Linnæus University
Sweden

Project with Embedded System

Remote Intruder Detection

- *Low-Cost Real-Time Person Segmentation Using Raspberry Pi and CNN*

Author: Michael Daun & Christopher Sandberg
Examiner: Hemant Ghayvat
Term: VT25
Course code: 2DT304

Table of contents

1 Introduction

The use of real-time video detection has grown significantly in applications ranging from security and surveillance to traffic monitoring and industrial automation [1][2]. Machine learning-based methods, such as object detection and image segmentation, play an increasingly central role in analyzing video streams. However, many existing solutions require high-end hardware or cloud processing, making them costly and resource intensive. This raises the question of whether lightweight embedded systems can achieve effective real-time detection at a lower cost.

One area where this challenge is particularly relevant is home security. In 2018, an article published in Svenska Dagbladet reported that 500,000 households in Sweden had installed a burglar alarm. At the time, the market for alarm systems was expected to grow further, and security companies anticipated a bright future for their industry [3]. However, data suggests that burglar alarms are often unreliable, expensive, and sometimes even counterproductive [4].

1.1 Background

Home alarm systems are more popular than ever, with an increasing number of households in Sweden installing alarms in the hope of deterring intrusions and, in the worst case, quickly alerting the police if an intruder breaks in. The market for burglar alarms continues to grow, generating increasing revenue [3]. However, whether these alarms truly benefit consumers is not entirely clear. According to a report from the *European Crime Prevention Network* (EUCPN) [4], there is no data supporting the claim that alarms have a deterrent effect. On the contrary, modern alarm systems may signal to burglars that a household contains valuable items, making it a more attractive target instead.

Furthermore, the alarm market is largely dominated by a few major players, whose standardized commercial systems create an incentive for burglars to learn how they work and how to bypass them undetected. Another common criticism is that these systems are not cost-effective and trigger false alarms so frequently that police and security companies often do not respond immediately to incoming alerts [4].

Sweden's largest alarm provider, *Securitas*, offers a package that includes a motion detector and a Wi-Fi camera. In addition to a one-time installation fee, Securitas charges a monthly subscription fee. They also offer video analysis, which can be activated upon motion detection and provides object recognition capabilities—available at an additional monthly cost.

In contrast to the often costly surveillance systems provided by major security firms like Securitas, open-source solutions, such as those offered by Home Assistant [5], provide a more affordable alternative. The Home Assistant system can serve as a cost-effective home surveillance solution using your own hardware. Its surveillance system relies on object detection, which identifies objects using bounding boxes.

1.2 Problem Definition

Surveillance and security systems increasingly rely on real-time person detection to enhance safety in public and private spaces. However, many existing solutions are expensive, hardware-intensive, and primarily use object detection with bounding boxes. By implementing a Raspberry Pi-based system with a U-Net segmentation model, we aim to explore the feasibility of a low-cost, embedded solution for person detection using image segmentation instead of object detection.

Unlike bounding boxes, image segmentation produces highly detailed masks, which can be more useful in certain applications and provide a more intuitive approach to tracking individuals in video feeds. This could lead to improved accuracy and efficiency in scenarios where precise person localization is required.

1.3 Research Questions

To what extent can a low-cost embedded system achieve sufficient accuracy and precision for real-time person detection using image segmentation, given a limited training dataset?

2 Related Work

2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of neural network primarily used for image processing, but it is also well-suited for any type of data that has a grid-like structure [6]. CNNs contain convolutional layers that process data by applying mathematical operations known as convolutions. These convolutions allow the network to detect patterns and features within an image. The output of each convolution is called a feature map, and each convolutional filter generates a separate channel that serves as input for the next layer in the model. By applying multiple filters at different scales, CNNs can detect and learn features at varying levels of abstraction [6].

In contrast to traditional fully connected layers, where each neuron has a unique weight for every input parameter, CNNs leverage *sparse interactions* and *parameter sharing*. This means that each neuron is connected only to a small region of the input rather than every input pixel. Additionally, the same set of filter parameters is applied across the entire image, enabling the network to efficiently recognize patterns regardless of their location [6].

As a result, CNN filters become specialized in detecting specific features in an image, such as edges, textures, or shapes, and they will activate neurons regardless of where these features appear in the image.

