Delft University of Technology
Master of Science Thesis in Embedded Systems

Pallas: Novel Sound Classification at the Edge

Max Groenenboom

Networked Systems

TU Delft Delft University of Technology

Pallas: Novel Sound Classification at the Edge

Master of Science Thesis in Embedded Systems

Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Max Groenenboom

February, 2023

Author
Max Groenenboom ()
()
Title
Pallas: Novel Sound Classification at the Edge
MSc Presentation Date
February 27, 2024

Graduation Committee

Dr. M. A. Zúñiga Zamalloa	Delft University of Technology
Dr. Kaitai Liang	Delft University of Technology

Abstract

Sound pollution is becoming an increasingly pressing issue in today's world. To effectively address it, it must be measured. To this end, Serval was developed, an edge-ai powered sound recognition solution. Its lack of accuracy, however, makes it difficult to deploy. This thesis examines the potential for improving this solution while staying within its technical limitations in order to raise the accuracy to satisfactory levels.

Multiple aspects of Serval were evaluated and compared to the current state-of-the-art: its data augmentation, the embedding it uses, and the hardware it runs on. Alternatives for each of these components were evaluated and each aspect was optimized.

The results show that after these improvements, the single-label F_1-score increased from 0.60 to 0.76, and the single- and multi-label combined F_1-score increased from 0.64 to 0.67. Finally, power consumption has been reduced by 14%, partially thanks to the usage of specialized hardware.

One issue that has yet to be adequately addressed is the size of the dataset. By increasing the number of samples, the accuracy could be further improved.

Preface

I present the master thesis Pallas: Novel Sound Classification at the Edge, about the development of an improved sound recognition sensor within time and device constraints. It is inspired by Serval, as developed and provided by SensingClues and OpenEars.

I want to thank Marco for his extended help and patience, and Michiel from AMS Institute for providing the OpenEars hardware and the Google Coral TPU. Furthermore I want to thank my parents for proofreading and practicing the presentations with.

Finally I want to thank my cat for offering emotional support throughout the process.

Max Groenenboom

Delft, The Netherlands
29th February 2024

Contents

Chapter 1

Introduction

Sound pollution has been a problem for humanity since ancient times, but with the rapid urbanization today, its scale and impact are growing. It is a major source of discomfort in both urban and rural areas. Recently, the Netherlands has seen an increase in the perception of sound pollution. In the capital city of Amsterdam, the measured noise levels are rising[1], while in the harbor city of Rotterdam, the airport is producing more noise than it is allowed to[2]. In both cities, traffic enforcement cameras are being developed [3][4] to monitor noise levels. Similar research is also taking place in numerous other countries[5][6].

To address a problem, it is necessary to measure it. The most straightforward way is to measure actual sound levels in the environment. A more advanced approach is to identify the source of the sound; knowing the source of sound pollution makes it easier to combat. This can be done by using sound classification. Sound classification can be done most easily using machine learning, of which a popular branch is neural networks. Sound classification with neural networks is a relatively new field [19] with a broad range of applications. These networks are used for speech recognition, sound classification, and other similar tasks.

This thesis builds upon Serval, a sound recognition model being developed by SensingClues. It uses the OpenEars hardware platform to monitor nature reserves and assist in wildlife protection. For instance a detected gunshot may signify poachers in the area while similar natural sounds be distinguished as well. A side project was launched to test Serval in an urban jungle environment, specifically in the city of Amsterdam. This project requires a different dataset than the one used in a nature reserves, however it is imbalanced: some classes it is able to correctly recognize close to 100% of the times, while other classes do not get close to 60%. This can be caused in part by an imbalance in the dataset used, more about this in chapter 4.

Another aspect that keeps this project behind is its dated dependencies. It is built in TensorFlow 1, which is deprecated, and the embedding used has

[1] https://www.parool.nl/ws-b3b0d9f2
[2] https://www.rijnmond.nl/nieuws/1550124
[3] https://www.parool.nl/ws-b43e01fc
[4] https://nos.nl/artikel/2382573
[5] https://www.bbc.com/news/uk-48564995
[6] https://www.nyc.gov/site/dep/news/22-005/

also been preceded by more modern variants. chapter 5 focuses on this aspect. Changing any of these aspects may cause the hardware to be insufficient to run the updated code. To overcome this problem, chapter 6 explores software and hardware possibilities that may be necessary to run the updated code.

1.1 Research Questions

For this thesis, we can formulate the following research questions:

- Can we reduce the class imbalance and improve the accuracy of the Serval model without changing the dataset?
- Is changing the embedding enough to reduce imbalance or improve the accuracy of a model?
- Can we get a model better than the Serval model while staying within computational and timing restraints?

1.2 Report structure

This thesis builds upon Serval, which has multiple problems. Its main drawback is its low accuracy, making it difficult to deploy in its intended scenario. Chapter 2 describes Serval and its components in-depth, and a number of options for improving it.

Because this thesis builds upon a number of existing solutions and research, chapter 3 explains the methods and research that made this thesis possible. It describes the main problem that Serval and this thesis aim to solve and provides an introduction into data augmentation, machine learning, and embeddings that may be necessary to understand this thesis.

