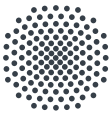
Institute of Electrical
and Optical Communications
Prof. Dr.-Ing. Georg Rademacher



Universität Stuttgart

Human Movement Detection via Fiber Optical Sensing

Master Thesis

Submitted at University of Stuttgart

Swapnil Bhavsar

Supervisor: M.Sc. Fabian Sauter (AP Sensing GmbH)

Start of Thesis: December 2024
End of Thesis: June 2025

Abstract

Perimeter monitoring is a cornerstone of security and surveillance. Distributed Acoustic Sensing (DAS) turns miles of optical fiber into a continuous, high-resolution microphone that captures tiny ground vibrations in real time. With recent developments in machine learning, never before seen new possibilities emerged to train models detecting and distinguishing between individual human activities based on DAS phase data. In this work, we present the spectrogram classifier framework developed in collaboration with AP Sensing to detect and distinguish various walking patterns and footsteps from DAS data. First, it transforms the raw DAS phase data into time frequency images (spectrograms), then feeds them into two modern vision networks: ConvNeXt V2 and EfficientNet. We train both ConvNeXt V2 and EfficientNet on two distinct dataset configurations: a single-channel stream sampled over 1.728 seconds, and a ten-channel stream sampled over 2 seconds yielding four total model-dataset combinations for direct performance comparison. On the ten-channel recordings, both models top 99% accuracy and produce almost no false alarms. By contrast, the one-channel setup drops to about 88% accuracy and fires off many more spurious alerts. This clear gap shows that giving the model a richer spatial picture (ten channels) dramatically boosts reliability. By combining GPU-accelerated preprocessing with these proven backbones, our pipeline moves from a lab prototype to a field-ready perimeter-monitoring tool.

Contents

Abbreviation

Abbreviation

Abbreviation	*Meaning*
API	Application Programming Interface
C-OTDR	Coherent Optical Time Domain Refractometry
CNN	Convolutional Neural Network
DAS	Digital Acoustic Sensing
DTGS	Distributed Temperature Gradient Sensing
DPU	Digital Processing Unit
EDFA	Erbium-Doped Fiber Amplifier
FBE	Frequency Band Energy
FIFO	First-In First-Out
GPU	Graphics Processing Unit
HDF5	Hierarchical Data Format version 5
I/O	Input/Output
IP	Internet Protocol
IU	Interrogator Unit
JSON	JavaScript Object Notation
ML	Machine Learning
RNN	Recurrent Neural Network
RGB	Red Green Blue
STFT	Short Time Fourier Transform
SSD	Solid State Drive
USB	Universal Serial Bus

List of Figures

Listings

1 Introduction

Perimeter monitoring is a cornerstone of modern security and surveillance of a location. A powerful technology for this application is Digital Acoustic Sensing (DAS) [8], which effectively transforms long stretches of fibre optic cable into a continuous array of highly sensitive microphones. These systems capture detailed acoustic data by measuring minute phase shifts in light traveling through the fibre, enabling the real-time detection of ground vibrations caused by various activities.

This thesis focuses on activities relevant to perimeter security, such as walking, climbing, or vehicle movement, as conceptually illustrated in the theoretical airport security layout in Figure 1.1. In such a scenario, a DAS system, with fiber optic cables indicated by the red lines, would be deployed to monitor the entire perimeter for potential intrusions.

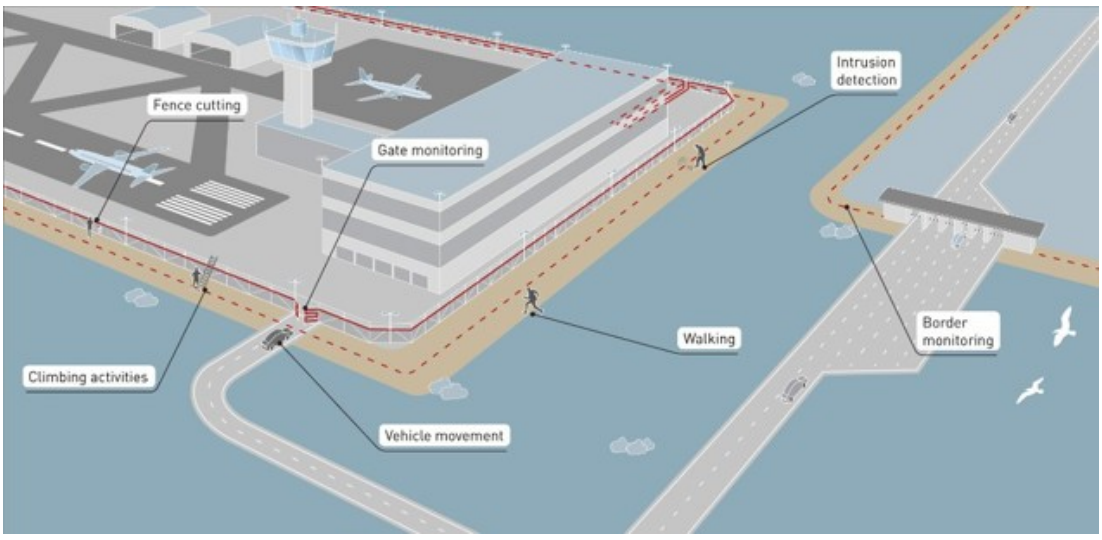


Figure 1.1: A theoretical perimeter monitoring setup at an airport [1].

While DAS provides a wealth of raw data, interpreting this data to reliably identify specific events is a significant challenge. To address this, this thesis presents the spectrogram classifier framework, a complete pipeline designed to detect and classify human footsteps from raw DAS phase data. The research presented was conducted in collaboration with AP Sensing, who provided the DAS system (Figure 1.2) and associated software used for data acquisition.

The core of the framework first transforms raw, one-dimensional DAS phase data into two-dimensional time-frequency representations called spectrograms. These spectrograms, which visually resemble images, are then fed into modern deep learning vision networks for classification. This approach leverages the advanced pattern-recognition capabilities of convolutional neural networks (CNNs) to distinguish between ambient noise and the unique signatures of footsteps.

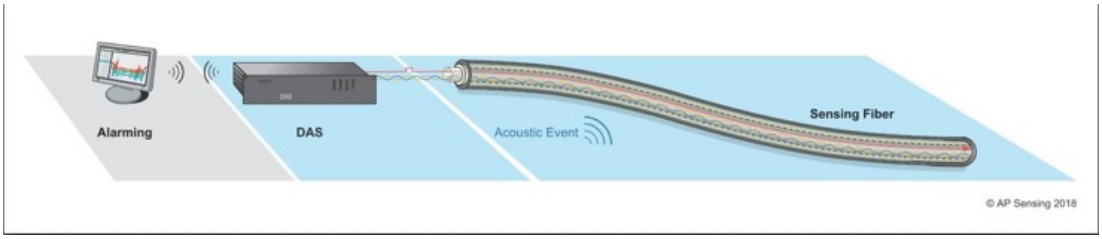


Figure 1.2: DAS setup [2].

To validate the framework and determine the most effective configuration, this thesis evaluates two state-of-the-art vision models: ConvNeXt V2 [9] and EfficientNet [10]. The performance of these models is rigorously tested by training and evaluating them on a custom dataset of walking patterns. By comparing their accuracy, false alarm rates, and real-time performance, this work identifies the optimal model and data configuration for a field-ready perimeter monitoring tool. Ultimately, this thesis demonstrates the efficacy of the Spectrogram Classifier Framework as a robust and reliable solution for security surveillance applications.

2 Background

This chapter details the principles of Distributed Acoustic Sensing (DAS), including the C-OTDR measurement principle, the components of the AP Sensing DAS hardware, and the functionality of the DAS Configurator application used for data acquisition and visualization. Then, it introduces the fundamentals of Machine Learning for image recognition, explaining key evaluation metrics derived from the confusion matrix, data augmentation techniques, and the basics of the PyTorch and Kornia libraries. The chapter concludes with a literature review that situates this work within the context of previous research on DAS-based activity detection.

2.1 Distributed Acoustic Sensing

Distributed Acoustic Sensing (DAS) transforms a standard fiber-optic cable into a dense array of thousands of virtual microphones. By sending pulses of laser light down the fiber and analyzing the coherent Rayleigh backscatter, a DAS system can detect minute physical disturbances acoustic vibrations or strain at any point along the cable's length. This enables precise localization and real-time monitoring of events over distances of several kilometers. Common applications include power-cable integrity checks, pipeline leak detection, train tracking, and perimeter security, where the system can continuously surveil and trigger alarms for anomalous activity.

2.1.1 Measurement Principle

The current measuring principle behind the DAS is measurement of acoustic vibrations based on the Coherent Optical Time Domain Refractometry (C-OTDR).

It measures the change in length and index of refraction of the fiber induced by the temperature changes and acoustic or seismic waves interacting with the cable [8]. Laser is sent into the fiber and the backscattered light is detected and analyzed.

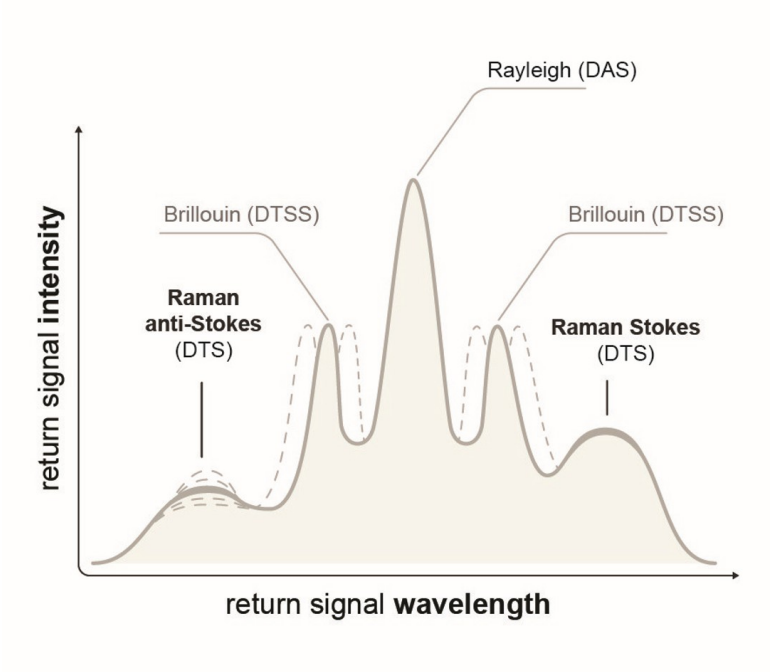


Figure 2.1: Spectral signal intensity distribution of the backscattered light [3]

Spectral signal intensity distribution of the backscattered light from the fiber is shown in Figure 2.1. DAS systems work on the Rayleigh Backscattering which is the highest peak in the Figure. It is a small fraction of light scattered by microscopic variations in fibre's refractive index. By sending the short pulses of laser and measuring the backscattering Rayleigh signal as a function of time, signals are localized along the fiber [11].

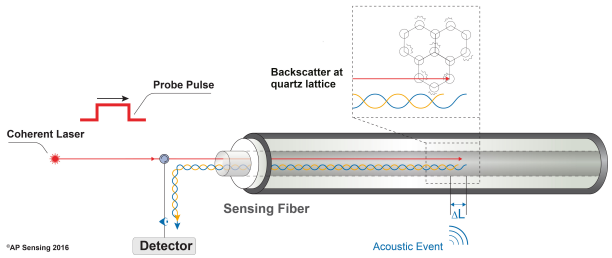


Figure 2.2: Working principle of DAS system [3]

Figure 2.2 shows how the laser is sent through the fiber optic cable and detected back after back scattering. When the acoustic event takes place, a tiny elongation or compression(ΔL) alters the phase relationship of the backscattered light signals. These signals are analyzed to show the frequency and amplitude disturbances. An Exact location of the acoustic event can be identified by measuring the time-dependent return of the light, the same principle used in radar echos.

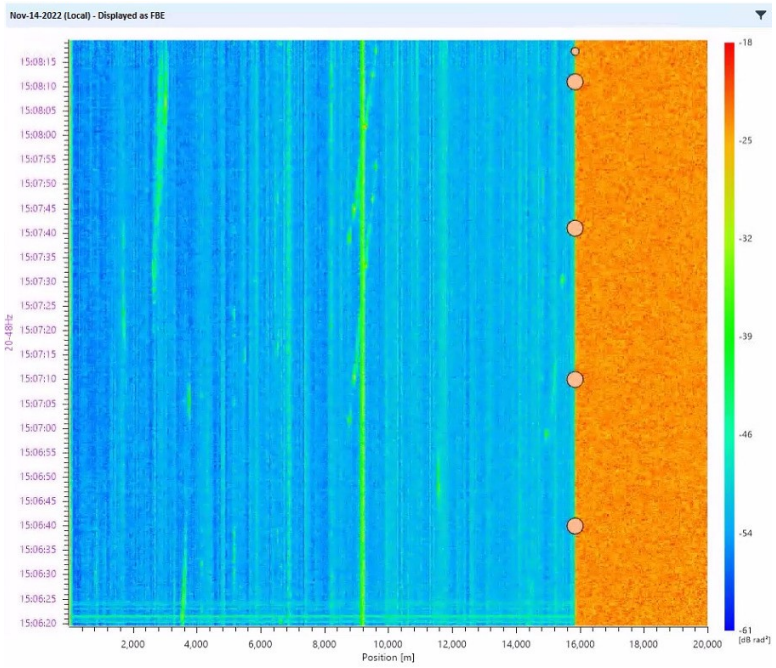


Figure 2.3: Waterfall Diagram [4]

After digitizing and filtering the analog signals, the data can be processed and analyzed over distance and time and displayed in the form of a waterfall diagram as shown in the Figure 2.3 where X-axis is the position on fiber and Y-axis is timestamp of recording. The data is recorded in the files in the HDF5 file format. The data from the HDF5 file is used to develop an algorithm which is used for the footsteps detection. Transforming the signal amplitude over time and in the frequency domain retrieves the acoustic event's frequency content. Using the acoustic event's location, amplitude, and frequency response, the system sets up an alarm for a potential threat-related activity.

2.1.2 AP Sensing DAS System

There are two main components of the AP Sensing DAS system - the Interrogator Unit(IU) and the Digital Processing Unit(DPU). Figure 2.4 shows the IU(top) and DPU(bottom) which are the part of the DAS system.

Figure 2.4: AP Sensing DAS device [3]

The IU(top) has the task of sending out highly coherent laser pulses into the fiber and thereby detecting the change in phase which is caused by Rayleigh backscattering. Analog signals containing the information in the form of frequency and amplitude are then sent to the DPU(bottom) for further processing. Figure 2.5 shows the rear panel of the IU and all the ports present.

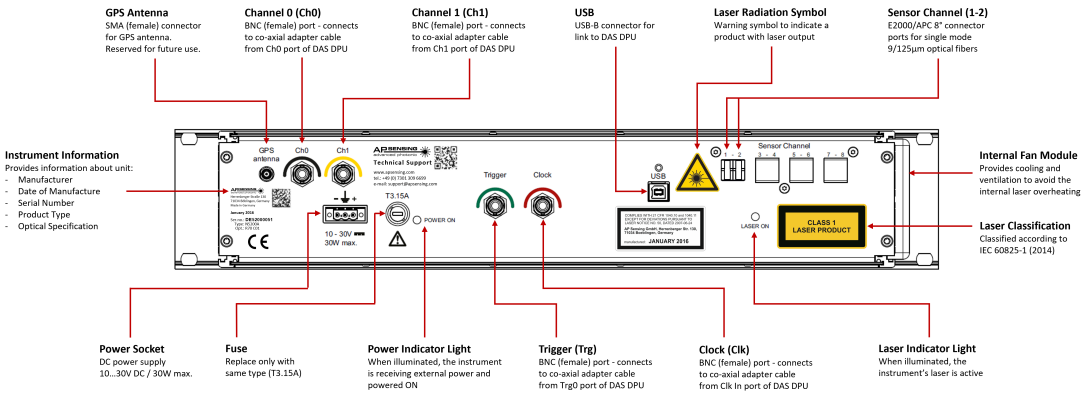


Figure 2.5: DAS IU rear panel view [3]

The IU(top) has two channels (horizontal and vertical polarization) which are

connected to the DPU(bottom). A USB connector is also connected to the DPU(bottom). A Power socket is needed to power the IU. The trigger which is used to trigger sending of laser pulse is also connected to the DPU. A Digitizer inside the DPU(bottom) is used for synchronization between the IU(top) and DPU(bottom). The IU(top) contains a reference coil of 120m optical fiber inside the case unit. The primary function of this reference coil is to monitor system noise and to checkw whether equipment is measuring accurately.

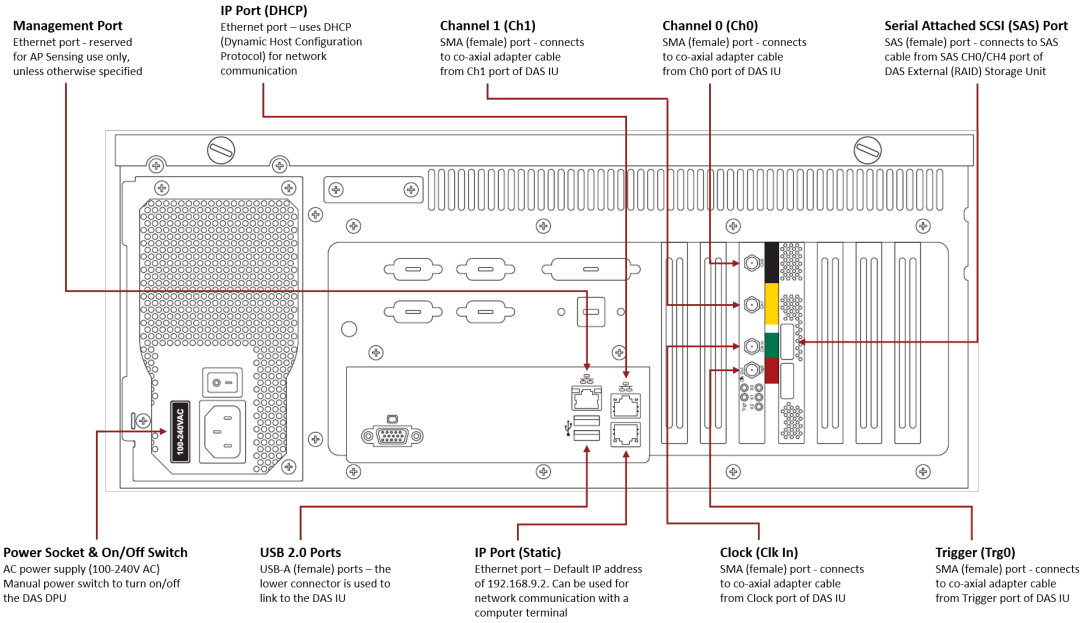


Figure 2.6: DAS DPU rear panel view [3]

Figure 2.6 show the rear view of the DPU of the DAS system. All the connections like the channels, USB, trigger and clock coming from the IU needs to be connected to the DPU. The DPU needs a separate power connection to power it up. There is also an Ethernet port which is used to connect the DAS system to network. The DAS system is connected to a computer via DAS Configurator application if the IP address of the system is known. All the calculations are started through this app. The data is stored in the form of HDF5 file. There are two SSDs available inside the DPU which can be used to store data. The stored data is automatically overwritten once when the capacity limit is reached, following the first-in, first-out (FIFO) approach. There is also other functionality that the data will be erased 30 days after the data is being stored.

The data outputted from the DPU is made available in the three different formats.