2.2 U-Net

U-Net is a machine learning model based on Convolutional Neural Networks (CNNs) that was first introduced in 2015 in the paper *U-Net: Convolutional Networks for Biomedical Image Segmentation* [7]. The authors aimed to develop a CNN-based segmentation model that required fewer training images while still achieving high accuracy. The proposed model was

specifically designed for biomedical applications and was trained on a dataset of only 30 images, which were subjected to extensive data augmentation. Despite the limited training data, U-Net outperformed competing models in a cell tracking challenge, achieving an Intersection over Union (IoU) accuracy of 92% [7].

The main innovation of U-Net was the extensive use of feature channels combined with skip connections, where high-resolution features from the encoder were directly transferred to the decoder. This preserved spatial information during upscaling, enabling more precise segmentation outputs with finer details.

2.3 Previous Research on Person Detection and Segmentation in Surveillance Systems

Image segmentation in surveillance video is much less common than object detection. Object detection is computationally lighter since it only needs to produce a bounding box, which can be rendered in a video feed using just a few coordinates. In contrast, image segmentation must classify every pixel in an image, making it more computationally demanding and less ideal for real-time video feeds.

However, one study closely aligns with our goals. *Human Segmentation in Surveillance Video with Deep Learning* [1] explores the use of image segmentation to distinguish people, labeled as the foreground, from the background. This study used SegNet, a deep neural network based on a convolutional neural network (CNN) architecture. Their dataset consisted of 16,832 training images, created using 35 different actors performing various movements in front of a green screen. The extracted silhouettes of the actors

were later combined with background images captured from surveillance cameras.

The study concluded, among other findings, that applying a nonlinear filter improved segmentation accuracy. This filtering step helped mitigate issues where the model struggled to differentiate the actor's silhouette from the pasted background. The model proposed by the authors achieved 99.79% validation accuracy. Using a Nvidia Titan GPU, they were able to process 640×360 images in 0.06 seconds, making it feasible for real-time video applications.

Another study [2] that attempted to utilize image segmentation for analyzing surveillance footage is a Taiwanese research effort aimed at developing a model that classifies the clothing of people captured by security cameras. The authors summarize the benefits of using segmentation techniques as follows: image segmentation provides pixel-perfect precision and, unlike bounding boxes, does not include clutter or noise around the target. This allows for more accurate color detection and recognition of smaller details. The authors aimed to create a system that can track individuals across different cameras using instance segmentation. This means that the segmentation mask highlights various types of clothing and accessories along with their corresponding attributes. For example, an individual could be categorized as having long brown hair, brown sneakers, blue jeans, and so on.

The system is based on two main models. The first is the YOLACT++ model, which detects and segments each person in the frame, isolating them by removing the background. A separate multi-CNN classifier is then used to analyze clothing attributes. The authors of [2] claim that they successfully developed a system fast enough for real-time usage, which also preserves motion patterns of individuals in the video feed.

3 Hardware

The hardware used in this project was primarily chosen based on availability. While hardware is crucial for training the CNN model, once training is complete, the specific hardware used for inference is less critical. Any component we used could be replaced with an equivalent alternative without significantly impacting the project's functionality. Below, we briefly describe each hardware component, its specifications, and its role in the project.

3.1 Raspberry Pi 2 B

The Raspberry Pi 2 Model B served as the primary logic board, responsible for controlling the camera remotely and hosting the web server that facilitates communication with the system and displays its output. Originally released in 2015, the Raspberry Pi 2 B is equipped with a 900 MHz quad-core 32-bit ARM Cortex-A7 processor and 1 GB of RAM [8]. The operating system, Raspberry Pi OS, is stored on a microSD card. Additionally, the board includes USB ports, which we used to connect a wireless network adapter for remote communication.

The Raspberry Pi was powered using a 5V 2.5A USB power supply. Initially, we encountered low voltage warnings from the operating system, indicating power instability. We mitigated this issue by disabling the graphical user interface (GUI) and communicating with the system exclusively via SSH (Secure Shell Protocol), reducing power consumption and ensuring stable operation.

3.2 Raspberry Pi Camera Module 2 NoIR

Introduced in 2016, this camera module is specifically designed for the Raspberry Pi and connects via its built-in camera interface. The module lacks

an IR filter, making it specialized for capturing images in low-light conditions when used with infrared illumination [9].

3.3 MacBook Air M1

We used a 2020 MacBook Air M1 with 16 GB RAM as our primary hardware for training our CNN models. The M1 is an ARM-based processor with a System-on-a-Chip (SoC) design, meaning that both the CPU and GPU share the same unified memory structure [8]. The TensorFlow library we used for training our models supported GPU acceleration on the M1, which sped up the training process.

