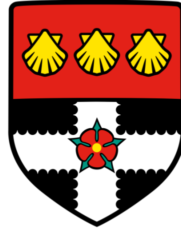

University of Reading

Department of Computer Science

Audio Classification using Machine Learning Techniques

Adam Taemur

Supervisor: Dr. Varun Ojha

A report submitted in partial fulfilment of the requirements of the University of Reading for the degree of Bachelor of Science in *Computer Science*

April 2020

Declaration

I, Adam Taemur, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Adam Taemur
April 2020

Abstract

Environmental Sound Classification is a subfield of audio classification that aims to classify sounds commonly heard in urban settings, such as vehicles, industrial tools, children, and music. Existing literature uses different deep learning models such as convolutional neural networks or pre-trained models to accurately classify relevant datasets to a high level of accuracy, with some achieving state-of-the-art levels. However, many fail to take into account the number of model parameters, which can lead to issues such as long training times, high storage space demand, and computationally intensive training and evaluation. This project implements two of the best performing models trained on the UrbanSound8K dataset and perform experiments that aim to maintain or increase the performance while reducing the overall number of model parameters. The best performing model contains 28% less parameters than its' original and achieved a marginally higher level of accuracy. A GUI is also developed to showcase the performance of the model against individual, unseen audio data. Users can load their own .WAV file and .h5 model and get a list of predictions the model makes for the audio file, which displays these predictions in a clear and easy to understand format.

Keywords: Machine Learning, Deep Learning, Neural Networks, Python, GUI

Report's total word count: 19,791 words (excluding reference and appendices) and 59 pages.

Acknowledgements

Dr. Varun Ojha, for the support and guidance provided throughout all stages of this project as my supervisor.

Mike Smales, and Zohaib Mushtaq and Shun-Feng Su, for the initial implementations of the models used in the experimentation that led to the primary aim's success.

My friends among the Computer Science 3rd years, for their friendship and support throughout the project as well as the year in general.

My family, for their support and advice given during stressful moments.

Amy, for her unwavering support and love throughout the project and the year.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem statement . . . 2
- 1.3 Aims and objectives . . . 2
  - 1.3.1 Primary . . . 2
  - 1.3.2 Secondary . . . 2
- 1.4 Solution approach . . . 3
- 1.5 Summary of contributions and achievements . . . 3
- 1.6 Report Organisation . . . 4

2 Literature Review **5**
- 2.1 Audio Classification and Environmental Sound Classification . . . 5
- 2.2 Critique of the review . . . 10
- 2.3 Summary . . . 11

3 Concepts: CNNs and Spectrograms **12**
- 3.1 Convolutional Neural Networks . . . 12
- 3.2 Spectrograms . . . 13

4 Methodology **15**
- 4.1 Implementations . . . 15
- 4.2 Experimentation . . . 19
  - 4.2.1 Soundscape Models . . . 21
  - 4.2.2 ESC Models . . . 22
- 4.3 Implementation and GUI Development: Libraries Used . . . 24
  - 4.3.1 Librosa . . . 24
  - 4.3.2 Keras . . . 25
  - 4.3.3 Tkinter . . . 25
- 4.4 Graphical User Interface Development . . . 26
  - 4.4.1 Features and Design . . . 26
  - 4.4.2 Development: Versions . . . 32
- 4.5 Summary . . . 38

5 Results and Discussion **39**
- 5.1 Experiment Results . . . 39
  - 5.1.1 Soundscape Models . . . 39
  - 5.1.2 ESC Models . . . 42
  - 5.1.3 ESC vs. Soundscape: Scatter Plots . . . 44
  - 5.1.4 Tradeoff: Accuracy vs. Number of Parameters . . . 46

5.2 Best Performing Model 46
5.3 Graphical User Interface and Testing 48
5.4 Significance of the findings 49
5.5 Limitations 51
5.6 Summary 52

6 Conclusions and Future Work 53
6.1 Conclusions 53
6.2 Future work 54

7 Reflection 55

Appendices 58

A Appendix A: Code Snippets 58
A.1 Results and Discussion: GUI Error Handling 58

List of Figures

2.1 Top of Tree Diagram, showing the models and datasets used 10
2.2 Papers that use CNNs and the UrbanSound8K Dataset 10
2.3 The other papers 10

3.1 A typical convolutional neural network architecture 13

4.1 .WAV file metadata 18
4.2 Experiment Flowchart 23
4.3 Initial GUI Flowchart 32
4.4 Final GUI Flowchart 33
4.5 Version 1.0 of the GUI 34
4.6 Version 1.1 of the GUI 34
4.7 Version 1.2 of the GUI 35
4.8 Version 1.3 of the GUI 36
4.9 Version 1.4 of the GUI 37
4.10 The final version of the GUI. Visually similar, but contains additional error handling. 37

5.1 Soundscape Models using MFCCs 44
5.2 Soundscape Models using Spectrograms 44
5.3 Soundscape Models using Melspectrograms 45
5.4 ESC Models using Log-Melspectrograms 45
5.5 Comparison of all models. Points in gold indicate the best performing models for the ESC and Soundscape variants 45

List of Tables

4.1 Soundscape Models 21
4.2 ESC Models 22

5.1 Results for the MFCC Soundscape Models 40
5.2 Results for the Spectrogram Soundscape Models 40
5.3 Results for the Melspectrogram Soundscape Models 41
5.4 Results for the ESC Models 43
5.5 Model Comparison 50

List of Abbreviations

SMPCS	School of Mathematical, Physical and Computational Sciences.
ESC	Environmental Sound Classification; may refer to either the task, the dataset, or the implemented model.
CNN	Convolutional Neural Network.
RNN	Recurrent Neural Network.
ReLU	Rectified Linear Unit.
LSTM	Long Short-Term Memory.
MFCC	Mel-Frequency Cepstral Coefficients.
SOTA	State-of-the-Art.
US8K	UrbanSound8K dataset.
GUI	Graphical User Interface.

Chapter 1

Introduction

1.1 Background

The problem stated in this dissertation is to classify different types of sounds that are typically found in an urban environment, such as a town centre, into appropriate categories. Such categories include, but are not limited to: construction tools, gunshots, street music, and car engines. A GUI is used to present the performance of the model by having the model make predictions on a loaded audio clip. The motivation is to find the best approach to solving this problem and implement and train the algorithm on a dataset of the aforementioned sounds. The project is centred around the use of a Convolutional Neural Network to classify audio samples to a high level of accuracy. Because of this, each audio sample is first transformed into a type of spectrogram before being processed into the appropriate format for training. The model is trained using Google Colab and is exported in an appropriate format to be used in the GUI application.

Audio classification is one of the most prominent fields of machine learning and deep learning. Also known as sound recognition or classification, this process is used by many modern AI and ML technologies and applications: in virtual assistants, such as Amazon Alexa and Siri, text-to-speech software and applications, and automatic speech recognition. There are different subtypes of audio classification that deal with different types of sounds and audio, such as music classification, which may include genres or individual instruments, natural language processing, which deals with human or artificial speech and language, and environmental sound classification, which deals with natural, urban, or otherwise environmental sounds such as animals, vehicles, people, and industrial sounds. The majority of these types use some form of artificial neural network, such as Convolutional Neural Networks, Recurrent Neural Networks, or general multi-layer perceptrons.

The neural network used in this dissertation is a Convolutional Neural Network (CNN), commonly applied to solving problems that use data with a grid-like topology, such as images or videos. With images, especially large ones, a fully connected neural network would require significantly more computational power to perform tasks due to the use of matrix multiplication in forwardpropagation and backpropagation. CNNs are distinguished from other networks by the use of convolution operations and sparse interactions rather than general matrix multiplication and full connection, ensuring that only a few neurons in a hidden layer are connected to the output and input without an impact on performance (Goodfellow et al., 2017) (Chapter 9.0).

1.2 Problem statement

The problem described in this dissertation is known as Environmental Sound Classification (ESC). Environmental Sound Classification is a supervised learning task that involves using a neural network or other form of machine learning to classify different environmental and urban sounds, such as those described in the background. It is similar to another problem called Sound Event Detection/Recognition (SED/SER), recognising and classifying sounds that are a part of a longer recording, which may overlap with others. With ESC, however, it does not aim to detect the sound's temporal start and end in a recording; instead, it only works with the isolated sounds themselves.

The three most popular datasets for training models for this task include the UrbanSound8K, ESC-10, and ESC-50 datasets, which the majority of ESC papers use. Each of the sounds in these datasets are short in duration (between 1-5 seconds) and are organised into either 10 (UrbanSound8K and ESC-10) or 50 (ESC-50) classes. All the files in the datasets use the .wav format, as it contains metadata relevant for data preprocessing and analysis, such as the sample rate, number of channels, and the byte rate.

1.3 Aims and objectives

1.3.1 Primary

The primary aim is to implement a method of classifiying urban sounds into categories, such as dogs barking, construction tools, and car horns. The selected model should be able to classify these sounds to a high level of accuracy, and be optimised in terms the number of parameters. The model should be serialisable, so that it will work with the GUI, and should be able to generalise to new, unseen examples, done via use of a test set and some other sounds that are not in the dataset.

Objectives: The initial objective is to research existing solutions to find the best possible method of performing Environmental Sound Classification, which is completed as part of the literature review. Papers that use different model types and architectures will be compared against each other in terms of performance from the results sections. After finding the best performing models, the next objective is to implement them in Jupyter Notebooks and perform several experiments. These experiments should aim to reduce their overall size while yielding similar, if not better, performance compared to the original model.

1.3.2 Secondary

The secondary aim is to develop a Graphical User Interface capable of showing the performance of the model, allowing a user to test the model against a new, unseen, and unlabelled audio file they upload. The GUI should include features and visualisations related to the model, the audio file, and the results and predictions of the model against the audio file. It should present this information in a clear but simple, user-friendly format, such as a graph or text.

Objectives: The initial objective is to plan the functionality of the GUI, using a flowchart to describe the different features of the application and how they are linked. Once this is complete, the next step is to find a programming language and library capable of creating the GUI; ideally, it should be the same language used for implementing the model. Next, the features of the GUI are planned out and implemented using the programming language and library, then implement error handling to ensure that the program will continue running through any issues or problems that could occur. The final objective is to test the GUI to

ensure that it meets the secondary aim, all of the features are working as intended, and that error handling features are working.

1.4 Solution approach

The first step of the solution approach involves researching existing solutions to environmental sound classification, understanding the inner workings of each of them, and comparing their performance against each other, according to their results sections. While some may be more relevant to the problem of environmental sound classification, others use audio datasets that are composed of other categories, including music genre, instruments, and spoken digits. These are still included as examples of audio classification in general, and how machine learning techniques can solve the problems mentioned in the papers. However, for comparisons, we only take into account those that use environmental sound datasets.

The comparisons have identified two best-performing relevant implementations: *Environmental Sound Classification using a Regularized Deep Convolutional Neural Network with Data Augmentation*, and *Sound Classification using Deep Learning*. The models from these sources were used for the experiments, with the aim of reducing the size/power required for the models while either maintaining or improving performance. To distinguish the two models, we refer to the former as the **ESC** model, and the latter as the **Soundscape** model. These experiments involved changing the model architecture and training parameters, such as adding or removing layers, changing filter sizes, or training each model for more epochs/differing batch sizes.

Next, the GUI from the secondary aim was designed and implemented. The initial design was created as a flowchart describing how each of the functions work, and includes various decision nodes as error handling. Afterwards, the GUI's sections/frames, buttons, and text/images were laid out in their appropriate positions first, before adding the features and functionality to the program. This was done to prevent having to redesign the layout later, and saved time that should be spent working on the functions.

Finally, the program was tested and run on the different model implementations and some sound files, to assess both the performance of the model on completely unseen data and the GUI itself.

1.5 Summary of contributions and achievements

The results of comparing and testing the model have shown that the best performing modification of the Soundscape model is the 7th experiment with the control datatype (MFCCs), where the filters in each layer remained constant and the batch size was 128. Compared to the original, this model took less than a minute longer to train and yielded a 0.8% decrease in accuracy, but had just 30,910 parameters overall. The best performing variant of the ESC model is experiment 2d, which took a similar amount of time to train but yielded a 2% increase in accuracy and contained 25% less parameters. Compared to the soundscape model, this variant has a marginally lower accuracy and a longer training time, but has 20% less parameters; an acceptable tradeoff in favour of a reduced size.

Additionally, training the model on individual, unseen data has shown that the model can accurately classify the majority of the sounds loaded, where the sounds are isolated samples. However, where background noise is present, the model will fail to classify the sounds accurately; when it does this, it tends to very confidently classify it as the incorrect category. This is usually due to how the model "understands" the data - as seen later, the

dataset sounds have very different waveform patterns for each class and doesn't account for much background noise in the samples.

1.6 Report Organisation

The rest of the report is organised into six other chapters:

Literature Review: This section describes the problem of environmental sound classification, as well as sound classification in general, in more detail. It describes the various papers and articles whose aim is to perform such classification, and the performance of each of them.

Primary Concepts: This section describes the relevant concepts of this project: Convolutional Neural Networks, and Spectrograms (including melspectrograms, MFCCs, and Log-Melspectrograms).

Methodology: This section describes the methodology of both aims and their subsequent objectives. It includes details on the implementations of the best performing models and the experiments performed on both. It then describes the graphical user interface's development, features, and design.

Results and Discussion: This section describes the results of the model experiments, and how they compare to the original models as well as the others described in the literature review. It also showcases the graphical user interface in action, and how it achieves the secondary aim and objectives. Finally, it describes the significance of the findings and any limitations.

Conclusion: This section summarises the investigated problem, aims, and objectives. It describes the findings that have been obtained by applying the methodology, implementations, and experiments. It also includes a section on future work, including critical analysis of the solution and anything possible improvements for similar future projects.

Reflection: The final section describes the learning experience of the project as a whole, focusing on the performance of the individual rather than the implementations/methodologies.

Chapter 2

Literature Review

The Literature Review describes the concept of Environmental Sound Classification (ESC) and discusses the different papers and articles that use different machine learning methods to solve the problem of ESC.

2.1 Audio Classification and Environmental Sound Classification

One of the most popular problems in machine learning and signal processing is audio classification. Most modern AI technologies make use of audio classification in some form, such as virtual assistants, text-to-speech software, and some natural language processing applications. Tasks that involve audio classification include the ability to recognise and classify music by artists and genre via transfer learning (Wang et al., 2019), classifying audio that contains voices using the Katz algorithm (Ali and Talha, 2018), and accent classification (Lu et al., 2020). Most audio classification tasks can be organised into one of three categories: Music Recognition (genre recognition, instrument recognition, and artist classification), Speech Recognition (voice recognition, accent recognition), and Sound Event Recognition/Detection. This dissertation focuses on a subtask within the last category: Environmental Sound Classification

Environmental Sound Classification is a subtype of audio classification that aims to classify different types of environmental and urban sounds; common audio events that tend to be more chaotic and unstructured in nature. This is in contrast to music and speech audio, which tend to have some pattern or structure to them (especially with the former). It is a type of supervised learning, and due to the complex nature of the data, many papers that perform ESC have used deep neural network models to perform the task, often to a high level of accuracy. Such models include Convolutional Neural Networks/CNNs (Piczak, 2015*a*) (Mushtaq and Su, 2020), Recurrent Neural Networks/RNNs (Li et al., 2019), and even hybrids that combine multiple neural networks (Palanisamy et al., 2020). The most popular of these models are CNNs, as ESC datasets contain samples that are of fixed length and deal with spatial data, rather than temporal data, which CNNs generally work well with. On the other hand, models such as RNNs are better suited to sequential data, such as stock price prediction, speech recognition, and some NLP tasks such as machine translation and text prediction.

Most of the papers that deal with ESC or audio classification use some form of data preprocessing was used to convert the audio data into a model-readable format. The universal method is to use some form of spectrogram. Spectrograms are representations of audio signals that displays the changes of the signal's frequencies over time, as opposed to traditional waveforms which shows changes in amplitude over time. Most paper use one type

of spectrogram, such as spectrograms themselves (Writer, 2019), melspectrograms (Dwivedi, 2018), or Mel-Frequency Cepstral Coefficients (MFCC) (Li et al., 2019). Some other papers combine different types of spectrograms and other data formats, mainly for experimentation and performance comparison:

- Mushtaq and Su (2020) compared two models that each used melspectrograms, MFCCs, and Log-Melspectrograms. The authors also used some data augmentation on the data. Of the three datatypes, log-melspectrograms achieved the highest accuracy with the model that does not use max-pooling, scoring 94.14% on the UrbanSound8K dataset, 81.25% on ESC-10, and 57% on ESC-50.

- Zhang et al. (2018) used log-melspectrograms and gammatone spectrograms, along with mixup, a method of generating training data by combining two randomly selected examples from the original dataset. The two datatypes performance similarly for all variants of the model (the model itself, the model with mixup, and the model with mixup and augmented data), with a mean difference of 0.46% for ESC-10, 1.66% for ESC-50, and 2.1% for UrbanSound8K.

- Palanisamy et al. (2020) used log-spectrograms, log-melspectrograms, MFCCs, and gammatone spectrograms on a pre-trained ImageNet CNN. They discovered that using melspectrograms with different window sizes and hop lengths in each channel gave the best performance.

For the models, many papers make use of neural networks. Most of the architectures are self-constructed by the authors, while some, such as Palanisamy et al. (2020) and Hartquist (2018), use pre-trained ResNet models (Inception, ResNet, and DenseNet for the former, and ResNet 18 for the latter). The concept of retraining an existing neural network to perform another task is known as transfer learning, formally defined as "the situation where what has been learned in one setting is exploited to improve generalisation in another setting" (Goodfellow et al., 2017). ResNet18, in particular, is an 18 layer deep convolutional neural network that has been pretrained on the ImageNet database, an image classification dataset developed by Princeton and Stanford Universities. The most popular subset of ImageNet, the ILSVRC set, has 1000 classes comprising 1,281,167 training images, 100,000 test images, and 50,000 validation images. Although Hartquist used the NSynth dataset containing annotated data on different musical notes, rather than environmental sounds, the format is largely the same; audio data converted into a form of spectrogram producing 2-dimensional data, sometimes with a 3rd dimension for different channels. The images take the same format, also with a possible 3rd dimension for separate RGB channels, thus ResNet18 is capable of being trained on the converted audio data.

