TinyML : Acoustic Burrowing Owl Vocalization Detection on STM32

SOFTWARE TEAM: ZACH LAWRENCE, ABHAY LAL, MAX SHEN
HARDWARE TEAM: REESE WHITLOCK, BENJAMIN SCOTT, KRUTI DHARANIPATHI

Microcontrollers are low cost, lightweight, and low power, making them the perfect device to monitor species in the wild. This paper explores the implementation of a resource-constrained machine learning model for acoustic analysis in research environments. Long-term monitoring and tracking of wildlife and endangered species in their natural environment is challenging due to human factors and logistical limitations. We present a microcontroller-based acoustic classification system that can identify the density of wildlife species in an ecological environment where human presence may be undesirable. We use the STM32H747I-DISCO board to record audio and run inference on the device using TinyML. The device is able to capture and classify audio from burrowing owls, allowing for research into their health, community, and behavior.

Additional Key Words and Phrases: Bird vocalization , Audio Classification, TinyML, STM32, Microcontroller, Convolutional Neural network, Burrowing Owl (Buow)

1 Introduction

Tiny machine learning (TinyML) is a fast growing field that focuses on deploying machine learning models on extremely resource constrained devices, such as microcontrollers (MCUs). These devices often have limited memory, low compute capability, and strict energy constraints, yet they are increasingly used in edge computing applications where always on, local intelligence is required.

This project presents the design, implementation, and evaluation of an embedded acoustic classification system capable of identifying burrowing owl vocalizations in real time using a TinyML model deployed on an STM32 microcontroller. The goal of this system is to enable low power, long term monitoring of owl populations in environments where human presence is either impractical or undesirable. Our system performs end to end audio capture, signal processing, and classification directly on the microcontroller, leveraging a fully quantized neural network to infer vocalization types on device.

We begin with the BUOWSET (burrowing owl set) dataset, a large collection of labeled burrowing owl audio clips provided by the San Diego Zoo Wildlife Alliance (SDZWA) and Engineering for Exploration (E4E). This dataset contains recordings across six vocalization categories, including alarm calls, chick begging, and ambient noise. These audio files are preprocessed into mel spectrograms, a time frequency representation that emphasizes perceptually meaningful frequency bands. We evaluate two lightweight convolutional neural network (CNN) architectures, MobileNetV2 [13] and ProxylessNAS [2], both commonly used in TinyML benchmarks. In addition, we design a custom model TinyOwlNet that is specifically tailored to operate within the MCU's strict memory and latency limits. All models are quantized to 8 bit integer format to reduce memory usage and improve inference efficiency.

The complete system is built to operate entirely on device. Audio is recorded using the MEMS microphone on the STM32 board, converted from Pulse Density Modulation (PDM) to Pulse Code Modulation (PCM) format, and then processed into mel spectrograms using a series of optimized digital signal processing (DSP) routines. These spectrograms are passed to the inference engine, which classifies the audio in real time. We use STMicroelectronics X-Cube-AI framework to compile and deploy the quantized models onto the MCU, and the entire inference process fits within the 512KB D1 cache of the Cortex-M7 core. By designing and profiling each step of this

Author's Contact Information: Software Team: Zach Lawrence, Abhay Lal, Max Shen
Hardware Team: Reese Whitlock, Benjamin Scott, Kruti Dharanipathi.

,
.

pipeline, we ensure that the system maintains real-time performance and meets memory constraints without relying on external computation or cloud services.

Our custom TinyOwlNet model achieves competitive accuracy while maintaining a compact footprint suitable for field deployment. Furthermore, the integration of audio preprocessing, neural inference, and efficient runtime scheduling on the STM32 platform exemplifies the principles of TinyML in a real world conservation application. By enabling smart, on device classification of wildlife sounds, this system represents a step toward more scalable, low power ecological monitoring solutions that can operate unattended in the field for extended durations.

2 Related Works

Deploying deep learning on microcontrollers (MCUs) has spurred the development of TinyML frameworks that jointly optimize model design and inference execution for severe resource constraints. One prominent example is MCUNet, which combines a neural architecture search (TinyNAS) with a memory optimized inference engine (TinyEngine) [9]. MCUNet's co design enables ImageNet scale models to run on off the shelf MCUs by tailoring both network structure and scheduling; it achieved the first >70% ImageNet top 1 accuracy on a microcontroller with dramatically lower memory usage than prior MobileNet or ResNet baselines. ProxylessNAS [2]itself was an earlier NAS approach that directly searched for efficient architectures on target tasks and hardware, yielding networks that surpassed MobileNetV2 in accuracy and speed by specializing to deployment constraints. These efforts illustrate how automated model design can push the limits of on-device inference. In parallel, the EdgeML initiative [11]explored ultra compact models like ProtoNN and Bonsai, which eschew deep convolutional networks for lightweight algorithms amenable to MCUs. Such models have demonstrated high accuracy (e.g. 98 % on MNIST) with as little as 6–8KB of memory, though their applicability to more complex audio/vision tasks is limited. Overall, modern TinyML strategies range from co-designed deep networks to radically small ML models, all aiming to expand the scope of AI on microcontrollers.

On device audio classification has emerged as a key TinyML application domain, with techniques developed for both speech and environmental sound recognition on embedded hardware. Keyword spotting (KWS) is a classic example [17], it showed that a small deep network can run continuously on a few kilobyte MCU and still achieve high accuracy (over 95% on a voice command task) by using an efficient depthwise separable CNN architecture. Their Hello Edge study demonstrated that with careful model selection (e.g. a depthwise CNN instead of a fully connected network), one can meet the strict memory and latency constraints of always on voice interfaces without sacrificing accuracy. Subsequent work has built full KWS pipelines on microcontrollers like the STM32, achieving real-time inference on 216 MHz Cortex-M7 cores with inference times on the order of tens of milliseconds. For instance, Wang and Li present a 12 class KWS system on an STM32F7 that produces predictions every 37 ms (including on-device mel spectrogram extraction) [16]. Notably, their baseline 8-bit CNN runs entirely on chip with 512KB of RAM, proving that non-trivial audio models can execute under tight MCU budgets. Beyond speech commands, researchers have also explored embedded environmental sound classification. Elliott et al. train Tiny Transformers on mel spectrogram features to classify office and ambient sounds, managing to outperform a traditional CNN with a model of only 6k parameters [4]. This Transformer based model, inspired by BERT but heavily pruned, achieved equal or better accuracy than a larger MFCC-based CNN while being small enough to fit on a microcontroller. Similarly, Choudhary et al. introduce LEAN, a two-stream audio model combining a learnable waveform encoder and a lightweight spectrogram CNN, which attains competitive results on the FSD50K sound dataset with a mere 4.5 MB memory footprint [3].

