Low-Power Person Detection for UAV Search and Rescue Operations

Tiny Machine Learning

Students: Timon Coucke, Gabriele Pattarozzi, Jakub Schwenkbeck.

Supervisor: Rahim Rahmani.

Department of Computer
and Systems Sciences

project 7.5 HE credits
Computer and Systems Sciences
Degree project at the master level
Autumn term 2025/26

Abstract

When rescue teams need to search for persons in remote or hard to access areas, they increasingly rely on Unmanned Aerial Vehicles (UAVs) because these can easily reach and cover large areas. These UAVs are equipped with onboard cameras to help rescuers locate the persons. However, continuously sending a live video feed to the ground team quickly drains the limited battery life of the UAV. To extend the operational time, this project proposes deploying a very lightweight person detection model capable of running local inference directly on the drone's hardware. The feed is transmitted to the ground team only when a person is detected in the camera frame. Running inference on the edge minimizes the latency by removing the need for cloud communication and enabling near real-time detection. This edge deployment introduces new challenges like the need to balance the trade-off between low power consumption, high detection accuracy, and low inference time to achieve optimal performance for real-world search and rescue missions.

Keywords: Edge computing, UAV, person detection, lightweight model, real-time inference, power efficiency.

Table of contents

Chapter 1: Introduction

Background

In remote or hard to access areas such as snowy mountains, maritime settings or other complex terrains, Search and Rescue (SAR) operations face challenges such as weather conditions or inaccessibility while having the urgency to locate missing persons. Unmanned Aerial Vehicles have become an increasingly valuable tool in these missions as they can cover large areas quickly and access locations which are difficult or dangerous for the human rescuers. These UAVs often have cameras to detect and locate humans from the air, but the continuous video transmission to the ground team of rescuers consumes a lot of power which limits the flight time and therefore efficiency while rescuing.

Advances in Tiny ML and edge computing provide a solution to this problem. By deploying lightweight person detection models directly on the UAV hardware it is possible to process the video frames locally and only transmit relevant frames when a person is detected. This can reduce the power consumption and minimize latency which is critical for SAR operations with time pressure. Achieving this requires balancing the model size, inference speed, and detection accuracy as well as optimizing hardware performance to maximize operational efficiency.

Problem Statement

In search and rescue operations, UAVs can be used to cover large and often inaccessible areas. The problem is that transmitting live video consumes a lot of power, limiting flight duration due to short battery life. There is a need for fast and energy-efficient on-board person detection models that can recognize humans in real time, allowing the UAV to send data only when necessary. This problem is significant because model efficiency directly improves UAV flight time, which increases coverage area and improves the chances of locating missing persons while keeping energy consumption to a minimum. That’s why balancing model size, accuracy, and inference speed is critical for practical, low-power SAR applications.

Research Questions and Objectives

Our project aims to investigate the following:

- Which lightweight person detection model (Mobilenet-SSD, YOLO, FOMO, EfficientDet) provides the best balance of accuracy, speed, and power efficiency for Edge UAV deployment?
- How can quantization techniques from our lecture be applied to reduce model size and energy consumption without significantly losing detection accuracy?
- How do different edge hardware platforms affect model performance, inference speed, and power draw?
- What is the best combination of model and hardware for a solar-powered UAV performing real-time search and rescue operations?

Chapter 2: Extended Background and Related Work

Literature review and contextualizing the work

Tiny Machine Learning enables deploying neural networks on devices with constrained resources like microcontrollers and small and mobile computers. Typical TinyML models have to operate under strict constraints (such as low memory space, low computational force, and limited power supply). The rather recent improvements in research ("MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers") and in frameworks like TensorFlow Lite, PyTorch Mobile, and Edge Impulse have accelerated research in real-time inference on the edge. This is crucial for UAVs, where communication bandwidth and energy are limited.

A good start in literature is exploring different quantization techniques to make Neural Networks for person detection more efficient in the size vs. accuracy trade-off. We considered Post Training Quantization (PTQ) and Quantization Aware Training (QAT) methods. PTQ offers a faster and simpler approach by quantizing a pre-trained model without additional fine-tuning, making it a good choice for scenarios with limited computational resources for training. On the other hand, QAT involves retraining the model with quantized parameters to lower accuracy loss due to quantization, but it requires more computational overhead and training time. This project deploys PTQ with a pre-trained and finetuned (transfer learned) model ("A Survey of Quantization Methods for Efficient Neural Network Inference").

After evaluating these methods, this project did not implement pruning techniques in the models. Pruning involves removing some redundant parameters from a trained network to reduce its size and computational demands. While pruning can effectively compress models, studies ("Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy") suggest it also can lead to performance degradation, especially when evaluated beyond standard test accuracy metrics. Studies like the one mentioned have shown that pruned networks might not generalize well to outlier data and can be less resilient to noise, which raises concerns about their reliability in real-world applications. Therefore, we focused on quantization techniques to achieve model efficiency without losing too much performance.

So, to contextualize, this project is settled in the field of Tiny Machine Learning and its application on UAVs for person detection, with a strong focus on different quantization techniques and their influence on the accuracy and other relevant benchmarks. It specifically investigates if a low-power approach is possible with good results, and explores the trade-offs for solar-powered platforms.