The main workload can be divided into three main topics. The first of these topics is data augmentation. Serval already employs data augmentation in its pipeline, however fine-tuning their approach may improve the quality of the input data and the trained model. Chapter 4 explores different methods of data augmentation, and we find that a method that combines an improved version of Serval's data augmentation with a novel approach improves the accuracy of the final model.

The second major topic is the embedding that Serval uses. Serval uses an embedding to facilitate the training of a model, which would otherwise be a rather costly and slow task due to the size and complexity of the model. Chapter 5 explores an alternative embedding and its possible configurations, and we discover that an unexpected configuration yields the best results.

Thirdly we explore the software and hardware platform OpenEars. As the alternative embedding mentioned above is a lot larger and slower compared to Serval's original embedding, the platform and the software may need adapting in order to run in real-time. chapter 6 explores the different considerations that may need to be taken to run a larger model in real time on the same hardware.

Finally the thesis concludes with a conclusion in chapter 7. Here we review everything we learned, reflect on the process of writing and working on this thesis, and conclude with some options for future work.

Chapter 2

Project inspiration: Serval

This thesis is based on Serval[1](Sound Event Recognition-based Vigilance for Alerting and Localization), a product currently being developed by SensingClues[2]: a Dutch foundation that develops smart solutions for nature conservation and wildlife protection, in collaboration with Sensemakers[3]: an Amsterdam-based community that focuses on IoT and smart city solutions. The purpose of Serval is to help with wildlife monitoring in remote areas. This chapter explains what Serval is and what can be improved. In addition, it presents potential concrete improvements that we propose.

2.1 Serval

Serval[4] is an algorithm for training a sound recognition model. It is being developed in conjunction with OpenEars[5], a software + hardware platform to deploy smart microphones and a server with a dashboard that displays the measurements and predictions of these sensors. One of these devices is shown in Figure 2.1. These sensors are intended to be used in wildlife reserves to track wildlife behavior - particularly elephants' - and human presence in the form of chainsaws, engine sounds, and gunshots. Sounds travel much further than visuals, making it possible to map large areas with relatively few and relatively inexpensive sensors. Mapping wildlife behavior is invaluable in scenarios where such wildlife needs to to be protected. Measuring human presence enables rangers to identify early on where poachers and illegal loggers are active, allowing them to more effectively combat them.

Serval is based on the Youtube-8M Tensorflow Starter Code[6] repository, a demo for using the Youtube8M[2] dataset. This demo and Serval are powered by Tensorflow 1[1], a machine learning framework for Python. Serval also uses data augmentation to reduce the class imbalance in the dataset and increase the overall amount of samples. Finally the embedding VGGish[13] is used to extract

[1] https://sensingclues.org/news/category/Serval+Sensor
[2] https://sensingclues.org/
[3] https://www.sensemakersams.org/
[4] https://github.com/SensingClues/serval
[5] https://github.com/SensingClues/OpenEars
[6] https://github.com/google/youtube-8m

