# Conclusion

The research presented in this thesis explored the development of an advanced human presence detection system using low-resolution thermal sensing and deep learning, moving from model design and evaluation to real-time embedded deployment and performance benchmarking against traditional PIR systems. At the outset, multiple machine learning models were trained and evaluated, including a Multi-Layer Perceptron (MLP) which served as a benchmark. The MLP, although computationally efficient with just 33,000 parameters, achieved only 80% test accuracy and demonstrated poor generalization due to its inability to capture spatial or temporal dependencies in thermal frames. In contrast, convolution-based models; particularly the proposed Temporal CNN with approximately 200,000 parameters achieved a test accuracy of 99%, a validation accuracy of 99.94%, and consistently outperformed the MLP and the larger 1D CNN (which had over 1 million parameters but only 90% test accuracy). These results established the Temporal CNN as the optimal balance between performance, generalization, and deployment feasibility.

A key innovation in this work was the introduction of a background removal concept based on Exponential Moving Average (EMA), which dynamically subtracted static heat sources such as heaters and electronics. During real-time inference, this model adapted only when no human was present, ensuring that heat signatures of occupants were not inadvertently integrated into the background. Empirical results showed that the system required approximately 60 frames (12 seconds at 5 FPS) to fully adapt to a new thermal baseline. This mechanism dramatically improved the robustness of detection, particularly in thermally cluttered environments such as washrooms with plumbing heat or bedrooms with active appliances. After adaptation, thermal contrast improved significantly; yielding high-confidence predictions and reducing false positives to near zero.

The full model was successfully deployed on a Raspberry Pi 4, achieving inference every 200 milliseconds with real-time output through GPIO signaling. Compared to a traditional PIR sensor setup, the thermal system demonstrated a slightly delayed average response (mean lag of 1.73 seconds) but achieved superior reliability in detecting static presence. While PIR sensors failed to sustain detection once motion ceased, the thermal system maintained a consistent signal throughout the entire occupancy period. Performance metrics further supported this: the thermal system achieved an F1 score of 0.90 compared to 0.74 for the PIR sensor, with a recall of 0.81, indicating how reliably the system detected presence when it was truly there, and a precision of 1.0, confirming that it did not raise any false alarms. These results highlight the system's strong balance between sensitivity and specificity, which is crucial for presence detection in real-world, low-motion environments.

In extending the solution to ultra-low-power edge devices, a simplified CNN model was adapted and quantized for deployment on the ESP32-S3 microcontroller. Despite the use of INT8 quantization and layer simplification to meet ESP-DL constraints, the resulting model achieved a validation accuracy of 96.06% and delivered 94% accuracy during live inference with real-time classification from raw thermal frames. This performance was achieved without background subtraction, confirming that the simplified model retained sufficient spatial feature learning to detect presence reliably. The entire firmware consumed only ~600 kB of flash, with less than 20% of IRAM usage, highlighting its efficiency and readiness for battery-powered applications.

In conclusion, this thesis demonstrated that a high-performance, deep learning-based presence detection system using thermal sensors can be realized on embedded platforms. The Raspberry Pi implementation offered a robust, adaptive solution with near-perfect accuracy and dynamic background learning, while the ESP32 deployment validated that deep learning models; when carefully compressed can still deliver real-time inference with only a minor 4–5% reduction in accuracy. Together, these outcomes illustrate the feasibility of deploying thermal presence detection both in powerful edge systems and resource-constrained IoT nodes, offering a privacy-compliant, reliable, and scalable alternative to conventional PIR-based methods.

# Future Work (revised version)

Building on the outcomes of this thesis, several promising directions can extend the capabilities, reliability, and applicability of the proposed system:

* **Expansion of training data and scenarios**: Future efforts should focus on incorporating more diverse datasets, including multiple occupants, occluded subjects, pets, and varying room geometries. Such variation will enhance model generalization and resilience in unconstrained real-world environments.
* **Advanced model compression and optimization**: Although ESP32 deployment was successful with quantized CNNs, further optimization can be pursued using structured pruning, quantization-aware training, and knowledge distillation. These techniques can minimize memory usage and inference latency while preserving accuracy on ultra-constrained edge devices.
* **Multimodal sensor fusion**: Combining thermal sensing with ultrasonic, RF (e.g., Wi-Fi CSI), or PIR sensors could improve detection robustness, especially under occlusion or low contrast conditions. Sensor fusion can also reduce ambiguity in edge cases such as partial presence or overlapping thermal signatures.
* **Integration into smart automation frameworks**: The system can be extended to interact with real-world IoT platforms for dynamic control of HVAC, lighting, or access systems. This will require real-time API interfacing and reliability testing under changing occupancy states.
* **Long-term deployment and calibration studies**: Extended field testing over several weeks or months can reveal potential challenges such as thermal drift, model degradation, and hardware wear. Continuous monitoring can also help assess adaptation needs in seasonally changing environments.
* **Privacy-preserving on-device learning**: As a long-term goal, enabling incremental learning or adaptation on-device without transferring raw thermal data could make the system suitable for GDPR-compliant, personalized use in sensitive spaces like homes or elder care facilities.