However, the M1 is not ideal for machine learning due to its limited GPU. Depending on the CNN architecture and dataset size, training sessions with the MacBook ranged from 20 minutes to 20 hours.

4 Software

This chapter describes the software components used throughout the project. The software was chosen based on development convenience, past experiences and support with our hardware.

4.1 Machine Learning

4.1.1 Python

Python was chosen due to its dominance in the TensorFlow ecosystem and its flexibility across different operating systems. Another benefit of Python is its extensive collection of libraries for data analysis and manipulation. Libraries such as Numpy and Matplotlib have been essential for processing and visualizing throughout our projects.

4.1.2 TensorFlow/Keras

TensorFlow is a widely used open-source framework used for machine learning and was originally created by Google 2015 [11]. Keras is an open-source Python-based API for creating various types of neural networks [12]. We primarily used TensorFlow/Keras to build and train our models.

4.1.3 PIL/Pillow

PIL (Python Imaging Library) is a library for opening, manipulating, and processing images in Python. First released 1995 as PIL and reworked with additional functionality 2011 under the new name Pillow [13]. We used Pillow for image preprocessing tasks, including loading images, resizing, cropping, and padding, to ensure they met the required input specifications before being processed by our neural network.

4.2 Raspberry Pi

4.2.1 Picamera

Picamera is a library created to make the usage of the camera module on the Raspberry Pi easy. We use this library to communicate with the camera.

4.3 Back-End Services

4.3.1 Flask

Flask is a lightweight web framework written in Python. It does not depend on any third-party libraries or tools, making it well suited for our Raspberry Pi with its limited computational resources. We used Flask to host the front-end interface on the Raspberry Pi, which both triggers the camera and displays the final output. Additionally, we ran a separate Flask server on our MacBook that received images from the Raspberry Pi, processed them using our trained model, and sent the results back.

5 Methodology

5.1 Compiling Datasets

To train a machine learning model for our specific needs, it was crucial to use data that closely resembled our expected environments. Specifically, we aimed to train the model on video surveillance data whenever possible. Therefore, we compiled a dataset consisting of indoor images of people. Each image in the dataset had a corresponding mask image.

Image 1. Image with its mask.

Each mask was a completely black image with a hand-drawn white silhouette of the person in the corresponding image. The masks were created so that, when overlaid on the original image, the silhouette would perfectly align with the person. Many image-mask pairs had to be constructed to ensure sufficient training data. In total, we compiled 216 pairs of images and masks.

We later searched for additional data to complement our dataset. We found several datasets on platforms such as Kaggle.com and Huggingface.co. In total, we gathered approximately 15,000 images of varying quality to combine with our own.

5.2 Preprocessing

A common technique for improving model performance during training is data augmentation. In our case, this meant generating modified copies of each image/mask pair to artificially expand our small dataset of only 216 pairs. Various augmentation techniques were applied, including random scaling, mirroring, panning, contrast adjustments, rotation, and noise injection. However, noise was never applied to the mask to preserve its accuracy.

The images were resized to 256 × 256 pixels, and their color values were normalized, with black pixels set to 0 and white pixels to 1. Since the downloaded images varied in format and aspect ratio, a black border was added to maintain uniformity.

Image 2. Artificially Distorted Image.

5.3 Building The Model

5.3.1 Structure and Layers

The first model was loosely based on the original U-Net [7] but simplified to reduce computational demands. We omitted some of the fine-grained details in the bottleneck layer to optimize performance. The model featured two encoding layers with max pooling, progressively downscaling the input

image from 256×256 → 128×128 → 64×64. The upscaling phase followed symmetrically, incorporating skip connections to retain important spatial features from the encoding phase.

The second model was more complex and ambitious in its design. It included seven symmetrical layers, reducing the image size through max pooling down to 8×8 pixels in the final layer, where it generated 2048 feature maps. In addition to this, batch normalization was applied to each encoding and decoding layer to help stabilize training by normalizing the input distribution at each step. Dropout layers were also introduced throughout the model to reduce overfitting by randomly dropping neurons during training, forcing the network to generalize better.

5.3.2 Hyperparameters

We used Adam (Adaptive Moment Estimation) as our optimizer, as it is well-suited for large datasets. Adam dynamically adapts the learning rate throughout the training process, making it effective for optimizing deep learning models.

