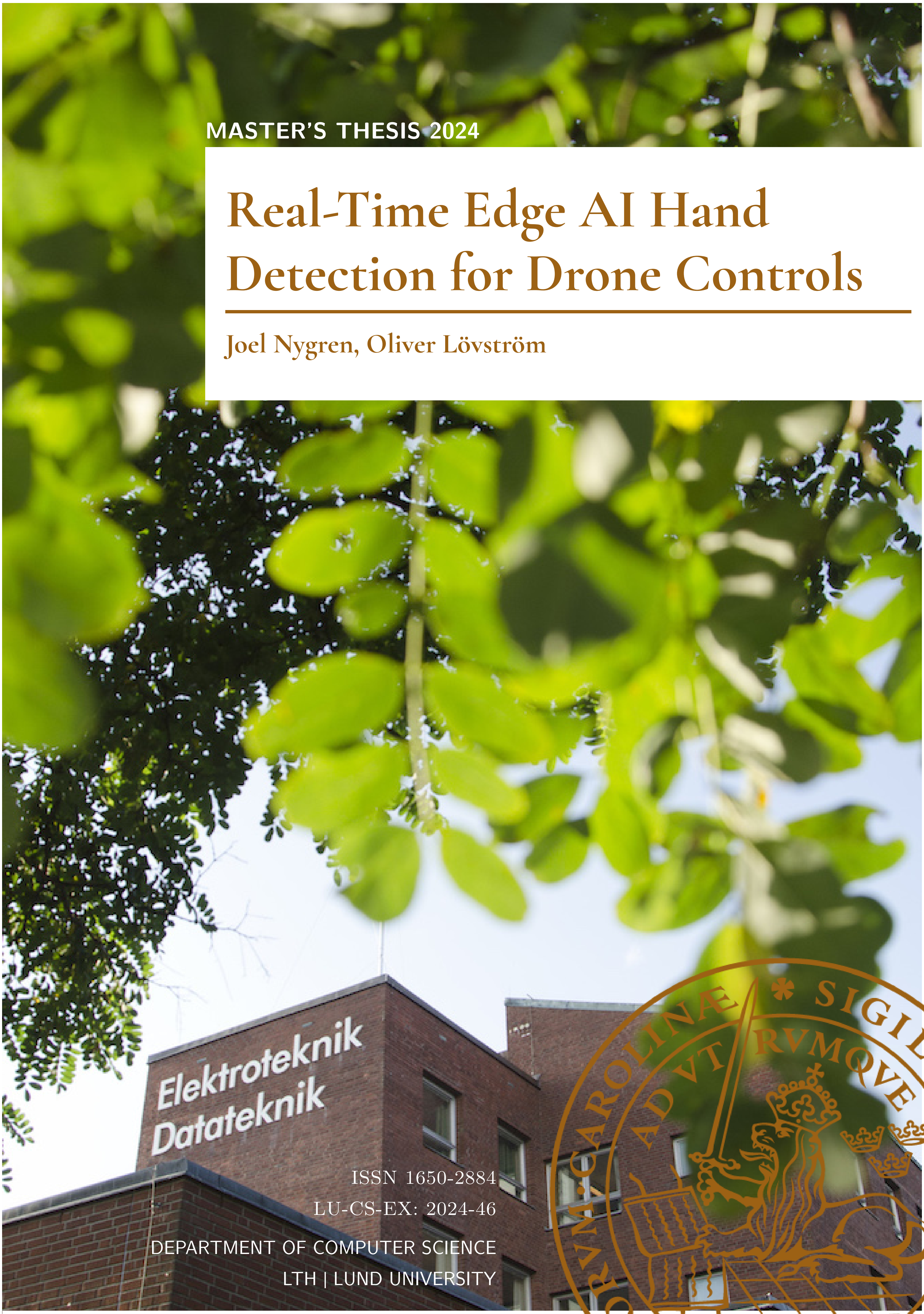
MASTER'S THESIS 2024

Real-Time Edge AI Hand Detection for Drone Controls

Joel Nygren, Oliver Lövström



ISSN 1650-2884
LU-CS-EX: 2024-46

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-46

Real-Time Edge AI Hand Detection for Drone Controls

Kontrollera Drönare med Handigenkänning

Joel Nygren, Oliver Lövström

Real-Time Edge AI Hand Detection for Drone Controls

Joel Nygren
jo7347ny-s@student.lu.se

Oliver Lövström
ol0464lo@student.lu.se

June 27, 2024

Master's thesis work carried out at Combine Control Systems AB.

Supervisors: Flavius Gruian, flavius.gruian@cs.lth.se
Irena Ilvesten, irena.ilvesten@combine.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

In recent years, advancements in computing and artificial intelligence have enabled complex computing on edge devices. Edge computing can be advantageous for inference time in real-time applications while removing the need for reliable connectivity.

To show potential challenges when deploying machine learning models to an edge device and how these challenges can be overcome, we approach a specific control problem. The goal is to make a Bitcraze Crazyflie, a nano-sized drone equipped with a camera to follow a hand.

We conclude that most of the challenges stem from the special properties of the onboard GAP8 processor, which makes it ultra-low power. These properties restrict the convolutional neural networks that can be deployed. We show how these challenges can be overcome by using suitable neural network architectures and training practices to achieve a tradeoff between precision and inference latency that is acceptable for our real-time application.

Keywords: GAP8, YOLO, Object Detection, AI deck, Crazyflie, AI

Acknowledgements

We would like to thank Simon Yngve and Irena Ilvesten for giving us the opportunity to conduct this master's thesis at Combine, providing the necessary material, useful insights and a pleasant environment to work in.

We would also like to thank our supervisor, Flavius Gruian, for the continuous feedback on this report.

Contents

1 Introduction **7**
1.1 Problem Definition . . . 7
1.2 Research Questions . . . 8
1.3 Division of Labor . . . 8
1.4 Layout . . . 9

2 Background **11**
2.1 Theory . . . 11
2.1.1 Robotics . . . 11
2.1.2 Object Detection . . . 11
2.1.3 Machine Learning . . . 14
2.1.4 Neural Networks . . . 14
2.1.5 Convolutional Neural Networks . . . 18
2.1.6 Quantization . . . 21
2.1.7 Control Systems . . . 23
2.2 Platform . . . 23
2.2.1 AI Deck . . . 24
2.2.2 Tools for DNN Deployment to the GAP8 . . . 25
2.2.3 Flow Deck . . . 26
2.2.4 Different Processors . . . 26
2.2.5 State Estimation and Different Controllers . . . 27
2.3 Related Work . . . 28
2.3.1 PULP-Frontnet . . . 28
2.3.2 MobileNet-Based Object Detection on the GAP8 . . . 28
2.3.3 Squeezed Edge YOLO . . . 29

3 Approach **31**
3.1 Method . . . 31
3.2 Implementation . . . 33
3.2.1 Reference Implementation . . . 33

3.2.2 Ultralytics 37
3.2.3 Dataset Creation 38
3.2.4 Model Selection 44
3.2.5 Squeezed Edge YOLO 44
3.2.6 Lightweight Models 47
3.2.7 Model Training and Evaluation 50
3.2.8 Integration 52

4 Evaluation 55
4.1 Metrics 55
4.1.1 Models 55
4.1.2 Full Implementations 56
4.2 Experimental Setup 57
4.2.1 Ultralytics 57
4.2.2 Workstation 57
4.3 Results 57
4.3.1 Model Selection 58
4.3.2 Transfer Learning 60
4.3.3 Hyperparameter Tuning 61
4.3.4 Quantization 62
4.3.5 Full Implementations 62
4.4 Discussion 63
4.4.1 Constraints on Networks and Data When Using GAPFlow 63
4.4.2 Lightweight Models 64
4.4.3 Flight Performance 65
4.4.4 Onboard Versus Remote Inference 66
4.4.5 Security and Ethical Implications 67

5 Conclusions 69
5.1 Further Work 70
5.1.1 Dataset and Training 70
5.1.2 Network Architecture 70
5.1.3 Control 71

Chapter 1
Introduction

Unmanned Aerial Vehicles (UAVs), commonly called drones, are a rapidly advancing technology with many possible applications in different fields, such as law enforcement, infrastructure inspection, and precision agriculture. With artificial intelligence (AI) integration, drones are transforming, unlocking new use cases when complex conclusions can be drawn based on sensor input. For wireless drones, machine learning models aiding control can be executed on-board using the drone's hardware or off-board at a ground control station. Onboard inference is an attractive feature because no additional hardware or signal reception is needed, and the latency can potentially be reduced [32].

This thesis explores the implementation of drone controls using object detection models running onboard a Bitcraze Crazyflie [6]. The Crazyflie platform, known for its compact and modular design, has gained significant adoption as a research drone in numerous academic institutions. With a module attached called the Flow deck, the Crazyflie can estimate its position onboard, enabling fully onboard control [5].

Recently, Bitcraze introduced the AI deck equipped with a camera and an ultra-low-power GAP8 processor [3], altogether weighing 4.4 grams. This functionality enables the drone to make decisions based on the camera's sensor input. However, the properties of this compact hardware, introduce additional challenges for deploying deep learning algorithms.

This thesis will explore the challenges of using object detection at the edge for drone control. Specifically, we will consider how machine learning models can be chosen, trained, and deployed to an ultra-low power device, such as the GAP8 processor on the AI deck, and how its output can be used to maneuver a nano-sized drone at the very edge.

1.1 Problem Definition

The objective of this thesis is to investigate the deployment of machine learning models on low-power embedded systems. To achieve this, we will use the Crazyflie drone with its AI deck and Flow deck to solve a specific real-time control problem. We will develop an interactive application where the Crazyflie should follow a person's hand, staying at a fixed distance from the hand and following when the hand is moved in any direction. The drone's rotation stays the same. This requires sufficient accuracy and latency in the well-explored object detection machine learning task where an object, in this case a hand, is both classified and located within an image that, in this case, is captured with the AI deck's camera.

1.2 Research Questions

Our research questions aim to answer how to develop and deploy deep learning models to the edge and what considerations we must make. We will also investigate how we can optimize the training practices to improve detection accuracy for our real-time application. In short, we will answer the following questions:

- What challenges arise when developing and deploying deep learning models to edge devices?
- What strategies and techniques can be used to address these limitations and challenges?
- How can training data be collected, and what training practices can improve detection accuracy?

The first question aims at identifying challenges that needs to be solved to achieve our real-time application. This gives a context to our other research questions.

Next, we will examine how to handle the platform's limitations, specifically how to adjust deep learning models to fit the target platform. Since we aim to solve a real-time control problem where inference time is crucial, we will investigate how to optimize machine learning models for inference time and detection accuracy and evaluate the trade-offs between these aspects.

The final question concerns the object detection model's accuracy. We hypothesize that training the network on platform-specific images will significantly improve detection accuracy. Therefore, we will investigate how to create a suitable dataset and evaluate its effectiveness. Additionally, we will explore training practices that could improve detection accuracy without increasing inference time.

1.3 Division of Labor

Table 1.1 outlines the division of labor during the two phases of our project: implementation and report writing. The implementation is further divided into six tasks.

The first task, control systems, was done together. Oliver then focused on sourcing and creating the datasets and the training procedures while Joel worked on model selection and deployment. Finally, we contributed equally to the evaluation metrics used to assess the implementation.

Regarding the report writing, Table 1.1 shows our contributions and the specific sections we each worked on. The report sections align closely with our respective responsibilities in the implementation phase.

Table 1.1: Division of labor per task.

Phase	Task		Joel	Oliver
Implementation	Controls		X	X
	Datasets			X
	Lightweight Models		X	
	Training Practices			X
	Model Deployment		X	
	Evaluation		X	X
Report	Introduction	1	X	X
	Theory	2.1.1-2.1.4		X
	Theory	2.1.5-2.1.6	X	
	Platform	2.2	X	
	Related Work	2.3	X	
	Method	3.1		X
	Implementation	3.2.1	X	
	Implementation	3.2.2-3.2.3		X
	Implementation	3.2.4-3.2.6	X	
	Implementation	3.2.8		X
	Metrics	4.1.1		X
	Metrics	4.1.2	X	
	Experimental Setup	4.2		X
	Results	4.3.1-4.3.3		X
	Results	4.3.4-4.3.5	X	
	Discussion	4.4.1	X	
	Discussion	4.4.2		X
	Discussion	4.4.3-4.4.4	X	
	Discussion	4.4.5		X
	Conclusions	5	X	X

1.4 Layout

From hereon, the report consists of 4 chapters. First, in chapter 2 Background, we outline some theoretical prerequisites, present the target platform, and how previous works have solved similar tasks. Then, in chapter 3 Approach, the overlaying methodology is first outlined as planned from the start, followed by a detailed description of the implementation. In the 4 Evaluation chapter, we explain our evaluation setup, present the final results, and discuss them. Finally, in chapter 5 Conclusion, we answer our research questions and suggest ideas for further work.

Chapter 2
Background

2.1 Theory

This section will cover the important aspects to consider when designing and deploying object detection deep neural networks (DNNs) to edge devices. We introduce the foundational concepts of robotics and object detection, which are essential for our thesis's control and hand detection parts. We will explore object detection and machine learning to understand how deep neural networks can be deployed to edge devices and optimized to improve detection speed and accuracy for real-time applications.

2.1.1 Robotics

At the core of our system is an intelligent agent—a drone capable of navigating from one location to another while accounting for various environmental conditions. The drone must predict its state, for example, position, based on its current status and the surrounding environment [31]. To facilitate this, the drone is equipped with sensors. These sensors provide data, which is processed to maintain an accurate understanding of the drone's state. The specific platform we will use is explained in detail in Section 2.2.

2.1.2 Object Detection

For the drone to follow a hand, it must be equipped with a camera and a method to identify the hand's location in the image. Object detection, a well-studied task within machine learning and computer vision, involves identifying objects in an image and pinpointing their locations using bounding boxes [31]. This capability enables the drone to track the hand accurately. During this section, we will assume that the network processes images.

One natural approach for object detection is using supervised machine learning classifiers. A supervised machine learning classifier takes an image as input and outputs a predicted

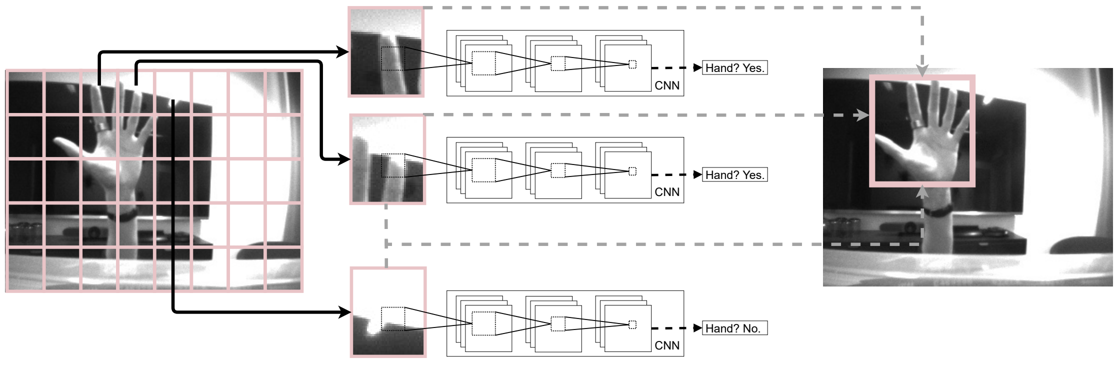


Figure 2.1: Object detection using sliding window approach. The image is divided into multiple parts, and the classifier network must be run multiple times to get bounding box predictions for each part. Based on the classifications, the bounding boxes are then computed.

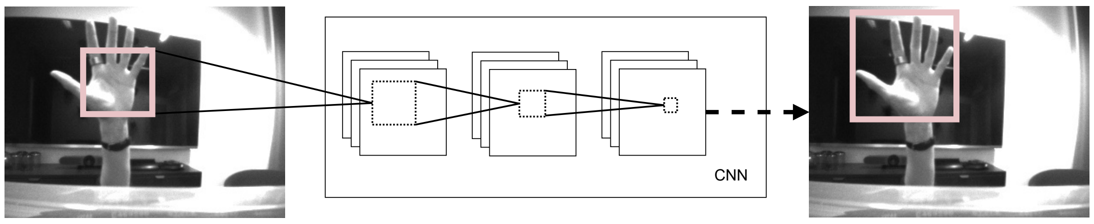


Figure 2.2: Single shot detection. By reframing the object detection problem as a regression problem, the output from single-shot detection gives all the bounding boxes, class, and respective confidence scores.

class. We can achieve object detection using a classifier together with a sliding window. An illustration of a sliding window approach can be found in Figure 2.1. First, the original image is split into multiple smaller overlapping images. Using a convolutional neural network (CNN), detailed in Section 2.1.5, a prediction is performed for each smaller frame, determining if an object is present within the image. Subsequently, high-confidence classifications are combined to construct the bounding boxes [31].

The issue with the above approach is that it is too slow for real-time applications. However, recent object detection models use single-shot detection (SSD), reframing the classification problem as a regression problem [30]. This means the objects can be detected, and bounding boxes are drawn from a single prediction; see Figure 2.2. Instead of running the machine learning algorithm multiple times for each window in the image, this approach looks at the entire image. This also reduces the number of false positives since the model can learn the context of the whole image. Additionally, it improves speed, making it suitable for real-time applications on edge devices [30].

Evaluation

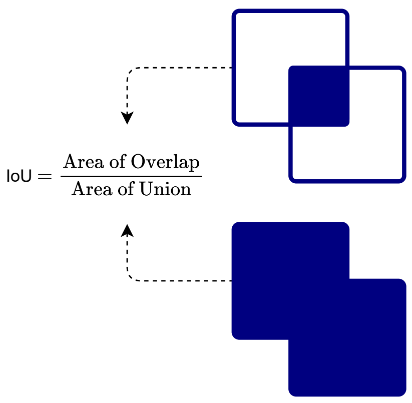
Evaluating object detection is essential for training and testing performance. In our thesis, we will use the following metrics to evaluate the detection model's performance: precision, recall, and mean average precision.

Precision measures the correctness of positive predictions, while recall assesses the model's ability to recognize all positive instances. The formulas for precision and recall are given in

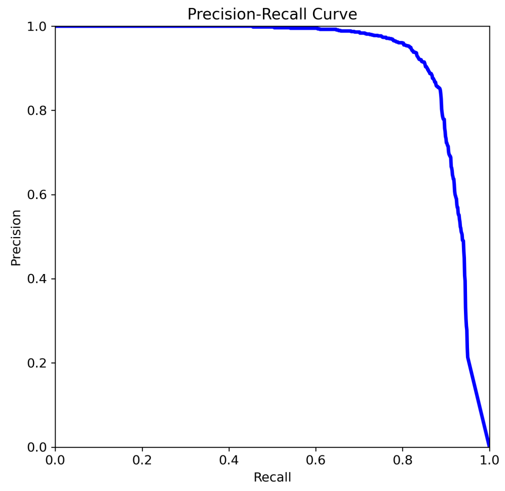
Equations 2.1 and 2.2, where TP are the true positives, FP are the false positives, and FN are the false negatives.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (2.1) \qquad \text{Recall} = \frac{TP}{TP + FN} \quad (2.2)$$

To classify if a bounding box is a true positive, false positive, or false negative. We must compare the ground truth to the predicted box. For this purpose, an intersection over union (IoU) is used. IoU measures the overlap between a predicted bounding box and the ground truth; see Figure 2.3a. To classify the prediction, we define an IoU threshold. The detection is a true positive if the actual IoU is greater than the threshold and the bounding box has not previously been matched. Otherwise, it is a false positive. All ground truths not matched by any prediction are counted as false negatives. From this, we can calculate the precision and recall using the formulas given in Equations 2.1 and 2.2.



(a) Intersection over union calculation. The IoU in the image is $\frac{1}{7}$.



(b) Precision-Recall curve at IoU threshold of 0.5.

Figure 2.3: Object detection evaluation metrics.

There is a trade-off between precision and recall. Therefore, we use the mean average precision (mAP), which captures this relationship in the metric. The mAP is given by Equation 2.3, where n is the total number of classes and AP(i) is the average precision for class i. In this thesis, we will work with one class, meaning the mAP equals the average precision. However, we will still use the mAP abbreviation since it is the standard abbreviation in the framework we use [20]. The average precision is given by the area under the precision-recall curve, the plot of the precision and recall for all predictions. Figure 2.3b gives an example of this precision-recall curve.

$$\text{mAP} = \frac{1}{n}\sum_{i=1}^{n} \text{AP}(i) \tag{2.3}$$

You Only Look Once

The approach we will use for object detection is the SSD machine learning algorithm, You Only Look Once (YOLO). The object detection machine learning problem is reframed as a regression problem by dividing the image into a $S \times S$ grid. For each cell, it predicts B bounding boxes, as well as a class confidence score for each box. The output from the model is the bounding box coordinates and the confidence score for each box. The confidence is given by the intersection over union (IoU) between the prediction and the ground truth. If the box contains no object, the confidence is close to zero and can be discarded [30]. This process is illustrated in Figure 2.4.