Most other papers, however, build their own neural network architecture. One example is Piczak (2015*a*). The paper used three datasets containing different categories of short sounds that are typically heard in an urban environment: ESC-50 (Piczak, 2015*b*), ESC-10, and UrbanSound8K (Salamon et al., 2014). ESC-50 consists of 2000 recordings organised into 50 classes, which themselves are loosely arranged into 5 categories: animals, nature soundscapes and water, human sounds, interior/domestic sounds, and exterior/urban noises. ESC-10 is a less-complex dataset that only contains 400 recordings arranged into 10 classes: dog bark, rain, sea, baby crying, clock ticking, sneezing, helicopters, chainsaw, rooster, and fire crackling. UrbanSound8K consists of 8732 recordings organised into 10 classes, and unlike ESC-50 or ESC-10, the sounds are strictly environmental and urban. The data is converted into log-scaled melspectrograms using the Librosa library, and the model used is a

deep 2D convolutional neural network with five layers, two of which were convolutional layers, using ReLU activation, mini-batch stochastic gradient descent, L2 regularisation, and dropout between layers.

The final predictions were made using either a majority-voting scheme or the probabilities of each class. The results for each dataset were split into five variants, depending on the segment length and voting method. The resulting performance showed that the CNN achieved a peak accuracy of around 73.5% on the UrbanSound8K dataset, using long segments and probability voting, 80% for the ESC-10 dataset using long segments with probability voting, and 65% on the ESC-50 dataset using short segments with probability voting. The probability-based predictions proved to be universally favourable over the majority voting method, and longer segments worked better for the UrbanSound8K dataset. Due to its' popularity and performance, the model has been used as a baseline for performance in many other papers, including Salamon and Bello (2016) and Zhang et al. (2018); Piczak mentions that the baseline results should not be treated as fine-tuned SOTA techniques, and that the real potential of CNNs for ESC requires further evaluation. According to ResearchGate, Piczak (2015*a*)'s model and results have over 400 citations in other papers.

Some papers perform data augmentation on the dataset to deal with the issue of data scarcity: insufficient amount of labelled data that can be used to train and/or test a model's performance. One notable example is Salamon and Bello (2016). In this paper, the authors only use the UrbanSound8K dataset and used a log-melspectrogram format, but with the addition of data augmentation techniques to deal with data scarcity within the dataset. These data augmentation measures include:

- **Time Stretching**: changing the speed of the audio sample without altering the pitch
- **Pitch Shifting**: two sets of pitch shifting values that alter the pitch of the audio sample, either raising or lowering it. A second set was added as initial experiments proved that pitch shifting is beneficial.
- **Dynamic Range Compression**: changes the dynamic range of an audio sample. DRC aims to compress, or reduce, an audio signal's range; either reducing the volume/amplitude of particularly loud sounds, or amplifying quieter ones.
- **Background Noise**: mixing one audio sample with another that contains background noises. These noises include street workers, traffic, people, and park noises.

The model used is a five layer convolutional neural network, with three convolutional layers, that uses cross-entropy loss optimised with mini-batch stochastic gradient descent to evaluate performance, known as SB-CNN. In addition to training with augmented data, the model also trained on the standard dataset for performance comparison. Each of the models were evaluated using 10-fold cross validation.

The performance of the SB-CNN (with and without data augmentation) was compared to the model in Piczak (2015*a*), known as the PiczakCNN, and a spherical k-means model that used dictionary learning (also with and without augmentation), known as SKM. The results showed that the models lacking data augmentation performed similarly to the PiczakCNN, with SKM performing slightly better than SB-CNN, with a mean accuracy of 0.74 and 0.73 respectively. Using data augmentation, both models performed better than PiczakCNN, and SB-CNN outperformed SKM on average, with a mean accuracy of 0.79 for the SB-CNN. A confusion matrix was also provided with the results of the SB-CNN with and without augmentation. Without augmentation the matrix showed that the most confused classes were

air conditioner with car engine, and jackhammers with drilling. With augmentation, a matrix showing the difference between the standard and augmented confusion matrices was used instead. These results show that data augmentation can be beneficial for training models on datasets that do not contain sufficient training examples, and can provide an extra challenge to models to evaluate their performance more deeply.

While most of the papers only use one type of neural network (or more than one, but each model is a certain type), some use hybrids or combinations of models. Two examples are Dwivedi (2018) and Palanisamy et al. (2020).

Dwivedi (2018) used two different models for music genre recognition on the Free Music Archive dataset (Defferrard et al., 2017), which contains short samples of songs in eight different genres:

- **Convolutional-Recurrent**: comprised of three 1D convolutional layers, with ReLU, batch normalisation, and maxpooling, which performs the convolution operation only along the time dimension of each sample. This is then fed into a 96-unit LSTM layer, which is used to find the short and long term structures of each song, and finally two dense layers. The final dense layer has 8 units with softmax activation to get the probabilities for each class. The resulting accuracy on the test set is 53%.
- **Parallel CNN-RNN**: aims to solve the problem of the recurrent layer in the Convolutional-Recurrent model relying on the temporal output of the convolutional layer. The input is passed to both CNN and bidirectional RNN sections in parallel, each of which has five layers, concatenates the outputs, and sends them to the same dense and softmax layers as the first model. The convolutional layers use 2D convolution and max pooling. The RNN section starts with a max-pooling layer to reduce the size of the input before sending it to a bi-directional Gated Recurrent Unit. This model achieved 51% on the test set.

Overall, the results do not show a very high accuracy compared to other literature. Dwivedi (2018) has stated that this may be due to two reasons. The first is that the sample size is too small for building a deep learning model from scratch, which resulted in overfitting for both of the models. The second reason is that the Free Music Archive dataset is complex in natur and some of the classes can easily be confused with one another.

On the other hand, Palanisamy et al. (2020) compared three ImageNet-based pre-trained models that were trained via transfer learning on the UrbanSound8K and ESC-50 datasets, as well as the GTZAN dataset, intended for music genre recognition. The two models The Inception and ResNet models, while unlike most basic convolutional neural network architectures, are not a hybrid of two different types of neural networks.

The hybrid model used in the paper is the DenseNet model. Developed by Huang et al. (2018), the model combines the idea of densely-connected neural networks with use of convolution layers. Where the convolution layers in a CNN are only connected to the previous layer in the network, and tend to have a set of convolution layers that eventually get connected to a dense layer that is then connected to the output, DenseNet proposes the use of "Dense Blocks" in between convolution and pooling layers. These dense blocks are made up of many connected layers, and each of these blocks obtain the feature maps from all of the previous layers as its input, being concatenated into a single tensor. Between each layer in a dense block, there is a "composite function" made up of three operations: batch normalisation, ReLU activation, and a 3x3 convolution that transforms the output of the previous layer before feeding it into the next layer. Between each dense block, there are two components: a 1x1 convolutional layer and an average pooling layer with size 2x2. These are known as

transition layers and serve to change the feature map size of the output of a block to be fed into the next block. The hyperparameter k, known as the growth rate, is a multiplier on the number of feature maps produced by each layer's composite function; essentially the number of additional channels for each of the layers.

The model was trained on four datasets. The first two are the CIFAR-10 and CIFAR-100 datasets, consisting of coloured 32x32 pixel images to be classified into 10 and 100 classes respectively, including, but not limited to: cars, animals, birds, and people. The second is the Street View House Numbers/SVHN dataset, also consisting of coloured 32x32 images but featuring digit images of street house numbers . The fourth dataset is the ImageNet ILSVRC 2012 dataset, which contains 1.25 million images for training and validation split into 1000 classes. Both CIFAR datasets also had data augmentation performed on them, mirroring and shifting, and the data was normalised using the channel means and standard deviations. For training and evaluation, there were six other variations of the model that used different k values and made use of compression to reduce the number of feature maps at the transition layers, and each were compared to other state-of-the-art models. These other models included ResNet (which also had variations), FractalNet, All-CNN, and Deeply Supervised Net. The results show that each of the six models managed to outperform all of the other models, some of which had substantially more parameters than the best-performing DenseNet model. FractalNet, in particular, contained 38.6m parameters, and on the CIFAR-10 dataset was beaten by a DenseNet model that contained only 0.8m parameters (the best performing variation of DenseNet, however, contained just over half of the number of parameters). This proves that it is possible to develop deep learning models with less parameters that are capable of outperforming other models that have significantly more parameters, and relates to the primary aim of this project, as specified in Chapter 1.

The authors have argued that using pre-trained models minimises the time and effort required for feature engineering and model design, and that the same methods based around transfer learning for different image tasks can apply to transfer learning from image data to spectrograms. They experimented with different data types, and found that the log-melspectrogram format was the best type for the task. As with the original DenseNet in Huang et al. (2018), the model used in the paper managed to outperform both the ResNet and Inception models on most of the datasets, including the UrbanSound8K dataset. The DenseNet model using pretrained weights performed better that other models on the same dataset, such as Zhang et al. (2018) (79%), Seker and Inik (2020) (82%), and even Piczak (2015*a*) (73%), proving that the use of transfer learning can be advantageous in harnessing the power of SOTA models trained on one dataset on other that has a similar structure, and that more innovative neural networks that combine multiple types can yield high levels of accuracy and potentially beat non-hybrid models.

To document and organise the findings of each paper, two methods were used. The first is a "technical powerpoint" that summarises the papers in terms of the data used, the model(s) used, and the overall performance, as well as general progress throughout the project. The second is a tree diagram that sections each paper based on the model used, the dataset, and the preprocessing format in a vertical tree format, which includes the final accuracies of each paper, with sometimes multiple accuracies being displayed if a paper has multiple models. The tree diagram made it easier to organise the papers into different categories for comparison and allows for quick referencing to the performance of each paper.

Due to the size of the diagram, it has been split into different sections. Figure 2.1 displays the higher branches of the diagram, which showing the variety in models used as well as the datasets, but do not show the papers themselves. Figure 2.2 shows the section of the tree

diagram dedicated to Convolutional Neural Networks that use the UrbanSound8K dataset. The two nodes that are highlighted in green are the two models that were eventually chosen for implementation later. Finally, Figure 2.3 shows the rest of the papers in the diagram, which use UrbanSound8K as well as other datasets that may not be ESC datasets, but were chosen to highlight the use of machine learning and neural networks for audio classification in general:

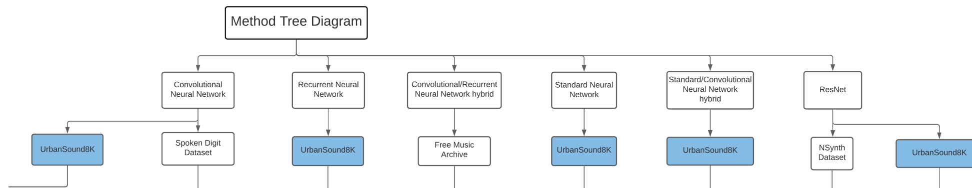

Figure 2.1: Top of Tree Diagram, showing the models and datasets used

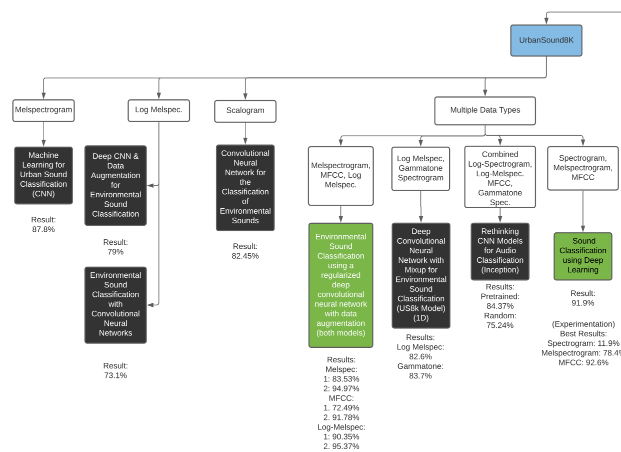

Figure 2.2: Papers that use CNNs and the UrbanSound8K Dataset

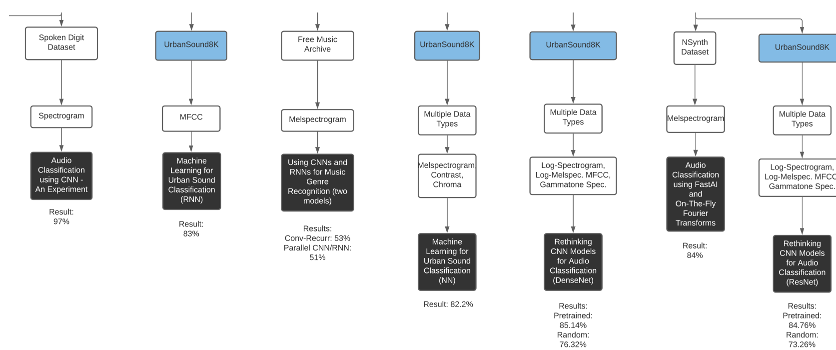

Figure 2.3: The other papers

2.2 Critique of the review

The existing literature has assessed the effectiveness of using deep neural networks to perform audio classification and ESC. The above analysis and comparison of the literature have shown

that these models are capable of performing this task to a high level of accuracy, and tends to generalise well to new, unseen data. Standalone Convolutional Neural Networks seem to be the most popular type of model compared to other types, such as RNNs, hybrid models, and standard neural networks. Many authors, including Piczak (2015*a*), Li et al. (2019), and Mushtaq and Su (2020), have mentioned that the ability for CNNs to classify natural images means that it can accurately classify spectrogram representations of audio, often to a similar level of accuracy, with Palanisamy et al. (2020) also proving that a deep CNN trained on ImageNet data is able to accurately classify audio samples from the UrbanSound8K dataset due to the similarity of the datatypes.

However, most papers do not take into account the number of parameters in the model. Some papers mention parameters, but only a few subsequently aim to optimise them. Huang et al. (2018), who developed the DenseNet model used in Palanisamy et al. (2020), mentioned that a variation of the model containing 0.8m parameters managed to outperform a SOTA model that contained 38.6m parameters, and that the overall best performing variation of DenseNet contained over half the number of parameters of the aforementioned SOTA model. Mushtaq and Su (2020) did mention the number of parameters for each model in the performance evaluation table. The majority of the results, however, actually highlight that the number of parameters increases with the level of accuracy: the models without pooling using Log-Melspectrograms did achieve the highest accuracy in their dataset, but had the highest number of parameters. This issue ties in to the primary aim of this project, which is to implement a model to classify the UrbanSound8K dataset, optimised for the number of parameters while yielding a similar or higher level of accuracy compared to the original model.

2.3 Summary

Existing literature have proven that using deep neural networks, such as Convolutional Neural Networks, Recurrent Neural Networks, hybrid models, and pre-trained models, to classify audio datasets, such as ESC-50, ESC-10, and UrbanSound8K, can yield high levels of accuracy. Piczak (2015*a*) trained a CNN on a low-level representation of the aforementioned datasets, achieving an accuracy of 65%, %, and 73.6% respectively. This paper was used as a baseline for performance evaluation in Salamon and Bello (2016), which used data augmentation in addition to a CNN and achieved a higher mean accuracy than Piczak's, Zhang et al. (2018), who used mixup to generate new training data in addition to augmentation and achieved high accuracies across all the datasets, and Seker and Inik (2020), who compared their model with many others, including Piczak's. Some papers use a pre-existing/pre-trained model on the datasets, via transfer learning, and managed to achieve state-of-the-art levels of accuracies. Palanisamy et al. (2020) used three pre-trained models: Inception, ResNet, and DenseNet. They experimented with each models' weights (whether pretrained or random), and compared the performance of the fine-tuned pre-trained models with those trained from scratch, and have managed to achieve an accuracy of over 90% for the ESC-50 and GTZAN datasets, and 87% for the UrbanSound8K dataset.

However, most of the literature did not aim to optimise the model in terms of the total number of parameters, which could result in the models using more parameters than is necessary. This project aims to overcome this issue by performing experiments on the best-performing and relevant models to try to minimise the number of parameters while either maintaining or outperforming the original.

Chapter 3

Concepts: CNNs and Spectrograms

3.1 Convolutional Neural Networks

As mentioned in Chapter 1, this project focuses on the application of convolutional neural networks to perform audio classification to a high level of accuracy.

Convolutional Neural Networks, or CNNs for short, are a type of neural network that uses convolution, rather than general matrix multiplication, in its hidden layers. It is best applied to any data that has a known, grid-like topology (Goodfellow et al., 2017), which includes non-sequential time-series data, or image data. One of the first known applications of a convolutional neural network is in document recognition by Lecun et al. (1998), where the authors demonstrated that CNNs managed to outperform other techniques in handwritten digit recognition. Since then, CNNs have been applied to many fields, including question answering systems for natural language processing (Amin and Nadeem, 2018), medical image classification for disease diagnosis (Yadav and Jadhav, 2019), and environmental sound classification.

Convolution layers in a CNN have the following sub-layers that perform the convolution operation:

- **Kernel**: also known as the filter. This is the element responsible for carrying out the convolution operation on the input. It is a small square matrix that is applied to input data multiple times, generating a smaller matrix containing scalar values. Convolution layers contain multiple filters that each learn a series of features, with each filter containing different values that perform a different type of convolution. The size of the output depends on two parameters: the kernel size and the stride length, which is how far the kernel is "slid" across the input. A small kernel size generally results in a larger ouput, while a higher stride length results in a smaller output, since the kernel is being slid far across the image.