For the loss function, we initially used the standard binary cross-entropy built into Keras. Later, we implemented custom loss functions, incorporating Dice loss and Intersection over Union (Jaccard loss) to better evaluate segmentation accuracy.

Image 3. Intersection Over Union

The batch size was set to 16 images that were processed through the network per iteration. Additionally, we allocated 20% of the total dataset as a validation set during training to monitor how well the model would generalize to new data.

5.4 Incorporating model with Raspberry Pi

Since our goal was to develop a real-time surveillance system, we needed a solution that would allow the Raspberry Pi to be triggered remotely. Due to the limited computational resources of the embedded system, we designed the Raspberry Pi's role to be as lightweight as possible. Its main responsibility was to capture images and send them to a backend server for processing. The results were then displayed on an HTTP page hosted by the Raspberry Pi.

To handle image processing, we set up a Flask server on a separate machine. This server listened to requests from the Raspberry Pi, processed the incoming images using the trained model, and returned the predictions. The processed results were then sent back to the Raspberry Pi for display on the HTTP page.

6 Results and Discussion

In this chapter, we discuss the results of our project. While machine learning problems often rely heavily on numerical data and statistical evaluation, we have chosen to focus primarily on qualitative observations and comparative image outputs. By analyzing visual results from different phases of the project, we aim to provide a clear understanding of how our model performs in practical scenarios. This section will therefore include various images from different stages of the project to illustrate our findings.

To ensure a consistent basis for comparison, we created a dedicated test set consisting of seven images from our own photo session, two images captured with the Raspberry Pi camera in poor lighting conditions in a new environment, and a couple additional images randomly selected from Kaggle.com. This test set is used throughout this section to evaluate and illustrate our model's performance.

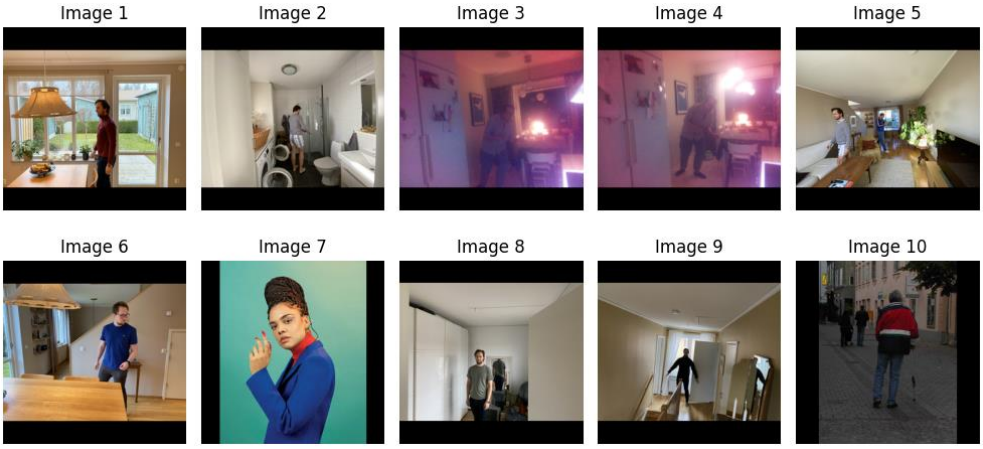

Image 4. Image test set.

6.1 Model Structure

We started with a simple model that had few layers and a low number of feature maps. While this model was fast to train, the results were unsatisfactory.

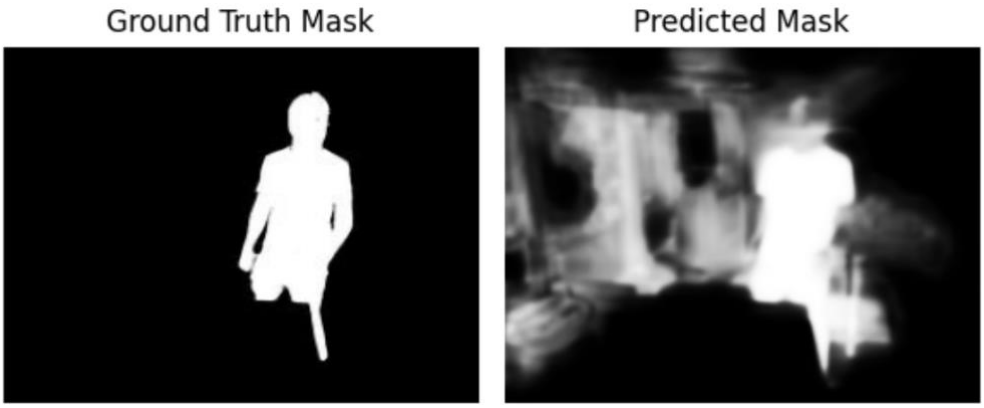

Image 5: Initial U-Net Model

By comparing the results with a more complex model, we can see that the simpler model in Image 5 lacks the high-level detail captured by the more advanced model shown in Image 6.