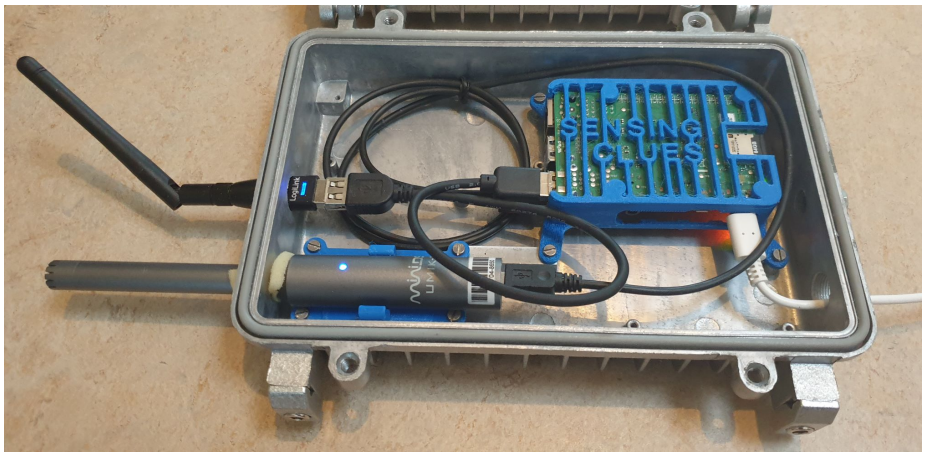


Figure 2.1: **An opened OpenEars device**

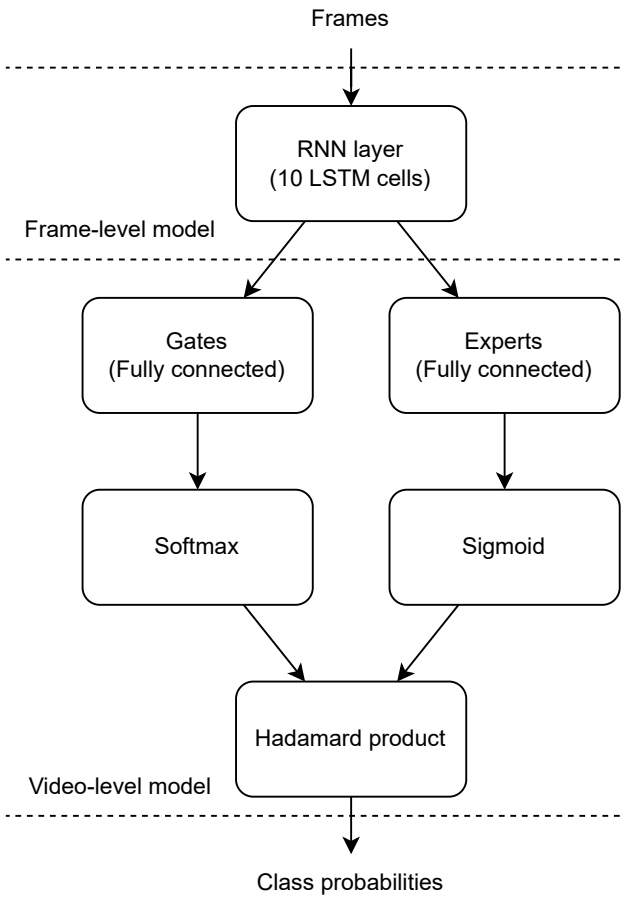


Figure 2.2: **The layers that the Serval model and the Youtube8M model consist of**

features from audio segments, facilitating the training of the Serval model. The precise layout and nature of VGGish is detailed in chapter 5.

The Serval model consists of two sub-models as shown in Figure 2.2: a frame-level model that runs on the variable-length list of frames of which the video (or audio) consists and a video-level model that runs on the output of the frame-level model. The frame-level model is a long short-term memory (LSTM) model consisting of 1 layer of 10 LSTM cells and the video-level model consists of a Mixture of (Logistic) Experts (MOE) model as proposed by Jordan and Jacobs [15]. This MOE model is implemented as two fully connected layers that are multiplied by each other. The model is intended to recognize which sound classes can be heard in the sample. To do this it calculates the probabilities that a given class is featured in the sample, after which this is compared with a predetermined threshold to decide whether this class will be considered to be present or not.

OpenEars is powered by a Raspberry Pi 4 Model B Rev 1.1 [7] and is being developed by SensingClues and Sensemakers. It uses a UMIK-1 omnidirectional USB microphone[8]. The software is based on devicehive[9], an open source IoT data platform, and communication is done using MQTT, a lightweight messaging protocol.

Additionally, Serval is coupled with an urban sound dataset collected by Waag Society and Sensemakers in the city of Amsterdam. The main goal of Sensemakers is to gain insight into the intensity and sources of noise pollution and signs of criminal activity while further developing the wildlife sensor in an easier environment.

2.2 Problems with Serval

Serval has some issues that need to be addressed. It is based on the Youtube-8M demo code, which is rather outdated (the most recent updates date back to 2019) and is powered by Tensorflow 1, which is no longer supported.

A mayor issue with Serval itself is caused by its structure. Serval is composed of 12 Jupyter (A web based interactive coding environment)[16] notebooks. These notebooks are great for interactive and collaborative work, but a lot harder to use from the command line or in an automated environment. Furthermore, a copy of the configuration is stored in duplicate in each notebook, making it difficult and prone to errors to change a single configuration or path, as these need to be changed in every file.

Finally, the accuracy of the model is inadequate, making it impossible to deploy the system in the real world, making this the most pressing problem to solve.

[7] `https://www.raspberrypi.com/products/raspberry-pi-3-model-b/`
[8] `https://www.minidsp.com/products/acoustic-measurement/umik-1`
[9] `https://github.com/devicehive/devicehive-audio-analysis`

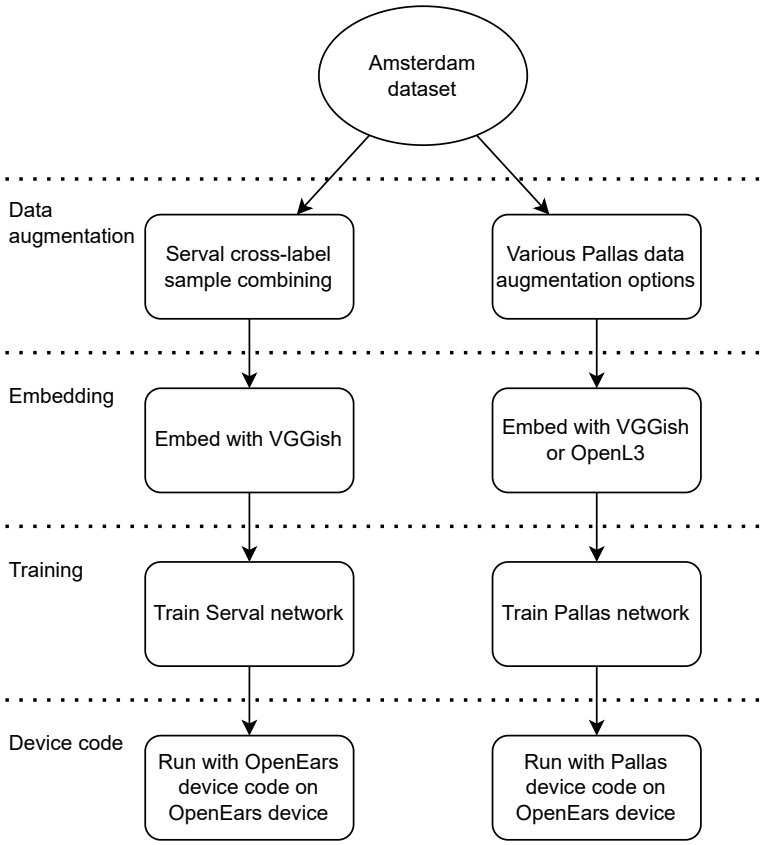


Figure 2.3: **Serval's components on the left compared to Pallas' components on the right**