Chapter 3: Methodology

This chapter describes the methodological approach adopted for the development and evaluation of the project. It details the technologies, frameworks, and tools used, the hardware setup, and the general approach to ensure reproducibility and efficiency.

Development methodology

The project followed an iterative, agile-inspired methodology with short development cycles. Each iteration combined experimental implementation and performance evaluation:

1. **Model Selection** - Comparing candidate detection models (MobileNet-SSD, YOLO, EfficientDet, FOMO) based on accuracy, inference speed, and power efficiency.

2. **Dataset** - Use the COCO Dataset for baseline training and use datasets like VisDrone, C2A, and Manipal for domain specific fine tuning.

3. **Model Optimization** - Using TensorFlowLite in Python for quantization (float16, int8, dynamic int8) and trying built-in sparsity techniques to reduce model size and memory usage.

4. **Hardware Benchmarking** - Deploying models on edge devices (Raspberry Pi 4B and 5) to measure inference time, memory usage, CPU load, and energy consumption.

Tools and technologies

Programming language: Python 3.11 has been selected for its ease of use, development speed, and great support for machine learning libraries. An older version has been selected because of Pi Camera 2 and TensorFlow Lite Runtime dependencies.

For building, training, and optimizing our models, the following frameworks and libraries have been used:

- TensorFlow – the main deep learning framework used for model training and evaluation.
- TensorFlow Lite – for optimizing models and deploying them on edge devices.
- Keras – high-level neural network API for TensorFlow model design.
- CUDA – NVIDIA’s parallel computing platform for GPU acceleration.
- Ultralytics YOLO – for implementing lightweight YOLO-based object detection models.

Hardware setup

For heavy model training tasks, the university lab PCs have been used, equipped with NVIDIA's ADA 4000 (~20 GB VRAM) or RTX 4080 (~16 GB VRAM) GPUs, which provide sufficient processing power for the high deep learning workloads.

After the model training, the optimized models have been deployed on a Raspberry Pi 5, which served as the edge device to simulate UAV onboard hardware for real-time inference testing.

Data and experimental environment

For datasets and pre-trained models, websites such as Kaggle, Tensorflowhub, and Hugging Face have been used. Some of the datasets explored were also private, so an access request was needed. More details can be found in the “Dataset selection” paragraph.

Edge Impulse cloud platform has been a useful resource for model deployment and optimization on edge devices.

Evaluation tools

System performance has been evaluated using Python scripts for metrics computation (accuracy, latency, FPS, system resources usage, temperature, etc.), taking the average among various detections and creating JSON profiling reports.

Reproducibility and versioning

Git has been used for the project's code version control, as well as **GitHub** to share and manage the code collaboratively.

For the final code delivery, a **Python Notebook** has been created to provide both a broad project documentation and easy reproducibility.

Timeplan

To structure the flow and workload of the project, we initially created a time plan to which the team stuck throughout the project development. As it shows, the main focus and time effort was on optimization and hardware benchmarking.

Task	W38	W39	W40	W41	W42	W43	W44	W45
Project plan	20/9							
Model selection and DataSet		28/09						
Model optimization			▨	12/10				
Hardware benchmarking				▨	19/10			
Solar-powered UAV design						26/10		
Project report and code	▨	▨	▨	▨	▨	▨	30/10	

Chapter 4: Testing and Implementation

Details of implementation and testing procedures

The initial implementation of the object detection algorithms has been done on normal PCs for comparing the models and datasets, and narrowing the selection of them. A modular script has been written to make it easy to switch between them, and a first testing comparison has been done, collecting execution times and videos of the results. The most promising models and datasets have then been selected to be implemented in the embedded system for further benchmarks.

Model Comparison and Selection

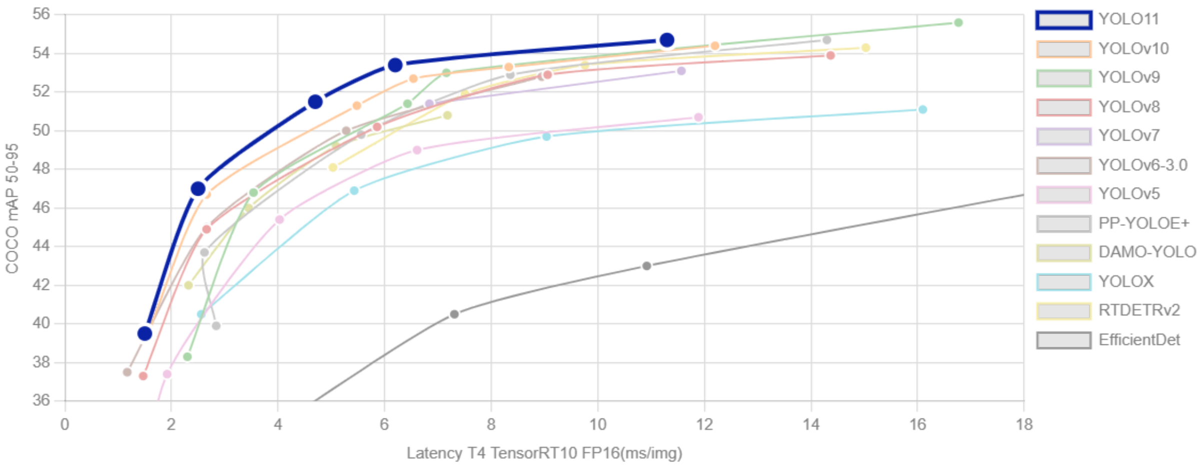
We started to compare different person/object detection models and compare their size, accuracy, inference speed, and power consumption. In total, we compared models from YOLO, FOMO, MobileNet-SSD, and EfficientDet.