- Phase (Dynamic Strain): This data type provides the raw signal of dynamic strain along the fiber, making it the most data-intensive of the three.
- FBE (Frequency Band Energy): This data type aggregates the energy within each defined frequency band. After the backscattered signals are captured at each distance channel, they are divided into separate frequency ranges (e.g., 8-20 Hz). The total energy in each band is then displayed in the waterfall plot in Figure 2.3 of the DAS Configurator Client. Its primary purpose is to differentiate between various acoustic events.
- DTGS (Distributed Temperature Gradient Sensing): This data type highlights gradual variations in the signal arising from bulk temperature or strain effects. Conceptually, it functions like an additional low-frequency band, as the phase signal is analyzed at frequencies below 0.5 Hz.

2.1.3 DAS Configurator Client

DAS Configurator Application is an easy-to-use graphical interface for AP Sensing DAS systems. It allows users to configure and run measurements on the DAS system. The recording and saving of the measured data is also be done which can be useful for analysis and testing of trained models like ConvNext V2 and EfficientNet. It also helps in analyzing and visualization of the data. The data is collected from the DPU which acts as a server. It is designed to continuously run to record the data. The DAS client configurator application is installed on a separate computer which is connected to the DAS DPU via Ethernet.

Figure 2.7 shows the graphical interface of the DAS Configurator Application. The top and bottom bars are fixed while all other charts and panels are easily configurable. They can be freely enabled or disabled. There is a live and analysis mode selection button which can be used to select the respective modes. Then the stop, record or restart button which can be used to the respective tasks with the session. The print button is used to print the screen to a pdf file. The pause/restart button is used to perform the respective operations on the waterfall

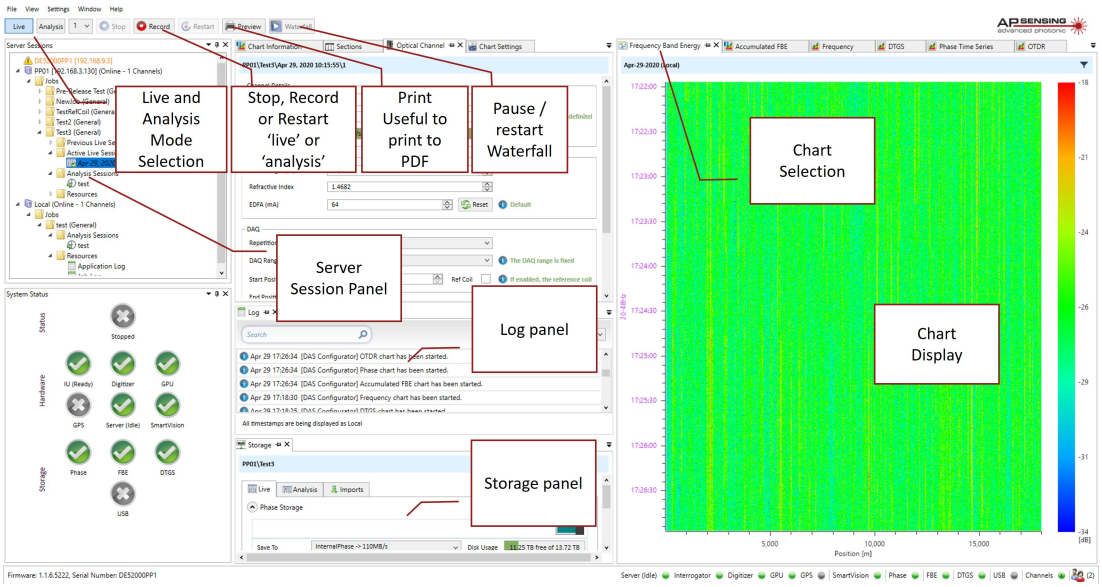


Figure 2.7: Graphical Interface of DAS Configurator Application [5]

diagram. Different charts like Phase, FBE and DTGS which can be selected for the layout. The server session panel where different recording session are stored in which all the data related to recording like the time of start and stop of the session, HDF5 files stored can be accessed. The storage panel shows the available disk space on the drive. The log panel is used to keep the log of the events which happen during the entire session.

Figure 2.8 gives an overview of the optical channel settings that needs to be done on the DAS system. All the configuration settings for the session are done here. The system specifies the channel's name using the "name" field, and it defines the recording time as the duration (in seconds) for which the recording is performed. The maximum data is 960MB/s and if the date rate is exceeded the field turns orange and a warning message is displayed or else its green. Fiber length is actual length of the cable. Refractive index is 1.4682 which is the default value for the single mode fiber. The value can be changed if the manufacturers value of the fiber is different. EDFA value is to be kept unchanged and is adjusted by the production team at AP Sensing. Pulse rate is the laser pulse rate of the DAS interrogator and maximum rate can be given by (math equation). Data rate is the rate of output traces per second, which can be lower than Pulse Rate to allow oversampling. The start and end position are the positions of start and

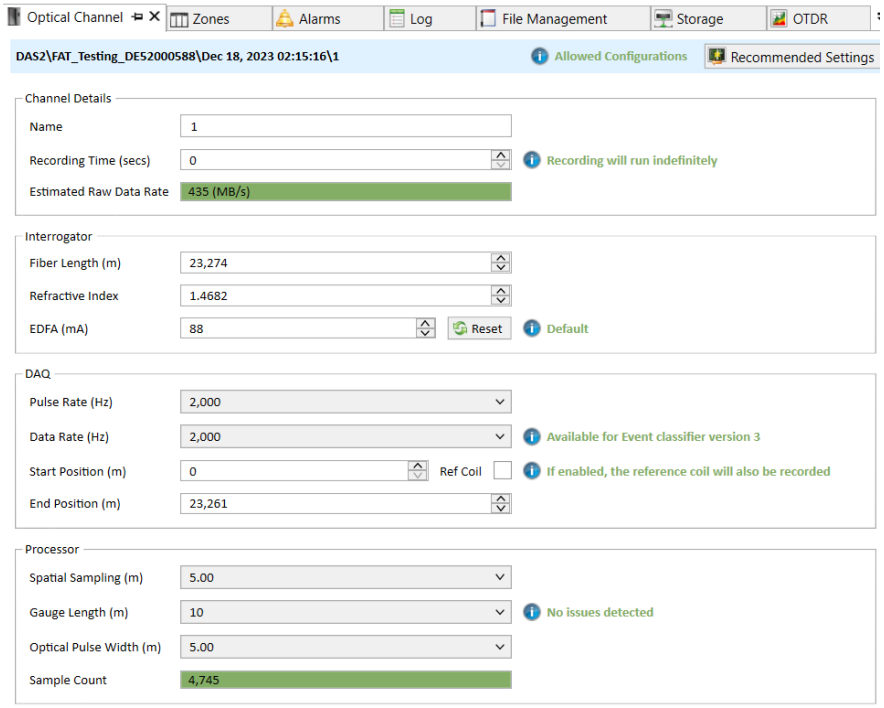


Figure 2.8: Optical Channel Window in DAS Configurator Application

end point of the recording respectively. The spatial sampling is sampling rate along the fiber length and can be adjusted from 1.25m to 40m. Gauge length is the distance between two phase measurements, determining spatial resolution and can be in the range 3.75m to 80m. Optical pulse width is the actual physical width of the pulse, adjustable by trained user which affects the DAS performance. The sample count are the number of samples to be processed. These are the total points in the fiber that are responsible for collecting the data which then further used for plotting the waterfall diagram or stored in HDF5 format. User can select which data can select the format of data (Phase, FBE or DTGS) as explained in previous section. The data from HDF5 files can be retrieved using any programming language as there are already packages or libraries available which can be used to access them.

2.2 Image Recognition using Machine Learning

Image Recognition is a process of detecting different patterns in the images and detecting similar images. Image Recognition is useful in crucial applications like autonomous vehicles, medical diagnosis and visual search. It has been revolutionized by deep learning techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) as it has better performance for image recognition. Machine learning methods for image recognition rely on algorithms trained on extensive collection of images. These systems automatically detect patterns and visual elements(edges, textures and colors) critical for categorization. After the training phase where the model is trained on a particular set of images, the model can automatically detect similar sort of patterns and images thus enabling accurate and scalable image analysis [12].

For a model to detect the similar sort of patterns, it needs to be trained and evaluated on a dataset. The dataset needs to be collected where the different kind of images need to be labelled and based on these labels the model is trained. Different augmentation techniques are also applied to the training dataset such as flipping images or rotation. The model training is thus generalized and is able to detect the images correctly irrespective of what new images the model comes across. The model should be trained in such a way that it doesn't memorize the patterns in images but actually learns the patterns and generalize them properly. Evaluation of the model needs to be done on the entirely different dataset which is not used in training phase. The performance of the model is determined by how well it performs on the evaluation dataset.

- **Data Preparation:** Preparing training and testing data by gathering the different kind of images on which the model needs to be trained. The preprocessing steps need to be applied on the images in Figure 3.3.3. These help in consistency of the data and help in proper visualization of the images.
- **Finding a model:** The pre-trained models are the models which are trained on the image dataset available online. These models are trained on large datasets and are available for use. The models can be used as a starting point for the training. The model can be trained on the dataset

available and then fine-tuned to get the best performance. The models can be used as a black box where the model is trained on the dataset and then used for evaluation.

- **Train the Model:** Training a model requires to provide a model with training data. Training data can be any form of images which are given to the model. Images are passed through various layers of the model from which understands the patterns. Model goes through this training data several times, the model will automatically determine the most crucial aspects. Model will acquire the ability to detect more characteristics and distinguish between various classes of data.

- **Evaluation Data:** To check whether the model is trained properly or not, the model needs to be evaluated on a different dataset which is not used in training phase. Evaluation dataset should be similar to the training dataset but should not contain any of the images from the training dataset. Evaluation dataset is used to check how well the model is trained and how well it can detect the patterns in the images. The performance of the model is determined using the confusion matrix. Confusion matrix is a table which shows the actual and predicted values of the model. Confusion matrix is used to check how well the model is trained and how well it can detect the patterns in the images.

2.2.1 PyTorch

PyTorch [13] is a machine learning library that provides an imperative and Pythonic coding style, emphasizing ease of use, seamless debugging, and alignment with other scientific computing libraries, while maintaining efficiency and supporting hardware accelerators like GPUs. The trend started in domain specific languages such as APL, MATLAB, R which turned multidimensional arrays (also known as tensors) into objects supported by mathematical primitives to manipulate them. Other libraries such as NumPy, Torch, Eigen and Lush made array-based(tensor) programming productive in general purpose languages such as Python and C++ [14].

Deep learning is computations on tensors, which are generalizations of a matrix that can be indexed in more than 2D. Data required is extracted in the form of NumPy arrays. Arrays are preprocessed according to the requirement in such a way that patterns in the images are clearly visible. Then NumPy data is converted into a tensor which then feed to a model for training. The model trains based on this tensor. There are various metrics which need to be monitored while training the model to see if the model is learning efficiently and not memorizing the patterns.

The efficacy of the model is the ability of a model to make the correct predictions. There are various metrics like Accuracy, Precision and Recall which need to be considered and are important to understand how well the model is trained. It revolves around categorizing the data points into predefined classes which is also known as classification problems. For instance, determining whether an email is a spam or not can be treated as an example of a binary classification. As the complexity of the model increases and the number of classes increase the intricacy of model increases. These metrics need to be monitored not only for the overall model but also over each class to see how well the model is classifying [15].

Confusion Matrix

The confusion matrix is a very important element in evaluating how well the model is trained. Model is improved by monitoring these metrics. The table 2.1 shows the different metrics. The comparison of actual outcomes with predicted outcomes is done. The class imbalance is understood here and based on that it is determined whether the problem is with a specific class or the whole model.

	Predicted		
	Positive	**Negative**	**Total**
Actual Positive	True Positive (TP)	False Negative (FN)	Total Positive
Negative	False Positive (FP)	True Negative (TN)	Total Negative
Total	Total Predicted Positive	Total Predicted Negative	Total Samples

Table 2.1: Confusion Matrix Example

The four main elements can be explained as follows which will give an overview of the models expected and actual outputs.

- **True Positive (TP)**: The model correctly identifies instances belonging to the positive class.
  Example: Spam emails automatically routed to the spam folder.
- **True Negative (TN)**: The model accurately identifies instances belonging to the negative class
  Example: Important emails delivered directly to the inbox.
- **False Positive (FP)**: The model incorrectly classifies negative instances as positive (Type I error). *Example*: Important emails mistakenly sent to the spam folder.
- **False Negative (FN)**:The model fails to identify positive instances, classifying them as negative (Type II error).
  Example: Spam emails slipping through the filter and appearing in the inbox.

There are different metrics like accuracy, precision and recall which can be calculated using the elements from the table 2.1. Ideally all the values should be high but depending on the use case the values are traded off to give the best performance for the case. Accuracy can be misleading as it fails to measure overall correctness when classes are imbalanced. So precision or recall need to be used to identify the performance of the minority class. The spam email detection example will give the better understanding for the same. The model labels all email not spam would get 95% accuracy but fails to detect any spam. If the model detects 80% of spam emails(high recall) but mislabels 20% of the legitimate emails(low precision) user can get annoyed by false alarms.

Accuracy

Accuracy is a metric that quantifies the overall performance of a classification model by measuring the proportion of correctly predicted instances (both true positives and true negatives) relative to the total number of instances. It is calculated using the formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \times 100\% \tag{2.1}$$

where:

- TP = True Positives
- TN = True Negatives
- FP = False Positives
- FN = False Negatives

Accuracy is often the first metric evaluated for classification tasks due to its simplicity and intuitive interpretation as a percentage of correct predictions. It works well when the dataset has balanced class distributions. However in imbalanced scenarios other metrics also need to be considered. In real life scenarios the cost of false negative (failing to identify a disease) might be more severe than a false positive in a medical diagnosis.

Precision

Precision is a metric providing how well the model predicts positive instances while minimizing the risk of false alarms. Precision is an important metric in classification problems where the cost is high for false positives. Precision is true positives from all positives detected by the model. The precision of a classification model is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP} \times 100\% \tag{2.2}$$

where:

- TP = True Positives
- FP = False Positives

Precision is very important when false positives are costly. False positives are very costly in instances like fraud detection systems, where flagging legitimate transactions leads to unnecessary investigation and customer dissatisfaction, potentially causing business loss. Precision only focuses on positive cases, neglecting all negative cases. A model achieves high precision by making fewer positive predictions, resulting in missing many positive cases.

Recall

Recall is a metric representing the proportion of actual positive cases identified by the model. Recall is also known as sensitivity or the true positive rate. The recall of a classification model is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN} \times 100\% \tag{2.3}$$

where:

- TP = True Positives
- FN = False Negatives

Recall is important where false negatives are costly. Recall focuses on finding all positive cases even with more false positives. A model may predict most instances as positive to achieve high recall. This leads to many incorrect positive predictions. Using these metrics and making adjustments as needed, the model is trained according to requirements.

2.2.2 Data Augmentations

Data augmentation implies a collection of different techniques to enhance the diversity and volume of training datasets to improve the model performance. The techniques include geometric transformations, color space adjustments, filter-based adjustments, filter-based methods, image mixing, random erasing. This helps in improving model generalization and diversifying the dataset thus helping in dealing with overfitting problem [16].

The data augmentation helps in generalizing the data better which will help in training the model with the patterns instead of model memorizing the images. Model will perform better on the unseen data when augmentation techniques are used.

Horizontal Flipping

Horizontal flipping mirrors the image along the vertical axis, thus creating a horizontally reversed image. This helps model to generalize better thus reducing the overfitting. Figure 2.9 shows the horizontal flip of cat image.

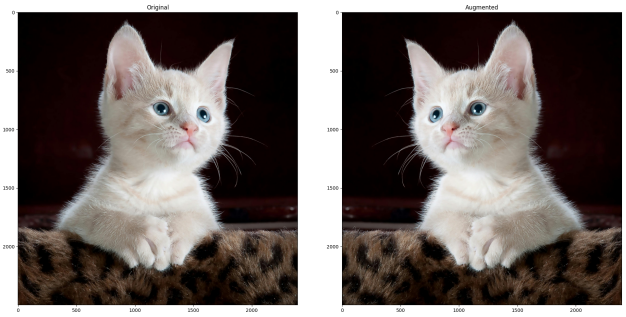


Figure 2.9: Horizontal Flip [6]

Vertical Scaling

Vertical scaling or amplitude scales the image on the vertical or amplitude scale, thus creating image which is stretched along the vertical or amplitude axis. This also helps in generalization and detects images which are stretched on the vertical scale. Figure 2.10 shows it with cat image how the image is stretched along the vertical scale or amplitude scale .

Horizontal Scaling

Horizontal scaling or time scaling is amplifying the image across the time scale or horizontal scale , thus creating image which is stretched across the time or horizontal axis. This also helps in generalization and even if the images which

Figure 2.10: Vertical Scaling [6]

are stretched on horizontal scale. Figure 2.11 shows this with a cat image where it is stretched along the time scale or horizontal scale for.

Figure 2.11: Horizontal Scaling [6]

Gaussian Noise Injection

In this augmentation, gaussian noise is injected into the image to increase the robustness and generalization of the model. The noise is added to the image by controlling the mean and standard deviation. Figure 2.12 demonstrates injecting the noise to a cat image with standard deviation set to 0.6 for demonstration purposes so that injection is visible.

2.2.3 Kornia

Kornia is an open-source computer vision library built on top of PyTorch2.2.1 that provides a familiar, NumPy-like API for classical vision operations (e.g.

Figure 2.12: Gaussian Noise Injection [6]

geometric transforms, filtering, feature detection) in a fully differentiable and GPU-accelerated form. Unlike the traditional 'torchvision.transforms' [17], which execute many augmentations on the CPU and then copy results to the GPU, Kornia implements each operation as a native PyTorch module or function that runs directly on CUDA. This tight integration allows to include data augmentations as part of the model's computation graph-making them differentiable, reducing host-device synchronization overhead, and enabling end-to-end optimization when desired.

In our spectrogram-based footstep classification pipeline, we use several Kornia augmentations (Horizontal Flip, Normalize) alongside custom PyTorch modules. By leveraging Kornia, we gain:

- **Performance**: All transformations (flips, normalizations, warps) execute directly on the GPU, avoiding costly host-device transfers (i.e., data is not sent back to the CPU for processing) and achieving higher throughput during training.
- **Differentiability**: Since each augmentation is a PyTorch module, gradients can flow through the transform when performing adversarial or meta-learning experiments that require backpropagation through data augmentation.
- **Composability**: Kornia's ops follow the same 'torch.nn.Module' and functional API conventions as PyTorch, allowing seamless integration into 'torchvision.transforms.Compose', 'torch.nn.Sequential', or custom training loops.

Together, these features make Kornia an ideal choice for high-performance, flexible data augmentation in end-to-end deep-learning pipelines [18].

2.3 Literature Review

DAS has emerged as a versatile tool in detection of various human activities, including footsteps by sensing the ground vibrations. Over the past decade, researchers have explored signal processing, Machine learning (ML), and environmental adaptions to enhance the detection accuracy. This section gives an overview of different studies which shows key advancements and challenges.

Time-frequency preprocessing. A crucial first step is turning raw DAS phase traces into a time-frequency representation that makes footsteps stand out. Ekimov and Sabatier [19] showed that human footsteps produce strong components in the 1-4 Hz band (with measurable energy up to 600 Hz outdoors) and used short-time Fourier transforms (STFTs) and spectrograms to isolate these bands. Likewise, K *et al.* [20] created synthetic datasets by slicing DAS recordings into 2 s windows and converting them into spectrograms-an approach we adopt directly, since it balances temporal resolution with computational cost.