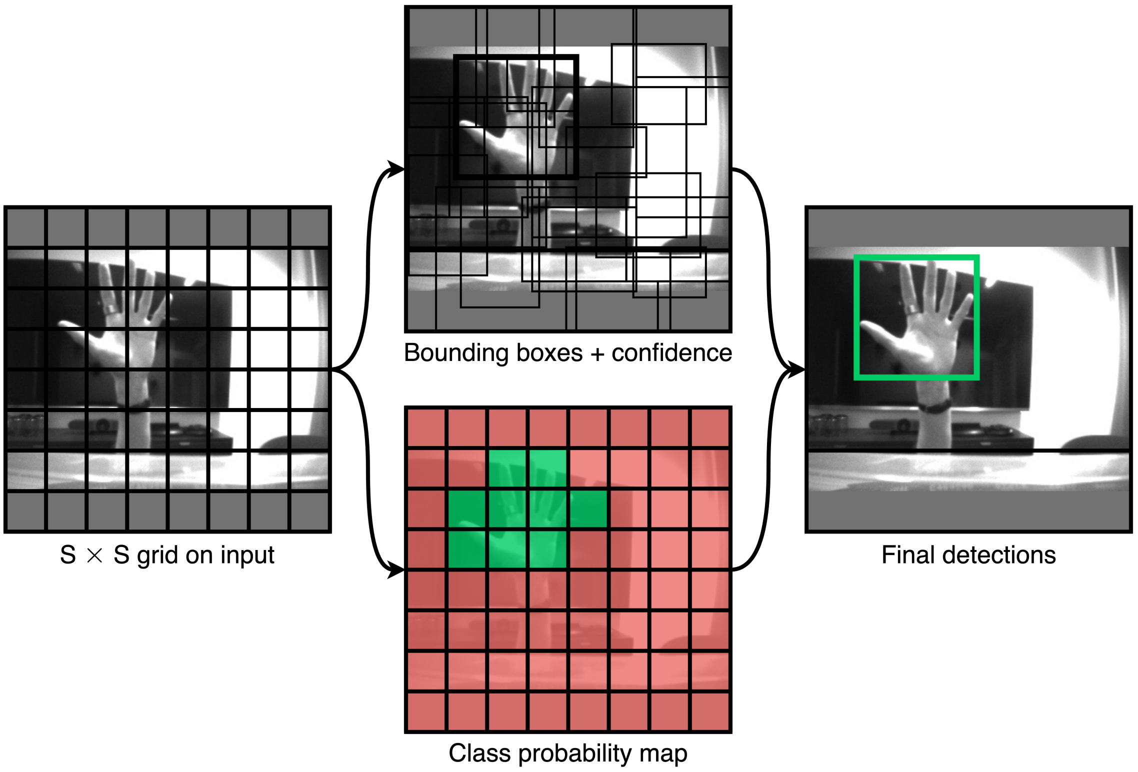


Figure 2.4: YOLO single-shot detection.

2.1.3 Machine Learning

While humans can easily identify hands in images, replicating this with deterministic algorithms is challenging. Machine learning bridges the gap by enabling machines to learn from data rather than requiring researchers to manually engineer features for these algorithms. [31]. By training on large datasets, machine learning models such as YOLO can learn to detect objects in images, enabling us to track the hand accurately.

2.1.4 Neural Networks

Object detection algorithms, such as YOLO use neural networks due to their ability to capture complex patterns in data. This capability makes neural networks particularly effective

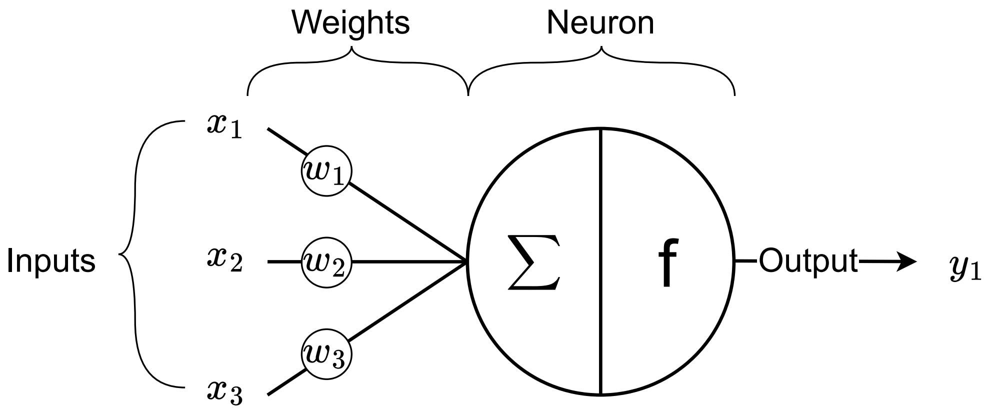


Figure 2.5: Machine learning neuron with three inputs and one output, this simple network is often called the perceptron. The network contains three weights commonly referred to as learnable parameters or parameters.

for image recognition and object detection tasks. A neural network consists of multiple layers, each comprising several neurons. Networks with multiple layers are often referred to as deep neural networks. To better understand the concepts of inference, activation functions, and training, we will take a closer look at the neuron, as these concepts can then be extrapolated to larger networks containing numerous neurons.

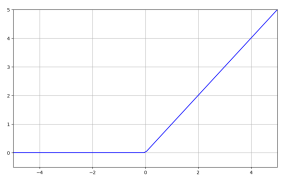
Neuron and Inference

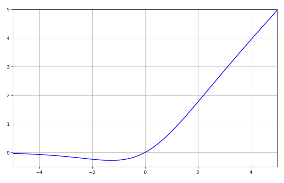
The neuron is the most basic component in a neural network algorithm, depicted in Figure 2.5. A neuron's function is to take several inputs and output a prediction, a process often called inference. Inference, also known as a forward pass, involves receiving multiple inputs, which are multiplied by weights and summed to compute the output. The summed values are then passed through an activation function to achieve a non-linear input-output relationship [31].

Inference and, specifically, inference time is crucial when deploying the algorithm on real-time edge AI devices because it directly affects the system's responsiveness and performance in practical applications.

Activation Functions

The role of the activation function is to achieve a non-linear input-output relationship. We will use three different activation functions in different situations to tweak the tradeoff between accuracy and latency. In recent years, the rectified linear unit (ReLU) has become popular because of its simplicity and fast execution, see Figure 2.6a. However, because its gradient is zero for all negative inputs, it might hinder training convergence. Leaky ReLU tries to alleviate this problem by having a slight slope even for negative inputs [26], see Figure 2.6c. Some state-of-the-art object detection models use Sigmoid-weighted linear units (SiLU) as an activation function, see figure 2.6b, which has a global minimum that gives a regularizing effect, avoiding large weights [9]. Though SiLU has some attractive properties, our target platform has special support for Leaky ReLU, which is why we use it in our deployed models.


(a) ReLU


(b) SiLU

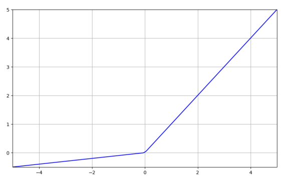

(c) Leaky ReLU

Figure 2.6: Three different activation functions.

Training

As stated earlier, the main function of the neuron is to take several inputs and output a prediction. To achieve good predictions given the input, the network must first be trained. Training a network involves updating its weights and biases based on the loss, a process known as backpropagation. Training can be broken down into three steps:

1. **Forward Pass:** Pass inputs through the neuron and obtain the prediction.
2. **Loss Calculation:** Calculate the error in the output using a loss function. The loss function shall be a good measure of the difference between the predicted output and the target value.
3. **Backpropagation:** Compute the gradient of the loss function with respect to the weights and bias. Adjust the weights and bias using an update rule, moving in the direction that decreases the loss with a learning rate η.

There are two approaches to training a neural network: training from scratch and transfer learning. In the first method, the network weights are initialized randomly. In the second approach, we use an already-trained model with predefined weights, which we then fine-tune using a different dataset. The advantage of using transfer learning is that it can reduce the data required to achieve good performance [39].

Warmup

In deep neural networks, optimization instabilities can occur if a large learning rate is used at the beginning of training. This problem can be solved using a less aggressive learning rate at the start of training. In this thesis, we will use a gradual warmup, which means increasing the learning rate with a constant for each warmup epoch to achieve the final learning rate [10].

Overfitting

A problem that can occur in DNNs is overfitting. Overfitting means the model fits the training data too well but fails to generalize to unseen data. Finding the optimal number of parameters to model the problem properly is challenging. If the number of parameters is too few, the model cannot capture complex input-output relationships, leading to underfitting. Conversely, if the number of parameters is too large, the model becomes too complex and overfits the training data. It is common to create a larger network and then use techniques to reduce overfitting. This report will explore the effect of regularization and data augmentation.

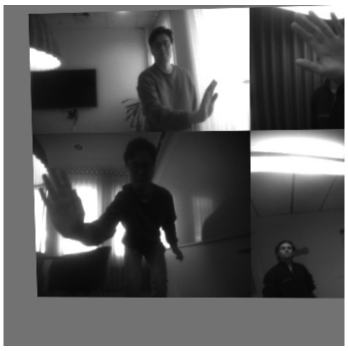
We will use L2 regularization, or weight decay, to prevent overfitting by adding a penalty for large weights, discouraging overly complex models. This introduces an additional parameter to the training.

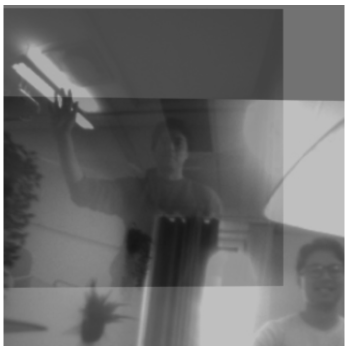
Data augmentation is another technique to improve generalization performance and reduce overfitting [37]. It increases image variety by applying transformations to each image, such as rotation, translation, or shearing. For augmentation, a policy is decided to perform the augmentation. In this thesis, we will use the RandAugment policy, which we have chosen because of its simplicity. RandAugment sequentially applies transforms to an image and can be described in a few lines of code; see Algorithm 1.

Neural networks require a fixed image size for input. However, image transformations can alter the image size. To address this, letterboxing ensures the data fits the specified dimensions. This process involves resizing the image to the required size while maintaining the original aspect ratio. The image is then padded with gray pixels to match the model's input size [20]. Examples of augmented training samples can be seen in Figure 2.7.

Algorithm 1: RandAugment

```
1: Initialize: transforms = [translate, scale, shear, perspective, fliplr, mosaic, mixup]
2: function RandAugment($N$, $M$)
3:     Input: $N$: Number of transformations to apply sequentially
4:     Input: $M$: Magnitude for all transformations
5:     sampled_ops ← RandomChoice(transforms, $N$)
6:     return $\{(op, M) \mid op \in$ sampled_ops$\}$
7: end function
```


(a) Augmentation example 1.


(b) Augmentation example 2.

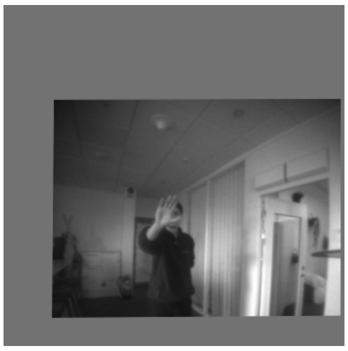

(c) Augmentation example 3.

Figure 2.7: RandAugment data augmentation.

Hyperparameter Tuning

In the previous sections, we discussed various parameters impacting training and generalization performance. These parameters, known as hyperparameters, include adjustments such

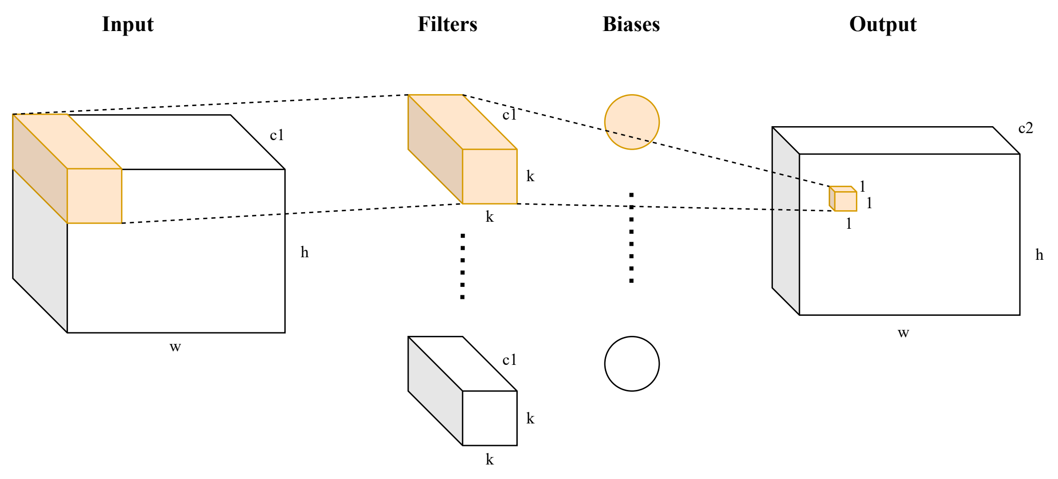


Figure 2.8: The parts involved in executing a convolutional layer. The highlighted parts are involved in the computation of a single output element.

as weight decay to increase regularization or changes to augmentation parameters to enhance the probability or scale of certain transformations. Hyperparameter tuning aims to find good hyperparameters that optimize the network's performance on validation data, with the overall goal of improving the model's performance on unseen data.

A natural approach would be to test several different networks with different combinations of hyperparameters. However, the vast number of hyperparameters and their infinite combinations make this a challenging task. To address this, hyperparameter tuning algorithms are designed to accelerate the tuning process. This thesis will utilize the Asynchronous Successive Halving Algorithm (ASHA), a straightforward hyperparameter optimization method that leverages aggressive early stopping. ASHA builds upon the Successive Halving Algorithm (SHA) [23].

The algorithm functions by taking a defined hyperparameter space, a number of configurations, and a stopping epoch. Using the hyperparameter space and the specified number of configurations, SHA generates various candidates. Each candidate is then evaluated after a few training epochs. Only the top-performing configurations are kept, and their training continues for more epochs. This process is repeated until the predefined number of epochs is reached. ASHA improves upon SHA by maximizing parallelism and optimizing speed, significantly enhancing the efficiency of the hyperparameter tuning process [23].

2.1.5 Convolutional Neural Networks

For our hand detection task, we will use CNNs, a type of neural network suitable for image inputs widely used for computer vision. A CNN contains at least one convolutional layer, see Figure 2.8, that implements the mathematical operation of discrete convolution. 2D convolution is typically used when dealing with images. It can be thought of as representing the input image as a tensor, a matrix with more than two dimensions, and sliding a kernel, also known as a filter, over the image, multiplying the kernel weights elementwise with input elements and summing them along with a bias. The filter is moved a number of pixels at a time, called the stride [16].

MACs and Parameters

When porting CNNs to an embedded system, optimizing its inference time and, thus, computational cost is important. To measure neural networks' computational cost, it is common to count the number of multiply-accumulate (MAC) operations, where a product is computed and added to a sum. MACs scale with, but are not the same as, the number of model parameters. Keeping the number of parameters low is important to us for the models to fit in the drone's small memory and minimize memory accesses.

In a conventional convolutional layer, with square filters of size k, stride one, and where the c_1 channel input of size $w \times h$ is padded to make the c_2 channel output have the same size, the MACs and number of parameters are computed as in Equations 2.4 and 2.5. We used these during optimization to verify that a layer's latency can be attributed to its sheer number of MACs and memory accesses to its number of parameters. This was then optimized. The MACs and model size are presented for our model in the Results section, and the part attributed to convolutional layers was calculated with these formulas.

A convolutional layer is shown in Figure 2.8. There are $c_1 \times k^2$ weights and one bias for each of the c_2 filters. Each of these parameters corresponds to one MAC when the filters are applied in $w \times h$ positions over the input. In practice, biases are often multiplied by one to be treated identically to the weights, simplifying execution and enabling optimizations [16], which is why they also result in MACs.

$$\text{MACs} = (c_1 \times k^2 + 1) \times c_2 \times w \times h. \tag{2.4}$$

$$\text{Number of Parameters} = (c_1 \times k^2 + 1) \times c_2. \tag{2.5}$$

Depthwise Separable Convolution

To decrease the number of MACs and parameters in the CNN models we deploy, we will try replacing some convolutional layers with a sparse approximation of them called depthwise separable convolution. They consist of two sequenced convolutional layers and an activation function after the last. With the same notations and assumptions as in the previous section, the first layer, implementing depthwise convolution, is shown in Figure 2.9. It has c_1 kernels of size $k \times k$ and depth one, each only operating on a single input channel. The second layer, implementing pointwise convolution, is shown in Figure 2.10 and has c_2 kernels of size 1×1 and depth c_1, each operating on all input channels.

The MACs and number of parameters are computed as in Equations 2.6 and 2.7. When comparing to 2.4 and 2.5, the difference is that the number of MACs and parameters do not scale with output channels c_2 in the depthwise convolution while the pointwise part is independent of kernel size k. Using this property, we made our models faster and smaller.

Because the depthwise convolution captures intra-channel relationships, and pointwise convolution captures inter-channel relationships, it is possible to capture relationships between input elements that are so close they fit inside the same kernel, regardless of channel, just like in normal convolution. But depthwise separable convolution inarguably has fewer parameters and worse representation ability than normal convolution[16]. When replacing convolutional layers with depthwise separable convolution, we hope it had redundant representation ability to begin with so we can achieve lower latency with little loss in accuracy.

$$\text{MACs} = [(k^2 + 1) \times c_1 \times w \times h] + [(c_1 + 1) \times c_2 \times w \times h] \tag{2.6}$$

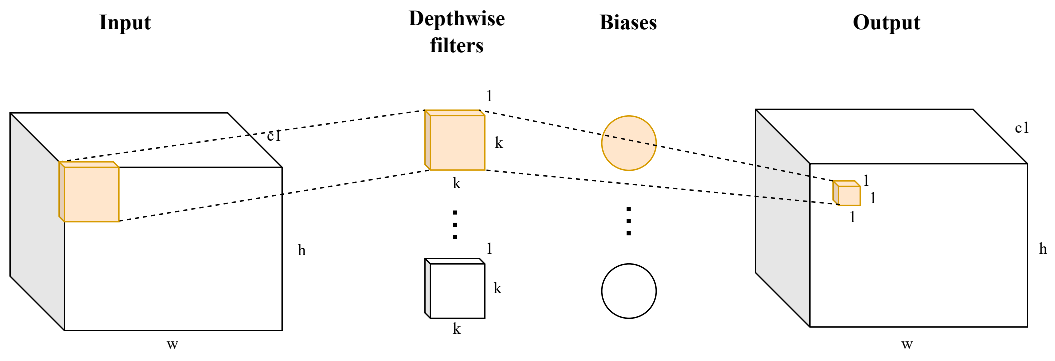


Figure 2.9: The depthwise part of depthwise separable convolution. Each filter is only applied to one input channel. The number of filters and, hence, output channels are equal to the number of input channels.

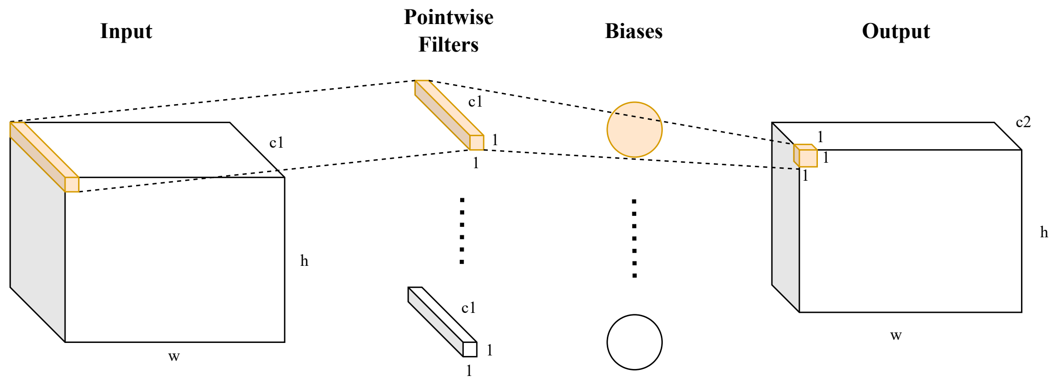


Figure 2.10: A pointwise convolutional layer; the second step in depthwise separable convolution.

$$\text{Number of Parameters} = [(k^2 + 1) \times c_1] + [(c_1 + 1) \times c_2] \tag{2.7}$$

Light Convolution

We will also try replacing some convolutional layers with a variation of depthwise separable convolution, where the pointwise convolution is performed before the depthwise. This variation will hereafter be referred to as light convolution. If the output contains fewer channels than the input, it reduces model size and MACs slightly compared to conventional depthwise separable convolution by having one depthwise kernel per output channel, instead of per input channel, see Equations 2.8 and 2.9. The operation order in itself is unimportant [8], but the reduced number of parameters might hurt the model's ability to capture complex patterns.

$$\text{MACs} = [(k^2 + 1) \times c_2 \times w \times h] + [(c_1 + 1) \times c_2 \times w \times h] \tag{2.8}$$

$$\text{Number of Parameters} = [(k^2 + 1) \times c_2] + [(c_1 + 1) \times c_2] \tag{2.9}$$

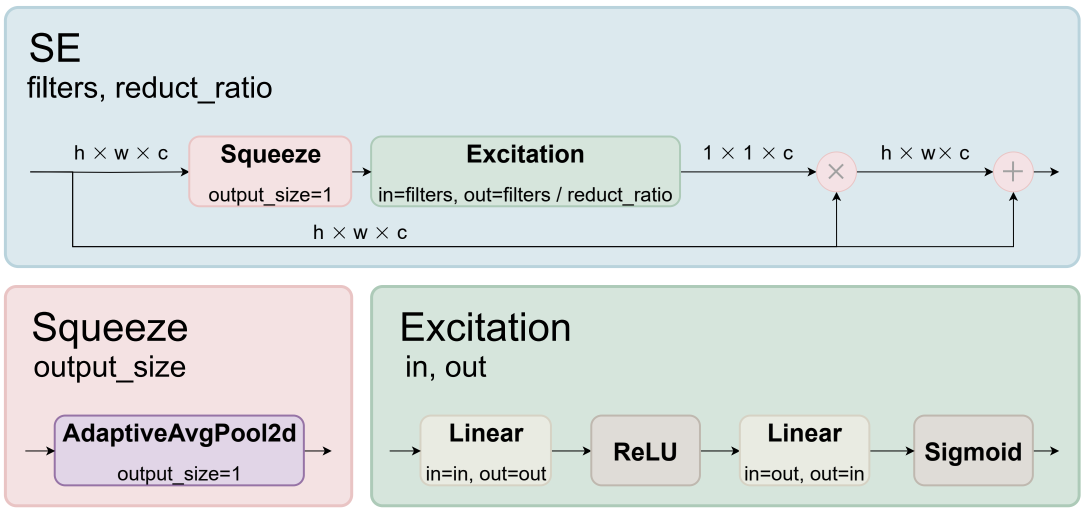


Figure 2.11: A Squeeze-and-Exciation block.