MobileNet-SSD is a Single-Shot Detection model optimized for resource-constrained devices like smartphones. It offers real-time inference, and it has a relatively small size, making it ideal for Computer Vision mobile applications. The downside is that it is quite old, and the accuracy is mediocre compared to modern YOLO or EfficientDet.

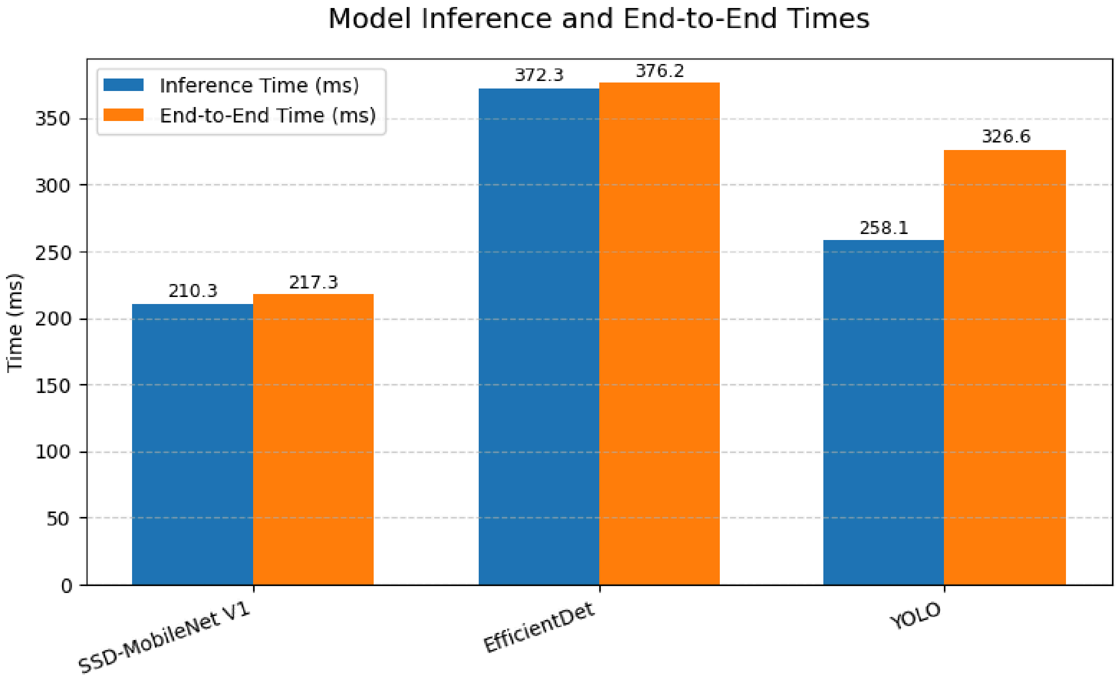
EfficientDet is an object detection model developed by Google with great accuracy and scalability, but in general, slower than YOLO on equivalent hardware and harder to optimize for inference on ARM without hardware acceleration.

YOLO model, developed by Ultralytics (“Ultralytics YOLO Docs”, n.d.), is a state-of-the-art real-time object detection model. It is available in multiple versions and formats (nano, small, large, etc.). Yolo11 is the current latest version (Yolo12 is actually available, but it seems to be a version developed by the community and not yet fully supported), and Yolo11n has been chosen since it is the lightest version and best suited for this project’s application.

Here follows a comparison between Yolo versions and other models, including EfficientDet, done by Ultralytics:



It is clear that from this benchmark, Yolov11 performs better than the other models in both precision and latency, but we will still test the main alternatives to prove that.



After doing this benchmark, it is clear that EfficientDet is too slow and it also has inferior accuracy. MobileNet is faster, but it performs worse than Yolo in detecting people. Yolo seems to have the best overall performance, since the inference time is the most important metric, and it also seems to detect more people.

FOMO ("FOMO"), Faster Objects, More Objects, is a machine learning algorithm developed by Edge Impulse. It brings object detection to highly constrained devices, using less processing power and memory than MobileNet SSD (which it is based upon). The problem with this model is that the only way to train it is through the Edge Impulse workspace, and due to budget restrictions, it is only possible to train it with a couple of hundred images (too few to obtain good results). Although the inference speed was impressive (about 20 ms per image), the model produced unusable results, detecting hundreds of objects in images that contained no humans.

In the end, YOLO11n has been chosen as the model to further train and optimize for its better overall performance.

Dataset Selection

To fine-tune and train the selected model, datasets were searched for on platforms such as Kaggle and Hugging Face. The focus was on domain-specific datasets containing overhead drone imagery of people in rural environments. The following is a complete list of the datasets that were explored:

- The COCO dataset ("COCO dataset") has been used as a baseline, since the majority of available models have been pre-trained with this dataset.
- The VisDrone dataset ("VisDrone Dataset") is one of the most cited in the UAV area, since it contains drone footage in different scenarios and with different detection classes (pedestrian, car, bike, etc.).
- The C2A dataset ("C2A Dataset: Human Detection in Disaster Scenarios") is specifically made for UAV human detection in disaster scenarios, but the images have a very low quality and they are often "artificial" (humans photoshopped into disaster pictures).
- The Manipal dataset ("Manipal UAV Dataset") requires a request to be sent to access it for academic purposes. It contains multiple shots taken from UAV videos with people detected.