Deep-learning classification. Once a spectrogram is in hand, convolutional neural networks (CNNs) have become the workhorse. Jakkampudi *et al.* [21] applied a CNN to 5 km of outdoor DAS data (2 m channel spacing) and reported 84% footstep vs background accuracy. To handle overlapping footsteps and changing directions, hybrid ConvLSTM architectures [22] have been proposed improving indoor tracking accuracy to about 81.5%. In our thesis, we build on these successes by comparing two state of the art backbones (ConvNeXt V2 and EfficientNet) and by introducing a targeted augmentation strategy to further reduce false negatives in challenging overlap scenarios.

Environment and noise adaptation. Outdoor DAS must contend with wind and distant traffic noise, while indoor signals suffer multipath reflections. Ekimov [19] emphasized that vibration energy above 600 Hz decays rapidly outdoors, which motivates our choice of filter bank design. Bublin [23] showed that combining DAS with auxiliary sensors (e.g. accelerometers) can improve robustness to transient events like excavator digging. Although we focus on single-sensor DAS here, our evaluation framework logs false-positive cases for future data-driven noise modeling or multimodal fusion.

Open challenges. Most prior work handles a single pedestrian or controlled corridor; detecting multiple simultaneous footsteps remains hard [21]. We likewise observe increasing false positives when multiple subjects share the same fiber segment. To address this, our model training pipeline incorporates random spatial and amplitude perturbations (Section 2.2.2) to encourage separation of overlapping patterns laying groundwork for multi-person DAS detection in future extensions.

Benchmarking ML on DAS. Shi and Zong [24] present one of the first end-to-end benchmarks comparing classical ML methods (SVM, Random Forest) against modern deep nets (CNNs, Transformers) on real DAS footstep data. They systematically evaluate different spectrogram parameters, data-augmentation recipes, and backbone architectures, and find that a lightweight vision-transformer slightly outperforms CNNs once trained with sufficient augmentation. Their results both validate our choice of ConvNeXt V2 and EfficientNet and underscore the importance of tuning augmentation hyperparameters.

Multimodal contrastive learning. Yadav *et al.* [25] fuse DAS spectrograms with co-recorded geophone signals in a metric-learning framework. By pulling together matching footstep pairs across sensors and pushing apart non-footstep pairs, they learn a joint embedding that boosts classification accuracy by 15-20 pp over single-sensor baselines. Although our current work is purely DAS-based, their fusion approach offers a clear path to tackle the false-positive challenges we observe when multiple subjects overlap (Section 2.2.2).

By combining the spectrogram windowing strategy of [20], the CNN-based classifiers of [21] and [22], the rigorous benchmarking from [24], and the multimodal fusion ideas of [25], this thesis delivers a robust, end-to-end footstep detector on both single- and ten-channel DAS data.

3 System Design

In this chapter, the end-to-end system design for capturing, processing, and leveraging phase data for training deep learning model training is presented. The experimental setup used describes the DAS deployment at the Energy Center in Hulb(Böblingen), the fiber-optic cable layout, and the HDF5 format [26] used to archive raw phase measurements. Introduction to a modular data processing pipeline from ingestion and HDF5 file handling, through STFT-based spectrogram generation and logarithmic scaling, to final data serialization illustrated via a block diagram that highlights each stage inputs, outputs, and dependencies. Finally, the model training and testing framework, specifying how ConvNext V2 and EfficientNet will consume these preprocessed spectrograms is outlined.

3.1 Experimental Setup for data acquisition

Data acquisition represents a fundamental component of this work, and every effort has been made to ensure that the collection process is robust and methodologically sound. The data is gathered using the DAS configurator application, as discussed and elaborated upon in section 2.1.3, it is then downloaded in HDF5 format. The data source is the Energy Building near the AP Sensing main office, where the DAS (Distributed Acoustic Sensing) system is installed. The DAS system, which operates using a state-of-the-art fiber optic cable deployed along a well-defined spatial route, captures walking patterns and other relevant phenomena. Figure 3.1 illustrates the complete fiber optic cable layout deployed at the Energy Building. This visualization is essential, as it provides context for the subsequent analysis and aids in understanding the spatial distribution of the data.

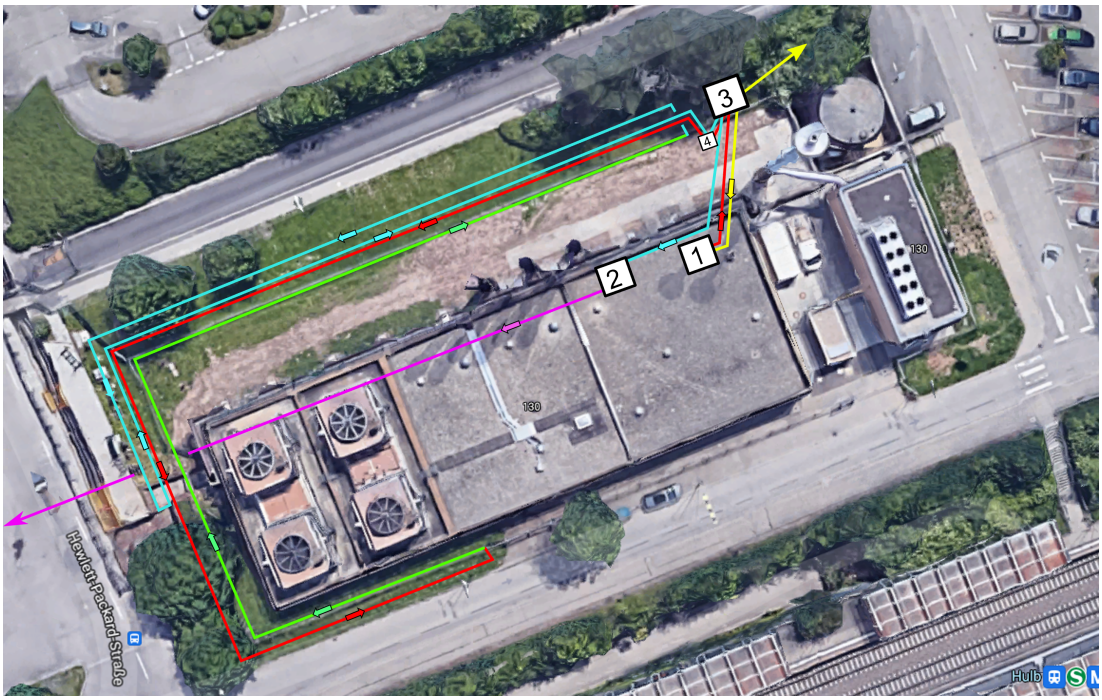


Figure 3.1: Fiber optic layout at the Energy Center [7]

The fiber-optic cable begins at the AP Sensing office (yellow) and then traverses four segments before reaching Herrenberger Street:

- **Trench near the fence (red, underground):** The cable runs below ground alongside the perimeter fence.
- **Fence line (green, above ground):** The cable is mounted directly onto the fence structure.
- **Distant trench (blue, underground):** The cable returns underground in a trench set back from the fence.
- **Final approach (purple, underground):** The cable continues beneath the ground toward Herrenberger Street.

Four numbered junctions mark critical splice and access points (Figure 3.1):

- **Junction 1:** Splice box receiving input from the AP Sensing office and feeding out to the building perimeter.

- **Junction 2:** Splice box receiving the perimeter feed and directing it onward toward Herrenberger Street.
- **Junction 3:** Main manhole outside the fence for underground routing.
- **Junction 4:** Smaller pit adjacent to the fence, also used for underground transitions.

This layout enables precise spatial localization of acoustic events along the cable route. The footstep data is recorded from the setup by walking around the cable route and logging it properly. DAS system records phase data that reflects variations corresponding to walking patterns. Phase data is stored in an in the HDF5 file in the form of `xarray` format a multi-dimensional data structure well-suited for handling large, complex datasets. Attributes such as data rate, gauge length, spatial channels which are set at the time of recording are stored in the HDF5 file in the form of metadata.

3.2 General structure of the system

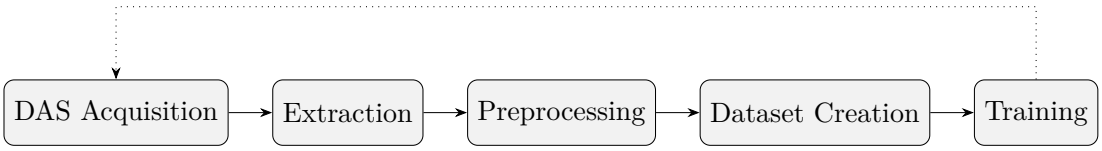


Figure 3.2: Block diagram of the general structure of the system

Figure 3.2 illustrates the general structure of the system what the thesis is trying to achieve. The system captures the data using the DAS system from the Energy building as explained in section 3.1. The walking data is recorded by taking into the consideration the layout of the cable and logging needs to be done properly to determine the exact time and spatial location along the cable where the footsteps are recorded.

The data is stored in the HDF5 format which stores the phase data with timestamps and the spatial locations. The footstep data is extracted from the HDF5 file and needs to be stored separately which is needed for training the model. Also, different types of background noise needs to be extracted from the HDF5

file which is needs for the model training. For clear distinction between the footsteps ans background data the data is preprocessed (section 3.3.3) and converted into spectrograms. The spectrograms are then used to create the dataset which is used for training the model. The spectrograms are used as the clear distinction between footsteps and background noise can be seen in the spectrograms. Using these spectrograms the model is trained and evaluation is done based on the accuracy of the model.

3.3 Internal Structure

Internally the structure of the system is split into various blocks as shown in Figure 3.2. The blocks are explained in detail in the following sections. The blocks are designed in such a way that they can be used independently and can be reused for other applications as well. Each block from Figure 3.2 has further smaller blocks which are explained in the following sections.

3.3.1 Data Acquisition

In the section 3.1 it is explained how the fiber optic cable is laid and using the DAS configurator application the data is recorded. The data is stored in the HDF5 format which is a file format used to store large amounts of data. Figure 3.3 gives the overview of data acquisition process. The data acquisition block is the main block from Figure 3.2 which records the data from the DAS setup using the DAS configurator application and stores it in the HDF5 file format. The phase data from this HDF5 file further visualized using the RGB array.

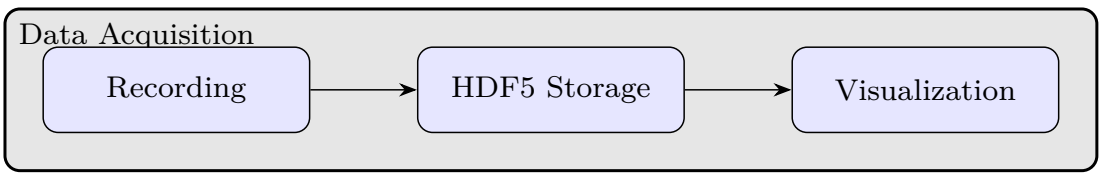


Figure 3.3: Data acquisition block: Recording phase data, storing in HDF5 format, and visualizing the data.

The data in HDF5 file is stored in the form of xarray format. The xarray format is a multi-dimensional data structure that allows for easy manipulation and analysis of large datasets. The phase data is stored in the HDF5 file with timestamps and the distance along the cable route at which the data is recorded. For better viewing the data is converted into a DataFrame which is a tabular format [27]. DataFrame contains the phase data with timestamps and the distance along the cable route at which the data is recorded. Table 3.1 shows the phase data in the form of a matrix with the distance along the cable route on the x-axis and time on the y-axis. Each cell in the table represents the phase data at a specific distance and time.

	Distance (m)				
Timestamps (UTC)	53638.285156	...	53701.992188	...	53726.496094
2024-10-21 10:27:40.000183+00:00	243.9019	...	1105.7420	...	854.9858
2024-10-21 10:27:40.001183+00:00	-908.2096	...	-3462.1868	...	2459.3839
⋮	⋮		⋮		⋮
2024-10-21 10:28:44.998183+00:00	-4907.9542	...	2141.9939	...	694.5940
2024-10-21 10:28:44.999183+00:00	-2337.2606	...	6081.5675	...	1535.1387

Table 3.1: Phase data from the HDF5 file in the DataFrame format

Phase data from the HDF5 file contains the numbers where the footsteps are not visible. To picturize the phase data, to view the footsteps clearly the phase data is plotted using matplotlib [28]. Data is converted into an RGB array using a Short-Time Fourier Transform (STFT) and a custom-designed filter bank. Filter banks are used to extract specific frequency bands from the phase data, allowing for a more focused analysis of the signal. The RGB array is then visualized using matplotlib, where each color channel corresponds to a different frequency band. The red channel represents low frequencies, the green channel represents mid frequencies, and the blue channel represents high frequencies. This RGB representation allows for a more intuitive understanding of the phase data and helps to identify patterns and anomalies in the signal. The application of a logarithmic transformation further enhances the visualization, making subtle patterns more evident. Figure 3.4 provides an illustrative RGB image of the raw phase data where the horizontal axis corresponds to the distance along the cable route and the vertical axis represents time, with time progressing from the top (earliest) to the bottom (latest). Notably, walking patterns can be observed in the red circle

and blue circle in the Figure 3.4.

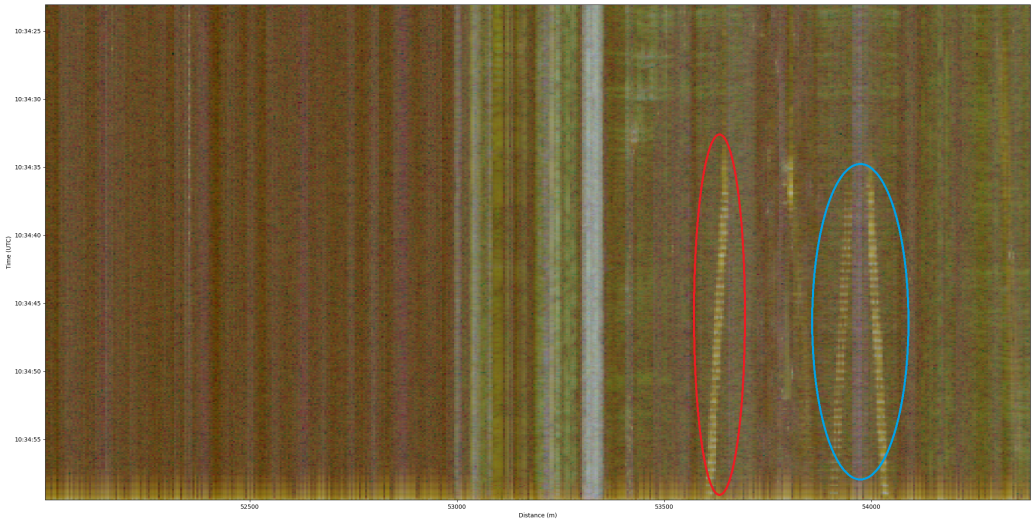


Figure 3.4: RGB plot for the phase data

3.3.2 Extraction

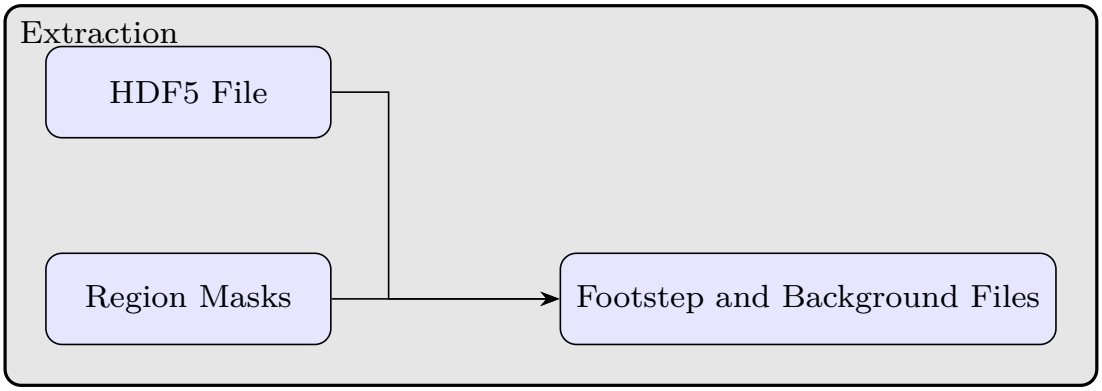


Figure 3.5: Extraction block: HDF5 file and Region masks are used to extract footstep and background data.

Figure 3.5 shows the internal structure of extraction block which serves to isolate and package all footstep and background segments for downstream processing. It takes two simple inputs the raw phase data stored in the HDF5 file and the corresponding region masks and produces a set of cleaned, per event files. This

abstract view hides the internal annotation details, focusing on the high-level data flow from stored measurements to ready to use footstep files.

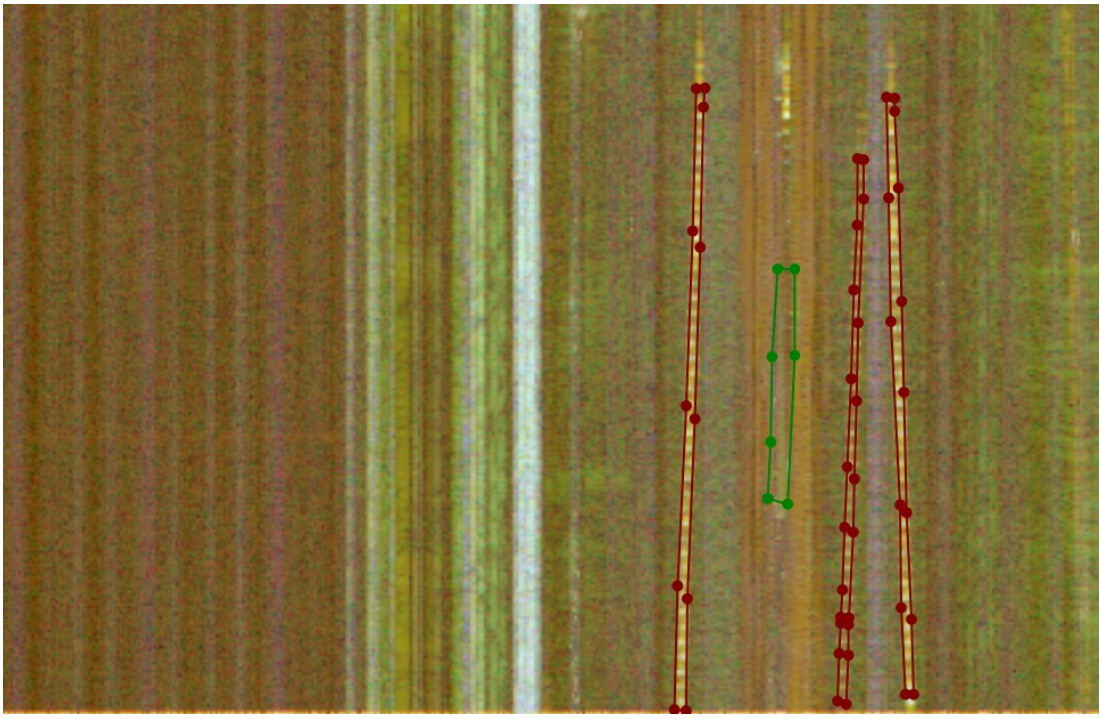


Figure 3.6: Region annotation of background and footstep data

Figure 4.1 shows the region masks for the annotation of the background and footstep data. The green annotation shows the background data and the red annotation shows the footsteps data. These annotations are used to store files in different folders such as background and walking. The annotation is done on the same HDF5 file shown in the Figure 3.4.

