Spying with a microphone

CM3202 One Semester Individual Project

Final Report

Mohammed Ahmed – AhmedMA10@cardiff.ac.uk

Supervisor: Kirill Sidorov – SidorovK@cardiff.ac.uk

Moderator: Carolina Fuentes Toro – FuentesToroC@cardiff.ac.uk

Computer Science BSc

School of Computer Science and Informatics, Cardiff University

Spring 2024

Abstract

This report explores the advancement and efficacy of acoustic side channel attacks on keyboards, specifically, recognising keystrokes by leveraging modern machine learning and audio processing techniques. The system, known as KRAMS (Keystroke Recognition using Augmented Mel-Spectrograms), integrates a Convolutional and self-Attention Network, known as a CoAtNet, with SpecAugment data processing to recognise keystrokes from their acoustic features, extracted using Mel-Spectrograms. The system operates on the command line interface, allowing users to train, evaluate, and simulate attacks efficiently. We present the underlying theories, such as the identification, isolation, and extraction of keystrokes from recordings, extraction of acoustic features from those keystrokes, and the generation of Mel-spectrograms. We also examine the concept of side-channel attacks using acoustic emanations and deep learning models. The project was developed through a structured pipeline from data recording and preprocessing to model training and evaluation. Our results demonstrate that this combination of technologies can detect keystrokes with remarkable accuracy of 99% when tested on a completely unseen testing dataset. This emphasises the potential vulnerability of acoustic emanation as a vector for cyberattacks and highlights the need for awareness and improved security measures against such vulnerabilities.

Acknowledgements

I would like to thank my supervisor Dr Kirill Sidorov for offering his advice and feedback during this project.

Content

1 Introduction

Cyberattacks represent an ever-present and evolving danger that threatens every stratum of society, from individual citizens to governmental and corporate infrastructures (Kamiya et al. 2019). Whilst most adopt basic precautions to shield themselves from common forms of cybercrime, such as covering the keypad when entering their PIN at ATMs (Cardaioli et al. 2021), they may not consider other, less obvious threats they are exposing themselves to. Notably, acoustic emanations produced when typing passwords on work laptops, or during online banking sessions can potentially be exploited to intercept sensitive information. This threat is not new, with successful attacks being demonstrated as early as 2004 (Asonov and Agrawal 2004) with methods that have continually improved as time progresses (Zhuang et al. 2009; Harrison et al. 2023).

As progress is made in this field, it becomes crucial to understand and educate about these vulnerabilities, which, importantly, is one of the main methods in mitigating vulnerabilities (Allodi et al. 2018; Baek and Kim 2019). To aid these efforts, we propose KRAMS: **K**eystroke **R**ecognition using **A**ugmented **M**el-**S**pectrograms, a simple-to-use command-line interface application that allows training, evaluation, and attack all in one button press. This project aims to leverage the most recent advancements in audio processing and machine learning, implemented by Harrison et al. (2023). Specifically, we focus on creating Mel-spectrograms from audio recordings of keystrokes and training a deep learning hybrid model that incorporates traditional Convolution with self-Attention mechanisms, known as a CoAtNet (Dai et al. 2021).

2 Background

This project is made up of various concepts and inspired by various pieces of existing literature. This section explores topics such as the nature of keystrokes, visual representations of acoustic signals, side-channel attacks using acoustic emanations, modularisation, and the concept of a deep learning model. Additionally, this section reviews prior research in the field, focussing on acoustic side-channel attacks on keyboards, and outlines the relevant deep learning model and data augmentation technique used.

2.1 Theory

The following section explains the theoretical foundations, methodologies used, and the significance of each approach in understanding and analysing keystroke dynamics and side-channel attacks.

2.1.1 Extracting Features from Keystrokes

This section focuses on the process of feature extraction from keystrokes, a critical step in understanding and interpreting the unique characteristics of keystroke interactions.

Keystroke

Understanding the nature of a keystroke is fundamental to this paper. A keystroke occurs when two specific events occur. The first event is *Key Down*, which is triggered continually for the length a key is physically pressed on the keyboard. The second event is *Key Up*, which is triggered when the key currently pressed is released. Hence, a *Keystroke* is defined as a "combination of the corresponding key down and key up events." (Banerjee et al. 2015)

The two events leading to a keystroke each both produce a distinct peak when the keystroke is observed in a waveform. This can be seen highlighted in Figure 1.

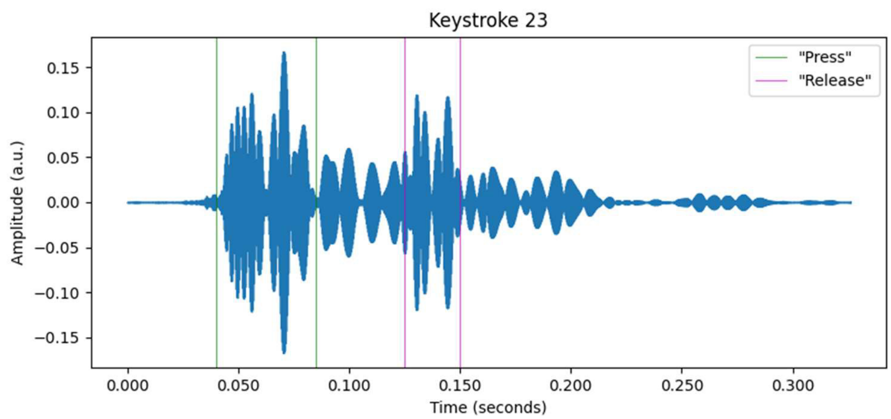

Figure 1: A waveform of a keystroke, clearly showing a "press" and a "release".

Visualising a Keystroke

Whilst keystrokes do produce distinct acoustic signatures (Asonov and Agrawal 2004; Roth et al. 2015) that can be visualised by waveforms, this representation lacks any frequency information, offers limited feature extraction abilities, and often include a lot of noise, none of which are ideal for data processing and, in turn, machine learning. Therefore, using spectrograms (time-frequency representations of sound) of these signals are advantageous, especially when training a deep learning model. Image representations provide the model more information that is easier to process, enhancing the model's ability to detect patterns and relationships within the data (Ciric et al. 2021). Additionally, models trained on image representations of acoustic signals are more robust, given that they generalise better (Perez et al. 2020), and allow for more levels of data augmentation, including flipping, rotating, scaling, cropping, translation, masking, and Gaussian noise, which can diversify the training set and support generalisation (Bahmei et al. 2022). Specifically, for spectrograms, applying time stretching and time-frequency masking has been shown to produce state-of-the-art performance during testing (see SpecAugment, Section 2.2.5).

Spectrograms and Mel Spectrograms

Spectrograms are a three-dimensional visual representation of audio signals displaying frequency along the y-axis, time along the x-axis, and spectral magnitude plotted as intensity of shading (Thomas et al. 1994), as seen in Figure 2. Spectrograms are a commonly used method of representing acoustic

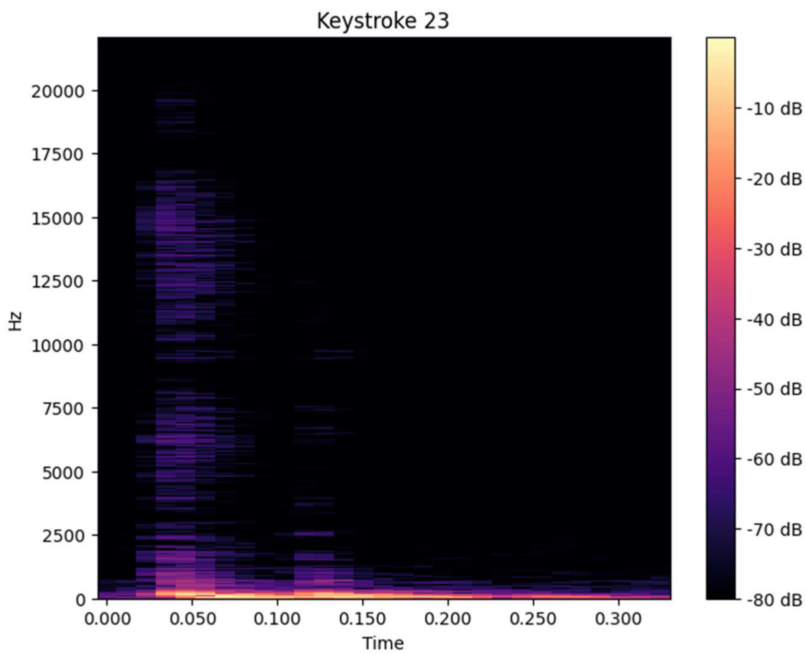

Figure 2: A spectrogram of a keystroke.

signals across various applications (Loughlin 2009).

Mel spectrograms are a specialised form of the spectrogram that converts the frequency scale along the y-axis to the Mel-scale (Kaneko et al. 2022), "a numerical scale ... which is proportional to the perceived magnitude of subjective pitch" (Stevens et al. 1937), as seen in Figure 3. This means each step on the scale is judged by listeners to be equal in distance from the next, despite the actual frequencies being logarithmically spaced. This more closely matches the human auditory system's response than regular linear frequency scales (Pedersen 1965). The result of this is a visual representation that

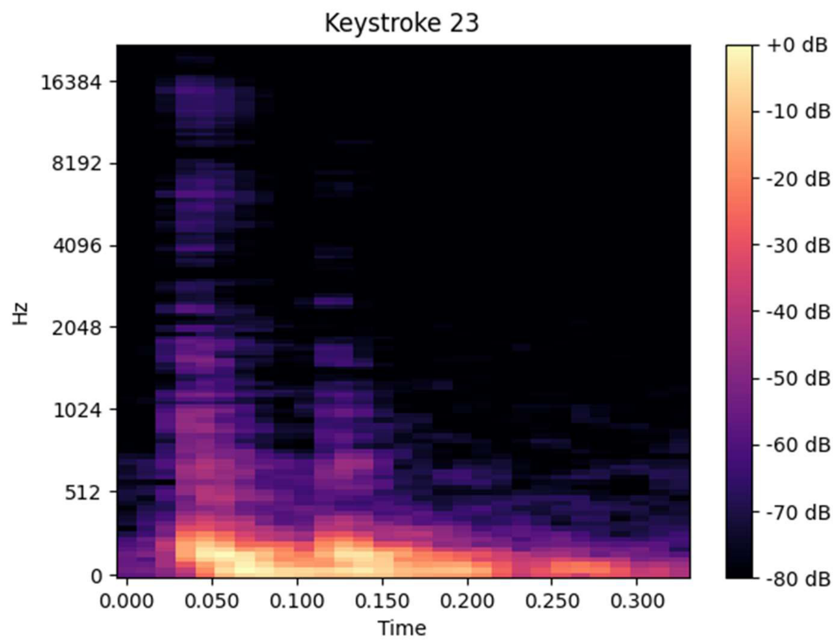

Figure 3: A Mel Spectrogram of a keystroke.

concentrates more detail into lower frequencies where human hearing is most sensitive (Sharan and Moir 2019).

Due to this, Mel-spectrograms represent sound in a visually recognisable manner (Dennis et al. 2011), which has been shown to encourage deep learning models to generalise better (Piland et al. 2023). Hence, Mel-spectrograms are to be used in this implementation.

2.1.2 Side-Channel Attacks using Acoustic Emanations

Side-Channel Attacks (SCAs) are "enabled by leakage of information from a physical cryptosystem" (Grassi et al. 2017), i.e., they are attacks that utilise physical artefacts that may be compromised unbeknownst to the user of the system.

Acoustic emanations are defined as "emanations in the form of free-space acoustical energy produced by the operation of a purely mechanical or electromechanical device equipment" (National Security Agency 1982).

Acoustic Side-Channel Attacks (ASCAs) are SCAs that make use of acoustic emanations. These have successfully been implemented in the past, for example, by Zhuang et al. (2009) (see Section 2.2.1), Cheng et al. (2020), and, more recently, Harrison et al. (2023) (see Section 2.2.3), who's research is the basis for this project.

2.1.3 Modularisation

Modularisation is a practice in which "Each task forms a separate, distinct program module" (Gouthier and Ponto 1970).

Rather than define modules from a flowchart of the stages a program may run through, it is better to define modules from a list of design decisions which are likely to change in their method of execution (Parnas 1972).

2.1.4 Deep Learning Models

Deep learning models are a subset of machine learning that use neural networks with multiple layers to model complex patterns and relationships in data. Specifically, these models can automatically learn to represent data through a hierarchy of concepts or features, making them highly effective for a range of applications from image recognition to natural language processing.

Neural Networks

Neural networks are the core of a deep learning model. These are made up of layers of interconnected nodes called neurons. Each node in a network is a small processing unit that performs simple calculations. The nodes are connected by edges that represent synapses, and each connection carries a weight that adjusts as learning progresses. (Kriegeskorte and Golan 2019)

Layers

Neural networks are characterised by their depth, which is defined by the number of layers they have. The three layers that a neural network can be constructed of are: the input layer, where the initial data enters the network, with each node in this layer representing a feature of the input data; the hidden layers, between the input and output layers where most of the processing is done, a number of these are usually used to enable modelling of complex data with high levels of abstraction; and the output layer, which produces the final results of the network such as class label in a classification task or value in regression. (Ismailov 2014)

Learning Process

Deep learning models learn by adjusting the weights of the connections in the network through a process known as backpropagation. During training, the network makes predictions, calculates the error of its predictions, known as loss, and then uses this error to update the weights to improve prediction accuracy. This process is repeated over many iterations, known as epochs. (Lillicrap et al. 2020)

Activation Functions

Activation functions are applied at each node in the neural network to help introduce non-linear properties to the network, allowing it to learn more complex patterns. Common examples include sigmoid, ReLU (rectified linear unit), and tanh. (Guo et al. 2019)

Data-driven Learning

Deep learning models require large amounts of data to learn effectively (Riazi et al. 2019), automatically extracting features from raw data during training, unlike traditional machine learning models that often require manual feature extraction (Cayir et al. 2018).

Applications

Deep learning has a wide range of applications, including image and speech recognition, natural language processing, autonomous driving, and more. Its ability to perform feature extraction makes it particularly effective for tasks involving unstructured data, such as text, images, and audio. (Deng and Yu 2014; Vieira 2017)

2.2 Prior Research

Keyboard acoustic emanations as a potential side-channel attack vector has had a substantial amount of exploration. The seminal work in this field was carried out by Asonov and Agrawal (2004), revisited and improved by Zhuang et al. (2009), and, more recently, improved by Harrison et al. (2023), who implemented SpecAugment data augmentation by Park et al. (2019) and the CoAtNet model introduced by Dai et al. (2021).