2.3 Proposed solutions: Pallas

In order to address both the issue of being outdated and the issues with its structure, the entire codebase must be rewritten. This new codebase goes by the name of Pallas. Pallas still uses Jupyter notebooks, but functions and constants are moved to separate Python files to enhance usability. Pallas is powered by Tensorflow 2 and a single configuration is used by all other functions and scripts.

To improve the accuracy of the model, several aspects must be revised. First, the data augmentation part of Serval is improved. Since the dataset is imbalanced and does not accurately reflect the final objective of the model, the dataset is augmented before being used to train the model. As this is a critical step in Serval, the data augmentation method used by Serval is evaluated and improved.

Also, the embedding is replaced with L^3-Net[3], or specifically OpenL3[10], an open-source Python library that makes L^3-Net much easier to use. This helps to improve the model's accuracy. The downside of this change is that OpenL3 and the embeddings it generates are a lot larger than VGGish and its embeddings.

Finally, the OpenEars code is powered by TensorFlow 1 as well. As Pallas

is powered by TensorFlow 2, certain work is necessary to run this new code on the device. Also, the device software and hardware must be optimized to ensure the much heavier OpenL3 model is able to run in real-time.

All components of Serval and their updated components in Pallas can be found in Figure 2.3.

Chapter 3

Required background and related work

3.1 Acoustic Event Detection

Serval can be seen as a potential answer to the acoustic event detection (AED), or specifically the environmental sound classification (ESC) problem: recognizing and categorizing sound events. These solutions are designed to constantly monitor sounds from the environment and detect and classify certain sound types. In the case of Serval, these are either urban or natural sounds, depending on the dataset used to train the model.

Comparable projects have been implemented around the world to measure noise pollution. Recently, researchers from New York University[6] worked on a similar sensor network deployed in New York City, called SONYC – Sounds of New York City. Both SONYC and Serval focus on measuring and analyzing environmental sounds. The main difference between these projects is that SONYC defines a complete framework that not only measures and analyses environmental sounds, but also visualize it and even formulate ways to actuate on it. Serval and Pallas on the other hand focus on (autonomous) hardware that analyses data on the edge, and reports to a client.

Another earlier project is the IDEA (Intelligent Distributed Environmental Assessment) project [7] from Flanders in Belgium, IDEA can be seen as a standard sensor network. The main difference in the sensors is that the sensors do not do any analysis but transmit the data to a server. Connected to this server are autonomous agents doing various kinds of analysis. Serval and Pallas of course do this analysis on the edge, to minimize the amount of data sent and reduce the privacy impact. Finally this approach requires a massive data collection and storage server, serving not only multiple sensors but also multiple analysis agents. Serval and Pallas on the other hand do not require such a server. For these only a small server would suffice.

A final project that was examined is MESSAGE[5] (Mobile Environmental Sensing System Across Grid Environments) from Newcastle University, deployed in the UK and Palermo, Italy. Again, this project has a rather different focus compared to Serval and Pallas. This project focuses on two parts. First focus point is on the hardware. An inexpensive autonomous *mote* was developed, in

order to make it cheaper and more attractive to deploy. Serval is still in its development phase, using a prototyping platform for its development. Furthermore, these *motes* do not support sound recognition, only sound pressure levels are recorded. The second focus is on clustering the sound pressure levels recorded depending on their environments: The type of road they are placed at, as well as the presence of objects such as schools, traffic lights and bus stops, and the effect of wind strength and wind direction compared to the street direction. The aim of Serval is the opposite, to sense at larger scales, to detect wildlife and human presence, and not local effects.

Of course, there are many more projects that focus on this problem. A handful of promising and differing solutions was chosen for comparison, while there may be more similar solutions.

3.2 Machine learning

Machine learning is a broad field with even broader application. It uses statistics and generalization to train models that are able to do a plethora of tasks, including classification, segmentation, and recently even content generation. Serval uses it both in its embedding, VGGish, which is a deep convolutional neural network (DCNN), and the downstream model, a long short-term memory, which is a type of recurrent neural network (RNN).

One of the earliest researches into ESC with DCNN was done by Piczak[20] in 2015. He considered that while DCNN has shown promising results in multiple fields already, including sound analysis such as speech recognition and music analysis, no attempt was made yet to apply DCNN to ESC. This research shows promising results, especially considering its datedness.

Another relatively new type of RNN is the attention-based model. This model is able to focus on different weights, depending on the context. Zhang et al.[28] used this method to accurately do ESC, albeit with a limited dataset. This method shows promise however, especially when considering the size of the model. Attention might be an interesting option to explore apart from using an LSTM.