3.3.3 Preprocessing

The files generated from the extraction block need to be preprocessed in such a way that the footsteps are clearly visible. To help this cause, the preprocessing block converts the extracted files into spectrograms. The spectrograms are generated using the Short-Time Fourier Transform (STFT). Figure 3.7 shows the block diagram of the preprocessing block with all the operations done on the data files to get the spectrograms.

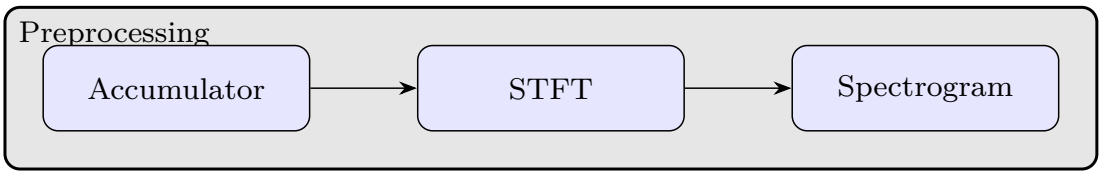


Figure 3.7: Preprocessing block: Footstep/Background files are processed through an Accumulator and STFT to produce Spectrograms.

Extracted data files from the extraction block are passed to the preprocessing block. Accumulator block takes the data files and integrates the phase-difference stream into a continuos phase signal and then the high-pass filters out slow drifts in such a way that only dynamic events like footsteps are visible. The STFT block takes this accumulated phase and converts it into a spectrogram. STFT is a mathematical technique that transforms the time-domain signal into the frequency domain, allowing for the analysis of the signal's frequency content over time. The output of the STFT is a spectrogram which is a visual representation of the signal's frequency content over time. Figure 3.8 shows such spectrogram which is generated from the footsteps data. The spikes seen in the spectrogram are the footsteps which are clearly visible.

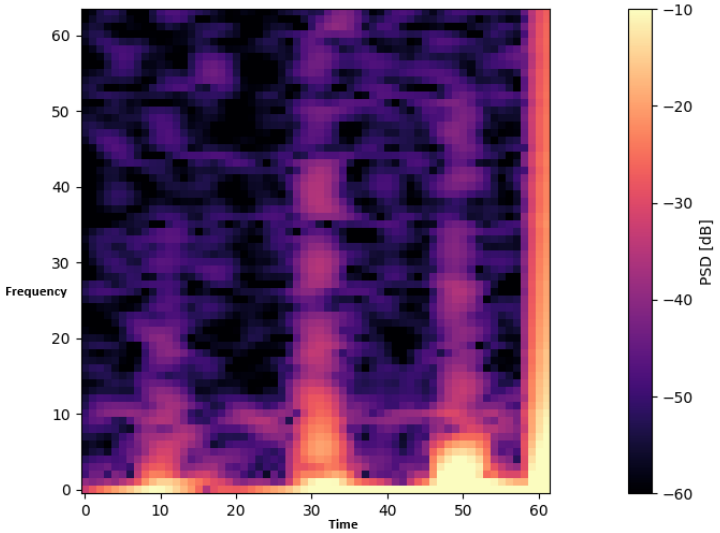


Figure 3.8: Spectrogram for walking data

Background noise is also be visualized in the spectrograms in a similar manner. Figure 3.9 shows the spectrogram for background noise data. The spectrograms has no pattern of footsteps here. The pattern of footsteps has spikes but back-

ground noise will have a constant pattern which doesn't have distortions. The footsteps will have a significant spikes in the spectrograms unlike the background noise. This is a clear distinction between the footsteps and background noise spectrograms.

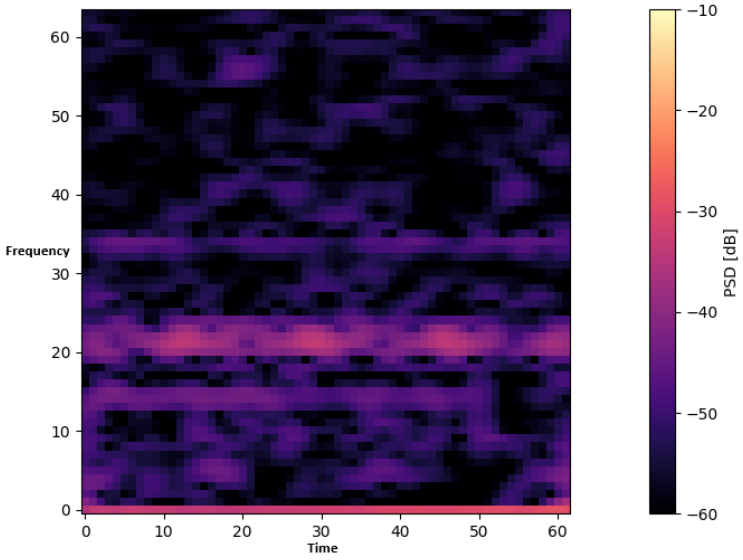


Figure 3.9: Spectrogram for background noise data

3.3.4 Training the model

Dataset created from the preprocessing block contains the footstep data and background data. Training block takes the dataset and trains the model using the ConvNext V2 and EfficientNet models. Figure 3.10 shows the block diagram of the training pipeline.

The dataset consists of the spectrograms of footstep and background data. The dataset is split into training and validation sets in the ratio of 80:20. Training set is being applied augmentations to generalize the dataset so the model learns the features of the dataset and not memorize the dataset. The augmentations are explained in the section 2.2.2. Training set is then passed to the model for training. Model is validated by just passing the validation set to the model. Model is trained and the checkpoint is saved. Checkpoint is the model weights which are saved after every epoch The model is then tested using the test set

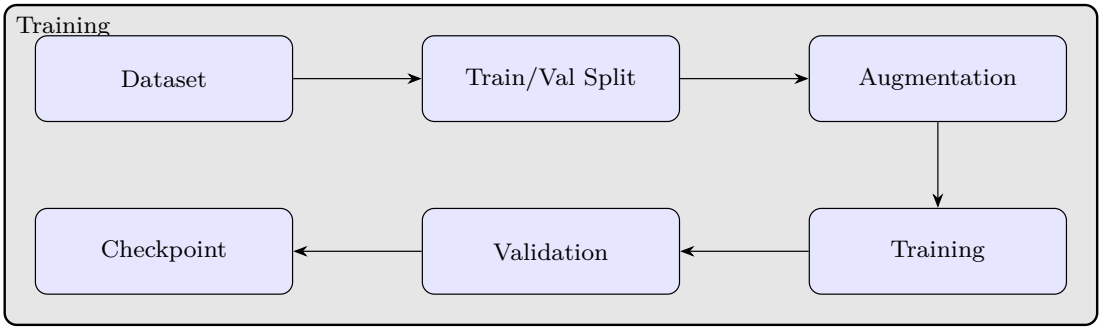


Figure 3.10: Model training block: Footstep/Background dataset is split into training and validation sets, augmented, and then trained and validated.

which is not used for training or validation. Test set is used to evaluate the performance of the model. Test set is passed to the model and the accuracy of the model is calculated using the checkpoint saved during the training. Testing follows the same process as validation, but done on the test set.

4 Implementation

In this chapter, we move from system design (Chapter 3) to the concrete realization of our pipeline. We detail how each component was implemented in practice from data extraction from the HDF5 files using region masks and custom scripts to signal preprocessing for spectrogram generation culminating in model training with various data augmentation techniques and checkpointing. Discussion on how the specific libraries and frameworks adopted, the structure of key code modules, configuration parameters, and the strategies employed to validate and manage intermediate outputs, providing a clear, end-to-end account of how the system was built, tuned, and prepared for deployment.

4.1 Extraction of Footstep and Background Data

Data collected from the setup explained in section 3.1 is stored in the HDF5 file. HDF5 file is visualized as explained in section 3.3.1 and shown in Figure 3.4. Region masks are used to extract the footstep and background with the help of annotations on RGB plot of the HDF5 file. RGB plot is used to enhance the visual interoperability, for which a Short-Time Fourier Transform (STFT) in tandem with a custom-designed filter bank. The application of a logarithmic transformation further enhances the visualization, making subtle patterns more evident. Figure 4.1 provides an illustrative RGB image of the raw phase data where the horizontal axis corresponds to the distance along the cable route and the vertical axis represents time, with time progressing from the top (earliest) to the bottom (latest). The RGB plot is opened using the LabelMe tool.

RGB plot has footsteps trails in it and we need to annotate it and extract the footsteps data using the RGB plot and HDF5 files. LabelMe tool is used to annotate the footsteps trail in RGB plot. LabelMe tool is a open-source image

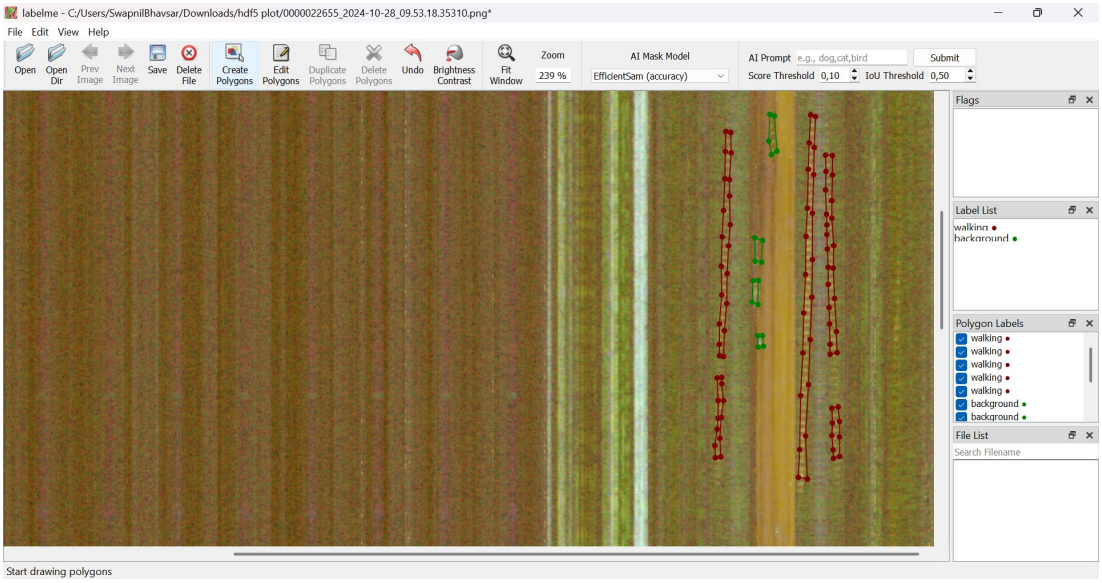


Figure 4.1: LabelMe tool for annotation

annotation tool that allows users to draw polygons on images and generate JSON files containing the binary masks corresponding to the annotated regions. The class labels are also added to the annotated regions. The extraction blocks takes the JSON masks and the HDF5 file which is used to extract the footstep data from the HDF5 file and store it in .npy or .bin file [29]. Figure 4.1 shows the window for the LabelMe tool. It contains a RGB plot for HDF5 file. It contains option to draw the polygons for the exact part which is needed for the dataset. Figure 4.1 has two different colors of polygons where red polygon is used to annotate footstep data and the green polygon is used to annotate the background data.

The output from Labelme includes JSON files, which contain binary masks corresponding to the annotated regions. A custom script is then used to extract the appropriate raw phase data segments from the HDF5 files by applying the binary masks. Listing 4.1 presents the core function that embodies the sample extraction process.

Listing 4.1: Core logic for extracting samples from a mask

```
def extract_samples_from_mask(mask, data_array, data_rate_hz,
                    window_len_sec, window_width_channels):
    window_length_traces = int(data_rate_hz * window_len_sec)
    hop_size_time = int(window_length_traces * 0.5)
    hop_size_space = max(1, int(window_width_channels * 0.5))
```

```
6
7       actual_mask = mask[::-1]
8       col_has_label = np.any(mask, axis=0)
9       row_has_label = np.any(mask, axis=1)
10      left_start = np.argmax(col_has_label)
11      right_end = len(col_has_label) - np.argmax(col_has_label
            [::-1])
12      top_start = _DOWNSAMPLE_FACTOR * np.argmax(row_has_label)
13      bottom_end = _DOWNSAMPLE_FACTOR * (len(row_has_label) - np.
            argmax(row_has_label[::-1]))
14
15      for start_x in range(max(0, left_start - hop_size_space),
16                               right_end - hop_size_space,
                                    hop_size_space):
17          for start_y in range(max(0, top_start - hop_size_time),
18                                   bottom_end - hop_size_time,
                                        hop_size_time):
19              chunk = data_array[start_y : start_y +
                    window_length_traces,
20                                       start_x : start_x +
                                            window_width_channels]
21              mask_chunk = actual_mask[start_y //
                    _DOWNSAMPLE_FACTOR :
22                                       (start_y +
                                            window_length_traces) //
                                            _DOWNSAMPLE_FACTOR,
23                                       start_x : start_x +
                                            window_width_channels]
24              if chunk.shape != (window_length_traces,
                    window_width_channels) or \
25                  np.mean(mask_chunk.astype(float)) <
                        _REQUIRED_MASK_PROPORTION:
26                  continue
```

In this function, several key computational steps are performed:

- The window length is calculated based on the data rate and the desired duration.
- A binary mask, generated from Labelme annotations, is processed to identify the columns and rows that contain labels.

- Nested loops are implemented to extract segments of the raw phase data along both spatial and temporal dimensions.
- The extracted segments, referred to as `chunk` and `mask_chunk`, are then verified based on their shape and the proportion of the label mask present before being stored for further processing.

The extracted files (.npy/.bin files) are stored in different directories based on the type of data (footstep or background) and the corresponding labels. Based on the labels, the data is divided into training and test sets. These sets are preprocessed before being used for training and testing the model.

4.2 Preprocessing of extracted Data

Following extraction, the extracted files needs to be preprocessed so that the clear distinction between the footsteps and background data can be made. The distinction is seen in the form of spikes which is visible in footstep data in comparison to background data as seen in Figure 3.8 and Figure 3.9 from section 3.3.3. The preprocessing is used to convert the raw phase data into a more interpretable format, specifically a spectrogram. This transformation is crucial for the subsequent training of the model, as it allows for the identification of patterns and features that are indicative of footstep events. The preprocessing pipeline is designed to handle the raw phase data efficiently, ensuring that the resulting spectrograms are both informative and suitable for machine learning applications. We use Scipy's signal tools [30] for filtering and STFT, torchaudio's Mel-scale filter banks [31].

The preprocessing routine, as demonstrated in Listing 4.2, involves reading the extracted data files, executing the STFT, and finally converting the resulting power spectrum into decibel (dB) values. The transformation to a logarithmic scale is critical, as it optimizes the dynamic range of the visualized data and makes both prominent and subtle patterns more discernible. We employ the `Visualizer` class which is the custom created class for these operations in the listing 4.2. In its constructor we build a Mel-filter bank matrix via `melscale_fbanks(..., mel_scale="htk",...)`, which yields finer resolution at low frequencies where footstep energy is concentrated and coarser resolution at high frequencies, improving

both compactness and interpretability of the spectrogram. The method `_accum` prepares raw data by taking a cumulative sum and then applying a high-pass filter to remove low-frequency drift. Finally, `preprocess` computes the STFT, squares the magnitude, and applies a $10\log_{10}(\cdot)$ transform to produce the final spectrogram. The logarithimic function make the spectrograms both visually clearer and more suitable to downstream classification.

Listing 4.2: Core processing for converting raw phase data to magnitude output

```
class Visualizer:
   def __init__(self, data_rate_hz: int = 1000):
        self._data_rate_hz = data_rate_hz
        self._f_bank = (
            melscale_fbanks(
                n_freqs= 128 // 2 + 1,
                f_min=0,
                f_max=500,
                n_mels=64,
                sample_rate=data_rate_hz,
                mel_scale="htk",
                norm=None,
            )
            .cpu()
            .numpy()
        )

    def _accum(self, x: np.ndarray) -> np.ndarray:
        x = np.cumsum(x, axis=0, dtype=np.int64).astype(np.
           float32)*INT_TO_RADIANS
        sos = signal.butter(2, 5, "hp", fs=1000, output="sos")
        return signal.sosfiltfilt(sos, x, axis=0)

    def preprocess(self, data: np.ndarray) -> np.ndarray:
        data_accum = np.cumsum(data, axis=0)
        f, t, stft = signal.stft(
            data_accum, fs=self._data_rate_hz,
            nperseg=128, noverlap=114,
            padded=False, scaling="psd", axis=0
        )
        stft = np.abs(stft) ** 2
        stft = np.matmul(stft.T, self._f_bank).T
        mag = 10 * np.log10(stft)
```

```
33          return mag
```

To facilitate the training of the deep learning models such as ConvNext V2 and EfficientNet. Two different models are trained so that a comparison can be made for their performances and accuracies, two separate datasets are generated through variations in the sample length and the number of spatial channels considered:

- The first dataset employs a sample length of 1.728 seconds and uses only 1 spatial channel. Sample length of 1.728 seconds as it contains 3 to 4 steps and the dataset with same configuration as there are already recorded files available. Based on the models trained on these, it is useful to detect shorter as well as longer footstep trails. The processing pipeline for this dataset is executed using the extraction procedure from Listing 4.1 and is subsequently converted into spectrograms following the process described in Listing 4.2. Figure 4.2 illustrates the resulting spectrogram for walking data acquired through a single spatial channel. In this Figure, the upper plot corresponds to the phase plot which reveals the unwrapped phase variations over time, whereas the lower plot represents the corresponding STFT spectrogram. Noticeably, the distinct spikes present in the spectrogram mark the precise moments when a foot makes contact with the ground.

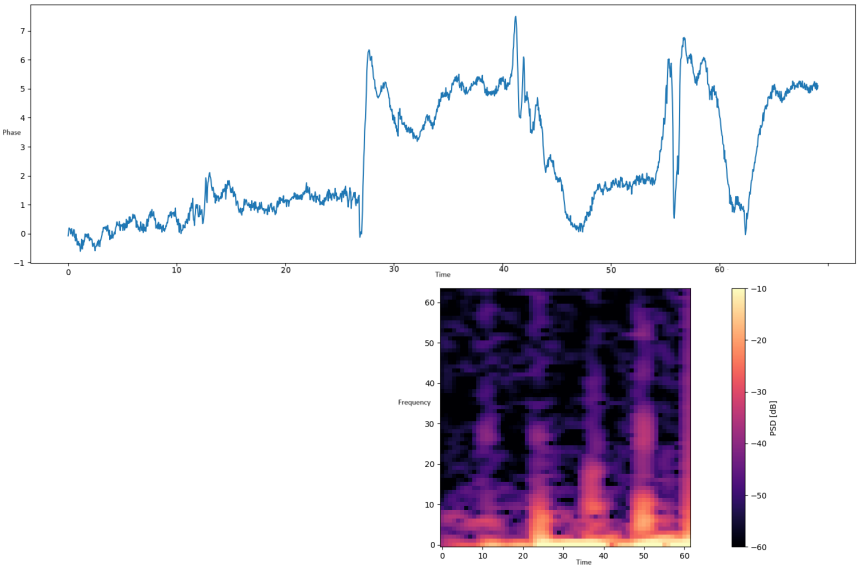


Figure 4.2: Spectrogram for walking data with 1 spatial channel

- The second dataset is designed for capturing more complex spatial and temporal interactions and uses a sample length of 2 seconds along with 10 spatial channels. The sample length of 2 seconds is taken which can be estimated as the time length in which a person can take 3 to 4 steps. The spatial channels record the data which are 5m apart. So the neighboring spatial channels record distortions based on the same footsteps and they can be detected many times by trained model. To avoid this 10 spatial channel are selected so model can exactly focus on footsteps and not the distortions in the neighboring spatial channels. The same extraction and preprocessing techniques are applied to this dataset. Figure 4.3 shows that the top plot represents the distribution of the spatial channels over time, while the bottom plot highlights the frequency distribution. Importantly, files that lack clearly discernible footsteps are manually filtered out, ensuring that the final dataset is diverse and rich in footstep patterns. This deliberate filtering is paramount as it enhances the model's ability to generalize across various walking patterns and substantially reduces the occurrence of false detections.

