The module also contains a button

for getting the previous pose, meaning that it will place all the joints in the same place as the previous image was. This was added to make the tool more productive as we are capturing images at such a fast rate that some of the joints may stay in the same place as the previous image so that the joints that move can then be adjusted. There are also tools for moving the entire pose up, down, left or right, and a button to save the current keypoints that have been placed.

To make the labeling logic work, we had to place a canvas module on top of the image rendered from Firebase Storage. The canvas module consists of a classic HTML/JavaScript canvas element [2]. This module listens to the current position of the mouse pointer on the image. When a joint in the label module is selected, and the mouse is clicked at a point on the image/canvas, a local React useState variable is updated for that joint. When all the joints have been labeled, the user can save the current pose. That pose will then be written to the Firebase Realtime Database as a record for that image. A screenshot of the label and canvas module can be seen in Figure 4.4.

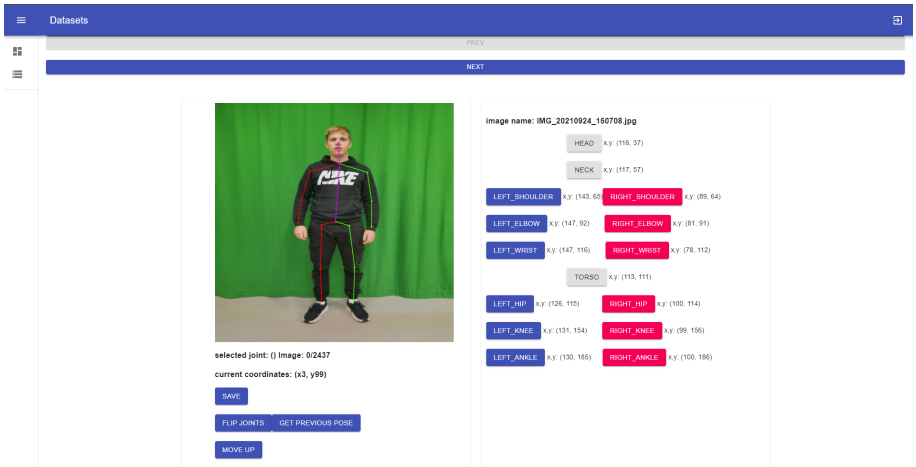


Figure 4.4: Screenshot of C3P0-JavaScript image labeling tool GUI.

Labeling images

Before uploading the images to Firebase, they have to be resized. Uploading images in the resolution of 2736x3648 is a waste of storage space and too large. Therefore we use a simple python script from Quantium's Firebolt module (section 4.1.1) to resize all the images to 1000x1000. After the images have been resized, we then upload the images. When the images are uploaded, the labeling process can begin. The users access the dataset through the dashboard, which redirects them to the labeling module. When in the labeling module, the user selects the joints and places them on the image, and uses the tools as they see fit. For this project we are using 15 joints; *head*, *neck*, *torso*, *left* and *right -shoulder*, *-elbow*, *-wrist*, *-hip*, *-knee*, and *-ankle* with x and y values ranging between 0 and 224. One important note is that if a joint is not visible, we are placing the join in x=*0* and y=*0* to mark them as unlabeled. When the joints for all the images have been placed and saved, we then export the JSON file that the Firebase Realtime database generates and move into the next steps of changing the background and data augmentation.

[2] https://www.w3schools.com/html/html5_canvas.asp

4.2.3 Image Labeling tool (C3P0-Python)

In combination with the labeling tool made with JavaScript that was made to run in the browser, we also made a simple yet even more powerful Python version of the labeling tool. Developing a Python version of the labeling software was to have a more accessible version to tweak and add functionality as needed and have the opportunity to work offline.

C3P0-Python is built with the Python framework Tkinter, which is the standard interface/GUI toolkit for Python [3]. The tool also utilizes the image package Pillow[4] for loading images and doing some processing such as rotating the images. The final Package that the software depends on is TensorFlow for Model Assisted Labeling (section 4.2.3).

One of the significant upsides to developing a Python version is that we are then able to utilize the powerful TensorFlow version made for Python to utilize Model Assisted Labeling(MAL) (section 4.2.3). Since the development of C3P0-Python, we fully transitioned out our JavaScript version of the tool and chose not to implement TensorFlow JS to that version of the software for MAL.

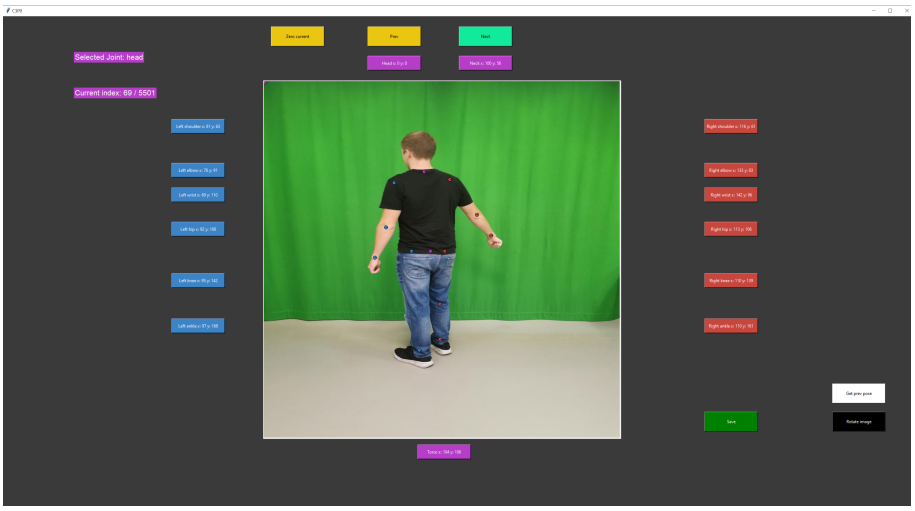


Figure 4.5: Screenshot of C3P0-Python image labeling tool GUI.

C3P0-Python behaves the same as the JavaScript version (section 4.2.2) when labeling images, except for some quality of life functionality, such as moving keypoints with keyboard shortcuts for more precise and faster labeling of the data. The user interface looks different, as seen in Figure 4.5. As previously mentioned, the JavaScript version of the tool was utilizing Firebase, and the tools that come with that database suite, the Python version of the software removes that extra dependency for a separate database and instead stores everything locally on the machine the software is running on. To see a user guide and code for C3P0-Python see Appendix C.

[3] https://docs.python.org/3/library/tkinter.html
[4] https://pillow.readthedocs.io/en/stable/

Model Assisted Labeling(MAL)

Model-assisted labeling is functionality where we use a previously trained model to help place keypoints in our images to speed up the labeling process. To help us place rough estimates of most keypoints, we use MoveNet Thunder 2.5.6. After implementing MoveNet, we quickly noticed that the model was easily confused with our data and returned too rough joint positions. For this project, we are also using keypoints for *neck* and *torso*; therefore, we could only use MoveNet for rough estimations on images that we were struggling to place keypoints.

4.2.4 Greenscreen

When all the images have been labeled with keypoints, the datasets are relatively small, as seen in table 4.1. However, since we want to have a lot of data for our model to generalize and learn from, we need to create more data. Earlier in this section, we discussed utilizing a greenscreen when capturing the images. To increase the dataset sizes, we replace the greenscreen background with a set amount of images to get the desired size we are after. To do this, we created a background replacement script. Nevertheless, before the backgrounds are changed, we again use our resize script to scale the images to their final size of 224x224.

The script works in the way that it masks out a set of pixels within an RGB range; in our case, the range is [0, 100, 0] to [120, 255, 130], where index 0 is red values, index 1 is green values, and index 2 is blue values. The mask that was just created is applied to the new background, and an inverted version of the mask is applied to the original image. These new images are then merged. To not have identical backgrounds for every image, we move the background image around for each iteration using the same background.

To keep the keypoint labels, the script takes the keypoints from the original greenscreen image and writes the new image to a new folder and these keypoints to a new JSON file that will become the base of the new dataset before further data augmentation. These steps are repeated for all the images in the original dataset. This means that a dataset with an original size of 2500 can be increased based on the number of new background images and augmentations one wants to apply.

4.2.5 Data Augmentation (DA)

To make the datasets bigger and more complex, we utilize data augmentation. To help with data augmentation, we are utilizing the framework imgaug[5]. Imgaug makes it easy to work with keypoint datasets.

One of the main reasons that Imgaug was used for this step is how it handles the image transformations in combination with keypoints; even as we are rotating, scaling, or shearing the images with the help of Imgaug's KeyPointsOnImage module[6], we can move the keypoints to the correct location in the new augmented image. An example of how the images are before and after data augmentation can be seen in Figure 4.6.

(a) Before (b) After (dataset 2) (c) After (dataset 2)

Figure 4.6: Data augmentation before and after.

4.2.6 Finished dataset

After all the image capture, labeling, background swapping, and data augmentation, we apply one last step to help with performance when we train our models. This step came about after some experimenting and crashes. Originally we modified the keypoint dataset code found in the Keras guide; *Keypoint Detection with Transfer Learning*[7].

However, this turned out to be highly inefficient with RAM usage, requesting over 128Gb with an image count of 206 346; this turned out to be too much for our hardware. Therefore we wrote a script to compile our datasets into a TensorFlow dataset[8]. The datasets are saved[9] so that we can load them into memory later when we are training our models on a server[10].

We reduce the ram usage from what was previously stated to around a third, and sometimes less with the same sized dataset. In Table 4.1 we are showing a complete list of our datasets and their size throughout their background and augmentation processes.

After using the newly formatted dataset for some training sessions, we returned to the original data approach. Rather than loading all the data at once, we split the datasets into eight parts and run separate training's on each of the splits. This was due to an issue we

[5] https://imgaug.readthedocs.io/en/latest/

[6] https://imgaug.readthedocs.io/en/latest/source/api_imgaug.html?highlight=keypointsOnImage#imgaug.imgaug.KeypointsOnImage

[7] https://keras.io/examples/vision/keypoint_detection/#prepare-data-generator

[8] https://www.tensorflow.org/api_docs/python/tf/data/Dataset

[9] https://www.tensorflow.org/api_docs/python/tf/data/experimental/save

[10] https://www.tensorflow.org/api_docs/python/tf/data/experimental/load

found when visualizing the data from a TensorFlow dataset that the data looked radically different compared to the original data. The main contribution that made it possible to use this dataset implementation method was that we got access to more powerful hardware that was able to run it.

Name	**Initial size**	**Bg swap size**	**DA size**	**Batch size**
Dataset 1	2023	103 173	206 346	64
Valset 1	414	19 004	38 088	64
Dataset 2	11000	330 000	660 000	flexible
testset 1	600	-	-	flexible

Table 4.1: List of data, validation, and test -sets throughout their creation phases, til completed sets. *Initial size* referrers to the data size from image capture. *Bg swap size* referrers to the size of the set after the backgrounds of the images has been swapped. *DA size* referrers to the number of images in the set after data augmentation has been applied. *Batch size* referrers to the size of each batch used in training during an epoch.

Dataset 1

Dataset 1 is our simplest dataset, which was produced to ensure that our approach would work as intended. We were calling it a simple dataset because it contains one person and the angle of the image is relatively the same. The person is rotating around and moving a bit, but the camera is relatively stationary.

After testing and training with dataset 1, we chose to retire it as a training set and instead use it as a test set.

Dataset 2

Dataset 2 contains more complex and realistic data compared to Dataset 1 (section 4.2.6). This dataset contains two people with multiple camera perspectives compared to dataset 1's static camera placement. An issue with dataset 2 is the sheer size of it, as seen in Table 4.1. Because of hardware limitations, we had to split the dataset into eight smaller sets. For dataset 2 we used the data augmentation parameters seen in table 4.2. An example of how the data looks after augmentation for can be seen in Figure 4.6(b) and 4.6(c).

Dataset 2 was experimented with to see if flipping the images and keypoints would help with getting more data for each joint, but after trying a few training runs with flipped images, the results was worse than without flipping.

Testset 1

Testset 1 is a simple testset containing 600 images of a subject in various poses and images containing no person. The set has not been background swapped nor augmented in any way. Examples of the data can be seen in Figure 4.7. This set is meant for testing the accuracy of our models so that we can measure their precision when predicting keypoints.

Augmentation	**Parameters**	Applied to data (%)
Rotate	Range(-20, 40)	100
Brightness	Range(0.7, 1.2)	100
Translate	x: Range(-50, 50) , y: Range(-50, 50)	100
Sheer	Range(-2, 2)	100
Scale	Range(0.5, 1.6)	100
Gaussian blur	Sigma(0, 1.9)	10
Linear contrast	range(0.75, 1.5)	10
Gaussian noise	(scale=(0.0, 0.05 * 255), per_channel=0.1))	10
Motion blur	Severity(1)	10
Snow filter	Severity(1)	10
Rain Filter	Severity(1)	10

Table 4.2: Augmentations for dataset 2, their parameters, and the amount of images the augmentations are applied to.

(a) Before (b) After (Testset 1) (c) After (Testset 1)

Figure 4.7: Example data from testset 1.

4.3 Models

This section covers how the different models for this project is created, and trained.

4.3.1 Input and Output shapes

Before we started creating our models, we had to determine the input shape of the data and how we wanted the model to output its predictions. A commonly used image size in the community is *224x224* pixels; therefore, we are going to be using that for input, as well as the three color channels RGB. The shape of the input data will then be 224x224x3.

For the output shape, we have to take into account that we are looking for *15* joints with two values, X and Y, meaning that our model will output *30* values. We are making two approaches; one fully convolutional approach, where the output layer is a SeparableConv2D layer. The reasoning behind this is that Convolutional layers are less sensitive to parameters. The second approach utilizes a Dense layer as the output layer. Both approaches are tested with all the different models discussed in this chapter.

Therefore it was decided that the output layers shape is *(1, 1, 30)* for ease of use.

4.3.2 Callbacks

When training all our models, we are using a set of callbacks. A callback is used on the fit function so that we can tap into the various stages of the training sycle[11]. The callbacks we discuss in this section are applied in the same manner for all our models.

Tensorboard

Tensorboard is what we are using for logging and visualizing the different training sessions. We have set the callback to log training and validation loss, as well as training and validation, mean absolute error(MAE) [12].

CustomLRCallback

To try and optimize the training as much as possible, we implemented a custom learning rate callback. With this callback, we can adjust the learning rate to a lower or higher value as we see fit for each epoch during a training session. It works by listening to changes in a specific file at the beginning of each epoch, and adjusts the learning rate accordingly for the next epoch. The code for the callback can be seen in Figure 4.8.

```
class CustomLRCallback(keras.callbacks.Callback):
    """Custom callback for learning rate scheduler.
    """

    def __init__(self, old_lr, model_name):
        super(CustomLRCallback, self).__init__()
        self.old_lr = old_lr
        self.model_name = model_name

    def on_epoch_begin(self, epoch, logs=None):
        with open(f"{self.model_name}-lr.txt", "r") as f:
            learning_rate = f.read()

        self.model.optimizer.learning_rate = float(learning_rate)
        print(f"Epoch {epoch}: Learning rate changed to
              {self.model.optimizer.learning_rate} from file.")
```

Figure 4.8: Custom learning rate modifier callback.

ReduceLROnPlateau

To get the best possible training results, we are using the ReduceLROnPlateau callback[13]. This callback looks at a parameter set by the user. If there are no improvements over a specified amount of epochs, it reduces the learning rate. In our case, we set it to look at validation loss with the patience of 5 epochs and a reduced factor of 0.5. By default, the learning rate of our starts at 0.005.

[11] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback
[12] https://www.tensorflow.org/tensorboard
[13] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau

ModelCheckpoint

ModelCheckpoint is used in conjunction with the fit function to save either the entire model or just the weights at some interval so that the model can be loaded later for further training[14].

We are using the callback to get the model at its best-performing state to take it into the next training. At the end of each epoch, the callback monitors the validation loss. If the current epoch performs better than the previous checkpoint saved, it overrides the last one with the new model. At the end of a training session, we have the checkpoint of the model where it performs at its best on that training data, and this checkpoint is what we are using for the next training.

EarlyStopping

EarlyStopping monitors a given value, such as loss, to check if the model is still improving during training. If the model is not improving, the training is stopped[15]. We have set the callback to monitor the validation loss to see if the value keeps decreasing. If it has not decreased for ten epochs, the training is stopped.

4.3.3 Transfer learning models

This section covers the implementations of MobileNet and ResNet50. These are well-known machine learning models in the community for their performance. For these model implementations we are going to be utilizing transfer learning (section 2.4.3).

MobileNetV2

MobileNet is the most relevant pre-made model for this project. MobileNet is a lighter-weight model than other models such as ResNet50. The network we are using for this project has 2 257 984 parameters, compared to VGG16's 138 357 544 parameters. MobileNet was created for mobile devices [18], meaning that it will perform better on devices such as smartphones that we are using in this project.

We are using MobileNet in a fully convolutional manner for this project, meaning there are no dense layers in the architecture. The architecture is set up for this project as follows; the Input layer taking in the shape (224x224x3), then the main Architecture of MobileNet. The MobileNet architecture is set to not include the fully-connected layer at the top of the network[16].

To find the rest of the architecture and hyperparameters for this model, we utilized Quantium's Albus module (section 4.1.3). The dataset that was used during the search was the first 147 015 images of dataset 2 (section 4.2.6). An important note is that we set the MobileNet backbone of the model to be non-trainable during the search. The search was structured in the following way:

[14] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint
[15] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping
[16] https://www.tensorflow.org/api_docs/python/tf/keras/applications/mobilenet/MobileNet

I Input layer I with an input shape of 224x224x3.

Mob MobileNet set as the backbone of the network.

SL_0 (Table 4.3) Search loop 0, Containing 1 Conv2D layer, batch normalization, ReLU as activation, Dropout.

SL_1 (Table 4.3) Search loop 1, Containing 1 SeparableConv2D layer, batch normalization, ReLU as activation, Dropout.

MP MaxPooling, pool size (2, 2), strides 1.

D Dropout with a value of 0.2

SC SeparableConv2D layer with 30 units, ReLU as its activation function.

O SeparableConv2D output layer.

The layers in the model architecture search is connected in the following order:

$$I - Mob - SL_0 - SL_1 - MP - D_0 - SC_2 - O$$

Conv2D units	SepConv2D units	Dropout	DOPS	Batch norm	Num layers
32-128	32-128	0.2-0.7	[True, False]	[True, False]	0-6

Table 4.3: Search loop 0 and 1 parameters for MobileNet, and ResNet architecture search. Conv2D units, SepConv2D units have a range of values with a step size of 32. Dropout is a range of values with a step size of 0.1. DOPS(dropout status) and Batch norm indicates if the given layer is active in the block or not. Num layers how many layers is created, range value with stepping size of 1.

After the search was completed we were left with the architecture that we were going to be using. The finished model architecture is as follows:

I Input layer I with an input shape of 224x224x3.

Mob MobileNet set as the backbone of the network.

$C2D_0$ Conv2D layer with 64 units, with ReLU as its activation function.

$C2D_1$ Conv2D layer with 128 units, using ReLU as its activation layer.

BN Batch normalization

MP MaxPooling, pool size (2, 2), strides 1.

DO Dropout set to 0,2.

$SC2D_0$ SeparableConv2D layer with 30 units, using batch normalization, and ReLU as activation.

O SeparableConv2D Output layer, with Sigmoid as its activation function.

The layers in the final model architecture is connected in the following order:

$$I - Mob - C2D_0 - C2D_1 - BN - MP - DO - SC2D_0 - O$$

For code building the search space see Appendix G.6.

When the architecture search was completed, we proceeded to train the model with the entirety of dataset 2 (section 4.2.6). To train the network, we first un-froze the first 100 layers of the MobileNet backbone to get more model performance; this is commonly known as fine-tuning. After the training was complete, we were left with the training plots seen in Figure 4.9. In the plots we see that the loss and MAE are steadily decreasing showing no signs of overfitting.

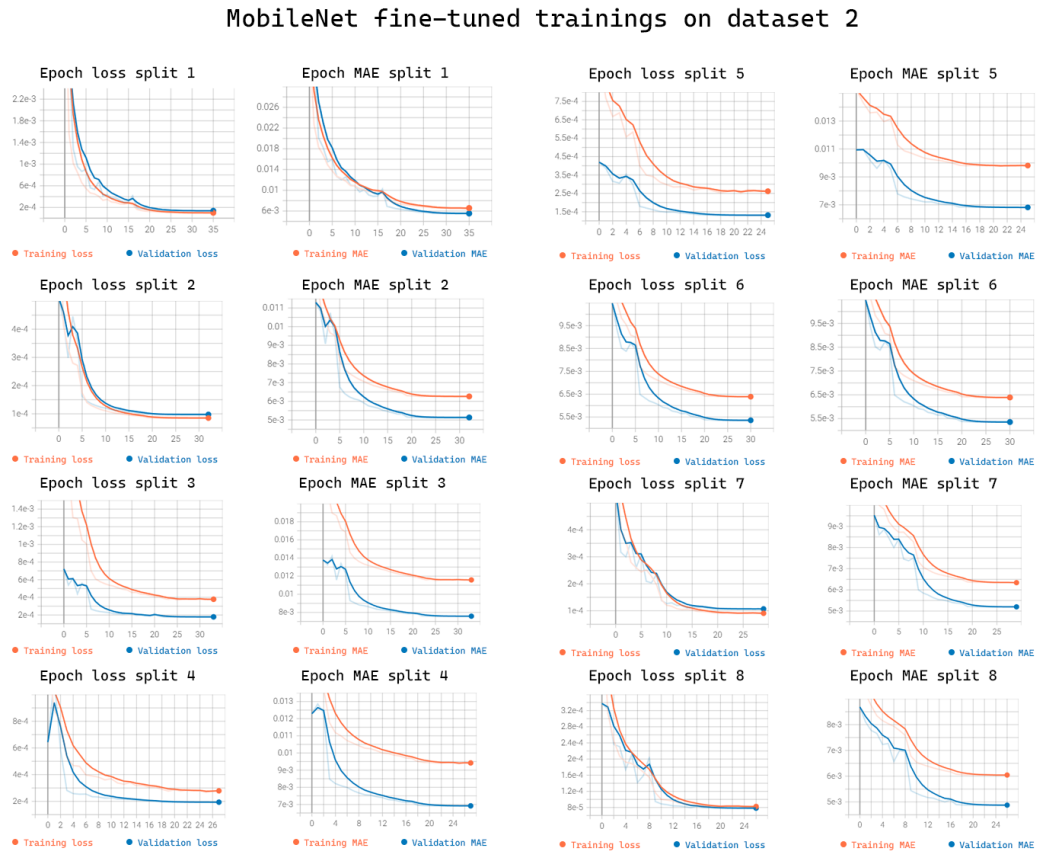


Figure 4.9: Plots from all MobileNet training sessions on dataset 2

ResNet50

ResNet is a remarkably robust model architecture. The model is well known for its ability to skip layers using so-called shortcuts or skip connections [14]. In most cases, ResNet models are implemented with either double or triple layer shortcuts, depending on the architecture [17]. ResNet50, the architecture we are using for this project, is based on the model ResNet-34. As the name suggests, ResNet50 is a 50-layer deep network, and compared to ResNet-34, it uses triple-layer shortcuts.

For this project, we are utilizing ResNet50, which has 23 587 712 parameters. Due to this model's sheer size, we are not expecting it to be the fastest performer in the later steps of this project. To find the architecture and hyperparameters for this model, we utilized Quantium's Albus module 4.1.3. The parameters used in the search can be seen in Table 4.3. During the search, we are using the first 147 015 images of our dataset 2 4.2.6 for training. The architecture search is set up in the following manner:

I Input layer I with an input shape of 224x224x3.

ResNet ResNet50 set as the backbone of the network.

SL_0 (Table 4.3) Search loop 0, Containing 1 Conv2D layer, batch normalization, ReLU as activation, Dropout.

SL_1 (Table 4.3) Search loop 1, Containing 1 SeparableConv2D layer, batch normalization, ReLU as activation, Dropout.

MP MaxPooling, pool size (2, 2), strides 1.