A crucial enabler for TinyML audio systems is model compression and inference optimization. Quantization to 8-bit (int8) weights and activations is now a standard practice to reduce model size and leverage efficient fixed point arithmetic on MCUs. Many frameworks (e.g. TensorFlow Lite Micro, X-Cube-AI) support quantization-aware training so that accuracy remains high after converting 32-bit models to int8. The above KWS system on STM32,

for example, uses quantized CNN kernels and even exploits ARM CMSIS-NN SIMD instructions to meet real-time speeds [16]. The authors also evaluate pruning: removing redundant weights to sparsify the network. Their findings indicate that structured pruning (e.g. removing entire filters) is far more beneficial on microcontrollers than arbitrary unstructured pruning, since structured sparsity can be mapped to skip whole operations without incurring irregular memory access overhead. In essence, pruned models need to be hardware friendly to truly yield speedups on MCU hardware. Other work focuses on optimizing memory usage during inference. Liberis and Lane propose reordering neural network operations to minimize the peak RAM needed for intermediate activations [8]. By carefully scheduling layer executions, they manage to deploy a standard CNN on an MCU with only 512KB of SRAM. This idea of memory aware scheduling is complementary to methods like network pruning: for instance, TinyEngine (from MCUNet) globally plans layer memory allocation and reuse, reducing peak usage by up to 4.8 times compared to naive layer by layer execution.

Our project's focus real-time bioacoustic monitoring on embedded platforms ties into a growing body of work on wildlife acoustics and conservation tech. Traditional passive acoustic monitoring often generates enormous recordings that must be post processed, but on-device AI can instead analyze audio locally and transmit only detections, saving power and bandwidth. Vuilliomenet et al. present acoupi, an open-source framework that integrates end-to-end audio capture, neural inference, and wireless reporting on devices like the Raspberry Pi [15]. They demonstrated acoupi with pre-trained models (BirdNET for bird species and BatDetect2 for bat calls) running autonomously in the field over a month long deployment. This shows the feasibility of continuous bioacoustic classification on low cost hardware. Our work on burrowing owl call detection is conceptually similar, but pushes into even more constrained hardware territory the STM32 microcontroller a custom-made lightweight CNN model for real-time inference.

On the applied side, [10] built a real-time audio classifier on an Arm Cortex-M0+ (RP2040), using Arm's CMSIS-DSP [1] for feature extraction and an 8-bit quantized TensorFlow Lite model for inference. Their pipeline processed 512-sample ($\sim$ 32 ms) audio frames in only 24 ms of compute time, leaving headroom for I/O and meeting real-time constraints. The fast computation for audio processing steps from Arm's CMSIS-DSP library contributed to our selection of the STM32 board. Similarly, [14] deployed a TinyML keyword spotting model on an STM32 Cortex-M4 board using the Edge Impulse studio. Their model was under 20 KB and the auto-generated C++ inference library leveraged the MCU's FPU, SIMD, and optimized CMSIS-DSP/NN kernels to maximize performance while minimizing RAM/flash usage (with the added benefit that all audio processing stays on device). Another experiment even ran a pruned RNN on an STM32 based hearing aid prototype, achieving 4.3 ms latency (well below a 10 ms audio threshold) after reducing model size by 47% [5]. These works informed our approach by underscoring the importance of model compression and quantization for memory constrained inference, the utility of vendor DSP libraries and peripheral optimizations for speed, and the need to carefully configure onboard sensors (e.g. microphone ADC/PDM interfaces) so that streaming audio data can be processed and inferred in real-time on the STM32 hardware. On tiny wake word benchmarks, MCUNet's specialized models outperformed earlier MobileNetV2 and ProxylessNAS networks while running 2-3 times faster and using 4 times less SRAM. [7] offer a comprehensive survey of TinyML, exploring the end-to-end toolchain from model building frameworks (e.g., TensorFlow Lite Micro, uTensor, Edge Impulse, NanoEdge AI Studio, STM32Cube.AI) to deployment platforms for executing ML workloads on edge devices [7]. They systematically analyze platform trade offs, including energy consumption, latency, memory footprint, and on-device accuracy, while discussing core constraints such as limited compute resources, fragmented tool support, dataset handling issues, and benchmarking challenges. The study concludes with recommendations for optimizing hardware software co-design and standardizing evaluation protocols . [6] propose TinyChirp, a TinyML enabled bird song classification system designed for low power acoustic sensors running on RIOT OS [6]. They evaluate multiple neural architectures (CNN-Time, Transformer-Time, CNN-Mel, SqueezeNet-Time), compare spectrogram vs. raw-audio inputs, and test quantization-based compression strategies. Their experiments measuring ROC curves,

accuracy, and F1 scores show that TinyChirp achieves precision above 0.98 while extending battery lifetime from just two weeks to eight weeks on a single charge.

3 Technical Material

3.1 Data Description

For this project, we utilized the BUOWSET dataset provided by the San Diego Zoo and E4E. This dataset contains human labeled acoustic data of burrowing owl vocalizations which is divided into the following 6 classes: *Cluck*, *Coocoo*, *Twitter*, *Alarm*, *Chick Begging*, and *no_buow* (no burrowing owl detected). The first 5 classes are types of burrowing owl sounds, and the sixth means that it is not a sound produced by a burrowing owl. 13021 3-second segments are no_bouw, taken from the same WAV files as the labeled detections. They were randomly selected, in equal number to the number of burrowing owl vocalizations in a given WAV file. There are 13022 human labeled detection segments, at varied lengths.

Fig. 1. Class-wise distribution for all the folds

3.2 Model Design

We have compared the performance of 3 models here. The MobileNetV2, ProxylessNAS, and our proposed TinyOwlNet. Using Transfer learning on MobileNetV2, ProxylessNAS pre-trained on Imagenet, we have fine-tuned them on the BUOWSET data. The TinyOwlNet curated for this task is compact and simple, specifically made for low-memory inference on the MCU. The audio segments from the BUOWSET are converted to mel spectrogram representations, which are then given as input to the model. The input dimensions are 1x64x258. The 64 Mel bands capture the frequency resolution, and the 258 frames are the temporal dimensions. This preprocessing converts the waveforms into a two-dimensional format based on time and frequency.