- The SeaDronesSee dataset (“SeaDronesSee dataset”) focuses on maritime footage for Search and Rescue operations. It has been explored only to propose a possible implementation of this project in this field.
- The UAVDT dataset focuses only on vehicles, so it is not suitable for this project.
- The DetReIDX dataset (“DetReIDX”) has not been explored due to lack of access permissions.
- The Heridal dataset is another alternative, not explored due to a lack of access permissions.

In the end, COCO dataset have been used as the general training dataset for the models (already done by the models developers), and the following datasets have been selected for transfer learning and further performance analysis: VisDrone, C2A, and Manipal.

Model optimization

After successfully training the YOLO models, the next step was to optimize them for deployment on the Raspberry Pi. TensorFlow Lite was used to convert the trained models into formats suitable for edge device inference. The final YOLO models were converted to four different versions: 32-bit floating point, 16-bit floating point, dynamic 8-bit integer, and full 8-bit integer quantization. Thanks to the Ultralytics framework, in the export process, it was possible to specify a custom validation dataset (included in the previously discussed datasets) to calibrate the 8-bit integer quantized model and to obtain better dynamic ranges.

In the initial optimization attempt, the models were quantized post-training using 3 different optimization options: `DEFAULT`, `OPTIMIZE_FOR_LATENCY`, `OPTIMIZE_FOR_SIZE`. However, it was determined that the two options, `OPTIMIZE_FOR_LATENCY` and `OPTIMIZE_FOR_SIZE`, have been deprecated and now function identically to the `DEFAULT` setting (“tf.lite.Optimize”, n.d.).

The new approach focused on the DEFAULT optimization and also used the EXPERIMENTAL_SPARSITY flag. “*This optimization handles examining the model for sparse tensors and converting them to an efficient storage format. It also works seamlessly with quantization, and you can combine them to achieve more aggressive model compression.” (“Build fast, sparse on-device models with the new TF MOT Pruning API*”).

The sparsity optimization was not used on the 32-bit float because it was only considered a benchmark model, nor in the fully integer quantized model, since the hardware platform used didn’t show promising results with this size. This led to the creation of six different model variants for benchmarking.

Hardware Benchmarking

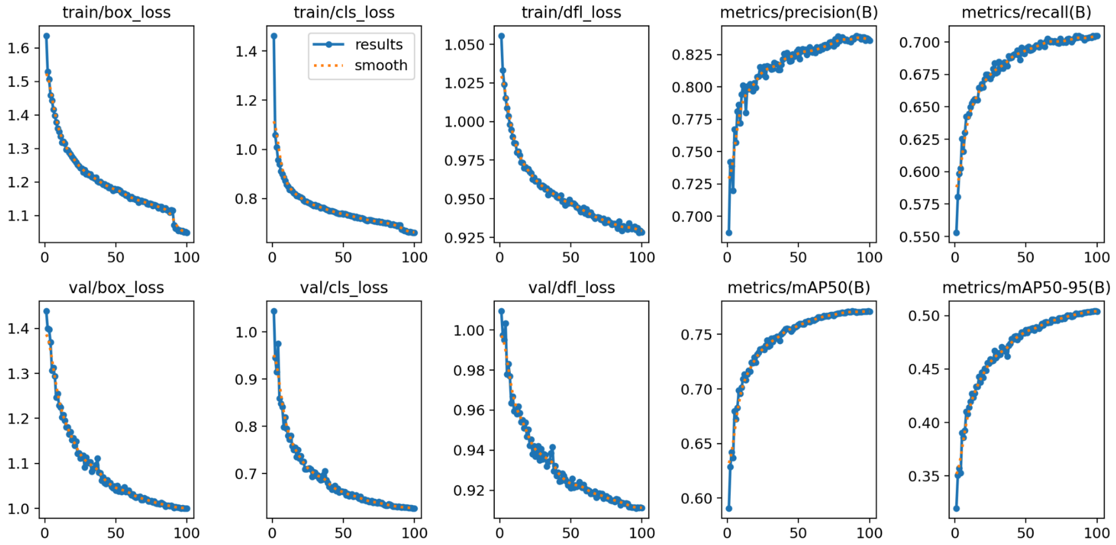
To evaluate the performance of the optimized models on the Raspberry Pi, different metrics were benchmarked: inference time, memory usage, CPU usage, CPU temperature, and estimated energy consumption.

- **Inference Time**: Calculated by measuring the execution time of multiple model predictions on test images. It was recorded using the default Python time library, taking the mean across multiple runs to minimize fluctuations caused by other processes.
- **Memory Usage:** Monitored using the memory_profiler Python library, which records the memory consumption of an individual function over a certain period. They were recorded at regular intervals of 10 milliseconds. Averaging these values gave a good approximation of the memory usage during model inference. The memory profiling was done in a separate inference run from the time benchmarking to ensure there was no interference with each other.
- **CPU Usage and Temperature**: The psutil Python package was used to measure CPU usage during inference. It gave a real-time percentage showing the overall processor load caused by the running model. For temperature monitoring, the Linux command `vcgencmd measure_temp` was called from within the Python benchmark script.