D Dropout with a value of 0.2

SC SeparableConv2D layer with 30 units, ReLU as its activation function.

O SeparableConv2D output layer.

The layers in the model architecture search is connected in the following order:

$$I - ResNet - SL_0 - SL_1 - MP - D_0 - SC_2 - O$$

To see the code for building the search space see Appendix G.5

[17] https://viso.ai/deep-learning/resnet-residual-neural-network/

When the architecture search was finished we were left with the following model:

I Input layer I with an input shape of 224x224x3.

ResNet ResNet50 set as the backbone of the network.

$C2D_0$ Conv2D layer with 96 units, with ReLU as its activation function.

BN Batch normalization

$C2D_1$ Conv2D layer with 64 units, using ReLU as its activation layer.

$C2D_2$ Conv2D layer with 96 units, using ReLU as its activation layer.

MP MaxPooling, pool size (2, 2), strides 1.

$SC2D_0$ SeparableConv2D layer with 64 units, using batch normalization, and ReLU as activation.

DO Dropout set to 0,2.

$SC2D_1$ SeparableConv2D layer with 30 units, using batch normalization, and ReLU as activation.

O SeparableConv2D Output layer, with Sigmoid as its activation function.

The layers in the final model architecture is connected in the following order:

$$I - ResNet - C2D_0 - BN - C2D_1 - C2D_2 - MP - SC2D_0 - DO - SC2D_1 - O$$

When the ResNet model architecture was found, we moved on to training the model with all eight splits of our dataset 2, and unfreezing the top of the model, in the same manner as we did with MobileNet. The plots of the training sessions can be seen in Figure 4.10. In the plots we that the loss and MAE are steadily decreasing, and showing no signs of overfitting.

See Appendix I.3 for the model building code.

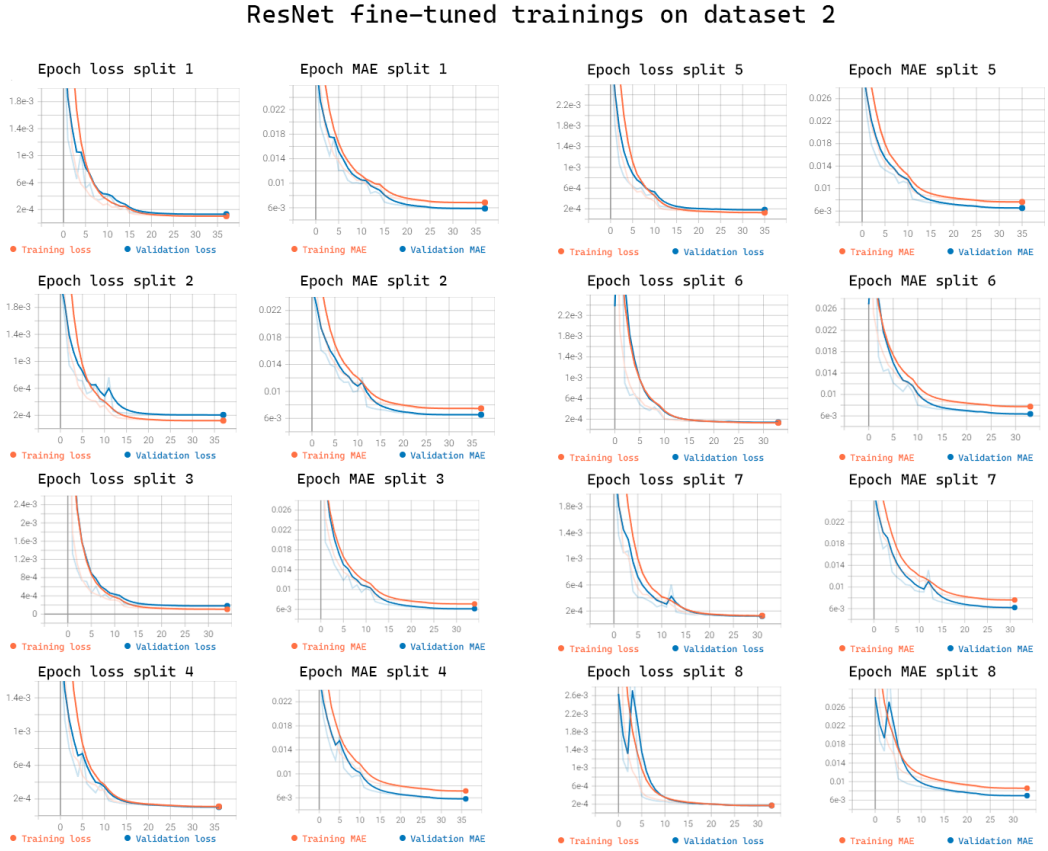


Figure 4.10: Plots from all ResNet training sessions on dataset 2

4.3.4 Full CNN architecture

To find the architecture and hyperparameters for this model, we utilized Quantium's Albus module (section 4.1.3). The search utilized our dataset 1 (section 4.2.6) containing 206346 training samples and 38088 validation samples. The parameters for the search are listed below in Table 4.4, and Table 4.5. The structure of the search was as follows:

I Input layer I with an input shape of 224x224x3.

CI Conv2D layer with tuner units_Input, using batch normalization, and ReLU as its activation function.

MP_0 MaxPooling, pool size (2, 2), strides 2.

SL_0 (Table 4.4) Search loop 0, Containing 1 Conv2D layer, batch normalization, ReLU as activation, Dropout.

MP_1 MaxPooling, pool size (2, 2), strides 2.

SL_1 (Table 4.4) Search loop 1, Containing 1 SeparableConv2D layer, batch normalization, ReLU as activation, Dropout.

MP_2 MaxPooling, pool size (3, 3), strides 3.

SC_0 (Table 4.5) SeparableConv2D layer with tuner SOU0, ReLU as its activation function.

MP_3 MaxPooling, pool size (2, 2), strides 2.

SC_1 (Table 4.5) SeparableConv2D layer with tuner SOU1, ReLU as its activation function.

MP_4 MaxPooling, pool size (2, 2), strides 2.

SC_2 (Table 4.5) SeparableConv2D layer with tuner SOU2, ReLU as its activation function.

MP_5 MaxPooling, pool size (2, 2), strides 2.

O Output layer, with Sigmoid as its activation function.

The layers in the model architecture search is connected in the following order:

$$I - CI - MP_0 - SL_0 - MP_1 - SL_1 - MP_2 - SC_0 - MP_3 - SC_1 - MP_4 - SC_2 - MP_5 - O$$

To see the code of running building the search space, see. Appendix G.2

After the search was competed we were left this the following model:

I Input layer I with an input shape of 224x224x3.

CI Conv2D layer with 352 units, using batch normalization, and ReLU as its activation function.

MP_0 MaxPooling, pool size (2, 2), strides 2.

$C2D_0$ Conv2D layer with 384 units, using batch normalization, and ReLU as its activation layer.

DO_0 Dropout set to 0,5.

$C2D_1$ Conv2D layer with 352 units, using ReLU as its activation layer.

$C2D_2$ Conv2D layer with 416 units, using batch normalization, and ReLU as its activation layer.

DO_2 Dropout set to 0,2.

$C2D_3$ Conv2D layer with 96 units, using batch normalization, and ReLU as its activation layer.

$C2D_4$ Conv2D layer with 128 units, using batch normalization, and ReLU as its activation layer.

DO_3 Dropout set to 0,3.

MP_1 MaxPooling, pool size (2, 2), strides 2.

$SC2D_0$ SeparableConv2D layer with 320 units, using batch normalization, and ReLU as activation.

$SC2D_1$ SeparableConv2D layer with 352 units, using ReLU as activation.

$SC2D_2$ SeparableConv2D layer with 128 units, using ReLU as activation.

DO_4 Dropout set to 0,5.

$SC2D_3$ SeparableConv2D layer with 256 units, using ReLU as activation.

$SC2D_4$ SeparableConv2D layer with 352 units, using batch normalization, and ReLU as activation.

$SC2D_5$ SeparableConv2D layer with 416 units, using ReLU as activation.

MP_2 MaxPooling, pool size (3, 3), strides 3.

SC_0 SeparableConv2D layer with 448 units, ReLU as its activation function.

MP_3 MaxPooling, pool size (2, 2), strides 2.

SC_1 SeparableConv2D layer with 224 units, ReLU as its activation function.

MP_4 MaxPooling, pool size (2, 2), strides 2.

SC_2 SeparableConv2D layer with 480 units, ReLU as its activation function.

MP_5 MaxPooling, pool size (2, 2), strides 2.

O Output layer, with Sigmoid as its activation function.

Num blocks	**Conv2D**	**SepConv2D**	**Dense**	**Batch norm**	**DPO**	**DPOS**
2-6	32-512	32-512	32-512	[True, False]	0.1-0.7	[True, False]

Table 4.4: Parameters for Search loop 1. **Num blocks**: How many blocks of layers are added to the model. **Conv2d, SepConv2D, Dense**: The number of neurons/units in the specified layer of a single block. **BatchNorm**: True if BatchNormalization layer is present in the block, otherwise false. **DPO**: percent of the input units to drop in single block dropout layer. **DPOS**: True if dropout is present in the block, otherwise false.

units_Input	**SOU0**	**SOU1**	**SOU2**	**DOPS**
32-512	32-512	32-512	32-512	[True, False]

Table 4.5: Parameters for all single layers outside of search blocks. Range steps set to 32.

To see the code for building the model see Appendix I.1.

After the search was completed, we trained the architecture from scratch with our dataset 2 (section 4.2.6). The model was trained on all eight splits of dataset 2; in Figure 4.11 are the training, validation, loss, and MAE plots for all eight training sessions. From the plots we see that the loss and MAE values are decreasing we see that the validation values are following the training values, and showing no sighs of overfitting.

Figure 4.11: Plots from all Full CNN training sessions on dataset 2

4.3.5 CNN with dense layers

After completing our search for a fully convolutional architecture, we moved on to a CNN architecture utilizing dense layers. To find the architecture and hyperparameters for this model, we utilized Quantium's Albus module (section 4.1.3). As with our Fully CNN model (section 4.3.4) we are used our dataset1 (Table 4.1) containing 206346 training samples and 38088 validation samples. The CNN with dense layers search utilizes the same search loops as our Full CNN search and a search loop for the dense layers. The parameters for the dense search loop can be seen in Table 4.4. To see the code for building the search space, see Appendix G.3

I Input layer I with an input shape of 224x224x3.

CI Conv2D layer with tuner units_Input, using batch normalization, and ReLU as its activation function.

MP_0 MaxPooling, pool size (2, 2), strides 2.

SL_0 (Table 4.4) Search loop 0, Containing 1 Conv2D layer, batch normalization, ReLU as activation, Dropout.

MP_1 MaxPooling, pool size (2, 2), strides 2.

SL_1 (Table 4.4) Search loop 1, Containing 1 SeparableConv2D layer, batch normalization, ReLU as activation, Dropout.

$Flatten_0$ Flatten from Conv2D to dense layers.

SL_2 (Table 4.4) Search loop 1, Containing 1 dense layer, batch normalization, ReLU as activation, Dropout.

O Output layer, with Sigmoid as its activation function.

The layers in the model is connected in the following order:

$$I - CI - MP_0 - SL_0 - MP_1 - SL_1 - Flatten_0 - SL_2 - O$$

The initial observation training CNN with dense layers is that the models are much more sensitive to parameters and becomes significantly larger than the fully CNN architecture that was previously trained (section 4.3.4). We did anticipate this but still wanted to research if CNN with dense layers was an option.

When the search was completed, we were left with an under-fitted model that was unnecessarily large that would not be adequate for this project.

Because of the results from the search we decided to run the search without SL_1, to see if that would make any difference to the model performance. The model was then connected in the following way:

$$I - CI - MP_0 - SL_0 - MP_1 - Flatten_0 - SL_2 - O$$

Even after changing the search space, the models reached as many as 200 million parameters, resulting in a shutdown of the training. Even though the search was stopped, the earlier models showed the same tendencies as the previous attempt.

To combat the issues, we continued to try and tune the architecture search without success. The models kept under-fitting. Due to these issues, CNN with dense layers was not further explored.

4.3.6 Residual CNN

After training our full CNN architecture and CNN with dense layers, we moved on to a CNN with residual blocks. This architecture search and hyperparameter tuning also used Quantium's Albus module (section 4.1.3). The data that was used in this search were our Dataset 1 (section 4.2.6). For search space code see Appendix G.4.

The search was structured in the following way:

I Input layer I with an input shape of 224x224x3.

CI Conv2D layer with tuner units_Input, with choice to use or not to use batch normalization, and ReLU as its activation function.

C_0 Conv2D layer with tuner units set to range(32-512), with choice to use or not to use batch normalization, and ReLU as its activation function.

RES_BLOCKS_0 Residual blocks search space containing batch normalization and separable Conv2D layers with relu as activation. See Table 4.6 and Figure 4.12

MP_0 MaxPooling, pool size (2, 2), strides 4.

SC_0 (Table 4.5) SeparableConv2D layer with 30 units, with batch normalization, and ReLU as its activation function.

MP_1 MaxPooling, pool size (2, 2), strides 4.

O Output layer, with Sigmoid as its activation function.

The architecture search is connected in the following order:

$$I - CI - C_0 - RES\_BLOCKS - MP_0 - SC_0 - MP_1 - O$$

Units	Batch norm	Number of blocks
32-512	[True, False]	2-6

Table 4.6: parameters for residual block search space. Units indicates how many units a SparableConv2D layer has, with a step size of 32. Batch norm, indicates if batch normalization is active or not for that block. Number of blocks indicates how many residual blocks is included in the architecture

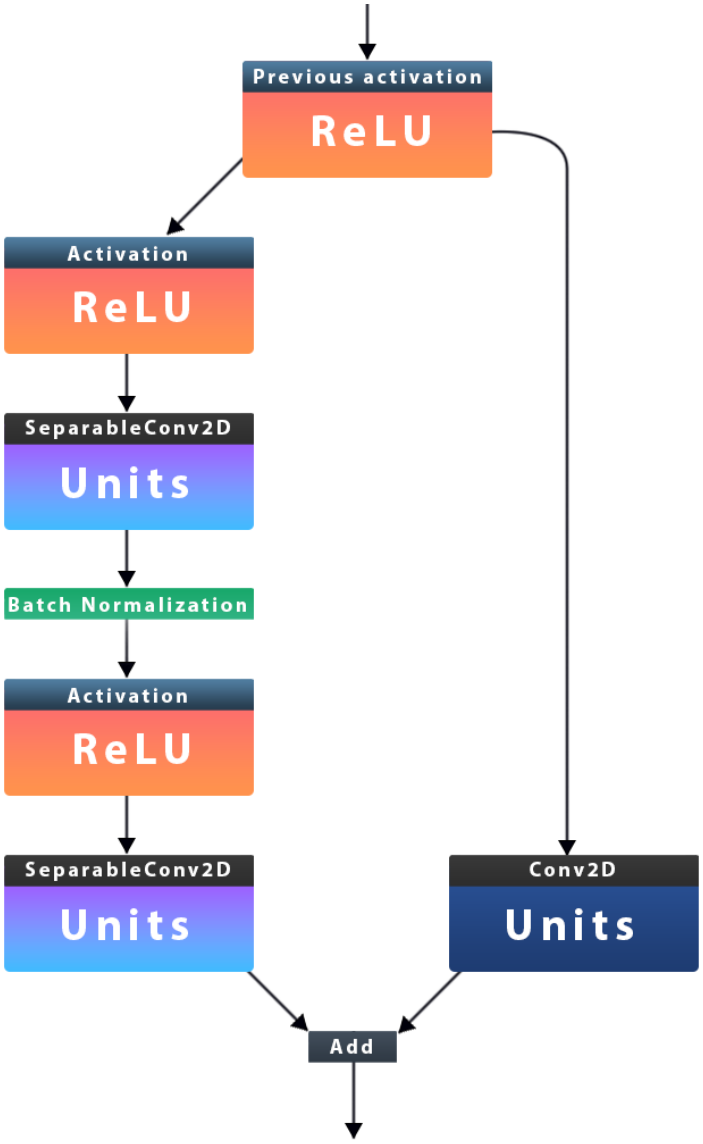


Figure 4.12: Architecture of the residual blocks used in the Residual CNN model.

After the search was completed we were left with the following architecture containing six residual blocks:

I Input layer I with an input shape of 224x224x3.

CI Conv2D layer 416 units, using ReLU as its activation function.

C Conv2D layer with 224 units, using batch normalization, and ReLU as its activation function.

Block 0 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 0 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 1 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 2 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 3 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 4 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

Block 5 Residual block containing SeparableConv2D layers and MaxPooling2D, see Table 4.7 for parameters for each layer, and Figure 4.12 to see how the layers are connected.

MP_0 MaxPooling, pool size (2, 2), strides 4.

SC_0 SeparableConv2D layer with 30 units, with batch normalization, and ReLU as its activation function.

MP_1 MaxPooling, pool size (2, 2), strides 4.

O Output layer, with Sigmoid as its activation function.

The blocks in the model is connected in the following order:

$$I - CI - C - Block0 - Block1 - Block2 - Block3 - Block4 - Block5 - MP_0 - SC_0 - MP_1 - O$$

For model building code see Appendix I.2

Layers	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5
SepConv2D (units)	320	416	256	160	416	32
Batch Normalization	Active	Active	Active	Active	Active	Active
Activation	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU
SepConv2D (units)	320	416	256	160	416	32
MaxPooling2D	Active	Active	Active	Active	Active	Active
Residual Conv2d (units)	320	416	256	160	416	32

Table 4.7: Layers and their parameters for each residual block in our residual model.

After the model architecture was found, we trained it on the eight splits of dataset 2. After the training session was completed, we were left with the plots seen in Figure 4.13.We see from the training plots that the training sessions did not go as smoothly as the previous model training sessions of our MobileNet, ResNet, and full CNN models. The validation loss and MAE in training split seven jumps at epoch eight before decreasing and falling below the training values.

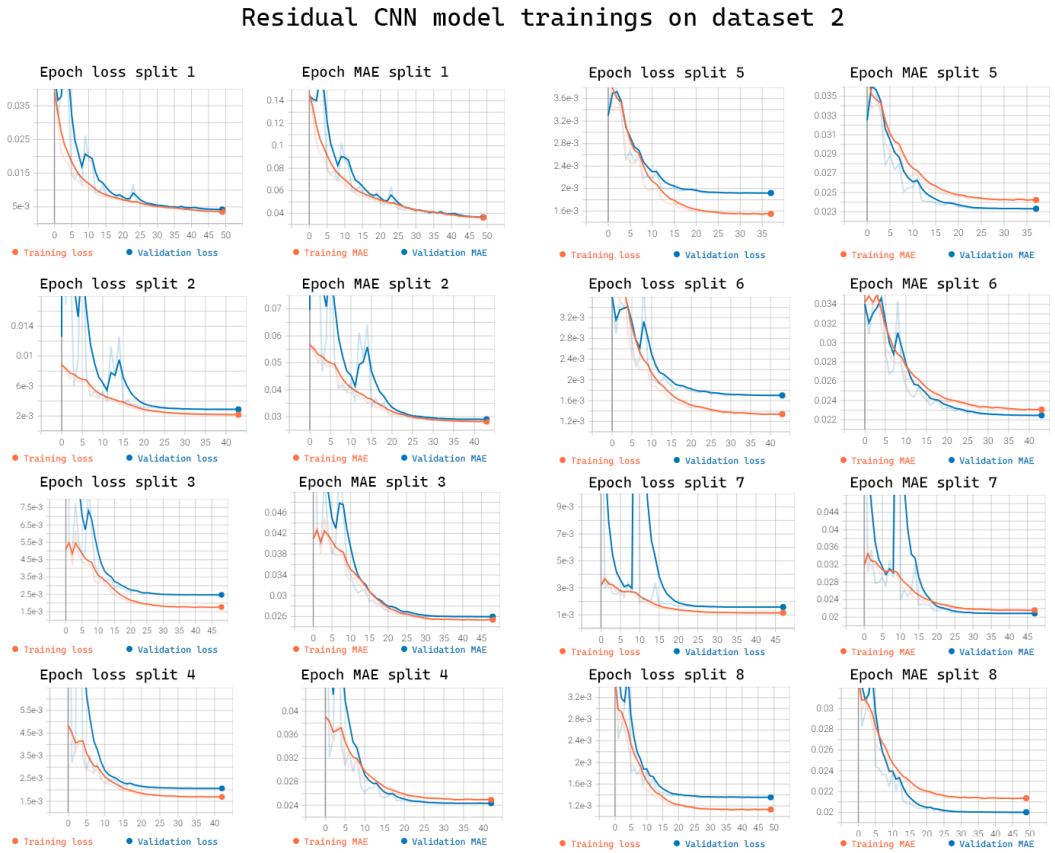


Figure 4.13: Plots from all Residual CNN training sessions on dataset 2.

4.4 Smartphone application - Luke

We will need to run our models on a smartphone; we need an application to capture images and feed those to our models. This section covers the implementation of the models on the smartphone and the development of the smartphone application.

4.4.1 Image capture

A core feature of our application is image capture. As we are running Android, we are going to be using the CameraX Library[18]. To get our application started, we utilized the CameraX tutorial *Getting Started with CameraX[32]* released by the Android team. In the tutorial, the developers talk about the functionality of the ImageAnalysis (IA) tool that is built into CameraX. Image Analysis(IA) provides CPU accessible images in our application [19].

After implementing the foundation for our application, we had to implement our ImageAnalysis functionality. To utilize IA, we are overriding the analyze function. As we want our prediction to start when the application starts, we are starting our IA in the *startCamera* function.

The analyze function uses an ImageProxy object that we will be utilizing as input data to our models. However, before we can use it, we have to convert it into a TensorBuffer object. To do the conversion we have to firstly convert our ImageProxy object into a bitmap, To do this conversion we are using so functionality from a TensorFlow Lite example *Pose Estimation* [39] namely their *YuvToRGBConverter* functionality, and their toBitmap function. The toBitmap function converts the *ImageProxy* from a YUV format and returns an RBG bitmap of our ImageProxy; we then create a *ImageProcessor* object that will make sure that the finished image is the correct size for our models. After the *ImageProcessor* object is ready we initialize a *TensorImage* object that holds *FLOAT32* values, and load the bitmap into that *TensorImage*. After the bitmap is loaded we initialize a *TensorBuffer* and load the buffer from our *TensorImage* object into it. When the buffer has been loaded, it is ready for our model to make a prediction on the image.

4.4.2 Implementation of models on device

Implementing a TF Lite model into an android app is relatively simple. However, before we implement our models, we have to convert them to a TFLite model. To handle the conversion process, we are using Qunatium's Sirius module seen in Appendix J.1.

After the TensorFlow models have been converted to TFLite models, we can import them into our Android Studio project, which takes care of all the TF lite dependences. After importing the models, we are then ready to load them and feed the images in our *analyze* function.

When a model has made a prediction, it returns a list of coordinates in a specific order that was decided when the training data was created. From there, we take the coordinates the model prediction returned and display them to the screen with our drawing module, as well as interpret the poses into steering signals (section 4.4.3) that can be sent to a vehicle.

[18] https://developer.android.com/jetpack/androidx/releases/camera
[19] https://developer.android.com/reference/androidx/camera/core/ImageAnalysis

4.4.3 Interpreting the poses

Before implementing the pose interpretation into our android application, we prototyped it in Python. This section describes Quantium's James module (section 4.1.9) and the translation from Python to Kotlin.

We chose to employ angle calculations to interpret the primary signals from the poses, namely between the hips and their corresponding arms on that side of the body. By following this approach, we can get five different signals from two angle calculations: forwards, reverse, left, right, and stop.

To find what signal to send at specific angles, we used our C3P0-python tool; we added a feature for finding angles for the joints in question. From that, we were able to find the range for each signal to use for pose interpretation. In Table 4.8 we see the angle range for the different signals. To calculate the angles in Python, we used the function seen in Figure 4.14.

```python
def __calculate_angle(self, point1, point2, point3):
    """
        Calculates the angle between three points.
        """
    a = np.array(point1)
    b = np.array(point2)
    c = np.array(point3)
    ba = a - b
    bc = c - b

    cosine_angle = np.dot(ba, bc) / (np.linalg.norm(ba) * np.linalg.norm(bc))
    angle = np.arccos(cosine_angle)
    degrees = np.degrees(angle)
    if degrees is None:
        return 0
    else
        return degrees
```

Figure 4.14: Function for getting the angle of three coordinates. First get the difference of a and b, c and b, then calculate the cosine of the angle, get the radians of the angle, and finally get the angle in degrees.