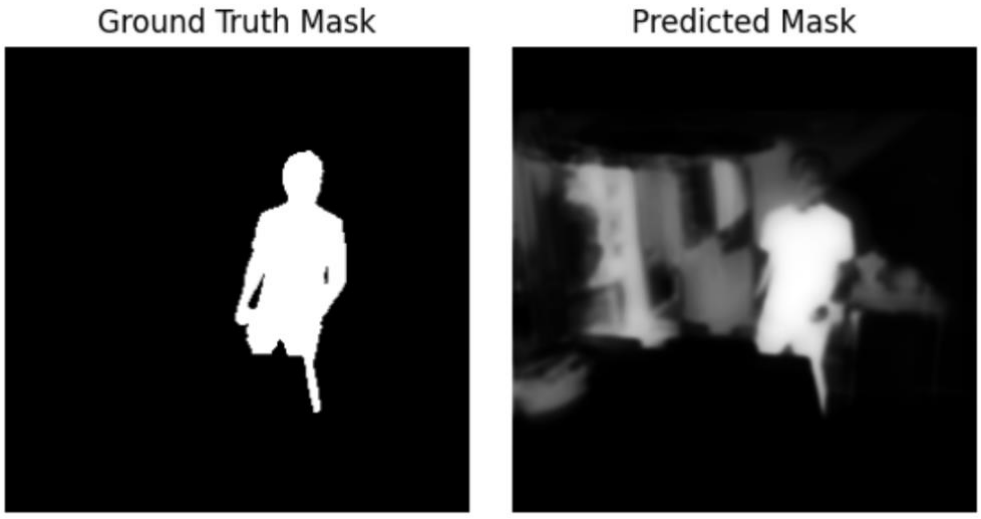

Image 6. Upgraded U-Net Model

6.2 Loss functions

The choice of loss function used to train the model had a significant impact on the results. Comparing Image 7 and Image 8, we can observe the difference between the Dice loss function and binary cross-entropy (BCE). The BCE loss consistently produced a “ghostly glow” around the target and appeared to be more generous in its interpretation of what should be included in the segmentation. Although the Dice loss did not always perform significantly better, it showed fewer issues with including pixels from inanimate objects. Similarly, the Intersection over Union loss (Jaccard loss) produced segmentation masks comparable to those from the Dice loss.

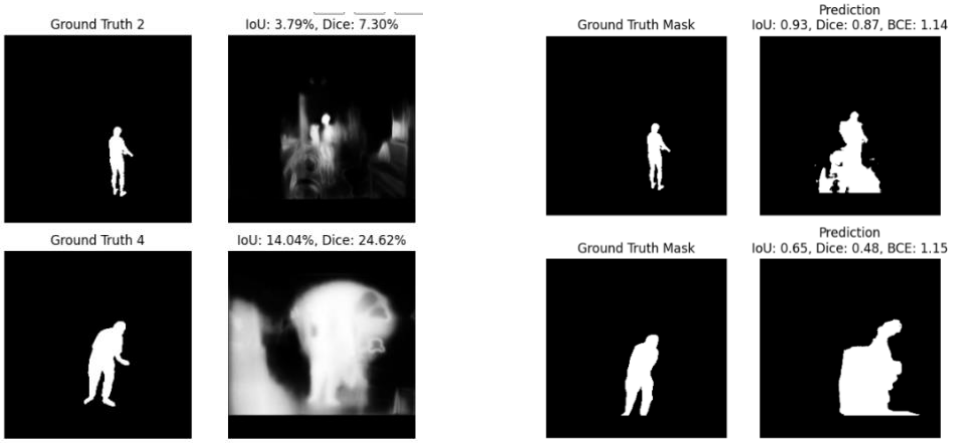

Image 7 (left) & Image 8 (right). Left shows model trained with Binary Cross-Entropy while the right model uses Dice loss.