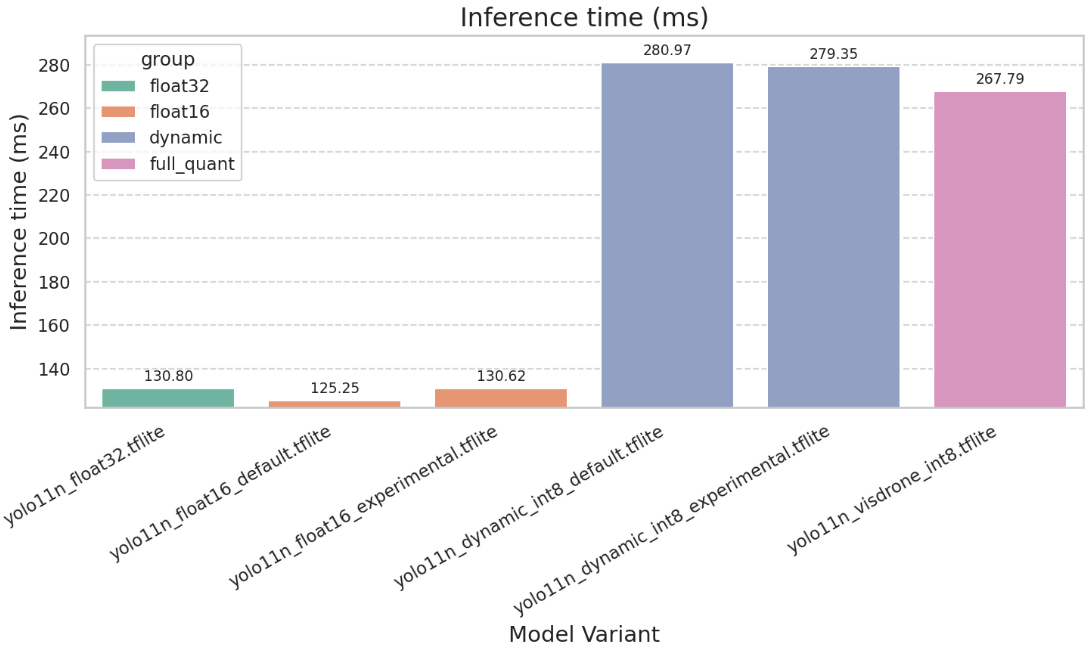
- **Energy Consumption:** The Raspberry Pi does not have a built-in power sensor, so the energy consumption was approximated based on the CPU usage and known estimated idle and max load power consumption of the Raspberry Pi 5. The estimation formula is: E = P_idle + (P_max - P_idle) * CPU_usage / 100. With P_idle = 2.7W and P_max = 7W ("Raspberry Pi 5 power consumption", n.d.).

These benchmarks were initially run on a Raspberry Pi 4B, but the lack of an active cooler and the high temperatures caused the CPU to throttle. The benchmarks have then been run again on a Raspberry Pi 5 with an active cooler, to prevent thermal throttling and to ensure consistent and comparable benchmarking results.

Chapter 5: Results

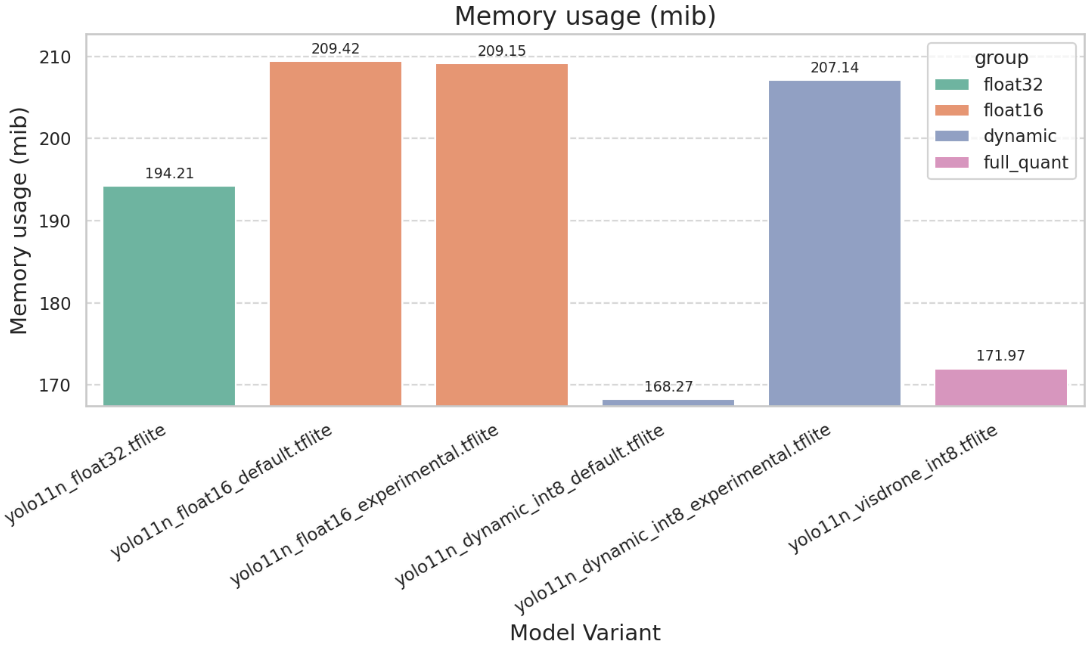


Above are shown the training results of the selected model with the specific training dataset (VisDrone), showing a great mAP value using the test dataset.