To ensure that the signal is correct if the subject is facing the camera or having their back to it, we check what side of the image each shoulder is. From that, we can adjust if the signal given with an arm should mean the opposite. An example of this can be seen in Figure 4.15.

In addition to the primary steering signal interpretation we have implemented a prototype for throttle input that looks at the angle between the shoulder, elbow and wrist. For now the feature is only added to the left and right signal while forward and reverse will have the default throttle value of 10%. This feature reuses the angle computing function, and the angle ranges can be seen in Table 4.9.

(a) Back to camera (b) Facing camera

Figure 4.15: Both signals in this image indicates 'left'.

Signal	**Angle range**
Left	50, 110
right	50, 110
Forward	110, 180
Reverse	110, 180

Table 4.8: Angle ranges for each primary signal in degrees.

After we were finished with testing in Python, we translated the code into Kotlin, our Android application with Kotlin, and implemented it into our Android application as a separate class. A Kotlin translated version of our Python angle calculation function can be seen in Figure 4.16.

Throttle	**Angle range**
10%	140, 180
25%	135, 100
50%	90, 40
10% default	<40, 180 <

Table 4.9: Angle ranges for throttle input in degrees

```kotlin
private fun angleCalc(a: Coordinate, b: Coordinate, c: Coordinate): Double{
    val ab = sqrt((a.x.toDouble()*224 - b.x.toDouble()*224).pow(2) + (a.y.toDouble()*224 - b.y.toDouble()*224).pow(2))
    val bc = sqrt((b.x.toDouble()*224 - c.x.toDouble()*224).pow(2) + (b.y.toDouble()*224 - c.y.toDouble()*224).pow(2))
    val ac = sqrt((a.x.toDouble()*224 - c.x.toDouble()*224).pow(2) + (a.y.toDouble()*224 -  c.y.toDouble()*224).pow(2))
    val angle = acos((ab.pow(2) + bc.pow(2) - ac.pow(2)) / (2 * ab * bc))
    val angleDeg = angle * 180 / Math.PI
    return angleDeg
}
```

Figure 4.16: Python function for angle calculation (Figure 4.14) translated to Kotlin.

To see an installation guide and code for the smartphone application see Appendix M

4.4.4 Application user interface (UI)

The UI of our app consists of one screen; The camera preview takes up the primary part of the screen. It is there to give a visual clue to the users and see what the HPE models are seeing.

The model selector is a drop-down menu in the top left corner of the screen (Figure 4.17(b)) that shows the currently selected model; if clicked on, it shows a list of the available models to use in the application.

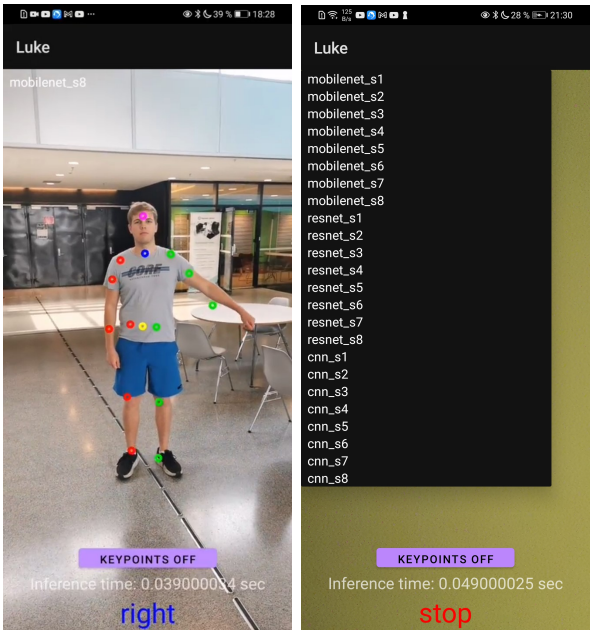
Next is the keypoint toggle button. By toggling the button, the user can choose if the 15 predicted keypoints will be drawn onto the camera preview.

Bellow, the keypoint button is an inference timer, showing how much time is elapsing for the currently selected model to make a prediction.

The last element of the application is the signal text-view showing the current detected steering signal; all signals have their respective color code, meaning if that particular signal is detected, the signal text is drawn in the respective color. A list of the different color codes can be seen in the list below.

- stop - red
- left - green
- right - blue
- forward - yellow
- reverse - magenta

A screenshot showing the application in use can be seen in Figure 4.17(a), during testing in an environment our machine learning models have never seen.



(a) Live image view (b) Model selection menu

Figure 4.17: Figure 4.17(a) showing smartphone application in live testing in an unseen environment for our machine learning model. Figure 4.17(b) showing model parts of the model selection drop-down menu

4.4.5 Sending signals to vehicles

There are multiple ways to send the steering signals from the smartphone to a vehicle depending on the vehicle. For us to send a signal, we have to consider the hardware running on the vehicle and the capabilities of that hardware.

API

Much like BO20G38's [31] approach with their API that uploads images to a remote computer, we could create a REST service that listens for a steering signal from the smartphone app. BO20G38's experiments with their API showed that uploading an image with their approach took 100ms, which is a respectable time with the hardware they used. As the data we would be sending would be smaller than an image file, we can technically speed this process up.

Firebase Real-time database

Another option is to utilize the Firebase Real-Time database, which we have previously used for this project. The smartphone application would send the steering signal to the database while the vehicle is listening for changes in the database. Firebase can easily be implemented into an Android application[20], and could also potentially be used for model hosting and added compute power.

[20] https://firebase.google.com/docs/android/setup#kotlin+ktx_2

Bluetooth data transfer

A different approach is to use an Android BluethoothSocket[21], where we can send data between two devices. With this approach, we have some concerns about speed, as we have to split the resources on the smartphone and make sure that buffers do not fill up if the vehicle machine cannot read the data quickly enough.

Cable Connection

The final option we looked into is a cable connection. Our smartphone application can write the steering signal to a local file on the phone. The phone can then have a cable connection to the machine running on the vehicle. The vehicle will then have a script running that is listening for changes in that specific file which contains the steering signals, and run by the commands in that file.

A concern we have with this approach is that writing to a file on a phone can demand a lot of resources which can slow down the entire process.

[21] https://developer.android.com/guide/topics/connectivity/bluetooth/transfer-data

Chapter 5

Testing and Evaluation

This chapter covers the testing and evaluation of our HPE models and reliability testing with our smartphone application. The chapter starts with section 5.3 which covers the accuracy testing of each of our models and comparing them against MoveNet thunder SinglePose. The following section 5.4.3 looks at how our models are performing on our smartphone. The last section 5.5 summarises the overall results of the project.

5.1 Evaluation of datasets

When developing machine learning models, it is necessary to have suitable datasets for the models to learn from. Because of this, it is necessary to evaluate the datasets made for this project before any conclusion is drawn about our models.

5.1.1 Dataset 1 & Valset 1

Dataset 1 and Valset 1 is our initial dataset and has apparent drawbacks, some of which are addressed by dataset 2 (section 5.1.2). The dataset contains images of one person in one outfit; this may cause the models trained on this set not to generalize well, meaning they will struggle with different people. The second issue with the dataset is that the camera is stationary, meaning it is not getting multiple perspectives of the subject in the image. These are the main reasons the dataset was migrated into a simple testing set.

5.1.2 Dataset 2

Dataset 2 is the main set used for training our models, and it is our largest and most complex dataset. The set addresses the issues of the fixed camera angle. It also contains two people in various outfits to try and tackle more of the generalization issue. However, it is not an optimal dataset compared to something like COCO (section 2.5.1). An issue with dataset 2 is its sheer size; depending on the hardware used to train the models, we needed to split the dataset into smaller sets of data to run model training.

5.2 Testset 1

Testset 1 is our main testing set. it resembles dataset 1 (section 4.2.6), but also dataset 2 (Section 4.2.6). The set contains 600 images of one person performing poses in various complexity, both natural and some fairly unnatural poses. This set is also made with a greenscreen so that the set can be increased in complexity.

5.3 Model accuracy testing

This section covers how the model accuracy testing was measured, conducted, evaluated and compared against MoveNet.

5.3.1 Mean keypoint distance from ground truth (MKDGT)

To test our models accuracy, we are using Quantium's Lupin module (Appendix K.1, section 4.1.8) for calculating the distance from predicted keypoints to ground truth keypoints. Lupin uses the euclidean distance formula, which is an application of the Pythagorean theorem [1] and can be written as the following:

$$Distance = \sqrt{(X_p - X_{gt})^2 + (Y_p - Y_{gt})^2}$$

To test our models we run predictions on 500 samples from a given testset. We then calculate the distance of each predicted keypoint to their corresponding ground truth keypoint and summarise them, and divide them by the number of samples to get a MKDGT.

5.3.2 MoveNet Thunder

To see how our model is performing we are going to compare it to MoveNet Thunder SinglePose (section 2.5.6), as this model is referred to in TensorFlow's documentation[2] as a *Ultra fast and accurate pose detection model*. MoveNet does differ from our models as it outputs 17 keypoints and confidence in those keypoint predictions, whereas our model outputs 15 keypoints and no confidence score. The joints used by MoveNet and the models made in this project can be seen in Table 5.1. To see how our models perform, we are looking at the joints that our models have in common with MoveNet and the two other joints we are using; torso and neck.

[1] https://www.khanacademy.org/math/geometry/hs-geo-analytic-geometry/hs-geo-distance-and-midpoints/v/distance-formula

[2] https://www.tensorflow.org/hub/tutorials/movenet

MoveNet Keypoints	Project models keypoints
nose	head
left eye	left ankle
right eye	left elbow
left ear	left hip
right ear	left knee
left shoulder	left shoulder
right shoulder	left wrist
left elbow	neck
right elbow	right ankle
left wrist	right elbow
right wrist	right hip
left hip	right knee
right hip	right shoulder
left knee	right writs
right knee	torso
left ankle	-
right ankle	-

Table 5.1: Joints used by MoveNet Thunder, and joints used by the models in this project.

MoveNet is trained and tested on a significantly more diverse set of data. For the testing we are performing for this project, our models are at an advantage as they have seen the person in the test data. However, they have never seen these images, and they differ from the training and validation data.

Joint	Distance
Head	26.649
R shoulder	58.86
R elbow	69.4
R wrist	86.42
L shoulder	35.76
L elbow	30.35
L wrist	87.47
R hip	21.702
R knee	40.102
R ankle	64.89
L hip	21.091
L knee	24.289
L ankle	18.384
Mean overall	**45.028**

Table 5.2: MoveNet Thunder single pose distance from ground truth on 500 samples from dataset 1, with an image size of 256x256

5.3.3 MobileNetV2 architecture

Our MobileNet architecture is the model we thought from the beginning would be the best performer; this is due to the previous work that has been done with MobileNet with models such as MoveNet, and the fact that it is made for mobile devices. From Table 5.3 we see that the version of our MobileNet model with the lowest overall mean is split 5 with a value of 18.403 overall mean pixel distance from ground truth keypoints.

	Split 1	**Split 2**	**Split 3**	**Split 4**	**Split 5**	**Split 6**	**Split 7**	**Split 8**
Head	9.784	11.603	13.186	13.822	10.076	8.300	9.384	7.143
Neck	3.728	3.760	3.819	3.932	3.803	3.610	3.420	3.263
R shoulder	37.103	38.857	40.741	40.707	36.426	37.179	36.694	38.466
R elbow	27.929	28.002	27.830	30.708	23.251	25.409	24.058	25.260
R wrist	34.410	38.749	37.304	37.858	34.214	35.527	34.098	34.114
L shoulder	5.509	7.389	6.038	7.755	5.835	6.759	8.055	9.160
L elbow	11.937	13.619	10.969	14.668	12.044	13.566	10.517	9.696
L wrist	13.432	16.988	17.238	16.360	13.473	13.217	18.975	12.195
Torso	8.802	9.045	9.036	9.083	9.258	8.893	9.188	9.148
R hip	10.647	11.194	10.794	10.970	10.591	10.450	10.133	10.439
R knee	38.531	37.663	36.083	36.167	38.047	39.349	39.362	38.677
R ankle	65.114	64.314	64.119	63.996	65.531	66.413	66.051	66.228
L hip	3.605	3.941	4.384	4.547	4.089	3.764	3.733	3.314
L knee	4.930	7.806	5.671	5.844	4.956	4.640	4.372	4.679
L ankle	4.629	4.838	5.409	4.919	4.451	4.106	4.418	4.396
Overall mean	**18.672**	**19.851**	**19.508**	**20.089**	**18.403**	**18.745**	**18.830**	**18.411**

Table 5.3: MobileNet testing on 500 224x224 pixel image samples from our dataset 1. Each value in split 1 - 8 represents the prediction average distance in pixels from the ground truth for each joint.

5.3.4 ResNet50 Architecture

We had high hopes for the ResNet architecture in terms of accuracy as the ResNet50 architecture is know for doing well in many image based tasks, but by looking at Table 5.4 we see that the accuracy is lacking compared to our MobileNet-based model. We observe the same issue with this architecture as we did with MobileNet, that there is a clear difference in accuracy between the left and the right sides. We see that the model that is trained for six splits has the best overall mean keypoint distance from ground truth.

	Split 1	**Split 2**	**Split 3**	**Split 4**	**Split 5**	**Split 6**	**Split 7**	**Split 8**
Head	14.092	11.420	12.306	12.998	13.992	9.479	12.667	13.411
Neck	3.752	3.570	4.237	4.312	4.110	3.571	4.375	4.130
R shoulder	39.904	38.701	40.602	37.860	40.073	37.022	38.300	38.317
R elbow	31.376	29.472	34.029	30.510	33.615	26.089	29.533	26.071
R wrist	45.652	45.942	48.831	44.829	50.217	38.709	47.042	40.069
L shoulder	9.981	5.625	14.864	7.435	8.223	8.748	11.705	8.566
L elbow	12.045	10.455	16.732	13.914	11.861	14.577	16.688	15.870
L wrist	18.259	17.350	18.388	19.007	17.120	16.051	22.615	16.993
Torso	8.513	9.159	8.684	9.182	9.098	9.246	9.057	9.041
R hip	11.920	11.074	11.080	11.976	11.128	11.659	11.857	10.840
R knee	38.512	36.408	38.127	38.539	37.123	39.144	39.665	38.565
R ankle	66.115	64.125	64.772	64.979	64.374	66.832	67.266	65.568
L hip	4.406	4.420	4.498	4.951	5.131	3.787	5.232	4.454
L knee	5.486	5.996	6.442	6.109	6.446	4.856	6.511	4.901
L ankle	5.198	5.390	5.590	6.207	6.440	4.963	6.278	4.557
Overall mean	**21.014**	**19.940**	**21.945**	**20.853**	**21.263**	**19.648**	**21.919**	**20.090**

Table 5.4: ResNet testing on 500 224x224 pixel image samples from our dataset 1. Each value represents the prediction average distance in pixels from the ground truth for each joint.

5.3.5 Full CNN architecture

Full CNN was the architecture we were hoping could show good potential. However, after running the tests on the models from the training splits, we observe that the accuracy is lacking compared to our MobileNet-based model. The model that was trained on four splits shows the best MKDGT score with an overall average of 26.111 pixels. When looking at the test results in Table 5.5 we see a reoccurring result that there is a significant difference between the left and right sides.

	Split 1	**Split 2**	**Split 3**	**Split 4**	**Split 5**	**Split 6**	**Split 7**	**Split 8**
Head	33.431	27.179	26.669	25.137	21.810	19.621	19.072	26.415
Neck	27.777	21.824	21.804	13.157	12.148	15.163	15.981	22.374
R shoulder	48.172	46.556	45.886	39.155	39.085	44.254	44.615	46.558
R elbow	48.250	56.637	55.534	39.403	44.196	47.397	44.315	42.359
R wrist	66.272	65.723	65.219	47.895	58.927	62.506	59.616	52.309
L shoulder	30.013	21.791	22.257	15.371	15.920	18.941	20.119	29.595
L elbow	39.723	31.515	31.367	27.234	35.967	36.166	36.789	37.372
L wrist	47.300	48.428	48.482	40.229	44.354	52.193	46.576	50.038
Torso	27.990	20.885	20.939	11.250	10.793	14.693	16.548	24.578
R hip	31.820	27.452	27.383	14.541	14.918	18.476	21.017	27.185
R knee	49.114	44.099	43.773	35.924	37.681	37.555	38.914	44.996
R ankle	66.177	62.799	63.013	58.879	60.021	59.295	60.264	60.476
L hip	28.047	19.096	18.936	7.864	8.145	11.258	14.424	22.813
L knee	27.108	20.112	19.502	8.072	9.200	13.300	15.793	26.282
L ankle	29.501	21.361	20.796	7.559	9.635	14.480	16.086	27.723
Overall mean	**40.046**	**35.697**	**35.437**	**26.111**	**28.186**	**31.019**	**31.341**	**36.071**

Table 5.5: Fully CNN testing on 500 224x224 pixel image samples from our dataset 1. Each value represents the prediction average distance in pixels from the ground truth for each joint.

5.3.6 Residual CNN

Residual CNN is the last trained model and showed good potential in training. In Table 5.6 we see the testing results of all eight splits model training splits. We observe that the model shows a significant difference in keypoint prediction's distance between the left and right sides. By looking at the training plots in Figure 4.13, it looks like sporadic and unstable training sessions in a majority of the splits. Residual CNN is our worst performing model, with a minimum overall mean distance from the ground truth value of 27.695 pixels at split 4.

	Split 1	**Split2**	**Split3**	**Split 4**	**Split 5**	**Split 6**	**Split 7**	**Split 8**
Head	15.564	34.619	69.217	31.247	53.079	72.536	58.395	57.385
Neck	9.3913	10.887	38.217	11.711	44.769	54.086	50.933	47.081
R shoulder	56.561	51.103	60.271	41.634	117.46	108.18	118.78	80.092
R elbow	73.605	63.483	55.914	29.557	51.328	54.377	73.973	54.683
R wrist	76.861	69.588	84.305	48.949	75.563	88.085	101.60	82.992
L shoulder	56.677	26.096	21.524	14.456	81.963	62.219	54.625	49.638
L elbow	79.995	41.643	27.918	26.678	106.29	81.985	77.304	63.056
L wrist	53.227	54.414	47.131	40.843	64.728	51.164	65.469	53.794
Torso	13.385	14.476	27.823	14.576	36.578	39.820	36.981	36.969
R hip	16.042	16.010	35.696	17.455	44.296	46.137	50.901	43.748
R knee	35.084	31.667	48.653	38.955	40.997	43.397	45.601	40.228
R ankle	56.854	53.457	61.939	58.597	52.913	51.415	51.260	52.108
L hip	16.000	14.837	19.350	11.418	29.638	33.931	28.389	32.027
L knee	22.551	18.829	10.575	12.627	22.920	23.065	22.763	23.386
L ankle	20.221	20.509	13.578	16.723	27.624	22.809	29.967	26.808
Overall mean	**40.135**	**34.775**	**41.474**	**27.695**	**56.677**	**55.547**	**57.797**	**49.600**

Table 5.6: Residual CNN testing on 500 224x224 pixel image samples from our dataset 1. Each value in split 1 - 8 represents the prediction average distance in pixels from the ground truth for each joint.

5.3.7 Testing best model splits

In Table 5.7 and 5.8, we see the best splits from all the different models tested on testset 1 and dataset 1 (section 4.2.6 and 4.2.6), where we observe that the MobileNet model has the lowest pixel distance out of the four models created for this project and MoveNet Thunder.

	MobileNet s5	**ResNet s6**	**Full CNN s4**	**Residual CNN s4**	**MoveNet**
Head	4.945	7.452	12.57	20.653	97.646
Neck	2.541	2.267	6.357	12.691	-
R shoulder	36.32	37.73	35.56	38.906	39.986
R elbow	17.19	24.78	17.17	22.807	24.593
R wrist	24.31	30.79	26.21	36.666	34.061
L shoulder	13.95	7.831	7.381	22.319	86.053
L elbow	15.97	12.48	10.32	38.685	71.748
L wrist	13.83	12.85	16.94	33.270	94.166
Torso	10.46	10.35	13.03	10.781	-
R hip	8.592	9.656	10.30	11.389	9.7600
R knee	39.13	37.58	38.94	41.990	35.355
R ankle	67.78	65.11	65.61	66.908	67.048
L hip	3.780	4.496	3.052	10.356	20.343
L knee	4.679	5.555	4.427	12.765	34.211
L ankle	5.433	5.927	7.637	14.369	74.364
Overall mean	**17.930**	**18.326**	**18.370**	**26.304**	**53.025**

Table 5.7: The best splits from each model mean predicted keypoint distance from ground truth on testset 1.

	MobileNet s5	**ResNet s6**	**Full CNN s4**	**Residual CNN s4**	**MoveNet**
Head	10.076	9.479	25.137	31.247	26.649
Neck	3.803	3.571	13.157	11.711	-
R shoulder	36.426	37.022	39.155	41.634	58.86
R elbow	23.251	26.089	39.403	29.557	69.4
R wrist	34.214	38.709	47.895	48.949	86.42
L shoulder	5.835	8.748	15.371	14.456	35.76
L elbow	12.044	14.577	27.234	26.678	30.35
L wrist	13.473	16.051	40.229	40.843	87.47
Torso	9.258	9.246	11.250	14.576	-
R hip	10.591	11.659	14.541	17.455	21.702
R knee	38.047	39.144	35.924	38.955	40.102
R ankle	65.531	66.832	58.879	58.597	64.89
L hip	4.089	3.787	7.864	11.418	21.091
L knee	4.956	4.856	8.072	12.627	24.289
L ankle	4.451	4.963	7.559	16.723	18.384
Overall mean	**18.403**	**19.648**	**26.111**	**27.695**	**45.028**

Table 5.8: The best splits from each model mean predicted keypoint distance from ground truth on dataset 1.

5.4 Reliability testing on device

This section covers the reliability testing of our model's on our smartphone as well as reliability testing the application.

5.4.1 Screen drawing module

The drawing module in the smartphone app is meant as a visual aid for the system user. When testing the app, we see some bugs in the way the predicted keypoints are scaled up from the normalized model output and drawn to the screen. The module is also not the fastest; the drawing lags a bit behind before catching up when moving the phone around fast.

5.4.2 Models

During the reliability testing on our smartphone, we are seeing varying results, which corresponds to earlier discussed problems such as our datasets (section 5.1), and which shows in our models accuracy (section 5.3).

As previously mentioned, our models are trained on our dataset 2 (section 4.2.6); when testing with the person that is in the majority of the images in that set, we see that the models are making predictions that are reflecting the accuracy testing scores, and steering signals are generated accordingly.

We see that the MobileNet and ResNet based models are generally generating the correct steering signal, where as the Full CNN and Residual CNN models are not usable. Throttle input is not calculated correctly due to the lack of accuracy on the wrists and elbow keypoints for all the models.

When testing with a different person, we see that the models are struggling to make good predictions emphasising our concerns about our datasets.

Another issue discovered when testing is that if multiple people enter the frame, the models are not always ignoring the other person in the image but making a single prediction from both people.

5.4.3 Model performances on device

When testing models on the device, we are looking at the inference time, meaning the time it takes for the models to make a prediction. By looking at Table 5.9 we see how much time each of the model architectures is using to make a single prediction. The results clearly show that the MobileNet architecture has a speed advantage over the other models.

Model	Inference time
MobileNet	50 ms
ResNet	0.3 sec
CNN	5.2 sec
Residual	1.4 sec
MoveNet Thunder	100 ms[33]

Table 5.9: Inference speed of our model architectures running on the CPU of a Huawei P30 Pro, while MoveNet is Running on a Google Pixel 5 CPU.

5.5 Results

This section looks at the overall results from model testing and testing the smartphone application.