The Jaccard loss is mathematically stricter in its interpretation of the results, so the statistics shown in Image 9 should not come as a surprise. In theory, a stricter loss function could be beneficial for performance. However, we did

not achieve any significantly better results when using Jaccard loss as our main loss function.

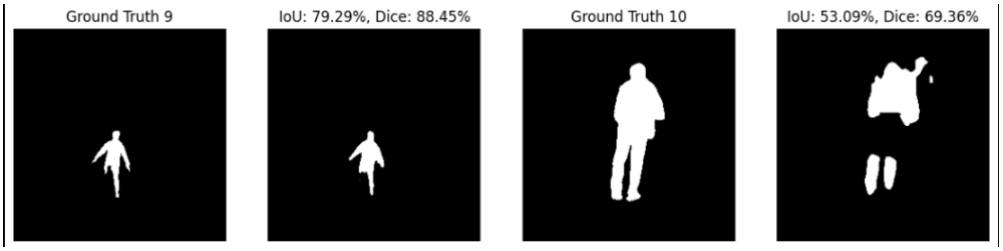

Image 9. Image displaying difference in Jaccard and Dice metrics.

6.3 Size of Training set

As we set out to develop a surveillance system using a limited set of images, we experimented with tuning hyperparameters and adjusting the model configuration to achieve the best possible results with our constrained dataset. Even after applying data augmentation with two different levels of intensity, we were unable to reach a satisfactory outcome. As seen in Image 10, the model sometimes demonstrates a basic understanding of what it should be detecting and highlighting in the output.

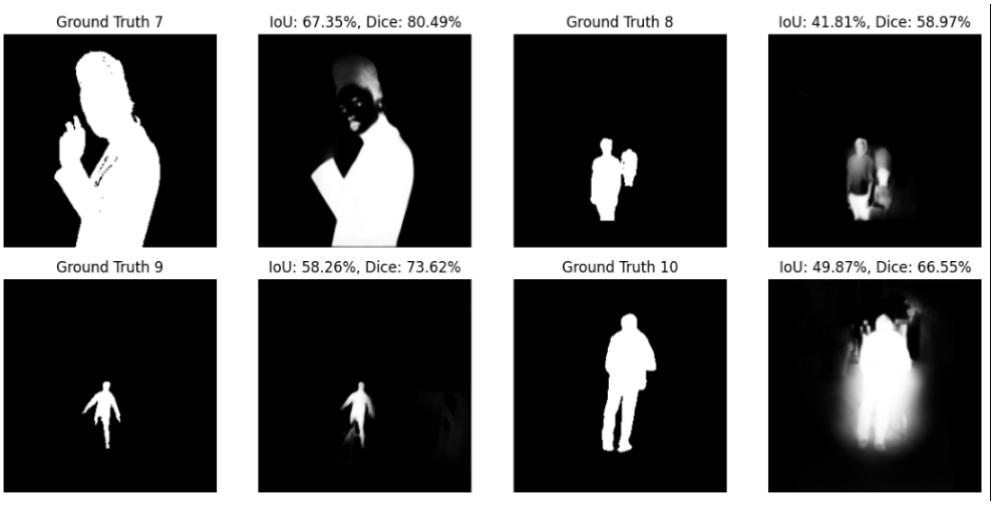

Image 10. 600 images model with binary cross-entropy. 200 images original and 400 augmented versions.

When tasked with predicting images taken by the Raspberry Pi camera, our model was unable to make accurate predictions using the models trained on our own dataset.

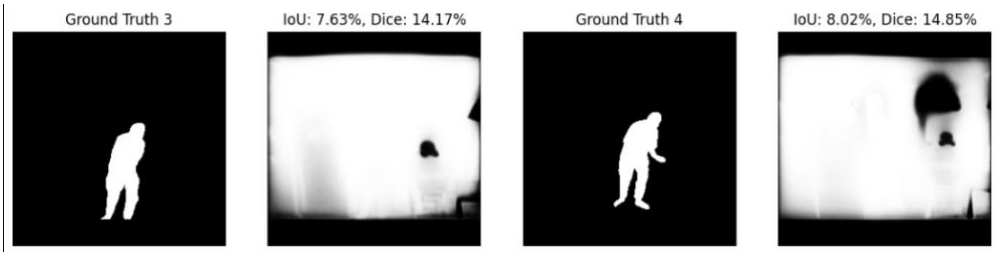

Image 11. Same model as previous image but worse predictions generated.