Squeeze-and-Excitation

In our final model, we will intersperse CNN layers with squeeze-and-excitation blocks. This type of layer, first introduced in [17], can be used between convolutional layers to improve performance by emphasizing which channel of features is most important. The block contains two components: squeeze and excitation; see Figure 2.11.

In the first step, weights for channel importance are calculated. The squeeze operation compresses each channel's information into a single value. This is achieved by applying average pooling to the feature maps, which reduces each channel's spatial dimensions to a single value.

In the second step, decisions are made about which features are important. The excitation operation takes the squeezed output and processes it through two fully connected layers. The first layer uses a ReLU activation function, and the second uses a sigmoid activation function to generate a weight for each channel. These weights determine the degree to which each channel's features should be emphasized or suppressed.

In the final step, the original input is multiplied by the importance weights from the excitation component, allowing the network to dynamically adjust the emphasis on different feature channels. This process helps the network emphasize feature importance, increasing accuracy.

2.1.6 Quantization

As our platform does not support larger datatypes, we will need to quantize our model before deploying it. Quantizing a deep neural network entails representing its weights, biases, and outputs with smaller datatypes. Floating point types are often replaced with integer types. This reduces the memory footprint during inference and can increase inference speed because arithmetic operations on smaller datatypes are typically faster, as are integer operations. This operation replacement is sometimes vital to eliminate operations not supported on the target platform [21]. In this section, the theory concerning quantization is laid out for the purpose of understanding our choice of quantization scheme and how it affects inference precision, memory footprint, and inference time.

Schemes

Different quantization schemes can be applied to achieve the quantization. They are classified into two categories: quantization-aware training, where the quantization is performed during training, and post-training quantization, which is performed after training a floating-point model. Furthermore, quantization schemes use different quantizers, which are ways of mapping between the resulting quantization levels represented by a smaller datatype and the real values represented by the original datatype. Because a smaller datatype can represent fewer values than a larger one, some precision will inevitably get lost, and different quantizers have different tradeoffs between precision, memory complexity, and time complexity during inference [21].

Uniform Symmetric Quantizer

We used a post-training quantization scheme with a uniform symmetric quantizer because it was the simplest scheme supported by our platform, and the results were sufficient. Uniform in this context means the steps between quantization levels correspond to equal steps in the real values, and symmetric means the quantized values represent ranges symmetric around zero. In practice, the actual mapping function between a real value r and its quantization level q is $r = S \cdot q$, where $q \in [-127, 127]$ and S is a 32-bit float. An asymmetric uniform quantizer would also have a constant int8 Z to shift the whole range of real values. The mapping would be $r = S(q - Z)$. Multiplying two real numbers r_1 and r_2 would then entail extra operations, which is why a symmetric mapping is simpler and faster [19].

Keeping Quantization Steps Small

In a uniform quantizer, the steps between real values that the quantization steps represent become larger if a larger range of values has to be represented. Therefore, the range should be kept as small as possible by using small-scale values S. In the used quantizer, all values in the same channel of weights or activations share the same scale value, S, in contrast to having one scale value per tensor. This allows for smaller-scale values that only have to map the quantized range $[-127, 127]$ to fit the smallest and largest real value in the channel and not in the whole tensor. Finding the scale values for weights is straightforward in post-training quantization because they are static, but for dynamic activations, statistics of the values are collected during inferences on sample data [34]. For both weights and activations, outlier values can result in the S value being unnecessarily large to include them, which is why different clipping methods can be applied. For example, only values that are within three standard deviations of recorded values can be included. In quantization-aware training, the quantization ranges can instead be learned to favor high precision in important intervals, which is why quantization-aware training can potentially achieve lower quantization errors [19]. Going back to symmetric contra asymmetric quantizers, an asymmetric quantizer can map to a smaller range if large real values only appear with one sign.

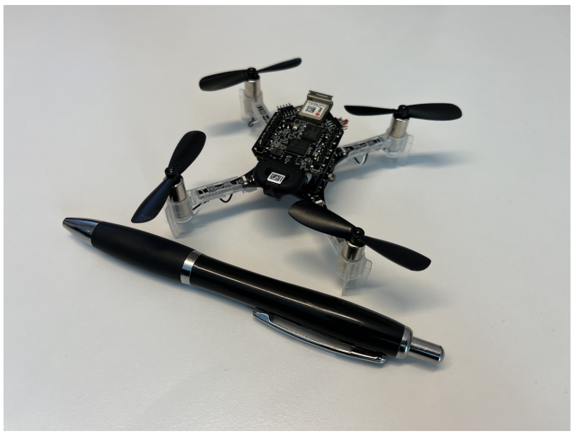


Figure 2.12: A Bitcraze Crazyflie 2.1 with AI deck 1.1 and Flow deck v2 attached.

2.1.7 Control Systems

A control system aims at making a variable reach and keep a desired value, called a setpoint. The variable could be anything from a temperature to a pressure but in this thesis it will be discussed as a position or velocity of a drone.

PID Controller

This thesis and some of our related works, use proportional-integral-derivative (PID) controllers to influence a controlled variable, for example a position, by iteratively measuring it and calculating an error from its setpoint. The manipulated variable, such as a velocity, is set to the sum of three terms proportional to the error, integral of the error and derivative of the error.

Kalman Filter

A Kalman Filter is an algorithm used to iteratively update the estimate of a value, such as a drone's position, by combining predictions based on previous estimates with new measurements to compute the next estimate. We use this to estimate the drone's position and one of our related works also employs a Kalman filter to track the drone's target.

2.2 Platform

To achieve our objective of controlling a drone using hand detection, we will use the Bitcraze Crazyflie 2.1 equipped with the AI deck 1.1 and Flow deck v2. This section lays out all the information necessary about this platform to motivate our choice of approach and evaluation as well as draw conclusions later. Bitcraze Crazyflie 2.1 is a palm-sized drone depicted in Figure 2.12, mainly used for research and education indoors. It has a battery life of around 5 to 7 minutes and is based on an STM32 microcontroller that runs the operating system FreeRTOS. In its firmware, the Crazyflie can estimate its own state, including position and velocity, using a gyroscope, accelerometer, and pressure sensors. Programs can be written to control flight, LED lights, and communicate with a personal computer via radio, among other functions. The functionality of the Crazyflie can be extended with hardware modules called decks. We will use Flow deck v2 to add extra sensory capabilities and AI deck 1.1 to add a camera and an extra processor specialized for executing neural networks.

2.2.1 AI Deck

The AI deck 1.1 is a hardware module attached to the Crazyflie equipped with WiFi communication capabilities, a Himax HM01B0 ultra-low power 320x320 monochrome camera, and a GAP8 processor for execution of Deep Neural Networks [3]. The Himax camera outputs 8-bit monochrome images with an active resolution of 324x324 but also supports a Quarter Video Graphics Array (QVGA) resolution of 320x240, in which case the actual output is images with a 324x244 active resolution [15].

GAP8

The main processor of the AI deck is a GAP8. It is manufactured by GreenWaves Technologies and consists of nine cores based on the RISC-V instruction set architecture. One core, the fabric controller (FC), is a high-performance microcontroller working at a maximum clock frequency of 250 MHz and is capable of sending tasks to the other eight cores, called the cluster (CL), and communicating with peripherals, which in the case of the AI deck include the camera, the WiFi module, extra memory, and the Crazyflies main microcontroller. The cluster comprises the remaining eight cores working at a maximum clock frequency of 175 MHz [13] and executes computationally heavy tasks in parallel. The clock frequencies of both the fabric controller and the cluster can be set dynamically to save battery [11]. It is worth mentioning that the GAP8 does not support floating point arithmetic but 8, 16, and 32-bit integer arithmetic. To use the 4x 8-bit SIMD MAC instructions, 8-bit integers should be used [14].

Memory Hierarchy

To save power, the nine cores of the GAP8 use different parts of an interesting memory hierarchy, which must be understood to make full use of it. Though we will later introduce tools that handle much of this automatically, understanding the memory hierarchy explains why the tools are needed and what their limitations are. The GAP8's internal and peripheral memory is organized as follows. The FC has its own 4 kB instruction cache, while all cluster cores share a 16 kB instruction cache. These are automatically updated. For data, the FC owns 8 kB of level 1 memory, and the cluster cores share 64 kB of level 1 memory that can be accessed within one clock cycle. Both FC and CL share a 512 kB level 2 memory that takes multiple clock cycles to access. One or more considerably slower level 3 memories can also be attached. To save power, the three levels of data memory are not mapped with a conventional cache mapping scheme to allow the processors to work with a single address space and abstract away transfers between the memory levels. Instead, transfers between the data memories are handled on a software level [11]. On the AI deck, two external level 3 memories are attached to the GAP8. One 8 MB memory called HyperRAM and one 128 MB persistent memory called HyperFlash. The HyperFlash is the GAP8's only persistent memory on the AI deck [3].

Porting DNNs to GAP8

To summarize some challenges when porting neural networks to the GAP8 processor, integer arithmetic, preferably with 8-bit precision, should be used, and the application must handle the transfer between the three memory levels. Managing the memory hierarchy is especially challenging since not even a single monochrome image from the Himax Camera with 8-bit pixels fits in the cluster's L1 memory. 324x244 = 79,056, which is larger than the 64,000 Byte L1 Memory. The data loaded into L1 should also be splittable in a way that allows the eight cluster cores to work on it in parallel. For complex networks, all weights cannot be kept in the L2 memory or even L3 HyperRam simultaneously. All instructions and data for the entire operating system and application must fit into the 128 MB HyperFlash.

2.2.2 Tools for DNN Deployment to the GAP8

To ease the deployment process to the GAP8 processor, GreenWaves Technologies provides the GAPFlow toolset [12]. This section briefly presents two of these tools to help readers understand why we chose to use them, what constraints they placed on the neural networks we used, and what impact they had on the results.

NNTool

The first tool in the GAPFlow toolset is the NNTool [12], which can load trained networks from TFLite or ONNX files, which are file formats for representing neural networks. In the case of TensorFlow Lite, an already quantized network can be loaded. Only a subset of all the neural network operations that TFLite and ONNX can represent are supported by GAPFlow. For example, cropping and padding are not fully supported yet. But the most common CNN operations are supported, it has successfully been used before, for example in [28]. After loading a neural network, NNTool can be used to modify it to be more suitable for running on the GAP8. The operations we used were adjusting, meaning the dimension order of tensors is changed from [Height]x[Width]x[Channel] to [Channel]x[Height]x[Width], fusing, meaning operations are fused wherever possible, and quantized.

The quantization is a post-training quantization where first, statistics about activation values are collected during inferences on sample data. Then, the network graph is traversed, applying a quantization according to the collected statistics and other constraints. Quantization data can also be loaded from a TFLite file, in which case the asymmetric quantization supported in TFLite is converted to a symmetric one. There are different options for, among other things, clipping methods and types of quantizers. By default, a uniform symmetric quantizer is used, with the quantized values represented as 8-bit integers.

By using NNTool for quantization, modern machine learning libraries like PyTorch or TensorFlow can be used to speed up the development of a model even though they do not directly support a quantization scheme most suitable for the GAP8. It also ensures that only hardware-supported integer arithmetics are used and saves memory, which, in turn, decreases inference time by limiting memory accesses.

After modifying the network, a state representing the modified network can be saved, which the next tool in the GAPFlow can read.

Autotiler

The second tool in GAPFlow is the Autotiler [11], which reads the state saved by the NNTool and generates C code implementing the network inference. The Autotiler can generate C code for common neural network layers such as convolution, ReLU, max and global pooling, fully connected, and more.

Remember that the GAP8 reduces power consumption by not having a data cache. Instead, it has memory levels with different address spaces that the application must efficiently use. This is where the Autotiler comes in. Given a memory budget in the L1, L2, and L3 memory and a saved network state from NNTool, the Autotiler tries to implement each network layer to optimally utilize the eight cluster cores and the memory hierarchy of the GAP8.

Consider, for example, a convolutional layer, where input tensor values are multiplied with kernel weights, accumulated with biases, and stored in an output tensor. Optimally, input, weights, and output should fit in the L1 memory simultaneously. However, as earlier mentioned, an entire input image might not even fit into the L1 memory, and the same applies to larger intermediate activation tensors. The Autotiler solves this by splitting the input and output into corresponding stripes called tiles allocated in L1 at the same time and letting the eight cluster cores perform convolutions in different areas of the tile. To minimize accesses to L2, the input is split into as few tiles as possible, making them as large as possible, under the constraint that an input tile, an output tile, a kernel, and a few other parameters must fit into the L1 simultaneously. As standard, the tiles are vertical.

It is up to the developer to only use convolutional layers that are possible to tile with respect to their input sizes, number of input channels, kernel sizes, strides, and number of kernels. Furthermore, all model parameters must fit in the persistent L3 memory.

The Autotiler makes it possible to deploy and optimize new network architectures with weights requiring way more memory than the L1 and L2 memories within minutes. This would take an incomparably longer time to implement manually in C, with the risk of errors and less efficient utilization of the hardware.

2.2.3 Flow Deck

To estimate the position of the drone, we will use the Flow deck v2. The Flow deck is a hardware module attached below the Crazyflie with sensors that enable the drone to estimate its position and rotation in three dimensions, along with the standard sensors. The height above ground is measured with a time of flight sensor and is, therefore, absolute with a few millimeters of precision. The position in the horizontal plane, on the other hand, is only known relative to the last position by measuring optical flow with a downward-facing camera. An absolute position is, however, estimated [5].

2.2.4 Different Processors

This section explains the roles of different processors we will develop towards and their means of communication in the Crazyflie ecosystem, as depicted in Figure 2.13. The main CPU of

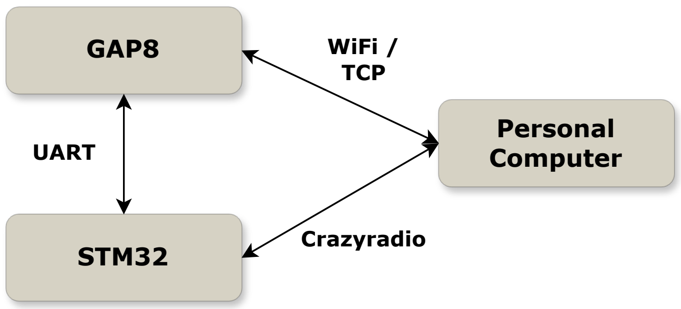


Figure 2.13: The different processors we will be using and their means of communication in the Crazyflie ecosystem.

a Crazyflie is an STM32 microcontroller that is programmable with custom C firmware to, among other things, get the drone's estimated state and set a wanted state [6]. The GAP8 is the AI deck's main processor, which can be programmed with a completely different C firmware to use the camera, stream data over WiFi, execute CNNs on its for the purpose optimized hardware, and communicate with the STM32 over a wired protocol called UART [6]. From a personal computer, Bitcraze's Python library cflib can be utilized to communicate with the GAP8 over WiFi/TCP and with the STM32 using a USB radio antenna called CrazyRadio [6]. The CrazyRadio can handle lower payloads than WiFi/TCP but has a more predictable latency [4].

2.2.5 State Estimation and Different Controllers

Out of the box, the Crazyflie has a stabilizer loop running, where its state is estimated based on sensor input, and motor voltages are computed based on a desired state, in our case, a position. The desired state will hereafter be called setpoint, which is a general term for a desired value in a control system. The gyroscope, accelerometer, optical flow, and height values are input into a Kalman filter that estimates the drone's position, velocity, and rotation. Attitude, in this context, means orientation. After the state estimation comes the controller, which takes a desired state called a setpoint consisting of a position or a velocity and an attitude or an attitude rate and outputs the desired power distribution to the motors. The standard controller is a cascaded PID controller with one PID controller for position, velocity, attitude, and attitude rate working at 100, 100, 500, and 500 Hz, respectively. When controlling towards a setpoint desiring a certain attitude rate, which is the mode closest to controlling motor voltage, only the attitude rate controller is used. When controlling position, which is the mode that offers the most abstraction, the position controller outputs a desired velocity to the velocity controller and the velocity controller a desired attitude to the attitude controller, and so on [6]. While controlling the position or velocity of the drone, the yaw, that is, the rotation around the vertical axis, but not the roll nor the pitch, can be controlled, as those must be used to achieve the appropriate acceleration [7]. Choosing what to control and, thus, how many layers of this cascaded PID controller to use will be discussed later.

2.3 Related Work

In this section, we will review previous works on deploying neural networks to the GAP8 processor. We arrived at these after searching LUBSearch for GAP8, and Google for GAP8 and object detection. The presented works are those we could find that specifically tackle object detection on the GAP8 or a similar machine-learning problem for a similar control problem with the AI deck.

2.3.1 PULP-Frontnet

Our idea is much inspired by [29], where a group of researchers made a Crazyflie with Flow deck and AI deck stay in front of a human's face at a fixed distance, following the person's translation and rotation around the vertical axis. Earlier in [27], an overlapping group of researchers compared an end-to-end approach where control signals are output directly from a machine learning model and a mediated approach outputting a high-level state containing the person's estimated pose and then using additional logic to compute control signals. The results were very similar. PULP-Frontnet suggested in [29] uses a mediated approach, taking a 160x96 image and outputting x, y, z, and θ representing a person's position and rotation. The prediction is sent from the GAP8 to the STM32, where an estimated state of both the human's current position and rotation, as well as its velocity and angular velocity, is updated in a Kalman filter. A setpoint containing the desired velocity and angle rate is computed from the drone's position and the human's position and velocity so that the drone can catch up within a fixed time period.

To achieve an impressive inference speed of 48 Hz, they reduced the feature map sizes by a factor of 16 within the first two layers by employing 5x5 convolution followed by max pooling, both with stride 2. They also made efficient use of the GAP8's hierarchical memory structure, using L3 only when necessary, and an 8-bit quantization scheme. Scaling down the input size to 80x48, they achieved 135 fps, still with good performance. PULP-Frontnet contains seven convolutional layers and one fully connected, resulting in 14.1 MMAC in the larger version operating at 48 fps and 4 MMAC in the smaller version operating at 135 fps.

Unlike their approach, we try to follow a hand instead of a face. We follow only translation, skipping rotation. Instead of having a network output for distance to the target, we output object width and height in the image and compute the distance based on camera properties.

2.3.2 MobileNet-Based Object Detection on the GAP8

This section presents two applications of MobileNet for object detection on the GAP8 to later support our decision not to use this approach. As part of a license plate reading IoT system, object detection on the GAP8 is used in [22]. They used a network architecture based on MobileNetV2-SSD and replaced the convolutional layers in the detection head with depthwise separable convolution. Though the work focuses on license plates, they achieved an mAP of 0.39 on the open images dataset containing many classes. The inference speed was 1.73 fps for detection on the GAP8.

MobileNet-SSD has also been used for human detection on the GAP8 in [24]. In their work, a Crazyflie with Flow deck and AI deck was made to follow a person with a fixed offset in translation, keeping the yaw rotation fixed. That approach achieved an inference frequency of 0.85 fps. The full implementation was evaluated with a step response that stabilized after about 10 seconds when the target took a one-meter step. A PID controller with only the P term was used for control, which resulted in much overshooting, given the low setpoint frequency. Instead of having a desired depthwise distance, they used a desired object pixel height, which worked well. In contrast, we compute an explicit distance to the target based on its size in the image.

2.3.3 Squeezed Edge YOLO

The most promising previously published model for our application is Squeezed Edge YOLO [18]. Their paper establishes that using a single-shot approach is vital on the GAP8 and presents their smaller version of YOLO-Net. They achieve a 130 ms inference latency corresponding to 7.7 frames per second and a mean average precision score of 0.95 on their dataset, training the network for only one object class at a time. For reference, YOLOv5, which can detect 80 object classes with near state-of-the-art performance, got a mAP of 0.96 on the same dataset. Figure 2.14 depicts the model they use. It takes an image input of 128x128x3, which is then passed through a network of convolutional layers interspersed with Squeeze-and-Excitation layers (SE). The convolutional layers use Leaky ReLU activation. Finally, the data is passed through two detection heads, which output the class, confidence, and normalized image coordinates for 320 possible bounding boxes.

In our thesis, we will replicate the architecture and modify it to use a different detection head, one-channel input instead of three-channel input, and a smaller input size with the same aspect ratio as our camera. We will also improve the inference time by replacing convolutional layers with depthwise separable or light convolution.