5.5.1 Overall model evaluation

We have created four different models with significantly different architectures in the span of this project; one MobileNetV2 based architecture, one ResNet50 based architecture, a Fully convolutional model, and a residual model that can be described as a toy/mini version of ResNet. After training and testing the models, our MobileNetV2-based architecture outperforms all the other models in speed (Table 5.9) and accuracy (Table 5.3).

However, we see a significant problem with all of our models when looking at the difference in accuracy between the left and right sides of the person in the images. We observe that the left side is more precise than the right by a substantial margin.

Even though the MobileNet-based model is the best performer, we still chose to implement all the different models from all the architectures into our smartphone application.

5.5.2 Overall smartphone application evaluation

Overall, the smartphone application performs well and does what it is intended to do. One of the primary functionalities the application is currently missing; the ability to send the steering signal to a vehicle, as discussed in section 4.4. The app itself is performing well but there are still improvements and application optimization that can be done for speed gains as we are discussing in section 6.3.

Chapter 6

Discussion

This chapter discusses the project results, if the project has answered the research questions, and what can be done with further development. Section 6.1 looks at the results from Chapter 5. Next is section 6.2 which looks at the research questions set in the beginning of the project. Lastly is section 6.3 discussing what can be done in further development of the project.

6.1 Results discussion

This section discusses the results from chapter 5, starting with subsection 6.1.1 looking at our datasets. Next is subsection 6.1.2 looking at the results of our models. Lastly is subsection 6.1.3 looking discussing our implementation of the smartphone application.

6.1.1 Datasets

We chose to create custom datasets to train our machine learning models for this project. The data collection process is based on BO20G38's approach of using a greenscreen to mass-produce data [31]. We chose to develop our dataset because we wanted the flexibility of having control over how the data is made through the entire pipeline, such that we can data augment the data as we see fit.

To label, our approach differs from BO20G38's as they were dealing with classification; we are dealing with a keypoint regression task. Therefore we looked into existing labeling tools but ultimately decided to create two proprietary labeling tools. Firstly developing a web-based tool that did the job but was difficult to modify. Because of that, we developed a second tool made in Python, which proved to be faster and more flexible.

Initially, we created our dataset 1 (section 4.2.6), which is a simple dataset that was mainly used to see if the approach would work for this project. The dataset proved that the approach was a valid method of data collection. However, the set itself was too specific, meaning that there was little variation due to having one person in a single outfit. We deemed the dataset not usable for training because of the mentioned issues, and it was migrated to be a testset, the set is not used for training any of the final models for this project.

To try to fix the variation issue, we created dataset 2 (section 4.2.6), which only contains images of 2 people, but this time with different outfits and camera angles. Dataset 2 is not

optimal as there are only two people present, making the data biased towards the subjects compared to a dataset such as COCO (section 2.5.1).

The final set of data we created was Testset 1 (section 4.2.6) for model evaluation in conjunction with our dataset 1. The set shares its simplicity with dataset 1, but it also contains poses of varying difficulty.

Even though we can mass-produce data with the approach that we used for this project, there are some apparent issues with the dataset that was created, where the primary issue being the lack of diversity in people being present in the datasets, as we struggled to find volunteers for our data collection process.

6.1.2 Models

We observe a clear difference in the left and right sides of the body when the model is predicting keypoints. We believe that the issue stems from the lack of diversity in our data and the general similarity of the left and right sides of the body. In retrospect, we should also have trained our models with the COCO dataset to see if there would be a substantial difference between training on our datasets and COCO.

MobileNet

As previously stated, MobileNet is a robust model architecture that is designed to run on mobile devices, and that shows in our precision testing as well (section 5.3) as our MobileNet architecture is our best performing model both precision and speed wise. As this is the model that shows the most potential for working in a fully developed system, we would choose to focus on this model primarily. After the testing of the model we deemed it the best out of the four made in this project, where as it is the fastest and the most precise.

ResNet

Our ResNet architecture was the one that we initially thought would be the best performing model precision-wise, but not speed-wise, as it is a large and complex model architecture. After testing the model both with test data and reliability testing we deemed the model usable but not in a environment where high speed is a big concern.

Full CNN

Initially, we thought that the Full CNN architecture would be a good valid option, but testing the precision (section 5.3 and Table 5.7) and speed (section 5.4.3) of the model we found the model not to be as precise as the other options as well as being a slow model. After the testing on test data and reliability testing we deemed the model not usable as the inference time is so slow and the precision is to low.

Residual

Our Residual CNN model did not perform as we had hoped. The precision of the model is lacking, and it is also too slow when making a prediction. Therefore, the model was used as an exercise to learn more about TensorFlow's Functional API[1] for creating more-complex

[1] https://www.tensorflow.org/guide/keras/functional

model architectures. After the testing on test data and reliability testing we deemed the model not usable as the inference time is so slow and the precision is to low.

Known issues

As mentioned in section 5.4, the models struggle if there are multiple people in the image frame. What can be done with this issue will be further discussed in section 6.3. It also seems to be an issue with the drawing module of the smartphone app where we are not getting the keypoints scaled to the correct screen size; therefore, the keypoint is drawn at a different screen resolution than the actual resolution is.

While testing, we also found that the optimal distance for the models to make a prediction is 2-4 meters, with decreasing precision beyond that.

6.1.3 Smartphone application

For this project, we created a purpose-built smartphone application based on Androids *Getting started with CameraX* Guide[2], to handle loading machine learning models, capture images, run inference on those images with our machine learning models, draw the keypoint predictions to the screen, and generate steering signals through calculating angles of selected joints. From the results in chapter 5, we see that the application does perform in the intended manner but is missing one prominent feature, namely the passing steering signals to a vehicle. As stated earlier in section 3.3.4 if there was available time, we would have looked into an actual implementation of sending signals, but as time was limited, we chose to explore the different options but not to implement it (section 4.4.5).

As mentioned in section 4.4, we are using Android's ImageAnalysis module, which captures images directly to the CPU instead of having to save an image to disk, as well as running our inference on the CPU of the phone. We are potentially leaving performance on the table by following this CPU-based approach, as many new smartphones today have a GPU.

The reliability testing of the application showed a great improvement to the signal interpretation from the previous work done by BO20G38[31], were we are able to generate the steering signals quicker and more accurately.

[2] https://developer.android.com/codelabs/camerax-getting-started

6.2 Goals

In this section we will discuss the research questions that was set at the beginning of the project in section 1.1.1.

6.2.1 RQ 1 What type neural network architecture yields the best keypoint detection precision for human pose estimation?

Having conducted architecture searches and hyperparameter tuning and exploring existing human pose estimation models using real-time image/video -streams, we have developed a lightweight solution utilizing MobileNet as the head of a Convolutional neural network architecture that has been tweaked and tuned over time. The MobileNetV2-based architecture shows that there are some issues with the training data. However, comparing the architecture presented in this project to an existing solution (MoveNet) on our testsets shows good potential in the presented MobileNet-based model created for this project.

6.2.2 RQ 1.1 Which neural network architecture yields the best performance in terms of precision and speed?

Through testing the models that were made for this project, we found that our MobileNet-based convolutional neural network model yields the best performance for both precision and speed, with a inference time of 50 ms and overall the best precision scores of our models. Nevertheless, it is essential to note that there are models such as MoveNet that are trained on more diverse data, and if tested in such an environment, we are confident that it would out-preform the models made in this project as they currently stand.

6.2.3 RQ 2 How close to real-time can a keypoint detection model run on a smartphone?

During the testing of our models, we see that our MobileNet-based convolutional neural network architecture manages to perform inference on our Huawei P30 Pro in 50 ms, giving it a frame rate of 20; this means that the MobileNet-based model created in this project outputs 20 predictions per second.

6.2.4 RQ 3 What methods can be used for sending steering signals from a smartphone to a vehicle?

As previously discussed in section 4.4.5, we have researched four different means of sending steering signals to a vehicle; API service, Firebase Real-time database, Bluetooth data transfer, and cable connection. However, we have not concluded which of the approaches is the superior one as we have not been able to test them, and the connection method would depend on the type of vehicle it would be implemented on.

6.3 Further development

This section covers potential improvements that can be done to the different modules of the projects in further development.

6.3.1 Datasets

As previously stated, our datasets are not optimal. However, it proves to be an efficient approach to quickly generate a large amount of data. To further improve our datasets, we would primarily collect images of more people, add new annotations for confidence scores for each keypoint, and a method of isolating a person in the image if there were to be multiple people in a frame; this could be bounding boxes or some type of image segmentation.

The pipeline for processing the data also has potential for further optimization and development; examples of this would be the following:

- Integrating Background swapping into labeling tool.
- Integrating data augmentation into labeling tool.
- Visualizing data augmentation parameters into labeling tool.
- User interface optimization/redesign.

6.4 Models

The models made for this project show potential for further improvement. The main issue for the models is the data, but also that most of them are relatively slow when running on a phone. To further develop the models, we would suggest the following:

- Taking measures to speed up inference time of models such as the Residual CNN model, ResNet model, and Full CNN model.
- Run hyperparameter tuning for new datasets.
- Consistent use of datasets during architecture searches.

6.5 Smartphone application

Even though the smartphone application is running as intended for the boundaries set for this project, some apparent flaws with the app can be improved upon.

The primary feature to be added to the app is the ability to send steering signals to a vehicle.

Further, the application could utilize the GPU of a phone if there is a GPU available to speed up the inference time.

Lastly, a general code clean-up for optimizing the app to be even faster than it is today.

6.6 Added features

Some safety features should be implemented for this system to be entirely usable in a completely uncontrolled environment. This section discusses some of the potential features that can be added to the system to make it safer for both the driver and the vehicle and a quality of life function like a ”follow me” feature.

6.6.1 System logging

Logging is a must-have tool for seeing what the system is doing, especially if it does something unexpected. A logging feature would give a driver or developer the means to diagnose the system in case of a malfunction and further improve the system. Some key things that can be logged are; pose estimation at intervals, vehicle connection, and inference time for checking system stability.

6.6.2 Obstacle detection and avoidance

Since the price of a vehicle can get quite expensive, there should be a safety module for obstacle avoidance. An example of this would be if a soldier and a UGV are in a forest, the vehicle should be able to steer away from oncoming obstacles if detected, but it would also have to return to the given path after the obstacle is avoided.

6.6.3 Driver safety

For the system to be safe there has to be some sort of driver safety module that looks at different scenarios. We suggest the following driver safety features:

- Minimum distance to driver. A safety function to make sure the driver is never run over.
- System arming. A feature that looks for a specific pose or gesture to arming and disarming the system.

6.6.4 Follow me

The current system implementation always requires some driver input for steering a vehicle. To make the system hands-free, a follow-me feature could be implemented. This feature would let the user signal the vehicle to follow them and stop following them. This feature would make the system genuinely hands-free; the user’s responsibility then would be to walk in a route that the vehicle would be able to drive. In addition to the follow-me feature, the system could be running the obstacle detection and avoidance module (section 6.6.2) to make sure it would not crash.

Chapter 7

Conclusion

This project aimed to continue the research from the previously written bachelor thesis written by Leistad and colleagues[31] by improving the system with their findings and implementing them on a smartphone. To accomplish the goals set for the project, we started researching human pose estimation as it would be a vital element of the system and getting a deeper understanding of the research field. We quickly discovered there are numerous research papers on the subject with many different approaches to the problem of human pose estimation.

Next, we had to research what type of data is available or what tools can be used for labeling custom data. We found labeling tools like makesense.ai and pre-made datasets like COCO, which are commonly used in the community.

Ultimately, we decided to make our own datasets and labeling tools to completely control the data. To make the data, we adopted BO20G38's greenscreen image capture approach for generating large quantities of data from a small set of data. We developed two separate data labeling tools, the first one being web-based and the second one being Python-based, running locally.

In the end, we had one training set that we used and two sets for testing. All the sets have flaws with a lack of subject diversity. The dataset used in training contains only two people, and the test data contains images of one person. These issues are reflected in the machine learning models' performance when seeing new people.

We developed four machine learning models with different architectures for the project through architecture searches and hyperparameter tuning using KerasTuner. From running the searches, we ended up with four architectures. Our MobileNetV2-based model architecture is the top performer with a mean pixel distance from ground truth of 17.930 on our testset 1 and 18.403 on dataset 1. In terms of speed, the inference time for the model is 50 ms running on the CPU of our Huawei P30 Pro in our smartphone application.

To run our human pose estimation model, we had to develop a smartphone application to handle image capture, model inference, and signal interpretation. The app was developed as a native Android application using Kotlin. The app does miss the feature of sending a steering signal to a vehicle as we could not implement it due to time constraints. However, the feature was discussed in section 4.4.5.

During the reliability testing of the machine learning models running in our smartphone app, we did find the issues that were brought up with data diversity which caused our models not to perform as hoped when seeing new people. We also discovered that the keypoint drawing module of the smartphone application is not as fast as it should be.

There is also a bug with the scaling to screen size of the keypoint from the normalized output from our models.

We believe that more time should be invested in the image capture phase to improve the model performance. Getting more subjects is essential to create a better dataset that the models can use to train and become more precise and better at generalizing. We would also want to implement the feature for sending steering signals to a vehicle, as this is an essential feature for the system to be operational. We would also suggest implementing system-wide logging for diagnostics to enhance the system's quality. Finally, we would like to add safety features for both the vehicle and the vehicle operators with obstacle avoidance and minimum distance to driver features. However, as the vehicle autonomy is outside this project's scope, we only mention them as possible improvements for a fully developed system.

We conclude that our project serves as an excellent next step from BO20G38's approach, showing significant improvements in model accuracy in a live environment, quicker model inference times, and more consistent steering signal interpretation. The MobileNet-based architecture presented in this project shows great potential as a starting foundation for building simple pose estimation models for steering autonomous vehicles as well as outperforming MoveNet Thunder in our testing. Even though the models created are sub-optimal, the project serves as a good foundation for building a final system.

Bibliography

[1] F. Lundh, *An introduction to tkinter*, http://www.tcltk.co.kr/files/TclTk_Introduction_To_Tkiner.pdf, 1999.

[2] P. F. Felzenszwalb and D. P. Huttenlocher, "Pictorial structures for object recognition," *International journal of computer vision*, vol. 61, no. 1, pp. 55–79, 2005.

[3] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[4] I. G. Maglogiannis, *Emerging artificial intelligence applications in computer engineering: real word ai systems with applications in ehealth, hci, information retrieval and pervasive technologies*. Ios Press, 2007, vol. 160.

[5] A. Dongare, R. Kharde, A. D. Kachare, *et al.*, "Introduction to artificial neural network," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 2, no. 1, pp. 189–194, 2012.

[6] M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele, "2d human pose estimation: New benchmark and state of the art analysis," in *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, 2014, pp. 3686–3693.

[7] *The opencv reference manual*, 2.4.9.0, Itseez, Apr. 2014.

[8] A. Toshev and C. Szegedy, "Deeppose: Human pose estimation via deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1653–1660.

[9] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[11] Itseez, *Open source computer vision library*, https://github.com/itseez/opencv, 2015.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[15] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," in *European conference on computer vision*, Springer, 2016, pp. 483–499.

[16] F. Chollet, *Deep Learning with Python*. Manning, Nov. 2017, ISBN: 9781617294433.

[17] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[19] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[20] Y. Chen, Z. Wang, Y. Peng, Z. Zhang, G. Yu, and J. Sun, "Cascaded pyramid network for multi-person pose estimation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[21] G. Ning, P. Liu, X. Fan, and C. Zhang, "A top-down approach to articulated human pose estimation and tracking," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018, pp. 1–6.

[22] G. Papandreou, T. Zhu, L.-c. Chen, S. Gidaris, J. Tompson, and K. Murphy, "Personlab: Person pose estimation and instance segmentation with a part-based geometric embedding model," in *ECCV*, 2018. [Online]. Available: https://arxiv.org/abs/1803.08225.

[23] S. Sahoo, "Residual blocks — building blocks of resnet," 2018. [Online]. Available: https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec.

[24] Y. Xiu, J. Li, H. Wang, Y. Fang, and C. Lu, "Pose flow: Efficient online pose tracking," *arXiv preprint arXiv:1802.00977*, 2018.

[25] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, and Q. Tian, "Centernet: Keypoint triplets for object detection," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6569–6578.

[26] W. Li, Z. Wang, B. Yin, Q. Peng, Y. Du, T. Xiao, G. Yu, H. Lu, Y. Wei, and J. Sun, "Rethinking on multi-stage networks for human pose estimation," *arXiv preprint arXiv:1901.00148*, 2019.

[27] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, *et al.*, *Kerastuner*, https://github.com/keras-team/keras-tuner, 2019.

[28] K. Sun, B. Xiao, D. Liu, and J. Wang, "Deep high-resolution representation learning for human pose estimation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 5693–5703.

[29] Y. Chen, Y. Tian, and M. He, "Monocular human pose estimation: A survey of deep learning-based methods," *Computer Vision and Image Understanding*, vol. 192, p. 102 897, 2020.

[30] A. B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte, *et al.*, *imgaug*, https://github.com/aleju/imgaug, Online; accessed 01-Feb-2020, 2020.

[31] R. s. Sjøvold Leistad, S. Lade Hellesø, S. Stafsengen Broen, and W. Svea-Lochert, "Maneuver Autonomous Vehicles with Arm Gestures," Bachelor's Thesis, Østfold University College, School of Computer Sciences, Halden, Norway, 2020.

[32] Googler, *Getting started with camerax*, https://developer.android.com/codelabs/camerax-getting-started#0, 2021.

[33] K. LeViet and Y.-h. Chen, *Pose estimation and classification on edge devices with movenet and tensorflow lite*, Jul. 2021. [Online]. Available: https://blog.tensorflow.org/2021/08/pose-estimation-and-classification-on-edge-devices-with-MoveNet-and-TensorFlow-Lite.html.

[34] Wikipedia, "Residual neural network," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Residual_neural_network.

[35] ——, "Vanishing gradient problem," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[36] F. Beletti, A. Oerlemans, Y.-H. Chen, and R. Votel. [Online]. Available: https://storage.googleapis.com/movenet/MoveNet.SinglePose%5C%20Model%5C%20Card.pdf.

[37] *Developer guides; android developers*. [Online]. Available: https://developer.android.com/guide.

[38] F. Lundh and A. Clark. [Online]. Available: https://pillow.readthedocs.io/en/stable/index.html.

[39] *Pose estimation*, https://www.tensorflow.org/lite/examples/pose_estimation/overview, Accessed: 10.08.2021.

Glossary

Batch norm Batch normalization is a method used with artificial neural networks to normalize the a layers input to make them more stable and faster (section 2.4.3. 39

Conv2D Convolutional2D is a layer definition for a convolution layer in TensorFlow. 39

data augment A means of altering data to create more data from a sample.. 69

DOPS Dropout stats, an indication if a dropout layer is present or not.. xv, 39

FFI Forsvarets forskningsinstitutt, the Norwegian defence sector's own research institution.. i

HPE Human Pose Estimation. xiii, 5, 6

inference The process of making a trained machine learning model predict on previously unseen data.. 56, 71

inference time The time it takes for a machine learning model to make a prediction.. 72

Kotlin Kotlin is the programming language that is used when developing native Android applications.. 54

MAL Model Assisted Labeling. 32, 33

MKDGT mean keypoint distance from ground truth, used to reassure a HPE models precision.. ix, 60

Python Python is a programming language, commonly used for machine learning.. 54

SepConv2D SeparableConv2D, is a deep learning convolutional layers used in TensorFlow (section 2.4.1).. 39

UGV Unmanned tracked vehicle.. xiii, 6

Code

Appendix A

C3P0-JS Code, installation, and usage

A.1 What is C3P0-JS

C3P0-JS is a web-based keypoint labeling tool for machine learning applications. It is a React service that provides a high-level interface that is easy to use and intuitive.

Link to Code on GitHub: https://github.com/wsvea-lochert/c3po-js

A.2 Installation

A.2.1 Step 1: git clone

Run the command in Figure A.1 to download the package:

Figure A.1: Command for cloning C3P0-JS from GitHub.

A.2.2 Step 2: install dependencies

Before trying to install the dependencies, make sure Node.js is installed on your system. If it is not installed follow the guide in this link: https://nodejs.org/en/

After Node.js is installed run the terminal command in Figure A.2 to install all project dependencies:

Figure A.2: Command for installing C3P0-JS dependencies.

A.3 Firebase Connection

Before you are able to use the package you have to set up a Firebase instance with Firebase Storage, authentication (Email/password sign in method) and Real-Time database. To get started with Firebase follow this link: https://firebase.google.com/

When your Firebase instances you have to get the Firebase Config, an example of how it looks can be seen in Figure A.3.

```
import firebase from 'firebase';

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
  var firebaseConfig = {
    apiKey: "AIzasdfyBbU1iASemfz7EKtaSF789_VaeXASLKKNJTdfg0",
    authDomain: "c3po-g8as2.firebaseapp.com",
    projectId: "c3poJS-g8as2",
    storageBucket: "c3poJS-e0s00.appspot.com",
    messagingSenderId: "118202056872",
    appId: "1:22143246056098:web:56ff72d56114c876c9c59g",
    measurementId: "L-7T39AKSEW"
  };

  // Initialize Firebase
  const fire = firebase.initializeApp(firebaseConfig);
  const storage = firebase.storage();
  const database = firebase.database();
  firebase.analytics();

  // export default fire;

  export {
    storage, database, fire as default
  }
```

Figure A.3: fire.js, containing what is needed for the service to connect to your Firebase instance.

This has to be saved as fire.js in *C3PO-js/src/* directory to be able to connect to Firebase.

A.4 Prepare the data

To use the tool, you first have to locate your images folder and make sure they are 1000x1000 pixels. To resize images Quantium's FireboltResizer (Appendix E.4) module can be used.

A.5 Run the tool

You are now ready to run the tool. To run the tool, run the command in Figure A.4

Figure A.4: Command for running C3P0-JS.

The service will start up and run on localhost:3000 by default, if you have another service running on that port you will automatically be prompted if you want the service to run on another port.

A.6 Start labeling

After the service is up and running you can now login, upload you data and start labeling! When the labeling process is finished go into your Firebase Console and download the data from you Real-Time database by selecting your your_dataset/dataset, and then Export JSON as seen in Figure A.5.

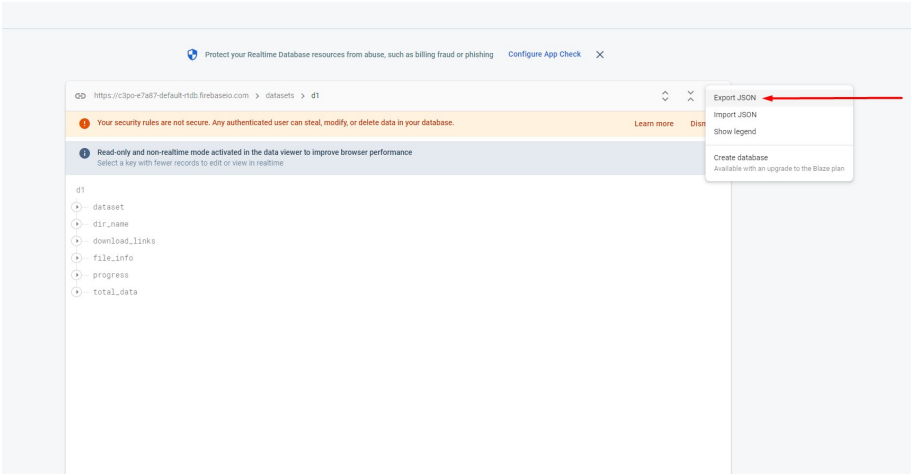


Figure A.5: Export JSON from Firebase Real-Time database.

Appendix B

Install Quantium

B.1 What is Quantium

Quantium is our primary data pre-processing and machine learning package that handles everything python related for this project.