2.2.1 Keyboard acoustic emanations (Asonov and Agrawal 2004)

Asonov and Agrawal (2004) demonstrated the feasibility of exploiting the sounds generated by a keystroke to determine the characters being typed by analysing the acoustic signatures of individual keys.

Their method involved first extracting features from signals of keystrokes from audio recordings, such as the Fast Fourier Transform (FFT) components, which represent the frequency characteristics, believing each key on the keyboard produces a unique frequency spectrum due to variations in mechanical construction and placement of the keys. Next, they used a Gaussian Mixture Model (GMM) classifier to analyse the extracted features and identify the corresponding keys. The GMM classifier was trained on a labelled dataset, and during the testing phase produced results with an accuracy that was promising, especially given it was among the earliest work in this domain.

However, their method was not without limitations. The accuracy of the attack was sensitive to the position of the microphone in relation to the keyboard, as well as background noise and variations in typing style.

2.2.2 Keyboard acoustic emanations revisited (Zhuang et al. 2009)

Following on from this research, Zhuang et al. (2009) further advanced the study of acoustic emanations from keyboards. Their research revisited the issue with preprocessing steps to enhance the quality of the sound signals, including noise reduction and signal segmentation to isolate keystrokes. After this, they extracted features focusing on both temporal and spectral characteristics of the keystrokes. They made use of Mel-Frequency Cepstral Coefficients (MFCCs), which are commonly used in speech and audio recognition for their ability to represent the power spectrum in a compact form using the Mel-scale, mentioned in Section 2.1.1. After extracting these features, they use Hidden Markov Models (HMMs) to recover text, a form of unsupervised learning. Once single keystrokes have been recovered, they use the Aspell spellcheck (Atkinson 2023) to attempt to improve their results, however, find the improvements to be minimal. Instead, they develop an equation that uses patterns in plaintext to create a confusion matrix that estimates the probability of each class being the next character. On top of this, they implement an n-gram language model that "models word frequency and relationship between adjacent word probabilistically." Finally, they experiment with multiple classifiers, including a neural network using Matlab's `newpnn()` function (The Mathworks Inc 2024a), linear classification using Matlab's `classify()` function (The Mathworks Inc 2024b), and Gaussian mixtures. This results in an attack that, with a 10-minute recording of unlabelled keystrokes typing English text, can recover up to 96% of the typed characters and, additionally, 90% of 5-character passwords containing random text in fewer than 20 attempts.

2.2.3 A Practical Deep Learning-Based Acoustic Side Channel Attack on Keyboards (Harrison et al. 2023)

Building on this research, more recent research by Harrison et al. (2023) has focused on leveraging deep learning models to improve the efficacy of these attacks. They begin by recording labelled training data in the form of multiple individual audio recordings, each containing repeated keystrokes of the same key. They then isolate and extract individual keystrokes from these recordings by using a similar method as used in existing literature, described in Section 2.1.1. Features from these keystrokes are extracted in the form of Mel-spectrograms, also described in Section 2.1.1. On top of this, a data augmentation method, called SpecAugment (Park et al. 2019) (see Section

2.2.5) is applied to artificially increase the amount of data available to the deep learning model, increasing robustness by encouraging the model to generalise more. Once the dataset is created, it is split into training, validation, and test datasets and used to train a CoAtNet (Dai et al. 2021) (see Section 2.2.4), a hybrid model combining traditional Convolution with self-Attention mechanisms that has shown superior performance in image classification tasks. This method allowed Harrison et al. to achieve accuracy of 95%, the highest seen without the use of a language model. Interestingly, this model was also shown to work over the video-conferencing software Zoom, achieving an incredible accuracy of 93%. This research sets the structure of the method used in this project.

2.2.4 CoAtNet: Marrying Convolution and Attention for All Data Sizes (Dai et al. 2021)

In computer vision, convolutional neural networks (ConvNets, CNNs) have long been the model architecture of choice, with AlexNet achieving an accuracy of 84.7% in the ImageNet LSVRC-2012 challenge, 10.8% better than the runner up. Self-attention models such as transformers have shown great success in natural language processing contexts but have not shown as much success in computer vision. In 2021, Dai et al. proposed the CoAtNet family of hybrid models with an aim to combine the strengths of ConvNets and transformers. They did this using two key insights, the first being that depthwise convolution and self-attention layers can be unified via a relative attention mechanism, enabling the model to capture both local and global dependencies effectively. The second, that stacking convolutional and self-attention layers can be surprisingly effective in achieving better generalisation and capacity, if done properly. When provided with only ImageNet-1K for training, containing 1,281,167 training images, the model achieved 86.0% top-1 accuracy. Additionally, when provided with ImageNet-21K, using 10 million training images, the model achieved 88.56% top-1 accuracy. Furthermore, when provided JFT-3B for training, containing 3 billion images, the model achieved a top-1 accuracy of 90.88%, which established a new state-of-the-art result.

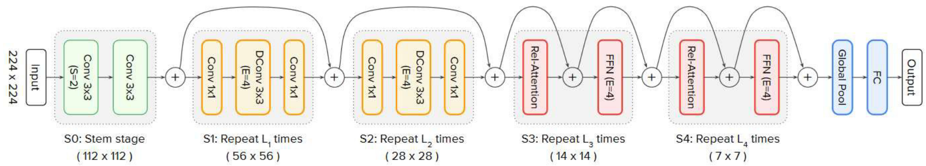

Figure 4: Overview of the CoAtNet (Dai et al. 2021)

2.2.5 SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition (Park et al. 2019)

Whilst deep learning has successfully been applied to speech recognition, models tend to overfit easily and require large amounts of training data. Data augmentation techniques are used to avoid this by increasing the variation of data in the dataset available to the model. In 2019, Park et al. proposed SpecAugment, a simple and computationally cheap-to-apply data augmentation method. There are 3 steps to applying SpecAugment. The first step is time warping which involves shifting the signal randomly in either direction up to a set distance parameter, as shown in Figure 5. Given the element of

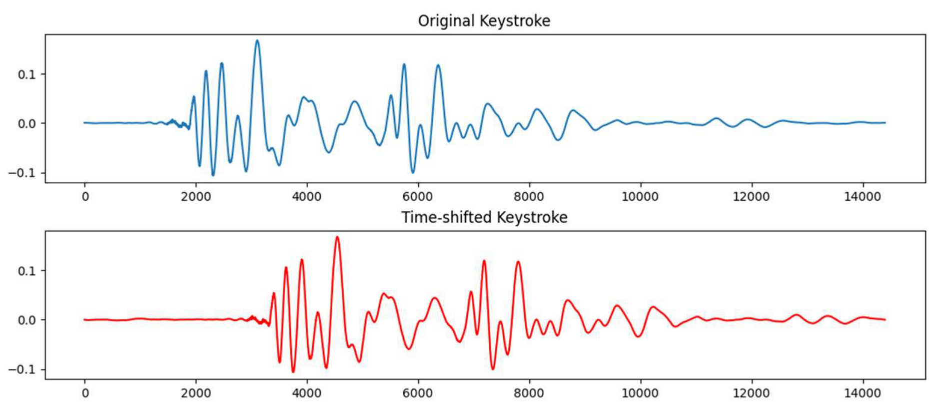

Figure 5: Visualisation of time warping.

randomness, this shift simulates slight inconsistencies in recording of data, increasing variation in input data and, therefore, increasing the robustness of the trained model by encouraging generalisation. The second step is frequency masking which involves randomly taking a fixed-sized segment of channels from the frequency axis and masking them to zero, blocking them off, as shown in Figure 6. This increases the robustness of the model against changes in spectral features by encouraging generalisation. The third step is time masking which involves randomly taking a fixed-sized segment of time steps from the time axis and masking them to zero, blocking them off, as shown in Figure 6. This

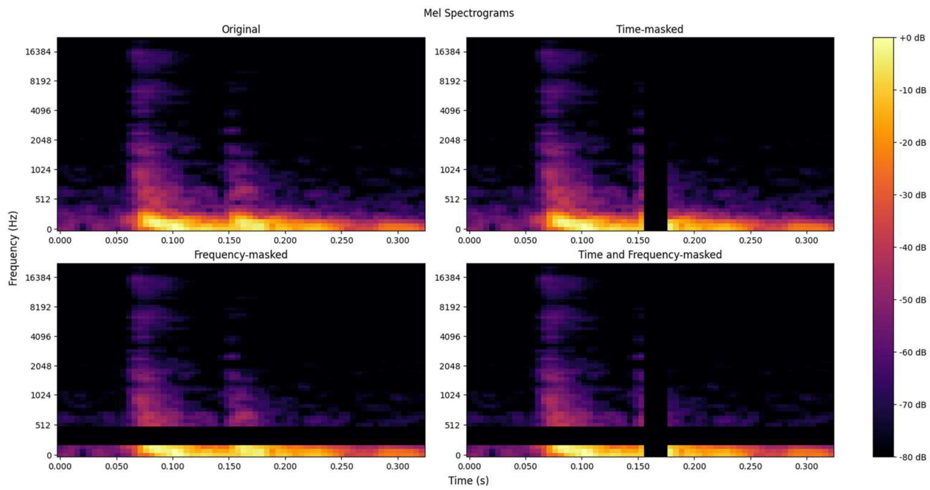

Figure 6: Visualisation of time, frequency, and time-frequency masking.

increases the robustness of the model against temporal features by encouraging generalisation. All three of these steps are also methods of artificially increasing the amount of data available to the model.

Using this data augmentation method, Park et al. achieved state-of-the-art performance on the LibriSpeech 960h and Switchboard 300h tasks. They achieved a 6.8% word error rate (WER) on LibriSpeech's test-other datatset without the use of a language model, and achieved a 7.2%/14.6% WER on the Switchboard/CallHome portion of the Hub5'00 test set without the use of a language model.

3 Specification

This section outlines the specifications for the project. It defines the objectives, scope, constraints, and requirements, both functional and non-functional, necessary to achieve the goals of the project.

3.1 Objectives and Scope

The aim of this project is to develop a keystroke recovery system for recordings of unknown keystrokes of alphabetic words or sentences based on labelled training data.

The development of this project requires recording, analysis, processing, and augmentation of audio data, and implementation, training, and evaluation of a deep learning model.

These steps can be described as individual modules of a larger working program which are designed, implemented, and refined independently of each other, whilst remaining compatible with each other. This is known as modularisation, mentioned in 2.1.1, and is to be a key concept during the development of this project.

3.2 Constraints

The project is dependent on having access to a substantial amount of labelled training data from the keyboard in question, and by limiting the character set by only allowing alphabetic characters. Training, validation, and test data should all be collected using a consistent typing force and speed, in a controlled environmental noise, and with a similar distance between the microphone and keyboard. The same microphone is to be used in the same room to collect all three datasets, typed by the same typist. The trained model is not expected to be compatible with keyboards it has not been trained on, however, it may be compatible with keyboards of the same model in the same environmental context.

3.3 Requirements

The primary requirement of the project is to develop a system for determining which keys have been pressed based on the sound they produce. This involves several layers of functionality to process, analyse, and learn from audio data.

3.3.1 Functional Requirements

Audio Data

F1. The program must be able to load audio files to be used as training and validation data.

F2. The program must be able to identify keystrokes in an audio file.

F3. The program must be able to extract identified keystrokes in an audio file.

F4. The program must be able to augment keystroke signals.

F5. The program must be able to generate Mel spectrograms from keystroke signals.

F6. The program must be able to augment Mel spectrograms.

Deep Learning Model

F7. The program must be able to train a deep learning model using Mel spectrograms.

F8. The program must be able to save and load trained deep learning models for future use.

F9. The program must be able to evaluate the deep learning model on unseen data.

3.3.2 Non-Functional Requirements

NF1. The program must be robust and resilient to handle errors or interruptions during audio file processing, model training, and evaluation.

NF2. The program must process audio files and perform identification, extraction, augmentation, and spectrogram generation operations within a reasonable time frame, even with large datasets.

NF3. The program must be intuitive to use and user-friendly, allowing users to easily provide audio files, configure parameters where necessary, and interpret evaluation results.

NF4. Documentation should be provided where necessary to guide users through the systems functionalities.

NF5. When evaluated, the program should provide at least 85% accuracy in a valid test case.

4 Design

The implemented system will be a combination of individual modules that will work together to make up the larger program. The dataset for the project will comprise of 26 audio files, one for each letter of the alphabet. Only 26 are to be used to create a minimum viable product, and once feasibility of the selected method is established, increasing the number of channels to include the spacebar, punctuation, and digits would be straightforward. Each of those 26 audio files will contain 25 keystrokes of the corresponding key on the keyboard. Whilst this may seem small, it is important to note that data augmentation is used to artificially increase the size of the dataset and, therefore, this provides a sufficient amount of initial data for a deep learning model. Figure 7

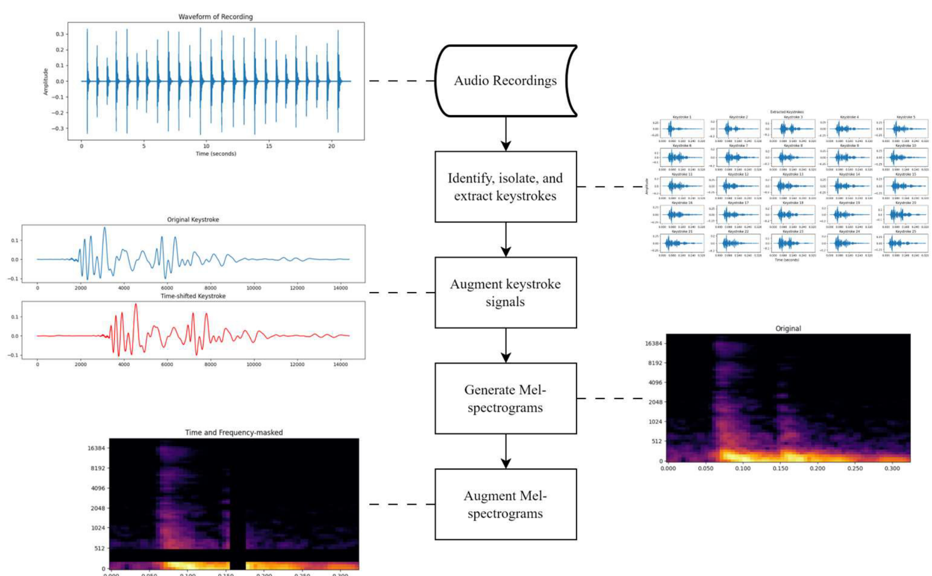

Figure 7: Data Processing Pipeline

demonstrates a simplified version of the data processing pipeline and, accordingly, the modules for the system are outlined below.