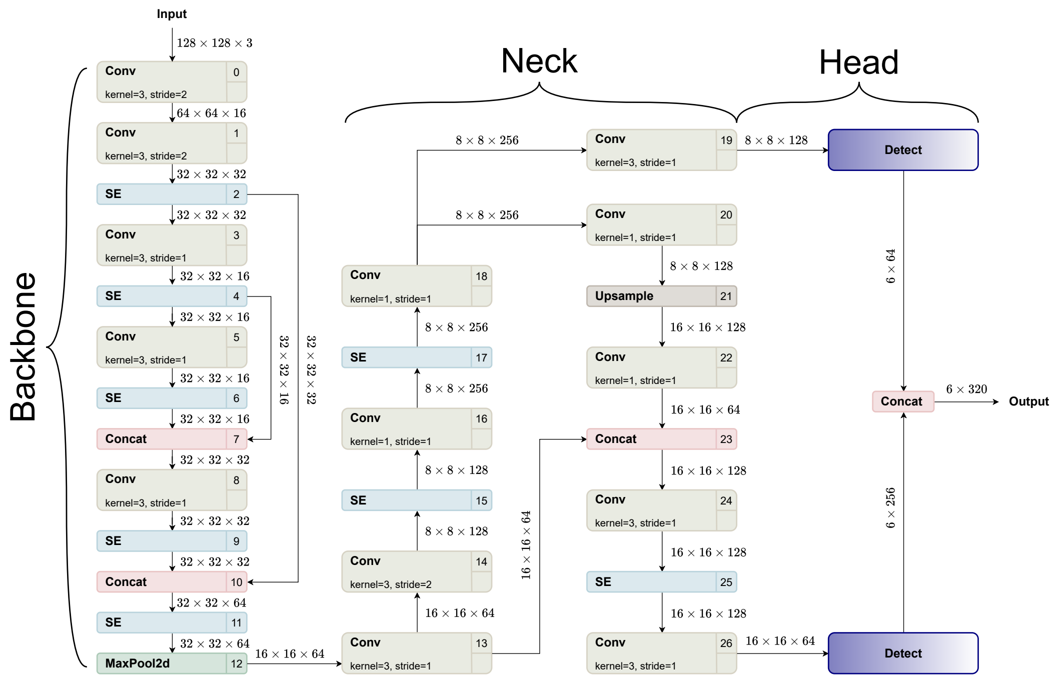


Figure 2.14: The network architecture of Squeezed Edge YOLO. The detect layers are YOLOv5 detection heads.

Chapter 3
Approach

3.1 Method

This section covers our method for implementing hand-following capabilities for the Crazyflie drone. For a quick overview, see Figure 3.1.

Our process began with a literature study to investigate different object detection frameworks. Using this in conjunction with the platform described in Section 2.2, we created a reference implementation with the model inference running on a personal computer. The reference implementation aimed to explore the Crazyflie platform and different control modes. This implementation also served as a baseline for comparing the final edge implementation. From hereon, the final implementation will be referred to as the edge implementation.

Following the reference implementation, we evaluated its performance and identified areas for improvement for the edge implementation. We noticed that the network used in the reference implementation could not be ported to the GAP8 and that the accuracy of monochrome images was subpar. Since we could not find a pretrained model for hand detection that we could run on our edge platform, we opted to train our own model.

For this purpose, we created a dataset using the images from the Himax camera. The dataset needed to be annotated, which could be sped up using a trained network. Therefore, we found preexisting datasets and trained a larger model to perform the annotations. The annotations were also manually reviewed to ensure they matched the hands in the image.

Concurrently, we selected the most promising network from related works, replicated it, and successfully deployed it to the GAP8 with the same latency.

We trained and evaluated the model, iteratively changing the network architecture to improve inference speed. Then, we selected one model and attempted to improve detection performance using transfer learning and hyperparameter tuning.

Finally, we evaluated the edge implementation, containing a trained and ported neural network and custom Crazyflie and AI deck firmware. Following the evaluation, we made a few improvements to improve flight performance.

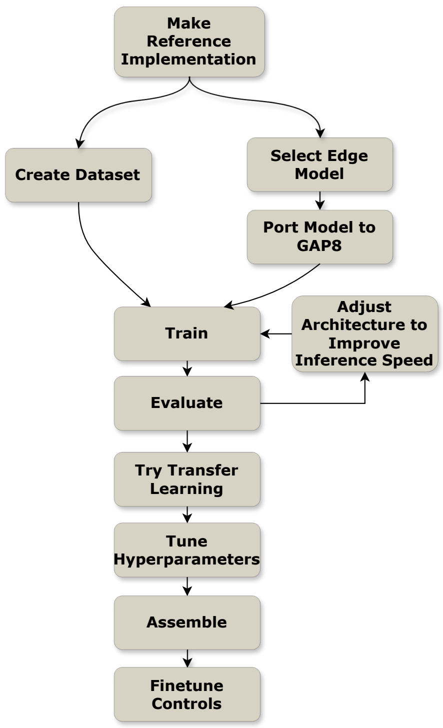


Figure 3.1: General flow of our method.

3.2 Implementation

This section will detail the steps taken to create the final implementation and motivate the choices we made. First, a reference implementation was made to investigate what performance and latency would be required by our hand detection model and how the output could be used to achieve satisfactory controls. The reference and final edge implementation are shown in figure 3.2 and 3.3. Using insights from the reference implementation, we created the dataset and selected an appropriate model architecture to train and deploy to the drone.

3.2.1 Reference Implementation

Before exploring the edge AI aspects of this thesis, we made a reference implementation where we implemented the desired flying behavior with the hand detection model running on an external computer. Though this will be detailed later, it can be seen in Figure 3.2 how the inference runs on a personal computer instead of on the drone as in the edge implementation shown in Figure 3.3. This allowed us to use a trained model early, which served three purposes.

First, it allowed us to explore the drone's ability to estimate its position and velocity and evaluate different control options.

Second, it gave us an insight into how a large machine-learning model would perform for hand detection on our images and how this would translate into actual flying ability.

Third, the reference implementation serves as a baseline that we try to beat with our edge implementation. Rather than measuring the position of the drone and hand through some external system and evaluating the drone's ability to follow a hand by, for example, a step response, we choose to compare the edge implementation to the reference implementation in terms of accuracy and latency.

MediaPipe

To detect hands, the reference implementation uses MediaPipe, a framework from Google that, among other things, has a model for hand pose estimation. The model is a single-shot detector that takes a color image as input and outputs the 3D position of 21 so-called hand landmarks at the tip and joints of each finger and one at the wrist [36]. We used this heavy model because it is already trained on hands and can handle pose estimation, which we wanted to do initially.

Overview

An overview of the reference implementation is shown in Figure 3.2. First, a grayscale image is captured by the Himax Camera on the AI deck and sent to the personal computer over a WiFi/TCP stream. On the personal computer, the grayscale image is converted to RGB by stacking it three times before passing it into the MediaPipe model, which takes color images. The cflib Python library is used to send the necessary parts of the MediaPipe output to the STM32 over CrazyRadio. On the STM32, our control loop runs, using the values sent from Python, to compute the final position that the built-in stabilizer loop is trying to achieve. If a hand was detected in the last image, two LED lights are set to flicker on the drone.

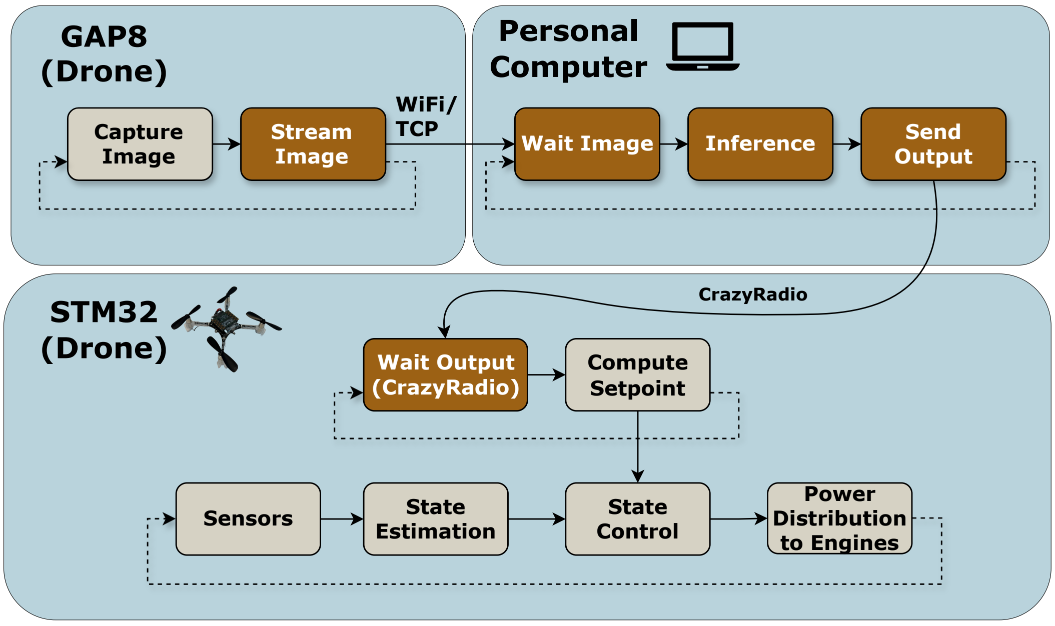


Figure 3.2: Overview of the reference implementation. The dark boxes are different from the edge implementation.

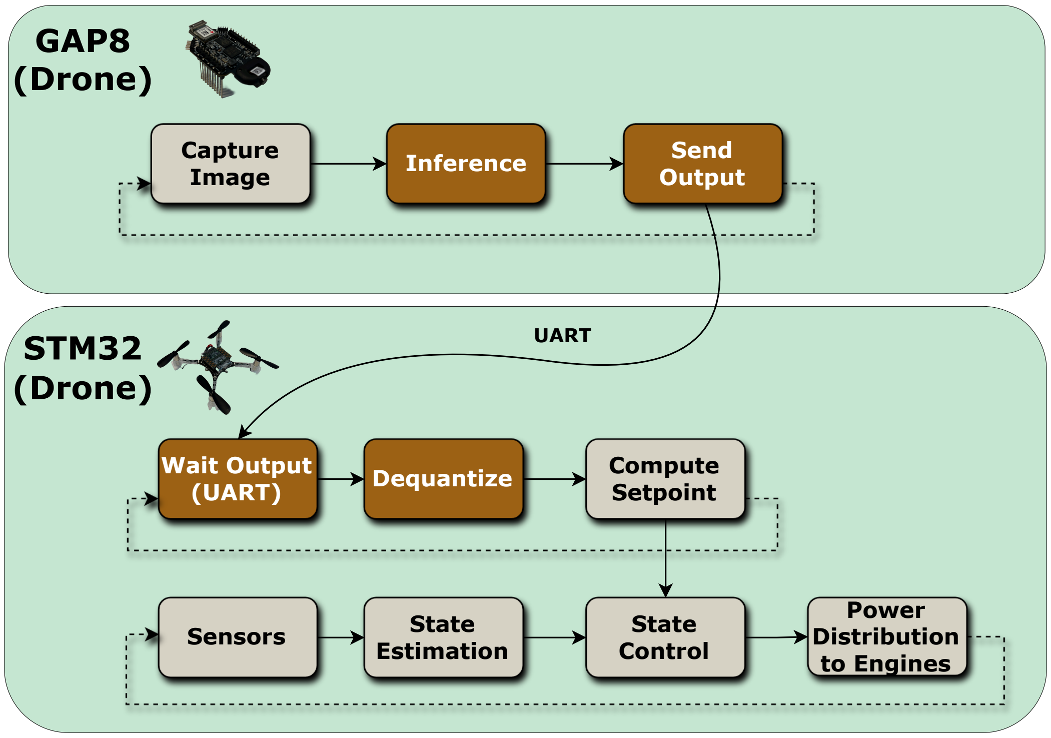


Figure 3.3: Overview of the edge implementation. The dark boxes are different from the reference implementation. Note that the inference is running onboard the drone, and the personal computer is cut out of the control loop.

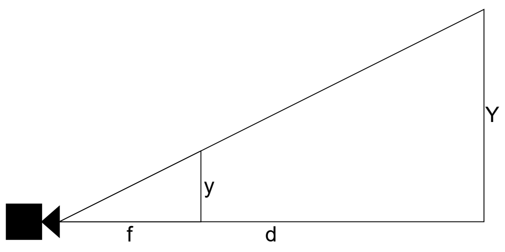


Figure 3.4: An object of real size Y and size y in the camera plane, at distance d from a pinhole camera with focal length f. The camera plane in front of the pinhole for viewability.

Concurrency in the Reference Implementation

As shown in Figure 3.2, the GAP8, personal computer, and STM32 can work partly in parallel. Later steps in the pipeline wait for earlier steps, but processing of the next image can start before the last one is finished. For example, a new image can be captured while inference runs for the last image. This does not decrease the latency of processing one image, but it does increase throughput as several images can be worked on simultaneously.

Difference Between Reference and Edge Implementation

In the edge implementation, the personal computer is cut out from the control pipeline by moving the CNN inference to the GAP8 on the drone and sending the output directly to the STM32 over UART. So, in the edge implementation, there is no communication with a personal computer for the control loop, as shown in Figure 3.3.

Position Control

In both the reference and edge implementation, a position setpoint is computed based on the position and size of the hand in the image.

First, the distance to the hand along the drone's camera axis is estimated with an inversely proportional relationship to the size of the hand in the image; the smaller the hand is in the image, the further away it is estimated to be. In the edge implementation, the hand size is the diagonal of the object detection box. In the reference implementation, it is the largest distance between the points detected in the palm. From figure 3.4, it can be derived with similar triangles that the distance d is inversely proportional to the size of the object y on screen, as in Equation 3.1. We experimentally determined the constant corresponding to $f \cdot Y$, adjusted for the fact that d is in meters and y is in normalized screen coordinates.

$$d = \frac{f \cdot Y}{y} \tag{3.1}$$

With the distance estimated, the setpoint is computed. The detected screen coordinates in the range [0,1] where (0,0) would be the top left corner are converted to centered screen coordinates where (-0.5, -0.5) is the top left corner and (0,0) is the screen center:

$$\begin{aligned} x_{centered} &= x_{screen} - 0.5 \\ y_{cetnered} &= y_{screen} - 0.5 \end{aligned}$$

The Crazyflie has a local coordinate system in meters, with the origin at its center and x, y, and z corresponding to forward, left, and up in that order. The hand coordinates in the drone's local coordinates system are computed as below. d is the distance along the camera axis, and w and h are the width and height of the field of view at a one-meter distance.

$$\begin{aligned} x_{local} &= d \\ y_{local} &= -x_{centered} \times w \times d \\ z_{local} &= -y_{centered} \times h \times d \end{aligned}$$

The Crazyflie also has a global coordinate system with the origin at the ground. It is in this coordinate system we perform the final position control. The hand's world coordinates are computed by adding them to the drone world coordinates, denoted with a superscript D, assuming the drone never rotates:

$$\begin{aligned} x_{world} &= x_{local} + x^{D}_{world} \\ y_{world} &= y_{local} + y^{D}_{world} \\ z_{world} &= z_{local} + z^{D}_{world} \end{aligned}$$

Finally, the desired setpoint position is at an offset in front of the hand, as computed below. The superscripts S and O denote setpoint and offset.

$$\begin{aligned} x^{S}_{world} &= x_{world} + x^{O}_{world} \\ y^{S}_{world} &= y_{world} \\ z^{S}_{world} &= z_{world} \end{aligned}$$

Velocity Versus Position Control

As mentioned in the section 2.2.5, the desired state of the drone, called setpoint, can be set as either an absolute position or a velocity. This section motivates our choice to compute a desired position from each image, according to the previous section. Before knowing how well the hand detection would perform, we predicted that velocity control would be easier to implement because it should be easier to know only in which direction the drone must move than also knowing how much it must move. However, in the reference implementation, the setpoint update frequency was limited. Even without any model inference, a new setpoint could only be produced at 6.1 Hz. This was the update frequency when only capturing an image, streaming it to the personal computer, and sending dummy values back without inferencing any machine learning model.

This low update frequency made it hard to control with velocity efficiently. We tried a proportional velocity controller where the velocity towards the target position is proportional to the distance from it. Then, a large coefficient caused the drone to easily overshoot its target before the next input from the camera. And with a lower coefficient, the position converged too slowly.

Because images could be processed so rarely in the reference implementation, we chose to control position instead of velocity. The same goes for the edge implementation, where the throughput is not much higher. By controlling position instead of velocity, we lose some freedom, letting the built-in position controller decide how fast the desired position is approached. But the flying behavior was satisfactory.

Reference Implementation Issues

In the reference implementation, the drone can follow slow hand movements, staying at a fixed offset position from the hand. However, the overall performance is impaired by the latency introduced by streaming the images over WiFi. The hand is often lost because MediaPipe's detection model is not made for monochrome images, and the position controller is suboptimal. In the edge implementation, we aim to address these issues by running the machine learning algorithm onboard the drone to reduce latency and train a model on grayscale images captured by the drone to increase accuracy.

3.2.2 Ultralytics

In the edge implementation, we want a model running onboard the drone that performs well on monochrome images. As will be motivated more detailed in section 3.2.4, MediaPipe could not run onboard the AI deck. Since the MediaPipe framework is generally closed, this issue could not be mitigated [36]. Therefore, we opted for another open-source computer vision and machine learning framework, Ultralytics.

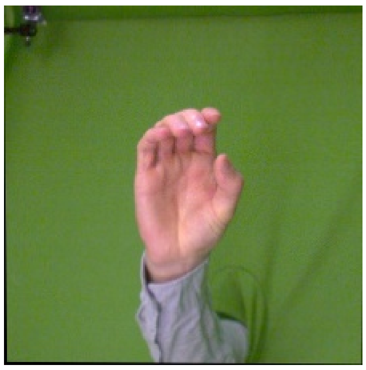
Ultralytics is a computer vision and machine learning framework built on PyTorch. The framework offers several models suitable for different computer vision tasks. The most suitable model for detecting hands is You Only Look Once (YOLO), an object detection model designed for multi-class object detection [20]. The benefits of using this framework are that the code is open-source, it has preexisting training and hyperparameter tuning pipelines, and it has support for building our own custom object detection network architectures.

Dataset Format

YOLO uses a specific dataset format for object detection. The different datasets must be in a directory with images and annotations. The matching image and annotation must have the same name. The annotations are given in a text file. Each text file shall contain one object per row in `class x_center y_center width height` format for object detection. Background images, in other words, images without specified objects, can either have an accompanying empty text file or no accompanying annotation file [35].

3.2.3 Dataset Creation

One large issue with the reference implementation was the model's poor detection accuracy for grayscale images or, rather, platform-specific images. To address this issue, we created a new dataset with images captured from the drone's Himax camera. To annotate the images from the camera, we trained an annotation model on open-source datasets. This section will detail the datasets we used and created during this thesis.


(a) FreiHAND

(b) SUN2012

Figure 3.5: Example images from FreiHAND and SUN2012.

FreiSUN Dataset

To create our own custom dataset, we first needed an efficient way to annotate it. We began by training a large annotation model using two open-source datasets: FreiHAND and SUN2012. The combined dataset will be referred to as the FreiSUN dataset. Additionally, the dataset will later be used to pretrain our final edge model in Section 3.2.7.

The FreiHAND dataset was selected since it contains images with one hand in each frame, similar to the images we will have in our dataset. Overall, the FreiHAND dataset contains 32,560 224x224 RGB images of hands, annotated with hand landmarks or keypoints in x, y, and z coordinates. An example image is shown in Figure 3.5a. The images each contain one hand in diverse positions on a green screen background. The dataset is extended by duplicating the images four times using different backgrounds [40]. One problem with the FreiSUN dataset is that it does not contain background images. Background images are important since they can decrease the number of false positive detections [35]. Because of this issue, we sourced another dataset, SUN2012. The SUN2012 dataset contains 16,837 RGB images of different resolutions with scenes of common objects [38]. An example image is shown in Figure 3.5b. The FreiHAND and SUN2012 datasets contain RGB images and have a different annotation format than the Ultralytics framework expects. Therefore, we needed to convert the annotations, convert the images to monochrome, and concatenate the datasets.

We began by preparing FreiHAND. By multiplying the coordinates by the camera intrinsic matrix, they were transformed into image space [40]. We processed the image space coordinates to align with the YOLO object detection annotation format described in Section 3.2.2. As for the bounding box, it was created by making a square that encompasses all the keypoints. Since the images are duplicated on four different backgrounds, we also duplicated the labels four times. Finally, we converted the images into grayscale and randomly split the dataset into training and validation sets with a 90-10 split ratio. This split was chosen because we will not evaluate models on this dataset, so there is no need for a test set. However, having a smaller validation set allows us to monitor if the training converges for the annotation model.

After this, we prepared the SUN2012 dataset. We removed images containing the labels **Person** and **People** to ensure the dataset contains no hands. We resized all images to 224x224 while maintaining the aspect ratio. We followed this with a conversion to grayscale. After removal, the remaining number of images was 13,386. This means the ratio of background images to annotated images is approximately 10%, which is in the recommended range given by Ultralytics [35]. Like the FreiHAND dataset, we randomly split into training and validation sets with the same split ratio. Finally, we concatenated the processed FreiHAND and SUN2012 datasets to create the FreiSUN dataset. The number of annotated images and background images for the training and validation set is displayed in Table 3.1.

Table 3.1: FreiSUN dataset specifications.