Link to Code on GitHub: https://github.com/wsvea-lochert/Quantium

B.2 prerequisites

- Python 3.8 =<
- IDE like PyCharm or vs code
- Python venv package, to get the package run the following command in a terminal window: *pip install virtualenv*

B.3 Download

Run the command in Figure B.1 to download the package:

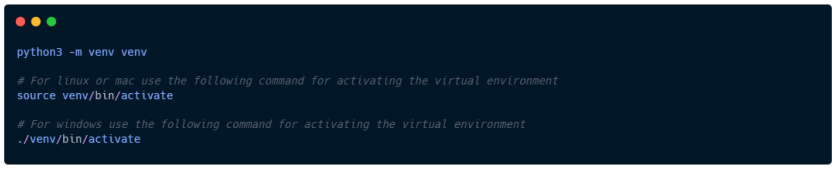
Figure B.1: Command for cloning Quantium from GitHub.

B.4 install dependencies

After you have cloned the git repo, open the project folder in your IDE, and a terminal window in the IDE. If you IDE has not created and activated a virtual environment you can create one and activate it with the command seen in Figure B.2

After your virtual environment is created and activated, run the command seen in Figure B.3 in the same terminal window you just used to activate the venv.

You are now ready to start using Quantium!



```
python3 -m venv venv

# For linux or mac use the following command for activating the virtual environment
source venv/bin/activate

# For windows use the following command for activating the virtual environment
./venv/bin/activate
```

Figure B.2: Commands for creating and activating your virtual environment

```
git clone https://github.com/wsvea-lochert/Quantium.git
```

Figure B.3: Command for installing Quantium's dependencies.

Appendix C

C3P0-Python Code, installation, and usage

C.1 What is C3P0-Python

C3P0-Python is a keypoint labeling tool for machine learning applications. It is a Python library that provides a high-level interface that is easy to use and intuitive.

Link to Code on GitHub: https://github.com/wsvea-lochert/C3P0-python

C.2 Installation

C.2.1 Step 1: git clone

Run the command in Figure C.1 to download the package:

```
git clone https://github.com/wsvea-lochert/C3P0-python.git
```

Figure C.1: Command for cloning C3P0-Python from GitHub.

C.2.2 Step2: install dependencies

If you are using a Python virtual environment, you can install the dependencies by running the command in Figure C.2 to isolate the package:

C.3 Prepare the data

To use the tool, you first have to locate your images folder and make sure they are 1000x1000 pixels. To resize images Quantium's FireboltResizer (Appendix E.4) module can be used.

After you have located you image folder add the path to *line 18* in *MainWindow.py*.

Figure C.2: Command for installing C3P0-Python dependencies.

Next create an empty JSON file and add the path to that file on *line 24* in *MainWindow.py*.

C.4 Run the tool

You are now ready to run the tool. To run the tool, run the command in Figure C.3

Figure C.3: Command for running C3P0-Python.

C.5 Keyboard shortcuts

See the available keyboard shortcuts for C3P0-Python in Table C.1

Key	Function
R	Save current pose
E	Next image
Q	Previous image
J	Next joint
L	Previous joint
G	Get previous pose
1	Move entire pose left
2	Move entire pose up
3	Move entire pose down
4	Move entire pose right
F	Get predicted pose from MoveNet
W	Move current keypoint up
S	Move current keypoint down
A	Move current keypoint left
D	Move current keypoint right
Z	Set current keypoint to 0, 0
V	Swich left and right side
T	Get next pose

Table C.1: Keyboard shortcuts for C3P0-Python

Appendix D

Filch

D.1 FilchUtils.py

```
#  Author: William Svea-Lochert
# Importing all necessary libraries
import os
import cv2
import json
import numpy as np
import pandas as pd
from PIL import Image
from tensorflow import keras
import matplotlib.pyplot as plt


def get_model(path: str):
    """
    :param path: Path to model folder/file.
    :return:
    """
    model = keras.models.load_model(path)
    return model


def get_models_from_folder(path: str):
    """
    :param path: Path to models folder.
    :return: a list of model names from folder
    """
    models = []
    for file in os.listdir(path):
        models.append(file)

    if len(models) == 0:
        raise Exception("No models found in folder.")
    else:
        print(f'Found {len(models)} models in {path}')
        return models
```

```
40
41
42 def get_pose(name: str, json_dict: dict, img_dir: str):
43     """
44     Function for getting a image and its keypoints.
45     Function Collected from: https://keras.io/examples/vision/keypoint_detection/.
46     :param name: image name
47     :param json_dict: json dictionary
48     :param img_dir: image directory
49     :return: Image data.
50     """
51     data = json_dict[name]
52     img_data = plt.imread(os.path.join(img_dir, data["image_path"]))
53
54     if img_data.shape[-1] == 4:  # If the image is RGBA convert it to RGB.
55         img_data = img_data.astype(np.uint8)
56         img_data = Image.fromarray(img_data)
57         img_data = np.array(img_data.convert("RGB"))
58     data["img_data"] = img_data
59
60     return data
61
62
63 def get_train_params(json_file, kp_def_location):
64     """"""
65     Get the training parameters from the json file.
66     :param json_file: The json file containing the training parameters.
67     :param kp_def_location: The location of the keypoint definition file.
68     :return: The training parameters.
69     """
70     with open(json_file) as infile:
71         json_dict = json.load(infile)
72
73     for i in json_dict:
74         for j in range(15):
75             x = float(json_dict[i]['joints'][j][0])
76             y = float(json_dict[i]['joints'][j][1])
77             json_dict[i]['joints'][j] = [x, y]
78
79     keypoint_def = pd.read_csv(kp_def_location)
80     keypoint_def.head()
81
82     colors = keypoint_def["Hex"].values
83     colors = ['#' + color for color in colors]
84     labels = keypoint_def["Name"].values.tolist()
85
86     samples = list(json_dict.keys())
87     return samples, json_dict, keypoint_def, colors, labels
88
89
90 def get_json_to_split(json_file: str):
91     """"""
92         Get the training parameters from the json file.
93         :param json_file: The json file containing the training parameters.
94         :return: The training parameters.
95         """
96     with open(json_file) as infile:
97         json_dict = json.load(infile)
```

```
98
99      for i in json_dict:
100         for j in range(15):
101             x = float(json_dict[i]['joints'][j][0])
102             y = float(json_dict[i]['joints'][j][1])
103             json_dict[i]['joints'][j] = [x, y]
104
105     samples = list(json_dict.keys())
106     return samples, json_dict
107
108
109 def __rename_files(path: str, name: str):
110     """
111     :param path:
112     :return:
113     """
114     file_list = os.listdir(path)
115     print(file_list)
116     for file_name in file_list:
117         os.rename(path+file_name, path+f'{name}.jpg')
118
119
120 def load_image(image_path):
121     """
122     load image and make it ready for prediction.
123     :param image_path: path to image
124     :return: np.array of image
125     """
126     img_in = cv2.imread(image_path)
127     image_color = cv2.cvtColor(img_in, cv2.COLOR_BGR2RGB)
128     image_resize = cv2.resize(image_color, (224, 224))
129     img = np.array(image_resize).reshape(-1, 224, 224, 3)
130     return img
131
132
133 def visualize_keypoints(keypoints, image_file, rot=False):
134     """
135     Function for visualizing keypoints prediction
136     :param keypoints: predicted keypoints
137     :param image_file: path to image
138     :param rot: if image should be rotated
139     :return: nothing
140     """
141     colours = ['F633FF', '00FF1B', '00FF1B', '00FF1B', '00FF1B', '00FF1B', '00FF1B', 'F633FF',
142                'FF0000', 'FF0000', 'FF0000', 'FF0000', 'FF0000', 'FF0000', 'F633FF']
143     colours = ['#' + color for color in colours]
144     fig, ax = plt.subplots()
145
146     for current_keypoint in keypoints:
147         # img = Image.fromarray(images)
148         image = plt.imread(image_file)
149         """rotate image 45 degrees to the right"""
150         # remove the next 3 lines if you don't want to rotate the image
151         if rot:
152             image = np.rot90(image, k=1, axes=(0, 1))
153             image = np.flipud(image)
154             image = np.fliplr(image)
155
```

```
156         ax.imshow(image)
157 
158         current_keypoint = np.array(current_keypoint)
159         # Since the last entry is the visibility flag, we discard it.
160         current_keypoint = current_keypoint[:, :2]
161         for idx, (x, y) in enumerate(current_keypoint):
162             # ax.scatter([x*12.214], [y*16.285], c=colours[idx],
163                          marker="x", s=50, linewidths=5)
164             ax.scatter([x], [y], c=colours[idx], marker="x", s=50, linewidths=5)
165             # ax.scatter([x*2.23], [y*2.23], c=colours[idx],
166                          marker="x", s=50, linewidths=5)
167 
168     plt.tight_layout(pad=2.0)
169     plt.show()
```

Listing D.1: FilchUtils.py, Python script containing functionality used by multiple modules in Quantium 4.1

Appendix E

Firebolt

E.1 FireboltUtils.py

```
1
2  import json
3  from imgaug.augmentables import Keypoint, KeypointsOnImage
4
5
6  def get_json_dict(input_json: str):
7      """
8      Function for getting the json dictionary.
9      :param input_json:
10     :return: A json dictionary containing all image names and their keypoints.
11     """
12
13     with open(input_json) as infile:
14         json_data = json.load(infile)
15
16     json_dict = {}
17
18     for i in range(len(json_data)):
19         tmp_obj = json_data['image' + str(i)]
20         y = {tmp_obj['image']: {'image_path': tmp_obj['image'],
21             'joints': [[tmp_obj['head']['x'], tmp_obj['head']['y']],
22                        [tmp_obj['left_ankle']['x'], tmp_obj['left_ankle']['y']],
23                        [tmp_obj['left_elbow']['x'], tmp_obj['left_elbow']['y']],
24                        [tmp_obj['left_hip']['x'], tmp_obj['left_hip']['y']],
25                        [tmp_obj['left_knee']['x'], tmp_obj['left_knee']['y']],
26                        [tmp_obj['left_shoulder']['x'], tmp_obj['left_shoulder']['y']],
27                        [tmp_obj['left_wrist']['x'], tmp_obj['left_wrist']['y']],
28                        [tmp_obj['neck']['x'], tmp_obj['neck']['y']],
29                        [tmp_obj['right_ankle']['x'], tmp_obj['right_ankle']['y']],
30                        [tmp_obj['right_elbow']['x'], tmp_obj['right_elbow']['y']],
31                        [tmp_obj['right_hip']['x'], tmp_obj['right_hip']['y']],
32                        [tmp_obj['right_knee']['x'], tmp_obj['right_knee']['y']],
33                        [tmp_obj['right_shoulder']['x'], tmp_obj['right_shoulder']['y']],
34                        [tmp_obj['right_writs']['x'], tmp_obj['right_writs']['y']],
35                        [tmp_obj['torso']['x'], tmp_obj['torso']['y']]
36                        ]}}
37         json_dict.update(y)
38
39     return json_dict
```

```
40
41
42 def get_kpsoi(keypoints, img_shape):
43     kps = KeypointsOnImage([
44         Keypoint(x=keypoints[0][0], y=keypoints[0][1]),
45         Keypoint(x=keypoints[1][0], y=keypoints[1][1]),
46         Keypoint(x=keypoints[2][0], y=keypoints[2][1]),
47         Keypoint(x=keypoints[3][0], y=keypoints[3][1]),
48         Keypoint(x=keypoints[4][0], y=keypoints[4][1]),
49         Keypoint(x=keypoints[5][0], y=keypoints[5][1]),
50         Keypoint(x=keypoints[6][0], y=keypoints[6][1]),
51         Keypoint(x=keypoints[7][0], y=keypoints[7][1]),
52         Keypoint(x=keypoints[8][0], y=keypoints[8][1]),
53         Keypoint(x=keypoints[9][0], y=keypoints[9][1]),
54         Keypoint(x=keypoints[10][0], y=keypoints[10][1]),
55         Keypoint(x=keypoints[11][0], y=keypoints[11][1]),
56         Keypoint(x=keypoints[12][0], y=keypoints[12][1]),
57         Keypoint(x=keypoints[13][0], y=keypoints[13][1]),
58         Keypoint(x=keypoints[14][0], y=keypoints[14][1])
59     ], shape=img_shape)
60
61     return kps
```

Listing E.1: FireboltUtils.py, Python script containing functionality used by the Firebolt module.

E.2 FireboltBackground.py

```
1
2  import os
3  import cv2
4  import json
5  import uuid
6  import numpy as np
7  from tqdm import tqdm
8  from random import randint
9  from typing import Optional
10 from Firebolt.FireboltUtils import get_json_dict, get_kpsoi
11 from Filch.FilchUtils import get_pose
12
13
14 class FireboltBackground:
15     def __init__(self, img_dir: str, input_json: str, output_dir: str,
16                  output_json_path: str, bg_dir: str, blanks_dir:
17                  Optional[str] = None, num_blanks: Optional[int] = None,
18                  blank_bg: Optional[str] = None):
19         """
20         :param img_dir: directory with images
21         :param input_json: Path to the input json file
22         :param output_dir: Path to the output directory
23         :param output_json_path: Path to the output json file
24         :param bg_dir: Path to the backgrounds directory.
25         """
26         self.img_dir = img_dir
27         self.input_json = input_json
28         self.output_dir = output_dir
29         self.output_json_path = output_json_path
30         self.bg_dir = bg_dir
31         self.json_dict = get_json_dict(self.input_json)
32         self.samples = list(self.json_dict.keys())
33         self.bg_dict = {}
34         self.blanks_dir = blanks_dir
35         self.num_blanks = num_blanks
36         self.blank_bg = blank_bg
37         self.image_counter = 0
38
39     def swap(self):
40         """
41         Run all processing on images.
42         :return:
43         """
44         self.__process()
45         self.__create_blanks()
46
47         with open(self.output_json_path, 'w') as outfile:
48             json.dump(self.bg_dict, outfile, indent=2)
49
50     def __process(self):
51         """
52         Process the images to create background swapped images.
53         :return:
54         """
55
56         for sample in tqdm(self.samples):
```

```
57              data = get_pose(sample, self.json_dict, self.img_dir)
58              image = data["img_data"]
59              keypoint = data["joints"]
60              kps = get_kpsoi(keypoint, image.shape)
61
62              for bg in os.listdir(self.bg_dir):
63                  new_image_name = str(os.path.splitext(bg)[0]) + '-' + str(sample)
64                  self.__swap_bg(sample, bg, new_image_name, False)
65
66                  self.bg_dict.update({'image' + str(self.image_counter):
67                                       {'image': new_image_name,
68                                        'head': {'x': kps.keypoints[0].x,
69                                                 'y': kps.keypoints[0].y},
70                                        'left_ankle': {'x': kps.keypoints[1].x,
71                                                       'y': kps.keypoints[1].y},
72                                        'left_elbow': {'x': kps.keypoints[2].x,
73                                                       'y': kps.keypoints[2].y},
74                                        'left_hip': {'x': kps.keypoints[3].x,
75                                                     'y': kps.keypoints[3].y},
76                                        'left_knee': {'x': kps.keypoints[4].x,
77                                                      'y': kps.keypoints[4].y},
78                                        'left_shoulder': {'x': kps.keypoints[5].x,
79                                                          'y': kps.keypoints[5].y},
80                                        'left_wrist': {'x': kps.keypoints[6].x,
81                                                       'y': kps.keypoints[6].y},
82                                        'neck': {'x': kps.keypoints[7].x,
83                                                 'y': kps.keypoints[7].y},
84                                        'right_ankle': {'x': kps.keypoints[8].x,
85                                                        'y': kps.keypoints[8].y},
86                                        'right_elbow': {'x': kps.keypoints[9].x,
87                                                        'y': kps.keypoints[9].y},
88                                        'right_hip': {'x': kps.keypoints[10].x,
89                                                      'y': kps.keypoints[10].y},
90                                        'right_knee': {'x': kps.keypoints[11].x,
91                                                       'y': kps.keypoints[11].y},
92                                        'right_shoulder': {'x': kps.keypoints[12].x,
93                                                           'y': kps.keypoints[12].y},
94                                        'right_writs': {'x': kps.keypoints[13].x,
95                                                        'y': kps.keypoints[13].y},
96                                        'torso': {'x': kps.keypoints[14].x,
97                                                  'y': kps.keypoints[14].y}}
98                                       })
99                  self.image_counter += 1
100
101     def __swap_bg(self, img, background, name: str, blank: bool):
102         """
103         :param img: image name
104         :param background: background image.
105         :param blank: check if we are making blank images or not.
106         :return:
107         """
108         if blank:
109             image = cv2.imread(self.blanks_dir + img)
110             background_image = cv2.imread(self.blank_bg + background)
111         else:
112             image = cv2.imread(self.img_dir + img)
113             background_image = cv2.imread(self.bg_dir + background)
114
```

```
115         image_color = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
116
117         image_copy = np.copy(image_color)
118         image_copy = cv2.resize(image_copy, (224, 224))
119
120         lower_green = np.array([0, 100, 0])  # [R value, G value, B value]
121         upper_green = np.array([120, 255, 130])
122
123         mask = cv2.inRange(image_copy, lower_green, upper_green)
124         masked_image = np.copy(image_copy)
125         masked_image[mask != 0] = [0, 0, 0]
126
127         background_image_color = cv2.cvtColor(background_image, cv2.COLOR_BGR2RGB)
128
129         # If the background image need to be resized uncomment this line.
130         bg_size = randint(224, 600)
131         background_image_resize = cv2.resize(background_image_color, (bg_size, bg_size))
132         # Depending on the width of the background image change randint parameters
133         background_image = np.roll(background_image_resize, 3 * randint(-bg_size, bg_size))
134
135         crop_background = background_image[0:224, 0:224]
136
137         crop_background[mask == 0] = [0, 0, 0]
138
139         final_image = crop_background + masked_image
140         final_image = cv2.resize(final_image, (224, 224))
141         final_image = cv2.cvtColor(final_image, cv2.COLOR_BGR2RGB)
142
143         # write image to folder
144         cv2.imwrite(self.output_dir + name, final_image)
145
146     def __create_blanks(self):
147         """
148         This function creates blanks images.
149         :returns: nothing
150         """
151         print(f'Creating {self.num_blanks} blank images, to balance dataset.')
152         counter = 0
153         backgrounds = os.listdir(self.bg_dir)
154         blanks = os.listdir(self.blanks_dir)
155
156         for i in range(self.num_blanks):
157             bg = backgrounds[randint(0, len(backgrounds) - 1)]
158             img = blanks[randint(0, len(blanks) - 1)]
159             name = f'{uuid.uuid4()}-{img}'
160             self.__swap_bg(img, bg, name, True)
161
162             self.bg_dict.update(
163                 {'image' + f'{str(self.image_counter)}': {
164                     'image': name,
165                     'head': {'x': 0, 'y': 0},
166                     'left_ankle': {'x': 0, 'y': 0},
167                     'left_elbow': {'x': 0, 'y': 0},
168                     'left_hip': {'x': 0, 'y': 0},
169                     'left_knee': {'x': 0, 'y': 0},
170                     'left_shoulder': {'x': 0, 'y': 0},
171                     'left_wrist': {'x': 0, 'y': 0},
172                     'neck': {'x': 0, 'y': 0},
```

```
                    'right_ankle': {'x': 0, 'y': 0},
                    'right_elbow': {'x': 0, 'y': 0},
                    'right_hip': {'x': 0, 'y': 0},
                    'right_knee': {'x': 0, 'y': 0},
                    'right_shoulder': {'x': 0, 'y': 0},
                    'right_writs': {'x': 0, 'y': 0},
                    'torso': {'x': 0, 'y': 0}
                    }
                })
            counter += 1
            self.image_counter += 1
```

Listing E.2: FireboltBackground.py, Python script for swapping greenscreen backgrounds to new backgrounds. The scrip uses both Filch (Appendix D.1 and FireboltUtils.py (Appendix E.1)

E.3 FireboltImageFlipper.py

```
1
2  import json
3  import numpy as np
4  from PIL import Image
5  from tqdm import tqdm
6  import imgaug.augmenters as iaa
7  from Filch.FilchUtils import get_pose
8  from Firebolt.FireboltUtils import get_json_dict, get_kpsoi
9
10
11 class FireboltImageFlipper:
12     """
13     Class to flip images, poses and keep the original images,
14     as well as expanding the output json file.
15     """
16
17     def __init__(self, img_dir: str, input_json: str,
18                  output_dir: str, output_json_path: str):
19         """
20         Initializes the class.
21         :param img_dir: str Path to the directory containing the images.
22         :param input_json: Path to the input json file.
23         :param output_dir: Path to the image output directory.
24         :param output_json_path: Path to the output json file.
25         """
26
27         self.img_dir = img_dir
28         self.input_json = input_json
29         self.output_dir = output_dir
30         self.output_json_path = output_json_path
31         self.json_dict = get_json_dict(self.input_json)
32         self.samples = list(self.json_dict.keys())
33         self.flip_json = {}
34
35     def flip(self):
36         """
37         Flips the images, poses and keeps the original images.
38         """
39
40         self.__process()
41
42         with open(self.output_json_path, 'w') as outfile:
43             json.dump(self.flip_json, outfile, indent=2)
44
45     def __process(self):
46         """
47         Processes the images and poses.
48         """
49         counter = 0
50
51         for sample in tqdm(self.samples):
52             data = get_pose(sample, self.json_dict, self.img_dir)
53             image = data["img_data"]
54             keypoint = data["joints"]
55             kps = get_kpsoi(keypoint, image.shape)
56             rot = np.rot90(image, k=1, axes=(1, 0))
```

```
57              original = Image.fromarray(rot)
58
59              original.save(self.output_dir + sample)
60
61              self.flip_json.update({'image' + str(counter): {'image': str(sample),
62                      'head': {'x': kps.keypoints[0].x, 'y': kps.keypoints[0].y},
63                      'left_ankle': {'x': kps.keypoints[1].x, 'y': kps.keypoints[1].y},
64                      'left_elbow': {'x': kps.keypoints[2].x, 'y': kps.keypoints[2].y},
65                      'left_hip': {'x': kps.keypoints[3].x, 'y': kps.keypoints[3].y},
66                      'left_knee': {'x': kps.keypoints[4].x, 'y': kps.keypoints[4].y},
67                      'left_shoulder': {'x': kps.keypoints[5].x, 'y': kps.keypoints[5].y},
68                      'left_wrist': {'x': kps.keypoints[6].x, 'y': kps.keypoints[6].y},
69                      'neck': {'x': kps.keypoints[7].x, 'y': kps.keypoints[7].y},
70                      'right_ankle': {'x': kps.keypoints[8].x, 'y': kps.keypoints[8].y},
71                      'right_elbow': {'x': kps.keypoints[9].x, 'y': kps.keypoints[9].y},
72                      'right_hip': {'x': kps.keypoints[10].x, 'y': kps.keypoints[10].y},
73                      'right_knee': {'x': kps.keypoints[11].x, 'y': kps.keypoints[11].y},
74                      'right_shoulder': {'x': kps.keypoints[12].x, 'y': kps.keypoints[12].y},
75                      'right_writs': {'x': kps.keypoints[13].x, 'y': kps.keypoints[13].y},
76                      'torso': {'x': kps.keypoints[14].x, 'y': kps.keypoints[14].y}
77                      }
78                     })
79              counter += 1
80
81              self.__flipper(rot, kps, sample, counter)
82              counter += 1
83
84
85      def __flipper(self, image, kps, name: str, counter: int):
86          """
87          Flips the image and the keypoints.
88          :param image: The image that is to be flipped.
89          :param kps: Keypoints for the current image.
90          :param name: Name of the current image being augmented.
91          :return: New keypoints for the flipped image.
92          """
93          seq = iaa.Sequential([
94              # flip image
95              iaa.Fliplr(1.0),  # horizontally flip 100% of the images
96          ], random_order=True)
97
98          image_aug, kps_aug = seq(image=image, keypoints=kps)
99
100         new_image_name = str(f'flip-{name}')
101
102         self.flip_json.update({'image' + f'{str(counter)}': {'image': new_image_name,
103                 'head': {'x': kps.keypoints[0].x, 'y': kps.keypoints[0].y},
104                 'left_ankle': {'x': kps.keypoints[8].x, 'y': kps.keypoints[8].y},
105                 'left_elbow': {'x': kps.keypoints[9].x, 'y': kps.keypoints[9].y},
106                 'left_hip': {'x': kps.keypoints[10].x, 'y': kps.keypoints[10].y},
107                 'left_knee': {'x': kps.keypoints[11].x, 'y': kps.keypoints[11].y},
108                 'left_shoulder': {'x': kps.keypoints[12].x, 'y': kps.keypoints[12].y},
109                 'left_wrist': {'x': kps.keypoints[13].x, 'y': kps.keypoints[13].y},
110                 'neck': {'x': kps.keypoints[7].x, 'y': kps.keypoints[7].y},
111                 'right_ankle': {'x': kps.keypoints[1].x, 'y': kps.keypoints[1].y},
112                 'right_elbow': {'x': kps.keypoints[2].x,  'y': kps.keypoints[2].y},
113                 'right_hip': {'x': kps.keypoints[3].x, 'y': kps.keypoints[3].y},
114                 'right_knee': {'x': kps.keypoints[4].x, 'y': kps.keypoints[4].y},
```

```
115                     'right_shoulder': {'x': kps.keypoints[5].x, 'y': kps.keypoints[5].y},
116                     'right_writs': {'x': kps.keypoints[6].x, 'y': kps.keypoints[6].y},
117                     'torso': {'x': kps.keypoints[14].x, 'y': kps.keypoints[14].y}
118                     }
119                 })
120 
121         save_img = Image.fromarray(image_aug)
122         save_img.save(f'{self.output_dir}{new_image_name}')
```

Listing E.3: FireboltImageFlipper.py, Python script for flipping images and keypoints horizontally. The scrip uses both Filch (Appendix D.1 and FireboltUtils.py (Appendix E.1)

E.4 FireboltResizer.py

```
1
2  import os
3  from PIL import Image
4
5
6  class FireboltResizer:
7      def __init__(self, directory, output, size):
8          self.directory = directory
9          self.output = output
10         self.size = size
11
12     def resize(self):
13         images = os.listdir(self.directory)
14         for image in images:
15             img = Image.open(self.directory + '/' + image)
16             resized = img.resize((self.size, self.size))
17             resized.save(self.output + '/' + image)
```

Listing E.4: FireboltResizer.py, Python script for resizing images in a directory.