The TinyOwlNet model has three sequential convolutional layers that are each followed by a max pooling layer and kernels of dimensions 3x3 throughout to extract the local patterns. The first convolutional layer applies 8 filters of size 3x3 with padding, then it is followed by batch normalization and ReLU activation functions to introduce nonlinearities. After this, a max pooling with a 2x2 window size reduces the dimensions, and then

again the previous pattern is repeated with 16 filters in the second convolutional layer, and lastly followed by 32 filters. In conclusion, with an adaptive average pooling, we reduced it to a 1x1 feature map per channel, which is passed to the final classification layer. The final resulting vector is of size 32, which is fed into a fully connected linear layer to give the output of 6 class probabilities.

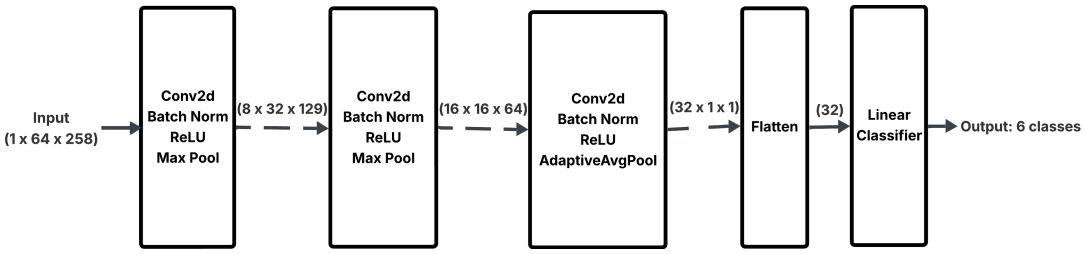


Fig. 2. TinyOwlNet Architecture Diagram

3.3 TinyML Methodology

In order to deploy our *TinyOwlNet* model to an embedded system for on-device audio classification, the following methodology was carried out to ensure inference did not exceed memory constraints. In order to decrease the memory required to store and pass inputs through the *TinyOwlNet* model, we quantize the model weights with int8 quantization. We use STM's proprietary inference runtime and code generation, *X-Cube-AI*, to generate optimized code, for our specific MCU, for passing inputs through the model on-device. In order to leverage this code generation, we converted the model artifact, which was trained using PyTorch, into the TensorFlow Lite format. In our case, we carried out both quantization and TensorFlow Lite conversion using a pipeline defined by Google's *AI Edge Torch* library, although in principle, any quantization and TensorFlow Lite conversion backend should suffice. Once the code was generated, we validated predictions on the MCU against the model off the MCU to ensure that inference was running as expected.

Our board of choice, the STM32H747I-DISCO board, has 2 cores, the Cortex M4 and Cortex M7. Due to the time and scope constraints and being in the discovery phase of the project, we decided to produce the entire pipeline using the M7 core, which has much higher-performance, instead of splitting the process between the cores. Our pipeline can be later optimized using both cores. The M7 core only has access to the level 1 (D1) cache, which is 512KB of RAM. Following int8 quantization and TensorFlow Lite conversion, the RAM required for the *TinyOwlNet* model, based on *X-Cube-AI*'s code generation profile, was 260 KB, and as such could be stored entirely in the D1 cache. This reduced the complexity of memory management with regards to inference. Additionally, due to our single-core design, we allocated all of the available Flash memory to M7 core, which totaled to 1.98 MB. This greatly exceeded the requirements for storing the model in Flash based on the code generation profile, which allocated 200 KB of Flash memory for model storage. As such, we maintain a sufficiently large overhead to store classification results for long-term deployments.

It should be noted, for the sake of reproducibility, that *X-Cube-AI*'s code generation using models quantized and/or converted to TensorFlow Lite format with ONNX was error-prone. Similarly, code generation targeting the TensorFlow Lite runtime, rather than the *X-Cube-AI* runtime, did not work in our case. Although *X-Cube-AI* shows improvements over TensorFlow Lite's runtime in terms of inference latency and memory constraints for other models [12], TensorFlow Lite is open-source whereas *X-Cube-AI* is not.

3.4 Audio Capture

On our MCU, we have developed a seamless audio processing pipeline that can then be used with real-time inference. The foundation of our system is built around the STM32H747I-DISCO development board, which we chose specifically for its robust processing capabilities and integrated peripherals. One of the key advantages of this board is its built-in MEMS MP34DT05-A microphone, which saves us from the complexity of external microphone interfacing and associated analog front-end circuitry. Getting the audio capture working properly required diving deep into the Board Support Package (BSP) documentation and understanding how to properly configure the microphone interface. The BSP acts as our bridge between the hardware-specific microphone drivers and our application code, handling the low-level details of the MEMS microphone communication protocols.

When we first capture audio from the microphone, the data comes in as a PDM data format stream, which is a high-frequency digital representation where audio information is encoded in the density of pulses rather than their amplitude. This audio is captured using a serial audio interface (SAI) and basic direct memory access (BDMA). BDMA puts the PDM data at a specified address in memory as the buffer fills. Managing the PDM buffer properly is crucial for maintaining real-time performance, especially since we are dealing with a continuous audio stream and capture. We do this by using two halves of the PDM buffer. There is a half-transfer callback and full-transfer callback initiated by BDMA, and in those functions we have to first invalidate the D1 cache by the buffer address before we access the data. This is to ensure that we get the latest data directly from RAM, because the data may be out of sync between the cache and RAM when the microphone data is being written directly to memory. After invalidating the D1 cache, we can then properly read the updated PDM buffer, written to using BDMA and SAI. The half of the buffer (first or second) that we use is determined by whether we are in the half transfer callback or full transfer callback. This ensures that we are not writing to and reading from the same part of the buffer at the same time.

3.5 Mel Spectogram

We then have to convert this PDM data into the format our model expects. First, we must convert PDM data into PCM format. The data must be converted into a PCM format because it is a more common and versatile format for audio processing and is much easier to convert into a mel spectrogram. This is because PDM format represents audio data as a high-density stream of 1-bit samples, while PCM uses a 16-bit representation for each sample, making it easier to transform into a mel spectrogram. We use the same strategy for the PCM buffer as we do for the PDM buffer and updated it accordingly, only writing to one half of the buffer at a time.