Images	**Train**	**Validation**
Annotated	117,216	13,024
Background	12,047	1,339
Total	129,263	14,363

FreiSUN Model

As stated, the FreiSUN dataset was prepared to ease image annotations for our Himax images. For this purpose, we trained the largest YOLOv8 model on the FreiSUN dataset, which will be referred to as the FreiSUN model.

Himax Dataset

We assumed that training the edge network on the images captured from the Himax camera would be important to increase detection accuracy. This hypothesis is later corroborated in Section 4.3.2, where the network trained on the Himax dataset severely outperforms the network trained on the FreiSUN dataset. Therefore, we created a custom Himax dataset using the FreiSUN model to annotate our data.

The Ultralytics framework recommends using over 1,500 images per class with 10,000 instances of labeled objects per class [35]. To achieve this, we decided to record 31 diverse scenes at five frames per second for 100 seconds, giving us approximately 15,500 images. We captured more images than the recommended amount to provide a buffer. This is because some images might be subpar and unsuitable for the dataset, and we might end up with more background images than necessary. The scenes were recorded at three different locations:

- Combine Office (com)
- E-building LUCAS (lucas)
- E-building foyer (efo)

Figure 3.6 presents examples from each location. In total, the dataset comprises 17,598 images of two persons. Background images were also generated at each location, mostly at the beginning and end of the clips. See locational distribution before review in Figure 3.7a.

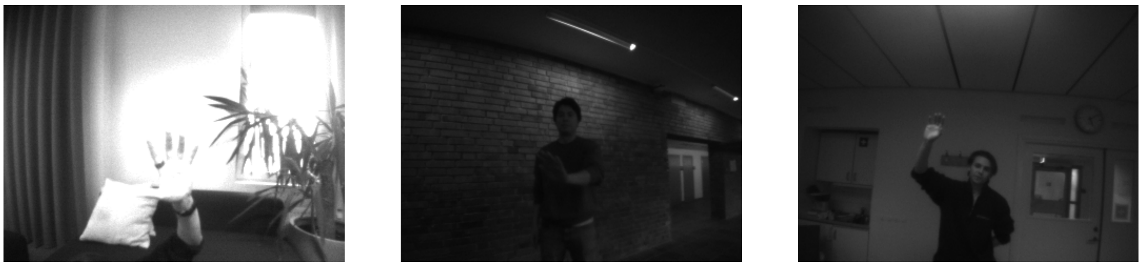

(a) Combine (com-south-3) **(b)** E-building foyer (efo-1) **(c)** LUCAS (lucas-12)

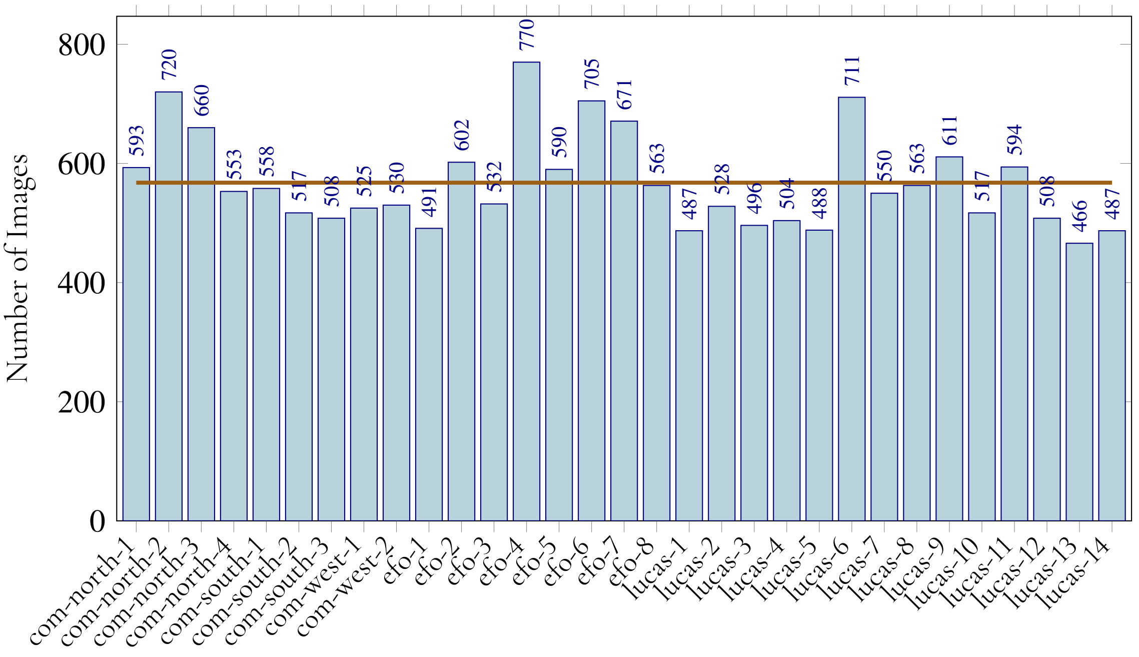
Figure 3.6: Example images from the Combine Office, E-building foyer, and LUCAS.

The Himax dataset was annotated using the FreiSUN model. The review process involved the following steps:

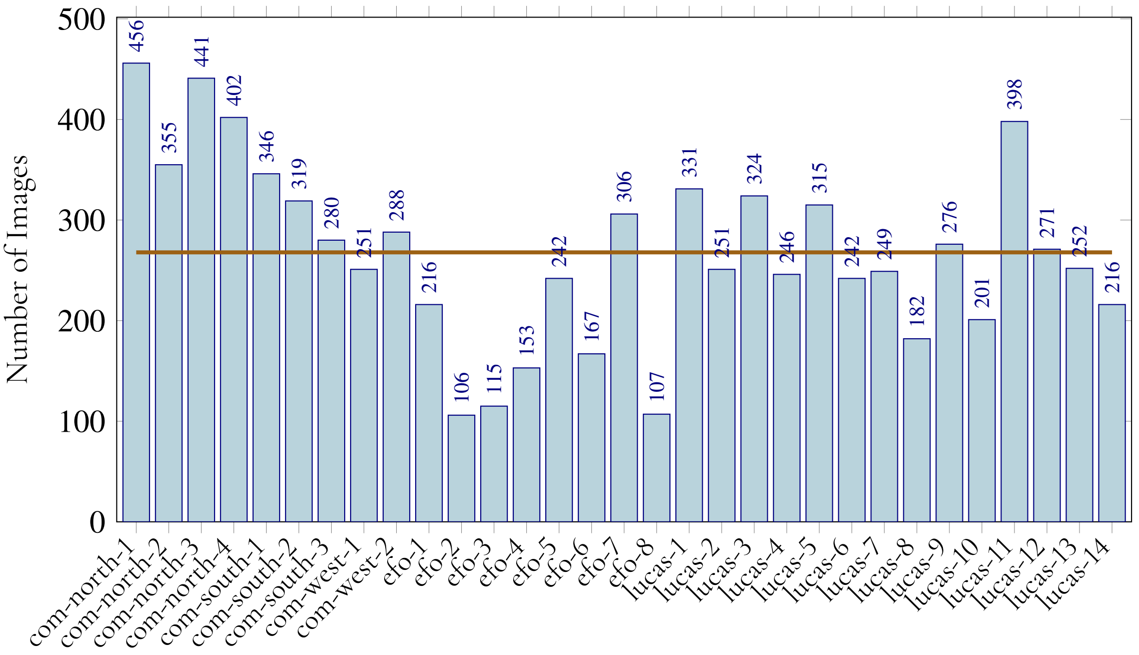
1. Annotations were performed using the FreiSUN model.
2. All predictions were manually reviewed as either pass or fail.
3. Background images were reviewed until a ratio of 90% object images to 10% background images was achieved.
4. The FreiSUN model was fine-tuned on the images that passed the review.
5. The process was repeated until we had a full annotated dataset.

This approach was motivated by the desire to speed up the image annotation process. We noticed that some scenes had better predictions using the original FreiSUN model, especially the images from the Combine office, so we began by annotating these scenes. Below, we will describe each step in the annotation process and motivate our stopping criteria for annotations. Table 3.2 displays the total number of correctly annotated images after each iteration.

1. The scenes from the Combine office were first annotated using the FreiSUN model. The annotations were manually reviewed, and the FreiSUN model was fine-tuned using the 673 images and annotations that passed review.
2. All the scenes from the Himax dataset were annotated using the fine-tuned FreiSUN model. The annotations were manually reviewed, and the FreiSUN model was fine-tuned again using the 3,076 annotations that passed review.
3. As we aimed for a dataset containing 10,000 images, we repeated step 2 twice. After these iterations, we had 8,034 annotations that had passed review. During the final review, we noticed most of the remaining images were background images, and since the number of images was close to the target of 10,000, we ended the process here. The locational distribution after review is shown in Figure 3.7b.



(a) Number of images per scene before review.



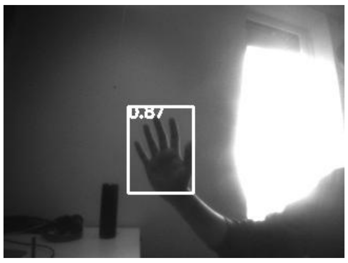
(b) Number of images per scene after review.

Figure 3.7: Comparison of the number of images per scene before and after review. The brown line indicates the average number of images per scene.

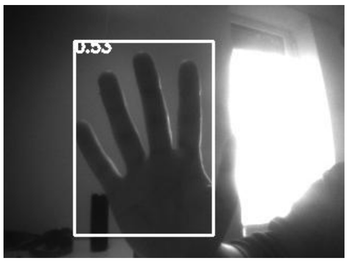
Table 3.2: Himax dataset, number of correct annotations after each iteration.

Scene	Iteration 1	Iteration 2	Iteration 3	Iteration 4
com-north-1	128	352	404	456
com-north-2	36	171	231	355
com-north-3	122	301	360	441
com-north-4	87	244	271	402
com-south-1	53	206	241	346
com-south-2	94	217	299	319
com-south-3	41	108	212	280
com-west-1	72	126	183	251
com-west-2	40	140	229	288
efo-1	N/A	36	84	216
efo-2	N/A	4	38	106
efo-3	N/A	2	30	115
efo-4	N/A	14	52	153
efo-5	N/A	48	124	242
efo-6	N/A	27	101	167
efo-7	N/A	4	67	306
efo-8	N/A	8	34	107
lucas-1	N/A	56	227	331
lucas-2	N/A	53	195	251
lucas-3	N/A	108	233	324
lucas-4	N/A	72	170	246
lucas-5	N/A	73	210	315
lucas-6	N/A	80	169	242
lucas-7	N/A	45	138	249
lucas-8	N/A	39	127	182
lucas-9	N/A	73	195	276
lucas-10	N/A	64	154	201
lucas-11	N/A	113	284	398
lucas-12	N/A	111	235	271
lucas-13	N/A	84	174	252
lucas-14	N/A	97	168	216
Total	673	3076	5639	8034

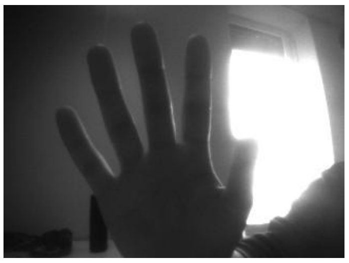
After manually reviewing the performance on new data, we noticed that the model had trouble picking up hands close to the camera, see figure 3.8. We expected this to cause problems in the edge implementation because the drone can never increase its distance from the hand if it can not detect that it is too close. Therefore, we created several more scenes with hands closer to the camera. We began by preparing eight scenes at the Combine office, which we expected to be sufficient to improve performance. The new scenes contain images of hands much closer to the camera. These scenes were then annotated using the fine-tuned FreiSUN model. Only the annotations with the highest confidence score were saved if a hand or several were detected in the image. Since this dataset was smaller, all images were manually reviewed, including reshaping bounding boxes. Finally, background images were removed to adhere to the 90-10 ratio. Six scenes were added to the training data, one to the validation data, and one to the test data. The detection performance was manually reviewed and deemed sufficient.



(a) Hand detected at further distance.

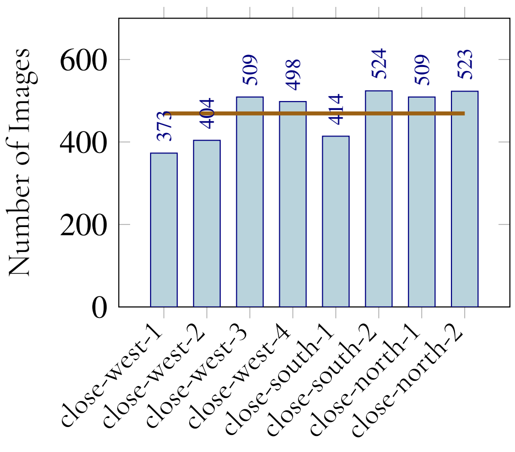


(b) Hand detected at medium range.

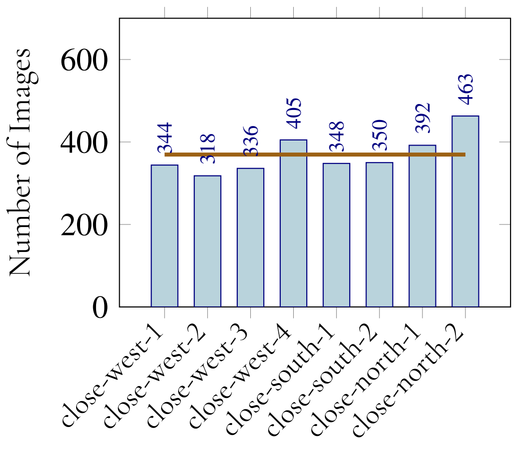


(c) Hand not detected at close range.

Figure 3.8: After training on the initial Himax Dataset, hands close to the camera were rarely detected.



(a) Number of images per scene before review.



(b) Number of images per scene after review.

Figure 3.9: Comparison of the number of images per scene before and after review. The brown line indicates the average number of images per scene.

This concludes the dataset creation. The final dataset contains 9090 images. See locational distribution per scene in Figures 3.7b and 3.9b. The dataset was split into nearly 80-10-10 splits, meaning 80% for training, 10% for validation, and 10% for testing, see Table

3.3. To avoid data leakage, the dataset was split by scenes; the scenes in each set were selected to achieve close to 80-10-10. The reasoning for choosing an 80-10-10 split is that we will use the validation performance to tune hyperparameters and then use the test set to evaluate the model's detection accuracy.

Table 3.3: Himax dataset specifications.

Images	Train	Validation	Test
Annotated	8502	1008	999
Background	588	86	77
Total	9090	1094	1076

3.2.4 Model Selection

Apart from having good detection accuracy on monochrome images, the model in our edge implementation needed to fit in the drone's memory and run with acceptable latency. We saw early that Mediapipe, which we used in the reference implementation, could not be deployed. Therefore, we ended up using a modified version of Squeezed Edge YOLO, which we previously mentioned in related work. This section will motivate our choice of this model for hand detection.

Initially, we tried porting MediaPipe to the GAP8. However, this was unsuccessful since the PReLU operation is not supported by GAPFlow, and presumably, the network would not run fast on the AI deck because of its size and use of floating point operations. We continued by trying the smallest YOLOv8 model, which also failed to port due to too large intermediate outputs. After investigating several different possible solutions and familiarizing ourselves with GAPFlow, we realized small changes like changing the stride number could make the model import fail. Therefore, we opted for a solution we knew would work.

By examining what similar models have been deployed on the GAP8, as presented in related works, PULP-Frontnet [29] had an impressive inference rate of 48 Hz, even in its largest version. However, we thought potential problems would be easier to troubleshoot if we used a more established architecture for object detection specifically. Squeezed Edge YOLO [18] and MobileNet-SSD used in [22] are [24] such networks. Though the works adapting them do not present the same performance metrics, Squeezed Edge YOLO performed almost as well as YOLOv5 when detecting one object class at a time on the AI deck and had a better inference rate of 7.7 Hz compared to 0.85 and 1.73 reported for the MobileNet-SSD. In the end, we chose to go with Squeezed Edge YOLO.

3.2.5 Squeezed Edge YOLO

We tried to recreate the work in the Squeezed Edge YOLO paper [18] by using the model configuration functionality in the Ultralytics framework, training the model, and exporting it as ONNX, a file format for representing neural networks, importing it into NNTool, where the network was adjusted, fused, and quantized. Inference C code was generated with the Autotiler, and the code was compiled and flashed into the AI deck's persistent memory.

Activation Function

We made changes in the Ultralytics library to use Leaky ReLU activation after convolutional layers, as in Squeezed Edge YOLO. By default, Ultralyics uses SiLU. However, GAPFlow has special support for Leaky ReLU, so using it improved the inference speed by 8 ms.

Ultralytics and GAPFlow

Ultralytics is built on PyTorch and can export both ONNX and TFLite files. However, since NNTool does not support all operations these file formats do, tinkering was required to make Ultralytics export a DNN file that NNTool would accept. The Ultralytics TFLite export works by first exporting an ONNX and then converting it to TFLite. Exporting a TFLite with an option for ONNX version 13 and using the intermediate ONNX forced the use of only operations supported by NNTool.

Monochrome

Because we used the monochrome camera on the AI deck instead of the RGB version used in the Squeezed Edge YOLO paper, we modified the network to take monochrome images. This required making some changes in the Ultralytics library, where it was hard-coded that the network should take three-channel images. Though this did not affect the inference time much since the first layer accounts for little of the total operations, some memory was saved on the GAP8.

Input Shape

Because our AI deck camera outputs 324x244 QVGA or 162x122 QQVGA images, we changed the network input size from 128x128 to 128x96, which almost corresponds to the same aspect ratio as QVGA and QQVGA. This was done to save memory and operations. A resize operation was then manually prepended to the ONNX network to resize the QVGA input to 128x96 using bilinear interpolation.

Detection Head

The last part of a YOLO-based object detection model is called the detection head. Here, the salient features enhanced by earlier layers are used for regression of the detection boxes and class confidences. The original squeezed Edge YOLO paper does not specify which detection head was used. However, after contacting the authors, we discovered they used the YOLOv5 head. Ultimately, we opted for the YOLOv8 head since it is fully supported in the Ultralytics library that we had committed to. We could easily achieve model export but not training with the v5 head because it uses a different output than the v8 head. Exporting an untrained model with a v5 head was useful to verify that we got the same latency and memory usage as in the squeezed Edge YOLO paper. After doing this, we moved on with the v8 head.

The YOLOv8 detection head is a decoupled head in which the regression of box position and sizes, as well as class confidences, uses two different sets of convolutional kernels. This differs from the coupled detection heads in YOLOv1 to YOLOv5, where the same weights are used for all outputs.

In YOLOv8, regression of both boxes and class confidence each use two convolutional layers with 3x3 kernels and one with 1x1 kernels, compared to the YOLOv5 head, which only uses one convolutional layer with 1x1 kernels shared by all outputs. When profiling them, we found that the v8 head took 59 ms and the v5 head only 1 ms.

Studies have shown that decoupled heads allow for better performance and faster training convergence because of the numerical difference between box regression and class confidence regression [25]. However, we do not compare accuracy between the heads.

Verification

After attempting to copy the Squeezed Edge YOLO network using the YOLOv5 head as in the paper but with changed input size and number of input channels, we achieved the same 130 ms latency as in their paper [18]. The measurements were done with the GAP8's dynamic clock frequencies set to the highest 250 MHz for the fabric controller and 175 MHz for the cluster cores. The VCD trace in Figure 3.10 shows an inference without the head and resembles the one in their report [18], which makes us confident the replication was successful. The VCD trace is without a head simply because it is an image we saved during early testing, but note that adding the YOLOv5 head should only add 1 ms. Oddly, reducing the input size from 128x128 to 128x96 did not decrease the inference speed compared to their network. With the YOLOv8 head that we could train, the inference latency was 189 ms at this stage.

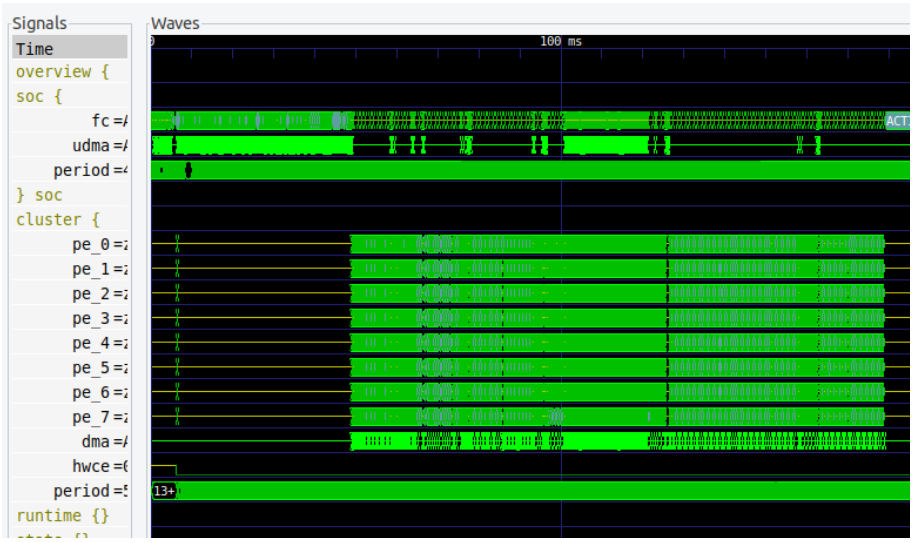


Figure 3.10: VCD trace of our copy of Squeezed Edge YOLO without the head.