- **Activation**: similar to a standard neural network, the activation function determines the output of the layer, deciding which units will fire and which will not. There are different activation functions, but the most popular used is Rectified Linear Unit, or ReLU, which fires if the value in a specific unit is greater than 0:

$$f(x) = max(0, x)$$

- **Pooling**: after activation, this operation reduces the size of the output by replacing the output value at a certain location with a "summary" of the nearby values, either by returning the maximum value from the location (max-pooling) or the average value

(average-pooling). It is a type of downsampling that makes the output invariant to any small translations of the input, which can be useful when determining the presence of a certain feature, such as eyes on a face or ears on a cat. In most cases, such as for typical cat/dog classifiers or facial recognition, this tends to be more important rather than the actual location of the feature. By reducing the size of the output, it reduces the amount of computational power required to process the data, and may also be seen as a form of dimensionality reduction.

A typical image classification CNN may have two to four of these layers before feeding the final pooled output to a dense layer, which serves as a method of learning non-linear combinations of the features outputted by the convolution layers. This dense layer also flattens the input to a one-dimensional vector before feeding it to another dense layer that contains one unit for each class in the dataset. Some convolutional neural networks also have dropout layers, which serve to deal with overfitting by randomly dropping some units and their connections from the network during training (Srivastava et al., 2014). Figure 3.1 shows a typical CNN architecture with two convolutional layers.

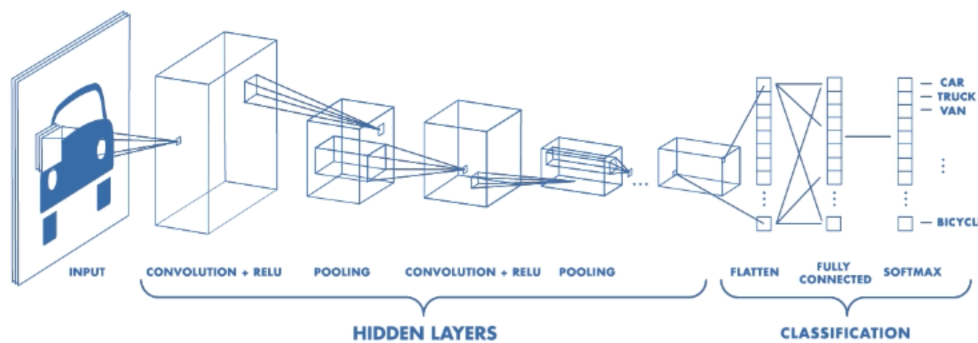

Figure 3.1: A typical convolutional neural network architecture

3.2 Spectrograms

For the implementations in chapter 4, the audio data is preprocessed into a type of spectrogram, since the network cannot handle raw audio data.

Spectrograms are representations of a signal that record the strength of certain frequencies that make up that signal over time, as opposed to a waveform that records the signal amplitude. It is often used in audio and signal processing to display the frequencies produced by animals, humans, and machinery. A spectrogram is a two-dimensional graph, with the time running from oldest (left) to newest (right) along the X-axis, and the frequencies from the lowest (bottom) to highest (top) along the Y-axis. There is also a third dimension that displays the presence and intensity of a signal at a specific time using colour.

Spectrograms are generated using a series of fast fourier transforms (FFT) applied to windows of a signal. The size depends on the number of windows, and the FFTs represent the variance of the frequencies as they occur in that window. The structure of a spectrogram is similar to that of image data, and is thus great for convolutional neural networks.

There are different types of spectrograms that represent the frequencies in other ways. One popular alternative is a melspectrogram, which overcomes the issue of some sounds appearing

too quiet in a spectrogram. A melspectrogram uses the mel-scale, a scale of pitches that are perceived to be equal in distance from each other and is essentially a logarithmic transformation of the signal's frequencies. Papers that used melspectrograms include Li et al. (2019), Dwivedi (2018), and Hartquist (2018). A popular formula devised by O'Shaughnessy (1987) to convert f hertz, or the frequencies of a spectrogram, into m mels is:

$$m = 2595 \log_{10}(1 + \frac{f}{700})$$

Which can also be written as:

$$m = 1127 \ln(1 + \frac{f}{700})$$

These mels are then plotted in place of the frequencies along the vertical axis.

A popular variant of melspectrograms, known as log-melspectrograms, is another alternative that is used. Rather than being a representation of the power of a frequency, log-melspectrograms represent the frequencies as decibels, which are log-scaled. Log-melspectrograms are used in Palanisamy et al. (2020), Mushtaq and Su (2020), and Salamon and Bello (2016).

Another popular type is Mel-Frequency Cepstral Coefficients, or MFCC for short. MFCCs are a common datatype used for speech recognition tasks, such as speaker verification (O'Shaughnessy, 1987). It represents the power spectrum of a sound taken from the mel scale (hence the use of "mel-frequency"). There are six steps to getting the MFCC from an audio signal (Sahidullah and Saha, 2012):

1. Multiply the signal by a tapered window.
2. Zero pad the signal.
3. Take a logarithm of the powers from each mel frequency.
4. Compute the power spectrum using a Fast Fourier Transform
5. Pass the signal through a triangular filter bank.
6. Calculate the Discrete Cosine Transformation of the signal, and get the resulting amplitudes.

Papers that use MFCCs include Smales (2019), Palanisamy et al. (2020), and Li et al. (2019). Both Li et al. (2019) and Smales (2019) note that MFCCs are the closest representation to how the human auditory system perceives sounds.

Chapter 4

Methodology

This chapter will focus on the methodology of the program, going over both aims and their objectives, as well as the implementations of the best performing models and the experiments that were performed on both.

4.1 Implementations

When choosing the best performing papers to implement, the primary deciding factor is the modifiability of the model used. Because the experiments will involve changing the model architecture and training parameters, as well as possibly the data preprocessing type, the model used in any of the best performing papers need to use a type whose architecture can be modified. Take, for instance, the three models used by Palanisamy et al. (2020): ResNet, DenseNet, and an Inception Model. All three models are pre-trained on the ImageNet database and are specifically used for transfer learning on data with a similar structure, which the authors have proved work very well on the UrbanSound8K dataset. Although these models have SOTA performance, the architectures of these models are not easily modifiable. The DenseNet model, for instance, is a hybrid model combining the idea of dense neural networks with convolutional layers. The different components making up the network include "dense blocks", composite functions between each layer in these blocks, and two additional layers between each block. Modifying the architecture of such a model would involve changing all of the aforementioned components individually for each experiment, which would be significantly more time consuming and computationally expensive than using a simpler, non-hybrid model.

From the papers in the tree diagram from Chapter 2, the two best performing papers with models that could easily be modified were Smales (2019) and Mushtaq and Su (2020). Both models are from the convolutional neural networks section, and both involve training their models on the UrbanSound8K dataset. The latter also trained their model on two additional datasets: the ESC-50 and ESC-10 datasets. Both models were implemented in Python using the Keras module for building the models and training them. For convenience, we will refer to the model in Smales (2019) as the **Soundscape** model, and the model in Mushtaq and Su (2020) as the **ESC** model.

Smales (2019) uses a four layer, two-dimensional convolutional neural network. The pre-processing technique used was mel-frequency cepstral coefficients, with the author mentioning that the datatype would be more similar to how humans process sounds. Each of the layers have an increasing number of filters, but the size of the kernel stays the same across all layers. The model architecture is described as below:

1. Convolutional Layer. Contains 16 filters, each with a size of 2x2, and the input shape

is set as (number of rows x number of columns x number of channels). The layer uses ReLU activation, and is followed by a max pooling layer of size 2 and dropout set to 0.2.

2. Convolutional Layer. Contains 32 fitlers, each with a size of 2x2, and ReLU activation. Followed by the same max pooling and dropout layers as the first layer
3. Convolutional Layer. Contains 64 filters, each with a size of 2x2, and ReLU activation, with the same max pooling and dropout as before.
4. Convolutional Layer. Contains 128 filters, each with a size of 2x2, and ReLU activation. This layer contains the same max pooling and dropout as before, but with the addition of a global average pooling function.
5. Dense Layer. This layer functions as the classification layer, with 10 units pertaining to each class. Softmax activation is used to determine the class. The model was compiled using categorical crossentropy loss, the Adam optimiser, and uses the standard accuracy metric, and contained 44,602 parameters overall. It is trained for 72 epochs and a batch size of 256. The model managed to achieve an accuracy of 98.1% and a test accuracy of 91.9% according to the article, showing that the model is able to generalise well to new examples.

Mushtaq and Su (2020) uses two convolutional neural networks: one that uses max pooling, and one that does not. The paper uses three different datasets, with each set being preprocessed into three formats: melspectrograms, log-melspectrograms, and mel-frequency cepstral coefficients. The authors used both the original datasets as well as the datasets with augmented data. There are four operations performed that deformed the data and resulted in additional examples used for training and testing. These methods were:

1. **Pitch Shifting**: The pitch of each audio signal is either **shifted positively** by a factor of +2, or **shifted negatively** by a factor of -2.
2. **Silence Trimming**: The silent part of an audio clip is trimmed, leaving only the parts containing sound.
3. **Time Stretching**: The length of each audio signal is either **sped up** by a factor of 1.2x, or **slowed down** by a factor of 0.7x.
4. **White Noise**: Slight white noise is added to a sound clip.

The augmentations resulted in a 600% increase in the size of each dataset, as well as a significant increase in the logical size. For the models, each have an increasing number of filters in each convolution layer, but also has a decreasing filter size, as with the Soundscape model. The two models are described below:

1. **CNN With Max Pooling**
   (a) Convolutional Layer. Contains 24 filters, each with a size of 5x5, and L2 regularisation set to 0.001. The layer uses ReLU activation, and is followed by a strided max pooling function of size 3.
   (b) Convolutional Layer. Contains 36 filters, each with a size of 4x4, and L2 regularisation set to 0.001. The layer uses ReLU activation, 'same' padding for MFCCs and 'valid' padding for Melspectrograms and Log-melspectrograms, and is followed by a strided max pooling function of size 2.

(c) Convolutional Layer. Contains 48 filters, each with a size of 3x3, and the same L2 regularisation. As with layer 2, it uses ReLu activation and either 'same' or 'valid' padding for MFCCs and Melspectrograms/Log-melspectrograms respectively. This layer is not followed by a max pooling function.

(d) Dense Layer. Contains 60 hidden units, and uses ReLU activation. This is followed by a dropout function set to 0.5 to prevent overfitting.

(e) Dense Layer, functioning as the classification layer. As with the Soundscape Model, this layer uses softmax activation and has hidden units that pertain to each class (for UrbanSound8K, this is 10).

2. **CNN Without Max Pooling**

   (a) Convolutional Layer. Contains 24 filters, each with a size of 5x5, L2 regularisation set to 0.001, and ReLU activation.

   (b) Convolutional Layer. Contains 36 filters, each with a size of 4x4, L2 regularisation set to 0.001, and ReLU activation. This layer uses 'same' padding for MFCCs and 'valid' padding for Melspectrograms and Log-melspectrograms.

   (c) Convolutional Layer. Contains 48 filters, each with a size of 3x3, and the same L2 regularisation and ReLU as before. Uses either 'same' or 'valid' padding for MFCCs and Melspectrograms/Log-melspectrograms respectively.

   (d) Dense Layer. Contains 60 hidden units, ReLU activation, and is followed by a dropout function set to 0.5.

   (e) Dense Layer, functioning as the classification layer.

Both models were compiled using the Adam optimiser, and trained for 100 epochs with a batch size of 32. For the original UrbanSound8K dataset using Log-Melspectrograms (which achieved the highest accuracy out of all of them), the first model contains a total of 116,002 parameters, while the second contains 3,171,682 (in the actual implementation, however, the second model contains only 33'634 parameters). The first model took 3 minutes and 30 seconds to train, while the second took 5 minutes and 25 seconds. The resulting accuracies for the models were 89.33% for the first model and 94.14% for the second model. With augmented data in addition to the original, the parameter counts remain the same, but the training takes significantly longer to complete; the first model took 20 minutes and 38 seconds to complete, a 489% increase compared to the original, while the second model took 32 minutes and 11 seconds to complete, which is a 494% increase. The resulting accuracies are 90.35% for the first model and 95.37% for the second. The second model outperforms the first model for both results, so the ESC model will use the **CNN without max pooling**.

The results of the ESC paper show that adding augmented data to the ESC-10 and ESC-50 datasets caused a significant increase in accuracy for both models (without having to change the architecture). The second model trained on the ESC-10 dataset, for example, had an increase of 13.69% when trained on augmented data. Augmentation had an even better effect on the ESC-50 dataset, of which the original had significantly worse overall performance than the others - the best performing model/data type combination, which is the second model with log-melspectrograms, performed similarly, if not worse, than the lowest performing models for the other datasets. This model only achieved 57% accuracy compared to the first model using MFCCs for ESC-10 (56.25%) and UrbanSound8K (68.61%). With data augmentation, the accuracy for this model increased by 32.28% and achieved an accuracy of 89.28%.

However, for the UrbanSound8K dataset, data augmentation did not result in any significant increase in accuracy. The best performing model on the original dataset had achieved the best performance across all model, dataset, and data type combinations, at 94.14%, which is already at a SOTA level of accuracy compared to other models, such as Salamon and Bello (2016) and Seker and Inik (2020). With augmentation, this model had only a marginal increase to 95.37%. Taking into account the tri-fold increase in storage size for the dataset (from 6.6GB to 18.9GB), as well as the amount of time and potentially resources required to train the model on additional complex data, this marginal increase is hard to justify and the project would be better off sticking with the original dataset. Additionally, the experiments for both models would be unfair, since the Soundscape model is not trained on augmented data and could take a significant hit in accuracy compared to the ESC model. Therefore, when training both models, only the original dataset will be used to ensure that the experiments are fair with respect to the dataset used.

The UrbanSound8K dataset uses the .WAV format, also known as the Microsoft WAVE format, one of the oldest and widely-used audio formats that is an extension of the RIFF container specification format. .WAV files contain lossless audio and contains metadata from the original RIFF specification. The metadata present in the files proved to be useful in analysing the dataset, as explained in the Python and Libraries section, and are divided into three chunks: the standard RIFF chunk, the 'fmt' or format chunk, and the data chunk. Figure 4.1 shows the organisation of metadata fields and chunks in a .WAV file.

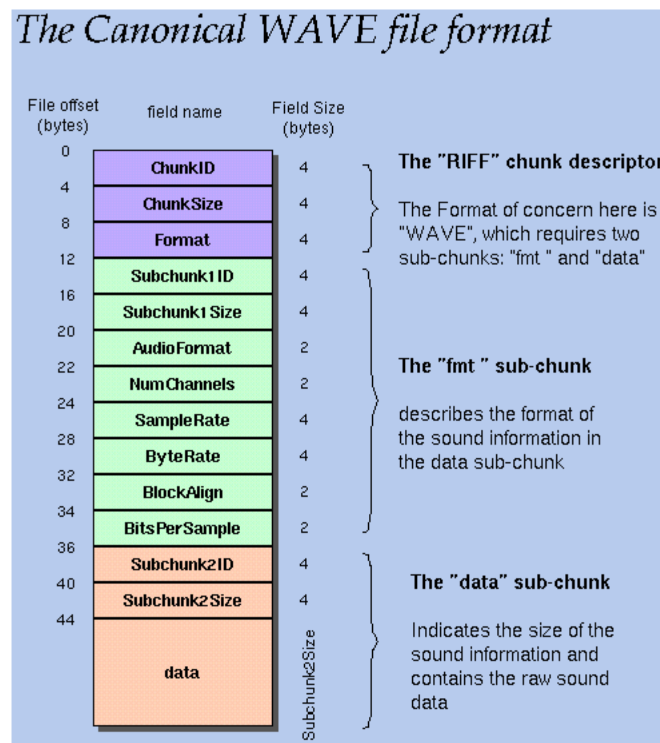

Figure 4.1: .WAV file metadata

Accessing the metadata in each audio file was performed in the Soundscape experiments. Smales (2019) developed wavfilehelper.py, a script that would loop through each audio file, access the metadata of each file, and provide a summary of the number of channels, sample rates, and bit depths of the dataset. This greatly helped with initial exploratory analysis, but was not a step in data preprocessing. The below code snippet shows the script itself.

```python
import struct

class WavFileHelper():
    def read_file_properties(self, filename):
        wave_file = open(filename,"rb")

        riff = wave_file.read(12)
        fmt = wave_file.read(36)

        num_channels_string = fmt[10:12]
        num_channels = struct.unpack('<H', num_channels_string)[0]

        sample_rate_string = fmt[12:16]
        sample_rate = struct.unpack("<I",sample_rate_string)[0]

        bit_depth_string = fmt[22:24]
        bit_depth = struct.unpack("<H",bit_depth_string)[0]

        return (num_channels, sample_rate, bit_depth)
```

Listing 4.1: wavfilehelper.py

```python
import pandas as pd
import os
import librosa
import librosa.display
from wavfilehelper import WavFileHelper

metadata = pd.read_csv('UrbanSound8K/metadata/UrbanSound8K.csv')

audiodata = []
for index, row in metadata.iterrows():
    file_name = os.path.join(os.path.abspath('UrbanSound8K/audio/'),'fold'+str(row["fold"])+'/',str(row["slice_file_name"])) # gets each file from each fold folder in US8K
    data = wavfilehelper.read_file_properties(file_name) # then gets the appropriate data (number of channels, sample rate, and bit depth)
    audiodata.append(data) # stores in array to be used to convert to dataframe

# Convert into a Pandas dataframe, with appropriate columns
audiodf = pd.DataFrame(audiodata, columns=['num_channels','sample_rate','bit_depth'])

print("Done!")
```

Listing 4.2: Application of wavfilehelper.py on the UrbanSound8K dataset

4.2 Experimentation

The experiments consisted of modifying each models' architectures and training parameters. The architecture refers to the individual layers and their parameters, while the training parameters are the number of epochs and the batch size used. All stages of each experiment, including the data visualisation, analysis, preprocessing, model creation, model training, model evaluation, and graphing, were done in two Jupyter Notebooks for the two different models. The experiments are designed with the purpose of reducing the overall number of parameters in each model while aiming to maintain or improve upon the performance of the original. Each model will have four performance metrics recorded:

- **Pre-Train Accuracy**: Before training, the compiled model is evaluated against the dataset and given an initial accuracy. This number seems to be random and has little effect on the overall performance of the model, as seen later.
- **Training Time**: How long the model takes to finish training in minutes and seconds. This is affected by the model's number of parameters (which usually gives away its' complexity), the number of parameters, and the data type used.
- **Post-Training Train Set Accuracy**: The final accuracy of the model on the training set. Also known as the resubstitution error, it is expected that this accuracy will be higher than that of the testset accuracy and won't be used as the final performance metric.
- **Post-Training Test Set Accuracy**: The final accuracy of the model on the test set. Also known as the generalization error, this will be the primary performance metric for the model.