To convert the data, we apply a PDM2PCM filter from the PDM2PCM library to convert the PDM signal into the PCM format. Once we have PCM data in the corresponding half of our buffer, we need to transform it into a mel spectrogram, which is the input format our trained model expects. This transformation involves several mathematical operations, starting with windowing the PCM samples, applying Fast Fourier Transforms to convert to the frequency domain, and then mapping these frequencies to the Mel scale using pre-computed filter banks. The Mel scale is particularly important because it mimics human auditory perception, emphasizing lower frequencies where most speech information resides. The challenge here is that the Mel filter banks themselves are quite large and computationally expensive to generate during runtime. We ended up pre-computing these filters and storing them in SRAM, which means we had to carefully manage our memory budget to accommodate both the filter coefficients and the working buffers needed for the spectral analysis. The entire Mel spectrogram generation process needs to happen in real-time, so we spent significant time optimizing the mathematical operations and ensuring our implementation could keep up with the incoming audio stream without introducing latency that would affect the user experience.

The final step involves formatting the generated Mel spectrograms into the exact input format expected by our inference engine. This might seem straightforward, but there are actually quite a few details to get

right. The tensor dimensions need to match exactly, the data types need to be correct, and we need to handle any normalization or scaling that was applied during the original model training. Once everything is properly formatted, the mel spectrogram data gets passed to our inference engine, where the trained model processes it to generate predictions. The entire pipeline from microphone input to model output needs to operate within strict timing constraints to maintain real-time performance, which meant we had to carefully profile each stage and optimize bottlenecks throughout the implementation process.

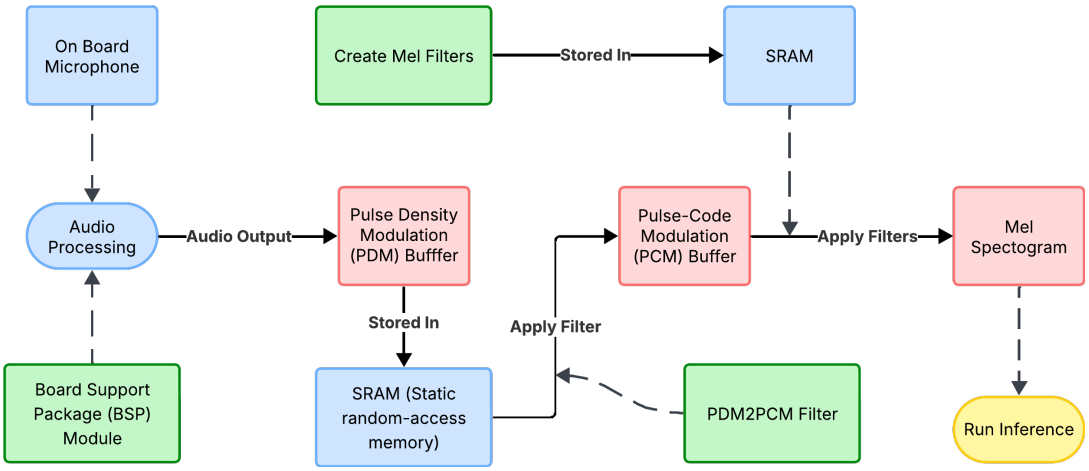


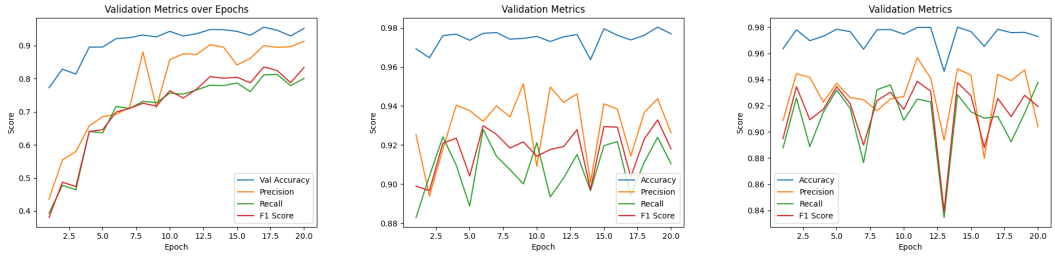
Fig. 3. On Board Processing Diagram

3.6 Results

We trained and evaluated three models: TinyOwlNet, MobileNetV2, and ProxylessNAS on the burrowing owl call classification task. The models were compared based on training loss, validation metrics and confusion matrix.

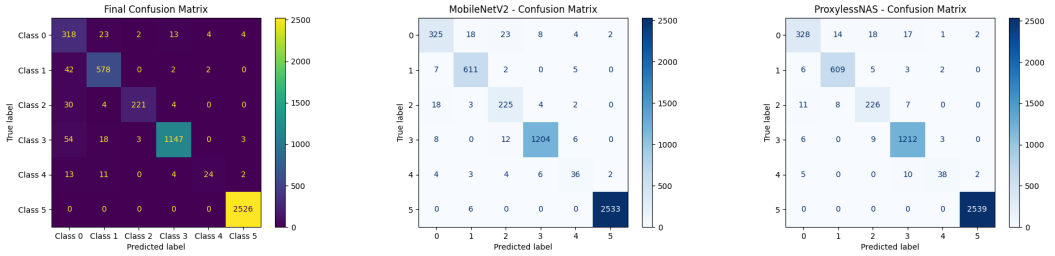
(a) TinyOwlNet (b) MobileNetV2 (c) ProxylessNAS

Fig. 4. Training Loss Curves for all models



(a) TinyOwlNet (b) MobileNetV2 (c) ProxylessNAS

Fig. 5. Validation Metrics (Accuracy, Precision, Recall, F1 Score)



(a) TinyOwlNet (b) MobileNetV2 (c) ProxylessNAS

Fig. 6. Confusion Matrices for all models

The training loss curves are shown in Fig. 4. TinyOwlNet showed stable convergence where both train and validation loss decreased smoothly across epochs. The gap between train and validation loss remained small which

indicates good generalization and low overfitting. For MobileNetV2, while the training loss kept decreasing, the validation loss had higher fluctuations especially after 10 epochs. This shows some instability while generalizing to validation data. ProxylessNAS showed similar trend but had slightly bigger spikes in validation loss after 10-12 epochs, which may point to model overfitting on few epochs.The validation metrics (accuracy, precision, recall, F1) for all models are shown in Fig. 5. TinyOwlNet had more consistent trends across the metrics where accuracy and precision gradually improved across epochs. The recall and F1 score also improved in a smoother manner. On other hand, MobileNetV2 and ProxylessNAS achieved higher peak accuracy (close to 98-99%) but showed more fluctuation across epochs specially in recall and F1 scores. This showed more fluctuation across epochs especially in recall and F1 scores.