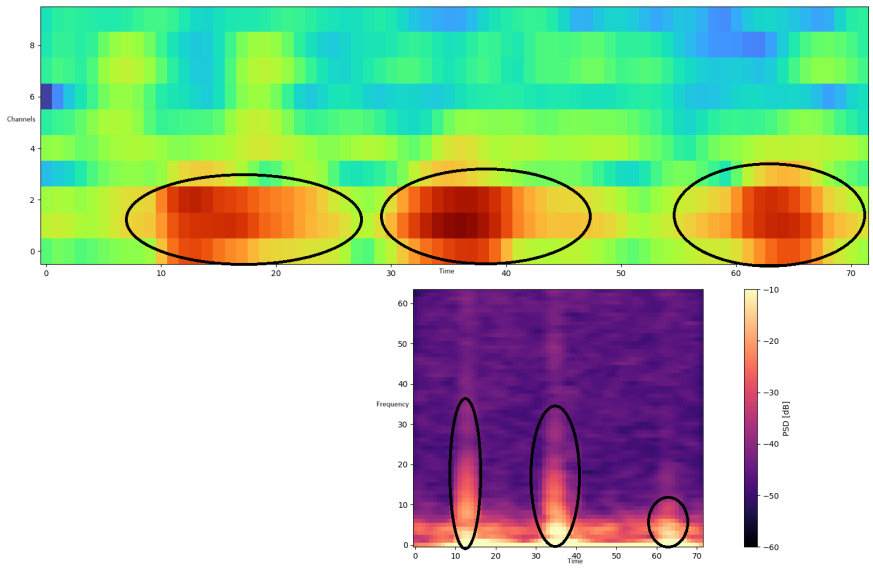


Figure 4.3: Spectrogram for walking data with 10 spatial channels

4.3 Dataset

The dataset is created with the help of preprocessing steps. The raw extracted files (.npy/.bin) are stored in the folder and before the dataset is loaded to train the model, the preprocessing steps are used. Dataset are of two types as described in the section 4.2.

- `Dataset-1D`: 1.728 seconds sample length and 1 spatial channel
- `Dataset-2D`: 2 seconds sample length and 10 spatial channels

Both datasets contains the `training` and `testing` folders which are used for the same purpose as the name suggests. Each of these folders has two folders `walking` and `background` which are different events that are used as labels for training the models.

`Dataset-1D` training folder 5818 walking samples and 2696 background samples. Testing folder for the same dataset contains 988 walking samples and 1590 background samples. `Dataset-2D` contains the same folders as `Dataset-2D` where training folder contains 1982 walking samples and 421 background samples. Testing folder contains 65 walking samples and 1682 bakcground samples.

`Dataset-2D` has much lesser samples compared to `Dataset-1D` because over the same HDF5 files the samples are created using 10 spatial channels. So in `Dataset-2D` the one sample consists of 10 samples clubbed together from `Dataset-1D`.

4.4 Training and Validation of the Model

The training of the model is done using the PyTorch framework [13]. Pytorch framework has rich ecosystem of libraries like which helps is leveraging of the models ConvNext V2 and EfficientNet from `timm` [32]. The dataset made with the help of preprocessing is used to train the model. The extracted data in the dataset folders is augmented using the different augmentation processes. Numpy arrays given out by the preprocessing function in listing 4.2. As the model requires the

tensor input, the numpy arrays are converted to tensors using the `torch.from_numpy` function. All the further steps before inputting the data to the model are carried out using the tensors. Normalization of the data is a very important step which is done before inputting the data to the model. The tensors for the case are not actually the image tensors so the mean and standard deviation are calculated for the spectrograms which are used for the normalization. Listing 4.3 shows the code for reading the data files and computing the mean and standard deviation. The folder path with all the data files(training and testing both) is given to the function `read_files_from_folder` which reads the data files and gives it the visualizer class that contains the function for spectrogram preprocessing shown in the Section 3.3.3. All the values are stacked together and given to the function `calculate_mean_std` which calculates the mean and standard deviation of the data. These mean and standard deviation values are used for the normalization of the data.

Listing 4.3: Reading data files and computing dataset statistics

```
1    def read_files_from_folder(folder_path: Path):
2
3        all_data = []
4        viz = Visualizer()
5
6        for file_path in folder_path.rglob("*"):
7            if file_path.suffix == ".bin":
8                with open(file_path, "rb") as f:
9                    data = np.fromfile(f, dtype=np.int64)
10               data = data.reshape((length, channels))
11           elif file_path.suffix == ".npy":
12               data = np.load(file_path)
13           else:
14               continue
15
16           stft_data = viz.data_process(data)
17           all_data.append(stft_data)
18
19       if not all_data:
20           raise ValueError(f"No valid data files found in {
                 folder_path!r}.")
21
22       stacked_data = np.stack(all_data, axis=0)
```

```
23          return stacked_data
24
25
26      def calculate_mean_std(data):
27          tensor_data = torch.from_numpy(data.astype(np.float32))
28          mean = torch.mean(tensor_data)
29          std = torch.std(tensor_data)
30          return mean, std
```

Augmentation processes are used to increase the size of the dataset and to make the model robust. The augmentation techniques for images are explained in Section 2.2.2. Similar augmentation techniques are applied to the spectrograms. Augmentations are not applied to all the files instead based on the probability it is applied randomly on the files before giving it to model. Some augmentations come from the module augmentation from the predefined library Kornia, while others are custom made (Listing 4.4). Below is a summary of each custom augmentation class:

Listing 4.4: Custom data augmentation modules

```
1   class RandomAffineHorizontalScalePyTorch(torch.nn.Module):
2       def __init__(self, scale=1.1, p=0.5):
3           super().__init__()
4           self.scale = scale
5           self.p = p
6
7       def forward(self, img: torch.Tensor) -> torch.Tensor:
8           if torch.rand(1).item < self.p:
9               N, C, H, W = img.shape
10              theta = torch.tensor([[[self.scale, 0, 0],
11                                     [0, 1.0,    0]]] * N,
12                                   dtype=img.dtype,
13                                   device=img.device)
14              grid = F.affine_grid(theta, img.size(),
                    align_corners=False)
15              img = F.grid_sample(img, grid, align_corners=
                    False)
16          return img
17
18
19  class RandomAmplitudeScaling:
```

```
20          def __init__(self, scale_range=(0.9, 1.1)):
21              self.scale_range = scale_range
22
23          def __call__(self, tensor: torch.Tensor) -> torch.Tensor:
24              factor = torch.empty(1).uniform_(*self.scale_range).
                    item()
25              multiplier = torch.ones(tensor.size(2), device=tensor
                    .device) * factor
26              multiplier = multiplier.view(1, 1, -1, 1)
27              return tensor * multiplier
28
29
30      class RandomGaussianNoise:
31          def __init__(self, mean=0.0, std=1.0, p=0.5):
32              self.mean = mean
33              self.std = std
34              self.p = p
35
36          def __call__(self, tensor: torch.Tensor) -> torch.Tensor:
37              if torch.rand(1).item() < self.p:
38                  noise = torch.randn_like(tensor) * self.std +
                        self.mean
39                  tensor = tensor + noise
40              return tensor
```

Explaination of what and every augmentation does to the dataset is as follows:

- `RandomAffineHorizontalScalePyTorch`:
  Randomly stretches each spectrogram horizontally by $\pm 10\%$. In the demonstration (Figure 4.4), this warp reduces three peaks to two, making the effect clearly visible. This helps the model generalize to variations in footstep segment length when recordings are truncated or extended.

- `RandomAmplitudeScaling`:
  Multiplies the spectrogram tensor by a random factor in the interval $[0.9, 1.1]$, varying its amplitude by up to $\pm 10\%$ making the signals weaker or stronger. In the demo we used a fixed factor of 1.5, so the darker peaks in the output illustrate this boost in intensity. This accounts for footsteps with weaker or stronger signal strengths.

- `RandomGaussianNoise`:
  Injects additive Gaussian noise (with mean and standard deviation calculated from the dataset) into the spectrogram. The resulting graininess ("pixelation") in the visualization reflects this perturbation, improving robustness to sensor and environmental noise.

- `kornia.augmentation.Normalize [18]`:
  Standardizes the spectrogram tensor using its computed mean and standard deviation (Listing 4.3), ensuring a consistent dynamic range for model inputs. You can see the changed value range compared to the raw input.

- `kornia.augmentation.RandomHorizontalFlip [18]`:
  Flips the spectrogram horizontally, augmenting the dataset by mirroring temporal patterns so the model learns footstep patterns in both forward and reverse temporal order.

Each augmentation's application is gated by `if torch.rand(1).item() < self.p:` which cleanly compares a Python float to the probability `p`, ensuring a defined chance of applying each transform.

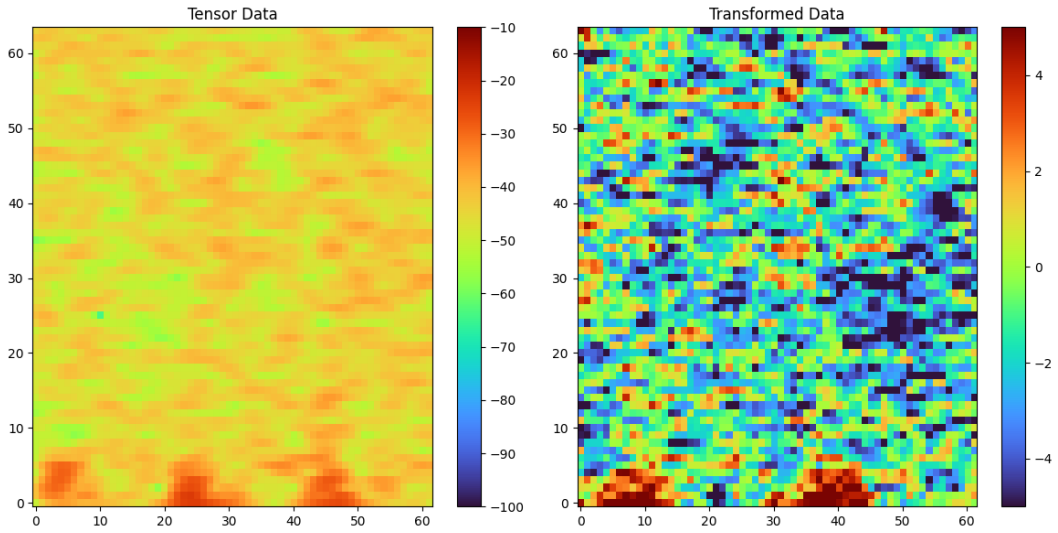


Figure 4.4: Augmentations of the input tensor

Figure 4.4 shows the implementation of the augmentation on the input tensor. Augmentations are enhanced for the sake of demonstration. Summarization of each augmentation applied on the data is as follows:

These transformations are applied in the custom dataset loader shown in Listing 4.5. PyTorch's recommended pattern is followed for creating a custom dataset from the documentation on PyTorch's website [13], which allows seamless integration with `DataLoader` and other utilities. The constructor takes a root directory containing subfolders named after each class (in our case, `walking` and `background`), and it builds internal lists of file paths and corresponding integer labels. The `__len__` method reports the total number of files (i.e. the number of samples per epoch), while `__getitem__` loads each file, applies the STFT-based processing, permutes axes to match PyTorch's `(C, H, W)` convention, and if a `transform` is provided it runs the custom augmentations before returning the `(tensor, label)` pair. This structure cleanly separates I/O, signal preprocessing, and augmentation, making the training loop straightforward.

Listing 4.5: Custom Dataset for loading and processing footstep files

```
1  class BinFileDataset(Dataset):
2      def __init__(self, root_dir: str, event_types: Dict[int,
           Union[str, List[str]]], transform=None):
3          self.root_dir = Path(root_dir)
4          self.transform = transform
5
6          # labels
7          self.event_types = event_types
8          self.class_to_idx = {cls_name: idx for idx, cls_name
               in event_types.items()}
9          self.file_list = []
10         self.labels = []
11
12         self.visualizer = Visualizer()
13         for event_idx, event_type in event_types.items():
14             subfolders = [event_type] if not isinstance(
                   event_type, list) else event_type
15             for subfolder in subfolders:
16                 cls_dir = self.root_dir / subfolder
17                 if not cls_dir.exists():
18                     raise ValueError(f"Subfolder {cls_dir}
                           does not exist.")
19                 for file_path in cls_dir.rglob("*"):
20                     if file_path.suffix in [".bin", ".npy"]:
21                         self.file_list.append(file_path)
```

```
22                              self.labels.append(event_idx)
23              if not self.file_list:
24                  raise ValueError(f"No data found in {self.
                        root_dir!r}.")
25
26          def __len__(self):
27              return len(self.file_list)
28
29          def __getitem__(self, idx):
30              file_path = self.file_list[idx]
31              label = int(self.labels[idx])
32              if label < 0 or label >= len(self.event_types):
33                  raise ValueError(f"Invalid label {label} for file
                        {file_path}.")
34
35              if file_path.suffix == ".bin":
36                  with open(file_path, 'rb') as f:
37                      data = np.fromfile(f, dtype=np.int64).reshape
                            ((length, channels))
38              elif file_path.suffix == ".npy":
39                  data = np.load(file_path)
40              else:
41                  raise ValueError(f"Unsupported file format: {
                        file_path.suffix}")
42
43              tensor_data = self.visualizer.data_process(data)
44              tensor_data = torch.from_numpy(tensor_data.astype(np.
                    float32)).contiguous()
45              tensor_data = tensor_data.permute(1,0,2).contiguous()
46
47              if self.transform:
48                  transformed_data = self.transform(tensor_data)
49                  return transformed_data, label
50              return tensor_data, label
```

The `BinFileDataset` class wraps the extracted data files into an iterable PyTorch dataset, making it easy to load and preprocess samples on the fly. In Listing 4.6, definition of two event types (`walking` and `background`) in the `event_types` dictionary is passed to `BinFileDataset` along with the root data path. Then splitting the resulting dataset into training and validation subsets using an 80/20

split; a `deepcopy` of the validation set ensures that training time augmentations do not leak into validation. All custom augmentations (Listing 4.4) plus normalization are applied to the training split, whereas the validation split only undergoes normalization with the precomputed mean and standard deviation. Each split is wrapped in a `DataLoader` (with specified `batch_size` and `num_workers`) to handle batching and parallel I/O.

For the model, we use the `timm` library to instantiate a backbone network (`efficient_b0.ra_in1k` or `convnextv2_atto.fcmae_ft_in1k`) with pretrained weights, specifying the correct number of output classes and input channels. To address class imbalance, we compute per class weights via `compute_class_weight` from scikit learn library and supply them to a weighted `CrossEntropyLoss`, which produces per sample losses. Finally, we optimize the model using the `Adam` optimizer (adaptive moment estimation) and log training metrics (loss, accuracy, etc.) to TensorBoard via `SummaryWriter`.

Listing 4.6: Dataset setup and model initialization

```
    # Define event labels
    event_types = {0: "background", 1: "walking"}

    # Create dataset and split
    dataset = BinFileDataset(root_dir='path/to/data', event_types
        =event_types)
    train_size = int(0.8 * len(dataset))
    val_size   = len(dataset) - train_size
    train_ds, val_ds = random_split(dataset, [train_size,
        val_size])
    val_ds = deepcopy(val_ds)

    # Assign transforms
    train_ds.dataset.transform = transforms.Compose([
        K.Normalize(mean=channel_means, std=channel_stds),
        K.RandomHorizontalFlip(p=1),
        RandomGaussianNoise(mean=0, std=1.5),
        RandomAmplitudeScaling((0.9, 1.1)),
        RandomAffineHorizontalScalePyTorch(scale=0.7, p=1),
    ])
    val_ds.dataset.transform = transforms.Compose([
        K.Normalize(mean=channel_means, std=channel_stds),
```

```
21      ])
22
23      # DataLoaders
24      train_loader = DataLoader(train_ds, batch_size=32,
            num_workers=8, shuffle=True)
25      val_loader   = DataLoader(val_ds,   batch_size=32,
            num_workers=8, shuffle=False)
26
27      # Model, loss & optimizer
28      model = timm.create_model("efficientnet_b0.ra_in1k",
            pretrained=True,
29                                num_classes=2, in_chans=1).to(
                                      device)
30
31      weights = compute_class_weight("balanced", classes=np.unique(
            all_labels), y=all_labels)
32      criterion = nn.CrossEntropyLoss(weight=torch.tensor(weights).
            to(device))
33      optimizer = optim.Adam(model.parameters(), lr=1e-4,
            weight_decay=1e-2)
```

The other important function used before training is `mixup_data`, which implements the MixUp augmentation strategy. It samples a mixing coefficient $\lambda \sim \mathrm{Beta}(\alpha, \alpha)$, shuffles each batch, and forms convex combinations of both inputs and labels. Listing 4.7 shows its implementation. During training, the loss is computed as

$$\mathcal{L} = \lambda\, \ell\Big(f(\tilde{x}), y_a\Big) + (1-\lambda)\, \ell\Big(f(\tilde{x}), y_b\Big),$$

encouraging smoother decision boundaries and improved generalization.

Listing 4.7: MixUp data augmentation function

```
1
2       def mixup_data(x, y, alpha=0.2):
3           if alpha > 0:
4               lam = np.random.beta(alpha, alpha)
5           else:
6               lam = 1.0
7           batch_size = x.size(0)
8           index = torch.randperm(batch_size).to(x.device)
```

```
9           mixed_x = lam * x + (1 - lam) * x[index]
10          y_a, y_b = y, y[index]
11          return mixed_x, y_a, y_b, lam
```

The training loop in Listing 4.8 runs for a fixed number of epochs. At the start of each epoch, the model is set to training mode and the running loss and accuracy meter are reset. For each mini-batch, inputs and labels are moved to the GPU, mixed via MixUp (producing `targets_a`, `targets_b`, and `lam`), and passed through the network. The combined MixUp loss is computed, backpropagated, and the Adam optimizer updates the weights. Accumulation of batch losses and updating the training accuracy meter is done, then computation and logging the average training loss and accuracy to TensorBoard.

After training, model is switched to evaluation and gradients are disabled. Looping over the validation loader, computing over the standard cross-entropy loss per batch, updating the validation accuracy meter, and accumulating losses is done. At epoch end, the average validation loss and accuracy are computed and logged. Once all epochs complete, a checkpoint containing the model's state dict, the optimizer's state dict, and the class-to-index mapping is saved for future use.