The training parameters modified are the number of training epochs and the batch size. The batch size defines the number of samples that the model trains on before updating its parameters. Smaller batch sizes means the model will update its parameters more frequently. Training that uses a batch size of 1 (meaning the model's parameters updates after each sample) is known as Stochastic Gradient Descent, while training with a batch size equal to the size of the training set is known as Batch Gradient Descent. Both of the models use a method known as mini-batch gradient descent, where the batch size is between 1 and the training set size, striking a balance between stochastic and batch gradient descent. Such batch sizes can have a regularising effect, and are more computationally efficient than using a size of 1, as the parameters do not have to be updated as frequently. Additionally, the amount of memory required for batch gradient descent scales with the size of the training set, which can be a limiting factor in many hardware setups (Goodfellow et al., 2017).

The number of epochs refer to the number of times that the model will train on the dataset. This has a direct effect on the time taken to train a model, but won't have as much of an effect on training as the batch size when compared to the original number of epochs; increasing the number of epochs but keeping the batch size the same won't guarantee a significant boost in performance.

For Soundscape, the experiments will additionally change the format of the data to see if it has any positive impact on performance. The models in the table below will be performed on these other formats and remain unchanged.

Below are two tables containing both sets of models. They include the model's ID #, the name, the architecture, the training parameters, and the number of model parameters.

4.2.1 Soundscape Models

Table 4.1: Soundscape Models

Data Types: MFCC / Spectrograms / Melspectrograms				
Model #	Model Description	Model Architecture	Training Parameters	# Parameters
1	Control Model	Control Model, specified in Implementations.	Batch Size: 256. Epochs: 72	44,602
2	Halved Filters	Control Model. Filters for Convolution Layers are: (8, 16, 32, 64).	Batch Size: 256. Epochs: 72	11,554
3	Four Layer	Control Model. Remove Last Convolutional Layer.	Batch Size: 256. Epochs: 72	42,602
4	Control, Less Filters	Control Model. All filters are 50: Average number of filters in control, minus 10.	Batch Size: 256. Epochs: 72	30,910
5	4, Increase Epochs	Model # 4.	Batch Size: 256. Epochs: 108.	30,910
6	4, Increase Batch Size	Model # 4.	Batch Size: 512. Epochs: 72.	30,910
7	4, Decrease Batch Size	Model # 4.	Batch Size: 128. Epochs: 72.	30,910

4.2.2 ESC Models

Table 4.2: ESC Models

Data Types: Log-Melspectrograms				
Model #	Model Description	Model Architecture	Training Parameters	# Parameters
1a	Control Model	ESC Model 2, specified in Implementations.	Batch Size: 32. Epochs: 100	33,634
1b	Control with More Epochs	Control Model.	Batch Size: 32. Epochs: 175	33,634
1c	Control with Higher Batch Size	Control Model.	Batch Size: 128. Epochs: 100	33,634
1d	Control with Lower Batch Size	Control Model.	Batch Size: 8. Epochs: 100	33,634
2a	Remove Layer, Change Filter	Control Model with no 3rd layer. Change both layers' filter counts to 48 each. Change the kernel size to 4x4 and 3x3 for both layers.	Batch Size: 32. Epochs: 100	25,150
2b	2 with More Epochs	Model # 2	Batch Size: 32. Epochs: 175	25,150
2c	2 with Higher Batch Size	Model # 2	Batch Size: 128. Epochs: 100	25,150
2d	2 with Lower Batch Size	Model # 2	Batch Size: 8. Epochs: 100	25,150
3a	Reduce Kernel Size, Increase Filters	Control Model, with all layers having 48 filters. Kernel sizes changed to 3x3 for the first layer and 2x2 for the second and third.	Batch Size: 32. Epochs: 100	22,558
3b	3 with More Epochs	Model # 3.	Batch Size: 32. Epochs: 175	22,558
3c	3 with Higher Batch Size	Model # 3.	Batch Size: 128. Epochs: 100	22,558
3d	3 with Lower Batch Size	Model # 3.	Batch Size: 8. Epochs: 100.	22,558
4a	Remove Dense Layer and Dropout	Control Model. Remove Dense Layer and Dropout across model.	Batch Size: 32. Epochs: 100.	30,574
4b	4 with More Epochs	Model # 4.	Batch Size: 32. Epochs: 175.	30,574
4c	4 with Higher Batch Size	Model # 4.	Batch Size: 128. Epochs: 100.	30,574
4d	4 with Lower Batch Size	Model # 4.	Batch Size: 8. Epochs: 100.	30,574

Below is a flowchart illustrating the process of training and evaluating a model on the UrbanSound8K dataset.

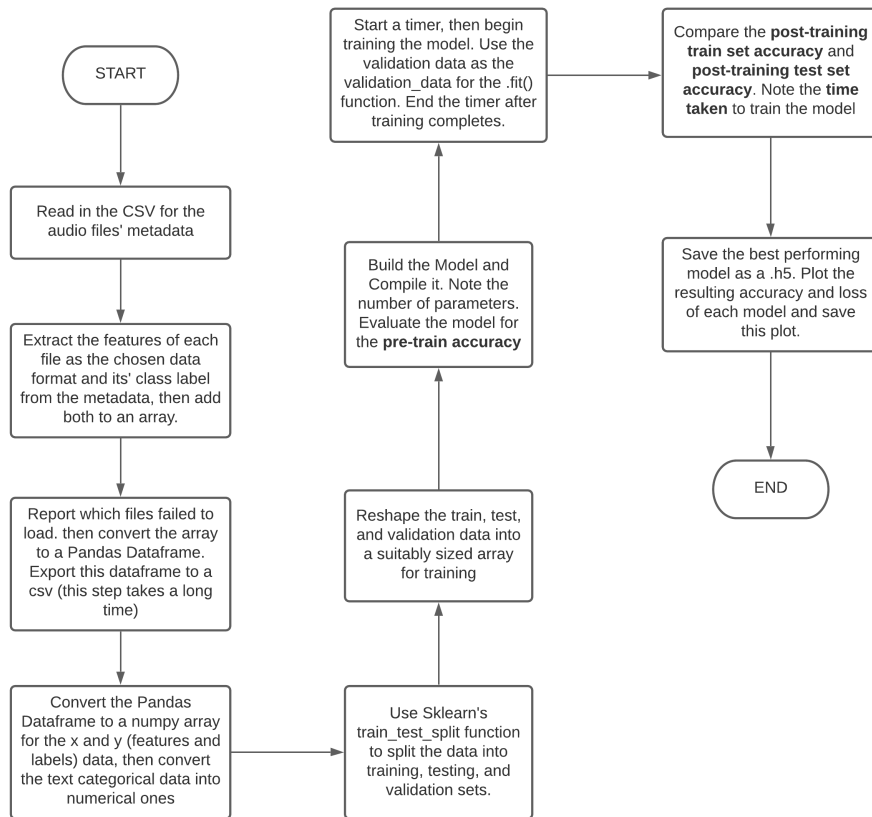

Figure 4.2: Experiment Flowchart

4.3 Implementation and GUI Development: Libraries Used

The implementations and the graphical user interface was developed using the Python programming language. Python was chosen for the implementations as it is one of the most popular languages for machine learning and deep learning, and has many libraries that allow developers to easily create, train, and evaluate models as well as preprocess data. Many of the papers mentioned in Chapter 2 were implemented using Python; with the method varying depending on the model type and architecture. Some papers, such as Hartquist (2018) and Palanisamy et al. (2020), chose to use a pre-trained model and apply transfer learning. In this case, the model can be loaded, trained, and evaluated in a few lines of code, and are typically limited in the modifications to the model architecture and training parameters that can be made. The majority of papers, however, use a ground-up approach to building models, with individual layers being added and configured. This approach allows developers to modify their models to a greater extent than using pre-trained models, modifying the model's architecture, parameters, training parameters, optimisers, regularisers, and many other components.

There are three main libraries that were used in the development and experimentation of each of the models, as well as the development of the GUI.

4.3.1 Librosa

Librosa is an audio and music processing library in Python, initially developed for music information retrieval by McFee et al. (2015). The library has functions that allow developers to load, process, and display audio files in applications through its' various APIs and modules. For this project, Librosa is used to load the UrbanSound8K dataset and convert it into the appropriate formats - melspectrograms, MFCCs, and log-melspectrograms. Some of the papers in Chapter 2, such as Li et al. (2019), Piczak (2015*a*), and Palanisamy et al. (2020), used Librosa to convert the dataset into the appropriate format, as well as for data augmentation tasks.

One important aspect of Librosa is the load feature. The UrbanSound8K dataset contains audio files that have different numbers of channels, sample rates, and bit depths. In the initial exploratory analysis, the wavfilehelper file was used to show the diversity of the dataset. The discoveries made were:

- The majority of the audio files were stereo files, meaning they had two channels, with only a few being mono files.
- There are a wide variety of sample rates for each audio file. The majority of the files had a sample rate of 44.1kHz, which is the standard rate for most audio files, followed by 48kHz. A few of the other files had rates varying from 8kHz to 192kHz.
- There is also a range of bit depths in the data, between 4-32 bits. The most popular is 16-bits, followed by 24 bits. A minority of files were 4, 8, and 32 bits.

The core load function in Librosa resamples an audio file to 22.05kHz, converts it to mono, and normalises the bit depth values to between -1 and 1. This is an important step in data preprocessing, as it ensures that the aforementioned parameters do not have any effect on training.

Another important aspect is the feature extraction module. Two functions are used in preprocessing to get the required features from the dataset: librosa.feature.melspectrogram, which converts loaded audio into a melspectrogram format, and librosa.feature.mfcc, which converts audio to MFCCs. Both are used in the experiments performed on the Soundscape

model. For the ESC model, after the melspectrogram function, the function librosa.core.power_to_db is run to convert the melspectrogram to a log-melspectrogram by transforming the power/amplitudes of the melspectrogram to decibel units.

After preprocessing, the MFCC audio files have a shape of (40, 174, 1). The first value represents the 40 MFCCs taken from the audio, the second represents the 174 frames in the MFCC (which also takes any padding for shorter samples into account), and the final value represents the mono channel. Spectrograms have a shape of (129, 394, 1); there are 129 windows, 394 frames which correspond to the segment times, and only one channel. Melspectrograms have a similar shape to MFCCs, (128, 174, 1), with 128 mels from the audio file. MFCCs have a lower number of rows and a smaller overall size (6960) than melspectrograms (22272) and spectrograms (50826).

In addition to the feature extraction module from Librosa, the spectrogram function from SciPy's signal module was used to convert the audio files to a standard spectrogram format for use in Soundscape's experiments. Unlike Librosa's mfcc and melspectrogram functions, however, the spectrogram function returns three variables: an array of frequencies, an array of times, and a spectrogram of the data, the last of which was used for preprocessing.

4.3.2 Keras

Keras is a high-level API for designing neural networks, acting as an interface to the lower-level TensorFlow library. Keras was used to build, train, and evaluate both sets of models, using the layers, models, optimizers, regularizers, and utils modules. Keras was chosen in the original papers containing the Soundscape and ESC models, and the authors have noted it's ease of use and modifications to the models proved to be easy to perform using this library.

Models are instantiated using the Sequential() function, indicating that it is a stack of layers to be compiled into an actual Model class. Afterwards, individual layers can be added to the model using add, which includes operation layers (such as convolutional, dense, or LSTM layers), sampling layers (such as max/average pooling and upsampling), activation layers (ReLU, LeakyReLU, etc.), and others (dropout, tensor operations, and reshaping). Compiling the model converts the above into the model class, and allows parameters such as the optimiser, loss function, and evaluation metrics to be set. After compilation, the summary function provides an overview of the model's layers, output shape, and number of parameters (both total and per layer).

The fit function trains the model on training and, optionally, validation data. The validation data is used to evaluate the loss metric at the end of an epoch, and can also be split from the training data provided. This function is where the training parameters, such as the number of epochs and batch size, are set. Assigning this function to a variable allows for plotting the accuracy and/or loss over time. This feature was used for plotting the accuracy and loss of each of the models in the experiments.

Finally, the evaluate function evaluates the trained model on the test set (or on the training set) to get the training and testing accuracies.

4.3.3 Tkinter

Tkinter is a Python library that is used for GUI design. It is the standard API to the open-source, cross platform windowing toolkit Tk, and is the most popular method of developing GUIs using Python (alongside PyQt). Tkinter was used to design the GUI for the secondary aim, and the different features for the secondary objectives, such as buttons, graphs, and textual descriptions. There is also support for embedding plots and charts produced in Matplotlib

in a Tkinter window, using the Matplotlib FigureCanvasTkAgg function to create the plot and the get_tk_widget function to position the plot, either in a pack-based or grid-based window.

The two sections of the GUI were created using LabelFrame, a container widget that can hold different elements such as buttons, text, and images. For organising elements, the grid method was used, as it allows for individual elements to be positioned in specific places according to a grid-like format. Tkinter also has the Menu object which enables the creation of a navigation bar at the top of the window, and this was used for some of the functions in the program such as loading a model and viewing information about the program.

In addition to Tkinter, the winsound library is used to play and stop loaded audio in the program. This was achieved using the PlaySound function.

4.4 Graphical User Interface Development

After performing experiments and finding the best performing model, the next major step is to develop a Graphical User Interface that is capable of showcasing the performance of a model on a new, unseen, and unlabelled example and presenting the results in a clear and user friendly format. This step ties in to the secondary aim and objectives mentioned in Chapter 1. Over the course of the development, five different versions of the program were made, which showed the evolution of its' features and capabilities. Each change in version consisted of a new or modified feature and, for later versions, bugfixing to ensure that the program continues running even if something goes wrong.

4.4.1 Features and Design

The GUI is a small window organised into two sections (using Tkinter frames). The first section is the display, which shows the waveform and spectrogram representation of the chosen audio file. Once a model is loaded, a summary of the model, showing each of the layers and the number of parameters, is also displayed below the audio file and buttons. There are four buttons in this section: a play and stop button that allows a user to listen to the audio file, a button for loading an audio file, and a button for running the model on the audio file and generating predictions from the model. The second section is the predictions, which is only shown when the model is run on an audio file This section has a bar graph, which shows the predictions of each category, in descending order, and a text representation of the predictions, also in order.

The GUI application uses a mixture of object orientated and functional approaches to development. A combination of these approaches allows the application to be developed modularly, making it significantly easier to add, remove, or modify the different functions and variables, and makes troubleshooting significantly easier. An example of where this appraoch helped was in loading an audio file. The load_audio function would check if a user loaded a valid audio file using the file dialog, rather than clicking cancel. If the user did click cancel, then the . The application consists of two main classes: the MainWindow class which contains elements for the GUI, and the ModelLoader class, which is responsible for loading a model and generating predictions from an audio file. The application is made up of three files:

- **main.py**: The file that starts the application. It calls on the mainwindow function runmain. This file has no functions or variables.
- **mainwindow.py**: The main file for the GUI. It contains the MainWindow class and has functions for drawing different sections of the GUI and handling the audio data. There are many variables for the GUI, such as the background/foreground colour, icon, and

window size. There are three variables for the audio: the audio itself, as a numpy array, the audio location, and the audio sample rate for drawing the spectrogram representation.

- **modelloader.py**: The main file for the ModelLoader class and model loading feature. The functions in this class deal with loading the model, converting loaded audio into a specific format, and getting predictions from the model on the audio. The only variable in the modelloader class is the model itself.

There are three main features of the program that aim to meet the secondary aim and objectives of the project.

Feature: Loading Audio

The first feature is the ability for users to load a single .WAV audio file into the program, tying in to the user being able to load a new, unseen, and unlabelled audio. The file should be up to 4 seconds in length, as is the case for the dataset, and fit into one of the ten categories as specified in the UrbanSound8K metadata.