4.1 Keystroke Extraction

This section details the process by which individual keystrokes are identified and extracted from audio recordings for subsequent feature extraction.

First, a single audio file is loaded from a specified file path and the audio signal and sampling rate are returned. The energy of the audio signal is calculated in overlapping windows, following the method outlined in Section 2.1.1. This involves performing the Fast Fourier Transform (FFT) windowed segments of the audio signal to get their frequency components. The resulting coefficients are summed to determine the total energy of the segment, which is then normalised to allow consistent analysis across different recordings. Using this normalised energy, prominent peaks that that are likely to correspond to keystrokes are identified. The boundaries of each keystroke are found using fixed-size buffers either side of the identified peaks. Segments of the audio signal that are between these keystroke boundaries are extracted as the signal for each keystroke.

For each of the previous stages, the option to plot the results on a graph is given.

4.2 Feature Extraction

Keystrokes are augmented and used to generate Mel-spectrograms, which, in turn, are augmented. Data augmentation in the form of SpecAugment, mentioned in Section 2.2.5, is applied to both the keystrokes and Mel-spectrograms.

4.2.1 Keystroke Augmentation

Keystrokes are augmented by randomly rolling the signal forwards or backwards by up to 10%, simulating slight variations in keystroke recording timing, artificially increasing the size of the dataset available to the deep learning model, and increasing robustness.

4.2.2 Mel-Spectrogram Generation

Mel-spectrograms of the augmented keystrokes are generated.

4.2.3 Mel-Spectrogram Augmentation

Mel-spectrograms are augmented by applying time and frequency masks of 10% at random positions, simulating obscurities in the recordings, artificially increasing the size of the dataset available to the deep learning model, and increasing robustness.

4.3 Deep Learning Model

The design of the training and evaluation of the deep learning model is outlined below, with a detailed implementation found in Section 5.7.

4.3.1 Training

A deep learning model is trained and validated on the augmented labelled Mel-spectrograms. The model, datasets, and results are stored and saved.

Train: Performs one training epoch using the training dataset, computing and backpropagating loss for each batch.

Validation: Evaluates the model using the validation dataset to calculate the average loss and accuracy.

Run: Configures the environment, initialises the model, constant parameters, and any other required variables and objects, including the training. validation and test datasets. Runs through the specified number of epochs, performing training and periodic evaluation, saving the best models. Stores and saves the final model, datasets, and results.

4.3.2 Evaluation

The deep learning model and test dataset are loaded, and the model is evaluated.

Load Model and Test Dataset: The trained deep learning model is loaded along with the test dataset generated during initial training. This allows for consistent and reproducible splits for training and evaluation.

Evaluate Model: The deep learning model is evaluated using the test dataset, returning the predicted labels.

Calculate Accuracy: The returned predicted labels are compared to the targets from the dataset and accuracy is calculated. A confusion matrix is created, and precision and recall are calculated.

5 Implementation

As described in the section above, there are multiple stages to completing this project. This section outlines the detailed process of implementing the system, covering audio data recording, keystroke extraction, feature extraction, training data processing, and model evaluation.

5.1 Audio Data Recording

The first step of the implementation was creating a dataset to be used for training, validation, and testing. Audio was recorded using a Samsung Galaxy S10+ (Samsung UK 2024) with the Samsung Voice Recorder (Samsung 2024). Recording quality was set to Medium (128kbps, 44100Hz) and recordings were monaural. The keyboard recorded was a havit Wired Mechanical Keyboard (Amazon 2024), with blue switches, which are typically described as tactile and producing audible feedback. The recording device was set 10cm away from the keyboard. This resulted in `26` audio-only MPEG-4 files (`.m4a`) recordings named `A.m4a`, `B.m4a`, `C.m4a`, ..., `Z.m4a`, each containing `25` keystrokes of the corresponding key. The files were then converted to wave (`.wav`) files using FFmpeg (FFmpeg developers 2024), an open-source suite of libraries for handling multimedia. This was done to allow simpler compatibility with the librosa Python library, however, is not necessary if a valid install of FFmpeg is available for librosa to use.

5.2 Keystroke Extraction – `keystroke_extractor.py`

To extract the individual keystrokes from each recording, a function was defined, called `load_recording()`, to first load a single recording. This function simply calls the `librosa.load()` function from the librosa library (McFee et al. 2023) with the file path of the recording, returning the signal and its native sampling rate. A function called `plot_waveform()` was defined to display this signal over time as a waveform, as shown in Figure 8.

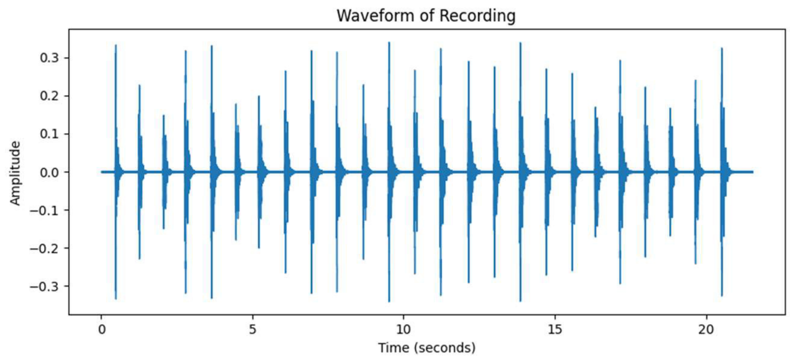

Figure 8: A waveform of a recording plotted using `plot_waveform()`.

As mentioned by Harrison et al. (2023), the majority of recent literature extract keystrokes using a similar method. This involves performing the Fast Fourier Transform (FFT) on the audio signal in overlapping windowed segments and summing the resulting coefficients to compute energy. This is then normalised and can be plotted on a graph to show the energy of the signal over time. A threshold is defined, and any peaks exceeding the threshold indicate the presence of a keystroke in the audio recording.

This method was initially the one chosen for this project, however, others were experimented with, and a better solution was found. The final method used was identical to the method mentioned, up until the definition of a threshold, and thus, a function called `process_keystrokes()` was defined that computes the energy of the signal using the `scipy.fft()` and `numpy.sum()` functions and returns it. A function called `plot_energy()` was defined to display the energy signal over time onto a graph, as shown in Figure 9.

Next, a function called `isolate_keystroke_peaks()` was defined that enters a loop ranging values between 0.01 and 1, incrementing in 0.01 steps. This value is used as the `prominence` parameter passed to the `scipy.signal.find_peaks()` function, along with the energy signal and a fixed `distance` parameter of `100` samples. This causes the function to only identify peaks with a minimum distance of at least `100` samples from each

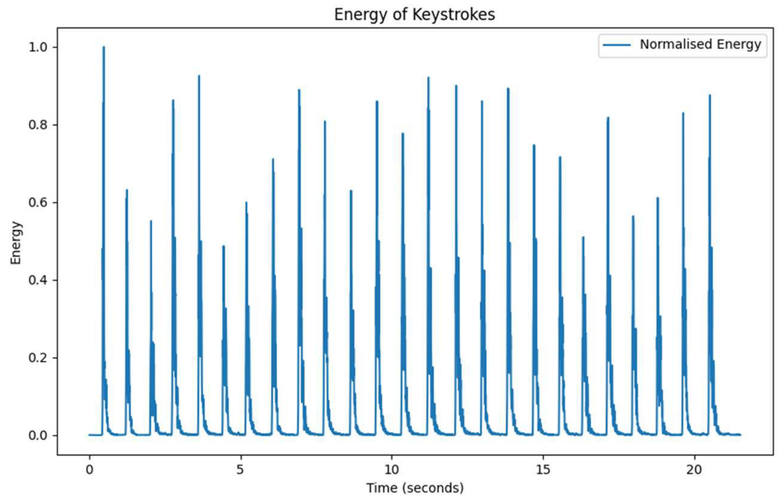

Figure 9: Energy of a recording displayed using `plot_energy()`.

other. The value for this was chosen after running a script called `peak_finder.py` which uses `np.diff()` on an array of peak values that the user identifies as being correctly annotated, the console log for which can be found in Appendix 1. The script found the minimum distance between peaks to be 125 samples, and therefore, an arbitrary 80% of that value, 100 samples, was chosen as the required minimum distance between peaks for `scipy.signal.find_peaks()`. This was found to be extremely effective, regardless of whether the number of peaks to find is known or not. In this specific case, the loop breaks once the specified 25 peaks are found. The function then returns the location of each peak in the signal in samples. A function called `plot_peaks()` was defined to plot the identified peaks above the energy signal over time on a graph, as shown in Figure 10.

The next step was to find the beginning and end of each keystroke in the audio signal. A function called `find_keystroke_boundaries()` was defined that takes the position of the peaks, the signal, and fixed `before` and `after` buffer values that denote the number of samples to include in the keystroke before and after the peak. The function calculates the boundaries which are stored as a 2-dimensional array and returned. A function called

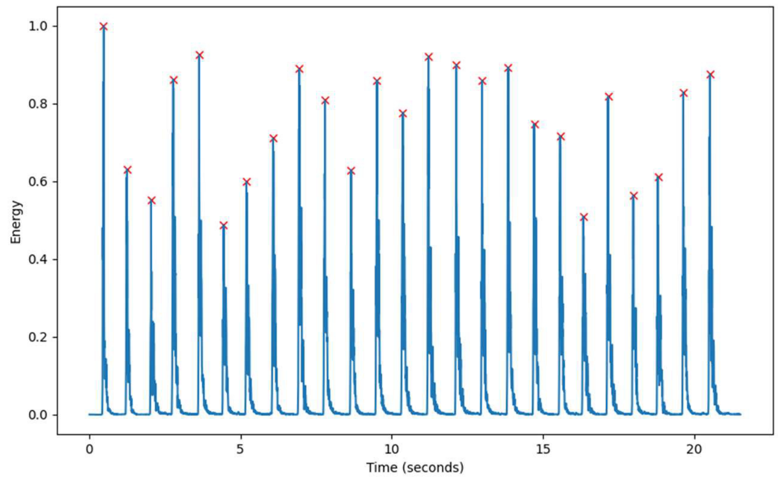

Figure 10: Peaks in energy over time plotted using `plot_peaks()`.

`plot_keystroke_boundaries()` was defined to display the “Start” and “End” of each keystroke, as shown in Figure 11.

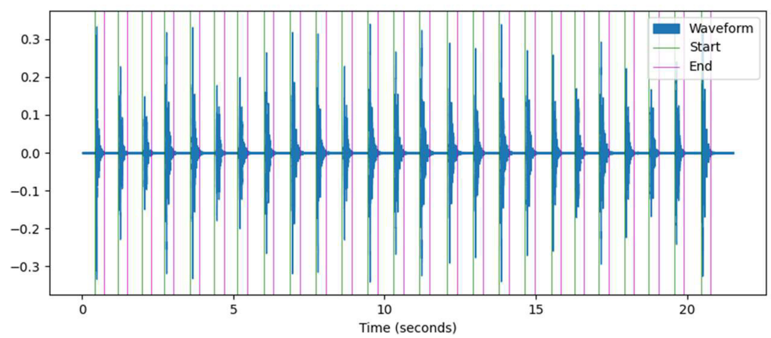

Figure 11: “Start” and “End” of each keystroke shown over the waveform of the recording using `plot_keystroke_boundaries()`.

The last step is to extract the keystrokes from the audio recording signal using these keystroke boundaries. A function called `isolate_keystrokes()` was defined that takes the keystroke boundaries and the audio recording signal. The function iterates through the list of start and end boundaries and slices the audio signal for each pair. The function pads the slice with `0`s if the start has been calculated before the beginning of the recording, or if the end is after the end of the recording. These keystroke slices are stored in an array and returned by the function. A function called `plot_extracted_keystrokes()` was defined to plot each of the 25 keystrokes as waveforms over time in a $5x5$ grid, as shown in Figure 12.

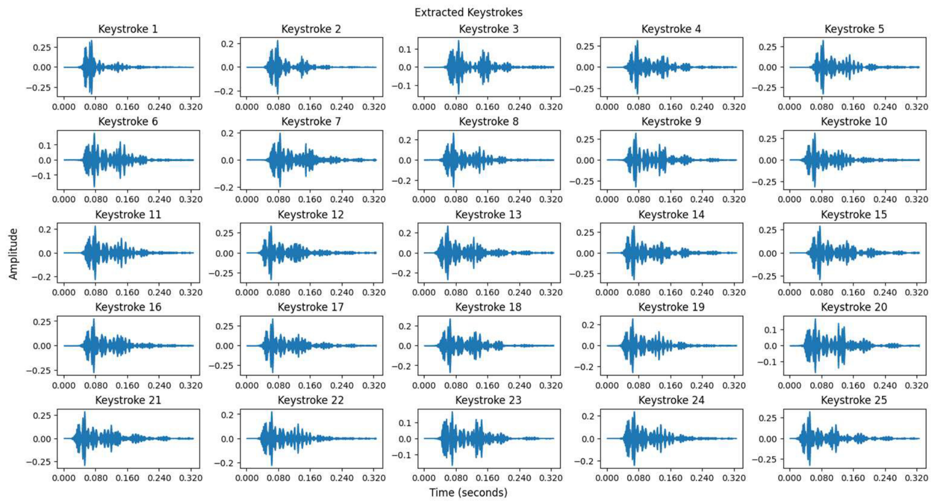

Figure 12: Extracted keystrokes over time as waveforms displayed using `plot_extracted_keystrokes()`

5.3 Feature Extraction – `feature_extractor.py`

Before extracting features from keystrokes, data augmentation in the form of SpecAugment, mentioned in Section 2.2.5, was applied.

The first step of this involved defining a function, called `signal_data_augmentation()`, that takes the extracted keystrokes and randomly time-shifts them by up to 10% either forwards or backwards. This simulates slight variations in keystroke recording and isolation timing, artificially increasing the size of the dataset available to the deep learning model and increasing robustness by causing the model to generalise more. These augmented keystrokes, along with the original ones, are stored in an array and returned by the function. A function called `plot_augmented_keystrokes()` was defined that displays the enhanced keystrokes, as shown in Figure 13.