Jatturas et al.[14] compare older feature-based methods with newer DCNN methods, comparing support vector machine (SVM) and multilayer perceptron (MLP) methods with a method consisting of a DCNN model for its feature extraction with fully connected layers as the classifier. The latter is similar to what Serval and Pallas are doing, using an external DCNN model as its embedding to do the feature extraction. The main difference with this paper is that Serval and Pallas use an LSTM instead of simply fully connected layers, although fully connected layers are still used for classification after the LSTM.

3.3 Data augmentation

In order to train any machine learning model, one must have training data. This data can be either labelled or unlabelled, depending on the type of machine learning. Depending on the complexity of the problem, it may be necessary to increase the amount of data or diversify the existing data. Data augmentation is a technique to modify existing training data or generate new training data.

This method can be applied to both image and audio data. The audio data is usually converted to a visual representation, a spectrogram, and then data augmentation can be applied to the spectrogram. This allows one to use image data augmentation methods on audio data. Many data augmentation techniques exist for images, with differing effectiveness and efficiency, as shown by Shorten and Khoshgoftaar[23]. These techniques include basic transformations such as flipping, rotation, panning, cropping, and noise injection. Other options include altering the brightness of the image, or any other color space transformations when using colored images, if possible in the problem context. Finally more complex techniques may randomly mix images, cut parts out, or even use specialized neural networks for generating completely new data.

When dealing with audio, there are more options besides image data augmentation on the spectrogram. One option is to combine multiple samples together. This can be done in multiple ways: Takahashi et al. [26] propose that combining samples of the same class yields new samples of that class: A sample containing two (different) cars should be tagged as a sample with cars. In multi-label classification, samples from different classes can be combined to yield multi-label samples: A sample containing both a car and a gunshot should be tagged as having both car sounds and a gunshot in it, in a multi-label classification task. This is a novel type of data augmentation that Serval uses. Other options include injecting noise into the samples, performing a time shift on the samples, or changing the speed, pitch or volume of samples, as discussed by Salamon and Bello [22].

Serval uses data augmentation to achieve two goals: first of all to rebalance the dataset, and secondly to get multi-label samples that are not present in the original dataset. This technique and other options will be evaluated to improve the final accuracy.

3.4 Embedding

Training a deep neural network to complete a complex task such as acoustic event detection (AED) requires a large dataset and a lot of computing power. An embedding is a relatively new solution that can be used when these requirements are not met. It consists of a model that extracts features from an input that can be used to complete the AED task. An example of this is a model that extracts facial features for face recognition. When using such an embedding, one only needs to train a model to complete the AED task using the features as input.

There are numerous examples of embeddings that have recently been developed. An early example is SoundNet, developed by Aytar et al.[4] in 2016 at Massachusetts Institute of Technology. It uses a student-teacher model to train a deep convolutional neural network to extract features from samples, which are validated by a one-layer SVM (Support Vector Machine), a machine learning

algorithm.

Concurrently at Google, Hershey et al.[13] developed VGGish (Named after VGG[24], which itself is named after the Visual Geometry Group of the University of Oxford). This was an experiment to show that deep convolutional neural network developed for image recognition are also excellent at recognizing sounds in spectrograms. This model was further adapted to an embedding and released to be used as such.

Meanwhile, Arandjelović et al.[3] developed L^3-net through a novel approach to train both a sound and video embedding: the correspondence between video and audio was used to train a network to predict whether an audio and a video segment are from the same video. This yields two sub-networks - one for video and one for audio. The audio sub-network proved to be an excellent embedding for sound recognition, almost on par with human sound recognition. While it is a much more complicated and large model, it is an excellent candidate to improve Serval's model accuracy.

Chapter 4

Data augmentation

The most obvious way to train a model is through supervised learning with a labelled dataset. However training a model for acoustic event detection (AED) is a challenge due to the need for large amounts of training data and the high cost of labeling such data.

As presented in section 3.3, data augmentation is one possible solution. Also, Serval already employs data augmentation, so exploring the different options here feels like a promising option. Furthermore data augmentation can be used as a way to eliminate or mitigate class imbalances. Serval is able to recognize multiple classes at the same time. If a sample contains both car noises and a gunshot, Serval is able to classify both of these sounds at the same time. This requires a dataset with multi-label samples, samples with sounds from multiple classes occurring at the same time or shortly after each other. However, data augmentation can be applied to a dataset with single-label samples to generate multi-label samples, which is necessary to train a model to do this.

As we can see in Figure 4.1, Serval's dataset suffers from heavy class imbalance: for each motor sample there are almost 17 gunshot samples. Furthermore with 3413 samples the amount of samples seems to be rather small for a modern dataset, considering the amount of samples in commonly used or even dated audio datasets as seen in Table 4.2. Furthermore, not all samples are in the same samplerate. This is evident in Table 4.1. While this should not be a problem for VGGish as it uses 16KHz input, OpenL3 uses 48KHz input which may compromise the improvements gained from using the newer embedding.

This chapter explores multiple possible data augmentation techniques. Section 4.1 details the data augmentation technique that Serval uses, while section 4.2 explores an improvement upon this method. Section 4.3 explores a method using noise to differentiate samples and finally section 4.4 details a method combining multiple of the mentioned methods to achieve a hybrid type of augmentation.