Quantization

Quantization is an essential part of porting DNNs to an edge device. As described in Section 2.2, we used the default, symmetrical, uniform, 8-bit quantization scheme supported in NNTool for this. Statistics regarding output and intermediate values were collected from inferences on 300 images from the validation set. By quantizing the original 32-bit floating point model into 8-bit integers, we reduce the model size, making deployment possible and minimizing memory accesses during inference. It also enables the use of the 4x 8-bit SIMD MAC instructions on the GAP8.

Quantized Network Input

In its quantized version, the network takes an input tensor where each element is an 8-bit signed integer in the range [-128,127], but we wanted it to take unsigned integers in the range [0,255] outputted by the camera. For this, we used NNTool to add a 1-bit right shift operation to the unsigned [0,255] input. The new leftmost bit becomes 0, and the least significant rightmost bit is dropped. The original seven most significant bits are mapped to the range [0,127] in the resulting signed integer that could represent the range [-128,127] if the first bit was not fixed to 0. Because one bit is dropped, we lose one bit of precision from the input with this approach. For example, both 254 and 255 would become 127. A slower but more precise method of mapping [0,255] integers to [-128:127] is to subtract 128. However, we encountered some errors when using this NNTool.

Handling Network Output

With only one class, 128x96 input size, and batch size ignored, YOLOv8 outputs a tensor of size 5x240 corresponding to x, y, width, height, and confidence for 240 cells. Normally, these are floating point values in the range [0,1]. But in the quantized version, they are integers in [0,127]. Using non-negative inputs and a symmetric quantizer, we only use half the [-128,127] signed int8 output range. But this did not hurt the precision significantly, as presented in the Results under section 4.3.4.

From the GAP8 fabric controller, after letting the C code generated by the Autotiler execute the inference on the eight cluster cores, we pick the output with the largest confidence and send the corresponding values to the STM32 over UART. On the STM32, supporting floating point values, we dequantize the received box by multiplying the quantized x, y, width, height, and confidence with the quantization scalar for the output tensor.

3.2.6 Lightweight Models

After successfully reimplementing Squeezed Edge YOLO with the YOLOv8 detection head, we wanted to improve the 189 ms inference latency to control the drone more rapidly. Note that the inference only constitutes part of the total time between setpoints. Capturing an image, sending the CNN output between the GAP8 and the STM32, and computing control signals consumed about 60 ms at this stage. In this section, we will detail the steps we took to create three optimized versions of Squeezed Edge YOLO. The four lightweight models will later be compared in the evaluation section.

Profiling

For profiling the inference speed, the Autotiler can add instructions to the inference C code, measuring elapsed clock cycles per layer and for the entire network. Dividing by the clock frequency yields latency. Additionally, the number of MACs per layer can be computed during tiling. By running an inference in GreenWaves clock cycle-accurate processor simulator

GVSOC, generating a VCD trace, and viewing it in GTKWave, utilization of each core and memory interface can be viewed per clock cycle, which helps identify poor memory usage and parallelization.

Paralellization and Memory Efficiency

A VCD trace, including an inference before optimizations, is shown in figure 3.11. Almost full activity in the lines *pe_0* through *pe_7* during roughly 183 ms, same as the inference time, shows that the Autotiler successfully parallelized the inference over the eight cluster cores. The *udma* row indicates that the L3 memory was, for the most part, not used during inference. However, because only the L3 memory is persistent, it was used before the first inference to copy the network weights into L2. L3 was also used extensively, roughly from millisecond 125 to 145 of the simulation and a few times more. Mitigating this was desirable because accessing the L3 memory is expensive.

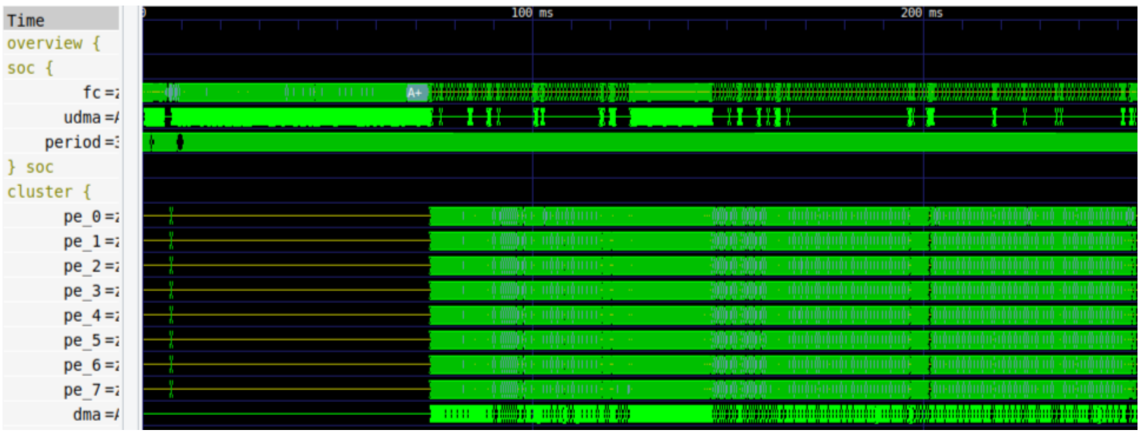


Figure 3.11: VCD trace of our network before optimization.

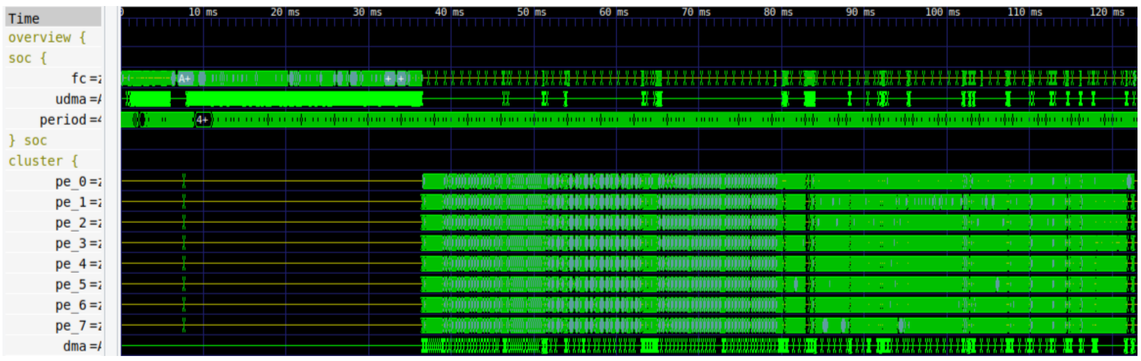
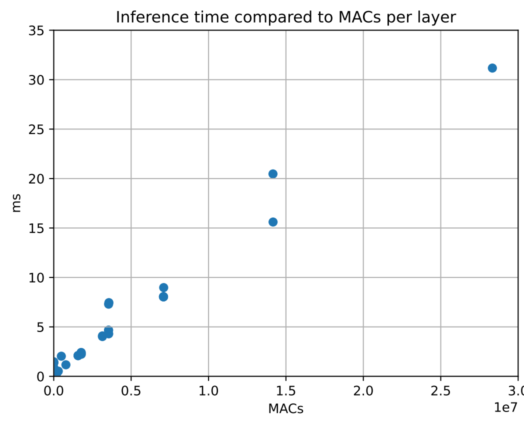


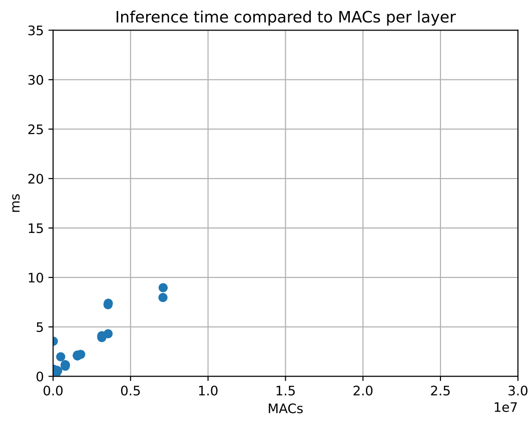
Figure 3.12: VCD trace of our network after optimization

Light Convolution in Network Neck

After ruling out problems with parallelization and memory accesses, we concluded that the long inference time could be attributed to the number of MACs in our network, which could



(a) Before Optimization.



(b) After Optimization.

Figure 3.13: Inference time compared to MACs per layer. The left plot shows our replication of Squeezed Edge YOLO with a YOLOv8 head. In the right plot, convolution has been replaced with light convolution in the three most time-consuming layers and the detection head.

be mitigated by replacing the most computationally heavy convolutional layers with light convolution. By plotting the measured inference time to the number of MACs per layer, see figure 3.13a, it became clear that inference time increased almost proportionally to the number of MACs. So, to improve the inference time, the number of MACs had to be reduced. As seen from the figure, three of the layers were especially computationally heavy. These were convolutional layers with kernel size 3x3 that resulted in many MACs because of their input size, input depth, and number of kernels. Therefore, we replaced them with what we call light convolution, described in section 2.1.5. Unfortunately, the Autotiler could not generate C code for light convolutional layers with Leaky ReLU. Therefore, we had to use normal ReLU on these three layers. Even though normal ReLU performs worse than leaky ReLU in some situations, as described in 2.1.4, changing the activation function had no significant performance cost. This will be seen in section 4.3.1 of the Results chapter, where the optimized model performs similarly to the original reimplementation of Squeezed Edge YOLO.

Using YOLOv8 Head With Light Convolution

The original Squeezed Edge YOLO uses the YOLOv5 detection head, but we used the standard head for YOLOv8. It has four convolutional layers with 3x3 kernels and two with 1x1 kernels, which is more time-consuming than the YOLOv5 head, which uses only one convolutional layer with 1x1 kernels. Our decoupled YOLOv8 head took 59 ms before optimization, while trying the coupled YOLOv5 head took only 1 ms. Therefore, we assumed the YOLOv8 head had redundancy in its ability to capture complex patterns. So, after replacing the 3x3 convolutional kernels in the YOLOv8 head with light convolution, it took 15 ms. When testing at this stage, both the latency at 86 ms and the detection accuracy were sufficient for a satisfying flight experience. The exact metrics can be found in Section 4.3.1.

Effects of Light Convolution

It can be seen how the three most time-consuming layers in Figure 3.13a are heavily optimized in 3.13b after being replaced by light convolution. Before optimization in Figure 3.13a, one outlying layer took considerably longer than it should have, according to the proportional relation between macs and latency. This layer took 21 ms, which matches the period with many L3 accesses in Figure 3.11. Fortunately, light convolution solved this problem since it reduces the number of weights needed from persistent L3 memory to execute a layer. Figure 3.12 shows better memory usage after optimizations. Note the different scales on the time axis.

All Configurations

Model Architecture	Comment
SE YOLO	Replication of Squeezed Edge YOLO with YOLOv8 head instead of YOLOv5 and 1x96x128 input instead of 3x128x128.
YOLF	SE YOLO with the three heaviest convolutional layers in the neck and all applicable ones in the head replaced with light convolution.
DWSConv	SE YOLO with all applicable convolutional layers replaced with depthwise separable convolution.
LightConv	SE YOLO with all applicable convolutional layers replaced with light convolution.

Table 3.4: Deployable lightweight model architectures we evaluated.

Table 3.4 summarises the different configurations we tried, using depthwise separable and light convolution in different layers. So far, we have optimized our replication of Squeezed Edge YOLO (**SE YOLO**) and arrived at a model architecture we will hereafter refer to as **YOLF** (You Only Look Fast). Because latency could be reduced with little loss in detection accuracy by using light convolution, we also tried replacing all convolutional layers in the network body and neck with light convolution, except those with input channels or kernel size one, which would not be applicable. This decreased the detection accuracy considerably, as will be seen in the Results under section 4.3.1. This configuration will hereafter be referred to as **LightConv**. It is possible that the performance decreased because too many layers got ReLU activation instead of Leaky ReLU. To control for this, we also tried using normal depthwise separable convolution, which is supported with Leaky ReLU in GAPFlow, in all applicable layers. This configuration will be referred to as **DWSConv**. There probably exists a better subset of convolutional layers to replace with depthwise separable or light convolution. However, we did not try every combination due to a lack of time.

3.2.7 Model Training and Evaluation

After selecting four lightweight model candidates for our real-time on-edge application, we trained and evaluated each based on inference time and detection accuracy. Following this evaluation, we selected the top-performing model, to which we applied transfer learning and hyperparameter tuning.

Training

We began by training the four models on the Himax dataset. Ultralytics recommends 300 epochs for training with datasets containing over 10,000 labeled objects [35]. However, as shown in Table 3.3, our Himax dataset contains fewer than 10,000 labeled objects. Therefore, we decided to train for 500 epochs to compensate for the smaller dataset size, as fewer data points result in fewer iterations per epoch. All models were trained on the Himax dataset for 500 epochs using the default settings. To ensure that 500 epochs were sufficient for later comparison, we looked at the training plots and could see that they converged.

Among the four candidates, the YOLF model demonstrated a balanced performance, providing a good trade-off between detection accuracy and inference time. Detailed results and evaluations can be found in Section 4.3.1. We proceeded by pretraining YOLF on the FreiSUN dataset for 100 epochs using the default settings. We opted for fewer than 300 epochs because the FreiSUN dataset is significantly larger, approximately ten times the size of the Himax dataset.

Next, we fine-tuned the pretrained YOLF model on the Himax dataset for an additional 500 epochs, using the same reasoning for the number of epochs as during initial training. The results from the pretrained and non-pretrained models were compared, and the pretrained model outperformed the non-pretrained one. Consequently, the pretrained YOLF model was selected for the next step. For the results and discussion, see Sections 4.3.1 and 4.4.2.

Hyperparameter Tuning

In this step, we used ASHA as described in Section 2.1.4 to perform two rounds of hyperparameter tuning for the pretrained YOLF model on the Himax dataset. We selected the hyperparameters based on domain knowledge. Table 3.5 describes all selected parameters.

As described in Section 2.1.4, to perform the hyperparameter search, the algorithm is given the hyperparameter space, a number of configurations, and a stopping epoch. For the first round of ASHA search, we kept the search space large. The stopping epoch was set at 100, as previous training plots indicated that convergence started to occur around this point. Testing 200 configurations using ASHA took approximately 20 hours on our hardware, detailed in Section 4.2.2. Due to time constraints, we determined that 200 iterations were sufficient.

In this initial round, we conducted a broader search across all hyperparameters. However, we observed that the initial tuning results were worse than the original results obtained with the default parameters, as discussed in Section 4.3.3. Consequently, we decided to narrow the search space to focus on the default parameters, which led to a slight improvement in accuracy.

To achieve our final model, we trained the pretrained YOLF model using the tuned hyperparameters on the Himax dataset for 1000 epochs. Additionally, we opted to stop using mosaics earlier, completing the training with 100 epochs without mosaics. The rationale behind these adjustments was that we only needed to train the model one last time, allowing us to increase the number of epochs. During our investigation of various models and sev-

Training Settings	
Parameter	**Description**
Weight Decay	Regularization term to prevent overfitting.
Warmup Epochs	Number of epochs to gradually increase the learning rate.
Warmup Momentum	Momentum parameter during the warmup phase.
Box	Weight of the bounding box regression loss.
Cls	Weight of the classification loss.
Dfl	Weight of the distribution focal loss.
Data Augmentation	
Parameter	**Description**
Translate	Amount of random translation augmentation.
Scale	Amount of random scaling augmentation.
Shear	Amount of random shear augmentation.
Perspective	Amount of random perspective transformation.
Fliplr	Probability of flipping the image horizontally.
Mosaic	Probability of applying mosaic augmentation.
Mixup	Probability of applying mixup augmentation.

Table 3.5: Description of hyperparameters [20].

eral rounds of training and tuning, we observed that ending mosaics earlier could improve performance on the validation set. As shown in Section 4.3.1, these changes also improved detection accuracy on the test set.

3.2.8 Integration

In this final part, we will integrate all components and manually review the flying performance to identify and implement improvements. To summarize the preceding sections, we developed the code for flying based on hand detection and created two datasets for training: FreiSUN and Himax. We chose YOLF as our model based on the trade-off between detection accuracy and inference time and selected the optimal parameters for training. This model was then trained and fine-tuned using the FreiSUN and Himax datasets.

In this final section, we will bring everything together by loading the YOLF model and the control code onto the AI deck and drone. We will then review the flight performance and make any necessary adjustments to optimize it.

Hand Detection

As described in Section 2.1.2, the model outputs all possible bounding boxes and their respective confidence scores. To detect a single hand, we keep only the bounding box with the highest confidence score for each prediction and discard the rest. However, the hand might not be present in the image in some cases. To address this, we established a confidence threshold. If the confidence score exceeds this threshold, we conclude that a hand is detected in the image, and the drone should move to the predicted position. Through experimentation, we determined that a confidence threshold of 0.5 effectively minimizes false positives while reliably detecting the hand.

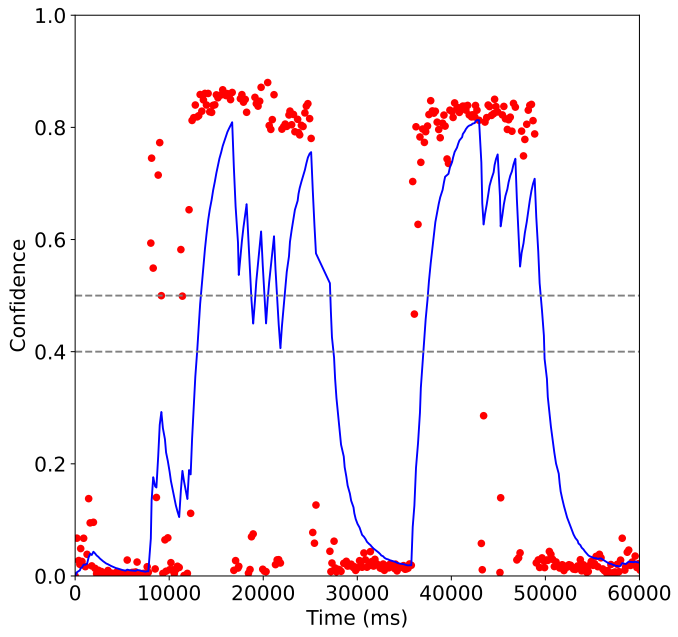


Figure 3.14: Confidence values of predictions before and after filtering.

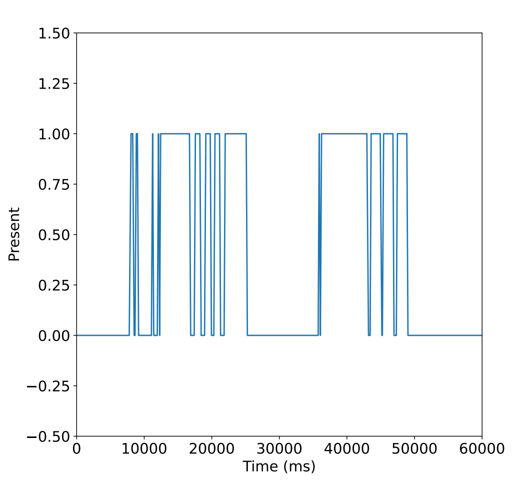
During flight tests, we observed during some light conditions that the drone frequently lost and regained sight of the hand, causing oscillations, which ultimately caused the drone to crash. This issue is illustrated in Figure 3.14, where confidence predictions are plotted as red dots. The plot shows that confidence scores above 0.5 often drop below this threshold in subsequent predictions. To address this, we applied a low-pass filter to smooth the predictions, obtaining the blue line in Figure 3.14. The low-pass filter is defined in Equation 3.2, where new_value is the new confidence value, prev_value is the previous low-pass filtered confidence, and β is the filter constant. The filter constant was set to $\beta = 0.10$.

$$\text{new\_value} = \text{new\_value} - \beta(\text{prev\_value} - \text{new\_value}) \tag{3.2}$$

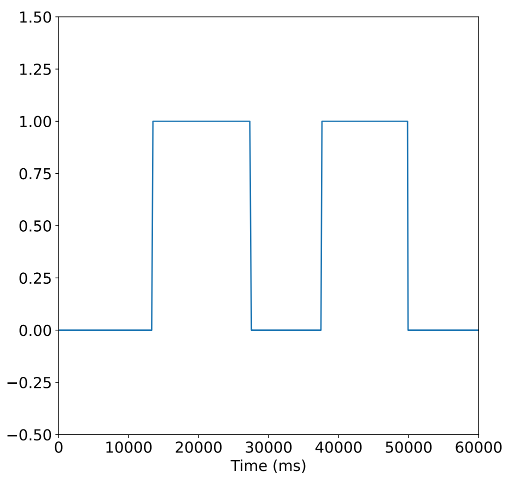
Additionally, we implemented a dynamic threshold, or deadband, to stabilize the detection. The deadband is shown as two dotted lines in Figure 3.14. If the hand was detected in the previous frame, the low-passed value must fall below 0.4 for the detection to be considered lost. Conversely, if the hand was not detected in the previous frame, the new confidence value must exceed 0.5 to confirm detection. If the new confidence value falls between 0.4 and 0.5, the detection status from the previous frame is maintained. The deadband thresholds and the filter constant were found through experimentation, as these values offered a good trade-off between initial detection speed and stability. Overall, this approach helped to prevent rapid oscillations in detection status.

Figure 3.15a shows the same timespan but highlights the binary hand presence using a confidence threshold of 0.5. It is evident that oscillations occur when the hand presence confidence rapidly drops from above 0.5 to below it. After applying the low-pass filter and the deadband, these oscillations are eliminated, as shown in Figure 3.15b.

Another adjustment we made was to increase the preferred distance between the drone and the hand. This helped with two challenges. First, if the drone is close to the hand, the



(a) Plot for hand presence before filtering.