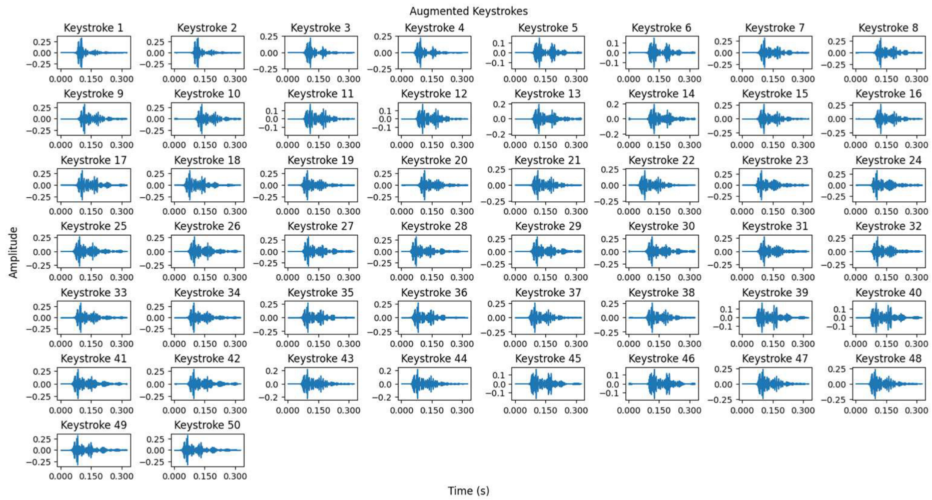

Figure 13: Augmented keystrokes displayed by `plot_augmented_keystrokes()`

Next, Mel-spectrograms of these augmented keystrokes are generated. To do this, a function called `generate_mel_spectrogram()` was defined which takes the keystroke signal, sampling rate, window size, and hop size and calls the `librosa.feature.melspectrogram()` function with the `n_mels` parameter set to `64`, returning a Mel-spectrogram of the keystroke. A function called `display_mel_spectrograms()` was defined to visualise these Mel-spectrograms, as shown in Figure 14.

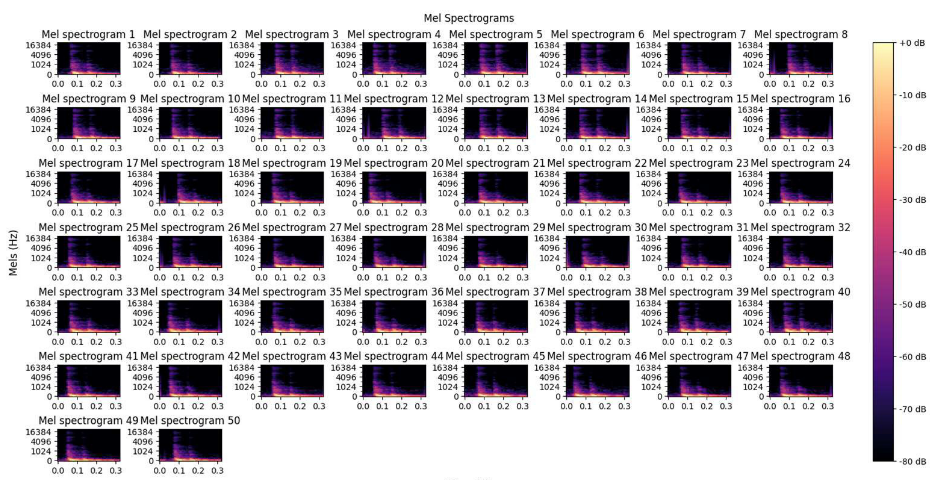

Figure 14: Mel-spectrograms displayed by `display_mel_spectrograms()`.

Once these Mel-spectrograms have been generated, SpecAugment data augmentation can be applied to them. To do this, a function called `mel_spectrogram_data_augmentation()` was created that instantiates two masks: a time mask using `torch.transforms.TimeMasking` and a frequency mask using `torch.transforms.FrequencyMasking`, which both mask up to 10% of their respective axes in random positions. From this, three new Mel-spectrograms are created: time-masked, frequency-masked, and time-frequency-masked. This simulates obscurities in recording such as background noise, therefore increasing the robustness of the model by causing it to generalise more, as well as artificially increasing the size of the dataset available to the model.

5.4 Training Data Processor – `training_data_processor.py`

To facilitate loading and handling of training data, `training_data_processor.py` was created. Its functions are outlined below.

`get_file_paths()` takes a directory and returns an array of file paths of audio files to be loaded, i.e., `directory\A.wav`, `directory\B.wav`, `directory\C.wav`, etc.

`data_processing_pipeline()` takes one audio recording file path, the window size and hop size, and before and after buffer values. Using these, it calls the functions from `keystroke_extractor.py` and `feature_extractor.py` to create fully augmented Mel-spectrograms of the keystrokes in the specified recording.

`unaugmented_data_processing_pipeline()` takes one audio recording file path, the window size and hop size, and before and after buffer values. Using these, it calls the functions from `keystroke_extractor.py` and `feature_extractor.py` to create Mel-spectrograms of the keystrokes in the specified recording, without calling the augmentation functions.

`process_recordings()` takes a list of file paths, the window and hop size, the before and after buffer values, and a augment flag. Using these, it iterates through the file paths and, for each one, calls `data_processing_pipeline()` if the augment flag is `True` or `unaugmented_data_processing_pipeline()` if the flag is `False`, and appends the resulting Mel-spectrograms to an array, as well as its file path, a label, and a target integer all to their own arrays.

`to_dataframe()` takes the resulting arrays from `process_recordings()` and converts them to a Pandas (The pandas development team 2024) dataframe.

`to_csv()` takes a dataframe and file path and saves the dataframe to the specified file path.

`from_csv()` takes the file path of a dataframe, loads it, and returns it.

5.5 Custom PyTorch Dataset – `RecordingDataset.py`

To aid training the model, a custom PyTorch Dataset was created. The class is outlined below.

`__init__()` takes a dataframe created by `training_data_processor.to_dataframe()` and the file path the data originates from. These are stored as properties.

`__len__()` returns the size of the dataframe provided, i.e., the size of the dataset.

`__getitem__()` takes an index and returns the Mel-spectrogram and target of the entry at that position in the dataframe as tensors.

`get_label()` takes an index and returns the label of the entry at that position in the dataframe.

`get_label_from_target()` takes a target and returns the label of an entry in the dataframe with a matching target.

5.6 CoAtNet Implementation – `coatnet.py`

This file contains an implementation of the CoAtNet family of hybrid models by Wu (2021), as mentioned in Section 2.2.4.

5.7 Train CoAtNet model – `train_model.py`

This stage of the development process involves defining the hyperparameters to be used, initialising the model and other required objects and variables, defining the training and validation loops, training the model, and storing the results.

5.7.1 Defining Hyperparameters

The hyperparameters used to train the model and process the dataset were based on those from literature by Harrison et al. (2023), with a few exceptions, namely, the batch size, the time-shift percentage, and the window size. These are presented in Table 1.

Table 1: Hyperparameters used to train the final model and process the dataset. Adapted from Harrison et al. (2023).

Parameter	Value
Epochs	1100
Epochs per Checkpoint	10
Batch Size	130
Loss Type	Cross Entropy
Optimiser	Adam
Maximum Learning Rate	$5e-10$
Maximum Time-shift Percentage	10%
Mask Percentage	10%
Number of Masks per Axis	2
Mel Bands	64
Window Size	1023
Hop Length	225
Before Buffer	$0.3 \times 14400 = 4320$
After Buffer	$0.7 \times 14400 = 10080$
Dataset Size	5200
Dataset Split Method	Random

Dataset Split Ratio	8:1:1
Normalised Data	Yes

5.7.2 Defining Constants

Once the above hyperparameters have been established, the first step in developing the training module is to define a subset of the hyperparameters as constants. This is done as seen below.

```
WINDOW_SIZE = 1023
HOP_SIZE = 225
BEFORE = int(0.3 * 14400)
AFTER = int(0.7 * 14400)
NUM_EPOCHS = 1100
EPOCHS_PER_CHECKPOINT = 10
BATCH_SIZE = 130
LEARNING_RATE = 0.0005
```

In addition to this, some file directories are defined at this point.

```
RECORDINGS_DIR = os.path.join("Recordings")
BASE_DIR = os.path.join("Results",
datetime.now().strftime("%Y%m%d%H%M%S"))
CHECKPOINT_DIR = os.path.join(BASE_DIR, "Checkpoints")
MODEL_DIR = os.path.join(BASE_DIR, "Model")
FIGURE_DIR = os.path.join(BASE_DIR, "Figures")
DATA_DIR = os.path.join(BASE_DIR, "Data")
```

5.7.3 Running Loop – `run()`

Inside a function called `run()`, first, `tdp.get_file_paths()` is called passing in `RECORDING_DIR` as a parameter. The returned file paths are passed into `tdp.process_recordings()` along with `WINDOW_SIZE`, `HOP_SIZE`, `BEFORE`, and `AFTER`. The resulting arrays are passed into `tdp.to_dataframe()` which is returned and stored in the variable `df`. This is passed into `RecordingDataset` to instantiate the dataset, stored in the variable `dataset`. The dataset is now split into three distinct sets called `train_dataset`, `validation_dataset`, and `testing_dataset` using `torch.utils.data.random_split()` with a ratio of $8:1:1$, i.e., 80% of the dataset for training, 10% of the dataset for validation, and 10% of the dataset for testing. Given the overall dataset size of 5200, this results in a training dataset that contains 4160 samples, a validation dataset that contains 520 samples, and a testing dataset that also contains 520 samples. Stratified sampling, involving randomly selecting a fixed number of keystrokes from each recording for each dataset, was attempted but found to be suboptimal, agreeing with the results found by Harrison et al. (2023). The training and validation datasets are passed into separate `torch.utils.data.DataLoader()` objects along with the `BATCH_SIZE`, `shuffle=True`, and `drop_last=True`. These are named `train_loader` and `validation_loader` and are used to allow batching to occur, i.e., allow `BATCH_SIZE` number of samples to be passed to the model at a time. The shuffle parameter is used to randomise the dataset before samples are selected, and the `drop_last` parameter avoids a case in which the last batch is smaller than the previous group of batches due to the dataset containing a number of samples that is not evenly divisible by the `BATCH_SIZE;` i.e., the data loader drops the last non-full batch on each iteration of the entire batched dataset. This is to be avoided because smaller batches inherently have a higher level of variance and can thus cause inconsistencies in the model.

Next, the CoAtNet model is defined. The parameters used to define the model are based by the CoAtNet variant CoAtNet-4 presented in the paper by Dai et al. (2021), with `[2, 2, 12, 28, 2]` blocks and `[192, 192, 384, 768, 1536]` hidden dimensions (i.e., channels) in the respective stages. However, CoAtNet-4 could not be used due to its default image resolution of $224x224$, whilst the Mel-spectrograms generated in this project are $64x64$. As well as this, the default number of channels for CoAtNet-4 is 1000, whilst in this case, we only require 26, one for each letter of the alphabet. Lastly, the default number of input channels for CoAtNet-4 is 3, whilst we only require 1, given the monaural audio recordings. Therefore, the model is instantiated directly using the following parameters and function call:

```
num_blocks = [2, 2, 12, 28, 2]
channels = [192, 192, 384, 768, 1536]
```

```
model = CoAtNet(image_size=(64, 64), in_channels=1,
num_blocks=num_blocks, channels=channels, num_classes=26)
```

Next, the Adam optimiser and Cross Entropy Loss criterion are instantiated using their default values. These were chosen based on their use in the paper by Harrison et al. (2023) as well as their use in the original CoAtNet paper (Dai et al. 2021). A variable called `device` is created which, if `torch.cuda.is_available()` equals `True`, stores the value `"cuda"`, otherwise, stores the value `"cpu"`. This is used in multiple instances when moving tensors to the GPU, if one is available, for hardware accelerated performance. The model is now moved to `device` using `model.to(device)`. Empty arrays are initialised to store training and validation loss values over time, so that they can later be reviewed and plotted. A best validation loss variable is also initialised to the value `float('inf')` which is to be used as a comparison variable.

At this point, all the initialisation is complete. The function now enters the training loop, which calls the `train()` and `validation()` functions. The training loop iterates through the values beginning at the number `1` to the value of the `NUM_EPOCHS` constant. The loop's iterable is wrapped with `tqdm()` from the tqdm library (da Costa-Luis et al. 2024) to enable progress metrics, including a progress meter and estimated remaining time. It begins with by calling the `train()` function (explained in 5.7.4), passing in the model, device, train dataset loader, optimiser, criterion, and current epoch number. Returned, is the loss value of that training epoch, which is appended to the training loss array created before the loop began. The loop now determines whether the current epoch is a checkpoint by using the modulo operator to calculate the remainder when the current epoch number is divided by `EPOCHS_PER_CHECKPOINT`. If this equals `0`, the current epoch is a checkpoint and the `validation()` function (explained in 5.7.5) is called, passing in the model, device, validation dataset loader, and criterion. Returned, is the loss value of that validation evaluation, which is appended to the validation loss array created before the loop began. This loss value is now compared to the best validation loss. If the new loss improves on the best validation loss, the best value is updated to the current value, and a checkpoint is created. This checkpoint stores the epoch number, the model's current state, the optimiser's current state, the last training loss, and the validation loss. This is saved to the `CHECKPOINTS_DIR` with a filename that includes the epoch number to make it unique when compared to other checkpoints in that run. The loop has now completed a single epoch iteration, and repeats `NUM_EPOCH` times.

When complete, the loop stores the final model state in the `MODEL_DIR` and plots the training and validation losses to a graph using Matplotlib, storing that in the `FIGURES_DIR`. Finally, the entire dataset is stored in the `DATA_DIR`, along with the training, validation, and testing indices, which can be used to retrieve the distinct datasets used in the current run.