E.5 FireboltDatasetCreator.py

```
1
2  import json
3  from tqdm import tqdm
4  from PIL import Image
5  import imgaug.augmenters as iaa
6  from Filch.FilchUtils import get_pose
7  from Firebolt.FireboltUtils import get_json_dict, get_kpsoi
8
9
10 class FireboltDatasetCreator:
11     """
12     FireboltDataset class for creating final json file for dataset.
13     """
14
15     def __init__(self, img_dir: str, input_json: str,
16                  output_dir: str, output_json_path: str):
17         """
18
19         :param img_dir: Path to image directory.
20         :param input_json: Path to input json file.
21         :param output_dir: Path to output directory.
22         :param output_json_path: Path to output json file.
23         """
24         self.img_dir = img_dir
25         self.input_json = input_json
26         self.output_dir = output_dir
27         self.output_json_path = output_json_path
28         self.json_dict = get_json_dict(self.input_json)
29         self.samples = list(self.json_dict.keys())
30         self.train_json = {}
31
32     def create(self, augment: bool = False):
33         """
34         Creates the dataset.
35         :param augment: Boolean to augment or not.
36         :return:
37         """
38         self.__process(augment)
39
40         with open(self.output_json_path, 'w') as outfile:
41             json.dump(self.train_json, outfile, indent=2)
42
43     def __process(self, aug: bool):
44         """
45         Processes the dataset.
46         :param aug: Boolean to augment or not.
47         """
48         for sample in tqdm(self.samples):
49             data = get_pose(sample, self.json_dict, self.img_dir)
50             image = data["img_data"]
51             keypoint = data["joints"]
52             kps = get_kpsoi(keypoint, image.shape)
53
54             if aug:
55                 seq = iaa.Sequential([
56                     iaa.Sometimes(
```

```
                        0.5,
                        iaa.Multiply((0.90, 1.10)),
                        iaa.Affine(rotate=(-20, 20),
                                   translate_px={"x": [-30, 30],
                                   "y": [-30, 30]}, shear=(-10, 10),
                                   scale=(0.80, 1.3))
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.imgcorruptlike.MotionBlur(severity=1)
                        #     iaa.GaussianBlur(sigma=(0, 0.6))
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.imgcorruptlike.Snow(severity=1)
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.Rain(drop_size=(0.10, 0.20))
                    ),

                ], random_order=True)
                image_aug, kps_aug = seq(image=image, keypoints=kps)
                save_img = Image.fromarray(image_aug)
                save_img.save(f'{self.output_dir}{sample}')

                for i in range(15):  # Check if the keypoints are out of bounds.
                    if kps.keypoints[i].x == 0 and kps.keypoints[i].y == 0:
                        kps_aug.keypoints[i].x = 0
                        kps_aug.keypoints[i].y = 0

                    if kps_aug.keypoints[i].x < 0 or kps_aug.keypoints[i].y < 0:
                        kps_aug.keypoints[i].x = 0
                        kps_aug.keypoints[i].y = 0

                    if kps_aug.keypoints[i].x > 224 or kps_aug.keypoints[i].y > 224:
                        kps_aug.keypoints[i].x = 0
                        kps_aug.keypoints[i].y = 0

                self.__update_json(kps_aug, sample)

            else:
                save_img = Image.fromarray(image)
                save_img.save(f'{self.output_dir}{sample}')

                for i in range(15):  # Check if the keypoints are out of bounds.
                    if kps.keypoints[i].x == 0 and kps.keypoints[i].y == 0:
                        kps.keypoints[i].x = 0
                        kps.keypoints[i].y = 0

                    if kps.keypoints[i].x < 0 or kps.keypoints[i].y < 0:
                        kps.keypoints[i].x = 0
                        kps.keypoints[i].y = 0

                    if kps.keypoints[i].x > 224 or kps.keypoints[i].y > 224:
                        kps.keypoints[i].x = 0
                        kps.keypoints[i].y = 0

```

```
115             self.__update_json(kps, sample)
116
117     def __update_json(self, kps, sample):
118         """
119         Updates the json dictionary.
120         :param kps: The keypoints for the current image
121         :param sample: The name of the current image.
122         :return:
123         """
124         self.train_json.update({sample:
125                                 {'image_path': sample,
126              'joints':
127                 [[kps.keypoints[0].x, kps.keypoints[0].y],
128                 [kps.keypoints[1].x, kps.keypoints[1].y],
129                 [kps.keypoints[2].x, kps.keypoints[2].y],
130                 [kps.keypoints[3].x, kps.keypoints[3].y],
131                 [kps.keypoints[4].x, kps.keypoints[4].y],
132                 [kps.keypoints[5].x, kps.keypoints[5].y],
133                 [kps.keypoints[6].x, kps.keypoints[6].y],
134                 [kps.keypoints[7].x, kps.keypoints[7].y],
135                 [kps.keypoints[8].x, kps.keypoints[8].y],
136                 [kps.keypoints[9].x, kps.keypoints[9].y],
137                 [kps.keypoints[10].x, kps.keypoints[10].y],
138                 [kps.keypoints[11].x, kps.keypoints[11].y],
139                 [kps.keypoints[12].x, kps.keypoints[12].y],
140                 [kps.keypoints[13].x, kps.keypoints[13].y],
141                 [kps.keypoints[14].x, kps.keypoints[14].y]
142                 ]}
143
144                                  }
145                                 )
```

Listing E.5: FireboltDatasetCreator.py, Python script creating the final JSON output file for our datasets. The script also augments the images of the set. The scrip uses both Filch (Appendix D.1 and FireboltUtils.py (Appendix E.1)

E.6 FireboltDataSplitter.py

```
1
2  import os
3  import json
4  from colorama import Fore
5  from Filch.FilchUtils import get_json_to_split
6
7
8  class FireboltDataSplitter:
9      def __init__(self, json_path, out_dir):
10         """
11         Initialize the class.
12         :param json_path: path to big json file.
13         :param out_dir: path to output directory where the files will be saved.
14         """
15         self.json_path = json_path
16         self.out_dir = out_dir
17         self.samples, self.json_dict = get_json_to_split(self.json_path)
18         self.split_value = int(len(self.samples))/8
19         self.split1, self.split2, self.split3, self.split4, self.split5,
20         self.split6, self.split7, self.split8 = {}, {}, {}, {}, {}, {}, {}, {}
21
22     def split_data(self):
23         """
24         Split the data into 8 parts then run the
25         __dump_json method to save the data into files.
26         :return:
27         """
28         print(Fore.GREEN, f'Split value: {self.split_value}\n'
29               f'Number of samples: {len(self.samples)}')
30
31         for i in range(len(self.samples)):
32             if i < self.split_value:
33                 self.split1[self.samples[i]] = self.json_dict[self.samples[i]]
34             elif self.split_value * 1 < i <= self.split_value * 2:
35                 self.split2[self.samples[i]] = self.json_dict[self.samples[i]]
36             elif self.split_value * 2 < i <= self.split_value * 3:
37                 self.split3[self.samples[i]] = self.json_dict[self.samples[i]]
38             elif self.split_value * 4 >= i > self.split_value * 3:
39                 self.split4[self.samples[i]] = self.json_dict[self.samples[i]]
40             elif self.split_value * 4 < i <= self.split_value * 5:
41                 self.split5[self.samples[i]] = self.json_dict[self.samples[i]]
42             elif self.split_value * 5 < i <= self.split_value * 6:
43                 self.split6[self.samples[i]] = self.json_dict[self.samples[i]]
44             elif self.split_value * 6 < i <= self.split_value * 7:
45                 self.split7[self.samples[i]] = self.json_dict[self.samples[i]]
46             elif self.split_value * 8 >= i > self.split_value * 7:
47                 self.split8[self.samples[i]] = self.json_dict[self.samples[i]]
48
49         print(Fore.CYAN, f'Number of samples in split 1: {len(self.split1)}
50                 \nNumber of samples in split 2: {len(self.split2)}
51                 \nNumber of samples in split 3: {len(self.split3)}
52                 \nNumber of samples in split 4: {len(self.split4)}')
53         print(Fore.CYAN, f'Number of samples in split 5: {len(self.split5)}
54                 \nNumber of samples in split 6: {len(self.split6)}
55                 \nNumber of samples in split 7: {len(self.split7)}
56                 \nNumber of samples in split 8: {len(self.split8)}')
```

```
57
58          self.__dump_json()
59
60      def __dump_json(self):
61          """
62          Dump the data into files.
63          :return:
64          """
65          with open(f'{self.out_dir}train_split1.json', 'w') as outfile:
66              json.dump(self.split1, outfile, indent=2)
67
68          with open(f'{self.out_dir}train_split2.json', 'w') as outfile:
69              json.dump(self.split2, outfile, indent=2)
70
71          with open(f'{self.out_dir}train_split3.json', 'w') as outfile:
72              json.dump(self.split3, outfile, indent=2)
73
74          with open(f'{self.out_dir}train_split4.json', 'w') as outfile:
75              json.dump(self.split4, outfile, indent=2)
76
77          with open(f'{self.out_dir}train_split5.json', 'w') as outfile:
78              json.dump(self.split5, outfile, indent=2)
79
80          with open(f'{self.out_dir}train_split6.json', 'w') as outfile:
81              json.dump(self.split6, outfile, indent=2)
82
83          with open(f'{self.out_dir}train_split7.json', 'w') as outfile:
84              json.dump(self.split7, outfile, indent=2)
85
86          with open(f'{self.out_dir}train_split8.json', 'w') as outfile:
87              json.dump(self.split8, outfile, indent=2)
```

Listing E.6: FireboltDataSplitter.py, Python script for splitting a JSON file into 8 new JSON files. The module uses Filch D.1

Appendix F

Dobby

F.1 DobbyAugmenter.py

```
1
2 import imgaug.augmenters as iaa
3
4
5 def get_augmentation_parameters():
6     """
7     :return: Returns the augmentation parameters for the DobbyAugmentert.
8     """
9     return iaa.Sequential([
10                     iaa.Sometimes(
11                         0.5,
12                         iaa.Multiply((0.90, 1.10)),
13                         iaa.Affine(rotate=(-20, 20),
14                                    translate_px={"x": [-30, 30], "y": [-30, 30]},
15                                    shear=(-10, 10),
16                                    scale=(0.80, 1.3))
17                     ),
18                     iaa.Sometimes(
19                         0.1,
20                         iaa.imgcorruptlike.MotionBlur(severity=1)
21                         #     iaa.GaussianBlur(sigma=(0, 0.6))
22                     ),
23                     iaa.Sometimes(
24                         0.1,
25                         iaa.imgcorruptlike.Snow(severity=1)
26                     ),
27                     iaa.Sometimes(
28                         0.1,
29                         iaa.Rain(drop_size=(0.10, 0.20))
30                     ),
31                     iaa.Sometimes(
32                     0.1,
33                     iaa.GaussianBlur(sigma=(0, 0.6))
34                     ),
35                     iaa.Sometimes(
36                     0,1
37                     iaa.LinearContrast((0.75, 1.5)),
38                     )
39                     iaa.Sometimes(
```

```
                    0,1
                    iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05 * 255),
                                              per_channel=0.1)
                    )

                ], random_order=True)


def get_validation_augmentation_parameters():
    """
    :return: Returns the augmentation parameters for the DobbyAugmentert.
    """
    return iaa.Sequential([
                    iaa.Sometimes(
                        0.5,
                        iaa.Multiply((0.90, 1.10)),
                        iaa.Affine(rotate=(-20, 20),
                                   translate_px={"x": [-30, 30], "y": [-30, 30]},
                                   shear=(-10, 10),
                                   scale=(0.80, 1.3))
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.imgcorruptlike.MotionBlur(severity=1)
                        #     iaa.GaussianBlur(sigma=(0, 0.6))
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.imgcorruptlike.Snow(severity=1)
                    ),
                    iaa.Sometimes(
                        0.1,
                        iaa.Rain(drop_size=(0.10, 0.20))
                    ),

                ], random_order=True)
```

Listing F.1: DobbyAugmenter.py, Python script for returning imgaug augmentation parameters.

F.2 DobbyDataset.py

```python
import gc
import numpy as np
from typing import Optional
from tensorflow import keras
from Filch.FilchUtils import get_pose
from imgaug.augmentables.kps import KeypointsOnImage, Keypoint


class DobbyDataset(keras.utils.Sequence):
    """
    A class that represents a dataset of keypoints.
    """
    def __init__(self, image_keys, aug, json_dict, image_dir: str,
                 batch_size: Optional[int] = 32, train: Optional[bool] = True):
        """
        Initializes a DobbyDataset object.
        :param image_keys: Names of the images
        :param aug: Augmentation pipeline. expecting iaa.Sequential()
        :param json_dict: Dictionary containing the keypoints and image names.
        :param image_dir: Path to image directory
        :param batch_size: Batch size for the dataset.
        :param train: If it is a training set or not.
        """
        self.image_keys = image_keys
        self.aug = aug
        self.batch_size = batch_size
        self.train = train
        self.image_dir = image_dir
        self.json_dict = json_dict
        self.on_epoch_end()

    def __len__(self):
        return len(self.image_keys) // self.batch_size

    def on_epoch_end(self):
        self.indexes = np.arange(len(self.image_keys))
        if self.train:
            np.random.shuffle(self.indexes)
            gc.collect()
            keras.backend.clear_session()

    def __getitem__(self, index):
        indexes = self.indexes[index * self.batch_size: (index + 1) * self.batch_size]
        image_keys_temp = [self.image_keys[k] for k in indexes]
        (images, keypoints) = self.__data_generation(image_keys_temp)

        return images, keypoints

    def __data_generation(self, image_keys_temp):
        batch_images = np.empty((self.batch_size, 224, 224, 3), dtype="int")
        batch_keypoints = np.empty(
            (self.batch_size, 1, 1, 30), dtype="float32"
        )

        for i, key in enumerate(image_keys_temp):
            data = get_pose(key, self.json_dict, self.image_dir)
```

```
57              current_keypoint = np.array(data["joints"])[:, :2]
58              kps = []
59
60              # To apply our data augmentation pipeline, we first need to
61              # form Keypoint objects with the original coordinates.
62              for j in range(0, len(current_keypoint)):
63                  kps.append(Keypoint(x=current_keypoint[j][0], y=current_keypoint[j][1]))
64
65              # We then project the original image and its keypoint coordinates.
66              current_image = data["img_data"]
67              kps_obj = KeypointsOnImage(kps, shape=current_image.shape)
68
69              # Apply the augmentation pipeline.
70              (new_image, new_kps_obj) = self.aug(image=current_image, keypoints=kps_obj)
71              batch_images[i,] = new_image
72
73              # Parse the coordinates from the new keypoint object.
74              kp_temp = []
75              for keypoint in new_kps_obj:
76                  kp_temp.append(np.nan_to_num(keypoint.x))
77                  kp_temp.append(np.nan_to_num(keypoint.y))
78
79              # More on why this reshaping later.
80              batch_keypoints[i,] = np.array(kp_temp).reshape(1, 1, 15 * 2)
81
82          # Scale the coordinates to [0, 1] range.
83          batch_keypoints = batch_keypoints / 224
84
85          return batch_images, batch_keypoints
```

Listing F.2: DobbyDataset.py, Python script for returning dataset to training function. The class inherits from Keras.utils.Sequence, the module uses Filch (Appendix D.1)

F.3 DobbyDelivery.py

```python
import numpy as np
from Filch.FilchUtils import get_pose
from Dobby.DobbyDataset import DobbyDataset
from Filch.FilchUtils import get_train_params
from Dobby.DobbyAugmenter import get_augmentation_parameters,
                                 get_validation_augmentation_parameters
from colorama import Fore

class DobbyDelivery:
    def __init__(self, json: str, kp_definitions: str, img_dir: str):
        self.json = json
        self.kp_def = kp_definitions
        self.img_dir = img_dir
        self.images = []
        self.keypoints = []
        self.samples, self.json_dict, self.kp_def, self.colors,
        self.labels = get_train_params(self.json, self.kp_def)

        self.train_aug = get_augmentation_parameters()
        self.validation_aug = get_validation_augmentation_parameters()
        self.__get_images_and_keypoints()
        self.train_set, self.val_set = self.__deliver_datasets()

    def __get_images_and_keypoints(self):
        for sample in self.samples:
            data = get_pose(sample, self.json_dict, self.img_dir)
            image = data['img_data']
            keypoint = data['joints']

            self.images.append(image)
            self.keypoints.append(keypoint)

    def __deliver_datasets(self):
        np.random.shuffle(self.samples)
        train_keys, val_keys = (
            self.samples[int(len(self.samples) * 0.20):],
            self.samples[: int(len(self.samples) * 0.20)],
        )
        return DobbyDataset(train_keys, self.train_aug,
                            self.json_dict, self.img_dir, 32, True),
               DobbyDataset(val_keys, self.validation_aug,
                            self.json_dict, self.img_dir, 32, False)

    def __print_data_info(self):
        print(Fore.GREEN, f'Dataset delivered by DobbyDelivery service,
              current img_dir: {self.img_dir}, current json: {self.json}')
        print(Fore.CYAN, f"Total batches in training set: {len(self.train_set)}")
        print(Fore.CYAN, f"Total batches in validation set: {len(self.val_set)}")
```

Listing F.3: DobbyDelivery.py, Python script responsible to collect the dataset from DobbyDataset(Appendix F.2), and deliver it to the training session.

F.4 DobbyTrainer.py

```python
from typing import Optional
from tensorflow import keras
from Dobby.DobbyDelivery import DobbyDelivery
from keras.callbacks import TensorBoard, EarlyStopping, ReduceLROnPlateau, ModelCheckpoint


class DobbyTrainer:
    def __init__(self, json: str, kp_def: str, images: str, checkpoint_dir: str,
                 log_dir: str, save_dir: str, name: str, model,
                 epochs: Optional[int] = 50, stopping_patience: Optional[int] = 3,
                 lr_patience: Optional[int] = 2):
        self.model_name = name
        self.checkpoint_dir = checkpoint_dir
        self.log_dir = log_dir
        self.save_dir = save_dir
        self.epochs = epochs
        self.stopping_patience = stopping_patience
        self.lr_patience = lr_patience
        data = DobbyDelivery(json, kp_def, images)
        self.train_data = data.train_set
        self.val_data = data.val_set
        self.model = model

    def train(self, ):
        tensorboard_callback = TensorBoard(log_dir=f"{self.log_dir}{self.model_name}/")
        early_stopping_callback = EarlyStopping(monitor="val_mae", patience=3)
        reduce_lr_callback = ReduceLROnPlateau(monitor="val_mae", factor=0.1, patience=2,
                                               verbose=1, mode="auto")
        checkpoint = ModelCheckpoint(filepath=f'{self.checkpoint_dir}{self.model_name}',
                                     monitor='val_mae', verbose=1, save_best_only=True,
                                     mode='min')

        self.model.compile(loss="mean_squared_error",
                           optimizer=keras.optimizers.Adam(0.001),
                           metrics=['mae'])
        self.model.summary()
        self.model.fit(self.train_data, validation_data=self.val_data, epochs=self.epochs,
                       callbacks=[tensorboard_callback, early_stopping_callback,
                                  reduce_lr_callback, checkpoint,
                                  CustomLRCallback(self.model_name)])

        self.model.save(f'{self.save_dir}{self.model_name}')


class CustomLRCallback(keras.callbacks.Callback):
    """Custom callback for learning rate scheduler.
    """

    def __init__(self, model_name):
        super(CustomLRCallback, self).__init__()
        self.model_name = model_name

    def on_epoch_begin(self, epoch, logs=None):
        # open a file and check if the learning rate is the same as the one in the file
        with open(f"{self.model_name}-lr.txt", "r") as f:
            learning_rate = f.read()
```

```
57              # if not, set the learning rate to the one in the file
58          self.model.optimizer.learning_rate = float(learning_rate)
59          print(f"Epoch {epoch}: Learning rate changed to
60                {self.model.optimizer.learning_rate} from file.")
```

Listing F.4: DobbyTrainer.py, is responsible for training single models. The module uses DobbyDelivery.py (Appendix F.3).

Appendix G

Albus

G.1 AlbusSearch.py

```
import keras_tuner as kt
from colorama import Fore
from typing import Optional
from tensorflow import keras
from Dobby.DobbyDelivery import DobbyDelivery
from keras.callbacks import TensorBoard, EarlyStopping


class AlbusSearch:
    """
    AlbusSearch is a class that is used to search for the best model.
    """
    def __init__(self, model, json: str, kp_def: str, img_dir: str,
                 log_dir: str, epochs: Optional[int] = 50):
        """
        Initializes the AlbusSearch class.
        :param model: Model builder collected from Albus.AlbusModels...
        :param json: Path to dataset json file.
        :param kp_def: Path to keypoint definition file.
        :param img_dir: Path to image directory.
        :param log_dir: Path to log directory.
        :param epochs: Number of epochs to train for.
        """
        self.model = model
        self.json = json
        self.kp_def = kp_def
        self.img_dir = img_dir
        self.epochs = epochs
        self.log_dir = log_dir
        self.data = DobbyDelivery(self.json, self.kp_def, self.img_dir)

    def search(self):
        self.__run_tuner(self.model)

    def __run_tuner(self, model):
        stop_early = EarlyStopping(monitor='val_loss', patience=3)

        tuner = kt.Hyperband(model,
                             objective='val_loss',
```

```
                                max_epochs=15,
                                factor=3,
                                directory='tmp/tb',
                                project_name='Quantium',
                                overwrite=True
                                )
        tuner.search(self.data.train_set, validation_data=self.data.val_set,
                     epochs=self.epochs,
                     callbacks=[stop_early, TensorBoard(self.log_dir)])

        # Get the optimal hyperparameters and print to console.
        tuner.results_summary()
        best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
        print(Fore.GREEN, best_hps)
```

Listing G.1: AlbusSearch.py, is responsible for running architecture searches and hyperparameter tuning for model search spaces in AlbusModels. The module uses DobbyDelivery (Appendix F.3).