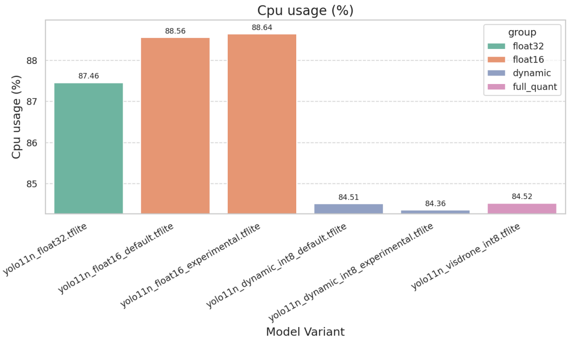
Inference Time: The float32 base model with no quantization acts as the baseline and achieves an inference speed of ~131 ms, while float16 slightly improves latency in the default version (~125 ms) but shows no gain in the experimental version. The Int8 models, both the dynamic one and the fully quantized, are significantly slower on

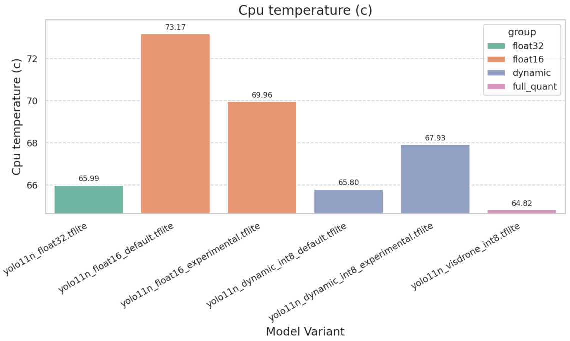
CPU, with inference times around 270–280 ms, most likely as most CPUs are not optimized natively for 8-bit integer operations and require extra instructions, which increases computation overhead



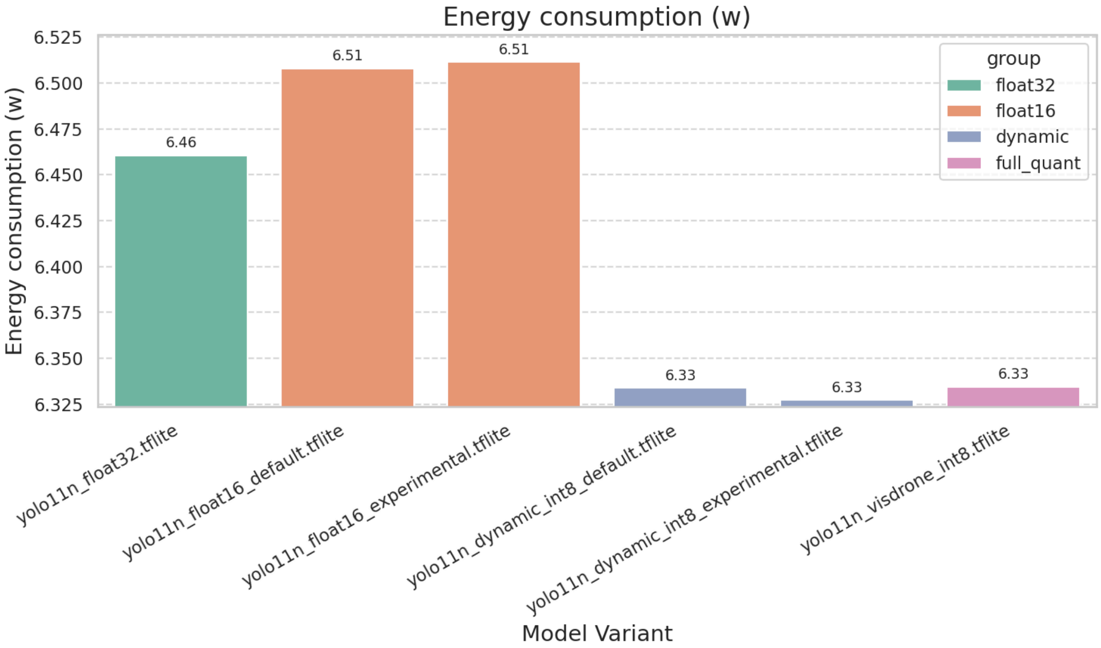
Memory Usage: Int8 quantization provides the most memory reduction with dynamic and full int8 quantized models using 168 and 172 MiB, compared to 194 MiB for float32. The experimental setting actually increases the memory usage to 207MiB for dynamic int8, probably due to missing optimization. Float16 actually increases memory usage slightly (~209 MiB) due to alignment and padding, and therefore offers no benefit in footprint.

Model Size: The models differ quite much in size depending on quantization. The float32 baseline is the largest, with around 10.2 MB of memory. Float16 reduces the size as expected roughly by half to 5.1 MB, while dynamic int8 achieves the smallest footprint at just 2.8 MB (~ x0.25). This shows that int8 provides the most efficient memory usage, which can be very useful for deployment on memory-constrained devices.