The final confusion matrices are shown in Fig. 6 which give detailed class wise performance:

- **TinyOwlNet:** For class 5 there were 2526 samples that were classified correctly. Although performance in most classes seems to be less than those in MobileNetV2 and ProxylessNAS.
- **MobileNetV2:** Showed higher correct predictions specially for class 5 (Chick Begging) where 2533 samples were correctly classified. Some confusion still exists for Class 0 and Class 1,2.
- **ProxylessNAS:** Very similar to MobileNetV2, slightly better for minority classes. It classified 2539 samples correctly for class 5 and handled class 2 and 3 marginally better.

Model	Accuracy	Precision	Recall	F1 Score
MobileNetV2	**0.977**	**0.926**	0.910	0.918
ProxylessNAS	0.973	0.904	**0.938**	**0.919**
TinyOwlNet	0.948	0.905	0.825	0.839

Table 1. Final Validation Metrics across Models

TinyOwlNet achieved a final validation accuracy of 94.83%, with a precision of 90.48%, recall of 82.49% and F1 score of 83.94% as seen in 1. This model provided more stable learning curves and balanced performance, which makes it suitable for resource limited deployment. MobileNetV2 achieved a higher validation accuracy of 97.70% with precision of 92.61%, recall of 91.03%, and F1 score of 91.80%. ProxylessNAS also reached high performance with 97.28% validation accuracy, 90.37% precision, 93.78% recall and 91.94% F1 score. While MobileNetV2 and ProxylessNAS outperformed TinyOwlNet in absolute metrics, they showed more fluctuations during training. TinyOwlNet was more stable across epochs while still providing competitive performance.

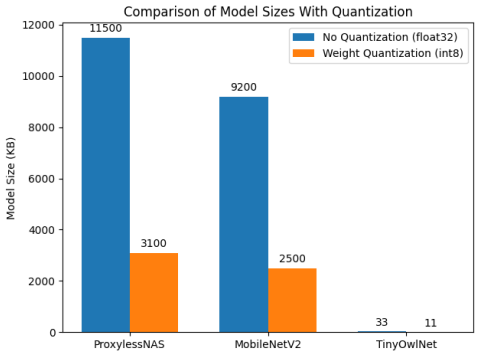


Fig. 7. Model Sizes before and after Quantization (KB)

We compare the model sizes before and after applying 8-bit weight quantization across the evaluated architectures. As shown in Fig. 7, ProxylessNAS reduces from 11500 KB to 3100 KB, and MobileNetV2 shrinks from 9200 KB to 2500 KB after quantization. Our custom TinyOwlNet achieves a significantly smaller footprint, with only 33 KB in float32 and 11 KB after quantization, demonstrating its suitability for the STM32.

4 Milestones

4.1 High-Level Deliverables (Key Milestones)

Milestone	**Description**	**Owner(s)**	**Evidence**	**Week**
M1:Replicate MCUNet	Reproduce 3 MCUNet results using TinyNAS & TinyEngine for pretained keyword detection model	Software Team	Desktop inference + profile report	Week 3
M2:Pre-process Owl Data and create baseline Owl Model Trained	Train and quantize audio classification model using owl dataset from the SD Zoo	Software Team	Model evaluation metrics compared across quantization schemes + saved model	Week 5
M3:STM32 Inference	Deploy model and run inference on STM32 board	Software + Hardware	On-device demo video with real-time audio data from microphone	Week 7
M4:Sleep Mode Added	Trigger sleep/wake mode via heuristics	Software + Hardware	Log showing mode triggers	Week 8
M5:Final System Test	Full system test (record + infer + sleep)	Full Team	Final demo + report	Week 10

Table 2. High-Level Deliverables – Software Team

Milestone	Description	Owner(s)	Evidence	Week
M1:Hardware Set-up	STM32 Board Exploration & Peripheral Scoping	Hardware Team	STM32 Manual + Github Repo	Week 2
M1:Peripheral Configuration on STM 32	initializing drivers for GPIO, DMA, and timer on the STM32 Board	Hardware Team	Timer + Interrupt on Github Repo	Week 3
M2:Audio Processing	Complete Audio Processing on the on board MEMS microphone	Hardware Team	Demonstarte Audio	Week 6
M3:STM32 Inference	Final Integration & Deploy model and run on STM32 board	Software + Hardware	On-device demo video	Week 7
M4:Sleep Mode Added	Trigger sleep/wake mode via heuristics	Software + Hardware	Log showing mode triggers	Week 8
M5:Final System Test	Full system test (record + infer + sleep)	Full Team	Final demo + report	Week 10

Table 3. High-Level Deliverables – Hardware Team

4.2 Weekly Milestones (Low-Level Schedule)

Week	Software Milestone	Hardware Milestone
3	Replicate MCUNet baseline using TinyNAS; begin profiling inference	Confirm STM32 development environment; Begin configuring peripherals
4	Use TinyEngine to compile model to C for STM32 deployment; test forward pass offline	Configure the audio peripheral and ensure audio is being properly recorded
5	Train and quantize owl model; test offline accuracy	Write embedded code for audio processing and storage
6	Run inference on STM32; monitor performance and latency	Align heuristics with audio activity levels
7	Compare quantized models; debug inference bottlenecks	Implement sleep and wake mode, and reduce power as much as possible
8	Full model deployment with logging on STM32	Profile power consumption under real input
9	Analyze performance; test noise handling; final tuning	Support testing and bug fixes
10	Record final demo; finalize performance metrics and documentation	Conduct final end-to-end test

Table 4. Weekly Schedule – Software & Hardware Teams

4.3 Hardware Milestones

4.3.1 STM32 Hardware Set-Up. For the hardware setup, we configured the old repository we were given to work in our setup. The old repository was missing a lot of drivers and had a lot of drivers located in incorrect

directories. We reconfigured this so that it works on the device that we are using. We have also set up all of the STM32 IDE and CubeMX that we need to use to work with the STM32 device.