The function that this feature uses is load_audio. This function, located in mainwindow.py, uses a Tkinter filedialog to choose a .WAV file in a folder of 'test' sounds. The "filetypes" parameter enable the filedialog to restrict the search to only .WAV files. It then checks if the user did load a file into the program; if not, the default section is drawn instead. If the user did load audio, the three audio variables (audiofile, audio, samplerate) will be set, with the last two using Librosa's load function. Finally, the draw_disp function is called to draw the elements of the audio in the display frame.

```
1     def load_audio(self):
2         # load audio using filedialog
3         filename = filedialog.askopenfilename(initialdir='test', title="
      Select a WAV file",
4                                               filetypes=((".WAV (Waveform
      Audio File Format)", "*.wav"),))
5         # check if a file has been loaded
6         # if not, then just draw the default window
7         if not filename:
8             self.audio = None
9             self.audiofile = None
10            self.draw_default()
11            self.isPredict = False
12            return None
13        # ERROR HANDLING FOR AUDIO
14        try: # Try to set the audio variables
15            self.audiofile = filename
16            self.audio, self.samplerate = librosa.load(self.audiofile)
17            self.draw_disp()
18        except Exception as e: # Unless error, just draw default window
19            self.audio = None
20            self.audiofile = None
21            self.draw_default()
22            self.isPredict = False
23            return None
```

Listing 4.3: The load_audio function in mainwindow.py

The display frame has a 'play' and 'stop' button that allows the user to listen to the sound. There are also a waveform and spectrogram representations of the audio file. The waveform has a transparent background, no axes, and is coloured red, giving the user an idea

of the structure of the sound before playing it. The spectrogram is a standard spectrogram representation of the audio file, with the viridis colour map highlighting changes in frequency. Below are both functions for displaying the waveform and spectrogram.

```
1     def show_audio_waveform(self):
2         # Get the audio waveform
3         aplot = plt.figure(figsize=(4, 1), dpi=100)
4         aplot.patch.set_facecolor(self.bgc)
5         aplot.add_subplot(111).plot(self.audio, color='red')
6         plt.gca().axis('off')
7         return aplot
```

Listing 4.4: Drawing a waveform

```
1     def show_audio_spectrogram(self):
2         # Get spectrogram of audio file
3         specplot = plt.figure(figsize=(4, 1), dpi=100)
4         specplot.patch.set_facecolor(self.bgc)
5         if self.audiofile: # if audio file is loaded
6             specshow = specplot.add_subplot(111).specgram(self.audio, Fs=
    self.samplerate)
7         plt.gca().axis('off')
8         return specplot
```

Listing 4.5: Drawing a spectrogram

In addition to loading audio, the user is able to choose the processed audio's format. There are four preprocessing options, tying in with the experiments: standard spectrogram, melspectrogram, MFCC, and log-melspectrogram. The user chooses the format using the toolbar, and this can be done at any stage of the program (without any audio, with only the model loaded, with both audio and model loaded, with predictions performed). There are two function used to change the data type: one in mainwindow, which sets the value of the "datatype" variable depending on the option chosen, and one in modelloader, which actually changes the audio type before it is trained on the model.

```
1         mdtype.add_command(label="Spectrogram", command=lambda: self.
    setdatatype(1))
2         mdtype.add_command(label="Melspectrogram", command=lambda: self.
    setdatatype(2))
3         mdtype.add_command(label="MFCC", command=lambda: self.setdatatype
    (3))
4         mdtype.add_command(label="Log-Melspectrogram", command=lambda:
    self.setdatatype(4))
```

Listing 4.6: A section of the draw_default code for selecting the datatype

```
1     def setdatatype(self, number):
2         self.datatype = number
```

Listing 4.7: Changing the datatype

The function below is the preprocessing function that was used in Smales (2019), but also adapted for the different data types available. Due to the varying length of the samples in the UrbanSound8K dataset, zero padding was performed on each audio file. The padding function took the max length of an audio file (4 seconds) depending on the datatype, and subtracted the length of the loaded sound to determine how much it needed to be pad by. The padding is then done using NumPy's pad function.

```
1         if option == 4:
2             try:
```

```
3                 audio, samplerate = librosa.load(sound)
4                 melspec = librosa.feature.melspectrogram(y=audio, sr=
    samplerate, n_mels=128)
5                 # Zero padding
6                 logmel = librosa.power_to_db(melspec)
7                 pad_width = 174 - logmel.shape[1]
8                 logmel = np.pad(logmel, pad_width=((0, 0), (0, pad_width))
    , mode='constant')
9             except Exception as e:
10                print("Error encountered in ", sound, ": ", e)
11                return None
12            return logmel
```

Listing 4.8: An example of processing audio in modelloader (in this case, processing it into a log-melspectrogram

Feature: Loading Models

The second feature is the ability for a user to load a model to run on the audio file, tying in with the user being able to load a model to eventually predict the audio file. This feature activates when the user clicks on the "Load Audio File (.wav)" button in the GUI. The default Keras serialisation format when using model.save is .h5. The model is stored in the modelloader class, which contains functions for loading a model, getting information about it, and running the model on the audio file.

The loadmodel function is first used to load a model into the program. As with the load_audio function, this uses a tkinter filedialog to search for a .h5 file of the model, restricting the search to only the .h5 format, and does not set a model if the user clicks "cancel" on the window. Once the user selects a model, the function calls the Keras load_model function, and sets the class's model variable to the model. This function is assigned to the load_model function in the mainwindow file, which is called in the "Load Model" command in the toolbar.

```
1     def loadmodel(self):
2         modelloc = filedialog.askopenfilename(initialdir='test', title="
    Select a model",
3                                               filetypes=((".h5 (
    Hierarchial Data Format)", "*.h5"),))
4         if not modelloc:
5             return 0
6         else:
7             try:
8                 model = load_model(modelloc)
9                 self.model = model
10            except Exception as e:
11                print("Error when loading model:", e)
12                self.model = 0
```

Listing 4.9: The loadmodel function in modelloader.py

Once the model is loaded, the getmodelinfo is used to get a description of the model, which first checks if there is a model loaded in the class. If there is, the model information is retrieved using the Keras summary function with the print_fn parameter set to add text to an array. This array is then converted to a string and is called in the mainwindow's load_model function, which stores this in the modelInfo label.

```
1     def getmodelinfo(self):
2         if self.model != 0:
3             modelinfo = []
4             # instead of printing to console, append to modelinfo!
```

```
5               self.model.summary(print_fn=lambda x: modelinfo.append(x))
6               modeldesc = "\n".join(modelinfo)
7               return modeldesc
8           else:
9               return "No Model Loaded"
```

Listing 4.10: The getmodelinfo function in modelloader.py

Feature: Getting Prediction from Loaded Audio

The last main feature is getting a prediction from the model on the loaded audio file. This ties in to the greater secondary aim, which is to evaluate a model on a chosen audio file. This feature involves running the model on the loaded audio file, getting predictions from the model, and visualising said predictions in the Predictions frame. This feature is activated when a user has loaded a valid model and audio file, set the correct data format, and clicks on the "Run Model on Audio" button.

The primary function in mainwindow responsible for this feature is the getpredictions function. The function starts by loading the .pkl file containing the class labels, then converts the loaded audio to the correct format. The audio is then converted to a Pandas dataframe with a blank class label and the model is run on the audio to get the predicted class using the Keras predict function.

Afterwards, the predictions text is generated in descending order. The first line of text shows the confidence of the predicted class, which varies depending on the score - if the model is above 90% confident, then it will output "It [audio] is most likely to be: [class]", but if it is only around 50%, it will output "It [audio] may be: [class]". This is followed by the predictions for the other classes. Finally, the function returns the predictions text and their values for plotting.

The function then is called in draw_pred in the mainwindow class, which plots the predictions bar chart and shows the text beneath it.

```
1    def getpredictions(self, sound, type):
2        # Get a dictionary of class IDs and classes to organise into
     predictions
3        # Use below code if classes pickle doesn't exist
4        # metadata = pd.read_csv('UrbanSound8K/metadata/UrbanSound8K.csv')
5        # classes = metadata[['classID','class']]
6        # classsort = classes.sort_values('classID')['class'].unique()
7        # classidsort = classes.sort_values('classID')['classID'].unique()
8        # classes = dict(zip(classidsort, classsort))
9        with open('classeslist.pkl', 'rb') as file:
10           classes = pickle.load(file)
11
12       # Get sound file, convert to model-readable, then create
     predictions list
13       sfile = []
14       class_lab = "None"
15       sound_data = self.convertSound(sound, type)
16       sfile.append([sound_data, class_lab])
17
18       # Now convert features to list for model prediction
19       sfile_df = pd.DataFrame(sfile, columns=['feature', 'class_label'])
20       sfilex = np.array(sfile_df.feature.tolist())
21       sfiley = np.array(sfile_df.class_label.tolist())
22       encoder = LabelEncoder()
23       sfilecat = to_categorical(encoder.fit_transform(sfiley))
```

```
24          sound = sfilex.reshape(sfilex.shape[0], sfilex.shape[1], sfilex.
    shape[2], 1)
25
26          # Create predictions list
27          try:
28              predicted = np.around((self.model.predict(sound) * 100)[0])
29              predictions = sorted(list(zip(list([i[1] for i in classes]),
    predicted)), key=lambda x: x[1], reverse=True)
30              pltx, plty = [i[0] for i in predictions], [i[1] for i in
    predictions]
31              xinorder = np.arange(1, len(pltx) + 1, 1)
32              # Now create a string to present it nicely...
33              predictions_str = ""
34              # Assess how "confident" the algorithm feels about the sound.
35              if 100 >= predictions[0][1] > 90:
36                  predictions_str += "It is most likely to be:\n1." +
    predictions[0][0] + ": " + str(
37                      predictions[0][1]) + ".\nFollowed by:"
38              elif 90 >= predictions[0][1] > 80:
39                  predictions_str += "It is very likely to be:\n1." +
    predictions[0][0] + ": " + str(
40                      predictions[0][1]) + ".\nFollowed by:"
41              elif 80 >= predictions[0][1] > 70:
42                  predictions_str += "It is fairly likely to be:\n1." +
    predictions[0][0] + ": " + str(
43                      predictions[0][1]) + ".\nFollowed by:"
44              elif 70 >= predictions[0][1] > 60:
45                  predictions_str += "It is likely to be:\n1." + predictions
    [0][0] + ": " + str(
46                      predictions[0][1]) + ".\nFollowed by:"
47              elif 60 >= predictions[0][1] > 50:
48                  predictions_str += "It should be:\n1." + predictions[0][0]
     + ": " + str(
49                      predictions[0][1]) + ".\nFollowed by:"
50              elif 50 >= predictions[0][1] > 40:
51                  predictions_str += "It may be:\n1." + predictions[0][0] +
    ": " + str(
52                      predictions[0][1]) + ".\nFollowed by:"
53              else:
54                  predictions_str += "It could be:\n1." + predictions[0][0]
    + ": " + str(
55                      predictions[0][1]) + "\nFollowed by:"
56              predictions.pop(0)  # Remove first element so as not include
    it in rest of string
57              # Print rest of predictions
58              index = 2
59              for category, prediction in predictions:
60                  predictions_str += "\n" + str(index) + ": " + category + "
    : " + str(prediction) + "%"
61                  index += 1
62
63              return predictions_str, pltx, plty, xinorder
64          except Exception as e:
65              predictions_str = "Unable to run model on data: \n" + str(e) +
     \
66                                  "\n\nThis could be due to an incompatibility
     with the model and the data format used\n" \
67                                  "Try switching to a compatible data type."
68              return predictions_str, [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
```

```
0]
```

Listing 4.11: The getpredictions function in modelloader.py

4.4.2 Development: Versions

Over the course of the development of the GUI, five different versions were produced that contain different features, appearances, and bugfixes. Before development started, two flowcharts were used to plan and outline the features and functionality of the application, similar to Figure 4.2.

The initial flowchart shows an early version of the GUI. This version allowed a user to upload their own file and run the model on the file, as with the final version, but also featured the ability to fully train the model on a dataset, as opposed to using a pre-trained model. The GUI would have logged the training and testing accuracies, training time, and plots of the model's training - the same plots that would be used to show the performance of each of the models specified in the implementations. This would mean that, in addition to accomplishing the secondary aim and objectives, users would be able to train their model on another dataset and evaluate the performance while using the application. However, in the end this design was scrapped, as training the model using the GUI would take a significantly longer amount of time than if the model was loaded pre-trained. Additionally, using the GUI to train the models was unnecessary, as the experiments were already performed in Jupyter notebook. Figure 4.3 shows the flowchart developed for the GUI prototype.

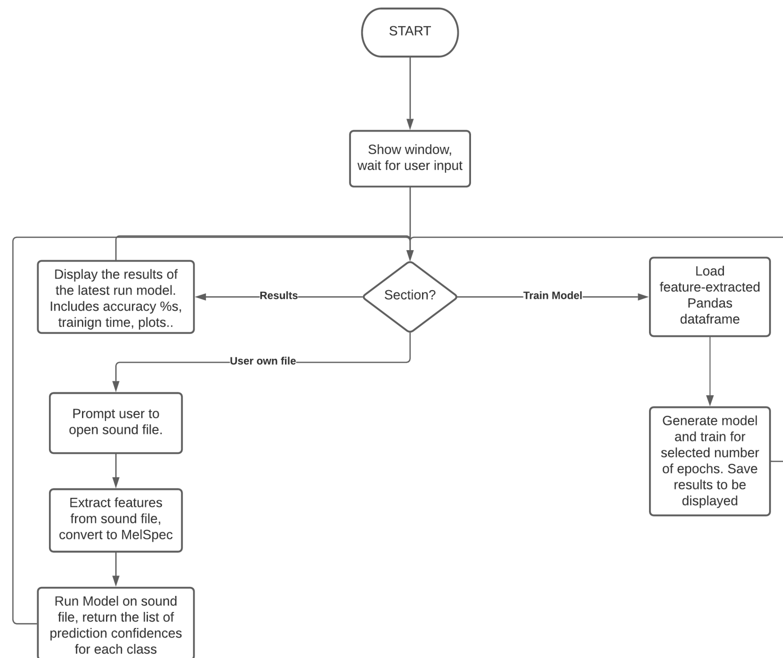

Figure 4.3: Initial GUI Flowchart

The final flowchart shows the features and functionality that would make it to the developed GUI. Rather than training the model in the application, only pre-trained models would be run on the audio file. The flowchart describes the program in more detail, including decision nodes to ensure that users are loading valid files and more options from the main decision node. This flowchart is used to design the final GUI. Figure 4.4 shows this flowchart.

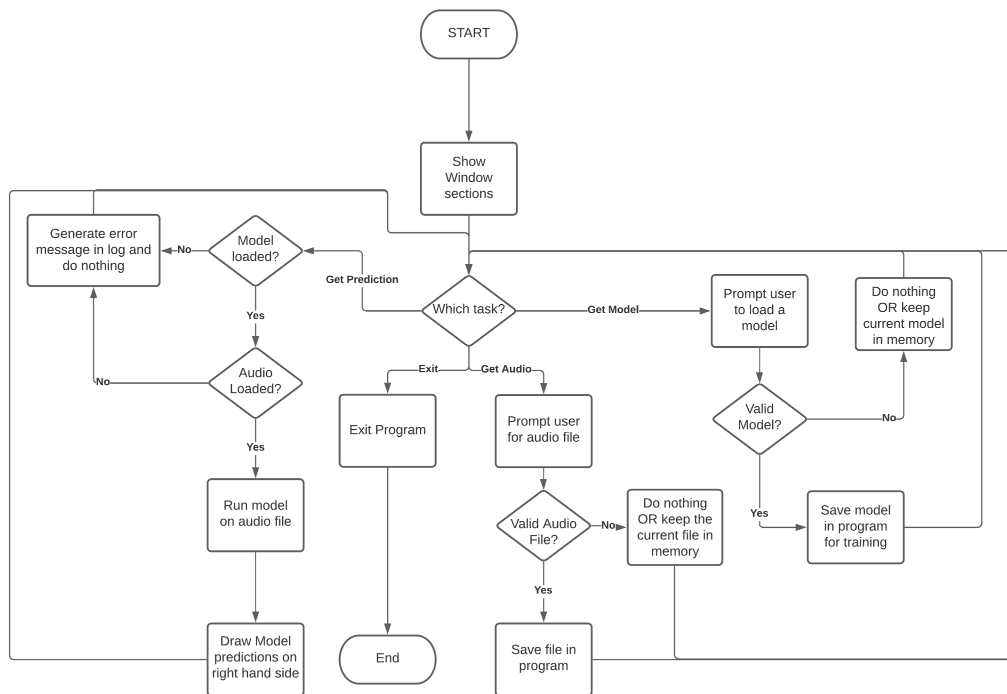

Figure 4.4: Final GUI Flowchart

Version 1.0 of the GUI was developed after the experiments were completed. This version contained both display and prediction frames, but did not feature the ability for a model to be loaded or predictions to be drawn, as the modelloader class was still in development. This version was developed to implement the "load audio" feature and ensure that it was working, before moving on to the other features. The audio's format could not be changed, and only the waveform representation was shown. This version featured the ability to switch between light and dark modes.

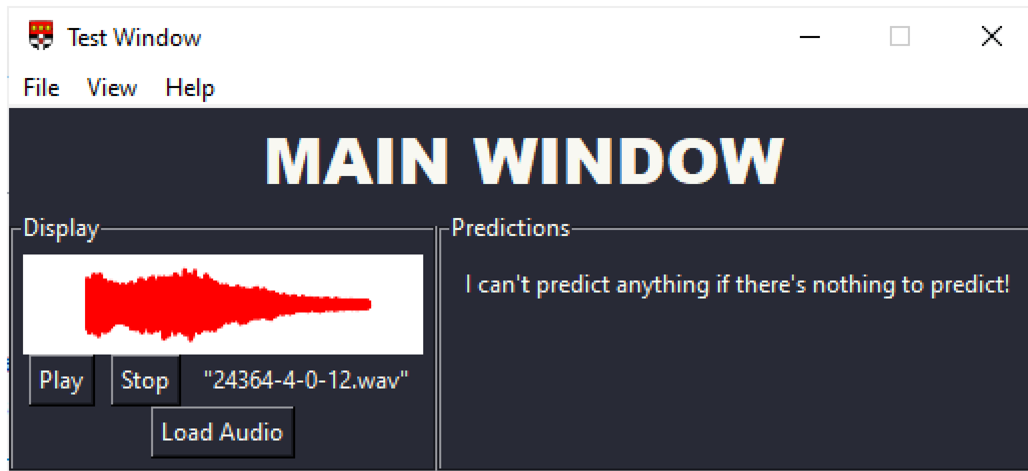

Figure 4.5: Version 1.0 of the GUI

Version 1.1 of the GUI allowed the user to load a model into the GUI and get a series of predictions from the model running on the audio file. However, only textual information was displayed, and the bar chart had not been implemented yet. The audio format could still not be changed, and the model would automatically run on the audio without a button to do so first. The waveform remained the only representation available, and both the audio name and the predictions were subject to text overlapping (in the event that another audio file/model was to be loaded, the previous text would not be cleared).