CPU Usage and Temperature: Float16 quantization actually increases CPU load and temperature. It is reaching up to 88.6% usage and 73°C, where the int8 models reduce CPU usage slightly to ~84.5% and also remain cooler (~65–68°C). The base Float32 sits in the middle and balances usage and thermal output.



Energy Consumption: Int8 models consume slightly less energy (~6.33 W) than float32 (6.46 W), reflecting lower CPU load, while float16 consumes slightly more (6.51 W), corresponding with its higher CPU activity.

Overall Trade-offs: Choosing the quantization definitely results in a trade-off between memory, speed, energy, and accuracy. Using float16 maintains accuracy close to the baseline f32 model and actually improves inference speed in some settings. But it also increases CPU load, temperature, and therefore also energy consumption, as there is overhead because of missing native f16 support on the hardware. Float16 models are about 5.1 MB in size, so roughly half of the float32 model.

Int8 quantization, on the other hand, achieves the largest reductions in the model size and the memory footprint down to only 2.8 MB, which also results in low power and CPU load. The tradeoff here is slow inference times as the CPUs are not fully optimized for 8-bit deep learning operations and require additional instructions or memory handling. Additionally, the aggressive int8 quantization introduces accuracy degradation.

The baseline float32 model provides a stable and balanced performance without optimizations, as the hardware is native. But it also has the highest model size with 10.2 mb, meaning it is not suited for low memory deployment.

Ultimately, float16 quantized models are best suited for applications prioritizing higher accuracy with slight speed gains, whereas int8 is preferable for memory- or energy-constrained environments, provided that the hardware can efficiently handle 8-bit operations.

Model name	Inference (ms)	Size (MB)	Memory (MB)	CPU temp (C°)	CPU usage %	Energy (W)
yolo11n_float32.tflite	131	10.4	194.21	65.99	87.46	6.46
yolo11n_dynamic_int8_default.tflite	281	2.9	168.27	65.80	84.51	6.33
yolo11n_visdrone_int8.tflite	268	2.9	171.97	64.82	84.52	6.33
yolo11n_float16_exp.tflite	131	5.3	209.15	69.96	88.64	6.51
yolo11n_float16_default.tflite	125	5.3	209.42	73.17	88.56	6.51
yolo11n_dynamic_int8_exp.tflite	279	2.9	207.14	67.93	84.36	6.33

Chapter 6: Discussion

Research Questions

RQ1: Which lightweight person detection model provides the best balance between accuracy, speed, and power efficiency for Edge UAV deployment?

Yolo11n model provides the best overall performance, featuring the best accuracy and quite good speed and power efficiency. While it can reach a sufficient real-time inference on small PCs like the Raspberry Pi 5, it is not meant for embedded systems (like Arduino or ESP32). A solution for that could be the FOMO model from Edge Impulse, but it is closed source, and not enough training has been performed with that due to the free plan account limitations.

RQ2: How can quantization techniques from our lecture be applied to reduce model size and energy consumption without significantly losing detection accuracy?

TensorFlow Lite includes different optimization options that can be set for conversion. Two of them were `OPTIMIZE_FOR_SIZE` and `OPTIMIZE_FOR_LATENCY`, which were meant to focus on a smaller model size or faster inference time. To answer this question, the logical choice would have been `OPTIMIZE_FOR_LATENCY`.

However, during the project, it has been found that these options no longer make a difference: both `OPTIMIZE_FOR_SIZE` and `OPTIMIZE_FOR_LATENCY` have been deprecated and now behave the same as the `DEFAULT` setting. This was a key finding, discovered when no difference was found in model size and benchmark results from these different settings. Further research confirmed this, as it is stated in the TensorFlow documentation. This finding is very valuable and can serve as a warning for others working in similar projects.

RQ3: How do different edge hardware platforms affect model performance, inference speed, and power draw?

To run a real-time computer vision person-detection algorithm, it is a good idea to start directly with small PCs as edge hardware, skipping the microcontrollers since they cannot support such an intensive task with good accuracy or sufficient speed. For this project, two main hardware platforms have been tested: the Raspberry Pi 4B and Raspberry Pi 5. Even though they are only one generation apart, the performance difference is quite significant, since the Pi 5 has been developed with AI applications in mind: almost a 10x inference speed has been measured, with the same accuracy and similar power draw.

To further improve the algorithm's performance, specific 8-bit AI accelerators can be implemented to exploit the INT8 quantization, like, for example, Hailo accelerators (“AI Accelerator Hailo-8 For Edge Devices”) or Axelera Metis (“Metis M2 AI acceleration card”).

A significant upgrade could be to switch to the NVIDIA Jetson platform: this would involve a higher energy consumption, but would feature a great performance increase, leveraging the GPU capabilities. This hardware has not been tested due to the lack of all necessary components.

RQ4: What is the best combination of model and hardware for a solar-powered UAV performing real-time search and rescue operations?

To answer this question, it is necessary to give a little context on UAV types used for SaR operations:

- Small/large multirotors, the most common type, ideal for hovering and covering small areas.
- Small fixed-wing, can cover distances fast, and are used to cover large areas.
- VTOL or hybrid VTOL, a combination of a quadcopter and a fixed wing.
- HALE platforms, specialized for long endurance and solar power.