We had some problems setting up the hardware because we cannot get some of our macOS devices to recognize the connected STM32 device. We spent many hours debugging trying to get this to work, but our efforts seemed to be futile. This meant that we were unable to use different computers and only had one laptop working to run the code we wrote. We decided to move on and focus our efforts on other parts of the project.

4.3.2 STM32 Peripheral Configuration. We set up the peripherals that were needed to use the on-board MEMS micro- phone which is what we were trying to use. To use the mic, we needed the GPIO clocks to be set and configured properly. We also need to have a way of storing the audio we receive from the buffer, so we created a buffer management to do that. To actually receive the data, we need to use both Serial Audio Interface (SAI) and Direct Memory Access (DMA). The SAI will get the audio data from the microphone in an analog format. DMA is then needed to transfer the audio data to be stored in the buffers where we can access it. This meant that we needed to connect one of the SAI instances to the DMA.

We ran into a lot of problems while setting up the peripherals. We had to set up SAI (Serial Audio Interface) and DMA (Direct Memory Access) and we used SAI instance 1 to connect it. It took some troubleshooting but we were finally able to connect SAI1 to DMA, but we found that when we tried to record audio, nothing was being loaded into the audio buffer, 7 which meant we weren't accessing the microphone properly. After doing a more deep-dive and finding a schematic, we noticed that the MEMS microphone was not connected to SAI1 but instead was only connected to SAI instance 4. We tried to make this switch, but because SAI4 doesn't connect to DMA, but instead only works with Basic Direct Memory Access (BDMA), so this was another change that we had to implement.

4.3.3 Audio Recording. We had found that using the MEMS microphone was becoming increasingly difficult, because we had tried many methods and built off of our mentor's work but were not able to communicate with it. After meeting with our mentor, we decided to switch to using an external mic that we connected to the input audio port on the device. We ended up using BSP (Board Support Package) to get the external microphone recording audio into the PDM Buffer. We found that the linker script for the old repo we were using had the wrong RAM Address (Random Access Memory). After finding this, we thought would help with using the MEMS microphone so we switched back to using that. This was because the incorrect RAM address/definition didn't affect DMA, which is why DMA with SAI1 was working, but it affected BDMA, which meant communicating with the microphone was impossible. After changing that file, we were finally able to record audio input using the on- board MEMs Microphone. We then changed the PDM buffer to a PCM buffer with a PDM to PCM filter library function, and we were able to hear audio record and play back in real time with the MEMs microphone.

We started this project trying to connect to the MEMS microphone had to completely change the microphone we were using because we had run out of strategies to communicate with it. So we switched tactics and decided to use the input port. As a temporary solution until we order a microphone, we are using regular wired earbuds that we connected to the audio port. We also ran into a problem when configuring the clocks and power. We accidentally set the power incorrectly which put the board into shut down mode and we were unable to connect to it. We fixed this by completely erasing the board to reset it. We also found that the reason the MEMs microphone isn't working is because the linker file had the wrong RAM address. After changing that then we got out audio to finally record and be stored

4.3.4 Inference. We successfully implemented and integrated the complete DSP pipeline on the STM32 platform to convert 16-bit PCM audio into Mel spectrograms suitable for on-device inference. This work builds directly on our previous pipeline tested on desktop, now adapted for embedded constraints. The embedded pipeline was validated using PCM buffers generated from owl audio .wav files and included the following steps: applying

a Hanning window, performing an FFT using the CMSIS DSP library [1], computing the power spectrum, applying a Mel filterbank, applying a logarithmic compression, and finally normalizing the spectrogram. Despite encountering memory limitations while storing Mel filter coefficients in SRAM, we optimized memory usage and filterbank storage to ensure stable performance. Once generated, the spectrogram was logged and visualized on the host to confirm that the embedded output matched expectations. Following validation, we integrated the Mel spectrogram output pipeline with the quantized CNN inference model on the STM32 board by sending the produced spectrogram into the *TinyOwlNet* classifier. This produced correct predictions for the owl audio sample used, confirming that the entire pipeline (signal acquisition, pre-processing, and inference) was functional and accurate on the device.

Our primary challenge stemmed from SRAM constraints, particularly with storing large Mel filter coefficients, which required aggressive memory optimization and reorganization. Additionally, because microphone capture was still under development and testing, we used a .wav-based PCM input as a stand-in for real-time audio.

4.3.5 Sleep mode. Initially, we planned on exploring implementing sleep mode heuristics along with power consumption profiling after implementing on-device inference for the end-to-end pipeline. Due to unexpected challenges and delays in completing the end-to-end pipeline, this quickly became slated as a "nice to have" feature rather than a necessary feature for the MVP. As expected, due to time constraints, we were unable to implement any sleep mode heuristics or power consumption profiling methods. However, it is worth mentioning that the time we spent generating code using *X-Cube-AI* pays off in this regard, as code generated using the platform allows for power consumption profiling bundled with the rest of the inference runtime's libraries. As such, the task of power consumption profiling should be made somewhat more trivial with *X-Cube-AI*.

4.3.6 Final system test. Because we were unable to implement power consumption profiling, and had just barely completed our end-to-end pipeline comprising the MVP by the end of the quarter, we had little time to carry out final system tests. We were, however, able to verify the model output from some inputs in our test dataset between on-device and desktop-ran audio classification using the full end-to-end pipeline. To this end, we were successfully able to test the record and infer steps of the process which we initially targeted, however without any sleep mode features. Implementing more robust analysis methods for power consumption and memory usage would be a great task for future work on this project, which we hope could be completed during this summer's REU program with Engineers for Exploration.