Listing 4.8: Core training and validation loop

```
1     def train(num_epochs=65):
2         train_acc = Accuracy(num_classes=2).to(device)
3         val_acc   = Accuracy(num_classes=2).to(device)
4         # Initialize optimizer, criterion, writer
5
6             for epoch in range(num_epochs):
7                 model.train()
8                 running_loss = 0.0
9                 train_acc.reset()
10
11                for inputs, labels in train_loader:
12                    inputs, labels = inputs.to(device), labels.to
                          (device)
13                    inputs, targets_a, targets_b, lam =
                          mixup_data(inputs, labels, alpha=0.2)
14
15                    optimizer.zero_grad()
16                    outputs = model(inputs)
```

```
17                      loss = lam*criterion(outputs, targets_a) \
18                           + (1-lam)*criterion(outputs, targets_b)
19                      loss.backward()
20                      optimizer.step()
21
22                      running_loss += loss.item()
23                      train_acc.update(outputs, labels)
24
25                  epoch_train_loss = running_loss / len(
                        train_loader)
26                  train_accuracy  = train_acc.compute().item()
27                  writer.add_scalar('Train/Loss',
                        epoch_train_loss, epoch)
28                  writer.add_scalar('Train/Accuracy',
                        train_accuracy,   epoch)
29
30                  model.eval()
31                  val_loss = 0.0
32                  val_acc.reset()
33
34                  with torch.no_grad():
35                      for inputs, labels in val_loader:
36                          inputs, labels = inputs.to(device),
                                labels.to(device)
37                          outputs = model(inputs)
38                          val_loss += criterion(outputs, labels).
                                item()
39                          val_acc.update(outputs, labels)
40
41                  epoch_val_loss = val_loss / len(val_loader.
                        dataset)
42                  val_accuracy   = val_acc.compute().item()
43                  writer.add_scalar('Val/Loss',     epoch_val_loss,
                        epoch)
44                  writer.add_scalar('Val/Accuracy', val_accuracy,
                         epoch)
45
46              torch.save({
47                  'model': model.state_dict(),
48                  'optimizer': optimizer.state_dict(),
49                  'class_to_idx': dataset.class_to_idx
```

```
50              }, 'checkpoint.pth')
```

Using the structure explained in the above listings the ConvNext V2 and EfficientNet models are trained using the 1 spatial channel and 10 spatial channel data. The logs which save Loss and Accuracy are monitored to see if the model are trained properly.

4.4.1 Testing of the model

The trained models have the checkpoints saved. The checkpoints have the weights saved for the model. The model is tested using the evaluation mode to see how the model performs. Listing 4.9 shows the core evaluation script used to assess model performance on held-out data.

Listing 4.9: Model test script

```
1     # Set random seed and load checkpoint
2     torch.manual_seed(42)
3     checkpoint = torch.load('effnet_with_preprocess_1d_v1.pth',
          weights_only=True)
4
5     # Prepare test dataset and loader
6     event_types = {0: "background", 1: "walking"}
7     transform = transforms.Compose([K.Normalize(mean=checkpoint['
          mean'], std=checkpoint['std'])])
8     test_ds = BinFileDataset(root_dir='path/to/test', event_types
          =event_types, transform=transform)
9     test_loader = DataLoader(test_ds, batch_size=32, num_workers
          =8, shuffle=False)
10
11    # Model setup
12    model = timm.create_model("efficientnet_b0.ra_in1k",
          pretrained=True,
13                              num_classes=2, in_chans=1)
14    model.load_state_dict(checkpoint['model_state_dict'])
15    device = torch.device('cuda' if torch.cuda.is_available()
          else 'cpu')
16    model.to(device).eval()
17
```

```
18      # Metrics and loss
19      criterion    = nn.CrossEntropyLoss(reduction='none')
20      test_acc     = Accuracy(task="multiclass", num_classes=2).to(
            device)
21      test_confmat = ConfusionMatrix(task="multiclass", num_classes
            =2).to(device)
22
23      # test loop
24      test_loss = 0.0
25      samples   = 0
26      with torch.no_grad():
27          test_acc.reset()
28          test_confmat.reset()
29          for inputs, labels in test_loader:
30              inputs, labels = inputs.to(device), labels.to(device)
31              inputs = inputs.permute(0,2,3,1,4).squeeze(3)
32              outputs = model(inputs)
33              test_loss += criterion(outputs, labels).sum().item()
34              samples   += labels.size(0)
35              preds = predict_with_threshold(outputs, 0.6)
36              test_acc.update(preds, labels)
37              test_confmat.update(preds, labels)
38
39      # Final metrics
40      avg_loss = test_loss / samples
41      acc      = test_acc.compute().item()
42      conf_mat = test_confmat.compute().cpu().numpy()
43
44      print(f"Test Loss: {avg_loss:.4f}, Accuracy: {acc*100:.2f}%")
45      print(f"Confusion Matrix:\n{conf_mat}")
```

In first step the random seed of reproducibility which is used for setting the random numbers. Setting the seed ensures that all subsequent calls to PyTorch's random number generator produce the same sequences, making the evaluation fully reproducible. The saved checkpoint is loaded, which contains the model weights as well as the normalization statistics (mean and standard deviation). The test dataset is constructed using the `BinFileDataset` from the listing 4.5 where the location for the test set and normalizing using the parameters from the checkpoint. A `DataLoader` wraps this dataset with a batch size of 32 and

multiple worker processes for parallel I/O.

Next, the model is setup to (`efficientnet_b0.ra_in1k` or `convnextv2_atto.fcmae_ft_in1k`) using the `timm.create_model` and setting the channels and classes are set with the same numbers. The model is moved to evaluation mode and the loss function is initializated using `CrossEntropyLoss` and key metrics like accuracy and confusion matrix are taken from `torchmetrics` [33] necessary to evaluate the performance of the model.

The evaluation loop (with gradients disabled) mirrors the validation phase from Listing 4.8. We iterate over the test loader, compute the standard cross-entropy loss for each batch, and update metrics such as accuracy and the confusion matrix to assess model performance. Finally, we apply the `predict_with_threshold` function which returns "walking" only if the activity probability exceeds a chosen cutoff and otherwise returns "background" to convert soft outputs into discrete class labels.

4.4.2 Comparison of training, validation and testing performance

The models are trained and tested with scripts and datasets as seen in the previous sections. The tensorboard logging is used to log the graphs of loss and accuracy. The four different types of checkpoints, which are trained and evaluated on the respective dataset. Ideally the loss should be around 0 and accuracy should be near 100%.

The graphs 4.5 show the loss and accuracy plots for training and validation on dataset-1D of ConvNext V2 model. The training is done over 65 epochs. Training loss (blue) starts at 0.37 which falls rapidly after first 10 epochs and oscillates between 0.18 and 0.25. Validation loss (orange) starts at 0.21 and steadily declines to around 0.12 indicating model continues to improve on unseen data even after traing loss plateaus. Training accuracy (blue) climbs from around 68% to 80% but shows some fluctuations once loss stabilizes. Validation accuracy (orange) starts at around 92% and rises to 95% remaining consistently high demonstarting good generalization.

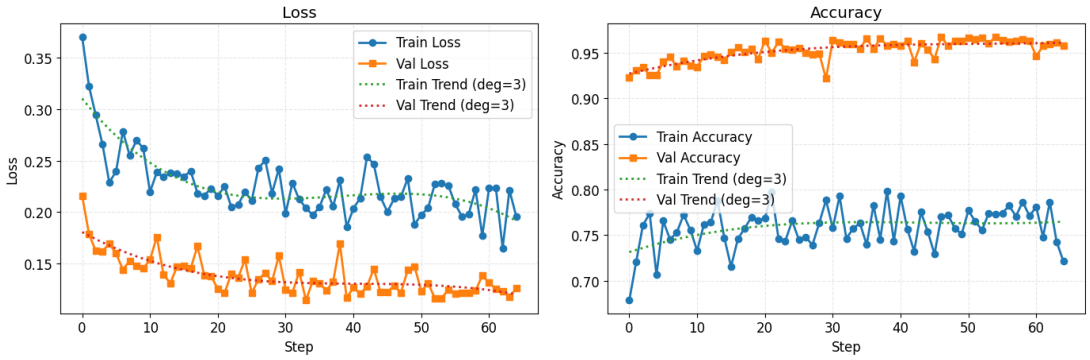


Figure 4.5: Loss and Accuracy plots for training and validation for ConvNext V2 on dataset-1D

Table 4.1 summarizes the ConvNext model's performance on the dataset-1D. The overall test loss of 0.36 and accuracy of 87.55% indicate that the model generalizes well to held-out examples. From the confusion matrix (lower block of the table), we see that of 1591 true background samples the model correctly labels 1454 (91.4% recall) and of 987 true walking events it correctly labels 803 (81.4% recall). Precision scores 88.8% for background and 85.4% for walking confirming the model makes relatively few false alarms in each class.

Table 4.1: Test results for the ConvNext V2 model on the dataset-1D

Metric	**Background**	**Walking**
Test Loss	0.3625	
Test Accuracy (%)	87.55	
Confusion Matrix		
True Background	1454	137
True Walking	184	803
Per-class Precision, Recall, F1 (%)		
Precision	88.77	85.43
Recall	91.39	81.36
F1 Score	90.06	83.34

The graphs 4.6 show the loss and accuracy plots for training and validation of EfficientNet model on dataset-1D. Over 65 epochs, training is done. The train loss (blue) starts at 3.05 and gradually drops till it reaches 0.281. The validation loss (orange) starts at around 2 and gradually drops till it reaches 0.18. The minor spike is observed around 30th epoch but recover quickly. The train accuracy

(blue) rises from 64% to 75% reflecting incremental learning. The validation accuracy (orange) starts at 72% and rises till 90% while plateauing between 88% and 93%. The learning summarizes good generalization.

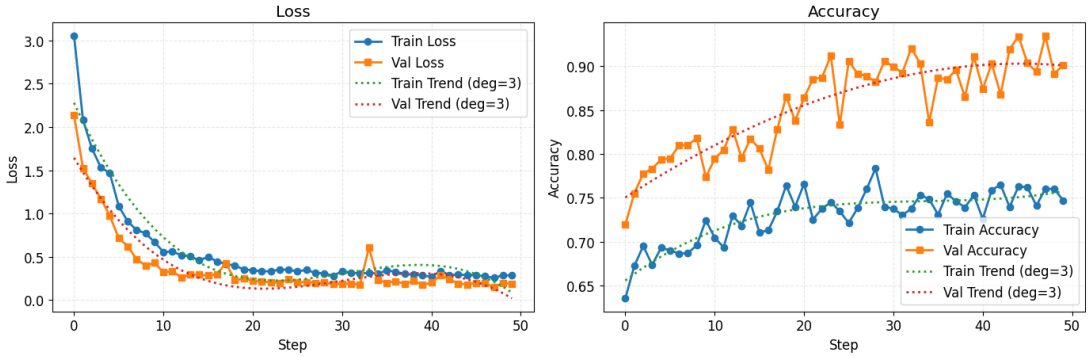


Figure 4.6: Loss and Accuracy plots for training and validation for EfficientNet on dataset-1D

Table 4.2: Test results for the EfficientNet model on dataset-1D

Metric	**Background**	**Walking**
Test Loss	0.2496	
Test Accuracy (%)	91.89	
Confusion Matrix		
True Background	1512	79
True Walking	130	857
Per-class Precision, Recall, F1 (%)		
Precision	92.08	91.56
Recall	95.03	86.83
F1 Score	93.54	89.13

Table 4.2 reports the EfficientNet models performance on the dataset-1D. The model achieves a test loss of 0.25 and overall accuracy of 91.89%, demonstrating strong generalization. Examining the confusion matrix, 95.0% of true background samples (1512 out of 1591) are correctly identified, while 86.8% of true walking events (857 out of 987) are detected. Precision remains high for both classes 92.1% for background and 91.6% for walking indicating few false alarms. Finally, F1 scores of 93.5% and 89.1% reflect a solid balance between precision and recall in each category.

The graph 4.7 shows the training and validation loss and accuracy plots of Con-

vNext V2 model on dataset-2D. The train loss (blue) and validation loss (orange) declines steeply where train loss drops from 0.56 to 0.20 at epoch 3 and validation loss plunges even faster from 0.88 to 0.06 at epoch 2. Train loss sees occasianl small upticks at epoch 5 but stablizes later. Validation loss remians low with tiny spikes at around epochs 11 and 21. Train accuracy (blue) starts around 57% and reaches 85% at epoch 10 then osscilates between 82%-87%. Validation accuracy (orange) skyrockets to 99% at epoch 3 and stay high with some minor dips at around at epoch 11 and 21 suggesting good learning.

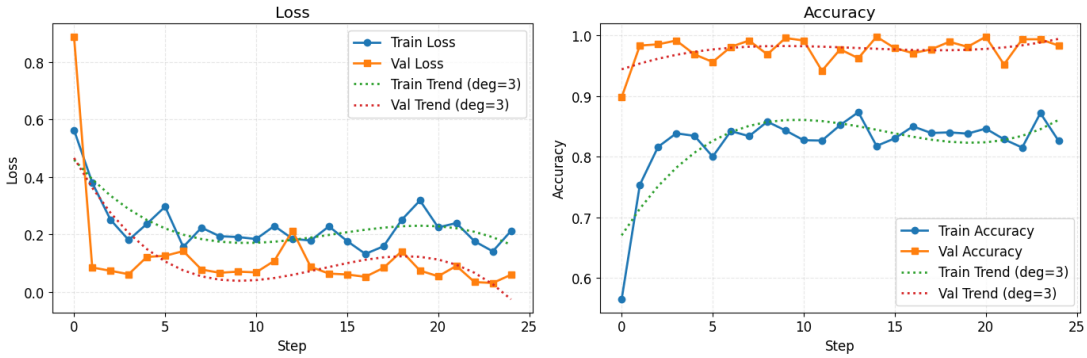


Figure 4.7: Loss and Accuracy plots for training and validation for ConvNext V2 with 10 spatial channels and 2 sample length

Table 4.3: Test results for the ConvNeXt V2 model on the dataset-2D

Metric	**Background**	**Walking**
Test Loss	0.0537	
Test Accuracy (%)	99.48	
Confusion Matrix		
True Background	1678	4
True Walking	5	58
Per-class Precision, Recall, F1 (%)		
Precision	99.70	93.55
Recall	99.76	92.06
F1 Score	99.73	92.80

Table 4.4 presents the ConvNext V2 model's performance on the dataset-2D. The extremely low test loss of 0.0537 and very high overall accuracy of 99.48% show that the model almost perfectly separates footsteps from background. In the confusion matrix, 1678 out of 1682 background samples are correctly classi-

fied (99.8% recall), and 58 out of 63 walking events are detected (92.1% recall). Precision remains excellent 99.7% for background and 93.6% for walking indicating virtually no false alarms in the background class and very few in the walking class. The resulting F1 scores (99.7% for background, 92.8% for walking) confirm a close to ideal balance of precision and recall.

The graph 4.8 shows the train loss and accuracy set for ConvNeXt V2 on dataset-2D. The model runs over 25 epochs. The train loss (blue) and validation loss (orange) have a rapid intial drops over first 3-4 epochs indicating model picks up on low-level features. After that decline is more gradual and settles at 0.2 to 0.3. The sharp spike at epochs 11-12 suggesting model destabilzes then model recovers. Train accuracy (blue) rises from 63% to 80% and pleateauing at the point. Validation accuracy (orange) jumps from 66% to 91% which dips at epoch 4 to 58% then agian recovers. The graph demonstarates the good genralization over the data.

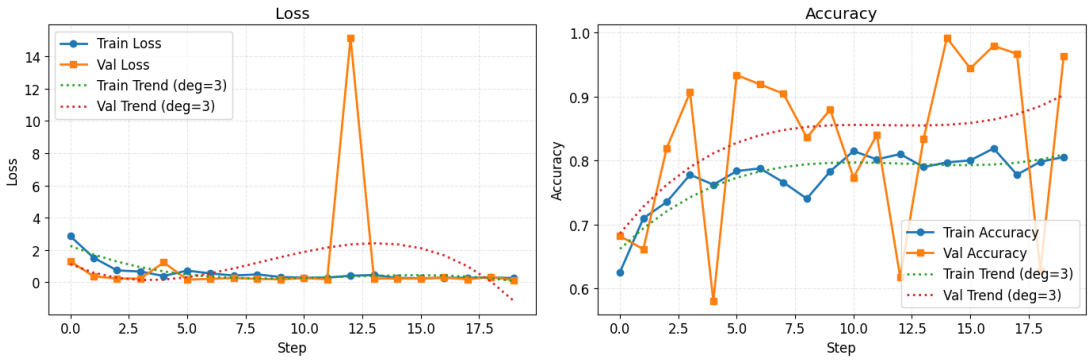


Figure 4.8: Loss and Accuracy plots for training and validation for EfficientNet with 10 spatial channels and 2 sample length

Table 4.4 summarizes the EfficientNet model's performance on dataset-2D With a test loss of only 0.934 and an overall accuracy of 99.48%, the network distinguishes footsteps from background almost flawlessly. The confusion matrix shows that 1679 out of 1682 background samples are correctly identified (99.8% recall), while 57 out of 63 walking events are detected (90.5% recall). High precision values 99.6% for background and 95.0% for walking indicate very few false positives. Finally, F1 scores of 99.7% and 92.7% confirm an excellent balance between precision and recall for each class.

Table 4.4: Test results for the EfficientNet model on the 2-D dataset

Metric	Background	Walking
Test Loss	0.0934	
Test Accuracy (%)	99.48	
Confusion Matrix		
True Background	1679	3
True Walking	6	57
Per-class Precision, Recall, F1 (%)		
Precision	99.64	95.00
Recall	99.82	90.48
F1 Score	99.73	92.68

The metrics in the graphs suggests that models shows the similar performance over the testing set for the same models training set. Except the ConvNext V2 model on dataset-1D, all the other show similar performance and are better than ConvNext V2 model on dataset-1D. The checkpoints should be evaluated on real-time across the data stream coming in through the DAS configurator application. Model should show similar performance as the performance on testing set.

4.5 Real-time evaluation Framework

For real time evaluation of the model, the data needs to be streamed to the model and based on the data the model is able to classify the data. This is done with the help of `spectrogram_classifier` module which is used for the real-time evaluation and developed with the help of `dasdpu` package. The `dasdpu` package is an internal package developed at AP Sensing for running and testing algorithms on DAS DPU or locally on HDF5 files. It is used for socket reading of Frequency Band Energy (FBE) and/or phases data. It also sends the events or alarms to the DPU such that are displayed in the Configurator. This package is one of the most important elements in the real-time evaluation script.

Spectrogram Classifier (`spectrogram_classifier`) is the module developed internally at AP Sensing which is used for evaluation of the models. The module

implements the online execution of the logic for the phase-based footstep detection. It ingests the raw phase samples from the DAS-acquisition service, buffers a short user-defined time window, and optionally computes the spectrograms for each channel before forwarding them to the trained model. When the model's output probability exceeds the user-set threshold, the pipeline emits a detection. The detections are tracked over time by grouping and assigning them track IDs, thus allowing the tracking of a moving disturbance along the cable. Once a track accumulates sufficient repeated detections (confidence), the alarm is raised. Figure 4.9 gives an overview of the structure of the `spectrogram_classifier` module and it's dependencies to all other packages. All the standard python libraries on which the module is dependant on are excluded. The dependencies with `dasdpu` and other packages is shown in the figure 4.9. The `system.py` is the file which which has all the logic files essential for the working of the module and is the file where all the changed files are imported.