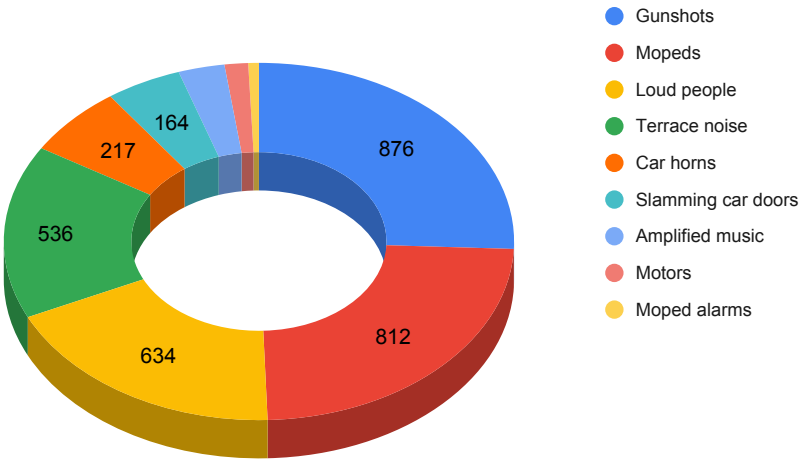


Figure 4.1: **The amount of samples in the dataset for each class. The imbalance between classes can clearly be seen.**

	16000	44100	48000	Total
Gunshots	0	876	0	876
Moped alarms	0	0	23	23
Mopeds	0	0	812	812
Car horns	85	0	132	217
Slamming car doors	0	0	164	164
Loud people	0	0	634	634
Motors	0	0	51	51
Terrace noise	0	0	536	536
Amplified music	66	0	34	100

Table 4.1: **The amount of samples per sample rate in the dataset for each class.**

Dataset	Year of introduction	# of Samples	# of classes
ESC-50[21]	2015	2000	50
ESC-10[21]	2015	400	10
SoundNet[4]	2016	2140000	N/A (unlabelled)
Audio Set[11]	2017	1789621	632
Serval	2020	3413	9

Table 4.2: **Different audio datasets with their amount of samples and amount of classes listed.**

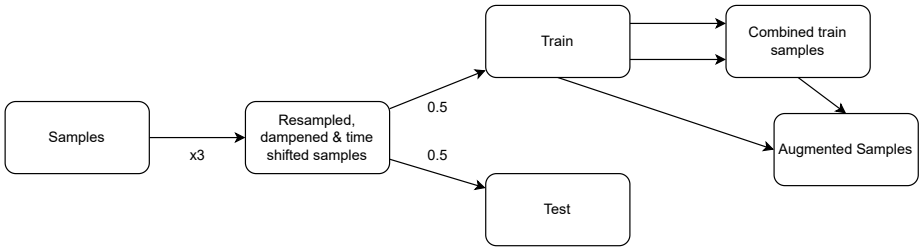


Figure 4.2: **Serval cross-label sample combining. First samples are resampled from their original frequency to 16KHz. These samples are then dampened by 6 and 12 dB. Then they are split 50%/50% in a train and a test set. Finally the training samples are combined to generate a larger and more balanced training set. For each combination a sample is combined with a softer sample of a different class.**

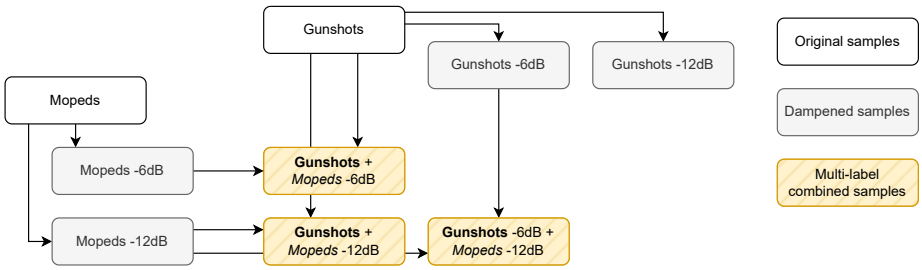


Figure 4.3: **Serval cross-label sample combining detail with only two classes. For each class, dampened samples are created. The original and dampened samples are combined with samples from other classes. The primary class is the class of the loudest sample used for the combination.**

4.1 Serval cross-label sample combining

A particular data augmentation technique that is interesting to us is is the technique employed by Serval, which involves combining samples with samples with a different label, yielding new multi-label samples. This technique will be referred to as Serval cross-label sample combining in the remainder of this thesis.

Figure 4.2 shows the steps in this data augmentation and Figure 4.3 illustrates the data augmentation process in detail with two example classes. If we take only the gunshots and mopeds classes, first all gunshot samples are time-shifted: they are cut in half at a random place and then wrapped around. Do note that this may cut in the middle of the relevant event, however this may happen in real time recordings as well and may improve the model's ability to correctly recognize in such cases. Then they are resampled to 16000 kHz (the frequency used by the embedding VGGish) and copies dampened by 6 dB and by 12 dB are created. These dampened samples will be referred to as gunshots -6dB and gunshots -12dB from now on. This is done for all classes. Half of these samples are reserved for validation.