G.2 AlbusModels/cnn.py

```
from tensorflow import keras
from keras import layers


def build_cnn_search(hp):
    input_shape = (224, 224, 3)

    input_layer = layers.Input(input_shape)

    x = input_layer
    x = layers.Conv2D(hp.Int('units_Input', min_value=96,
                      max_value=512, step=32), 3,
                      padding="same", name="first_conv")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

    for i in range(hp.Int('Conv_layers', 2, 6)):
        x = layers.Conv2D(hp.Int('units_' + str(i), min_value=96,
                                 max_value=512, step=32), 3,
                                 padding="same",
                                 name="conv_layer_"+str(i))(x)

        if hp.Choice("BatchNorm" + str(i), [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)

        if hp.Choice("conv_dropout_C" + str(i), [True, False]):
            x = layers.Dropout(hp.Choice("conv_dropout_" + str(i),
                               values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                               name="conv_dropout_" + str(i))(x)

    x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

    for j in range(hp.Int('separable_layers', 2, 6)):
        x = layers.SeparableConv2D(hp.Int('sep_units_'+str(j), min_value=96,
                                   max_value=512, step=32), 3,
                                   padding="same", name="separable_layer_"+str(j))(x)

        if hp.Choice("BatchNorm" + str(j), [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        if hp.Choice("sep_dropout_C" + str(j), [True, False]):
            x = layers.Dropout(hp.Choice("sep_dropout_" + str(j),
                                         values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                                         name="sep_dropout_" + str(j))(x)

    x = layers.MaxPooling2D(3, strides=3, padding="same")(x)

    x = layers.SeparableConv2D(hp.Int('sep_out_units_0', min_value=96,
                                      max_value=512, step=32), kernel_size=5,
                                      strides=1, activation="relu",
                                      name="sep_out_0")(x)

```

```
57      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
58
59      x = layers.SeparableConv2D(hp.Int('sep_out_units_1', min_value=96,
60                                        max_value=512, step=32), kernel_size=3,
61                                        strides=1, activation="relu",
62                                        name="sep_out_1")(x)
63
64      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
65
66      x = layers.SeparableConv2D(hp.Int('sep_out_units_2', min_value=96,
67                                        max_value=512, step=32), kernel_size=2,
68                                        strides=1, activation="relu", name="sep_out_2")(x)
69      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
70
71      outputs = layers.SeparableConv2D(30, kernel_size=1, strides=1, activation="sigmoid",
72                                       name="sep_output")(x)
73      # outputs = layers.Flatten()(x)
74
75      model = keras.Model(input_layer, outputs)
76      model.summary()
77
78      hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
79
80      model.compile(loss="mean_squared_error",
81                    optimizer=keras.optimizers.Adam(hp_learning_rate),
82                    metrics=['mae'])
83      return model
```

Listing G.2: AlbusModels/cnn.py, delivers the cnn architecture and hyperparameter tuning search space to AlbusSearch.

G.3 AlbusModels/dense.py

```
from tensorflow import keras
from keras import layers


def model_builder(hp):

    shape = (224, 224, 3)

    input_layer = layers.Input(shape)
    x = input_layer

    x = layers.Conv2D(hp.Int('units_Input', min_value=16, max_value=128, step=8),
        3, padding="same", name="first_conv")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

    for i in range(hp.Int('Conv_layers', 2, 6)):
        x = layers.Conv2D(hp.Int('conv_units_' + str(i),
                                 min_value=16, max_value=128, step=8),
            3, padding="same", name="conv_layer_"+str(i))(x)

        if hp.Choice("BatchNorm" + str(i), [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)

        if hp.Choice("conv_dropout_C" + str(i), [True, False]):
            x = layers.Dropout(hp.Choice("conv_dropout_" + str(i),
                values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                name="conv_dropout_" + str(i))(x)

        x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

    for j in range(hp.Int('separable_layers', 2, 6)):
        x = layers.SeparableConv2D(hp.Int('sep_units_'+str(j), min_value=32,
            max_value=512, step=32), 3, padding="same",
            name="separable_layer_"+str(j))(x)

        if hp.Choice("sep_BatchNorm" + str(j), [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        if hp.Choice("sep_dropout_C" + str(j), [True, False]):
            x = layers.Dropout(hp.Choice("sep_dropout_" + str(j),
                values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                name="sep_dropout_" + str(j))(x)

    x = layers.Flatten()(x)

    for j in range(hp.Int('dense_layers', 1, 4)):
        x = layers.Dense(hp.Int('dense_units_'+str(j), min_value=32,
            max_value=500, step=32), name="dense_layer_"+str(j))(x)

        if hp.Choice("dense_BatchNorm" + str(j), [True, False]):
            x = layers.BatchNormalization()(x)
```

```
57
58          x = layers.Activation("relu")(x)
59          if hp.Choice("dense_dropout_C" + str(j), [True, False]):
60              x = layers.Dropout(hp.Choice("dense_dropout_" + str(j),
61                  values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
62                  name="dense_dropout_" + str(j))(x)
63
64      outputs = layers.Dense(30, activation="sigmoid", name="output")(x)
65
66      model = keras.Model(input_layer, outputs)
67      model.summary()
68
69      hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
70
71      model.compile(loss="mean_squared_error",
72                    optimizer=keras.optimizers.Adam(hp_learning_rate),
73                    metrics=['mae'])
74      return model
```

Listing G.3: AlbusModels/dense.py, delivers the dense architecture and hyperparameter tuning search space to AlbusSearch.

G.4 AlbusModels/residual.py

```python
from tensorflow import keras
from keras import layers


def build_residual_search(hp):
    shape = (224, 224, 3)

    inputs = layers.Input(shape)
    x = inputs

    # Entry block
    x = layers.Rescaling(1.0 / 255)(x)
    x = layers.Conv2D(hp.Int('units_input', min_value=32,
                             max_value=512, step=32), 3,
                             strides=2, padding="same")(x)

    if hp.Choice("BatchNorm1", [True, False]):
        x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.Conv2D(hp.Int('units_2', min_value=32, max_value=512, step=32), 3,
                             padding="same")(x)

    if hp.Choice("BatchNorm2", [True, False]):
        x = layers.BatchNormalization()(x)

    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    for i in range(hp.Int('layers', 2, 6)):
        size = hp.Int(f'unit_block-{str(i)}', min_value=32, max_value=512, step=32)
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)

        if hp.Choice("BatchNorm3", [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)

        if hp.Choice("BatchNorm4", [True, False]):
            x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(size, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    x = layers.MaxPooling2D(2, strides=4, padding="same")(x)
    x = layers.SeparableConv2D(30, kernel_size=2, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
```

```
57      x = layers.Activation("relu")(x)
58      x = layers.MaxPooling2D(2, strides=4, padding="same")(x)
59
60      outputs = layers.SeparableConv2D(30, kernel_size=1, strides=1,
61                                       activation="sigmoid")(x)
62
63      model = keras.Model(inputs, outputs)
64      model.summary()
65
66      hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
67
68      model.compile(loss="mean_squared_error",
69                    optimizer=keras.optimizers.Adam(hp_learning_rate),
70                    metrics=['mae'])
71
72      return model
```

Listing G.4: AlbusModels/residual.py, delivers the residual cnn architecture and hyperparameter tuning search space to AlbusSearch.

G.5 AlbusModels/resnet.py

```
from tensorflow import keras
from keras import layers


def build_resnet_search(hp):
    # Load the pre-trained weights of MobileNetV2 and freeze the weights
    backbone = keras.applications.ResNet50(
        weights="imagenet", include_top=False, input_shape=(224, 224, 3)
    )
    backbone.trainable = False

    inputs = layers.Input((224, 224, 3))
    model = keras.applications.resnet50.preprocess_input(inputs)
    model = backbone(model)

    for i in range(hp.Int('Conv_layers', 0, 6)):
        model = layers.Conv2D(hp.Int('units_' + str(i), min_value=32,
                              max_value=128, step=32), 3, padding="same",
                              name="conv_layer_" + str(i))(model)

        if hp.Choice("BatchNorm" + str(i), [True, False]):
            model = layers.BatchNormalization()(model)

        model = layers.Activation("relu")(model)

        if hp.Choice("conv_dropout_C" + str(i), [True, False]):
            model = layers.Dropout(hp.Choice("conv_dropout_" + str(i),
                                   values=[0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                                   name="conv_dropout_" + str(i))(model)

    model = layers.MaxPooling2D(2, strides=1, padding="same")(model)

    for j in range(hp.Int('separable_layers', 0, 6)):
        model = layers.SeparableConv2D(hp.Int('sep_units_' + str(j), min_value=32,
                                       max_value=128, step=32), 3, padding="same",
                                       name="separable_layer_" + str(j))(model)

        if hp.Choice("sep_BatchNorm" + str(j), [True, False]):
            model = layers.BatchNormalization()(model)

        model = layers.Activation("relu")(model)
        if hp.Choice("sep_dropout_C" + str(j), [True, False]):
            model = layers.Dropout(hp.Choice("sep_dropout_" + str(j),
                                   values=[0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                                   name="sep_dropout_" + str(j))(model)

        # model = layers.MaxPooling2D(3, strides=3, padding="same")(model)

    model = layers.Dropout(0.2)(model)
    model = layers.SeparableConv2D(
        30, kernel_size=5, strides=1, activation="relu")(model)
    outputs = layers.SeparableConv2D(30, kernel_size=3, strides=1,
                                     activation="sigmoid")(model)

    model = keras.Model(inputs, outputs, name="keypoint_detector")

```

```
57      hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
58
59      model.compile(loss="mean_squared_error",
60                    optimizer=keras.optimizers.Adam(hp_learning_rate),
61                    metrics=['mae'])
62      model.summary()
63      return model
```

Listing G.5: AlbusModels/resnet.py, delivers the ResNet architecture and hyperparameter tuning search space to AlbusSearch.

G.6 AlbusModels/mobilenet.py

```
from tensorflow import keras
from keras import layers


def build_mobilenet_search(hp):
    # Load the pre-trained weights of MobileNetV2 and freeze the weights
    backbone = keras.applications.MobileNetV2(
        weights="imagenet", include_top=False, input_shape=(224, 224, 3)
    )
    backbone.trainable = False

    inputs = layers.Input((224, 224, 3))
    model = keras.applications.mobilenet_v2.preprocess_input(inputs)
    model = backbone(model)

    for i in range(hp.Int('Conv_layers', 0, 6)):
        model = layers.Conv2D(hp.Int('units_' + str(i), min_value=32,
                                     max_value=128, step=32), 3,
                                     padding="same",
                                     name="conv_layer_" + str(i))(model)

        if hp.Choice("BatchNorm" + str(i), [True, False]):
            model = layers.BatchNormalization()(model)

        model = layers.Activation("relu")(model)

        if hp.Choice("conv_dropout_C" + str(i), [True, False]):
            model = layers.Dropout(hp.Choice("conv_dropout_" + str(i),
                                   values=[0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                                   name="conv_dropout_" + str(i))(model)

    model = layers.MaxPooling2D(2, strides=1, padding="same")(model)

    for j in range(hp.Int('separable_layers', 0, 6)):
        model = layers.SeparableConv2D(hp.Int('sep_units_' + str(j), min_value=32,
                                              max_value=128, step=32),
                                              3, padding="same",
                                              name="separable_layer_" + str(j))(model)

        if hp.Choice("sep_BatchNorm" + str(j), [True, False]):
            model = layers.BatchNormalization()(model)

        model = layers.Activation("relu")(model)
        if hp.Choice("sep_dropout_C" + str(j), [True, False]):
            model = layers.Dropout(hp.Choice("sep_dropout_" + str(j),
                                   values=[0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
                                   name="sep_dropout_" + str(j))(model)

        # model = layers.MaxPooling2D(3, strides=3, padding="same")(model)

    model = layers.Dropout(0.2)(model)
    model = layers.SeparableConv2D(
        30, kernel_size=5, strides=1, activation="relu")(model)
    outputs = layers.SeparableConv2D(30, kernel_size=3, strides=1,
                                     activation="sigmoid")(model)

```

```
57      model = keras.Model(inputs, outputs, name="keypoint_detector")
58
59      hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
60
61      model.compile(loss="mean_squared_error",
62                    optimizer=keras.optimizers.Adam(hp_learning_rate),
63                    metrics=['mae'])
64      model.summary()
65      return model
```

Listing G.6: AlbusModels/mobilenet.py, delivers the MobileNet architecture and hyperparameter tuning search space to AlbusSearch.

Appendix H

Alastor

H.1 AlastorTrainer.py

```
import os
from colorama import Fore
from Filch.FilchUtils import get_model
from ModelBuilder.cnn import cnn
from ModelBuilder.resnet import resnet
from ModelBuilder.residual import residual
from ModelBuilder.mobilenet import mobilenet
from Dobby.DobbyTrainer import DobbyTrainer


class AlastorTrainer:
    def __init__(self, json_folder: str, image_folder: str, kp_def: str,
                 checkpoint_dir: str, save_dir: str, log_dir: str):
        self.json_folder = json_folder
        self.image_folder = image_folder
        self.kp_def = kp_def
        self.checkpoint_dir = checkpoint_dir
        self.save_dir = save_dir
        self.log_dir = log_dir
        self.splits = os.listdir(self.json_folder)
        self.previous_split = ''
        self.compile_status = True

    def run(self):
        print(Fore.GREEN, "Running AlastorTrainer")
        self.splits.sort()
        for split in self.splits:
            split_name = os.path.splitext(split)[0]
            print(split_name)
            print(Fore.GREEN, "Running AlastorTrainer on split:", split)
            if '1' in split_name:
                mobilenet_model = mobilenet()
                resnet_model = resnet()
                residual_model = residual()
                cnn_model = cnn()
            else:
                mobilenet_model = get_model(f'{self.checkpoint_dir}
                                            mobilenet-{self.previous_split}/')
                resnet_model = get_model(f'{self.checkpoint_dir}
```

```
                                        resnet-{self.previous_split}/')
            residual_model = get_model(f'{self.checkpoint_dir}
                                        residual-{self.previous_split}/')
            cnn_model = get_model(f'{self.checkpoint_dir}
                                  cnn-{self.previous_split}/')
            self.compile_status = False

        print(Fore.GREEN, f"Running DobbyTrainer, MobileNet-{split}")
        dobby_mobilenet = DobbyTrainer(json=f'{self.json_folder}{split}',
                        kp_def=self.kp_def, images=self.image_folder,
                        checkpoint_dir=self.checkpoint_dir,
                        log_dir=self.log_dir, save_dir=self.save_dir,
                        name=f'mobilenet-{split_name}', model=mobilenet_model,
                        compiler=self.compile_status)
        dobby_mobilenet.train()

        print(Fore.GREEN, f"Running DobbyTrainer, ResNet-{split_name}")
        dobby_resnet = DobbyTrainer(json=f'{self.json_folder}{split}',
                                    kp_def=self.kp_def,
                                    images=self.image_folder,
                                    checkpoint_dir=self.checkpoint_dir,
                                    log_dir=self.log_dir,
                                    save_dir=self.save_dir,
                                    name=f'resnet-{split_name}',
                                    model=resnet_model,
                                    compiler=self.compile_status)
        dobby_resnet.train()

        print(Fore.GREEN, f"Running DobbyTrainer, CNN-{split_name}")
        dobby_cnn = DobbyTrainer(json=f'{self.json_folder}{split}',
                                 kp_def=self.kp_def,
                                 images=self.image_folder,
                                 checkpoint_dir=self.checkpoint_dir,
                                 log_dir=self.log_dir,
                                 save_dir=self.save_dir,
                                 name=f'cnn-{split_name}',
                                 model=cnn_model,
                                 compiler=self.compile_status)
        dobby_cnn.train()

        print(Fore.BLUE, f"Running DobbyTrainer, Residual-{split_name}")
        dobby_residual = DobbyTrainer(json=f'{self.json_folder}{split}',
                        kp_def=self.kp_def,
                        images=self.image_folder,
                        checkpoint_dir=self.checkpoint_dir,
                        log_dir=self.log_dir, save_dir=self.save_dir,
                        name=f'residual-{split_name}',
                        model=residual_model,
                        compiler=self.compile_status)
        dobby_residual.train()
        self.previous_split = split_name

    print(Fore.GREEN, "AlastorTrainer finished")
```

Listing H.1: AlastorTrainer.py, is responsible for training all our models on all splits of a given dataset without stopping. The module uses DobbyTrainer (Appendix F.4)

Appendix I

ModelBuilder

I.1 ModelBuilder/cnn.py

```
from tensorflow import keras
from keras import layers


def cnn():
    shape = (224, 224, 3)

    input_layer = layers.Input(shape)
    x = input_layer

    x = layers.Conv2D(352, 3, padding="same", name="first_conv")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

    x = layers.Conv2D(384, 3, padding="same")(x)  # Units_0
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.Dropout(0.5)(x)

    x = layers.Conv2D(352, 3, padding="same")(x)  # units_1
    x = layers.Activation("relu")(x)

    x = layers.Conv2D(416, 3, padding="same")(x)  # units_2
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.Dropout(0.2)(x)

    x = layers.Conv2D(96, 3, padding="same")(x)  # units_3
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.Conv2D(128, 3, padding="same")(x)  # units_4
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.Dropout(0.3)(x)

    x = layers.MaxPooling2D(2, strides=2, padding="same")(x)

```

```
40      x = layers.SeparableConv2D(320, 3, padding="same")(x)  # sep_units_0
41      x = layers.BatchNormalization()(x)
42      x = layers.Activation("relu")(x)
43
44      x = layers.SeparableConv2D(352, 3, padding="same")(x)  # sep_units_1
45      x = layers.Activation("relu")(x)
46
47      x = layers.SeparableConv2D(128, 3, padding="same")(x)  # sep_units_2
48      x = layers.Activation("relu")(x)
49      x = layers.Dropout(0.5)(x)
50
51      x = layers.SeparableConv2D(256, 3, padding="same")(x)  # sep_units_3
52      x = layers.Activation("relu")(x)
53
54      x = layers.SeparableConv2D(352, 3, padding="same")(x)  # sep_units_4
55      x = layers.BatchNormalization()(x)
56      x = layers.Activation("relu")(x)
57
58      x = layers.SeparableConv2D(416, 3, padding="same")(x)  # sep_units_5
59      x = layers.Activation("relu")(x)
60
61      x = layers.MaxPooling2D(3, strides=3, padding="same")(x)
62
63      x = layers.SeparableConv2D(448, kernel_size=5, strides=1,
64                                 activation="relu", name="sep_out_0")(x)
65      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
66      x = layers.SeparableConv2D(224, kernel_size=3, strides=1,
67                                 activation="relu", name="sep_out_1")(x)
68      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
69      x = layers.SeparableConv2D(480, kernel_size=2, strides=1,
70                                 activation="relu", name="sep_out_2")(x)
71      x = layers.MaxPooling2D(2, strides=2, padding="same")(x)
72
73      outputs = layers.SeparableConv2D(30, kernel_size=1, strides=1,
74                                       activation="sigmoid", name="output")(x)
75
76      model = keras.Model(input_layer, outputs, name="McFly_cnn_50epochs")
77      return model
```

Listing I.1: ModelBuilder/cnn.py, is responsible for delivering our cnn architecture to a training session.

I.2 ModelBuilder/residual.py

```
# This function returns a residual model.
# Written by William Svea–Lochert, Halden, Norway 2021.

from tensorflow import keras
from keras import layers


def residual():
    shape = (224, 224, 3)

    inputs = layers.Input(shape)
    x = inputs

    # Entry block
    x = layers.Rescaling(1.0 / 255)(x)
    x = layers.Conv2D(416, 3, strides=2, padding="same")(x)
    x = layers.Activation("relu")(x)
    x = layers.Conv2D(224, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    sizes = [320, 416, 256, 160, 416, 32]

    for size in sizes:
        x = layers.Activation("relu")(x)

        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)
        x = layers.Activation("relu")(x)

        x = layers.SeparableConv2D(size, 3, padding="same")(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(size, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    x = layers.MaxPooling2D(2, strides=4, padding="same")(x)
    x = layers.SeparableConv2D(30, kernel_size=2, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2, strides=4, padding="same")(x)

    outputs = layers.SeparableConv2D(30, kernel_size=1, strides=1,
                                     activation="sigmoid")(x)

    model = keras.Model(inputs, outputs)
    model.summary()

    return model
```

Listing I.2: ModelBuilder/residual.py, is responsible for delivering our residual architecture to a training session.

I.3 ModelBuilder/resnet.py

```
from tensorflow import keras
from keras import layers
from keras.callbacks import TensorBoard, EarlyStopping, ReduceLROnPlateau, ModelCheckpoint


def resnet():
    # Load the pre-trained weights of MobileNetV2 and freeze the weights
    backbone = keras.applications.ResNet50(
        weights="imagenet", include_top=False, input_shape=(224, 224, 3)
    )
    backbone.trainable = True
    print("Number of layers in the base model: ", len(backbone.layers))
    fine_tune_at = 100

    for layer in backbone.layers[:fine_tune_at]:
        layer.trainable = False

    inputs = layers.Input((224, 224, 3))
    model = keras.applications.resnet50.preprocess_input(inputs)
    model = backbone(model)

    model = layers.Conv2D(96, 3, padding="same", name="conv_layer_0")(model)
    model = layers.BatchNormalization()(model)
    model = layers.Activation("relu")(model)

    model = layers.Conv2D(64, 3, padding="same", name="conv_layer_1")(model)
    model = layers.Activation("relu")(model)

    model = layers.Conv2D(96, 3, padding="same", name="conv_layer_2")(model)
    model = layers.BatchNormalization()(model)
    model = layers.Activation("relu")(model)

    model = layers.MaxPooling2D(2, strides=1, padding="same")(model)

    model = layers.SeparableConv2D(64, 3, padding="same", name="separable_layer_0")(model)
    model = layers.Activation("relu")(model)

    model = layers.Dropout(0.2)(model)
    model = layers.SeparableConv2D(
        30, kernel_size=5, strides=1, activation="relu")(model)
    outputs = layers.SeparableConv2D(30, kernel_size=3,
                                     strides=1, activation="sigmoid")(model)

    model = keras.Model(inputs, outputs, name="keypoint_detector")

    return model
```

Listing I.3: ModelBuilder/resnet.py, is responsible for delivering our ResNet architecture to a training session.

I.4 ModelBuilder/mobilenet.py

```
from tensorflow import keras
from keras import layers


def mobilenet():
    # Load the pre-trained weights of MobileNetV2 and freeze the weights
    backbone = keras.applications.MobileNetV2(
        weights="imagenet", include_top=False, input_shape=(224, 224, 3)
    )
    backbone.trainable = True
    print("Number of layers in the base model: ", len(backbone.layers))
    fine_tune_at = 100

    for layer in backbone.layers[:fine_tune_at]:
        layer.trainable = False

    inputs = layers.Input((224, 224, 3))
    model = keras.applications.mobilenet_v2.preprocess_input(inputs)
    model = backbone(model)

    model = layers.Conv2D(64, 3, padding="same", name="conv_layer_0")(model)

    model = layers.Activation("relu")(model)

    model = layers.Conv2D(128, 3, padding="same", name="conv_layer_1")(model)

    model = layers.BatchNormalization()(model)

    model = layers.Activation("relu")(model)

    model = layers.MaxPooling2D(2, strides=1, padding="same")(model)

    model = layers.Dropout(0.2)(model)
    model = layers.SeparableConv2D(30, kernel_size=5, strides=1,
                                   activation="relu")(model)
    outputs = layers.SeparableConv2D(30, kernel_size=3,
                                     strides=1, activation="sigmoid")(model)

    model = keras.Model(inputs, outputs, name="keypoint_detector")

    return model
```

Listing I.4: ModelBuilder/mobilenet.py, is responsible for delivering our MobileNet architecture to a training session.