4.4 Software Milestones

4.4.1 Reproduce MCUNet results. The initial goal of this milestone was to replicate some of the results from the MCUNet paper in order to understand what we could re-purpose for our TinyML audio classification model. In particular, we wanted to replicate the test accuracy of some of the MCUNet baseline models as well as their latency, SRAM, and Flash usage using their inference runtime, TinyEngine. Towards this end, we were able to load MCUNet's baseline models and replicate their test accuracies, however we quickly realized that profiling the latency, SRAM, and Flash usage with TinyEngine would require us to load the models onto our STM32 board. Because the hardware team was not ready for us to load any models onto the board, we moved on to exploring model training using MCUNet's powerful TinyNAS pruning method. Towards model training, we were expecting to find training scripts or at the very least some scripts leveraging TinyNAS in order to perform a search of the network, as this is a major contribution of the MCUNet paper. We found, however, that the Han Lab did not share any such scripts, and as such there are a variety of unanswered GitHub Issues asking our same question, "how can we train a custom model using TinyNAS?". This led us to train alternatives to MCUNet, specifically looking at TinyML frameworks the MCUNet team compared their results to. We targeted the MobileNetV2 and

ProxylessNAS model architectures as baselines to explore, as they are more widely documented, for the most part, as compared to MCUNet.

4.4.2 Pre-process owl data and train baseline model. After identifying baseline models of interest, the ProxylessNAS and MobileNetV2 models were trained using the BUOWSET dataset that contains human-labeled acoustic data of burrowing owl vocalizations which is divided into 6 classes that are *Cluck*, *Coocoo*, *Twitter*, *Alarm*, *Chick Begging*, and *no_buow*. The BUOWSET metadata was parsed to extract the labels, segment durations and after this did fold assignments for each audio clip for train , test and validation. We loaded all the `.wav` files using `torchaudio` and resampled them to a uniform sampling rate of 44.1 kHz. In terms of pre-processing we applied a two step check for clipping and zero-padding each clip to a fixed 3-second duration in order to maintain uniformity, after which we which we transformed waveforms into mel spectrograms with 64 mel bands, hop length of 512, FFT size of 1024 in order to capture the audio features. From here, we performed per-sample z-score normalization on mel spectrograms to standardize input distributions which would be better for model inputs and applied a 5-fold stratified split based on the data, ensuring clips from the same source file do not leak across folds. From these 5 folds on data we used the 3 folds for training the data , 1 fold for validation and 1 for testing. In terms of training, we trained both models for 20 epochs each, using a batch size of 16 and Adam optimizer. These are lightweight models and hence were chosen for our task. During trainingm, we performed metric analysis to record loss, accuracy, precision, recall, and F1-score across all epochs. Notably, ProxylessNAS showed am improved macro F1-score over MobileNetV2 (0.92 vs. 0.89), with the most significant gain observed in the minority class Chick Begging, improving F1 from 0.67 to 0.77. We found that our validation loss was a bit high across both models but it is in range with what we expected based on results from the paper that the model is based on. There appeared to be overfitting from our training loss graph, which led us to consider training a smaller model.

4.4.3 Quantize PyTorch models and convert to TF-Lite format. This milestone was separated from the initial high-level milestones near the mid-point of the quarter, as we realized that this task required some decision-making to ensure that our choice of quantization and conversion backend would be supported by whichever inference runtime we targeted for deployment. Because there were, at the time, three choices for inference runtime, between *TF-Lite Micro*, *X-Cube-AI*, and *TinyEngine*, we spent a long time attempting to resolve issues regarding deprecated TF-Lite conversion backends and quantization methods. In the end, we developed two pipelines for quantization and TF-Lite conversion, one of which leveraged Google's *AI Edge Torch* library, and one of which leveraged the ONNX library, such that the PyTorch model was converted to ONNX, which was then quantized and converted to TF-Lite format. Both of these methods successfully allowed us to quantize the baseline models with int8 quantization, resulting in a roughly 3× decrease in on-disk model size, thereby making the models more readily deployable to a device with limited Flash memory. We also successfully verified that the test accuracy of the baseline models after quantization only decreased by 5%, and were able to successfully generate C header files from the models for use with TF-Lite Micro.

4.4.4 STM32 on-device inference and TinyOwlNet training. After quantizing the models successfully, we came to the most time-consuming and most challenging milestone of the project. As mentioned prior, there were 3 inference runtimes of interest, each of which came with its own slew of benefits and drawbacks. *TinyEngine* promised the lowest latency and most optimized memory management, however its code generation was largely deprecated and poorly documented. The code generation expected some metadata wrapped around the quantized model that suggested the developers intended for a user to quantize their model with a specific backend and TF-Lite conversion method, however this was not well specified. These issues led us to explore *X-Cube-AI*, STM's proprietary, close-sourced inference runtime, which had code generation support and is optimized for STM boards. The option for code generation made this runtime very appealing, however we were bottlenecked by

our model size. The runtime's code generation includes a profiling step in which the model uploaded during code generation is profiled for the necessary Flash and RAM required to store and run inference on the model. We realized that, in order to fit a model onto our board, it would need to be about 10× smaller than what we currently had. This led us to design and train our TinyOwlNet model, and prolonged this milestone greatly due to the unforeseen challenges with inference runtimes. We were able to design TinyOwlNet based on the CNN encoder-type architecture present in MobileNetV2 and ProxylessNAS, but with roughly 10× fewer layers. After quantizing the TinyOwlNet model, we were able to successfully generate code for the *X-Cube-AI* runtime, which we were able to validate on dummy inputs and pass off to the hardware team for integration with the full end-to-end pipeline. It should be noted that we did not end up working with the inference runtime we were considering, *TF-Lite Micro*, which is notably well-documented compared to the other two runtimes, but has little to no code generation support, and as such we were worried about the possible time sink in writing the model architecture code by hand.

4.4.5 Sleep mode. This milestone was not completed for the same reason that the hardware team could not complete their sleep mode milestone. Sleep mode heuristics and power consumption profiling were slated as outside the scope of the MVP nearing the mid-point of the quarter. Due to unexpected challenges and delays with generating code for on-device inference, we were unable to complete the sleep mode milestone.

4.4.6 Final system test. This milestone was also not fully completed for the same reason that the hardware team could not fully complete final system tests regarding power consumption and memory usage analysis with sleep mode. We believe our final systems test, however, regarding the recording and inference pipeline meet our MVP goals, and we would like to reiterate that gathering the aforementioned metrics would be a great target for the REU students this summer!

5 Conclusion

This project successfully demonstrates the deployment of TinyML for real-time, long-term, and low-power acoustic monitoring of animals on resource-constrained microcontrollers. The significance of this is that it is an unobtrusive way to accurately monitor wildlife in its natural environment for a significant amount of time. The benefit of using TinyML and not just recording audio is the classification element of the model. Storing all audio would require enormous amounts of post-processing, whereas this approach allows for intelligently classifying audio directly on-device, and only storing/recording significant noises, significantly reducing labor overhead and allowing for longer deployment.