(b) Plot for hand presence after filtering.

Figure 3.15: Plots for hand presence.

hand can easily be moved sideways outside the drone's field of view before it has time to adjust. Second, if the hand is close to the drone, it can easily be moved so close it does not fully fit in the camera. Then, the drone can not comprehend its size and estimate its distance properly. By increasing the preferred distance from 40 cm to 50 cm, we gave the drone more time to move sideways and backward before it was too late to do so at all.

Replacing QVGA With QQVGA

At this stage, the camera capturing a new picture took roughly 60 ms out of 139 ms between setpoints. By changing the camera resolution from QVGA at 324x244 to QQVGA at 162x122, we reduced this latency to 54 ms. This did not affect detection accuracy since 162x122 is still larger than the 128x96 input to the first convolutional layer.

Chapter 4
Evaluation

In this chapter, we first motivate how we evaluated the different models we considered and our final implementation. Then, the results are presented, which motivate our final choice of model and prove that we solved our real-time control problem. The results are discussed in accordance with our research questions, and some other interesting considerations are brought up in the discussion subsection.

4.1 Metrics

This section defines the metrics used to evaluate the edge model and the full implementation. First, we outline the metrics used to select the optimal model for inference onboard the drone. Second, we define the metrics for comparing the reference and edge implementation.

4.1.1 Models

To evaluate the models, we used the metrics given in Table 4.1. The first four metrics determine the model's detection accuracy, while the last three metrics indicate how challenging it is to train and run inference on different hardware. We will use detection accuracy as an all-encompassing term that refers to the model's ability to locate on hand in an image.

Motivation for Mean Average Precision and Fitness

To evaluate the different models' detection accuracy, we use two measures of mean average precision: mAP_{50} and mAP_{50-95}. Additionally, we use the fitness score provided in the Ultralytics framework. mAP_{50} and mAP_{50-95} are commonly used for evaluating object detection models and serve different purposes [2].

mAP_{50} denotes the mean average precision for a set IoU threshold of 0.5. This means that only half of the detection box has to overlap with the ground truth to count as a correct detection. So mAP_{50} captures if the model can find objects at all. mAP_{50-95} denotes the average of several mAP values computed with different IoU thresholds ranging from 0.5 to 0.95, where, in the last one, 95% of the detection box has to overlap with the ground truth for a correct detection. So mAP_{50-95} puts a higher emphasis on a correctly detected location. The IoU is never demanded to be one because an object can be interpreted as fully localized, even if the detection box deviates a few pixels from the ground truth. As stated in Section 2.1.2, since we only classify one class of objects, the mean average precision is equal to the average precision for our only class.

Additionally, the fitness score is computed as a weighted combination of mAP_{50} and mAP_{50-95}, given by Fitness = $0.1 mAP_{50} + 0.9 mAP_{50-95}$. Ultralytics provides the latest and best weights after each training session. The epoch that achieves the highest fitness score determines the best weights. We used the best weights to evaluate the test set.

Table 4.1: Metrics for evaluating the models and implementation.

Metric	Description
Precision	Measures, the correctness of positive predictions, defined in Section 2.1.2.
Recall	Measure, the model's overall ability to recognize all positive occurrences, defined in 2.1.2.
Mean Average Precision (mAP)	A measure of overall detection accuracy, using precision and recall, defined in Section 2.1.2.
Fitness	A weighted combination of detection accuracy metrics.
Number of Parameters	The total number of trainable parameters in the model.
Memory Footprint	The amount of memory required to store the model.
Latency	The time taken to process an input and produce an output.

4.1.2 Full Implementations

To evaluate full implementations, specifically the edge implementation where the inference runs onboard the drone and the reference implementation where the inference runs on a personal computer, we used the metrics listed below:

- **Setpoint Latency**: Delay between setpoints. This dictates how often the drone can adjust its position.
- **Latency of Other Activities** This was measured to evaluate what takes time except inference in the different implementations.

Motivation for Setpoint Latency

To measure latency in an entire implementation, we chose to measure the latency between when setpoints are set, which we call setpoint latency. This corresponds to from point B to point C in Figure 4.1. Note the parallelism in Figure 4.1; the GAP8 can start working on a new image while the STM32 is still processing the last. Therefore, the setpoint latency from point B to point C is not the same as the latency from when an image is captured to

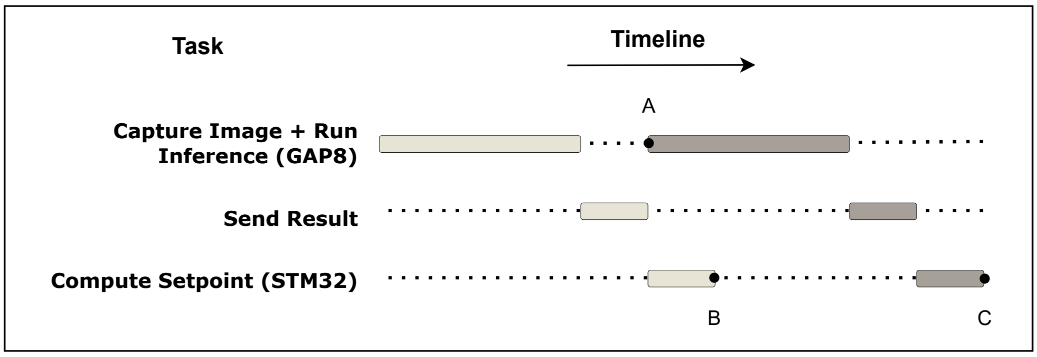


Figure 4.1: Timeline of two subsequent setpoints being computed, partly in parallel, in the edge solution. The scales are not proportionate to real measurements.

when a setpoint has been computed, corresponding to point A to point C in Figure 4.1. This latency is arguably more relevant because it measures how up-to-date the current setpoint is. However, we chose to present setpoint latency because it can easily be measured on a single processor. As will be seen, the setpoint latency reflects the speed of our entire control loop well because the majority of the delay is attributed to tasks that happen in sequence. The mean latencies for QQVGA image capture at 55.0 ms and inference at 85.63 ms performed in sequence on the GAP8 constitutes almost exactly the mean setpoint latency of 139.20 ms, as will later be seen in Tables 4.13, 4.7 and 4.12.

4.2 Experimental Setup

In this section, we will describe the experimental setup so that the results can be reproduced.

4.2.1 Ultralytics

We will use Ultralytics version 8.2.4 for all results presented in the report. We have made some changes in the repository to allow it to train and validate using monochrome images.

4.2.2 Workstation

All models were trained on the configuration in Table 4.2, which will also be employed to compute inference time and detection accuracy. In the text, we will refer to this computer as the workstation.

4.3 Results

In this section, we present all the findings that helped us determine the best lightweight model and evaluate the edge implementation. The evaluation is divided into five parts. First, we present metrics of the different lightweight models. Based on detection accuracy and inference time on the drone, we selected the best model. Then, The accuracy of the selected

Table 4.2: Workstation specifications.

Component	Specification
OS	Ubuntu 22.04
GPU	GIGABYTE GeForce RTX 3090 GAMING OC 24GB
CPU	AMD Ryzen 9 7900X3D 4.4 GHz 140MB
RAM	Kingston 32GB (2x16GB) DDR5 6000MHz CL36 FURY BE
Storage	Samsung 990 PRO M.2 NVMe 2TB

Table 4.3: Configurations for lightweight models.

Model	Parameters	Operations	L1	L2	L3 Ram	L3 Flash
	[10^3]	[MMAC]	[kB]	[kB]	[kB]	[kB]
SE YOLO	1,304	135.9	46.7	260	1,324	1,324
YOLF	526	51.9	46.7	260	507	550
DWSConv	414	29.2	46.7	260	375	442
LightConv	412	31.5	46.7	260	372	438

model was improved using transfer learning and hyperparameter tuning, the results of which are presented. Finally, the quantization results will be presented, and the final edge implementation will be evaluated.

4.3.1 Model Selection

In this part, we will train and test the four lightweight models summarized at the end of Section 3.2.6. The four models and their respective number of parameters, operations, and memory usage are displayed in Table 4.3. The initial SE YOLO model has the highest number of parameters, operations, and memory usage. The three additional models were created under the hypothesis that reducing the number of MACs would improve inference time on the AI deck. To investigate this claim, we will train the models on the Himax dataset for 500 epochs using the default Ultralytics parameters for training as explained in Section 3.2.7. The best model will be selected based on detection accuracy and inference time on the GAP8 processor.

We begin the model selection by presenting the validation and test results for all the lightweight models. Table 4.4 presents the training results for all the models in terms of precision, recall, mAP, and fitness. To construct the table, the epoch with the best fitness score was chosen from the training results, as this is the model we will use to perform predictions on the test set. In Table 4.4, it is evident that SE YOLO scores the highest in detection accuracy. SE YOLO has the best precision, mean average precision, and subsequent fitness score. However, YOLF has a slightly higher recall. The takeaway from Table 4.4 is that SE YOLO and YOLF are close in detection accuracy, at least on the validation set, while LightConv and DWSConv fall behind in accuracy. We expect a similar result when performing predictions on the test set.

To evaluate the test set, we selected the best weights for each model based on the fitness metric and ran the predictions. The results in Table 4.5 highlight the detection accuracy metrics. It is evident from the table that SE YOLO outperforms all other models. However,

Table 4.4: Lightweight model validation results.

Model	Precision	Recall	mAP $_{50}$	mAP $_{50-95}$	Fitness
SE YOLO	**88.95**	77.40	**87.92**	**55.70**	**58.92**
YOLF	87.81	**78.39**	86.94	54.53	57.77
DWSConv	83.90	67.59	80.07	45.82	49.25
LightConv	85.65	69.19	79.47	47.78	50.95

Table 4.5: Lightweight model test results.

Model	Precision	Recall	mAP $_{50}$	mAP $_{50-95}$	Fitness
SE YOLO	**86.15**	**69.73**	**79.09**	**50.57**	**53.43**
YOLF	85.74	66.67	78.55	49.88	52.74
DWSConv	77.40	57.25	66.52	39.20	41.93
LightConv	83.69	61.16	72.70	43.85	46.73

YOLF also demonstrates good detection accuracy, with only a 0.69 difference in fitness score compared to SE YOLO. As expected, the detection accuracy for LightConv and DWSConv lags behind the previously mentioned models, with LightConv having a fitness score of 6.70 points lower than SE YOLO and DWSConv scoring 11.5 points lower.

Following the validation and test performance, we investigated the model inference time on the workstation and GAP8. The GAP8 inference time is of greater importance since the objective is to make a real-time application in which the drone can follow the hand. To compute the inference time, we measured the time taken to perform individual predictions for all 1,076 images in the test set. The inference results for the workstation, shown in Table 4.6, include the minimum, maximum, mean inference time, standard deviation, and the mean rate of predictions on the workstation. The mean inference time is similar for all the models, with SE YOLO again being the best model with a mean inference time of 2.13 ms or approximately 496 processed frames per second. However, it is also the model with the largest standard deviation of 0.77 ms.

As stated previously, the inference time on the GAP8 is of greater importance and will be discussed more thoroughly. The GAP8 is a much smaller computing platform, so the inference time can be assumed to be much greater, with greater deviations in mean inference time between the different models. In Table 4.7, it is evident that the fastest model is DWSConv, which achieves approximately 18 frames per second. LightConv achieves 17 frames per second, one frame less than DWSConv. However, YOLF has an approximately 2.2 times faster inference time than SE YOLO. Important to note as well is that the standard deviation for

Table 4.6: Inference time on the workstation.

Model	min [ms]	max [ms]	mean [ms]	std [ms]	mean rate [Hz]
SE YOLO	**2.05**	27.39	**2.13**	0.77	**469.36**
YOLF	2.11	21.12	2.20	0.58	455.55
DWSConv	2.18	19.50	2.29	0.53	436.15
LightConv	2.16	**19.29**	2.24	**0.52**	445.41

Table 4.7: Inference time on the GAP8.

Model	min [ms]	max [ms]	mean [ms]	std [ms]	mean rate [Hz]
SE YOLO	189.04	189.34	189.18	0.04	5.29
YOLF	85.56	85.71	85.63	0.02	11.68
DWSConv	**54.77**	**54.86**	**54.82**	**0.01**	**18.24**
LightConv	57.92	58.01	57.97	0.01	17.25

Table 4.8: YOLF test results.

Dataset	Precision	Recall	mAP $_{50}$	mAP $_{50-95}$	Fitness
FreiSUN	17.67	7.31	6.15	2.60	2.95
Himax	85.74	66.67	78.55	49.88	52.74
FreiSUN+Himax	**86.99**	**72.30**	**81.00**	**52.26**	**55.13**

the inference time is much lower for the workstation compared to the GAP8.

Due to time constraints, we selected one model for further improvements. Later, we will perform hyperparameter tuning, with just one round taking approximately 20 hours per model. Performing this for all models would require around 140 hours, assuming everything goes smoothly. Given this limitation, we focused on selecting one model for further improvements.

To summarize the results from this section, YOLF and SE YOLO demonstrate comparable detection accuracy on both the validation and test sets, with SE YOLO slightly outperforming YOLF by a fitness score margin of 0.69. However, SE YOLO is the slowest model, achieving only around five frames per second. In contrast, the other models all exceeded ten frames per second, with both LightConv and DWSConv outperforming YOLF in terms of speed. Despite this, YOLF was selected for further improvements due to its comparable detection accuracy and over 2.2 times faster inference speed compared to SE YOLO.

4.3.2 Transfer Learning

Following model selection, we investigated the effects of transfer learning on detection accuracy. For this purpose, the YOLF model was pretrained on the FreiSUN dataset for 100 epochs and then further trained on the Himax dataset for 500 epochs. The reasoning behind the number of epochs is detailed in Section 3.2.7. After training, the best model was selected based on the fitness score. Since the primary focus is the network's detection accuracy on new data, this section presents only the test results.

Table 4.8 displays the detection results on the test set for the YOLF model trained on different datasets. Initially, YOLF was trained on the FreiSUN dataset, achieving a fitness score of 2.95. When YOLF was trained on the Himax dataset, the fitness score improved significantly to 52.74. These results demonstrate the importance of using platform-specific images for training. Furthermore, the pretrained YOLF model, first trained on FreiSUN and then on Himax (referred to as FreiSUN+Himax) in Table 4.8, showed even greater improvement. Transfer learning increased all detection metrics, including the fitness score, from 52.74 to 55.13.

Based on these findings, transfer learning was deemed successful, and we will use this

Table 4.9: ASHA validation results.

Model	Precision	Recall	mAP $_{50}$	mAP $_{50-95}$	Fitness
Default	87.94	75.32	84.74	52.93	56.11
Round 1	88.47	**79.86**	87.01	50.95	54.56
Round 2	**91.36**	79.65	**89.19**	**55.09**	**58.50**

pretrained YOLF model in the next step for hyperparameter tuning.

4.3.3 Hyperparameter Tuning

Using the pretrained YOLF model, we tuned the hyperparameters to improve detection performance. Each round of ASHA search ran up to 100 epochs. Therefore, we extracted the best epoch within the first 100 from the pretrained YOLF model trained on Himax. This process enabled a comparison between the default hyperparameters and the newly tuned ones.

In the first round of hyperparameter tuning, we used a wider search space, but the detection accuracy was lower than that achieved with the default parameters, as shown in Table 4.9. Consequently, we shrank the hyperparameter space, as detailed in Section 3.2.7. The second round of tuning yielded better results, with a higher fitness score of 58.50 compared to 56.11 for the default parameters, as presented in Table 4.9. For the discussion of optimizing the hyperparameter space, the parameters before and after the successful tuning are shown in Table 4.10.

Table 4.10: Hyperparameters before and after tuning.

Hyperparameter	Before Tuning	After Tuning
Weight Decay	0.00050	0.00022
Warmup Epochs	3.00	24.86
Warmup Momentum	0.937	0.651
Box	7.50	8.08
Cls	0.50	0.79
Dfl	1.50	1.56
Translate	0.10	0.30
Scale	0.50	0.37
Shear	0.00	0.50
Perspective	0.00	0.00009
Fliplr	0.50	0.39
Mosaic	1.00	0.24
Mixup	0.00	0.02

We continued by training the pretrained and tuned YOLF model on the Himax dataset for 500 epochs, referring to this as the Tuned model in Table 4.11. It is important to note that while optimizing hyperparameters can give better detection accuracy on the validation. However, this does not mean that the performance on unseen data must be better; it is possible to overfit the validation data. Therefore, we used the best weights to run predictions on the test set to ensure fair evaluation of before and after hyperparameter tuning.

Table 4.11: Tuning test results.

Model	Precision	Recall	mAP$_{50}$	mAP$_{50-95}$	Fitness
Base	86.99	72.30	81.00	52.26	55.13
Tuned	87.99	72.57	81.41	52.57	55.46
Final	**89.69**	**72.77**	**81.97**	**52.95**	**55.86**

Comparing the Tuned model with the Final model in Table 4.11, we observed a slight increase in detection accuracy due to hyperparameter optimization. Additionally, we found that training without mosaics during the last 100 epochs and extending the training to 1000 epochs also contributed to performance improvements. The detection accuracy of the final model is presented in Table 4.11.

In summary, by pretraining the model and performing hyperparameter tuning, we increased the fitness score from 52.74 to 55.86, representing a 3.12 percentage point. The final model also outperformed the best initial SE YOLO model, achieving a fitness score of 55.86 compared to 53.43 on the test set and having a 2.2 times faster inference time.

4.3.4 Quantization

We evaluated the results of post-training quantization qualitatively by plotting the detection boxes from running inferences with and without quantization. Three examples are shown in Figure 4.2. The black boxes are before, and the white ones after quantization. The white numbers are the confidence scores. As seen in the figure, the results only differed by a few pixels for these three examples. We dare to say these examples are representative of how much the full-sized and quantized models differed when walking around filming for a few minutes in the office. We did not use our test set for this because transferring them to the drone and running inference one by one would take a whole day. As would running inferences on the entire test set in the GAP8 simulator.

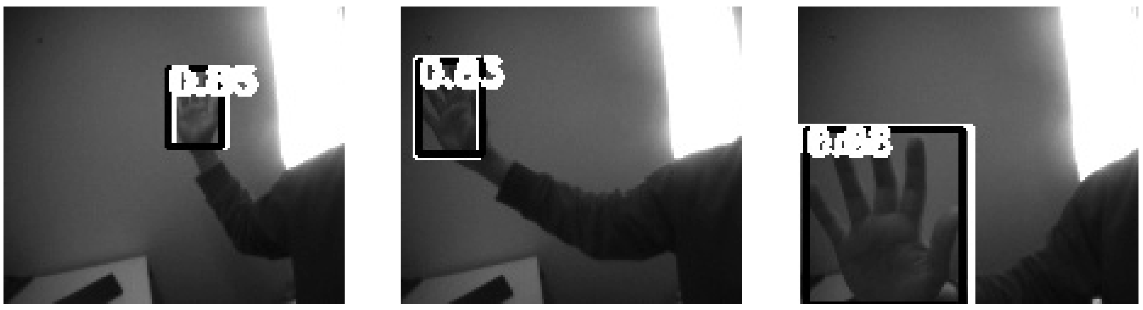


Figure 4.2: Detection results before and after quantization.

4.3.5 Full Implementations

As seen in Table 4.12, we managed to beat our reference implementation that had a sufficient setpoint frequency for position control with our fully on-board approach that also performed

Table 4.12: Setpoint latencies in the reference and edge implementation.

Implementation	min [ms]	max [ms]	mean [ms]	std [ms]	mean rate [Hz]
Reference	**124.45**	1479.04	170.04	46.31	5.88
Edge	137.03	**140.61**	**139.20**	**0.43**	**7.18**

Table 4.13: The durations of the main activities in the reference and edge implementation, other than inferences.

Activity	Unit	Implementation	min [ms]	max [ms]	mean [ms]	std [ms]
Capture QQVGA	GAP8	Edge	54.00	55.00	54.02	0.12
Capture QVGA	GAP8	Ref	60.00	61.00	60.44	0.50
WiFi Stream	GAP8	Ref	61.00	182.00	100.76	14.56
Radio	PC	Ref	0.20	0.57	0.25	0.04
UART	GAP8	Edge	0.00	1.00	0.07	0.25
Compute Setpoint	STM32	Edge & Ref	0.03	1.52	0.09	0.13

well on the monochrome images taken with the Himax camera.

To aid later discussion regarding how much of the decreased setpoint latency can be attributed to the model inference and how much can be attributed to other differences between the two implementations, we display the latencies of other activities in Table 4.13. For each profiled activity, statistics regarding its latency are shown, along with which processor it was profiled from and which implementation uses it. *Edge* denotes the final edge implementation where the inference runs onboard the drone, and *Ref* denotes the reference implementation where the inference runs on a personal computer.

4.4 Discussion

This section will discuss the results according to our research questions and bring up some less central considerations that are also interesting.

4.4.1 Constraints on Networks and Data When Using GAPFlow

The NNTool and Autotiler aid much of the porting and optimization of TFLite and ONNX models to the GAP8 processor, leaving only a few considerations to the developer, a subset of which are listed in this section. First, only TFLite and ONNX operations supported by GAPFlow can be used. Second, all weights and the largest simultaneous intermediate outputs must fit in the HyperFlash along with the operating system and instructions. Third, the Autotiler does not tile convolutional kernels like input and output tensors in convolutional layers, which puts a lower bound to how narrow the tiles can get. An input tile of size: [kernel width] × [input height] or [kernel height] × [input width], along with the corresponding output tile, kernel, and other parameters, must fit into L1 simultaneously, which limits the size of, kernels, input tensors and intermediate tensors in the used neural network.