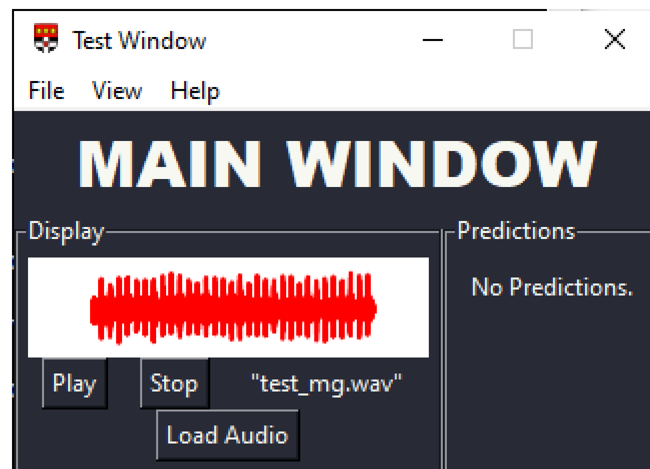

Figure 4.6: Version 1.1 of the GUI

Version 1.2 of the GUI resulted in a major overhaul of the application, with the title and window titles being changed to a more appropriate name. The waveform representation of the audio file was shown more clearly, with the white background being removed in favour of

the window's background colour. The spectrogram representation of the audio file was also shown above the waveform, and the issue of overlapping text was overcome. The predictions frame displayed a bar chart in addition to text, and the user could now manually run the model on the audio file rather than having it run manually. This was implemented to save time in case the user loaded the wrong model, as rather than having to wait for the model to finish predicting the audio file, the user could simply load another one.

However, the user could not change the format of the audio, which would present problems if the model could not run on the current format (all testing was done with log-melspectrograms on the ESC model). Additionally, the predictions bar chart's bars were hard to separate from each other, and the labels on the x axis were not displayed clearly.

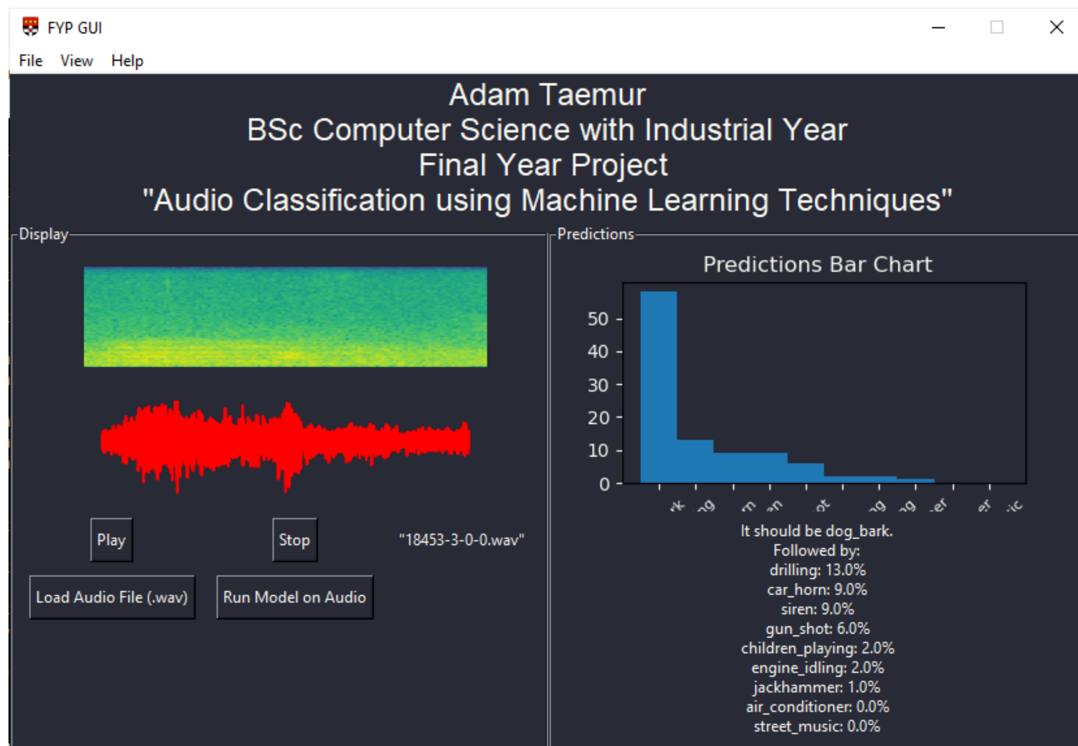

Figure 4.7: Version 1.2 of the GUI

Version 1.3 fixed the bar chart's appearance and issues with the x labels, and more clearly indicated which bars represented which classes. Users could also see a summary of the loaded model, using the Keras summary function. However, the user still could not change the audio format, and the window would not automatically resize with the summary function. The latter meant that if the user was to load a model with a long summary (as in Figure 4.8), then load a smaller model, the window would not resize itself to fit to the smaller model, leaving blank space that the previous model's description took up. The same would happen with the predictions frame.

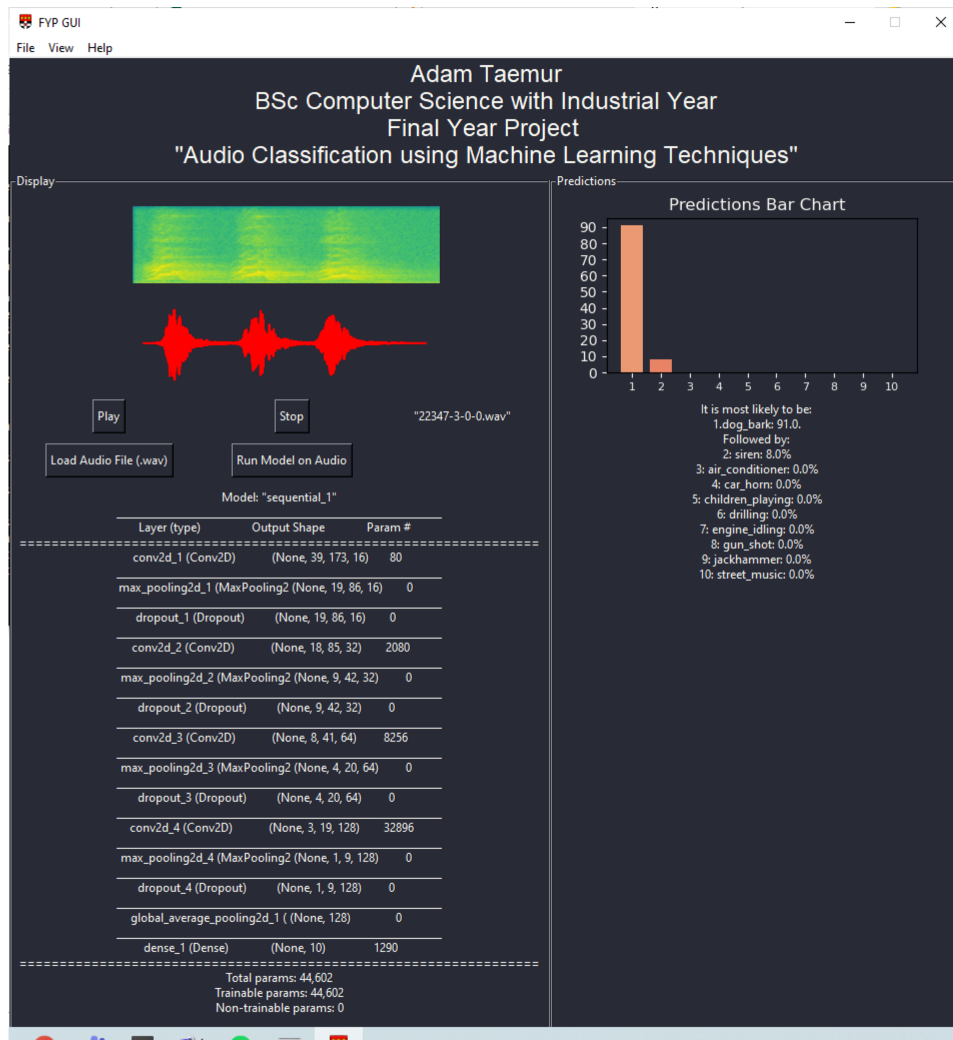

Figure 4.8: Version 1.3 of the GUI

Version 1.4 implemented the ability for a user to select the data type/format for the audio file. It also fixed the issue of the window resizing to fit a smaller or larger model, and provided error handling in the event that the model was run on an incompatible data format. This version would ultimately remain the final design of the application. An additional version, 1.5, would be developed to fix the issue of the user loading an invalid audio file or model.

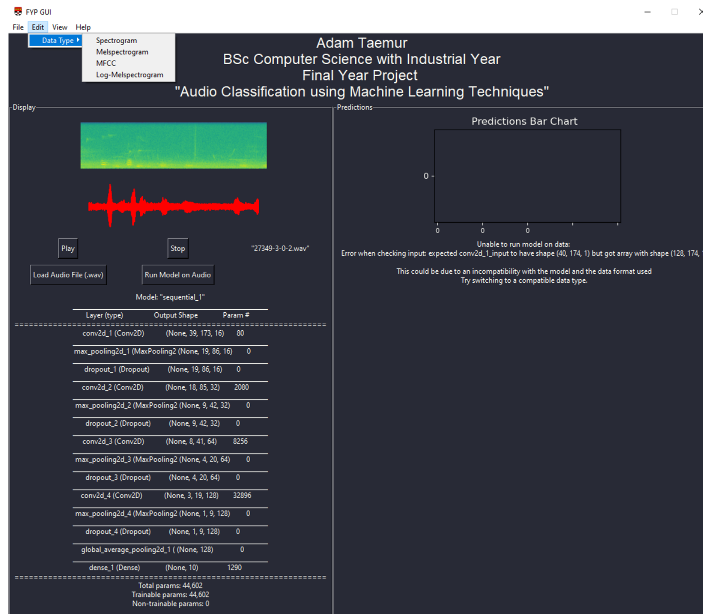

Figure 4.9: Version 1.4 of the GUI

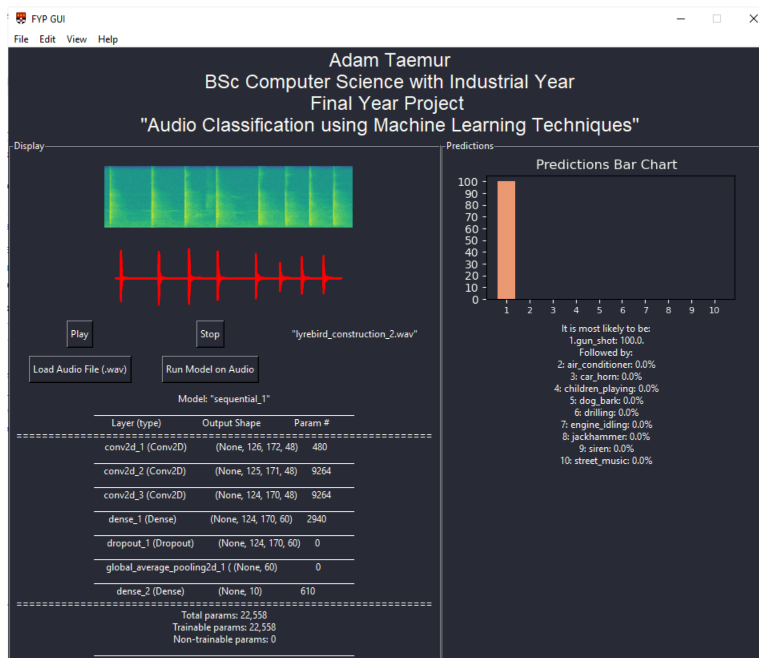

Figure 4.10: The final version of the GUI. Visually similar, but contains additional error handling.

4.5 Summary

The final implementations were selected based on their reported accuracy and modifiability. The two selected implementations were Smales (2019), known as the Soundscape model, and the model without max pooling from Mushtaq and Su (2020). The UrbanSound8K remained as is; data augmentation was not done as it would unfairly impact the performance of the Soundscape model and would require significantly more storage space and possibly longer training times. The experiments made modifications to the models' architectures, data types (for Soundscape), and training parameters, with the aim to reduce the overall number of parameters while maintaining or improving upon the accuracy of the original.

The GUI was designed with the secondary aim and objectives in mind. The Python programming language, as well as the Librosa, Tkinter, and Keras libraries, was used to design the GUI. The GUI has three main functions: load a valid .WAV file, load a model (in the .h5 format), and run the model on the audio to get predictions. Development comprised of two initial designs; one with training occurring in the GUI and one without. The one without training was ultimately chosen as it would take too long to perform and the models would have already been trained beforehand. The application went through five different versions, which affected the features, functionality, and visual design.

Chapter 5

Results and Discussion

The results and discussion section covers the results of both models' sets of experiments and the discussion of these results, relating them to the aims and objectives specified in Chapter 1. This section will also cover the GUI, how it has met its' aim and objectives, and testing and bugproofing that has taken place. Finally, the limitations of the project are discussed, including how to potentially overcome them in future projects.

5.1 Experiment Results

The experiments performed on the models in Chapter 4 record four different metrics for comparison: the Pre-Training Accuracy, the Training Time, the Post Training Trainset Accuracy/Resubsitution Accuracy, and the Post Training Testset Accuracy/Generalisation Accuracy. Below are the tables containing the results of the experiments.

5.1.1 Soundscape Models

The results of the experiments on the Soundscape model are shown in the following tables. Each table corresponds to a different datatype, but the variations of the model and training parameters remain the same throughout.

Table 5.1: Results for the MFCC Soundscape Models

MFCC					
Model #	Model Description	Pre-Train Accuracy	Training Time	Post-Training Trainset Accuracy	Post-Training Testset Accuracy
1	Control Model	4.92%	0:59	94.7%	90.5%
2	Halved Filters	10.47%	0:36	78.4%	76.4%
3	Remove Last Layer	7.96%	1:17	88.3%	85.1%
4	Control, Less Filters	11.79%	1:30	92.4%	88.6%
5	4, Increase Epochs	11.28%	1:28	96.06%	89.4%
6	4, Increase Batch Size	12.48%	1:28	82.8%	80.0%
7	4, Decrease Batch Size	9.79%	1:36	95.1%	89.7%

Table 5.2: Results for the Spectrogram Soundscape Models

Spectrogram					
Model #	Model Description	Pre-Train Accuracy	Training Time	Post-Training Trainset Accuracy	Post-Training Testset Accuracy
1	Control Model	10.83%	6:50	11.7%	9.97%
2	Halved Filters	9.80%	3:32	11.7%	9.97%
3	Remove Last Layer	16.68%	9:04	11.4%	11.5%
4	Control, Less Filters	17.25%	N/A	Unable to Converge	Unable to Converge
5	4, Increase Epochs	14.90%	N/A	Unable to Converge	Unable to Converge
6	4, Increase Batch Size	11.29%	N/A	Unable to Converge	Unable to Converge

Table 5.3: Results for the Melspectrogram Soundscape Models

Melspectrogram					
Model #	Model Description	Pre-Train Accuracy	Training Time	Post-Training Trainset Accuracy	Post-Training Testset Accuracy
1	Control Model	7.56%	3:02	69.3%	67.5%
2	Halved Filters	11.79%	1:36	59.5%	57.9%
3	Remove Last Layer	7.67%	4:02	61.1%	59.0%
4	Control, Less Filters	11.28%	3:36	74.1%	70.9%
5	4, Increase Epochs	11.28%	4:10	78.9%	76.5%
6	4, Increase Batch Size	13.11%	4:40	63.6%	61.4%
7	4, Decrease Batch Size	3.95%	4:32	78.5%	76.7%

From these results, we can see that the MFCC data format produces the best results for the Soundscape models, which is the default data format, and that changing the data format doesn't have any positive effects on the model. This is especially true for spectrograms, in which the performance gets significantly worse for all of the models and ultimately fails to converge for the 4th, 5th, and 6th models. The poor performance of the model on spectrograms and the lesser performance on melspectrograms can be explained by the differences in data shape, and the fact that the model architecture was built to only use MFCC data, as opposed to other papers that used models that could handle different data types, such as Palanisamy et al. (2020) (Log-Spectrograms, Log-Melspectrograms, MFCCs, Gammatone Spectrograms) and Zhang et al. (2018) (Log-Melspectrograms, Gammatone Spectrograms), and the ESC model (Melspectrograms, MFCCs, Log-Melspectrograms).

The 4th model for both MFCCs and Melspectrograms is the most promising variant, having achieved a level of accuracy closest to the original model, compared to models 2 and 3. Because of this, it was chosen for the experiments that modified the training parameters. Experiment 7 is the highest scoring variant of all of the models for both MFCCs and Melspectrograms, and demonstrated that decreasing the batch size can have a positive effect on the training accuracy, equal to increasing the number of epochs.

For the ESC experiments, the datatype remained the same, as the Soundscape experiments demonstrated that changing the data format for the model did not yield any increase in accuracy or other positive effects. Mushtaq and Su (2020) also applied both original ESC models on melspectrograms and MFCCs alongside log-melspectrograms, and discovered that both models performed the best on the last format on the UrbanSound8K dataset.

5.1.2 ESC Models

Table 5.4: Results for the ESC Models

Log-Melspectrograms					
Model #	Model Description	Pre-Train Accuracy	Training Time	Post-Training Trainset Accuracy	Post-Training Testset Accuracy
1a	Control Model	10.7%	17:40	90.1%	87.5%
1b	Control Model with More Epochs	12.7%	29:50	95.21%	90.21%
1c	Control Model with Higher Batch Size	3.4%	16:51	82.6%	80.8%
1d	Control Model with Lower Batch Size	12%	20:07	92.1%	86.1%
2a	Remove Layer, Change Filters	9.9%	15:17	83.6%	80.7%
2b	2 with More Epochs	11.0%	26:57	86.9%	84.2%
2c	2 with Higher Batch Size	10.2%	15:01	72.2%	69.6%
2d	2 with Lower Batch Size	13.2%	17:34	93.05%	89.12%
3a	Reduce Kernel Size, Increase Filters	12.7%	17:05	83.4%	81.5%
3b	3 with More Epochs	4.6%	29:59	87%	84.2%
3c	3 with Higher Batch Size	5.1%	16:54	75.1%	73.6%
3d	3 with Lower Batch Size	12%	19:14	87.9%	85%
4a	Remove Dense Layer and Dropout	4.5%	11:14	89.04%	85.46%
4b	4 with More Epochs	12.1%	19:48	86.96%	84.2%
4c	4 with Higher Batch Size	11.1%	10:39	75.1%	73.6%
4d	4 with Lower Batch Size	12.1%	14:34	93%	89.1%

Like with the Soundscape models, we can see that decreasing the batch size generally leads to an increase in accuracy for all the model variants, as well as increasing the number of epochs (with the exception of 4b). We can also see that the pre-train accuracy does not have any effect on the final accuracy for both sets of models. Model 3a has a higher pre-train accuracy than model 3b, for instance, but does not have a higher post-training accuracy for both the training and test sets. The training time is mostly affected by the number of epochs, but can be affected by the batch size as well. Lower batch sizes mean the model will have to update its' parameters more often, leading to a slightly longer training time.