	1	2	3	4	5	6	7	8	9
Gunshots (1)	1236								
Mopeds (2)	1493	1266							
People noise (3)	1497	1497	966						
Terrace noise (4)	1493	1494	1491	753					
Car horns (5)	1484	1493	1482	1482	291				
Slamming car doors (6)	1486	1488	1483	1484	1447	204			
Amplified music (7)	1478	1486	1476	1470	1422	1413	156		
Motors (8)	1463	1460	1455	1438	1361	1326	1280	81	
Moped alarms (9)	1403	1424	1407	1382	1220	1118	1056	767	36
Total primary	13033	11608	9760	8009	5741	4061	2492	848	36
Total	13033	13101	12754	12487	11682	11449	11237	10631	9813

Table 4.3: **The amount of samples per class combination after cross-label sample combination and the total amount of samples per class. Horizontally are the primary labels, vertically the secondary labels. The total primary counts all samples where the primary class is the given class. The total either counts all samples where the given class is present–either as primary or as secondary label.**

Subsequently, a number of samples are created by combining gunshot samples with softer moped samples. This results in three new multi-label sample sets: gunshots combined with mopeds -6dB, gunshots combined with mopeds -12dB, and gunshots -6dB combined with mopeds -12dB. These samples have gunshots as their primary label and mopeds as their secondary label, as the gunshot sample was louder than the moped sample. No other combinations are made as the moped samples may not be louder than the gunshot samples due to the gunshot samples being earlier in the order of classes. This seemingly arbitrary decision will be addressed in section 4.2. For each possible combination allowed by these rules, by default 500 samples are created. Finally, all dampened single-class samples are also included in the dataset.

Why does Serval use this data augmentation? First of all, this is an effective way to rebalance the classes. As previously seen in Figure 4.1, there is a significant imbalance between the amount of samples of classes. The augmented dataset in Table 4.3 demonstrates that this imbalance has been completely eliminated, and the number of samples has been significantly increased. Additionally, this data augmentation technique allows the model to be trained for multi-label classification without requiring a multi-label annotated dataset.

It can also be observed that the single-label samples on the diagonal are still highly imbalanced and few compared to the multi-class samples. This may compromise the model's ability to accurately label classes that contain only a single sample. Moreover, combinations above the diagonal are omitted. These are samples where the 'later' occurring class has a higher volume. This results in the moped alarms class having only dampened samples in the augmented dataset, which may compromise the model's ability to recognize this class.

Additionally, dampening is done before doing a train test split. This results in samples and their dampened variants being in both of the sets. For example if we take a random sample, the original volume sample may be in the train set, with the dampened variant in the test set. These samples may be too similar causing the test metrics to be unreliable by making it harder to detect overfitting.

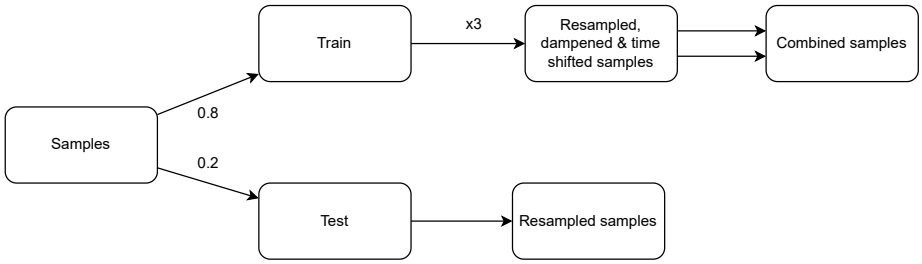


Figure 4.4: **Improved sample combining. Similar to Serval cross-label sample combining, however some of the flaws were fixed. First of all the train/test split was done before the dampening. Furthermore no uncombined samples are added, instead single-label samples are combined samples as well**

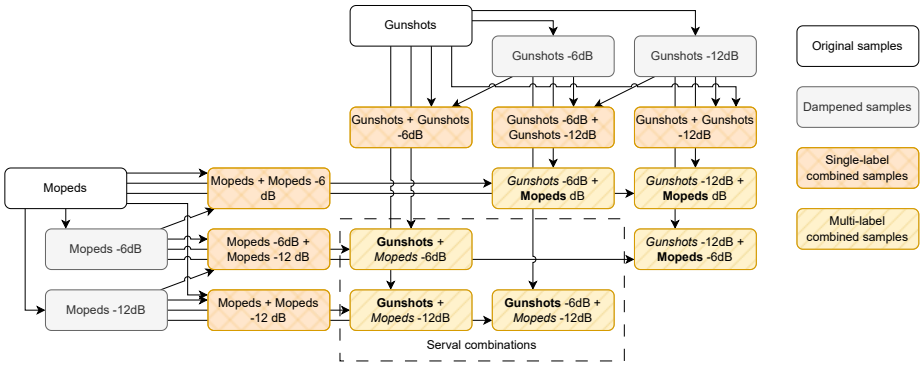


Figure 4.5: **Improved sample combining detail with only two classes. For each class, dampened samples are created. Highlighted are the sample combinations shown in Figure 4.3. Combinations where mopeds are louder than gunshots are now additionally created, as well as single-label combinations of two different volume gunshot samples**

4.2 Improved sample combining

The most straightforward way to improve the data augmentation technique mentioned in section 4.1, is to allow the omitted combinations mentioned in the same section. This is illustrated in Figure 4.5. First the samples on the diagonal, which are single-class samples, can be combined samples as well; Takahashi et al. [26] proposed that two same-class samples can also be combined to create a new sample, thus balancing the classes better. Second, the samples above the diagonal are omitted in Serval's data augmentation. This causes the augmented dataset to remain imbalanced. This also can be improved to yield a more balanced augmented dataset.