5.7.4 Training Loop – `train()`

The `train()` function takes the model, device, training dataset loader, optimiser, criterion, and current epoch number in as parameters. First, the model is set to train mode using `model.train()`, as this can affect parameters such as batch normalisation and dropout. A float variable to be used to track the running loss is initialised with a value of `0.0`. The batch loop is now entered, which iterates over the batches provided by the training dataset loader. Each batch contains a `BATCH_SIZE` sized array of data and corresponding target values. These are first moved to the device, and then the data has an extra dimension added to represent the number of channels, in this case, one for the monaural audio. Before the model is given data and backpropagation is started, the gradients are explicitly set to `0` using `optimiser.zero_grad()` (PyTorch [no date][a]). This is due to PyTorch's default behaviour of accumulating gradients on backward passes, which is useful when using minibatches, as in this case (PyTorch [no date][b]). Hence, at the beginning of each training loop, the gradients are set to `0` to ensure correct tracking. The batch of data is then passed through the model for prediction, the output of which is returned. This is passed into the criterion along with the batch of targets which computes the loss between them. The gradient of that loss is then computed using `loss.backward()` with respect to the model parameters, and then the model's parameters are updated based on those gradients. Finally, the running loss is updated by adding the current loss with an addition assignment.

At this stage, the training loop is complete. The function calculates the average loss over all the batches provided by the training dataset loader, and returns it, completing the function.

5.7.5 Validation Loop – `validation()`

The `validation()` function takes the model, device, validation dataset loader, and criterion in as parameters. First, the model is set to evaluation mode using `model.eval()`. A float variable to be used to track the running loss is initialised with a value of `0.0`. In addition to this, an integer to count the number of correctly predicted samples is initialised with a value of `0`, as well as an array used to store any misclassified examples. The function then enters the following with statement: `with torch.no_grad()`. This disables gradient computation which, for model evaluation, is not necessary and therefore improves performance and memory usage. Inside this statement, the batch loop is entered, which iterates over the batches provided by the validation dataset loader. As in the training loop, each batch contains a `BATCH_SIZE` sized array of data and corresponding target values. These are again moved to the device and data has an extra dimension added. The batch of data is passed through the model for prediction and the output is returned. This is passed into the criterion along with the batch of targets, and the loss between them is computed. The running loss is now updated by adding the current loss with an addition assignment.

Misclassified samples are now logged. The misclassified samples are first identified by extracting the predicted class labels from the output logits of the model, which are then compared to the target labels, returning Boolean values where each value is `True` if the prediction matches the target, and `False` otherwise. The number of `True` values is summed and added to the variable counting correctly predicted samples using an addition assignment.

At this point, the validation loop is complete. The function calculates the average loss over all the batches provided by the validation dataset loader, as well as the accuracy percentage of samples correctly identified, which is printed out. The misclassified samples array is then formatted and printed to the console. Finally, the average loss is returned, completing the function.

5.8 Model Evaluator – `model_evaluator.py`

To aid in evaluating the model against a recording of unknown keystrokes, `model_evaluator.py` was created. Its functions are outlined below.

`load_and_prepare_model()` takes the path in which the model to be evaluated is stored, as well as the device to load the model onto in as parameters. If the model ends with `"model.pth"`, it is assumed to be the final model and is loaded directly. Otherwise, it is assumed that the path that has been passed in is the path to a checkpoint, and the model is read from the checkpoint dictionary stored in that file location. The model is then moved to the device, set to evaluation mode, and returned.

`predict_keystrokes()` takes the model, Mel-spectrograms of the unknown keystrokes from the recording, and the device in as parameters. An empty array is initialised to store the predicted output labels from the model. The function enters a with `torch.no_grad()` statement, disabling gradient computation which is not necessary for evaluation, improving performance and memory usage. The function iterates over each Mel-spectrogram that has been passed in, first moving it to the specified device. The Mel-spectrogram is then "unsqueezed" twice in order to add two dimensions to it, which represent the batch size and number of channels respectively, in this case, both `1` given the single Mel-spectrogram and monaural audio. The Mel-spectrogram is then passed to the model to predict its label, which returns output logits. The index of the maximum value in the output along dimension `1`, representing the labels predicted by the model, is extracted. This corresponds to the most likely class label predicted by the model. The extracted index is converted to the character it represents and is appended to the output labels array. The character is also printed to the console, allowing real-time observation of predictions as soon as they are available. Finally, the output labels array is returned by the function.

5.9 Main File – `KRAMS.py`

To bring the project together, `KRAMS.py` was created. If the script is run as the main function, i.e., `if __name__ == "__main__"`, then the `argparse` library is imported to handle user arguments. An `ArgumentParser` object is created with the name of the project and a description. Arguments are added using the library's `add_argument()` function. Specifically, an argument for the path to the directory containing the training recordings, an argument for the path to the recording to attack, and an argument for the number of keystrokes present in the attack recording. The arguments inputted by the user are then parsed using the library's `parse_args()` function. Assuming the user has entered a valid set of arguments, training parameters are now defined and passed into the `krams()` function. This function starts by importing the necessary libraries. It then calls `train_model.run()` with the required training parameters which, once the model has been trained, returns the `base_dir` storing the results. The attack recording is then processed using `tdp.unaugmented_data_processing_pipeline()` which is called with all the necessary arguments, returning the extracted Mel-spectrograms. The trained model is then loaded using `me.load_and_prepare_model()` and passed into `me.predict_keystrokes()` along with the Mel-spectrograms, which predicts and prints the keystrokes from the attack recording.

6 Results and Evaluation

This section presents the results of the implementation and the evaluation of the trained models. It describes the evaluation files and their functions, discusses the performance of the models, and examines the classification reports, confusion matrices, and potential improvements to the implementation process.

6.1 Evaluation Files

To aid with evaluation, two module files and a client file were created.

6.1.1 Data Loader – `trained_model_data_loader.py`

To facilitate with loading the data saved by the run of the implementation, `trained_model_data_loader.py` was created. Its functions are outlined below.

`get_dataset()` takes the `DATA_DIR` path of the run and loads the file containing the entire dataset, returning it.

`get_train_dataset()` takes the `DATA_DIR` path of the run and loads the file containing the entire dataset. It then loads the file containing the indices of the training dataset. Once loaded, it creates a `torch.utils.data.Subset` instance of the dataset, containing only the samples present in the training dataset, which is returned.

`get_validation_dataset()` takes the `DATA_DIR` path of the run and loads the file containing the entire dataset. It then loads the file containing the indices of the validation dataset. Once loaded, it creates a `torch.utils.data.Subset` instance of the dataset, containing only the samples present in the validation dataset, which is returned.

`get_test_dataset()` takes the `DATA_DIR` path of the run and loads the file containing the entire dataset. It then loads the file containing the indices of the test dataset. Once loaded, it creates a `torch.utils.data.Subset` instance of the dataset, containing only the samples present in the test dataset, which is returned.

6.1.2 Evaluation Metrics – `trained_model_evaluation_metrics.py`

`check_accuracy()` takes the target labels and the predicted output labels in as parameters, both as arrays. These are named `target_labels` and `output_labels` respectively. The function starts by ensuring the length of `target_labels` and the length of `output_labels` match. If so, the function continues by initialising a counter to keep track of the number of correctly identified characters. The function then iterates through the `target_labels` and compares them to the `output_labels`. Each time there is a match, the correctly identified characters counter is incremented. Once the loop is complete, the function calculates accuracy as a percentage by dividing the value in the correctly identified characters counter by the length of `target_labels` and multiplying the result by `100`. Finally, the function prints the accuracy and returns it.

`classification_report()` takes the target labels and the predicted output labels in as parameters, both as arrays. These are used to create a Scikit-learn classification report (Pedregosa et al. 2011) using `sklearn.metrics.classification_report()` which displays precision, recall, F_1-score, and support across each class, and as an average over all the classes.

`confusion_matrix()` takes the target labels and the predicted output labels in as parameters, both as arrays, and a title for the confusion matrix. These are used to create a Scikit-learn confusion matrix using `sklearn.metrics.ConfusionMatrixDisplay.from_predictions()`.

6.1.3 Model Evaluation Client – `trained_model_evaluation.py`

To evaluate the best version of the trained model, `trained_model_evaluation.py` was created. First, `BASE_DIR`, `DATA_DIR`, `MODEL_DIR`, and `CHECKPOINT_DIR` are all defined, leading to the final model's base directory and subdirectories. Next, the entire dataset and the test dataset are loaded using `tmdl.get_dataset()` and `tmdl.get_test_dataset()` respectively. A variable is created which, if `torch.cuda.is_available()`

equals `True`, stores the value `"cuda"`, otherwise, stores the value `"cpu"`. This, along with the path to the model or checkpoint to evaluate, is passed into `me.load_and_prepare_model()`. The resulting model object is stored. The Mel-spectrograms are separated into their own array from the test dataset and passed into `me.predict_keystrokes()`, along with the model and device. The output labels are returned and printed. The targets from the test dataset are converted into labels and separated into their own array, which is printed. The target labels and output labels arrays are passed into `tmem.check_accuracy()` that returns the final accuracy, which is printed. The target labels and output labels arrays are then passed into `tmem.classification_report()` that returns a Scikit-learn classification report, which is printed. Finally, the target labels and output labels arrays are passed into `tmem.confusion_matrix()` that creates and returns a Matplotlib plot containing a Scikit-learn confusion matrix, which is then displayed.

6.2 Model Evaluation Metrics

The following section focuses on explaining the different metrics used to evaluate the model, the metrics the model achieved, and what they represent.

6.2.1 Performance

The implementation in Section 5 was run using Google Colab (or simply, Colab), a hosted Jupyter Notebook service that provides access to computing resources. Notably, Colab provides access to GPUs, such as Nvidia A100s, V100s, and T4s. The code was copied into a Colab instance and run on an Nvidia V100 GPU to strike a balance between performance and cost of compute units. The computation was carried out successfully, taking 3 hours, 42 minutes, and 30 seconds (3:42:30) to run, according to tqdm, with an average iteration time of 12.14 seconds. This resulted in a subdirectory in the `"Results/"` directory named `"20240427130931/"`. As expected, this contains the relevant `Checkpoints`, `Data`, `Figures`, and `Model` subdirectories. The resulting directory was plugged into the evaluation client. Two models were evaluated: the most recent, and therefore, best validation checkpoint, and the final model. The best checkpoint was recorded at epoch 130 and, therefore, is referred to as Checkpoint 130. The dataset used for testing was produced during training, as detailed in Section 5.7.3. As mentioned, the overall dataset contained 5200 samples and was split $8:1:1$ to create the training, validation, and testing datasets. This resulted in a testing dataset containing 520 labelled Mel-spectrograms split from the original overall dataset, and has not been seen by the model during training.

6.2.2 Classification Reports

Table 2 and Table 3 display summaries of the classification reports for Checkpoint 130 and the final model, respectively. These summaries include the averaged precision, recall, F_1-score, and support metrics. The full classification reports detailing the metrics for each class can be found in Appendix 2 and Appendix 3 for Checkpoint 130 and the final model, respectively.

Table 2: Classification Report for Checkpoint 130

	Precision	Recall	F_1-Score	Support
Accuracy	–	–	0.99	520
Macro Average	0.99	0.99	0.99	520
Weighted Average	0.99	0.99	0.99	520

Table 3: Classification Report for the final model

	Precision	Recall	F_1-Score	Support
Accuracy	–	–	0.95	520
Macro Average	0.96	0.95	0.95	520
Weighted Average	0.95	0.95	0.95	520

Precision measures the subset of *true* positive label predictions among *all* positive label predictions made by the model, i.e.,

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall measures the subset of *true* positive label predictions among *all actual* positive instances in the dataset, i.e.,

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F_1-score is the harmonic mean of precision and recall and considers both false positive and false negatives, making it useful for imbalanced datasets that may skew precision and recall. F_1-score is therefore defined as:

$$F_1\text{-score} = \frac{\text{True Positives}}{\text{True Positives} + \frac{1}{2}(\text{False Positives} + \text{False Negatives})}$$

Support is simply the number of *actual* occurrences of each class in the dataset, i.e., the number of instances in each class. (Acharya 2024).

As shown in Table 2, Checkpoint 130 performed exceptionally well when evaluating the test dataset, with an average weighted precision, recall, and F_1-score of 0.99. Table 3 shows that the final model also performed exceptionally well, whilst slightly underperforming when compared to Checkpoint 130, with an average weighted precision, recall, and F_1-score of 0.95. The final model likely underperforms compared to Checkpoint 130 as it likely started overfitting at around 800 epochs, when analysing the loss graph (see Figure 15, Section 6.2.3).

6.2.3 Loss

The loss graph, displaying the training and validation loss over time, can be seen in Figure 15, with Figure 16 scaled to ignore a significant spike in training loss at around 150 epochs. During this spike, the training loss jumps to around 340, before quickly settling down. This could indicate an inconsistency in the training dataset, or potentially a misconfiguration of hyperparameters. After this initial spike, both the training and validation loss both remain relatively low and stable, with only slight fluctuations, typical as the model adjusts its weights based on learning. Validation loss closely mirrors the training loss, suggesting that the model is not overfitting significantly at this point. However,

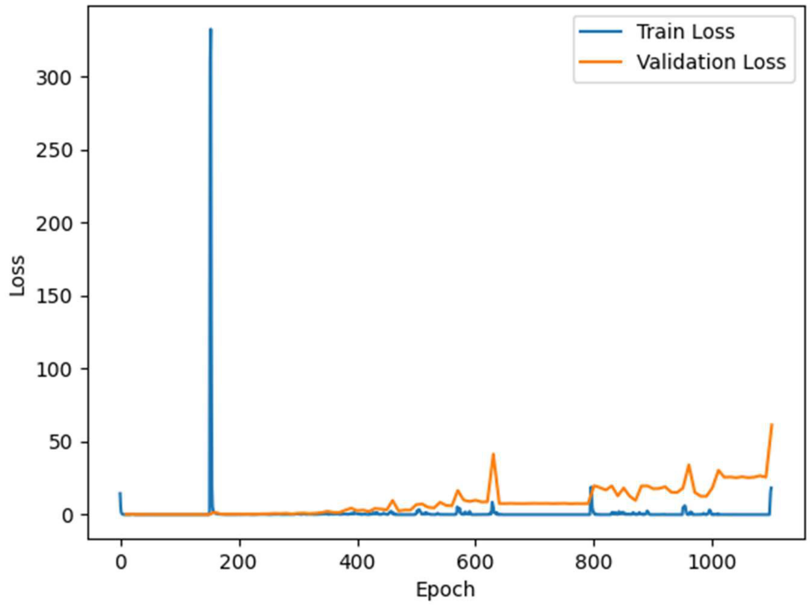

Figure 15: Trained model training and validation loss graph.