In summary, this thesis establishes a robust foundation for thermal sensing-based human presence detection in embedded systems. Through expanded datasets, smarter model adaptation, sensor fusion, and broader system integration, future developments can lead to scalable, intelligent, and privacy-conscious automation in real-world environments.

# Appendix E – Source Code and Implementation Scripts

This appendix includes core portions of the deep learning training pipeline implemented for human presence detection using low-resolution thermal data. The Python script below covers the essential steps: data loading, background modeling, feature extraction, model architecture (CNN + Conv1D), model training, evaluation, and conversion to deployment ready formats. Due to its length, only representative segments are shown. Full source code and data are available upon request or can be accessed via the project repository.

## E.1.1 Data Loading and Preprocessing

import os  
import pandas as pd  
  
def load\_data(folder):  
 return pd.concat([  
 pd.read\_csv(os.path.join(folder, f))  
 for f in os.listdir(folder) if f.endswith('.csv')  
 ], ignore\_index=True)  
  
human\_data = load\_data("Dataset\_all/Human")  
no\_human\_data = load\_data("Dataset\_all/No Human")

## E.1.2 Adaptive Background Model using EMA

import numpy as np  
  
def compute\_background(frames, alpha=0.1):  
 bg\_model = frames[0]  
 for i in range(1, len(frames)):  
 bg\_model = alpha \* frames[i] + (1 - alpha) \* bg\_model  
 return bg\_model

## E.1.3 Background Subtraction Function

def remove\_background(frame, bg\_model):  
 return np.clip(frame - bg\_model, 0, None)

## E.1.4 Model Input Preparation

sequence\_length = 5  
cnn\_seq = []  
feat\_seq = []  
y\_seq = []  
  
for i in range(len(X) - sequence\_length):  
 cnn\_seq.append(X[i:i+sequence\_length].reshape(sequence\_length, 32, 32, 1))  
 feat\_seq.append(features[i:i+sequence\_length])  
 y\_seq.append(labels[i + sequence\_length - 1])

## E.1.5 Feature Extraction (Gradients, Hot Region Variance, Edges)

from scipy.ndimage import sobel  
  
def extract\_all\_features(sample):  
 grid = sample.reshape(32, 32)  
 grad\_x = np.diff(grid, axis=1)  
 grad\_y = np.diff(grid, axis=0)  
 sobel\_x = sobel(grid, axis=0)  
 sobel\_y = sobel(grid, axis=1)  
 edge\_mag = np.hypot(sobel\_x, sobel\_y)  
 hot\_region = grid[grid > grid.mean()]  
  
 return [  
 np.mean(np.abs(grad\_x)), np.var(grad\_x),  
 np.mean(np.abs(grad\_y)), np.var(grad\_y),  
 np.var(hot\_region),  
 np.mean(edge\_mag), np.var(edge\_mag)  
 ]

## E.1.6 Hybrid CNN + Conv1D Model Architecture

from tensorflow.keras.models import Model  
from tensorflow.keras.layers import (  
 Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout,  
 TimeDistributed, Conv1D, GlobalAveragePooling1D, Concatenate  
)  
  
cnn\_input = Input(shape=(5, 32, 32, 1))  
feat\_input = Input(shape=(5, 7))  
  
cnn\_branch = TimeDistributed(Conv2D(16, (3,3), activation='relu'))(cnn\_input)  
cnn\_branch = TimeDistributed(MaxPooling2D((2,2)))(cnn\_branch)  
cnn\_branch = TimeDistributed(Conv2D(32, (3,3), activation='relu'))(cnn\_branch)  
cnn\_branch = TimeDistributed(MaxPooling2D((2,2)))(cnn\_branch)  
cnn\_branch = TimeDistributed(Flatten())(cnn\_branch)  
cnn\_branch = Conv1D(32, 3, activation='relu', padding='same')(cnn\_branch)  
cnn\_branch = GlobalAveragePooling1D()(cnn\_branch)  
  
feat\_branch = Conv1D(16, 3, activation='relu', padding='same')(feat\_input)  
feat\_branch = GlobalAveragePooling1D()(feat\_branch)  
  
combined = Concatenate()([cnn\_branch, feat\_branch])  
combined = Dense(128, activation='relu')(combined)  
combined = Dropout(0.3)(combined)  
out = Dense(1, activation='sigmoid')(combined)  
  
model = Model(inputs=[cnn\_input, feat\_input], outputs=out)  
model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