5.1.3 ESC vs. Soundscape: Scatter Plots

Below are a series of scatterplots comparing the number of parameters with the test set accuracy for each set of models and variants.

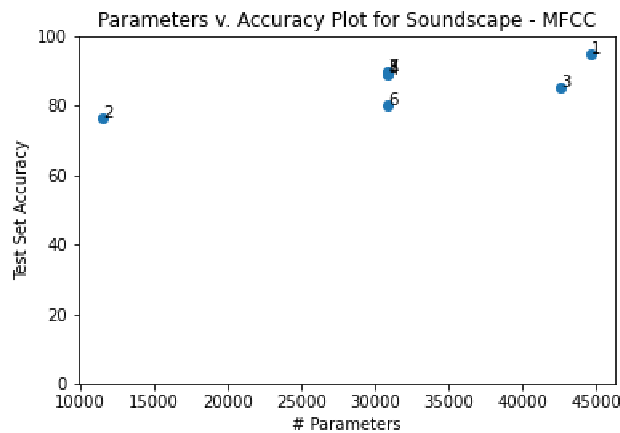

Figure 5.1: Soundscape Models using MFCCs

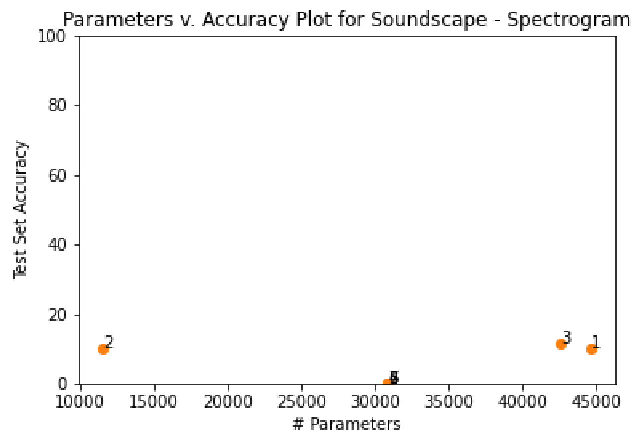

Figure 5.2: Soundscape Models using Spectrograms

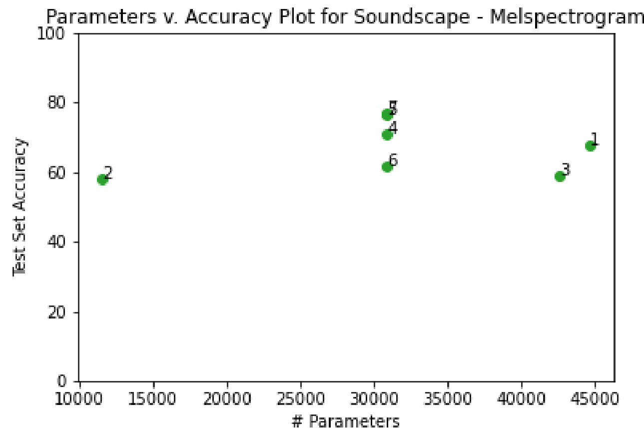

Figure 5.3: Soundscape Models using Melspectrograms

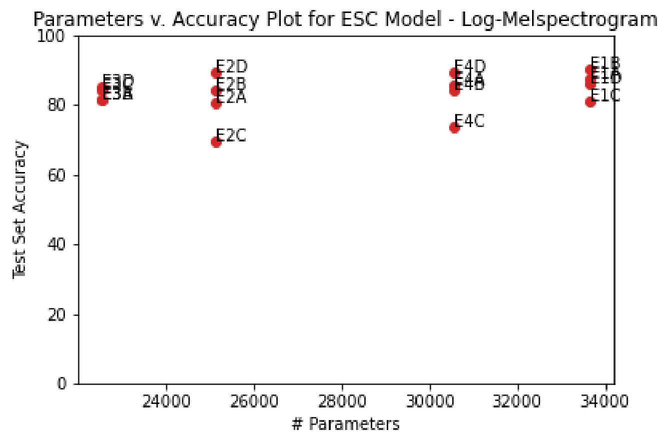

Figure 5.4: ESC Models using Log-Melspectrograms

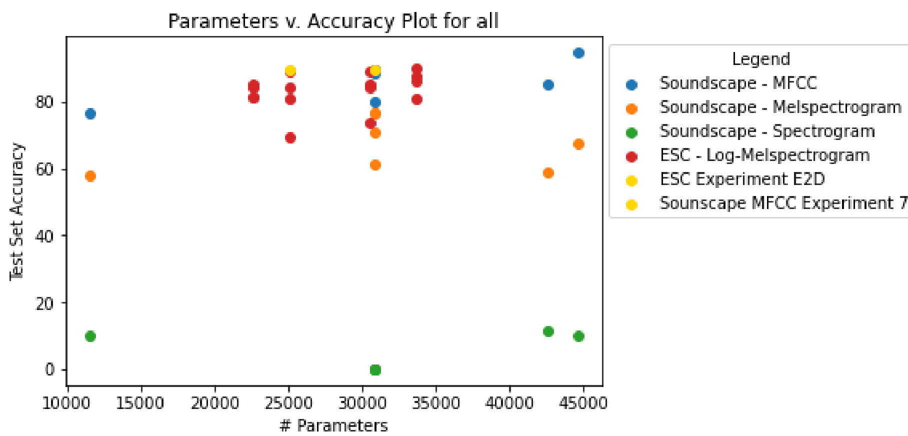

Figure 5.5: Comparison of all models. Points in gold indicate the best performing models for the ESC and Soundscape variants

5.1.4 Tradeoff: Accuracy vs. Number of Parameters

There is a tradeoff between the post training train set accuracy and the number of parameters that will determine which is the best overall model. Since the primary aim of this project is to optimise either of these models for the overall number of parameters WHILE maintaining or improving upon the performance of the original, a model variant that has less parameters than the original is not guaranteed to be chosen as the best if its accuracy suffers significantly as a result; when it gets to the point that the benefit of a lower parameter count wouldn't be able to make up for its' poor performance. For example, the 2nd Soundscape model, where the filter counts were halved across all layers, contained 11,554 parameters, an approximately 75% decrease in parameters, and achieved a test set accuracy of 76.4%, a 12% decrease compared to the original. This would be the perfect model for accomplishing part of the primary aim, which is to optimise the model in terms of number of parameters; however, the decrease in accuracy would make the smaller size worthless as it can be argued that making any modifications to the model that results in less parameters would be a valid solution, rather than one that also maintains or exceeds the accuracy of the original model.

The same can be said for the accuracy. Improving upon the accuracy of the original model without considering the number of parameters is easy; simply adding additional layers or changing the parameters of each layer such that it results in an increase (i.e. adding more filters) would make the model deeper and thus achieve a higher accuracy. This would result in a higher test set accuracy for the model which accomplishes part of the primary aim, which is to improve upon the performance of the original model, but failing to accomplish it in full as the model is not optimised for the number of parameters.

5.2 Best Performing Model

The best performing variant of the Soundscape model is the fourth model using MFCCs with a lower batch size (Experiment 7). This model achieved a test set accuracy of 89.7%, 0.8% lower than that of the original model, and had a total of 30,910 parameters, a 30% reduction in parameters compared to the original, which had 44,602. This model set the number of filters in each layer to 50; where the total number of filters across all layers in the original model was 240 (16+32+64+128), this model has a total of 200, a 16% reduction in total filters. The batch size was halved to 128, and the epochs remained the same. This model took slightly longer than the original model to train (1:36 and 0:59 respectively).

The architecture of this model is as follows:

1. Convolutional Layer with 50 filters, a kernel size of 2x2, and input size of (None, 40, 174, 1). ReLU activation. Followed by Max Pooling of size 2x2 and dropout of 0.2.
2. Convolutional Layer with 50 filters, a kernel size of 2x2. ReLU activation. Followed by Max Pooling of size 2x2 and dropout of 0.2.
3. Convolutional Layer with 50 filters, a kernel size of 2x2, ReLU activation. Followed by Max Pooling of size 2x2 and dropout of 0.2.
4. Convolutional Layer with 50 filters, a kernel size of 2x2, ReLU activation. Followed by Max Pooling of size 2x2 and dropout of 0.2.
5. Dense Layer. 10 units, with softmax activation.

The best performing variant of the ESC model is model 2d. This model achieved an accuracy of 89.4%, which is an increase by 1.9% compared to the original, and has 25,150

parameters, 25% less than the original (33,634). This model has two convolutional layers, each with 48 filters for a total of 96, 11% less than the original. The kernel size has been increased to 4x4 for the first layer and 3x3 for the second, similar to the second and third layers of the original. The batch size was reduced to 8 from 128, a significantly higher difference than that of the Soundscape model, and took a similar amount of time to train.

The original ESC model specified in Mushtaq and Su (2020), which uses data augmentation and does not use max-pooling, achieved an accuracy of 94.14%, a difference of 4.74% compared to model 2d. However, as mentioned in Chapter 4, data augmentation was not performed on the dataset for this project to ensure the Soundscape models did not unfairly suffer in accuracy; therefore this loss in accuracy is deemed an acceptable tradeoff.

The architecture of model 2d is as follows:

1. Convolutional Layer with 48 filters, a kernel size of 4x4, and input size of (None, 128, 174, 1). ReLU activation and L2 regularisation set to 0.001.
2. Convolutional Layer with 48 filters, a kernel size of 3x3. ReLU activation and L2 regularisation set to 0.001. Uses valid padding for the input.
3. Dense Layer with 60 filters, ReLU activation. Followed by Dropout of 0.5.
4. Dense Layer. 10 units, with softmax activation.

Compared to the ESC model, the best performing Soundscape model took significantly less time to train and obtained a slightly higher accuracy. The faster training time can be explained by the higher batch size than the ESC model, which means the model won't have to update its' parameters as often. Additionally, the model has slightly less parameters compared to the original than the ESC model does (30% vs. 25%) while maintaining a similar level of accuracy compared to the original. The ESC model, however, ultimately has the least parameters overall compared to the original and the best Soundscape, and the accuracy exceeded that of the original.

The tradeoff between the number of parameters and the accuracy comes into play here. The difference in accuracy between the best ESC and Soundscape models is marginal at best, but the ESC model has 20% less parameters than the Soundscape model, making this difference an acceptable sacrifice for a model with less parameters. **This makes ESC model 2d the best performing model overall**.

To prove the effectiveness of the ESC model, several individual audio clips not from the UrbanSound8K dataset but from the classes in the dataset were tested on the model using the GUI. Some of these audio clips are from animals that imitate the sounds, such as a rhinoceros beetle that sounds like a jackhammer or a lyrebird making construction sounds, while other audio clips are from genuine sources, including gunshots, a police siren, and a car horn. The purpose of these tests is to show that the model is able to generalise to new examples that are not in the dataset.

The following are the sounds used to test the model, and the results of the model on these sounds (whether it correctly classifies them or not, and why):

- **Shooting Ranges**: two audio clips. One is from an outdoor shooting range with a single gunshot and some minor noise. Another is from an indoor shooting range with two gunshots and more noise than the first clip (including speech); in addition, the gunshot itself echoes more. The model failed to correctly classify both of these audio files, misclassifying them as children playing. This could be due to the presence of noise in the background, which is similar for both the children playing and the street music audio files. The second file, in particular, has longer spikes of amplitude than the first

file does, which could be confused as a child screaming or loud singing in the other classes.

- **Lyrebird Sounds**: The Lyrebird is a bird that is known to be able to mimic natural and artificial sounds such as chainsaws, other animals' calls, and alarms. The two clips that were chosen were the bird imitating a drill, and a sledgehammer. The model managed to correctly classify the first audio file, but classified the second file as gunshots. This is due to the fact that the second audio file contains short, individual spikes in amplitude, similar to some of the gunshot samples, and that the model can be confused by audio samples in one class that have a similar shape to another.

- **Dogs Barking**: Two samples: one of a dog barking in the distance, and a faint dog bark. The dog barking in the distance had short spikes in amplitude in addition to minor background noise, and was correctly classified as a dog bark; however, the second audio file was misclassified as a gunshot. The second file was shorter than the first and had a single dog bark in the middle of the file. In contrast to being an individual spike in amplitude, this file had a 'rounder' shape, indicating a more built-up amplitude than a sudden increase.

5.3 Graphical User Interface and Testing

The final version of the Graphical User Interface, specified in chapter 4, has met the secondary aim and objectives of the project. The application allows a developer to test their model on individual unseen audio files to evaluate its' ability to generalise to new examples. The primary features are loading a valid .WAV file, loading a serialised model (.h5 format), and running the model on the audio file to get the predicted class, in the form of text and a bar chart displaying the predictions in descending order.

In order to ensure the application can persist through any errors or bugs that may occur during use, some basic testing and validation was performed to find and deal with any bugs from the features. Over the course of the development, three major issues were found that could impact the performance of the application. These issues were solved through use of Python try-except statements, similar to the try-catch statements in other programming languages, which are blocks of code that allow developers to deal with any errors that occur during runtime. This may include printing the error, breaking the loop (if it is in a for/while loop), or exiting the program.

The three issues that the user may face when using the application are:

1. **Invalid Audio**: The user loads a corrupted or otherwise invalid .WAV file into the program. The load_audio function would normally take the audio, store it as the mainwindow's audiofile, audio, and samplerate variables, then draw the display section using these variables. A try-except block is used to deal with this issue. If the audio is invalid, the block will set the audio, audiofile, and samplerate variables to None, print the error to the console, and return None on the function. The user can still load another valid audio file without having to restart the application.

2. **Invalid Model**: The user loads a corrupted or otherwise invalid .h5 file into the program. Mainwindow's load_model function uses its modelloader's loadmodel function, which works similarly to the load_audio function. The user would then select a .h5 file from the test directory, and the function would set the object's model variable to the selected model. A try-except block is used to detect if the model is invalid; if it is, then the

function prints the error to the console and sets the model variable to 0. Like with the invalid audio, the user can then select a valid model without having to restart the application.

3. **Incompatible Data Format**: The user loads a valid audio file and model, but the preprocessing format is set to one that is incompatible with the model. This is dealt with in the modelloader's getpredictions function, again using a try-except block. The function will preprocess the audio file normally, but will then try and run the model on the file to get predictions. If the data type is incompatible, the predictions string will state that the model cannot run on the audio file and print the error, and the function returns the string and zero arrays for the bar chart. This string would then be displayed in the predictions section below an empty bar chart.

Refer to Appendix A for the appropriate sections in the code that handle these errors.

5.4 Significance of the findings

The experiments conducted during this project prove that many machine and deep learning models used for environmental sound classification can be optimised for the number of parameters, while maintaining or improving upon the level of accuracy the original model achieved. The best performing Soundscape and ESC models contain less parameters than the original models do, yet manage to maintain (Soundscape) or improve upon (ESC, without data augmentation) the performance of the control/original models. This has been accomplished by making strategic changes to both the model architecture and training parameters, similar to what was performed in the experiments. With the right adjustments, the accuracy of a model can be improved and the model's size/number of parameters can be reduced. Reducing the number of parameters can also result in faster runtimes during training and lower storage requirements. This can be particularly useful for ensemble learning methods, which combine the predictions from multiple connected neural networks to improve the test set accuracy and reduce variance of predictions (Brownlee, 2018). The combination of many deep models with lots of parameters can result in a long training time and a high storage requirement, largely due to the number of parameters to update during training. Taking an approach similar to this project by experimenting with the architecture and training parameters of the models, and the training parameters of the ensemble, can lead to an ensemble that trains quicker, requires less computing power to train, and less space to store.

Both the ESC and Soundscape models manage to outperform many other models trained on the UrbanSound8K and other similar datasets, further proving that a model can achieve SOTA results while being optimised for the number of parameters. Table 5.5 compares the results of the best performing ESC and Soundscape models to other models.

The Graphical User Interface makes it easier for developers to test their models on new individual audio clips. The application is user friendly and easy to use, can handle invalid audio/models and incompatible data formats without crashing or requiring a restart, and presents the predictions in a clear, insightful manner, more visually appealing than a console interface. Developers and researchers can use the application to test their model against any classes of sounds that it may confuse with others and identify any similarities between these classes using the waveform and spectrogram visualisations. The GUI also shows the benefits of using a front-end interface to display the results of a model; in the future, more complex applications that may allow for multiple audio files to be loaded simultaneously or different models for comparison can be developed, in a similar fashion to this application.

Table 5.5: Model Comparison

Model	Data Type	Accuracy
Mushtaq and Su (2020) (with Data Augmentation)	Log-Melspectrogram	95.3%
Smales (2019)	MFCC	91.9%
Soundscape Model 7	MFCC	89.7%
ESC Model 2D	Log-Melspectrogram	89.4%
Li et al. (2019) (CNN)	Melspectrogram	87.8%
Palanisamy et al. (2020) (DenseNet, Pretrained)	Combination	85.14%
Palanisamy et al. (2020) (ResNet , Pretrained)	Combination	84.76%
Palanisamy et al. (2020) (Inception, Pretrained)	Combination	84.37%
Hartquist (2018)	Melspectrogram	84%
Zhang et al. (2018) (1D CNN)	Gammatone Spectrogram	83.7%
Li et al. (2019) (RNN)	MFCC	83%
Seker and Inik (2020)	Scalogram	82.45%
Li et al. (2019) (ANN)	Multiple	82.2%
Salamon and Bello (2016) (with Augmentation)	Log-Melspectrogram	79%
Piczak (2015*a*)	Log-Melspectrogram	73.1%
Dwivedi (2018) (Convolutional-Recurrent)	Combination	53%
Dwivedi (2018) (Parallel CNN-RNN)	Combination	51%

5.5 Limitations

During the experiments performed on both models, as well as the development of the GUI, there were some limitations that affected the potential of the project.