The `spectrogram_classifier.execution.aggregator` and `spectrogram_classifier.execution.detector` are the top level files responsible for the execution. The `spectrogram_classifier.logic.inference` returns the list of detections from the tensor. Position in terms of distance is mapped from `spectrogram_classifier.logic.position` from the spatial channels. All the remaining files are not crucial for detection but serve as the support and enables in smooth running of the system. The `spectrogram_classifier` module is edited to adjust for the preprocessing steps done while training the model and models ConvNext V2 and EfficientNet are also added in the pipeline for easy selection and switching. The preprocessing section is inserted in the `spectrogram_classifier.logic.preprocessing` which is same as the training. The model is inserted in the `spectrogram_classifier.model` is also same as the training.

Listing 4.10: Preprocessing for conversion of raw phase data to spectrogram in the framework

```
1 class SpectrogramPreprocessorStft:
2     def __init__(self, model_params: Parameters, data_rate_hz:
          int = 1000, device: torch.device = torch.device("cuda")):
3         self._data_rate_hz = data_rate_hz
4         self._device = device
5         self._multiplier = data_rate_hz // 500
6         self._nperseg = 128 * self._multiplier
```

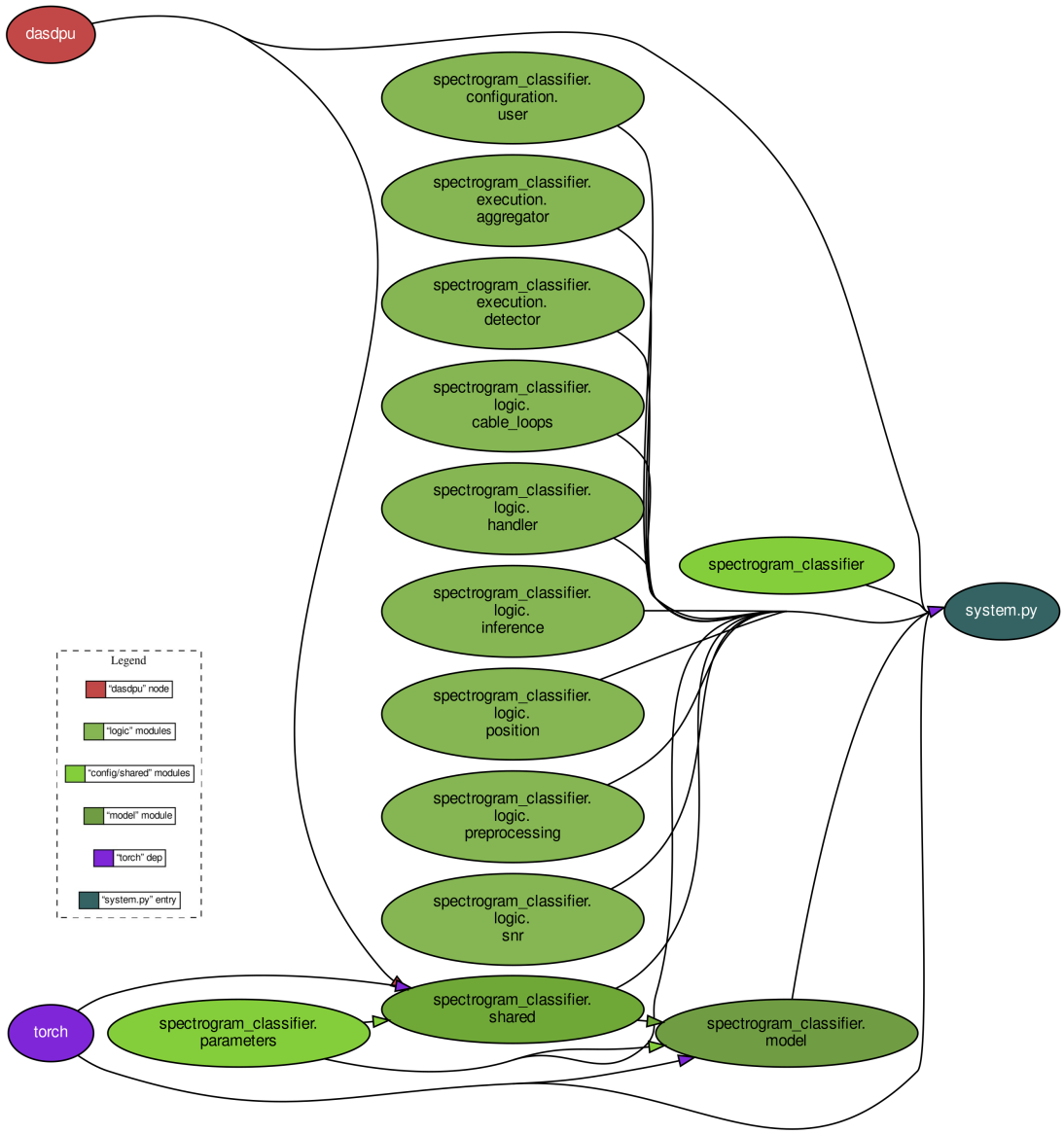


Figure 4.9: Dependency Graph of `spectrogram_classifier` module

```
7           self._noverlap = self._nperseg - 14 * self._multiplier
8           self._f_bank = (
9               melscale_fbanks(
10                  n_freqs=self._nperseg // 2 + 1,
11                  f_min=model_params.f_min,
12                  f_max=model_params.f_max,
13                  n_mels=model_params.n_mels,
14                  sample_rate=data_rate_hz,
15                  mel_scale="htk",
```

```
16                 norm=None,
17             )
18             .cpu()
19             .numpy()
20         )
21
22     def preprocess(self, data: np.ndarray, axis: int) -> tuple[np
           .ndarray, np.ndarray]:
23         data_accum = self._accum(data, axis=axis)
24         f, t, stft = signal.stft(
25             data_accum,
26             fs=self._data_rate_hz,
27             nperseg=self._nperseg,
28             noverlap=self._noverlap,
29             padded=False,
30             scaling="psd",
31             axis=axis,
32         )
33         stft = np.abs(stft) ** 2
34         stft = np.matmul(stft.transpose((0, 2, 1)), self._f_bank)
               .transpose((0, 2, 1))
35         mag = 10 * np.log10(stft)
36         return
37
38     def process(self, tensor_data: torch.Tensor) -> torch.Tensor:
39         tensor_numpy = tensor_data.cpu().numpy()
40         numpy_result = self.data_process(tensor_numpy, axis=-1)
41         return torch.from_numpy(numpy_result).float().cuda()
42
43     def _accum(self, x: np.ndarray, axis: int) -> np.ndarray:
44         x = np.cumsum(x, axis=axis, dtype=np.int64).astype(np.
               float32) * INT_TO_RADIANS
45         sos = signal.butter(2, 5, "hp", fs=1000, output="sos")
46         x = signal.sosfiltfilt(sos, x, axis=axis)
47         return x
48
49     @property
50     def device(self) -> torch.device:
51         return self._device
```

The listing 4.10 shows the processing which is same as the listing in the section

listing 4.2. The main difference is that data is in the form a tensor in listing 4.10 so it first needs to be converted into a numpy for the STFT function which is used in preprocessing in training pipeline. After the STFT function in `preprocess` the shape needs to be considered as the streamed data for realtime since it needs to be in the format `[batch, frequency_bins, time]` in comparison to training where the shape is adjusted after extracting the data in listing 4.5. The `_accum` function has the cumulative sum where it is done on the time axis as the data distorts and the model will not be able to do the detections.

Listing 4.11: Model evaluation mode in the framework

```
def get_effnet_b0(params: Parameters) -> torch.nn.Module:
    return timm.create_model("efficientnet_b0", num_classes=
        params.n_classes, in_chans=params.model_n_channels)

def get_convnextv2(params: Parameters) -> torch.nn.Module:
    return timm.create_model("convnextv2_atto", num_classes=
        params.n_classes, in_chans=params.model_n_channels)

def create(cls, weights_path: str | Path, device=torch.device("
    cpu")) -> Model:

    model_data = torch.load(weights_path)
    state_dict = model_data["model_state_dict"]
    mean = model_data["mean"]
    std = model_data["std"]

    params = Parameters(
        epochs=100,
        batch_size=32,
        test_batch_size=32,
        learning_rate=1e-3,
        weight_decay=1e-4,
        dataset_mean=mean.reshape((1, 1, 1, 1)),
        dataset_std=std.reshape((1, 1, 1, 1)),
        label_smoothing=0.1,
        accum_phase=False,
        highpass_cutoff_hz=None,
        expected_data_rate_hz=1000,
        n_fft=512,
        hop_size=256,
```

```
28          pad=0,
29          center=True,
30          mel_scale=False,
31          n_mels=64,
32          f_min=0,
33          f_max=500,
34          model_name="effnet-b0",
35          model_d=64,
36          n_classes=2,
37          model_n_channels=10,
38          normalization="global",
39      )
40
41      model = create_model(params, state_dict)
42      model = TimmModelWrapper(model, mean=mean, std=std)
43      model = model.eval()
44
45      return cls(model, device, params)
```

The listing 4.11 shows the evaluation mode for ConvNext V2 and EfficientNet. These functions are used in the top layer of the model where the parameters like classes and input channels are given to the functions from parameters that are used. The parameters used in the model are set to the same values which are used while training. The mean and standard deviation calculated during the training ans stored in the checkpoint are extracted and used for normalization of data. The model is created using `create_model` function which calls either `get_convnextv2` or `get_effnet_b0` based on the model name. The model is wrapped in `TimmModelWrapper` which is used to adjust the shape and set the normalization parameters. The model is then set to evaluation mode for returning detections

5 Evaluation

In this chapter, assessment of trained models is done on truly unseen data-recordings captured via the DAS configurator but never used during training or validation. The steps such as employing a real-time processing script that streams raw phase data, applies the same preprocessing pipeline, and loads the saved model checkpoints for footstep detection. The evaluation is done using the `spectrogram_classifier` framework which is explained in the section 4.5. Finally, comparison of the performance of different architectures and concise analysis of their relative strengths and weaknesses will be given.

5.1 Real-time Evaluation of the Model

Evaluation of the model is performed by loading the data recordings from the DAS configurator application and streaming the data to the model using the `spectrogram_classifier` framework. Framework is designed to handle live data streams and give out the detection on the DAS configurator application. Evaluation also needs the checkpoint files of the trained models. Sample length(seconds) and number of spatial channels need to be configured based on the checkpoint files used as the model is trained on two different types of dataset.

Listing 5.1 shows the `YAML` configuration file [34] for the real-time evaluation framework. The `window_len_secs` and `hop_len_secs` parameters together define the duration of each analysis frame and the time shift between successive frames, respectively, controlling how the continuous DAS stream is chopped into overlapping segments for spectrogram generation. In the activities section identifiers are assigned used by the DAS Configurator to label detections, while the `zones` section defines spatial intervals via `start_m` and `end_m` within which the classifier is active. Finally, `min_prob` sets the minimum confidence threshold for

an event to be reported, filtering out low-probability (and likely false-positive) detections. Careful tuning of these settings is key to achieving efficient, reliable performance.

The `device` parameter specifies where the model is loaded (e.g., `cuda` or `cpu`), and `dtype` sets the tensor data type used throughout processing. Incoming DAS data are buffered for up to `max_age_sec` seconds, and once the model reports at least `min_confidence` detections within that window, an alarm is issued. To control memory use, the framework processes data in fixed-size blocks of channels, as defined by `n_channels_per_block`, which should match the host's available RAM or GPU capacity. Enabling `save_debug_data` will write every raw detection to `.npy` files for offline inspection. Each channel is classified as either walking or background. Consecutive "walking" samples that span at least `min_size_m` meters trigger an event, and if that event continues into the next time step within `max_distance_m` meters, it is merged and its confidence count increments. Since we do not apply any SNR-based filtering, `min_snr` is fixed to zero. Finally, `tensor_idx` selects which output index corresponds to each class (e.g. 1 for walking, 0 for background). All of these parameters remain fixed during evaluation to ensure a consistent runtime behavior.

Listing 5.1: YAML configuration for the real-time DAS evaluation framework

```yaml
window_len_secs: 2
hop_len_secs: 1
device: cuda
dtype: float16
n_channels_per_block: 500
save_debug_data: true
activities:
  person_walking:
    max_age_sec: 10
    min_confidence: 4
    min_size_m: 0
    max_distance_m: 15
    min_snr: 0
    tensor_idx: 1
    zones:
      - start_m: none
        end_m: none
```

```
18          min_prob: 0.92
```

For our evaluation, we recorded a straight-line footstep trail of approximately one minute's continuous walking about 45 individual steps performed by a different person than those in the training set to test across subject generalization. The walker maintained a steady pace without turns or pauses along the same fiber route used for training, isolating pure step-to-step detection performance. Because our models were trained on very short clips (2-5 footsteps), evaluating them on this longer sequence stresses the pipeline's temporal grouping logic. All detections and alarms were logged within a 75 m zone of interest and then compared against the known ground-truth step positions to measure both spatial and temporal accuracy. We also verified the pipeline under both slower and faster gait speeds and observed very similar spatial temporal detection patterns; accordingly, our detailed results focus on the steady-pace walking described above.

5.1.1 ConvNext V2 Model evaluation

The ConvNext V2 model is trained on two different datasets - Dataset-1D and Dataset-2D as explained in section 4.3. There are two checkpoints for ConvNext V2 model for both datasets. Using these checkpoints the evaluation is done for over the data collected.

The checkpoint file for dataset-1D is loaded into the framework for evaluation. The `window_len_secs` is set to 1.728 seconds and `hop_len_secs` is set to 1 second. Zone is set between 600 to 675 meters to focus on the area as too many detections are generated if the entire files given and application is not able to handle them. Threshold is set to 0.9 to remove the false positives. Figure 5.1 shows the detections after running the evaluation framework on the dataset-1D.

The detections are shown with the help of purple bubbles. The smaller bubbles are events and bigger bubbles are alarms. As seen in the figure 5.1 there are around 70 detections which shows model is able to detect footsteps but also on the neighboring spatial channels. This shows detections is not perfect as the detections does not align with the footsteps and exact position cannot be determined.

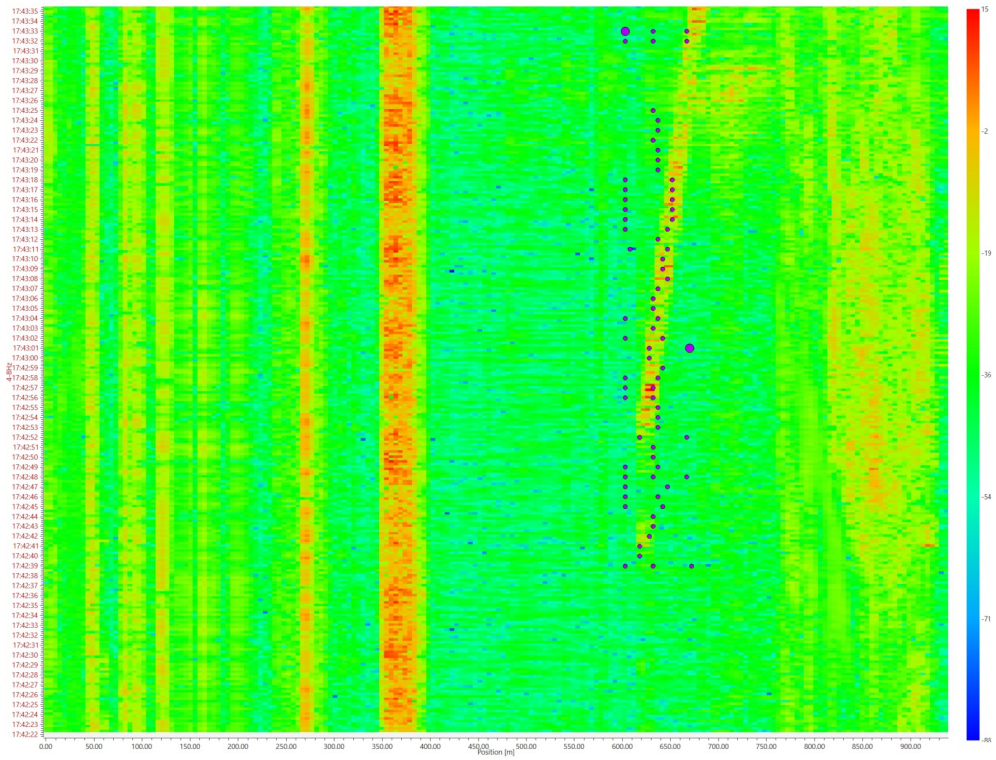


Figure 5.1: DAS Configurator Application detections for ConvNext V2 model on dataset-1D

Now the checkpoint file for dataset-2D is loaded into the framework for evaluation. The `window_len_secs` and `hop_len_secs` are set to 2 seconds and 1 seconds respectively. Rest of the other parameters are kept the same. Figure 5.2 shows the around 35 detections when evaluation framework is ran. The major difference from dataset-1D is that the footstep trail is more clear and there are lot less false positives with detections more aligned to the actual footstep trail.

From the accuracies over the respective datasets in section 4.4.2 it was evident that the model with dataset-2D is better than the dataset-1D as the accuracies were 87% and 99% respectively. The false positives are much lesser and the footsteps are more accurate in dataset-2D.

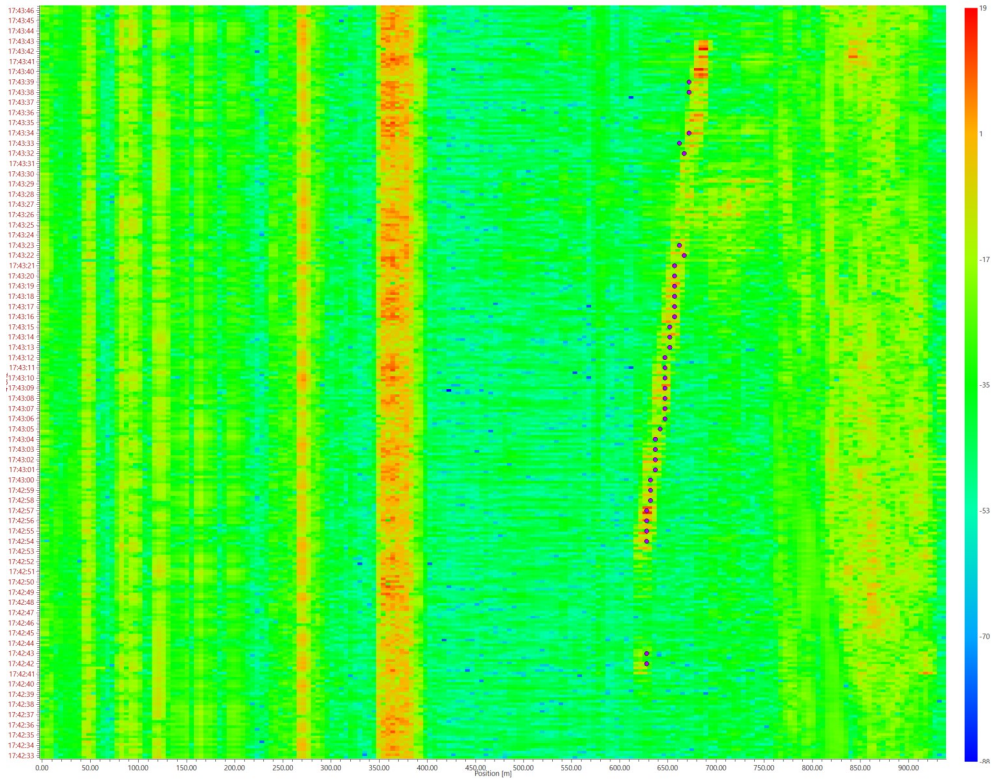


Figure 5.2: DAS Configurator Application detections for ConvNext V2 model on dataset-2D

5.1.2 EfficientNet Model evaluation

EfficientNet Model is also trained in a similar fashion as ConvNext V2 model with dataset-1D and dataset-2D. Models with their respective checkpoints are evaluated by keep The EfficientNet model is selected in the framework in listing 4.11 and the number channels are also set according to the checkpoint file.