## E.1.7 Model Conversion and Export

import tensorflow as tf  
  
converter = tf.lite.TFLiteConverter.from\_keras\_model(model)  
converter.target\_spec.supported\_ops = [tf.lite.OpsSet.TFLITE\_BUILTINS]  
tflite\_model = converter.convert()  
  
with open('TemporalCNNHeater.tflite', 'wb') as f:  
 f.write(tflite\_model)

## E.2 Real-Time Inference on Raspberry Pi (Python)

This section outlines the core Python code used for live inference on Raspberry Pi, involving background subtraction, feature extraction, and TensorFlow Lite inference with heatmap visualization.

## E.2.1 Load TFLite Model

interpreter = tflite.Interpreter(model\_path="TemporalCNNHeater.tflite")  
interpreter.allocate\_tensors()  
input\_details = interpreter.get\_input\_details()  
output\_details = interpreter.get\_output\_details()  
cnn\_input\_index = input\_details[0]['index']  
features\_input\_index = input\_details[1]['index']  
output\_index = output\_details[0]['index']

## E.2.2 Background Subtraction with EMA

def remove\_background(frame, use\_radiator\_bg=True):  
 bg\_model = bg\_model\_radiator if use\_radiator\_bg else bg\_model\_no\_radiator  
 bg\_model = bg\_model.reshape(32, 32)  
 adjusted\_frame = frame - bg\_model \* 0.9  
 adjusted\_frame[adjusted\_frame < -1.5] = -1.5  
 return adjusted\_frame

## E.2.3 Feature Extraction from Frame

def extract\_all\_features(sample):  
 return extract\_gradient\_features(sample) + extract\_hot\_region\_variance(sample) + extract\_edge\_features(sample)

## E.2.4 Live Inference Loop

if len(sequence\_buffer) == sequence\_length:  
 cnn\_input = np.array(sequence\_buffer).reshape(1, sequence\_length, 32, 32, 1).astype(np.float32)  
 features\_array = np.array(features\_buffer)  
 features\_normalized = (features\_array - feature\_mean) / feature\_std  
 features\_input = features\_normalized.reshape(1, sequence\_length, -1).astype(np.float32)  
 interpreter.set\_tensor(features\_input\_index, cnn\_input)  
 interpreter.set\_tensor(cnn\_input\_index, features\_input)  
 interpreter.invoke()  
 prediction = interpreter.get\_tensor(output\_index)[0][0]  
 predicted\_label = 1 if prediction >= 0.5 else 0

## E.3 Embedded Inference on ESP32-S3 (C++)

This section includes the core embedded C++ code segments deployed on the ESP32-S3 using the ESP-IDF framework. It highlights UART initialization, thermal frame parsing, quantization for the INT8 model, and inference queue logic.

## E.3.1 UART Initialization

void uart\_init() {  
 uart\_config\_t uart\_config = {  
 .baud\_rate = 115200,  
 .data\_bits = UART\_DATA\_8\_BITS,  
 .parity = UART\_PARITY\_DISABLE,  
 .stop\_bits = UART\_STOP\_BITS\_1,  
 .flow\_ctrl = UART\_HW\_FLOWCTRL\_DISABLE,  
 .source\_clk = UART\_SCLK\_APB,  
 };  
 ESP\_ERROR\_CHECK(uart\_param\_config(UART\_NUM\_1, &uart\_config));  
 ESP\_ERROR\_CHECK(uart\_set\_pin(...));  
 ESP\_ERROR\_CHECK(uart\_driver\_install(...));  
}

## E.3.2 Frame Synchronization and Parsing

if (std::string((char\*)&buffer[sz - 7], 7) == "<START>") {  
 syncing = true;  
 buffer.clear();  
}  
...  
if (tail == "<END>") {  
 // Valid frame received and parsed  
}

## E.3.3 Quantization and Tensor Creation

for (int i = 0; i < FRAME\_FLOAT\_COUNT; ++i) {  
 float val = float\_data[i];  
 int8\_t q = (int8\_t)roundf((val - QUANT\_ZERO\_POINT) / QUANT\_SCALE);  
 tensor->element[i] = q;  
}

## E.3.4 Inference Task and app\_main ()

frame\_input\_queue = xQueueCreate(2, sizeof(Tensor<int8\_t> \*));  
register\_presence\_det(frame\_input\_queue, NULL, false);  
xTaskCreate(uart\_receive\_task, "uart\_receive\_task", UART\_TASK\_STACK, NULL, UART\_TASK\_PRIORITY, NULL);

**Full source code and implementation scripts are available at:** <https://github.com/yourusername/thermal-presence-detection>