The first limitation is the number of implementations chosen for the experiments. The implementations were chosen based on the overall accuracy and modifiability of the models specified in the literature review and tree diagram; the Soundscape and ESC models were chosen as they had achieved a very high level of accuracy and were created from scratch. The papers had used the Keras library to build and train the model, which made it easy to implement in a Jupyter Notebook and begin experimentation. Although this led to ESC model 2D being the best performing model with the least parameters, thus achieving the primary aim and objectives, implementing other papers that may not have scored as high as Smales (2019) and Mushtaq and Su (2020), but still has a relatively high/SOTA level of accuracy and can be modified, would have led to more models to modify and optimise, with the potential of developing a model that has even less parameters than 2D or one that manages to achieve a higher accuracy. For example, Seker and Inik (2020) achieved an accuracy of 82.45%. Even though it does not perform as well as other models such as Palanisamy et al. (2020) or Hartquist (2018), it is still easily modifiable and could be a good model to experiment with alongside the other two. Even if it doesn't have a major increase in accuracy, it would still be proof that making changes to a model's architecture and training parameters to reduce the number of parameters while maintaining or improving upon the level of accuracy is possible and beneficial.

The lack of data augmentation is another limitation that had an impact on the performance of the ESC model. The original model in Mushtaq and Su (2020) used data augmentation techniques to modify the UrbanSound8K dataset and artificially generate additional training samples, including pitch shifting, time stretching, dynamic range compression, and the addition of background noise. The results show that the models trained on the original datasets with augmented data achieved a higher accuracy than those than train on only the original dataset, which is true for all models, datasets, and data types. The justification behind only using the original dataset is to ensure that it wouldn't unfairly affect the Soundscape model's performance, as the original did not use augmentation; however, the ESC paper did show that data augmentation can improve the performance of a model, even if it is trained on only the original dataset. Using augmented data would also increase the storage demand for the dataset and result in models taking a significantly longer time to train. Even if the Soundscape model wasn't built to deal with augmented data, it would still be worth training and evaluating it on the data to see if it had any hidden potential. Data augmentation would also have boosted the accuracy of the ESC model to the paper's level, in turn potentially boosting the performance of the variants as well, and would not have had any negative impact beyond training the model.

The experiments had modified the model's architecture and training parameters to optimise the models in terms of number of parameters. The modifications that were made resulted in four unique models for both implementations, and modified the architecture in a manner that aims to reduce its size, not make it deeper or more complex (changing the number of filters, kernel size, and removing layers). Although these experiments ultimately led to the best performing models specified above, they missed out on additional modifications that could have been made to other areas of the model architecture, such as the regulariser, the pooling method (for Soundscape), and the optimiser (during the compilation stage). There were also other training parameters that could have been modified, such as the use of validation data, multiprocessing, and more combinations of epochs and batch sizes (for example, reducing the

number of epochs but setting the batch size to 1, essentially Stochastic Gradient Descent). These further modifications to the model and training parameters could have explored the models further and discover better performing models that have even less parameters than 2D or Soundscape Model 7.

5.6 Summary

The results of the experiments show that the best performing variants of the Soundscape and ESC models were the 7th model using MFCCs (for Soundscape) and 2D (for ESC). The best Soundscape model contained 30910 parameters, which is a 30% reduction compared to the original model, and achieved a test set accuracy of 89.7%. The best ESC model contained 25150 parameters, 25% less than the original, and achieved a test set accuracy of 89.4%. The ESC model was chosen as the best performing model, as the difference in accuracy between the Soundscape and ESC models were marginal in comparison to the difference in number of parameters, and is deemed as an acceptable sacrifice for a model with as few parameters as possible. This model has achieved the primary aim and objectives specified in chapter 1.

With the right adjustments to a model's architecture and training parameters, many models that deal with Environmental Sound Classification can be improved and optimised for the number of parameters with a maintained or improved level of performance. A notable example is with ensemble methods that use a combination of neural networks, which could be prone to long training times and high storage and computation demands due to the number of models running simultaneously.

The GUI has managed to achieve the secondary aim and objectives. The final version of the application has features that allow a developer to test their model against individual audio clips, and features error handling that can deal with issues surrounding the primary features and allow it to persist through these issues and continue running. The application makes it easier to test a model against unseen audio files using a simple, user-friendly interface that presents results in a clear and easy to understand format, compared to a traditional command line interface.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Audio classification is a prominent field in machine and deep learning, with Environmental Sound Classification being a particularly popular subtask. Several papers that deal with ESC or other similar types of audio classification were reviewed and compared against each other. The main finding is that, while the performance of most of these models on their respective datasets are high, the majority do not take into account the number of parameters or the model's size, which leads to models that require more storage space as well as more computing power and time to train than may be necessary. The primary aim of this project is to implement a model capable of performing this task by classifying the UrbanSound8K environmental sound classification dataset to a high level of accuracy, while reducing the final number of parameters. The secondary aim is to develop a GUI that allows a user to test their model against unseen data, and display the results of the model on the data.

The two models chosen for implementation and experimentation are the **"Soundscape"** (Smales, 2019) and **"ESC"** (Mushtaq and Su, 2020) models, due to their high test set accuracies and modifiability compared to other models. The experiments aim to maintain or improve upon the performance of the original model while reducing the number of model parameters, and modified their architecture, training parameters, and (for the Soundscape model) the data format.

The results of these experiments found the best performing variations of both models. For the Soundscape model, Experiment 7 using MFCCs achieved a marginally lower accuracy than the original but had 30% less parameters, deemed as an acceptable tradeoff for the reduction in parameters. For the ESC model, model 2D achieved a slightly higher accuracy than the original and a 25% reduction in the number of parameters compared to the original. The best performing model overall is ESC model 2D as it had the least parameters out of the two and managed to achieve a slightly higher accuracy than its' control model. This model has achieved the primary aim of the project. The results of the experiments prove that, with the right adjustments to a model's training parameters and architecture, it can maintain and even improve upon its' original accuarcy while containing fewer parameters, in turn benefitting from a faster runtime, less complexity, less storage space demand, and less computational power required for training and evaluation.

The final GUI has features that enable a user to choose an individual .WAV file to load into the application, select a model and a compatible data format, and get the prediction of the audio file from the model. To display the results, the GUI uses a bar chart and text describing the predictions for each class in order. The GUI was tested using model 2D, and proves to be resilient against common issues that may arise, such as invalid audio or model files and

a model using an incompatible data format. It manages to achieve the secondary aim, and provides a simpler, more user-friendly interface to test a model, displaying information in a more visually appealing and easier to understand format.

6.2 Future work

The results of this project have shown that it is possible to optimise a model's number of parameters while maintaining or improving upon the accuracy of the original. It also shows that a well-developed GUI makes it easier to evaluate the performance of said model against individual, unseen data. However, as mentioned in chapter 5, there were several limitations and challenges that affected the outcome of the project and limited its' potential. These limitations included the depth of the experiments performed on the models, the lack of data augmentation, and the number of chosen models, and while the project's outcome is largely a success, it can be made better by overcoming these limitations.

Future work for this project or similar ones should take into account these limitations when deciding which models and data to use, and the experiments to perform. For example, rather than only using the top two models in terms of accuracy, the top four or five models should be chosen instead to allow for more models for experimentation. Even if the top two models end up outperforming the rest, these additional models will prove that models can be optimised for their number of parameters while maintaining a similar level of accuracy to the original. Additionally, training and evaluating the models on different datasets (including augmented data) will expose the models to more data and see whether the model variations perform similarly on other datasets, further proving the optimisation hypothesis.

Future work may also improve upon the GUI in this project. Restricting the application to loading a single file and model may make evaluating a model(s) more time consuming; thus one possible improvement may be to allow for multiple models and/or audio files to be loaded at once. For multiple models, users can evaluate each of them on the audio file, with the ability to easily switch between each model's prediction. For multiple audio files, users can create a subset that a model can generate predictions for; if the subset includes compatible labels, then the predictions section may even include performance metrics of the model, such as the accuracy, loss, and/or F-score.

Chapter 7

Reflection

At first, the project seemed challenging, especially with background research. I had no prior experience with convolutional neural networks (or deep learning in general) or spectrograms, so learning about both of these concepts proved frustrating at first, especially while balancing this project with other modules in the autumn term. However, over time and by utilising sources that provided easy-to-understand illustrations and descriptions of the models and types of spectrograms, I was able to get a good understanding of the model and data formats, which made the implementation and experimentation stages go smoothly. Identifying the problem statement was easy, but to understand how to solve it, I had to break it down into different stages to understand what tasks should be done, and when to set their deadlines. This developed my project management skills, including time management, scheduling and prioritisation of tasks, and problem solving. The literature review stage allowed me to develop and apply key research skills, including finding the relevant sources and papers, critical analysis and evaluation of these papers, and comparing them against each other. The implementations and experiments required a sense of creativity and a good understanding of each model, including how their architecture and training parameters affect the performance, and developed my decision making skills to decide the trade-off between the number of parameters and accuracy of the best models. Developing the GUI was easy, as I already had prior knowledge and experience of Python and GUI development from second year. Although the project was application based rather than pure software engineering, I still developed software engineering skills such as Object-Oriented Design, testing and debugging, problem solving, and logical thinking when designing, developing, and testing the application. The overall experience of the project was positive, and I felt confident and proud of the work that I have achieved and the self-development I have gone through.

Although the project was largely a success, one challenge that I had faced is the development process of the GUI, which felt a bit disorganised and should have had more of a structure to it. I felt as though I would have benefitted from a more formally planned approach to development; tasks such as devising several visual designs of the GUI's layout before development and outlining how each of the functions will be implemented (possibly using pseudocode) would have made the development process much smoother and may have resulted in an even better final version. This was not the case, however, as I had to focus on ensuring that the final model was able to achieve the primary aim and objectives, leaving less time to plan and develop the GUI.

References

Ali, Z. and Talha, M. (2018), 'Innovative method for unsupervised voice activity detection and classification of audio segments', *IEEE Access* **6**, 15494–15504.

Amin, M. and Nadeem, N. (2018), 'Convolutional neural network: Text classification model for open domain question answering system'.

Brownlee, J. (2018), 'Ensemble learning methods for deep learning neural networks'.

Defferrard, M., Benzi, K., Vandergheynst, P. and Bresson, X. (2017), 'Fma: A dataset for music analysis'.

Dwivedi, P. (2018), 'Using cnns and rnns for music genre recognition'.
URL: *https://towardsdatascience.com/using-cnns-and-rnns-for-music-genre-recognition-2435fb2ed6af*

Goodfellow, I., Bengio, Y. and Courville, A. (2017), *Deep Learning*, MIT Press.

Hartquist, J. (2018), 'Audio classification using fastai and on-the-fly fourier transforms'.
URL: *https://towardsdatascience.com/audio-classification-using-fastai-and-on-the-fly-frequency-transforms-4dbe1b540f89*

Huang, G., Liu, Z., van der Maaten, L. and Weinberger, K. Q. (2018), 'Densely connected convolutional networks'.

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

Li, J., Wang, Y., Zhu, H. and Zhang, Y. (2019), 'Machine learning for urban sound classification'.

Lu, H., Zhang, H. and Nayak, A. (2020), 'A deep neural network for audio classification with a classifier attention mechanism'.

McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E. and Nieto, O. (2015), librosa: Audio and music signal analysis in python, *in* 'Proceedings of the 14th Python in Science Conference', pp. 18–25.

Mushtaq, Z. and Su, S.-F. (2020), 'Environmental sound classification using a regularized deep convolutional neural network with data augmentation', *Applied Acoustics* **167**, 107389.
URL: *https://www.sciencedirect.com/science/article/pii/S0003682X2030493X*

O'Shaughnessy, D. (1987), *Speech Communication: Human and Machine*, Addison-Wesley Publishing Company.

Palanisamy, K., Singhania, D. and Yao, A. (2020), 'Rethinking cnn models for audio classification'.

Piczak, K. J. (2015*a*), 'Environmental sound classification with convolutional neural networks', *2015 IEEE 25th International Workshop on Machine Learning for Signal Processing (MLSP)* .

Piczak, K. J. (2015*b*), 'ESC: Dataset for Environmental Sound Classification'.
URL: *https://doi.org/10.7910/DVN/YDEPUT*

Sahidullah, M. and Saha, G. (2012), 'Design, analysis and experimental evaluation of block based transformation in mfcc computation for speaker recognition', *Speech Communication* **54**(4), 543–565.
URL: *https://www.sciencedirect.com/science/article/pii/S0167639311001622*

Salamon, J. and Bello, J. P. (2016), 'Deep convolutional neural networks and data augmentation for environmental sound classification', *CoRR* **abs/1608.04363**.
URL: *http://arxiv.org/abs/1608.04363*

Salamon, J., Jacoby, C. and Bello, J. P. (2014), A dataset and taxonomy for urban sound research, *in* 'Proceedings of the 22nd ACM International Conference on Multimedia', MM '14, Association for Computing Machinery, New York, NY, USA, p. 1041–1044.
URL: *https://doi.org/10.1145/2647868.2655045*

Seker, H. and Inik, O. (2020), Convolutional neural networks for the classification of environmental sounds, *in* '2020 The 4th International Conference on Advances in Artificial Intelligence', ICAAI 2020, Association for Computing Machinery, New York, NY, USA, p. 79–84.
URL: *https://doi.org/10.1145/3441417.3441431*

Smales, M. (2019), 'Sound classification using deep learning'.
URL: *https://mikesmales.medium.com/sound-classification-using-deep-learning-8bc2aa1990b7*

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014), 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research* **15**(56), 1929–1958.
URL: *http://jmlr.org/papers/v15/srivastava14a.html*

Wang, L., Zhu, H., Zhang, X., Li, S. and Li, W. (2019), 'Transfer learning for music classification and regression tasks using artist tags', *Lecture Notes in Electrical Engineering Proceedings of the 7th Conference on Sound and Music Technology (CSMT)* p. 81–89.

Writer, T. E. (2019), 'Audio classification using cnn - an experiment'.
URL: *https://medium.com/x8-the-ai-community/audio-classification-using-cnn-coding-example-f9cbd272269e*

Yadav, S. and Jadhav, S. (2019), 'Deep convolutional neural network based medical image classification for disease diagnosis', *Journal of Big Data* **6**(113).
URL: *http://jmlr.org/papers/v15/srivastava14a.html*

Zhang, Z., Xu, S., Cao, S. and Zhang, S. (2018), 'Deep convolutional neural network with mixup for environmental sound classification', *CoRR* **abs/1808.08405**.
URL: *http://arxiv.org/abs/1808.08405*

Appendix A

Appendix A: Code Snippets

A.1 Results and Discussion: GUI Error Handling

Below are snippets of the appropriate functions in modelloader.py and mainwindow.py that deal with the errors specified in the GUI in chapter 5

```
        # ERROR HANDLING FOR AUDIO
        try: # Try to set the audio variables
            self.audiofile = filename
            self.audio, self.samplerate = librosa.load(self.audiofile)
            self.draw_disp()
        except Exception as e: # Unless error, just draw default window
            self.audio = None
            self.audiofile = None
            self.draw_default()
            self.isPredict = False
            return None
```

Listing A.1: Error handling for invalid audio, from listing 4.3

```
        try:
            model = load_model(modelloc)
            self.model = model
        except Exception as e:
            print("Error when loading model:", e)
            self.model = 0
        return None
```

Listing A.2: Error handling for invalid model, from listing 4.9

```
        # Create predictions list
        try:
            predicted = np.around((self.model.predict(sound) * 100)[0])
            predictions = sorted(list(zip(list([i[1] for i in classes]),
    predicted)), key=lambda x: x[1], reverse=True)
            pltx, plty = [i[0] for i in predictions], [i[1] for i in
    predictions]
            xinorder = np.arange(1, len(pltx) + 1, 1)
            # Now create a string to present it nicely...
            predictions_str = ""
            # Assess how "confident" the algorithm feels about the sound.
            if 100 >= predictions[0][1] > 90:
                predictions_str += "It is most likely to be:\n1." +
    predictions[0][0] + ": " + str(
                        predictions[0][1]) + ".\nFollowed by:"
```

```
13            elif 90 >= predictions[0][1] > 80:
14                predictions_str += "It is very likely to be:\n1." +
    predictions[0][0] + ": " + str(
15                    predictions[0][1]) + ".\nFollowed by:"
16            elif 80 >= predictions[0][1] > 70:
17                predictions_str += "It is fairly likely to be:\n1." +
    predictions[0][0] + ": " + str(
18                    predictions[0][1]) + ".\nFollowed by:"
19            elif 70 >= predictions[0][1] > 60:
20                predictions_str += "It is likely to be:\n1." + predictions
    [0][0] + ": " + str(
21                    predictions[0][1]) + ".\nFollowed by:"
22            elif 60 >= predictions[0][1] > 50:
23                predictions_str += "It should be:\n1." + predictions[0][0]
     + ": " + str(
24                    predictions[0][1]) + ".\nFollowed by:"
25            elif 50 >= predictions[0][1] > 40:
26                predictions_str += "It may be:\n1." + predictions[0][0] +
    ": " + str(
27                    predictions[0][1]) + ".\nFollowed by:"
28            else:
29                predictions_str += "It could be:\n1." + predictions[0][0]
    + ": " + str(
30                    predictions[0][1]) + "\nFollowed by:"
31            predictions.pop(0)  # Remove first element so as not include
    it in rest of string
32            # Print rest of predictions
33            index = 2
34            for category, prediction in predictions:
35                predictions_str += "\n" + str(index) + ": " + category + "
    : " + str(prediction) + "\%"
36                index += 1
37
38            return predictions_str, pltx, plty, xinorder
39        except Exception as e:
40            predictions_str = "Unable to run model on data: \n" + str(e) +
     \
41                              "\n\nThis could be due to an incompatibility
     with the model and the data format used\n" \
42                              "Try switching to a compatible data type."
43            return predictions_str, [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]
```

Listing A.3: Error handling for data format incompatible with model, from listing 4.11