The checkpoint file for dataset-1D is loaded into the framework nad the parameters are setup. The `window_len_secs` is set to 1.728 seconds and `hop_len_secs` is set to 1 second. Zone is selected the same as ConvNext V2 model which is between 600 to 675 meters to focus on a specific region. Threshold value is also kept same to 0.9 to make sure that only the detections with utmost certainty are recorded.

Figure 5.3 shows the about 70 detections by the model on dataset-1D. Detections are shown with purple bubbles with smaller ones as the events and bigger ones

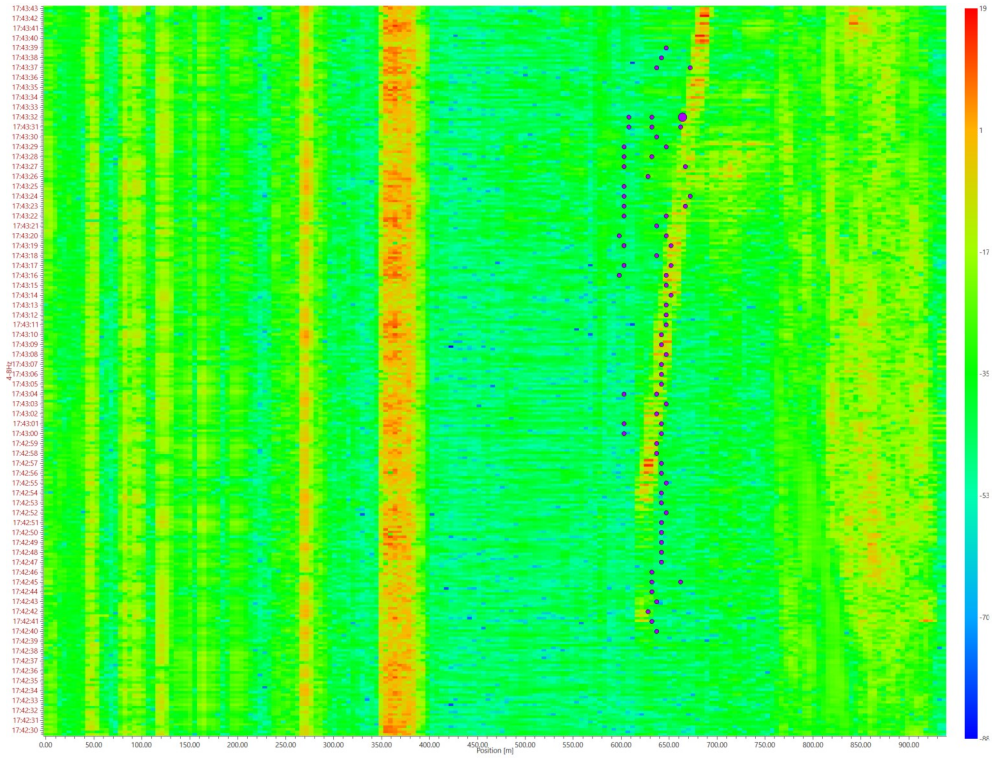


Figure 5.3: DAS Configurator Application detections for EfficientNet V2 model on dataset-1D

as alarms. The detections are similar to the ConvNext V2 model for dataset-1D. In a similar fashion, the model is able to detect the trail but with it there are lot of false positives on neighboring spatial channel. The exact position of footsteps is not determinable.

Dataset-2D checkpoint file is loaded into the framework and parameters remain the same. Spatial channels are set to 10. The `window_len_secs` is et to 2 seconds and `hop_len_secs` is et to 1 seconds. Zone is kept the same to focus on 600 to 675 meters. Threshold is also kept same at 0.9. Figure 5.4 shows about 35 detections made when the evaluation framework is ran. The detections are much more accurate and focused on the footstep trail. Model performs similarly to the ConvNext V2 model on dataset-2D.

The accuracies for the EfficientNet model for dataset-1D and dataset-2D are same (99.43%). Even though the accuracies are same the detections in dataset-2D model are much more accurate and lesser False positives.

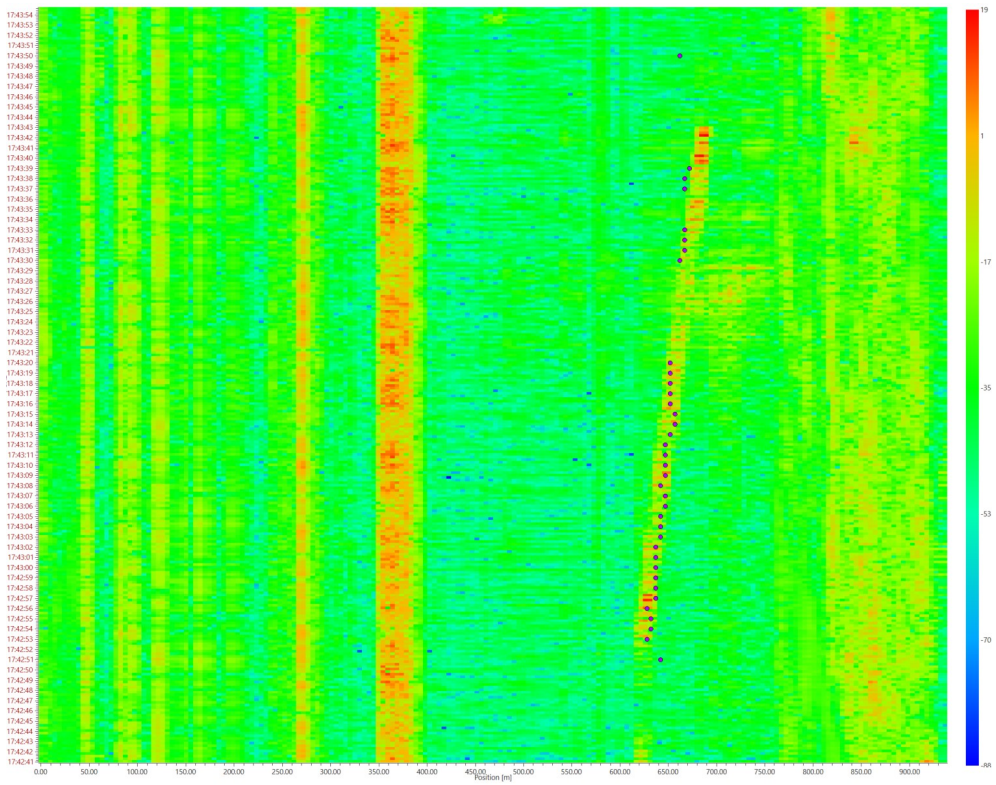


Figure 5.4: DAS Configurator Application detections for EfficientNet V2 model on dataset-2D

5.2 Comparison of the Model Performances

The models are evaluated over the framework for both the models and for the checkpoints trained over dataset-1D and dataset-2D in subsections 5.1.1 and 5.1.2. The dataset-1D shows about 70 detections and dataset-2D shows about 35 detections. It is evident that the models trained over dataset-2D are performing much better than models trained over dataset-1D with lot less false positives and better localization of footsteps. Even though the accuracies are quite similar for almost every model trained except for the ConvNext v2 model trained over dataset-1D which 87% this, the dataset-2D models are performing much better.

The reason for this can be explained the model is trained over dataset-2D has the data with 10 spatial channels because of which the model is able to learn the features of the channel where the detections are made and the model is able to distinguish between the channels. The model focuses on the channel where the

footsteps distortions are maximum and not give out the false positives which are observed in the neighboring channels. Same cannot be said for the dataset-1D trained models as the model is trained with only 1 spatial channel. The model doesn't know to distinguish between the channels and gives out the detections when it observes the distortions. The false positives can be much higher if the zone is not fixed and the entire file is given to the model. This can lead to too many detections and application is not able to handle them. Also, if there are too many false positives it becomes difficult to find the actual footstep and thus not able to pin point to the exact location. Each footstep is not detected and model is not trained over very short clips of 3 to 5 footsteps.

Thus from the evaluation results, it is clear that the models trained over 10 spatial channels are preforming better than the models trained over 1 spatial channel irrespective of the model selection.

All results above are evaluated over the 600-675 m zone, where noisy regions are absent. In Figures 5.1, 5.2, 5.3, and 5.4, the 350-400 m segment (highlighted in red) exhibits elevated background noise, leading to random false positive detections. Raising the confidence threshold (`min_prob`) reduces these spurious alarms but risks missing faint footsteps, especially outside the clean zone.

6 Conclusion

In this thesis, a robust system to train the model over different datasets and evaluation of those models over the streamed data from DAS configurator application is explained. The data is collected from a set experimental setup location (Energy Building) and different models such as ConvNext V2 and EfficientNet models are selected for same. Data contains the footsteps and dataset was created by extracting the footsteps from the raw data. Two different types of datasets were created, one with 1 spatial channel and 1.728 seconds of sample length and other with 10 spatial channels and 2 seconds. Dataset contains two labels one for background noise and other for footsteps. Data from the datasets needs to be preprocessed before training the models.

Prepocessing steps included the following:

- `Spectrogram Generation:` The raw phase data is transformed into spectrograms using the STFT function. Spectrograms shows the clear spikes for the footsteps making it easier for the model to learn patterns.
- `Normalization:` Spectrogram tensors are normalized by calculating the mean and standard deviation over the entire dataset. It helps in stabilizing the training and ensuring the smooth convergence of model.
- `Data Augmentation:` Different data augmentation techniques such as horizontal flipping, amplitude scaling, horizontal stretching and addition of gaussian noise were used. This helps in increasing the diversity in dataset and improving generalization for the model.

Using the preprocessed data, the model are trained and tested over the different datasets. The training and testing results are analyzed and compared. For the ConvNext V2 model, the testing accuracies are 87.55% and 99.48% for 1 spatial channel and 10 spatial channel datasets respectively. The losses for the same

model are 0.36 and 0.05 respectively. For the EfficientNet model, the testing accuracies are same at 99.43% for 1 spatial channel and 10 spatial channel datasets respectively. The losses are also same at 0.09 for both dataset. The results show the similar performance for both models on the 10 spatial channel dataset while EfficientNet model performs better on 1 spatial channel dataset.

The trained models are evaluated over the unknown data and the similar performance is seen on the streamed data from the DAS configurator application. Although a precise overall accuracy cannot be computed in real time, the models reliably highlight the footstep trail, mirroring their test dataset results. It was observed that the 10 spatial channel models are performing better than the 1 spatial channel models as it gives a lot lesser false positives. Detections are a lot focused on 10 spatial channel dataset as the model is able to learn patterns from multiple channels more effectively.

6.1 Future Work

Model training is an iterative process to keep improving the performance of the model. The model is trained and tested over the experimental setup proposed. Thus there are several areas which will be looked upon to improve the performance of the model.

6.1.1 Improving the Dataset and Training the Model

After evaluating the model, there are lot of false positives in the detections especially in the noisy zones. These can be adjusted by increasing the threshold but there might be chance of fainter footsteps not being detected. The evaluation framework stores the detections as .npy files. False positives can be picked from them manually and added to the dataset. The background noise which is detected as footsteps are added to the dataset. Using the updated dataset, the model can be trained which helps model to learn the patterns more effectively and improve the performance of the model thus improving the accuracy and reducing the false positives.

This process can be repeated iteratively to keep improving the performance of the model. To generalize the model, the dataset can be extended by adding more data from different locations and different people. This will help in improving the performance of the model.

6.1.2 Addition of different activities

The current model is used to detect footsteps by distinguishing them from background noise. The model can be extended to detect different activities such as fence cutting, climbing and digging. Dataset can be created for these activities and model can be trained on them. Using the model trained on these activities, the model can be used for perimeter monitoring.

Perimeter monitoring can be used in various locations such as airports, borders and national parks. Different activities based on the location can be added to the model. These models can be deployed to the locations and can be used for real time monitoring of the activities.

6.2 Perspective

In the thesis, a demonstration of the viability of the phase based footstep detection using spectrogram classifiers on both single and multi spatial channel DAS data, it opens broader possibilities for future avenue. One direction is to increase the robustness of the model by training it repeatedly by improving the dataset with the help of evaluation framework.

Second direction is to extend the model to detect different activities which can be used for perimeter monitoring and surveillance. The model can be used for the location specific activities and can be served for the needs of the location.

Together, these extensions will move from the proof of concept to a more general, scalable DAS based perimeter monitoring platform.

Bibliography

[1] AP Sensing Marketing Hub, "Gpec.png." Social media image/template stored in the Marketing Hub SharePoint workspace. Accessed: 2023-10-05.

[2] AP Sensing, *DAS - 2P Squared Technology*. AP Sensing, october 2024 ed., 2024. Available in the AP Sensing SharePoint archive. Accessed: 2025-04-04.

[3] AP Sensing, *DAS User's Guide (Ed. 1)*. AP Sensing, august 2021 ed., 2021. Available in the Project Olympus SharePoint archive. Accessed: 2023-10-05.

[4] AP Sensing, *DAS Waterfall*. AP Sensing, november 2024 ed., 2024. Available in the AP Sensing SharePoint archive. Accessed: 2025-04-04.

[5] AP Sensing, *DAS Configurator Client User's Guide (Ed. 7)*. AP Sensing, 7 ed., 2024. Available in the AP Sensing SharePoint archive. Accessed: 2025-04-04 [[1], [3], [7]].

[6] Pixabay, "White and grey kitten on brown and black leopard print textile," 2023. Photo retrieved from Pexels. Accessed: 2023-10-05.

[7] AP Sensing, *DAS Demo Setup at Energiezentrale Hulb*. AP Sensing, 2024. Available in the AP Sensing SharePoint archive. Accessed: 2025-04-28.

[8] G. L. Duckworth, "Distributed sensing applications of rayleigh scattering in fiber optic cables," in *Imaging and Applied Optics, OSA Technical Digest (online)*, Optica Publishing Group, 2013.

[9] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "Convnext v2: Co-designing and scaling convnets with masked autoencoders." arXiv preprint arXiv:2301.00808, 2023.

[10] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pp. 6105–6114, 2019.

[11] A. Masoudi, J. H. Snook, T. Lee, M. Beresna, and G. Brambilla, "Application of ultra low-loss enhanced backscattering fiber in high spatial resolution distributed acoustic sensors," in *27th International Conference on Optical Fiber Sensors, Technical Digest Series (Optica Publishing Group)*, 2022.

[12] K. Chandni, A. P. Singh, A. K. Rai, and A. S. Chauhan, "Image recognition using machine learning techniques," in *2023 International Conference on Sustainable Emerging Innovations in Engineering and Technology (ICSEIET)*, (Ghaziabad, India), pp. 347–350, 2023.

[13] PyTorch, *Get Started Locally*. PyTorch, 2024. Available on the PyTorch official website. Accessed: 2025-04-04.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library." arXiv preprint arXiv:1912.01703, 2019.

[15] Encord, "Classification metrics: Accuracy, precision, recall." `https://encord.com/blog/classification-metrics-accuracy-precision-recall/`, 2023. Accessed: 2023-10-05.

[16] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 60, 2019.

[17] PyTorch Vision Developers, "torchvision: Datasets, transforms, and models for computer vision." `https://pytorch.org/vision/`, 2024. Accessed: 2025-05-01.

[18] E. Riba, D. Mishkin, J. Shi, D. Ponsa, F. Moreno-Noguer, and G. Bradski, "A survey on kornia: an open source differentiable computer vision library for pytorch," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020.

[19] A. Ekimov and J. M. Sabatier, "Ultrasonic wave generation due to human footsteps on the ground," *J. Acoust. Soc. Am.*, vol. 121, no. 3, pp. EL114–EL119, 2007.

[20] K. S, D. D, S. D, I. M, C. E, and R. P, "Acoustic based footstep detection in pervasive healthcare," *Annu Int Conf IEEE Eng Med Biol Soc*, pp. 4974–4977, 2021.

[21] S. Jakkampudi, J. Shen, W. Li, A. Dev, T. Zhu, and E. Martin, "Footstep detection in urban seismic data with a convolutional neural network," *The Leading Edge*, vol. 39, pp. 654–660, 09 2020.

[22] Y. Zhou, J. K. Yeoh, Y. E. Li, and W. Solihin, "Large scale indoor occupant tracking using distributed acoustic sensing and machine learning," *Building and Environment*, vol. 247, p. 111005, 2024.

[23] M. Bublin, "Event detection for distributed acoustic sensing: Combining knowledge-based, classical machine learning, and deep learning approaches," *Sensors*, vol. 21, no. 22, p. 7527, 2021.

[24] Y. Shi and J. Zong, "Benchmarking machine learning architectures for distributed acoustic sensing footstep detection," *IEEE Sensors Journal*, vol. 25, no. 5, pp. 1234–1245, 2025.

[25] A. Yadav, S. Kumar, and T. Lee, "Contrastive multimodal embeddings for footstep recognition with das and geophones," in *Proc. ACM Int. Conf. on Embedded Sensor Systems (EmSys)*, pp. 67–78, 2021.

[26] The HDF Group, *HDF5 – Data Model, Library, and File Format*, 2024. Accessed: 2025-05-26.

[27] W. McKinney, "Data structures for statistical computing in python," *Proceedings of the 9th Python in Science Conference*, pp. 51–56, 2010.

[28] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[29] K. Wada, "Labelme starter guide." Available at `https://labelme.io/docs/install-labelme-pip`, 2024. Accessed: 2025-04-04.

[30] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Ä. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, A. Vijaykumar, A. P. Bardelli, A. Rothberg, A. Hilboll, A. Kloeckner, A. Scopatz, A. Lee, A. Rokem, A. Woods, A. R. Fulton, C. E. Masson, C. Häggström, C. Fitzgerald, C. Nicholson, C. Hagen, D. Pasechnik, E. Olivetti, E. W. Martin, E. Wieser, F. Silva, F. Lenders, F. Wilhelm, F. Young, F. Price, F. Ingold, F. Allen, G. R. Lee, G. Audren, G. Probst, H. Dietrich, H. Silterra, J. T. Webber, J. Slavic, J. Nothman, J. Buchner, J. Kulick, J. L. Schönberger, J. V. de Miranda Cardoso, J. Reimer, J. Harrington, J. H. Rodr'ıguez, J. Nunez-Iglesias, J. Kuczynski, Y. Vazquez-Baeza, and S. . Contributors, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[31] PyTorch Audio Developers, "torchaudio: Audio and signal processing in pytorch." `https://pytorch.org/audio/`, 2024. Accessed: 2025-05-01.

[32] R. Wightman, "Pytorch image models." GitHub repository, `https://github.com/huggingface/pytorch-image-models`, 2019.

[33] Lightning AI, "Torchmetrics: Modular metrics for pytorch." GitHub repository, 2021. Accessed: 2025-05-26.

[34] C. Evans, I. döt Net, and O. Ben-Kiki, "YAML Ain't Markup Language (YAML) Version 1.2 specification." `https://yaml.org/spec/1.2/spec.html`, 2009. Accessed: 2025-05-26.

Declaration

I declare that this master thesis is my original work and that only the indicated means have been used in this draft. Parts that are quotes or paraphrases of other works are identified as such by indicating the references.

This thesis has not been submitted in any form for another degree or diploma at any university or other institution and has not been published elsewhere.

The electronic version is identical to the printout.

Stuttgart, den July 19, 2025

Name: Swapnil Bhavsar

Unterschrift: _____