As we found our models with our own datasets having a hard time generalizing to unseen images, we tried adding images from datasets we found online.

In Image 12, we see the result from a model trained on 3,000 original images, augmented to a total of 6,000 images. The results varied greatly, while the model occasionally produced decent outputs, none of the models consistently delivered acceptable results across all unseen test images.

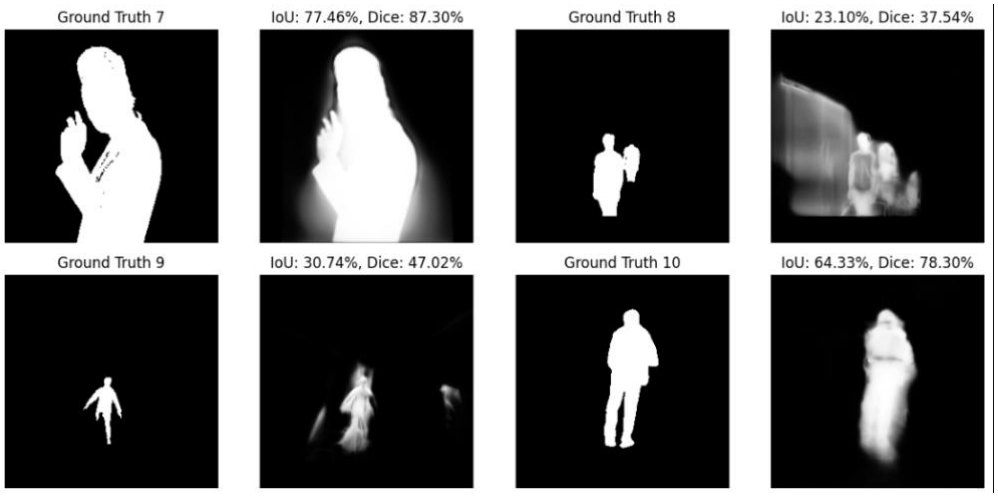

Image 12. Result model with 6000 images, same hyperparameters as model in image 10.

In our final model, we used all the images collected from online datasets together with our own. The total dataset consisted of 15,000 original images and an additional 15,000 augmented copies. Despite the larger dataset, the

model's performance on unseen still images remained unsatisfactory, with outputs often appearing chaotic. However, in a continuous video stream, the results were more acceptable, as poor individual frames blended into the sequence and were less noticeable in context.

Image 13. The left image shows a good frame from the video stream, while the right image displays a poor frame.

6.4 Surveillance system

With our two Flask servers set up, one on the Raspberry Pi and one on the MacBook, we began sending images between them and measuring the time taken from image capture to processing, prediction, and return to the Raspberry Pi. Our initial tests showed an average time of 2 seconds from capture to return.

We managed to reduce this time by adjusting the camera configuration, changing the capture resolution from the full 3280 × 2464 to 1024 × 1024. This reduced both the amount of data being sent and the need for extensive preprocessing, as the new format was closer to what our model expected.

With this adjustment, the average processing time per frame dropped to 450 ms. While it could still be perceived as a video feed with high latency, there is clear room for optimization in the overall system flow.

```
Image taken and processed in: 0.435 seconds
192.168.0.153 - - [23/Mar/2025 08:43:37] "GET /analyze HTTP/1.1" 200 -
Image taken and processed in: 0.417 seconds
192.168.0.153 - - [23/Mar/2025 08:43:38] "GET /analyze HTTP/1.1" 200 -
Image taken and processed in: 0.451 seconds
192.168.0.153 - - [23/Mar/2025 08:43:38] "GET /analyze HTTP/1.1" 200 -
Image taken and processed in: 0.45 seconds
192.168.0.153 - - [23/Mar/2025 08:43:39] "GET /analyze HTTP/1.1" 200 -
Image taken and processed in: 0.434 seconds
192.168.0.153 - - [23/Mar/2025 08:43:39] "GET /analyze HTTP/1.1" 200 -
Image taken and processed in: 0.514 seconds
192.168.0.153 - - [23/Mar/2025 08:43:40] "GET /analyze HTTP/1.1" 200 -
```

Image 14. Time elapsed from image capture on the Raspberry Pi to receiving the processed result.