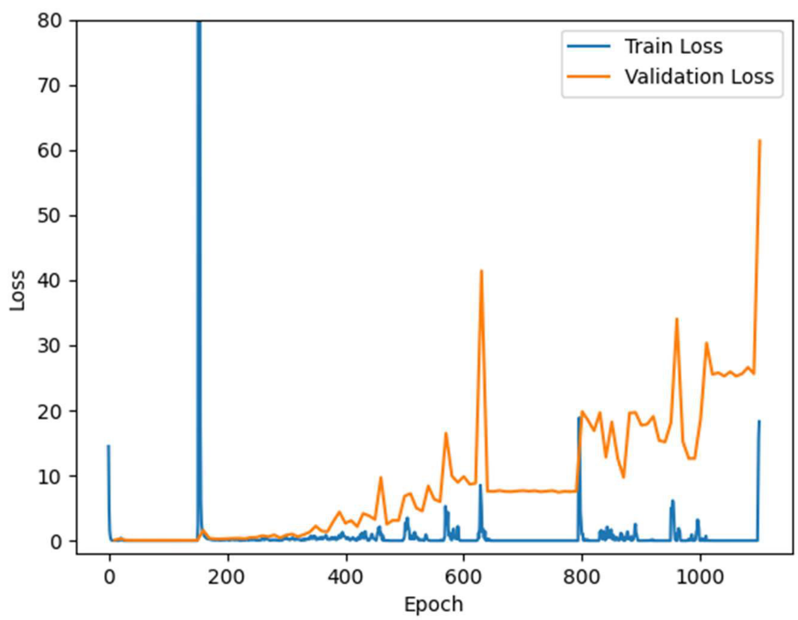

Figure 16: Trained model training and validation loss graph with maximum loss limited to 80.

at around 800 epochs, there is noticeable variability in both training and validation loss, with an increased number of spikes particularly in the validation loss. This could suggest that the model may be starting to overfit, or that it is sensitive to specific batches of data. It may also suggest that the learning rate needs to be adjusted in order to stabilise the loss towards the end of the training.

6.2.4 Confusion Matrices

Figure 17 and Figure 18 display confusion matrices of Checkpoint 130 and the

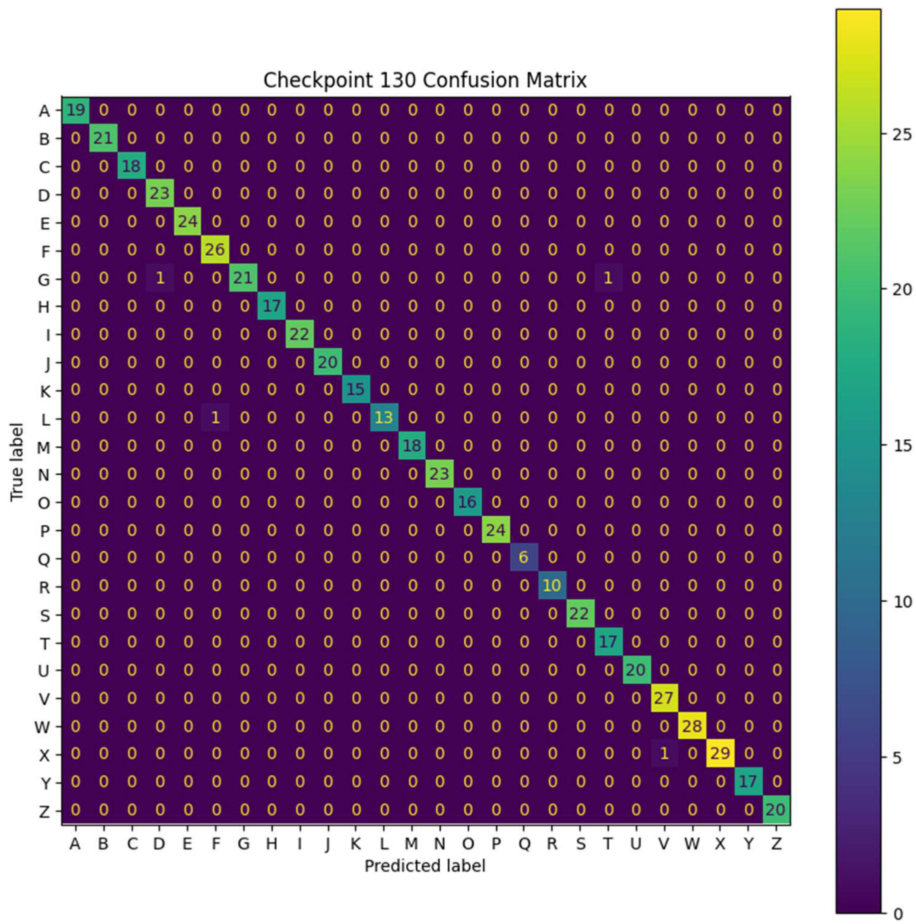

Figure 17: Confusion Matrix for Checkpoint 130

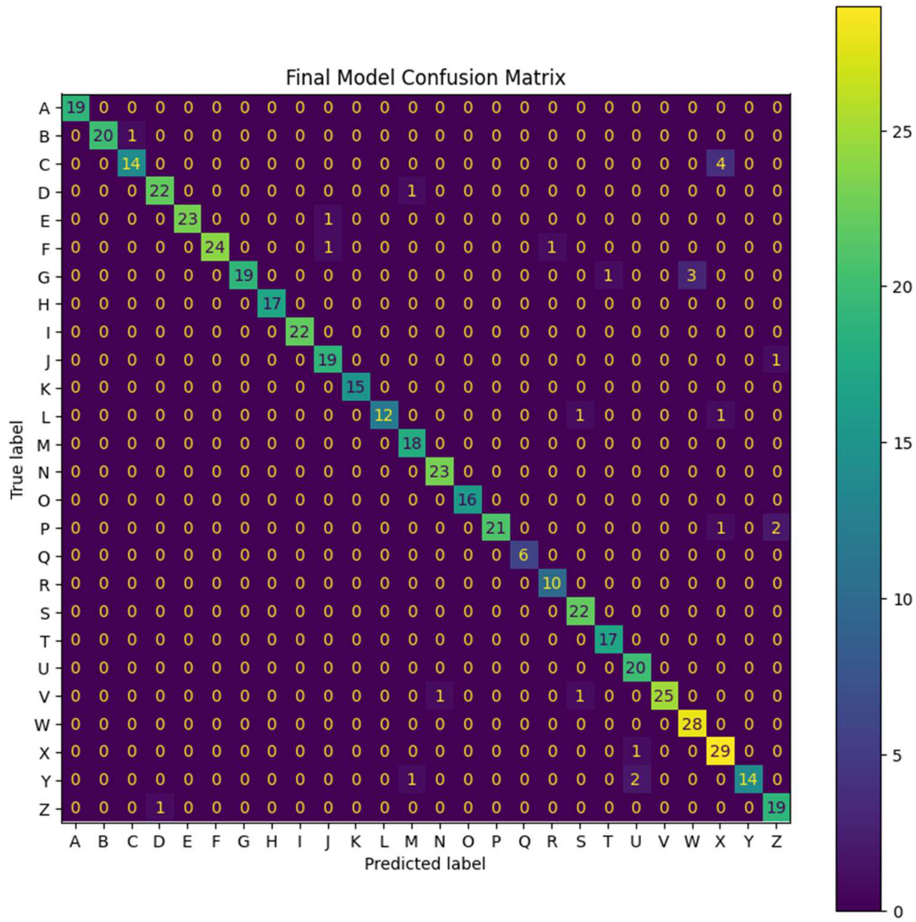

Figure 18: Confusion Matrix for the final model.

final model respectively. A confusion matrix is used to evaluate the performance of a classification model by comparing predicted classifications against the actual classifications (Ting 2017). That is, the matrix records the number of times each label on the x-axis was classified by the model as being the label on the y-axis. Thus, in a perfect scenario, the cells where $x = y$ would contain the value of the number of instances of each label in the dataset, and any cells where $x \neq y$ would contain 0. As can be seen in Figure 17, Checkpoint 130's confusion matrix has essentially all 0s in cells where $x \neq y$, with only 4 exceptions. These exceptions are highlighted in Table 4, which indicates that out of four misclassifications, one was only one key away from the correct key, and that two others were only two keys away.

Table 4: Misclassified labels when evaluating test dataset using Checkpoint 130.

True Label	**Predicted Label**	**Distance Between Keys**
G	T	1
X	V	2
G	D	2
L	F	5

The case is similar when looking at Figure 18. The final model's confusion matrix has essentially all 0s in cells where $x \neq y$, however, there is a higher number of exceptions to this case, with 26 misclassified labels. Table 5 highlights the four most common misclassifications. The most misclassified key pairing, C with X, was misclassified 4 times, however, C and X are only one key away from each other on the keyboard, so this is not an extremely unusual result. The second most misclassified key pairing, G with W, was misclassified 3 times, and is less explainable. It is likely that the final model struggled with differentiating these keys due to it overfitting towards the end of training, and therefore failing to fit unseen data.

Table 5: Four most common misclassifications when evaluating the test dataset using the final model.

True Label	**Predicted Label**	**Misclassifications**
C	X	4
G	W	3
Y	U	2
P	Z	2

6.3 Evaluation on Different Keyboard

Whilst this model was never expected nor intended to be compatible with any keyboard other than the one used for training, the model was evaluated on an entire dataset of data from a different keyboard. Specifically, the model was evaluated on a dataset of keystrokes from a Dell Inspiron 7415 2-in-1 (Dell Inc. 2024). This dataset contains 25 keystrokes of each of the 26 alphabetic keys, resulting in 650 samples. When compared to the mechanical keyboard used to train the model, this laptop keyboard is much quieter, with keys featuring a much shorter travel distance. It was expected that the model would simply not be able to differentiate the keys from each other, and this was the case.

6.3.1 Classification Report

Table 6 displays a summary of the classification report for Checkpoint 130 evaluating the laptop keyboard. As can be seen, the model performed extremely poorly, achieving an average precision of 0 and accuracy F_1-score of 0.04. The full classification report detailing the metrics for each class can be found in Appendix 4.

Table 6: Classification Report for Checkpoint 130 evaluating laptop keyboard.

	Precision	Recall	F_1-Score	Support
Accuracy	–	–	0.04	650
Macro Average	0.00	0.04	0.00	650
Weighted Average	0.00	0.04	0.00	650

6.3.2 Confusion Matrix

Figure 19 displays the confusion matrix of Checkpoint 130 evaluating the laptop keyboard. As mentioned, ideally, all the cells where $x \neq y$ would contain 0 and all cells where $x = y$ would contain the number of instances of that class in the dataset, in this case 25. However, as can be seen in Figure 19, that is not the case. In all cases except 3, the model predicted the label of the sample provided to be C. It is most likely that the model simply could not distinguish the keys from each other and the closest key to classify the samples as was C. This supports the assumption that the keyboard would not work on different keyboards that it had not been trained on.

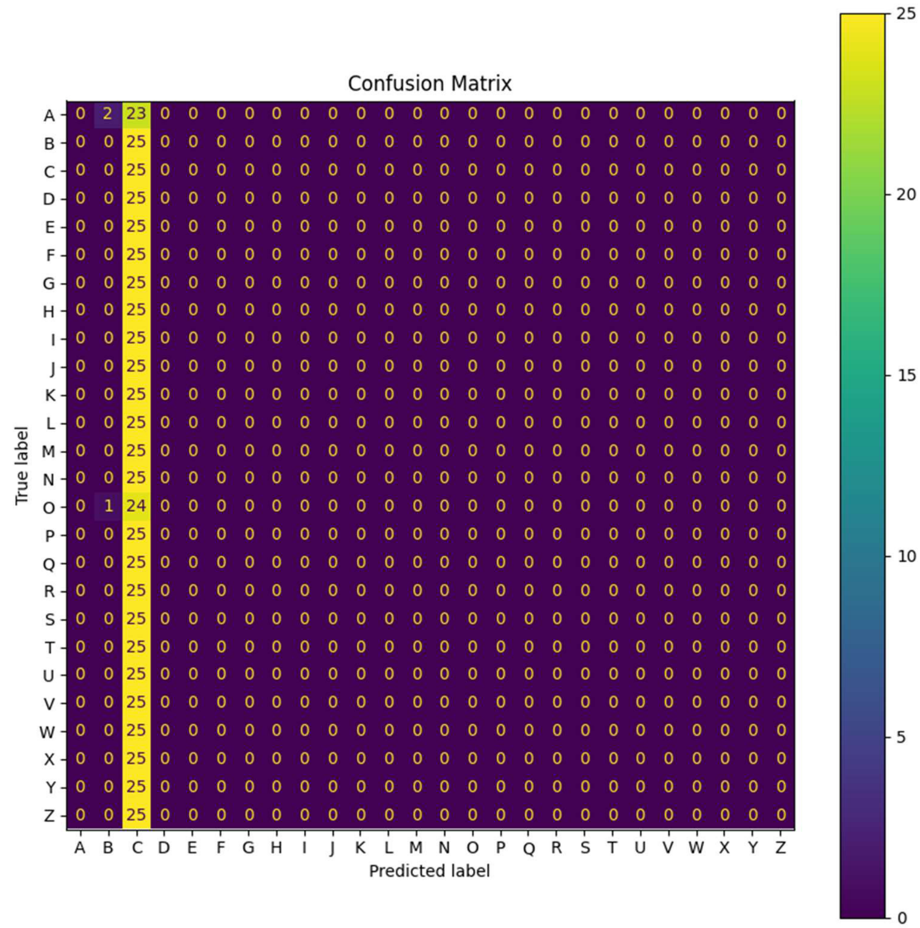

Figure 19: Confusion Matrix for Checkpoint 130 evaluating the laptop keyboard.

6.4 Potential Improvements

Whilst the trained models perform extremely well, an even more robust and effective model could be developed. The adjustments below serve as potential improvements to the training process of the model.

6.4.1 Adjustment of Learning Rate

The initial spike and later fluctuations in loss suggest that the learning rate may be suboptimal. Implementing a learning rate scheduler that decreases the learning rate over epochs may help eliminate these inconsistencies and ensure a smoother convergence, potentially leading to a more refined final model.

6.4.2 Data Examination

Anomalies in training data that coincide with loss spikes could impact the model's ability to learn optimally. Examining the batch of data during this spike could reveal an otherwise hidden anomaly in data. Cleansing this data could reduce these spikes and therefore the model's ability to converge.