Fourth, representative input data is needed for post-training quantization. The quantization error must be checked to be acceptable [11].

4.4.2 Lightweight Models

In the results, we compared the four lightweight models in terms of detection accuracy and inference time. After comparison, we decided that YOLF had the best trade-off between inference time and accuracy, making it the most suitable model for our real-time application.

Depthwise Separable Convolution and Light Convolution

Because light convolution is not as common as depthwise separable convolution, it was not supported with Leaky ReLU in PyTorch or GAPFlow, which is why we used it with ReLU. From our tests, it can not be said what effect the activation function had and what effect the order of depthwise and pointwise convolution had on the accuracy when trying to replace all relevant layers with depthwise separable convolution or light convolution.

Inference Time Versus Detection Accuracy

As previously stated, our results indicate a trade-off between inference time and detection accuracy. Lower detection accuracy in a real-time application like ours can lead to a higher effective inference time. For example, if a hand is detected in every other image, the effective detection rate is half of the actual rate. This assumes that all bounding boxes are correct, which is not always the case. Accurate detection is also crucial for providing correct bounding boxes and thereby determining the correct new setpoint for the drone. Our investigation concluded that YOLF offers the best balance between detection accuracy and inference time.

Our model architecture could still contain layers with many redundant parameters and, thus, a redundant ability to capture complex patterns. Replacing more layers with depthwise separable convolution or light convolution could improve inference speed without too much loss in accuracy. Other approaches to shrinking the model, such as reducing the input size, could also increase the inference speed.

Improving Detection Accuracy

During model training, we saw that training on platform-specific images had the largest impact on detection performance. However, transfer learning and hyperparameter tuning are notable techniques that can slightly improve detection accuracy without the need for more data or increasing the inference time.

The hyperparameter tuning led to improved performance, although the difference was marginal this time. As shown in Table 4.10, a slightly reduced weight decay proved more effective. This improvement can be attributed to the model previously underfitting the training data. By decreasing the weight decay, we allow the model to learn more complex patterns within the data.

This adjustment is particularly relevant, given the nature of our images. Our dataset consists of fairly similar images, suggesting that a degree of "overfitting" to the training data is desirable. Lower weight decay does this by reducing the regularization, thereby enabling the model to capture the more specific and consistent patterns in our platform-specific images.

Overall, the tuning process had the most significant impact on the augmentation parameters. We observed that techniques designed to increase dataset diversity, such as translation, scaling, and shearing, were important. This result seems appropriate since our dataset contains many near-duplicate images, as each scene was recorded at five frames per second.

However, augmentation parameters intended to introduce more complexity to each image, such as mosaics and mixups, remained low. This can be explained by the nature of our dataset and its intended application. YOLO is typically designed for multiclass object detection with one or multiple objects in each image. In contrast, our application focuses on predicting a single hand. Techniques like mosaics and mixups can often introduce multiple objects in the same image, which is undesirable for our use case.

We theorize that further optimization of the hyperparameters could enhance performance. This will be discussed in more detail in Section 5.1.1.

4.4.3 Flight Performance

Qualitatively, the drone's ability to react to a hand in front of it is as follows. The drone flickers its LED lights for as long as an upright palm is in its field of view, within approximately three meters distance. It positions itself 50 cm in front of the hand with a motion that is perceived as doing so directly. It can not, however, follow fast motions. If the hand is waved fast, going outside its field of view before its 139 ms setpoint delay and position step response allows it to adjust, it has no chance to follow.

Unsolved Control Challenges

Our position controller requires that we know exactly where to fly and that calculations are correct. The computed position is subject to a handful of assumptions and possible sources of error. We mention a few in this section. First, everyone's hand is assumed to be equally large and in the exact same pose. Angeling the hand reduces its size on the screen. Second, the camera plane is assumed to be straight, but the Himax camera has a slight fish-eye effect. Third, the constant relating hand size and distance was experimentally picked with potential errors in measured distance and detected hand size. Fourth, the camera is not pointing completely straight, and different AI decks have different pitches on the camera.

All these errors and assumptions would lead to a proportionally erroneous offset from the drone that it must move. For example, if someone has small hands, the distance will be estimated as far, and the offset position will be larger. The drone will then move too far. This could be solved with a velocity controller with a derivative term. When noticing the drone is moving too quickly and will overshoot, it could compensate.

Flow Deck Limitations

With the Flow deck attached, the Crazyflie can estimate its absolute horizontal position, but it works best in well-lit situations, above a mate, non-black floor with a non-repetitive pattern [1]. Errors in estimated translation accumulate to an erroneous absolute position. We did much of our testing over a floor with a repetitive grid pattern, which caused the drone to move longer than it should when a desired position was set. By processing images at a sufficient rate, the drone never diverged from the desired location. But some swaying could be seen, about a few centimeters in length. When flying over a brighter floor with a natural pattern, this was reduced to about one centimeter.

4.4.4 Onboard Versus Remote Inference

Inference running onboard the drone can potentially achieve a higher setpoint frequency by cutting out the image streaming. In our case, our fully onboard implementation beat the setpoint frequency in our baseline implementation with remote inference. We used the reference implementation to determine the latency to aim for as a step in developing a working fully onboard application, proving its feasibility, and exploring DNN model development targeting edge devices. But with the two implementations done, a comparison between the two approaches is in place.

For the rest of this discussion, we neglect the time it takes to send the inference results to the drone's main microcontroller and compute a setpoint, as this only took 0.39 ms in the reference implementation and 0.34 ms in the edge implementation.

With our setup, the approach with remote inference can approximately achieve a minimal setpoint latency as in Equation 4.1 since the image capture and inference are done in parallel. If we used our final YOLF model on the personal computer and QQVGA image resolution, the time to capture an image would exceed the inference time, so the first term of Equation 4.1 would be 54 ms. We never profiled streaming of QQVGA images, but assuming it is a fourth of the time to stream QVGA images, the second term of Equation 4.1 would be 25 ms, resulting in a setpoint latency of about 79 ms. We mention this so as not to unfairly claim that our final edge implementation is better than remote inference on the given hardware. We have only concluded that our edge implementation with a setpoint latency of 139.20 ms beat the 170.04 ms latency we knew we needed from our reference implementation.

$$\text{Latency} = \max([\text{Capture Image}],\ [\text{Remote Inference}]) + [\text{Stream Image}] \tag{4.1}$$

$$\text{Latency} = [\text{Capture Image}] +\ [\text{Edge Inference}] \tag{4.2}$$

$$\text{Latency} = \max([\text{Capture Image}],\ [\text{Edge Inference}]) \tag{4.3}$$

With the setup in our edge implementation, the minimal setpoint latency can be approximated with Equation 4.2. By comparing Equation 4.1 and 4.2, it becomes clear that capturing an image is indispensable, and the edge implementation is faster if the edge inference is faster than streaming the image.

If the image capture and inference can be done in parallel on the drone, the minimal setpoint latency in the edge approach would be approximated as in Equation 4.3. We did not look into parallel image capture and inference, but for onboard implementations in general, it is possible. Then, an edge implementation has an even larger latency advantage over central computation.

4.4.5 Security and Ethical Implications

Working with machine learning and computer vision on edge devices raises significant ethical and security concerns, including the dual-use dilemma, algorithm bias, and privacy issues.

AI is a dual-use technology, meaning it has the potential for both beneficial and harmful applications [31]. Our thesis focuses on an edge AI application for drones, specifically utilizing object detection onboard AI-powered drones. This technology has numerous beneficial uses, such as aiding first responders. For instance, the company Skydio employs fully autonomous drones that can be deployed rapidly, potentially saving lives [33].

However, the same technology could be detrimental if misused. Attaching explosives to autonomous drones could turn them into autonomous weapons, demonstrating a risk if the technology falls into the wrong hands. Ultimately, the interpretation and ethical considerations of these applications often depend on the perspective of the observer.

Another significant issue in the computer vision and machine learning space is fairness and bias. Machine learning algorithms can perpetuate societal biases [31]. In our thesis, we trained an object detection algorithm, which is likely to overfit the characteristics of the individuals in the dataset. This means its detection accuracy might be poorer on individuals with differing appearances. However, as this is a research project and not a full-scale application ready for production, we have determined that this dataset is sufficient to demonstrate our research questions as outlined in Section 1.1.

Chapter 5
Conclusions

In this final chapter, we will answer the research questions posed in section 1.2 and present ideas for future works.

- What challenges arise when developing and deploying deep learning models to edge devices?

Most of the challenges we faced when deploying deep learning models to the edge stemmed from the unique features of the GAP8 processor, which make it ultra-low power. The relatively small computational power and memory on edge devices put size constraints on which networks can fit in the memory and be run within an acceptable timeframe. The memory levels on the GAP8 not being cache mapped leave the challenge of planning data transfers efficiently to the application. Furthermore, the GAP8's SIMD instructions assuming 8-bit word length and lack of floating point instructions advocates for using 8-bit integers.

- What strategies and techniques can be used to address these limitations and challenges?

Tools can be used to optimize neural networks for efficient hardware utilization, leaving the developers to create architectures that meet the tools' constraints and decide on a model size that balances accuracy and latency. We used Greenwaves NNTool to quantize our networks to 8-bit integers, decreasing memory footprint and latency without significant loss in accuracy. To hard-code memory transfers during inference, we used Greenwaves Autotiler, which allowed us to deploy a model much larger than the smallest two memory layers but forced our network to fulfill requirements discussed in section 4.4.1. To increase the inference speed when utilizing the hardware efficiently, we reduced the number of MACs in our network by using a variation of depthwise separable convolution without much loss in accuracy.

- How can training data be collected, and what training practices can improve detection accuracy?

In this thesis, we observed that platform-specific training data significantly improved detection accuracy, which could be further improved using transfer learning and hyperparameter tuning. To achieve this, we created our own dataset using the drone camera containing over 9,000 images. These images were annotated using a model trained on open-source datasets, which reduced the amount of manual labor. As for the training practices, we noticed that transfer learning had the biggest improvement in detection accuracy. During the hyperparameter tuning, we saw that techniques that aim to increase diversity were very important, while augmentation strategies that increase complexity are less so.

5.1 Further Work

In this section, we present some potential continuations of our work.

5.1.1 Dataset and Training

As discussed in Section 4.4.5, the bias and fairness of our model are not ideal since the dataset only contains two different persons. To improve upon this, we suggest a larger dataset be created that encompasses a wider area of people with differing characteristics. Additionally, during flight, we noticed that the hand is best detected if placed with the palm facing the camera; this is how we intended the application to work since we calculate the distance from the hand based on the diameter of the bounding box. However, this means the dataset is very specific to our problem.

Furthermore, during the training, specifically the hyperparameter tuning, we noticed a few things that could be improved. For a similar problem to ours in which one hand should be detected, we suggest focusing on parameters that maximize the dataset diversity while focusing less on augmentation techniques that aim to introduce more complexity, such as mosaics.

During the thesis, we have also omitted some tuning parameters, some of which might be beneficial if a larger hyperparameter tuning session is run. For example, testing different optimizers and learning rates could prove beneficial in increasing detection performance.

5.1.2 Network Architecture

We used a network architecture called Squeezed Edge YOLO that we modified slightly to decrease the inference time. This section describes further work regarding network architectures.

Ground-up Approach

In our thesis, we worked with large deep neural networks for object detection in an attempt to fit these larger models onto the target platform. However, using a ground-up approach, in which layers are added until sufficient detection accuracy is reached, might be preferable to finding the optimal model for the problem. As seen in the PULP-Frontnet 2.3.1 paper, they were able to achieve regression of the 3D location and vertical rotation of a face with a much smaller network containing only a total of 8 layers.

Further Investigation of Depthwise Separable and Light Convolution

Using depthwise separable convolution and what we call light convolution has the potential to significantly reduce the inference speed without necessarily affecting performance. We only tried three different variations of replacing layers with light convolution or depthwise separable convolution, but it is possible that replacing a different subset of layers would achieve a better performance-latency tradeoff.

Using a combination of light and depthwise separable convolution could potentially be useful. The pointwise convolution changes the number of channels and the depthwise convolution has one kernel per either input or output channel, depending on whether it is performed first or last. This means that if a low number of MACs is desired, the depthwise convolution should be performed before pointwise in layers where the channels increase and after in layers where the channels increase. If more parameters are desired, the order should be the opposite.

Other Ways of Increasing Inference Speed

There are a number of ways the current model architecture can be modified to achieve better inference speed, potentially without much loss in performance. We did not try these due to time constraints. As earlier mentioned, most of the inference speed is consumed by the multiply-accumulate operations in the convolutional layers. These can be reduced by decreasing the input size or keeping the same input size but performing max-pooling early. Layers can be removed, and the number of intermediate channels can be reduced. A coupled YOLO detection head can be used. It is plausible that the 15 ms decoupled head we used is overkill because we only detected one object class, while even the 1 ms coupled YOLOv5 head can handle 80 object classes. Another approach could be to use a decoupled head but compute the confidences first, and then only compute one box instead of always computing 240 alternatives. Roughly 10 ms goes to computing the boxes in our implementation.

5.1.3 Control

A smaller model with a faster inference speed would open up more interesting alternatives for control. One approach could be to use velocity control, keep track of the target's position and velocity in a Kalman filter, and add the target's estimated velocity to the drone's desired velocity as in [29].

Bibliography

[1] Bitcraze AB. *Go with the Flow: Relative Positioning with the Flow Deck.* `https://www.bitcraze.io/2023/11/go-with-the-flow-relative-positioning-with-the-flow-deck/`. (Accessed: 2024-04-03). 2024

[2] Sean Bell et al. "Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016, pp. 2874–2883. DOI: `10.1109/CVPR.2016.314`

[3] Bitcraze AB. *AI deck 1.1.* `https://www.bitcraze.io/products/ai-deck/`. (Accessed: 2024-02-14). 2024

[4] Bitcraze AB. *Crazyradio 2.0.* `https://www.bitcraze.io/products/crazyradio-2-0/`. (Accessed: 2024-05-24). 2024

[5] Bitcraze AB. *Flow deck v2.* `https://www.bitcraze.io/products/flow-deck-v2/`. (Accessed: 2024-02-27). 2024

[6] Bitcraze AB. *Repository Overview.* `https://www.bitcraze.io/documentation/repository/`. (Accessed: 2024-05-31). 2024

[7] Bitcraze Forums. *Flying Crazyflie to a specific position setpoint.* `https://forum.bitcraze.io/viewtopic.php?t=4125`. (Accessed: 2024-03-18). 2024

[8] François Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 1251–1258

[9] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning". In: *Neural networks* 107 (2018), pp. 3–11

[10] Priya Goyal et al. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. 2018. arXiv: `1706.02677 [cs.CV]`

[11] GreenWaves Technologies. *GAP8 Manual.* `https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html`. (Accessed: 2024-02-14). 2024

[12] GreenWaves Technologies. *SDK for Greenwaves Technologies GAP8 IoT Application Processor*. `https://github.com/GreenWaves-Technologies/gap_sdk`. (Accessed: 2024-04-08). 2024

[13] GreenWaves Technologies. *Ultra low power GAP processors*. `https://greenwaves-technologies.com/low-power-processor/`. (Accessed: 2024-04-05). 2024

[14] GreenWaves Technologies. *Which AI models can run at the Very Edge?* `https://greenwaves-technologies.com/which-ai-model-can-run-on-the-very-edge/`. (Accessed: 2024-05-22). 2024

[15] Himax Technologies. *HM01B0 Datasheet*. `https://www.uctronics.com/download/Datasheet/HM01B0-MWA-image-sensor-datasheet.pdf`. (Accessed: 2024-04-05). 2024

[16] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017)

[17] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-Excitation Networks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7132–7141. DOI: `10.1109/CVPR.2018.00745`

[18] Edward Humes, Mozhgan Navardi, and Tinoosh Mohsenin. "Squeezed Edge YOLO: Onboard Object Detection on Edge Devices". In: *arXiv preprint arXiv:2312.11716* (2023)

[19] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713

[20] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. *Ultralytics YOLO*. `https://github.com/ultralytics/ultralytics`. Version 8.0.0. (Accessed: 2024-05-31). 2024

[21] Uday Kulkarni et al. "A Survey on Quantization Methods for Optimization of Deep Neural Networks". In: *2022 International Conference on Automation, Computing and Renewable Systems (ICACRS)*. 2022, pp. 827–834. DOI: `10.1109/ICACRS55517.2022.10028742`

[22] Lorenzo Lamberti et al. "Low-Power License Plate Detection and Recognition on a RISC-V Multi-Core MCU-Based Vision System". In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: `10.1109/ISCAS51556.2021.9401730`

[23] Liam Li et al. "A system for massively parallel hyperparameter tuning". In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 230–246

[24] Ellen Lindgren. *Target Recognition and Following in Small Scale UAVs*. 2022

[25] Shihan Liu et al. "EdgeYOLO: An edge-real-time object detector". In: *2023 42nd Chinese Control Conference (CCC)*. IEEE. 2023, pp. 7507–7512

[26] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. Atlanta, GA. 2013, p. 3

[27] Dario Mantegazza et al. “Vision-based Control of a Quadrotor in User Proximity: Mediated vs End-to-End Learning Approaches”. In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 6489–6495. DOI: 10.1109/ICRA.2019.8794377

[28] Mozhgan Navardi, Edward Humes, and Tinoosh Mohsenin. “E2EdgeAI: Energy-Efficient Edge Computing for Deployment of Vision-Based DNNs on Autonomous Tiny Drones”. In: *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. 2022, pp. 504–509. DOI: 10.1109/SEC54971.2022.00077

[29] Daniele Palossi et al. “Fully Onboard AI-Powered Human-Drone Pose Estimation on Ultralow-Power Autonomous Flying Nano-UAVs”. In: *IEEE Internet of Things Journal* 9.3 (2022), pp. 1913–1929. DOI: 10.1109/JIOT.2021.3091643

[30] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788

[31] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson Education, 2021. ISBN: 9781292401171

[32] Hazim Shakhatreh et al. “Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges”. In: *IEEE Access* 7 (2019), pp. 48572–48634. DOI: 10.1109/ACCESS.2019.2909530

[33] Skydio Inc. *Skydio Public Safety*. https://www.skydio.com/solutions/public-safety/. (Accessed: 2024-05-28). 2024

[34] *TensorFlow Lite 8-bit quantization specification*. https://www.tensorflow.org/lite/performance/quantization_spec. (Accessed: 2024-04-05). 2024

[35] Ultralytics Inc. *Ultralytics YOLO Docs*. https://docs.ultralytics.com/. (Accessed: 2024-05-31). 2024

[36] Andrey Vakunov et al. “MediaPipe Hands: On-device Real-time Hand Tracking”. In: (https://mixedreality.cs.cornell.edu/workshop). 2020

[37] Sebastien C Wong et al. “Understanding data augmentation for classification: when to warp?” In: *2016 international conference on digital image computing: techniques and applications (DICTA)*. IEEE. 2016, pp. 1–6

[38] Jianxiong Xiao et al. “SUN database: Large-scale scene recognition from abbey to zoo”. In: June 2010, pp. 3485–3492. DOI: 10.1109/CVPR.2010.5539970

[39] Fuzhen Zhuang et al. “A comprehensive survey on transfer learning”. In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76

[40] Christian Zimmermann et al. “Freihand: A dataset for markerless capture of hand pose and shape from single rgb images”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 813–822

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-05

EXAMENSARBETE Real-Time Edge AI Hand Detection for Drone Controls
STUDENTER Joel Nygren, Oliver Lövström
HANDLEDARE Flavius Gruian (LTH), Irena Ilvesten (Combine Control Systems AB)
EXAMINATOR Michael Doggett (LTH)

Edge AI: Real-Time Hand Detection on a Palm-Sized Drone

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joel Nygren, Oliver Lövström**

Nowadays, microprocessors exist everywhere, from your smart fridge to your smart plant pot. Together with advancements in computing, it enables the integration of artificial intelligence (AI) for these devices. To investigate this possibility, we have implemented an AI system onboard a palm-sized drone, enabling it to follow a hand.

Today, computers are getting smaller, while artificial intelligence models are getting more sophisticated. This introduces a new possibility: running AI on compact systems, also known as edge AI.

Edge computing is increasingly in demand due to its advantages. Since no data has to be sent to a central processing server, performing computations directly on the device reduces processing time for real-time systems. Additionally, there is no need for a connection between the device and an external system, eliminating the need for network coverage.

To investigate the challenges of edge AI, we have developed a real-time edge AI application on a palm-sized drone. The drone is equipped with an ultra-low-power processor and a camera, together weighing less than five grams. Using these, the onboard AI algorithm can detect hands in real-time, enabling the drone to reposition itself to stay in front of the hand.

When deploying software to an edge device, the small memory and computational power limit the size of the program. We found that one of the hardest challenges in this work was to program the specialized processor on the drone. Apart from being small, it has some unique properties that make it ultra-low power, consuming around 1,000 times less power than a personal computer. It does not have any data cache. Normally, a computer has a RAM and one or several smaller but faster cache memories, between which data is automatically transferred to make them work like one large and fast memory. But on this device, all memory accesses during execution have to be planned beforehand in a smart way so that the smaller, faster memories are used the most, and the larger, slower memories are used the least.

To utilize the memory hierarchy efficiently, we used neural networks, where the execution could be split into tasks, each fitting into the smallest and fastest memory. As a result, we were able to produce a model that fits into the memory, can be executed with over 11 frames per second, and can detect a hand accurately enough to allow for a satisfying interactive flying experience.