Our primary contributions include the complete end-to-end pipeline for acoustic classification of barn owls on the STM32H747I-DISCO board. This encompasses real-time audio capture from the on-board microphone, eliminating the need for an external microphone, real time mel spectrogram generation using optimized DSP routines, and on-device neural network inference. We successfully designed and trained TinyOwlNet, a custom lightweight CNN architecture specifically optimized for microcontroller constraints, achieving 94.8% validation accuracy while fitting within the 512KB D1 cache limitation. Our quantized TinyOwlNet model requires only 260KB of RAM and 200KB of Flash memory, well within the constraints of typical microcontrollers, while maintaining real-time performance for continuous acoustic monitoring.

We have created a successful foundational MVP for future work to be built upon. Although we have implemented the full pipeline, there are many optimizations that should be done to fully take advantage of the microcontroller's capabilities. Sleep mode implementation as well as monitoring heuristics would significantly extend battery life for longer field deployment periods. Multi-core optimization utilizing both the Cortex-M4 and Cortex-M7 would allow for better processing efficiency as well as more complex models.

This work represents a significant step towards utilizing TinyML in wildlife monitoring and allow researchers with limited resources to study wildlife for long periods of time without disturbing their natural habitat, as well as having to analyze large continuous audio datasets. By showing how TinyML can run efficiently on low-cost, low-power microcontrollers, we open up new possibilities for scalable, distributed monitoring networks for conservation research of endangered species.

References

[1] Arm Limited. 2024. *CMSIS-DSP: Digital Signal Processing Library for Arm Cortex-M Processors.* https://arm-software.github.io/CMSIS_5/DSP/html/index.html.

[2] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. https://arxiv.org/abs/1812.00332. *arXiv preprint arXiv:1812.00332* (2018).

[3] Shwetank Choudhary, CR Karthik, Punuru Sri Lakshmi, and Sumit Kumar. 2022. LEAN: light and efficient audio classification network. https://arxiv.org/abs/2305.12712. In *2022 IEEE 19th India Council International Conference (INDICON)*. IEEE, 1–6.

[4] David Elliott, Carlos E Otero, Steven Wyatt, and Evan Martino. 2021. Tiny transformers for environmental sound classification at the edge. https://arxiv.org/abs/2103.12157. *arXiv preprint arXiv:2103.12157* (2021).

[5] Igor Fedorov, Marko Stamenovic, Carl Jensen, Li-Chia Yang, Ari Mandell, Yiming Gan, Matthew Mattina, and Paul N. Whatmough. 2020. TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids. https://arxiv.org/abs/2005.11138. Interspeech 2020, DOI:10.21437/Interspeech.2020-1864.

[6] Zhaolan Huang, Adrien Tousnakhoff, Polina Kozyr, Roman Rehausen, Felix Bießmann, Robert Lachlan, Cedric Adjih, and Emmanuel Baccelli. 2024. TinyChirp: Bird Song Recognition Using TinyML Models on Low-power Wireless Acoustic Sensors. *arXiv preprint arXiv:2407.21453* (2024). https://arxiv.org/abs/2407.21453.

[7] Rakhee Kallimani, Krishna Pai, Prasoon Raghuwanshi, Sridhar Iyer, and Onel López. 2023. TinyML: Tools, Applications, Challenges, and Future Research Directions. *arXiv preprint arXiv:2303.13569* (2023). https://arxiv.org/abs/2303.13569.

[8] Edgar Liberis and Nicholas D Lane. 2019. Neural networks on microcontrollers: saving memory at inference via operator reordering. https://arxiv.org/abs/1910.05110. *arXiv preprint arXiv:1910.05110* (2019).

[9] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. https://arxiv.org/abs/2007.10319. *Advances in neural information processing systems* 33 (2020), 11711–11722.

[10] Sandeep Mistry. 2021. End-to-End TinyML Audio Classification with the Raspberry Pi RP2040. https://blog.tensorflow.org/2021/09/TinyML-Audio-for-everyone.html. TensorFlow Blog, 6 Oct 2021.

[11] Sebastian Müksch, Theo Olausson, John Wilhelm, and Pavlos Andreadis. 2020. Quantitative Analysis of Image Classification Techniques for Memory-Constrained Devices. https://arxiv.org/abs/2005.04968. *arXiv preprint arXiv:2005.04968* (2020).

[12] Anas Osman, Usman Abid, Luca Gemma, Matteo Perotto, and Davide Brunelli. 2021. TinyML Platforms Benchmarking. arXiv:2112.01319 [cs.LG] https://arxiv.org/abs/2112.01319

[13] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv preprint arXiv:1801.04381* (2018). https://arxiv.org/abs/1801.04381.

[14] Daniel Situnayake. 2020. Keyword Spotting on STM32 Using Edge Impulse. https://www.edgeimpulse.com/blog/train-a-tiny-ml-model/. Edge Impulse Blog, 12 Nov 2020.

[15] Aude Vuilliomenet, Santiago Martínez Balvanera, Oisin Mac Aodha, Kate E Jones, and Duncan Wilson. 2025. acoupi: An Open-Source Python Framework for Deploying Bioacoustic AI Models on Edge Devices. https://arxiv.org/abs/2501.17841. *arXiv preprint arXiv:2501.17841* (2025).

[16] Jingyi Wang and Shengchen Li. 2022. Keyword spotting system and evaluation of pruning and quantization methods on low-power edge microcontrollers. https://arxiv.org/abs/2208.02765. *arXiv preprint arXiv:2208.02765* (2022).

[17] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello edge: Keyword spotting on microcontrollers. https://arxiv.org/abs/1711.07128. *arXiv preprint arXiv:1711.07128* (2017).

6 Acknowledgments

ChatGPT Citation: We used OpenAI's ChatGPT to assist with literature review, drafting, and editing of the Introduction and Related Work section. ChatGPT was also used for help with writing and editing other parts of the report.

We would like to thank Engineering for Exploration (E4E) and the San Diego Zoo Wildlife Alliance (SDZWA) for their invaluable support throughout this project. This work would not have been possible without the BUOWSET dataset, which they generously provided. We also appreciate their guidance and expertise in the field of wildlife monitoring, which helped shape our understanding of the real-world applications of this system.