6.4.3 Early Stopping

Implementing an early stopping mechanism could prevent overfitting and conserve computational resources, which can be costly, especially for multiple, long training sessions. For example, the Google Colab V100 GPU instance used in this case costs approximately 4.85 computational credits per hour, with 100 credits costing £9.72.

6.4.4 On-the-fly Batch Data Augmentation

Currently, the SpecAugment data augmentation technique is applied to the entire dataset at the beginning of the training process. Whilst this provides a static augmented dataset, it lacks the variability and randomness that could potentially lead to a more robust model. This could easily be implemented by integrating it into the custom RecordingDataset, which contains the function that provides the training dataset loader each data and target pair. This would ensure the model does not see the same version of input data twice.

7 Conclusions

To summarise, this report has demonstrated the feasibility and effectiveness of a keystroke recovery system that utilises Mel-spectrograms and deep learning techniques. The proposed system successfully incorporates modern advancements in audio processing and machine learning, specifically through the implementation of a CoAtNet model combined with SpecAugment data augmentation. The evaluation of the model demonstrated exceptional accuracy, with Checkpoint 130 achieving an impressive 99% accuracy rate on the 520-character test dataset, while the final model had a respectable 95% accuracy rate. These results imply a robust system that is confident in extracting and identifying keystrokes based on acoustic emanations.

The project's success helps highlight the potential for acoustic side-channel attacks to be a real threat to sensitive information, underscoring the importance of understanding and mitigating such vulnerabilities. The project's success also highlights how advancements in machine learning can effectively be applied to enhance side-channel attack methodologies, thereby raising awareness of potential cyber threats and the need for improved security measures.

8 Future Works

While the developed solution performed respectfully, several areas for future improvement and research were identified, as outlined below.

1. The initial spike and later fluctuations in loss suggest that the **learning rate may be suboptimal**. This could be improved by **implementing a learning rate scheduler** to decrease the learning rate over epochs to reduce inconsistencies and produce a more refined final mode.
2. Another level of **analysis of the dataset**, particularly anomalies that coincide with loss spikes, could reveal issues affecting the model's ability to learn optimally. Addressing this could allow the model to converge better.
3. Implementing **early stopping** could prevent overfitting of the final model and conserve computational resources which would be beneficial for long training sessions considering the costs associated with cloud-based GPU instances like Google Colab.
4. Enhancing **data augmentation** by incorporating an **on-the-fly** approach to each batch would add much more variability and randomness to the training data, potentially leading to a more robust model that is encouraged to generalise more.
5. **Expanding the dataset** to include the space bar, numeric, and special characters would allow the system to be used in a much wider range of situations and scenarios. This would be simple to implement by increasing the number of channels in the classifier but was not achieved due to time constraints.
6. Future work could involve **testing the system in more varied real-world scenarios** to assess its robustness against different environmental noise levels, microphone placements, keyboard types, and typing styles.

9 Reflection on Learning

The development of this project has been an enriching experience, providing insight into the theoretical, technical, and practical aspects of machine learning, signal processing, and cybersecurity. I have learnt several key lessons and valuable skills that will be beneficial for both my personal and professional development.

Whilst I had a structured plan for completing this report, some stages, specifically developing code to train the model, took far longer than originally anticipated, impacting the overall timeline. I had assumed that I was aware of the amount of time each stage would take, when in reality, this was not the case. I should have researched the amount of time the more challenging stages would potentially take me and allocated my time to these stages accordingly. I had originally dedicated two weeks to developing code to extract keystrokes and their features, two weeks to developing code to train the model, and another two weeks to developing code to recover keystrokes from the model. However, I spent three weeks, four weeks, and one week on these stages respectfully, meaning I overran and had to restructure the remainder of my time around the tasks still to complete. This experience has emphasised the importance of thorough preparation and realistic time management.

Leading on from the point above, whilst I was extremely eager to learn, my knowledge of machine learning was fairly minimal at the beginning of the project. I had assumed I would be able to pick up the basics fairly quickly by simply reviewing other implementations of similar projects, however, due to the number of elements required to bring a machine learning project together, this was not the case. Consequentially, I found dedicating focused time to learning and ensuring I understood the fundamentals was crucial in enhancing my knowledge and skills in the machine learning field, highlighting the importance of ongoing education in areas outside of my current knowledge space.

Given that I had prior knowledge of audio processing, I had assumed that this would be enough to understand and comprehend that portion of the project. However, this project required me to recollect and refresh my understanding, reinforcing the value of always continuing to build on foundational knowledge and revisit familiar topics to ensure complete understanding.

Initially, I had attempted to use my own hardware to develop, train, and analyse the model. Given that I have a dedicated Nvidia GPU, I had assumed it would be possible to use CUDA and cuDNN to train the model. However, it quickly became apparent that this would not be sufficient given the amount of video memory (VRAM) required to load the complex deep learning model chosen and all the Mel-spectrograms. I had underestimated the benefits of using a hosted development service that provides dedicated GPU resources, like Google Colab on even the free tier. Once I had made the switch, the performance improvements significantly streamlined the development and testing processes, allowing me to test many more potential hyperparameters in a much shorter amount of time. This experience taught me the importance of leveraging available resources and services to enhance development efficiency, and to not rule out any options without giving them a fair trial.

Given my reliance of personal hardware mentioned above, each training run took a considerable amount of time and therefore limited my ability to experiment with different hyperparameters effectively. Given my method was based on existing literature, I believed using the hyperparameters mentioned in the paper would be sufficient for my training set, however, this was not the case, and a fair amount of experimentation was required to find ideal values. I believe if I had switched to using Colab sooner, and had more time to experiment, I could have found even more optimal hyperparameters, potentially improving the model's performance.

Continuing this point, I believe the challenges I faced with spikes in the loss graph were compounded by my initial choice to use personal hardware. More time and experimentation with hyperparameters may have improved this.

Despite these issues, the project was ultimately successful and achieved an extremely respectable accuracy. This demonstrates that even in the event of unforeseen challenges, it is possible to adapt around them and overcome them. Through this process, I have learnt the importance of continually refining my approach to any task or challenge I may face, and that any setbacks or difficulties are, in essence, extremely valuable learning opportunities. I believe I have improved on skills I already possessed, especially Python development, as well as learnt many new skills in the fields of machine learning, signal and audio processing, and cybersecurity. I have grown a deeper appreciation for the synergy between theoretical concepts and practical applications, seeing what started as a simple written plan turn into a working application.

10 Bibliography

Acharya, N. 2024. *Understanding Precision, Recall, F1-score, and Support in Machine Learning Evaluation | by Nirajan Acharya | Medium*. Available at: https://medium.com/@nirajan.acharya666/understanding-precision-recall-f1-score-and-support-in-machine-learning-evaluation-7ec935e8512e [Accessed: 2 May 2024].

Allodi, L., Cremonini, M., Massacci, F. and Shim, W. 2018. The Effect of Security Education and Expertise on Security Assessments: the Case of Software Vulnerabilities. Available at: https://arxiv.org/abs/1808.06547v1 [Accessed: 4 May 2024].

Amazon. 2024. *havit Wired Mechanical Gaming Keyboard and RGB Mouse Combo Set UK Layout, Blue Switch Mechanical Anti-Ghosting Keyboard 105 Keys + 4800Dots Per Inch Programmable Wired Gaming Mouse, Black : Amazon.co.uk: PC & Video Games*. Available at: https://www.amazon.co.uk/gp/product/B07XHJ4QK6/ [Accessed: 29 April 2024].

Asonov, D. and Agrawal, R. 2004. Keyboard acoustic emanations. *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004* 2004, pp. 3–11. Available at: https://doi.org/10.1109/SECPRI.2004.1301311 [Accessed: 26 April 2024].

Atkinson, K. 2023. GNU Aspell. Available at: http://aspell.net/ [Accessed: 5 May 2024].

Baek, S. and Kim, Y.G. 2019. Efficient Vulnerability Management Process in the Military. *2019 International Conference on Platform Technology and Service (PlatCon)*, pp. 1–5. Available at: https://doi.org/10.1109/PLATCON.2019.8669420 [Accessed: 4 May 2024].

Bahmei, B., Birmingham, E. and Arzanpour, S. 2022. CNN-RNN and Data Augmentation Using Deep Convolutional Generative Adversarial Network for Environmental Sound Classification. *IEEE Signal Processing Letters* 29, pp.

682–686. Available at: https://doi.org/10.1109/lsp.2022.3150258 [Accessed: 27 April 2024].

Banerjee, S., Syed, Z., Bartlow, N. and Cukic, B. 2015. Keystroke Recognition. *Encyclopedia of Biometrics*, pp. 1067–1073. doi: 10.1007/978-1-4899-7488-4_205.

Cardaioli, M., Cecconello, S., Conti, M., Milani, S., Picek, S. and Saraci, E. 2021. Hand Me Your PIN! Inferring ATM PINs of Users Typing with a Covered Hand. *Proceedings of the 31st USENIX Security Symposium, Security 2022*, pp. 1687–1704. Available at: https://arxiv.org/abs/2110.08113v3 [Accessed: 3 May 2024].

Cayir, A., Yenidogan, I. and Dag, H. 2018. Feature Extraction Based on Deep Learning for Some Traditional Machine Learning Methods. *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, pp. 494–497. Available at: https://ieeexplore.ieee.org/document/8566383/ [Accessed: 9 May 2024].

Cheng, P., Ethem Bagci, I., Roedig, U. and Yan, J. 2020. SonarSnoop: active acoustic side-channel attacks. *International Journal of Information Security* 19, pp. 213–228. Available at: https://doi.org/10.1007/s10207-019-00449-8.

Ciric, D., Peric, Z., Nikolic, J. and Vucic, N. 2021. Audio Signal Mapping into Spectrogram-Based Images for Deep Learning Applications. *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pp. 1–6. Available at: https://doi.org/10.1109/INFOTEH51037.2021.9400698 [Accessed: 27 April 2024].

da Costa-Luis, C. et al. 2024. tqdm: A fast, Extensible Progress Bar for Python and CLI. Available at: https://zenodo.org/records/3551211 [Accessed: 1 May 2024].

Dai, Z., Liu, H., Le, Q. V. and Tan, M. 2021. CoAtNet: Marrying Convolution and Attention for All Data Sizes. *Advances in Neural Information Processing Systems* 5, pp. 3965–3977. Available at: https://arxiv.org/abs/2106.04803v2 [Accessed: 30 April 2024].

Deng, L. and Yu, D. 2014. Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing* 7(3–4), pp. 197–387. Available at: http://dx.doi.org/10.1561/2000000039 [Accessed: 9 May 2024].

Dennis, J., Tran, H.D. and Li, H. 2011. Spectrogram image feature for sound event classification in mismatched conditions. *IEEE Signal Processing Letters* 18(2), pp. 130–133. doi: 10.1109/LSP.2010.2100380.

FFmpeg developers. 2024. FFmpeg. Available at: https://ffmpeg.org/ [Accessed: 1 May 2024].

Gouthier, R. and Ponto, S. 1970. Designing Systems Programs. p. 274. Available at: http://www.amazon.com/dp/0132019620 [Accessed: 25 April 2024].

Grassi, P.A., Garcia, M.E. and Fenton, J.L. 2017. Digital identity guidelines: revision 3. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf [Accessed: 25 April 2024].

Guo, Y., Sun, L., Zhang, Z. and He, H. 2019. Algorithm Research on Improving Activation Function of Convolutional Neural Networks. *Proceedings of the 31st Chinese Control and Decision Conference, CCDC 2019*, pp. 3582–3586. doi: 10.1109/CCDC.2019.8833156.

Harrison, J., Toreini, E. and Mehrnezhad, M. 2023. A Practical Deep Learning-Based Acoustic Side Channel Attack on Keyboards. *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 270–280. doi: 10.1109/EUROSPW59978.2023.00034.

Dell Inc. 2024. *Inspiron 7415 2-in-1 Setup and Specifications | Dell US*. Available at: https://www.dell.com/support/manuals/en-us/inspiron-14-7415-2-in-1-laptop/inspiron-7415-2-in-1-setup-and-specifications/keyboard?guid=guid-f4d96b7c-8a4a-4f24-8b88-1f81d17905bb [Accessed: 10 May 2024].

Ismailov, V.E. 2014. On the approximation by neural networks with bounded number of neurons in hidden layers. *Journal of Mathematical Analysis and Applications* 417(2), pp. 963–969. doi: 10.1016/J.JMAA.2014.03.092.

Kamiya, S., Kang, J.-K., Kim, J., Milidonis, A. and Stulz, R.M. 2019. Risk Management, Firm Reputation, and the Impact of Successful Cyberattacks on Target Firms. *SSRN Electronic Journal*. Available at: https://papers.ssrn.com/abstract=3135514 [Accessed: 3 May 2024].

Kaneko, T., Tanaka, K., Kameoka, H. and Seki, S. 2022. iSTFTNet: Fast and Lightweight Mel-Spectrogram Vocoder Incorporating Inverse Short-Time Fourier Transform. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* 2022-May, pp. 6207–6211. Available at: https://arxiv.org/abs/2203.02395v1 [Accessed: 27 April 2024].

Kriegeskorte, N. and Golan, T. 2019. Neural network models and deep learning. *Current Biology* 29(7), pp. R231–R236. doi: 10.1016/J.CUB.2019.02.034.

Lillicrap, T.P., Santoro, A., Marris, L., Akerman, C.J. and Hinton, G. 2020. Backpropagation and the brain. *Nature Reviews Neuroscience 2020 21:6* 21(6), pp. 335–346. Available at: https://www.nature.com/articles/s41583-020-0277-3 [Accessed: 9 May 2024].

Loughlin, P.J. 2009. Time-frequency and position-wavenumber acoustic signal analysis. *The Journal of the Acoustical Society of America* 126(4_Supplement), pp. 2206–2206. Available at: /asa/jasa/article/126/4_Supplement/2206/654229/Time-frequency-and-position-wavenumber-acoustic [Accessed: 27 April 2024].

McFee, B. et al. 2023. librosa/librosa: 0.10.1. Available at: https://zenodo.org/records/8252662 [Accessed: 30 April 2024].

National Security Agency. 1982. *NACSIM 5000 Tempest Fundamentals*. Available at: https://cryptome.org/jya/nacsim-5000/nacsim-5000.htm [Accessed: 1 February 2024].

Park, D.S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E.D. and Le, Q. V. 2019. SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* 2019-September, pp. 2613–2617. Available at: http://arxiv.org/abs/1904.08779 [Accessed: 4 May 2024].

Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12), pp. 1053–1058. Available at: https://dl.acm.org/doi/10.1145/361598.361623 [Accessed: 25 April 2024].

Pedersen, P. 1965. The Mel Scale. *Journal of Music Theory* 9(2), p. 295. doi: 10.2307/843164.

Pedregosa, F. et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, pp. 2825–2830. Available at: http://scikit-learn.sourceforge.net. [Accessed: 2 May 2024].

Perez, A.F., Sanguineti, V., Morerio, P. and Murino, V. 2020. Audio-Visual Model Distillation Using Acoustic Images. *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 2843–2852. Available at: https://doi.org/10.1109/WACV45572.2020.9093307 [Accessed: 27 April 2024].

Piland, J.C., Czajka, A. and Sweet, C. 2023. Model Focus Improves Performance of Deep Learning-Based Synthetic Face Detectors. *IEEE Access* 11, pp. 63430–63441. Available at: https://doi.org/10.1109/ACCESS.2023.3282927 [Accessed: 10 May 2024].

PyTorch. [no date][a]. *torch.optim.Optimizer.zero_grad — PyTorch 2.3 documentation*. Available at: https://pytorch.org/docs/stable/generated/torch.optim.Optimizer.zero_grad.html [Accessed: 1 May 2024].

PyTorch. [no date][b]. *Zeroing out gradients in PyTorch — PyTorch Tutorials 2.3.0+cu121 documentation*. Available at: https://pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html [Accessed: 1 May 2024].

Riazi, M.S., Darvish Rouani, B. and Koushanfar, F. 2019. Deep Learning on Private Data. *IEEE Security and Privacy* 17(6), pp. 54–63. doi: 10.1109/MSEC.2019.2935666.

Roth, J., Liu, X., Ross, A. and Metaxas, D. 2015. Investigating the Discriminative Power of Keystroke Sound. *IEEE Transactions on Information Forensics and Security* 10(2), pp. 333–345. Available at: https://doi.org/10.1109/TIFS.2014.2374424 [Accessed: 26 April 2024].

Samsung. 2024. *Voice Recorder*. Available at: https://galaxystore.samsung.com/prepost/000004401542 [Accessed: 29 April 2024].

Samsung UK. 2024. *Specifications | Samsung Galaxy S10 | Samsung UK*. Available at: https://www.samsung.com/uk/smartphones/galaxy-s10/specs/ [Accessed: 29 April 2024].

Sharan, R. V. and Moir, T.J. 2019. Acoustic event recognition using cochleagram image and convolutional neural networks. *Applied Acoustics* 148, pp. 62–66. doi: 10.1016/J.APACOUST.2018.12.006.

Stevens, S.S., Volkmann, ; J and Newman, ; E B. 1937. A Scale for the Measurement of the Psychological Magnitude Pitch ⊠. *J. Acoust. Soc. Am* 8, pp. 185–190. Available at: https://doi.org/10.1121/1.1915893 [Accessed: 27 April 2024].

The Mathworks Inc. 2024a. Design probabilistic neural network - MATLAB newpnn - MathWorks United Kingdom. Available at: https://uk.mathworks.com/help/deeplearning/ref/newpnn.html [Accessed: 5 May 2024].

The Mathworks Inc. 2024b. (Not recommended) Classify data using trained deep learning neural network - MATLAB classify - MathWorks United Kingdom. Available at: https://uk.mathworks.com/help/deeplearning/ref/seriesnetwork.classify.html [Accessed: 5 May 2024].

The pandas development team. 2024. pandas-dev/pandas: Pandas. Available at: https://zenodo.org/records/10697587 [Accessed: 30 April 2024].

Thomas, T.G., Pandey, P.C. and Agashe, S.D. 1994. A PC-Based Multi-resolution Spectrograph. *Iete Journal of Research* 40(2–3), pp. 105–108. Available at: https://doi.org/10.1080/03772063.1994.11437177 [Accessed: 27 April 2024].

Ting, K.M. 2017. Confusion Matrix. *Encyclopedia of Machine Learning and Data Mining*, pp. 260–260. Available at: https://doi.org/10.1007/978-1-4899-7687-1_50 [Accessed: 2 May 2024].

Vieira, A. 2017. Business applications of deep learning. *Ubiquitous Machine Learning and Its Applications*, pp. 39–67. doi: 10.4018/978-1-5225-2545-5.CH003.

Wu, C.-H. 2021. coatnet-pytorch/coatnet.py at master · chinhsuanwu/coatnet-pytorch. Available at: https://github.com/chinhsuanwu/coatnet-pytorch/blob/master/coatnet.py [Accessed: 30 April 2024].

Zhuang, L., Zhou, F. and Tygar, J.D. 2009. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)* 13(1). Available at: https://dl.acm.org/doi/10.1145/1609956.1609959 [Accessed: 1 February 2024].

11 Appendices

11.1 Appendix 1

```
PCRecordings\A.wav
Peaks not found.

PCRecordings\B.wav
np.diff(peaks): [184 190 191 182 193 186 191 190 181 176 195 176
184 180 193 180 180 191 178 196 185 193 185 190]
min(np.diff(peaks)): 176
Correctly identified peaks? (y/n): y

PCRecordings\C.wav
np.diff(peaks): [184 185 192 182 177 186 175 176 196 193 188 187
196 171 221 192 207 188 188 198 198 195 194 188]
min(np.diff(peaks)): 171
Correctly identified peaks? (y/n): y

PCRecordings\D.wav
Peaks not found.

PCRecordings\E.wav
np.diff(peaks): [ 26 163 188 212 206 213 193 186 195 193 190 187
194 194 413 207 194 191 187 213 210 200 200 186]
min(np.diff(peaks)): 26
Correctly identified peaks? (y/n): n

PCRecordings\F.wav
np.diff(peaks): [206 199 201 209 214 216 198 203 204 402 203 203
197 209 220 199 207 214 204 214 187 200  30 179]
min(np.diff(peaks)): 30
Correctly identified peaks? (y/n): n
```

```
PCRecordings\G.wav
np.diff(peaks): [189 189 193 207 197 206 217 205 208 198 199 185
192 198 184 194 206 203 202 209 205 197 183 196]
min(np.diff(peaks)): 183
Correctly identified peaks? (y/n): y

PCRecordings\H.wav
np.diff(peaks): [190 185 206 202 196 197 204 183 166 183 201 195
197 180 203 186 191 202 187 178 189 180 185 198]
min(np.diff(peaks)): 166
Correctly identified peaks? (y/n): y

PCRecordings\I.wav
np.diff(peaks): [165 171 180 186 180 193 188 178 187 168 169 175
186 215 171 180 174 169 167 186 186 185 190 183]
min(np.diff(peaks)): 165
Correctly identified peaks? (y/n): y

PCRecordings\J.wav
np.diff(peaks): [190 178 188 188 188 195 180 186 198 193 195 235
157 180 182 175 176 179 181 191 184 180 181 189]
min(np.diff(peaks)): 157
Correctly identified peaks? (y/n): y

PCRecordings\K.wav
np.diff(peaks): [186 175 179 186 182 180 171 172 179 173 173 161
159 164 165 174 175 167 179 179 171 163 181 187]
min(np.diff(peaks)): 159
Correctly identified peaks? (y/n): y

PCRecordings\L.wav
np.diff(peaks): [168 172 170 168 171 168 161 165 163 191 176 162
163 175 172 167 168 162 170 163 163 166 163 170]
min(np.diff(peaks)): 161
```

```
Correctly identified peaks? (y/n): y

PCRecordings\M.wav
np.diff(peaks): [153 164 168 141 155 149 162 166 162 146 157 184
170 172 156 166 169 164 155 190 159 160 163 177]
min(np.diff(peaks)): 141
Correctly identified peaks? (y/n): y

PCRecordings\N.wav
np.diff(peaks): [160 147 180 164 153 177 165 154 170 165 169 168
181 180 157 165 161 167 185 177 199 172 195 192]
min(np.diff(peaks)): 147
Correctly identified peaks? (y/n): y

PCRecordings\O.wav
np.diff(peaks): [160 159 159 165 156 165 156 154 157 165 157 163
163 157 155 160 162 152 161 161 151 159 153 158]
min(np.diff(peaks)): 151
Correctly identified peaks? (y/n): y

PCRecordings\P.wav
np.diff(peaks): [159 164 161 164 160 169 157 154 165 176 173 167
162 156 163 149 160 163 181 179 176 175 166 180]
min(np.diff(peaks)): 149
Correctly identified peaks? (y/n): y

PCRecordings\Q.wav
np.diff(peaks): [154 154 175 177 161 166 185 149 164 144 152 151
161 150 161 143 157 152 155 156 164 156 162 167]
min(np.diff(peaks)): 143
Correctly identified peaks? (y/n): y

PCRecordings\R.wav
np.diff(peaks): [140 143  32 128 174 168 171 177 160 167 160 174
```

```
155 161 166 172 175 169 171 173 144 170 336 180]
min(np.diff(peaks)): 32
Correctly identified peaks? (y/n): n

PCRecordings\S.wav
np.diff(peaks): [147 144 162 186 193 177 180 177 174 191 174 177
177 171 184 199 183 208 180 186 207 170 185 180]
min(np.diff(peaks)): 144
Correctly identified peaks? (y/n): y

PCRecordings\T.wav
np.diff(peaks): [147 171 162 148 168 146 153 151 139 139 182 175
169 175 162 163 160 165 156 165 187 173 168 163]
min(np.diff(peaks)): 139
Correctly identified peaks? (y/n): y

PCRecordings\U.wav
np.diff(peaks): [148 156 162 162 162 158 157 158 145 177 155 154
156 167 158 153 144 161 158 151 158 152 160 163]
min(np.diff(peaks)): 144
Correctly identified peaks? (y/n): y

PCRecordings\V.wav
np.diff(peaks): [156 152 155 151 145 146 142 148 133 152 147 135
125 139 148 136 137 139 141 154 152 161 155 131]
min(np.diff(peaks)): 125
Correctly identified peaks? (y/n): y

PCRecordings\W.wav
np.diff(peaks): [134 173 197 186 197 190 184 210 185 188 171 176
173 179 197 198 209 176 181 187 195 205 195 196]
min(np.diff(peaks)): 134
Correctly identified peaks? (y/n): y
```

```
PCRecordings\X.wav
np.diff(peaks): [154 168 162 162 191 174 194 179 219 211 190 190
212 201 172 203 194 170 191 183 174 182 170 184]
min(np.diff(peaks)): 154
Correctly identified peaks? (y/n): y

PCRecordings\Y.wav
np.diff(peaks): [148 154 159 150 164 163 170 159 169 165 167 195
168 186 205 182 182 174 206 194 176 190 176 208]
min(np.diff(peaks)): 148
Correctly identified peaks? (y/n): y

PCRecordings\Z.wav
np.diff(peaks): [171 160 175 172 172 176 183 172 183 185 185 181
187 183 189 180 224 172 188 192 200 194 184 188]
min(np.diff(peaks)): 160
Correctly identified peaks? (y/n): y

Minimum difference between peaks: 125
```

11.2 Appendix 2

	Precision	Recall	F1-Score	Support
A	1	1	1	19
B	1	1	1	21
C	1	1	1	18
D	0.96	1	0.98	23
E	1	1	1	24
F	0.96	1	0.98	26
G	1	0.91	0.95	23
H	1	1	1	17
I	1	1	1	22
J	1	1	1	20

K	1	1	1	15
L	1	0.93	0.96	14
M	1	1	1	18
N	1	1	1	23
O	1	1	1	16
P	1	1	1	24
Q	1	1	1	6
R	1	1	1	10
S	1	1	1	22
T	0.94	1	0.97	17
U	1	1	1	20
V	0.96	1	0.98	27
W	1	1	1	28
X	1	0.97	0.98	30
Y	1	1	1	17
Z	1	1	1	20
Accuracy	–	–	0.99	520
Macro Average	0.99	0.99	0.99	520
Weighted Average	0.99	0.99	0.99	520

11.3 Appendix 3

	Precision	**Recall**	**F_1-Score**	**Support**
A	1	1	1	19
B	1	0.95	0.98	21
C	0.93	0.78	0.85	18
D	0.96	0.96	0.96	23
E	1	0.96	0.98	24
F	1	0.92	0.96	26

G	1	0.83	0.9	23
H	1	1	1	17
I	1	1	1	22
J	0.9	0.95	0.93	20
K	1	1	1	15
L	1	0.86	0.92	14
M	0.9	1	0.95	18
N	0.96	1	0.98	23
O	1	1	1	16
P	1	0.88	0.93	24
Q	1	1	1	6
R	0.91	1	0.95	10
S	0.92	1	0.96	22
T	0.94	1	0.97	17
U	0.87	1	0.93	20
V	1	0.93	0.96	27
W	0.9	1	0.95	28
X	0.83	0.97	0.89	30
Y	1	0.82	0.9	17
Z	0.86	0.95	0.9	20
Accuracy	–	–	0.95	520
Macro Average	0.96	0.95	0.95	520
Weighted Average	0.95	0.95	0.95	520

11.4 Appendix 4

	Precision	**Recall**	**F_1-Score**	**Support**
A	0.00	0.00	0.00	25
B	0.00	0.00	0.00	25
C	0.04	1.00	0.07	25

D	0.00	0.00	0.00	25
E	0.00	0.00	0.00	25
F	0.00	0.00	0.00	25
G	0.00	0.00	0.00	25
H	0.00	0.00	0.00	25
I	0.00	0.00	0.00	25
J	0.00	0.00	0.00	25
K	0.00	0.00	0.00	25
L	0.00	0.00	0.00	25
M	0.00	0.00	0.00	25
N	0.00	0.00	0.00	25
O	0.00	0.00	0.00	25
P	0.00	0.00	0.00	25
Q	0.00	0.00	0.00	25
R	0.00	0.00	0.00	25
S	0.00	0.00	0.00	25
T	0.00	0.00	0.00	25
U	0.00	0.00	0.00	25
V	0.00	0.00	0.00	25
W	0.00	0.00	0.00	25
X	0.00	0.00	0.00	25
Y	0.00	0.00	0.00	25
Z	0.00	0.00	0.00	25
Accuracy	–	–	0.04	650
Macro Average	0.00	0.04	0.00	650
Weighted Average	0.00	0.04	0.00	650

11.5 Appendix 5

Source code can be found at: https://github.com/moahmed0987/KRAMS