7 Conclusion

In this project, we explored the feasibility of using a low-cost embedded system for real-time intruder detection through image segmentation. By implementing a U-Net-based convolutional neural network (CNN) on a Raspberry Pi and processing images remotely via a Flask server, we aimed to create an affordable alternative and light weight solution to commercial security solutions.

Our results indicate that while lightweight embedded systems like the Raspberry Pi can be a key component in a real-time surveillance architecture, several challenges remain. The primary limitation was the model's ability to generalize across diverse environments, particularly when working with a small, self-compiled dataset. Augmenting our dataset and integrating additional training images from online sources helped improve performance, but the model still struggled with unseen test images. Gathering third party data applicable for our subject matter deemed to be somewhat challenging.

Loss function selection played a crucial role in segmentation accuracy. Dice loss and Intersection over Union provided more precise segmentations compared to standard binary cross-entropy, reducing false positives and improving the model's ability to isolate human figures. However, even with optimized hyperparameters and extensive dataset augmentation, the model did not consistently deliver reliable predictions on individual frames.

Despite these limitations, the system demonstrated reasonable performance in a continuous video stream, where inconsistencies in individual frames were less noticeable. By reducing image resolution and optimizing the communication flow between the Raspberry Pi and the Flask server, we achieved an inference time of approximately 450ms per frame—sufficient for a low-latency surveillance system.

Future improvements could include more extensive dataset collection, transfer learning with pre-trained models, and further optimization of the model architecture. Additionally, integrating edge computing techniques directly on the Raspberry Pi, rather than relying on remote processing, could enhance real-time performance. Additionally, to minimize the perceived effect of rogue frames there could be post processing done to multiple frames to reduce the few poor predicted frames.

Overall, this project demonstrates the potential of low-cost embedded systems for machine learning-based surveillance the opportunities and challenges in deploying CNN-based segmentation models in resource-constrained environments.

8 References

[1]: Gruosso, Monica, et al. “Human Segmentation in Surveillance Video with Deep Learning.” Multimedia Tools and Applications, vol. 80, no. 1, 2021, pp. 1175–99, https://doi.org/10.1007/s11042-020-09425-0.

[2]: Tseng, Chien-Hao, et al. “Person Retrieval in Video Surveillance Using Deep Learning–Based Instance Segmentation.” Journal of Sensors, vol. 2021, no. 1, 2021, https://doi.org/10.1155/2021/9566628.

[3]: J. Edvardsson, “Rädslan för inbrott guld för larmbolagen,” *Svenska Dagbladet*, Sep. 16, 2018. [Online]. Available: https://www.svd.se/a/J1o1PJ/radslan-for-inbrott-guld-for-larmbolagen

[4]: European Crime Prevention Network (EUCPN), *The Prevention of Domestic Burglary*, Brussels: EUCPN Secretariat, 2021. [Online]. Available: https://eucpn.org/sites/default/files/document/files/2106_The%20prevention%20of%20domestic%20burglary_LR.pdf

[5]: Home Assistant, “Home Assistant,” [Online]. Available: https://www.home-assistant.io/

[6]: A. Lindholm, J. Lindholm, and A. Lilliesköld, *Machine Learning: A First Course for Engineers and Scientists*. Cambridge, UK: Cambridge University Press, 2022.

[7]: O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation.” Medical Image Computing and Computer-Assisted Intervention -- MICCAI 2015, vol. 9351, Springer International Publishing AG, 2015, pp. 234–41, https://doi.org/10.1007/978-3-319-24574-4_28.

[8]: “Raspberry Pi 2 Model B Technical Specifications,” Adafruit Industries, [Online]. Available: https://cdn-shop.adafruit.com/pdfs/raspberrypi2modelb.pdf. [Accessed: 03-Mar-2025].

[9]: Raspberry Pi Foundation, “Camera Module v2 – NoIR,” *Raspberry Pi*, [Online]. Available: https://www.raspberrypi.com/products/pi-noir-camera-v2/. [Accessed: 03-Mar-2025].

[10]: Apple Inc., “MacBook Air (M1, 2020) - Technical Specifications,” *Apple Support*, 2024. [Online]. Available: https://support.apple.com/en-us/111883. [Accessed: 03-Mar-2025].

[11]: *TensorFlow*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow. [Accessed: 03-Mar-2025].

[12]: *Keras*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Keras. [Accessed: 03-Mar-2025].

[13]: *Python Imaging Library*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Python_Imaging_Library. [Accessed: 03-Mar-2025].