The area that can be dedicated to solar panels can range from $0.1\,m^2$ for small quadcopters, to $1\,m^2$ for small fixed-wing, to $10\,m^2$ for large UAVs designed to be solar-powered. If we consider a power generation of around $1000\,W/m^2$, a solar efficiency of ~20% and a realistic daytime average of ~50%, we get:

UAV type	Power draw	Power generated	Available for AI
Small quadcopter	~200 W	~10 W	Nothing
Medium quadcopter	~500 W	~50 W	Nothing
Small fixed-wing	~100 W	~100 W	Almost nothing
Large solar UAV	~300 W	~1000 W	~500-700 W

From the highly simplified data above, it is possible to see that a solar-powered UAV is able to sustain itself only if it is specifically designed for that. In that case, a large amount of power is available for the onboard hardware.

However, if the solar panels are considered only for powering the onboard hardware, the Raspberry Pi 5 chosen for this project and the YOLO11n model have a power consumption that stays way below the power generated by the four example configurations listed above. Based on those results, the best combination will be the Yolo11n model with float16 quantization.

Evaluation of Project Goals

The project successfully achieved its main objectives: A lightweight person detection system capable of real-time inference on edge hardware was developed and benchmarked for UAV search and rescue footage. The YOLO11n model had the best balance between accuracy, inference speed, and power consumption among all tested models (RQ1). Quantization using TensorFlow Lite significantly reduced model size and memory usage, while maintaining a similar detection accuracy (RQ2).

The Raspberry Pi 5 testing proved that edge inference is viable for real-time detection, but performance remains dependent on the edge hardware (RQ3). A broader selection of hardware would have let this research produce better comparisons and recommendations about which edge device to choose for which kind of UAV or Search and Rescue mission.

Finally, a suitable optimized model and hardware combination for solar UAVs has been selected (RQ4).

Conclusion and Future Work

This research project showed that person detection on UAVs can be achieved by deploying optimized lightweight models on edge devices. YOLO11n has been selected as the computer vision model, which offers an effective tradeoff between performance and power efficiency for real-time detection tasks, when combined with TensorFlow Lite quantization. Edge computing reduces latency, conserves bandwidth and energy, and increases UAV autonomy, proving that local inference is a valid solution for search and rescue scenarios.

Future work should focus on the real world UAV deployment to confirm these findings under realistic environments and in motion conditions. The integration of additional sensors, such as thermal or infrared cameras, could improve reliability in low visibility environments and could use sensor fusion for better results. Adaptive model switching could allow UAVs to balance accuracy and energy consumption, using faster and smaller models for general scanning and switching to higher precision models to confirm the detection of what might be a person.

Further optimization could be implemented, like inference frequency and frame rate adjustments based on power availability, increasing operational time even more. Finally, a broader benchmark using specialized hardware such as the Coral TPU, Jetson Nano, or Hailo accelerators would lead to a better understanding of possible performance and provide more recommendations for the design of UAV systems for search and rescue missions, both battery powered and solar powered.

References

GitHub project repository: https://github.com/JakubSchwenkbeck/low-power-person-detection-uav-sar

“AI Accelerator Hailo-8 For Edge Devices.” n.d. Hailo AI. Accessed October 27, 2025. https://hailo.ai/products/ai-accelerators/hailo-8-ai-accelerator/.

“Build fast, sparse on-device models with the new TF MOT Pruning API.” n.d. https://blog.tensorflow.org/2021/07/build-fast-sparse-on-device-models-with-tf-mot-pruning-api.html#Compression%20and%20Latency%20Improvements.

“C2A Dataset: Human Detection in Disaster Scenarios.” n.d. Kaggle. https://www.kaggle.com/datasets/rgbnihal/c2a-dataset.

“COCO dataset.” n.d. COCO - Common Objects in Context. https://cocodataset.org/.

“DetReIDX.” n.d. DetReIDX: A Stress-Test Dataset for Real-World UAV-Based Person Recognition. https://www.it.ubi.pt/DetReIDX/.

“FOMO.” n.d. Edge Impulse Documentation. https://docs.edgeimpulse.com/studio/projects/learning-blocks/blocks/object-detection/fomo.

“Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy.” n.d. https://arxiv.org/pdf/2103.03014.

“Manipal UAV Dataset.” n.d. https://github.com/Akshathakrbhat/Manipal-UAV-Person-Dataset.

“Metis M2 AI acceleration card.” n.d. Accessed October 27, 2025.

“MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers.” n.d. https://proceedings.mlsys.org/paper_files/paper/2021/file/c4d41d9619462c534b7b61d1f772385e-Paper.pdf.

“Python time library.” n.d. https://docs.python.org/3/library/time.html.

“Raspberry Pi 5 power consumption.” n.d. https://www.tomshardware.com/reviews/raspberry-pi-5.

“SeaDronesSee dataset.” n.d. https://seadronessee.cs.uni-tuebingen.de/dataset.

“A Survey of Quantization Methods for Efficient Neural Network Inference.” n.d. https://arxiv.org/abs/2103.13630.

“tf.lite.Optimize.” n.d. https://www.tensorflow.org/api_docs/python/tf/lite/Optimize.

“Ultralytics YOLO Docs.” n.d. Ultralytics YOLO Docs: Home. https://docs.ultralytics.com/.

“VisDrone Dataset.” n.d. Ultralytics YOLO Docs. https://docs.ultralytics.com/datasets/detect/visdrone/.