Appendix J

Sirius

J.1 SiriusConverter.py

```python
import os
import tensorflow as tf
from colorama import Fore, Back, Style
from Filch.FilchUtils import get_model, get_models_from_folder


class SiriusConverter:
    """
    Class to convert a model to a Tensorflow Lite model.
    """
    def __init__(self, directory: str, output_directory: str):
        self.directory = directory
        self.output_directory = output_directory

    def __convert_models(self):
        """
        Converts all models in the directory to Tensorflow Lite models.
        :return:
        """
        models = get_models_from_folder(self.directory)
        for model_name in models:
            model = get_model(f'{self.directory}{model_name}')
            converter = tf.lite.TFLiteConverter.from_keras_model(
                model)
            tflite_model = converter.convert()

            # check if file exists if not create it
            if not os.path.exists(f'{self.output_directory}{model_name}.tflite'):
                print(Fore.RED, f'{model_name} does not exist, creating it!')
                with open(f'{self.output_directory}{model_name}.tflite', 'w+') as f:
                    f.write(tflite_model)
            else:
                print(Fore.GREEN, f'{model_name} already exists, adding model information...')
                with open(f'{self.output_directory}{model_name}.tflite', 'wb') as f:
                    f.write(tflite_model)
```

Listing J.1: SiriusConverter.py, is responsible for converting our trained models into TFLite models. The module utilizes FilchUtils.py (Appendix D.1).

Appendix K

Lupin

K.1 LupinExamin.py

```
import numpy as np
from colorama import Fore
import cv2
from typing import Optional
import matplotlib.pyplot as plt
from Filch.FilchUtils import get_train_params, get_pose, get_model, load_image


class LupinExamin:
    def __init__(self, json: str, img_dir: str, kp_def: str, model_dir: str,
                 visualize: Optional[bool] = False, num_samples: Optional[int] = 500):
        """
        Initialize the class.
        :param json: path to json file.
        :param img_dir: path to image directory.
        :param kp_def: path to keypoint definition file.
        :param model_dir: path to models directory.
        """
        self.json = json
        self.img_dir = img_dir
        self.kp_def = kp_def
        self.model_dir = model_dir
        self.samples, self.json_dict, self.kp_def, self.colors, self.labels =
                                        get_train_params(self.json, self.kp_def)
        self.images = []
        self.gt_kps = []
        self. list_of_predictions = []
        self.list_of_gt = []
        self.visualize = visualize
        self.selected_samples = self.__get_test_samples()
        self.num_samples = num_samples

        self.head_errors = []
        self.left_shoulder_errors = []
        self.right_shoulder_errors = []
        self.left_elbow_errors = []
        self.right_elbow_errors = []
        self.left_wrist_errors = []
        self.right_wrist_errors = []
```

```
40          self.left_hip_errors = []
41          self.right_hip_errors = []
42          self.neck_errors = []
43          self.torso_errors = []
44          self.left_knee_errors = []
45          self.right_knee_errors = []
46          self.left_ankle_errors = []
47          self.right_ankle_errors = []
48
49          self.head_gt = []
50          self.left_shoulder_gt = []
51          self.right_shoulder_gt = []
52          self.left_elbow_gt = []
53          self.right_elbow_gt = []
54          self.left_wrist_gt = []
55          self.right_wrist_gt = []
56          self.left_hip_gt = []
57          self.right_hip_gt = []
58          self.neck_gt = []
59          self.torso_gt = []
60          self.left_knee_gt = []
61          self.right_knee_gt = []
62          self.left_ankle_gt = []
63          self.right_ankle_gt = []
64
65          self.head_distance = 0
66          self.left_ankle_distance = 0
67          self.left_elbow_distance = 0
68          self.left_hip_distance = 0
69          self.left_knee_distance = 0
70          self.left_shoulder_distance = 0
71          self.left_wrist_distance = 0
72          self.neck_distance = 0
73          self.right_ankle_distance = 0
74          self.right_elbow_distance = 0
75          self.right_hip_distance = 0
76          self.right_knee_distance = 0
77          self.right_shoulder_distance = 0
78          self.right_wrist_distance = 0
79          self.torso_distance = 0
80
81      def __get_test_samples(self):
82          num_samples = 500
83          selected_samples = np.random.choice(self.samples, num_samples, replace=False)
84
85          for sample in selected_samples:
86              data = get_pose(sample, self.json_dict, self.img_dir)
87              image = data["img_data"]
88              keypoint = data["joints"]
89              self.images.append(image)
90              self.gt_kps.append(keypoint)
91
92          return selected_samples
93
94      def __predict(self, image_path: str, index: int, model):
95          """
96          Test the model on a single image.
97          :param image_path: path to image.
```

```
98          :param index: index of image in dataset.
99          :return: prediction keypoints, and ground truth keypoints.
100         """
101         img_file = image_path
102         img = load_image(image_path)
103         predictions = model.predict(img).reshape(-1, 15, 2) * 224
104         ground_truth = np.array(self.gt_kps[index])
105         ground_truth.reshape(-1, 15, 2)
106
107         if self.visualize:
108             self.__visualize_keypoints(predictions, img_file, [ground_truth])
109         return predictions, ground_truth
110
111     def __run_test(self):
112         """
113         Run the test, and print the results.
114         :return: nothing.
115         """
116         for i in range(len(self.selected_samples)):
117             pred, gt = self.__predict(self.img_dir + self.selected_samples[i], i)
118             self.list_of_predictions.append(pred)
119             self.list_of_gt.append(gt)
120
121         controlled_pred, controlled_gt = self.__check_if_valid()
122         self.__append_errors(controlled_pred)
123         self.__append_gt(controlled_gt)
124         self.__calculate_distance()
125
126     def __check_if_valid(self):
127         """
128         Check if the predictions are valid or if they are not usable to measure the error.
129         :return: list of valid predictions, and list of valid ground truths.
130         """
131         p, g = [], []
132         invalid = 0
133
134         for i in range(len(self.list_of_predictions)):
135             checker = False
136
137             for j in range(15):
138                 # print(list_of_gt[i][j][0])
139                 if self.list_of_predictions[i][0][j][0] < 10 or
140                         self.list_of_predictions[i][0][j][1] < 10:
141                     checker = True
142                     invalid += 1
143                 elif self.list_of_gt[i][j][0] < 1 or
144                         self.list_of_gt[i][j][1] < 1:
145                     checker = True
146                     invalid += 1
147             if not checker:
148                 p.append(self.list_of_predictions[i])
149                 g.append(self.list_of_gt[i])
150         print(Fore.RED, f'{invalid} invalid samples found...')
151         return p, g
152
153     def __append_errors(self, predictions):
154         """
155         Append errors to the list.
```

```
156         :param predictions: list of predictions.
157         :return: nothing
158         """
159         for i in predictions:
160             self.head_errors.append(i[0][0])
161             self.left_ankle_errors.append(i[0][1])
162             self.left_elbow_errors.append(i[0][2])
163             self.left_hip_errors.append(i[0][3])
164             self.left_knee_errors.append(i[0][4])
165             self.left_shoulder_errors.append(i[0][5])
166             self.left_wrist_errors.append(i[0][6])
167             self.neck_errors.append(i[0][7])
168             self.right_ankle_errors.append(i[0][8])
169             self.right_elbow_errors.append(i[0][9])
170             self.right_hip_errors.append(i[0][10])
171             self.right_knee_errors.append(i[0][11])
172             self.right_shoulder_errors.append(i[0][12])
173             self.right_wrist_errors.append(i[0][13])
174             self.torso_errors.append(i[0][14])
175 
176     def __append_gt(self, gt):
177         """
178         Append ground truth to the list.
179         :param gt: list of ground truths.
180         :return: Nothing.
181         """
182         for i in range(len(gt)):
183             self.head_gt.append(gt[i][0])
184             self.left_ankle_gt.append(gt[i][1])
185             self.left_elbow_gt.append(gt[i][2])
186             self.left_hip_gt.append(gt[i][3])
187             self.left_knee_gt.append(gt[i][4])
188             self.left_shoulder_gt.append(gt[i][5])
189             self.left_wrist_gt.append(gt[i][6])
190             self.neck_gt.append(gt[i][7])
191             self.right_ankle_gt.append(gt[i][8])
192             self.right_elbow_gt.append(gt[i][9])
193             self.right_hip_gt.append(gt[i][10])
194             self.right_knee_gt.append(gt[i][11])
195             self.right_shoulder_gt.append(gt[i][12])
196             self.right_wrist_gt.append(gt[i][13])
197             self.torso_gt.append(gt[i][14])
198 
199     def __calculate_distance(self):
200         """
201         Calculate the distance between the ground truth and the predictions,
202         and print it to the console.
203         """
204 
205         for i in range(len(self.head_errors)):
206             self.head_distance += np.sqrt((self.head_errors[i][0] - self.head_gt[i][0])
207                     ** 2 + (self.head_errors[i][1] - self.head_gt[i][1]) ** 2)
208             self.left_ankle_distance += np.sqrt((self.left_ankle_errors[i][0] -
209                     self.left_ankle_gt[i][0]) ** 2 + (self.left_ankle_errors[i][1] -
210                     self.left_ankle_gt[i][1]) ** 2)
211             self.left_elbow_distance += np.sqrt((self.left_elbow_errors[i][0] -
212                     self.left_elbow_gt[i][0]) ** 2 + (self.left_elbow_errors[i][1]
213                     - self.left_elbow_gt[i][1]) ** 2)
```

```
214             self.left_hip_distance += np.sqrt((self.left_hip_errors[i][0] -
215                 self.left_hip_gt[i][0]) ** 2 + (self.left_hip_errors[i][1] -
216                 self.left_hip_gt[i][1]) ** 2)
217             self.left_knee_distance += np.sqrt((self.left_knee_errors[i][0] -
218                 self.left_knee_gt[i][0]) ** 2 + (self.left_knee_errors[i][1] -
219                 self.left_knee_gt[i][1]) ** 2)
220             self.left_shoulder_distance += np.sqrt((self.left_shoulder_errors[i][0] -
221                 self.left_shoulder_gt[i][0]) ** 2 + (self.left_shoulder_errors[i][1] -
222                 self.left_shoulder_gt[i][1]) ** 2)
223             self.left_wrist_distance += np.sqrt((self.left_wrist_errors[i][0] -
224                 self.left_wrist_gt[i][0]) ** 2 + (self.left_wrist_errors[i][1] -
225                 self.left_wrist_gt[i][1]) ** 2)
226             self.neck_distance += np.sqrt((self.neck_errors[i][0] - self.neck_gt[i][0])
227                 ** 2 + (self.neck_errors[i][1] - self.neck_gt[i][1]) ** 2)
228             self.right_ankle_distance += np.sqrt((self.right_ankle_errors[i][0] -
229                 self.right_ankle_gt[i][0]) ** 2 + (self.right_ankle_errors[i][1] -
230                 self.right_ankle_gt[i][0]) ** 2)
231             self.right_elbow_distance += np.sqrt((self.right_elbow_errors[i][0] -
232                 self.right_elbow_gt[i][0]) ** 2 + (self.right_elbow_errors[i][1] -
233                 self.right_elbow_gt[i][0]) ** 2)
234             self.right_hip_distance += np.sqrt((self.right_hip_errors[i][0] -
235                 self.right_hip_gt[i][0]) ** 2 + (self.right_hip_errors[i][1] -
236                 self.right_hip_gt[i][0]) ** 2)
237             self.right_knee_distance += np.sqrt((self.right_knee_errors[i][0] -
238                 self.right_knee_gt[i][0]) ** 2 + (self.right_knee_errors[i][1] -
239                 self.right_knee_gt[i][0]) ** 2)
240             self.right_shoulder_distance += np.sqrt((self.right_shoulder_errors[i][0] -
241                 self.right_shoulder_gt[i][0]) ** 2 + (self.right_shoulder_errors[i][1] -
242                 self.right_shoulder_gt[i][0]) ** 2)
243             self.right_wrist_distance += np.sqrt((self.right_wrist_errors[i][0] -
244                 self.right_wrist_gt[i][0]) ** 2 + (self.right_wrist_errors[i][1] -
245                 self.right_wrist_gt[i][0]) ** 2)
246             self.torso_distance += np.sqrt((self.torso_errors[i][0] - self.torso_gt[i][0])
247                 ** 2 + (self.torso_errors[i][1] - self.torso_gt[i][0]) ** 2)
248
249             # print("Model: ", model_name)
250             print(Fore.MAGENTA, "Average Head distance: ",
251                 self.head_distance / len(self.head_errors))
252             print(Fore.MAGENTA, "Average Neck distance: ",
253                 self.neck_distance / len(self.left_hip_errors))
254
255             print(Fore.RED, "Average Right_shoulder distance: ",
256                 self.right_shoulder_distance / len(self.left_hip_errors))
257             print(Fore.RED, "Average Right_elbow distance: ",
258                 self.right_elbow_distance / len(self.left_hip_errors))
259             print(Fore.RED, "Average Right_wrist distance: ",
260                 self.right_wrist_distance / len(self.left_hip_errors))
261
262             print(Fore.GREEN, "Average Left_shoulder distance: ",
263                 self.left_shoulder_distance / len(self.left_hip_errors))
264             print(Fore.GREEN, "Average Left_elbow distance: ",
265                 self.left_elbow_distance / len(self.left_ankle_errors))
266             print(Fore.GREEN, "Average Left_wrist distance: ",
267                 self.left_wrist_distance / len(self.left_hip_errors))
268
269             print(Fore.MAGENTA, "Average Torso distance: ",
270                 self.torso_distance / len(self.left_hip_errors))
271             print(Fore.RED, "Average Right_hip distance: ",
```

```
272                 self.right_hip_distance / len(self.left_hip_errors))
273             print(Fore.RED, "Average Right_knee distance: ",
274                 self.right_knee_distance / len(self.left_hip_errors))
275             print(Fore.RED, "Average Right_ankle distance: ",
276                 self.right_ankle_distance / len(self.left_hip_errors))
277 
278             print(Fore.GREEN, "Average Left_hip distance: ",
279                 self.left_hip_distance / len(self.left_hip_errors))
280             print(Fore.GREEN, "Average Left_knee distance: ",
281                 self.left_knee_distance / len(self.left_hip_errors))
282             print(Fore.GREEN, "Average Left_Ankle distance: ",
283                 self.left_ankle_distance / len(self.left_ankle_errors))
284 
285     def __visualize_keypoints(self, keypoints, image_file: str, gt_keypoints,
286                               rot: Optional[bool] = False):
287         """
288         Visualize keypoints on image.
289         :param keypoints: Predicted keypoints.
290         :param image_file: path to image.
291         :param gt_keypoints: Ground truth keypoints.
292         :param rot: True if images needs to be rotated, false otherwise.
293         :return:
294         """
295 
296         colours = ['#' + color for color in self.colors]
297         fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 10))
298 
299         plt.rcParams.update({'font.size': 16,
300                              'text.color': 'white', })
301 
302         image = plt.imread(image_file)
303         """rotate image 90 degrees to the right"""
304         # remove the next 3 lines if you don't want to rotate the image
305         if rot:
306             image = np.rot90(image, k=1, axes=(0, 1))
307             image = np.flipud(image)
308             image = np.fliplr(image)
309 
310         ax1.imshow(image)
311         ax1.set_title('Prediction')
312         ax2.imshow(image)
313         ax2.set_title('Ground Truth')
314 
315         for current_keypoint in keypoints:
316             current_keypoint = np.array(current_keypoint)
317             # Since the last entry is the visibility flag, we discard it.
318             current_keypoint = current_keypoint[:, :2]
319             for idx, (x, y) in enumerate(current_keypoint):
320                 ax1.scatter([x], [y], c=colours[idx], marker="x", s=50, linewidths=5)
321 
322         for current_keypoint in gt_keypoints:
323             current_keypoint = np.array(current_keypoint)
324             # Since the last entry is the visibility flag, we discard it.
325             current_keypoint = current_keypoint[:]
326             for idx, (x, y) in enumerate(current_keypoint):
327                 ax2.scatter([x], [y], c=colours[idx], marker="x", s=50, linewidths=5)
328 
329         plt.tight_layout(pad=2.0)
```

```
330         plt.show()
```

Listing K.1: LupinExamin.py is responsible to calculating the distance from predicted keypoints to ground truth keypoints to measure a models accuracy. The module uses Filch (Appendix D.1)

Appendix L

James

L.1 JamesPredict.py

```python
from typing import Optional
from Filch.FilchUtils import load_image, get_model, visualize_keypoints


def predict(image_path, model_path, visualize: Optional[bool] = False):
    """
    Predict the image using the model
    :param image_path: path to the image
    :param model_path: path to the model
    :param visualize: whether to visualize the keypoints
    :return: the predicted keypoints
    """
    image = load_image(image_path)
    model = get_model(model_path)
    prediction = model.predict(image).reshape(-1, 15, 2) * 224
    if visualize:
        visualize_keypoints(prediction, image_path, False)
    return prediction
```

Listing L.1: JamesPredict.py is responsible to making a model prediction, the script uses Filch (Appendix D.1)

L.2 JamesAngle.py

```
import numpy as np
from JamesPredict import predict
from colorama import Fore


class JamesAngle:
  def __init__(self, image: str, model: str):
    self.prediction = predict(image, model, True)
    self.person = self.__get_person()
    if self.person:
      self.left_arm, self.right_arm, self.hips, self.core, self.left_leg,
          self.right_leg, self.is_facing = self.__get_joints()
      self.left_range = range(50, 110, 1)
      self.right_range = range(50, 110, 1)
      self.forward_range = range(110, 160, 1)
      self.reverse_range = range(110, 160, 1)

  def __get_person(self):
    """
      Returns there is a person in the image.
    """
    counter = 0
    for i in range(15):
      if self.prediction[0][i][0] < 20 and self.prediction[0][i][1] < 20:
        counter += 1
    if counter > 5:
      return False
    else:
      return True

  def __get_joints(self):
    """
      Returns the all joints except head, and checks which way the person is facing.
    """
    left_arm, right_arm, hips, core, left_leg, right_leg = [], [], [], [], [], []

    # print(self.prediction[0])
    left_arm.append(self.prediction[0][5])
    left_arm.append(self.prediction[0][2])
    left_arm.append(self.prediction[0][6])
    right_arm.append(self.prediction[0][12])
    right_arm.append(self.prediction[0][9])
    right_arm.append(self.prediction[0][13])

    hips.append(self.prediction[0][4])
    hips.append(self.prediction[0][11])

    core.append(self.prediction[0][7])
    core.append(self.prediction[0][14])

    left_leg.append(self.prediction[0][4])
    left_leg.append(self.prediction[0][1])

    right_leg.append(self.prediction[0][11])
    right_leg.append(self.prediction[0][8])

```

```
57      if self.prediction[0][5][0] > self.prediction[0][12][0]:
58        return left_arm, right_arm, hips,  core, left_leg, right_leg, True
59      else:
60        return left_arm, right_arm, hips,  core, left_leg, right_leg, False
61
62    def __get_signal(self):
63      """
64        Returns the signal that the person is facing.
65        Returns: str
66      """
67      if self.person:
68        left_arm_angle = self.__calculate_angle(self.left_arm[0],
69                         self.left_arm[1], self.left_arm[2])
70        right_arm_angle = self.__calculate_angle(self.right_arm[0],
71                          self.right_arm[1], self.right_arm[2])
72
73        if (self.hips[0][0] < 10 and self.hips[0][1] < 10) or
74                (self.hips[1][0] < 10 and self.hips[1][1] < 10):
75          left_signal_angle = 0
76          right_signal_angle = 0
77        elif self.left_arm[0][0] < 10 and self.left_arm[0][1] < 10 and
78                self.left_arm[1][0] < 10 and self.left_arm[1][1] < 10:
79          left_signal_angle = 0
80          right_signal_angle = 0
81        elif self.right_arm[0][0] < 10 and self.right_arm[0][1] < 10 and
82             self.right_arm[1][0] < 10 and self.right_arm[1][1] < 10:
83          left_signal_angle = 0
84          right_signal_angle = 0
85        else:
86          left_signal_angle = self.__calculate_angle(self.hips[0],
87                              self.left_arm[0], self.left_arm[1])
88          right_signal_angle = self.__calculate_angle(self.hips[1],
89                               self.right_arm[0], self.right_arm[1])
90
91        if self.is_facing:
92          if int(float(left_signal_angle)) in self.left_range and
93              int(float(right_signal_angle)) in self.right_range:
94            signal = 'stop'
95          elif int(float(left_signal_angle)) in self.reverse_range and
96                  int(float(right_signal_angle)) in self.reverse_range:
97            signal = 'reverse'
98          elif int(float(left_signal_angle)) in self.forward_range or
99                  int(float(right_signal_angle)) in self.forward_range:
100           signal = 'forward'
101         elif int(float(left_signal_angle)) in self.left_range and
102                 int(float(right_signal_angle)) not in self.right_range:
103           signal = 'right'
104         elif int(float(left_signal_angle)) not in self.left_range and
105                 int(float(right_signal_angle)) in self.right_range:
106           signal = 'left'
107         else:
108           print(Fore.CYAN, 'HIT ELSE')
109           signal = 'stop'
110       else:
111         if int(float(left_signal_angle)) in self.left_range and
112              int(float(right_signal_angle)) in self.right_range:
113           signal = 'stop'
114         elif int(float(left_signal_angle)) in self.reverse_range
```

```
115             and int(float(right_signal_angle)) in self.reverse_range:
116           signal = 'reverse'
117         elif int(float(left_signal_angle)) in self.forward_range or
118             int(float(right_signal_angle)) in self.forward_range:
119           signal = 'forward'
120         elif int(float(left_signal_angle)) in self.left_range and
121             int(float(right_signal_angle)) not in self.right_range:
122           signal = 'left'
123         elif int(float(left_signal_angle)) not in self.left_range and
124             int(float(right_signal_angle)) in self.right_range:
125           signal = 'right'
126         else:
127           print(Fore.CYAN, 'HIT ELSE')
128           signal = 'stop'
129
130       self.print_signal_info(left_arm_angle, right_arm_angle,
131       left_signal_angle, right_signal_angle, signal)
132     else:
133
134       print(Fore.RED, 'Signal: stop, no person in the image!')
135
136   def print_signal_info(self, left_arm_angle, right_arm_angle,
137               left_signal_angle, right_signal_angle, signal):
138     print(Fore.YELLOW, '----------------------Signal data ----------------------')
139     print(Fore.RED, self.right_arm)
140     print(Fore.GREEN, self.left_arm)
141     print(Fore.BLUE, f'Person is facing camera: {self.is_facing}\n')
142
143     # print(Fore.RED, f'Right arm angle: {right_arm_angle}')
144     # print(Fore.GREEN, f'Left arm angle: {left_arm_angle}\n')
145
146     print(Fore.RED, f'Right signal angle: {right_signal_angle}')
147     print(Fore.GREEN, f'Left signal angle: {left_signal_angle}\n')
148
149     print(Fore.MAGENTA, f'Signal: {signal}')
150     print(Fore.YELLOW, '-----------------------------------------------------------')
151
152   def __calculate_angle(self, point1, point2, point3):
153     """
154       Calculates the angle between three points.
155       """
156     a = np.array(point1)
157     b = np.array(point2)
158     c = np.array(point3)
159     ba = a - b  # Difference between a and b
160     bc = c - b  # Difference between c and b
161
162     cosine_angle = np.dot(ba, bc) / (np.linalg.norm(ba) * np.linalg.norm(bc))
163     angle = np.arccos(cosine_angle)
164     degrees = np.degrees(angle)
165     if degrees is None:
166       return 0
167     else:
168       return degrees
169
170   def run(self):
171     self.__get_signal()
```

Listing L.2: JamesAngle.py is responsible for calculating the angles from keypoints and return a steering signal the module uses JamesPredict (Appendix L.1)

Appendix M

Luke - Smartphone app: Install and run

M.1 What is Luke

Luke is a Android app made to run human pose estimation models. The app generates steering signals for manoeuvring autonomous vehicles based on predictions from a HPE model.

Link to Code on GitHub: https://github.com/wsvea-lochert/Luke

M.2 prerequisites

- Android Studio
- Android smartphone running Android 10 or higher

Note that the application will not run on a emulator as it needs the camera hardware.

M.3 Download

Run the command in Figure M.1 to download the package:

Figure M.1: Command for cloning Luke from GitHub.

M.4 Install and run

Open the project folder in Android Studio and build the project. When the project is done building plug your phone into your computer and run the project, the app will then be installed to your phone, and is ready to use.