Concretely this means that while Serval would not generate samples from gunshots -12dB and mopeds -6dB, because here the moped samples are louder than the gunshots samples, this improved method fixed this problem and also generates samples of gunshots -12dB combined with mopeds -6dB. Applying this change yields us the combinations table shown in Table 4.4. This method

	1	2	3	4	5	6	7	8	9
Gunshots (1)	750	750	750	750	750	750	750	750	750
Mopeds (2)	750	750	750	750	750	750	750	750	750
People noise (3)	750	750	750	750	750	750	750	750	750
Terrace noise (4)	750	750	750	750	750	750	750	750	750
Car horns (5)	750	750	750	750	750	750	750	750	750
Slamming car doors (6)	750	750	750	750	750	750	750	750	750
Amplified music (7)	750	750	750	750	750	750	750	750	750
Motors (8)	750	750	750	750	750	750	750	750	750
Moped alarms (9)	750	750	750	750	750	750	750	750	750
Total primary	6750	6750	6750	6750	6750	6750	6750	6750	6750
Total	12750	12750	12750	12750	12750	12750	12750	12750	12750

Table 4.4: **The amount of samples per class combination after improved sample combination and the total amount of samples per class.**

will be referred to as Improved cross-label sample combining.

Because this effectively means we get over 2 times as many samples, the amount of samples per possible combination is halved from 500 to 250 to get approximately the same total amount of samples as with Serval cross-label sample combining.

A second change is the shift from a 50%/50% train/test split to a 80%/20% train/test split, as illustrated in Figure 4.4. This is a more standard balance between the train and test sets, and it should allow of a better model at the cost of less reliable accuracy metrics. Because the Serval training code does not use a validation set and because of the size of the dataset, I opted to do a train/test split just like the Serval cross-label sample combining technique. An important difference however is that the train/test split is done before any dampening. This ensures that a sample does not have similar (dampened or original) samples in the other set, and that the test samples are unmodified instead of potentially dampened.

As shown in the new table, all classes now have an equal amount of primary-labelled samples. This eliminates the possible imbalance due to classes lacking 'easier' samples. Furthermore, this modified data augmentation technique now also generates single-label combined samples (samples gained by combining two samples with the same class), while Serval only generates multi-label combined samples (samples gained by combining two samples with different classes). This eliminates the class imbalance in single-class samples.

4.3 Noise sample augmentation

The most basic rebalancing technique is to simply train with samples from underrepresented classes multiple times. This rebalances the dataset but could introduce major overfitting in the underrepresented classes. One way to avoid overfitting is to add a layer of noise to the samples as proposed by Salamon and Bello [22]. Depending on the level of noise, this may differentiate the samples sufficiently to eliminate the overfitting. While Salamon and Bello use background noises such as street and park ambient noise, I opted to use generated pink noise, which, according to Szendro et al. can be regarded as the most natural type of noise[18]. It is also approximately the sound generated by wa-

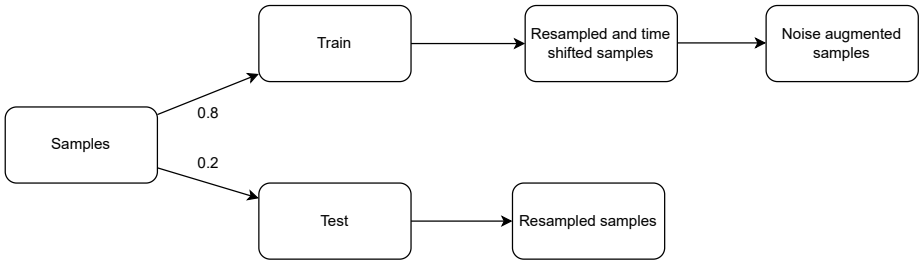
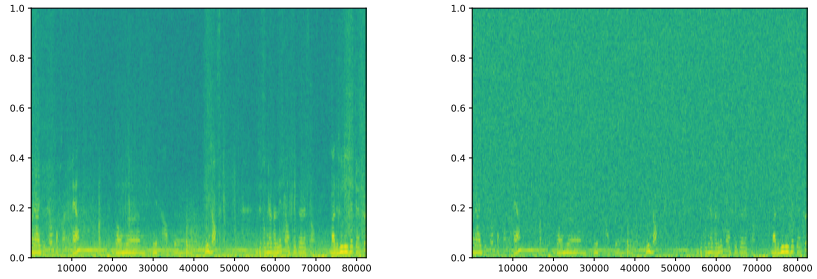


Figure 4.6: **Noise sample augmentation. A layer of pink noise is added to the resampled and time shifted samples. This only happens to train samples. Because no combinations are being made, the dampening is not necessary and is skipped.**



(a) **Without noise** (b) **With noise**

Figure 4.7: **Spectrogram of a sample before and after adding noise**

terfalls. As shown in Figure 4.6, no dampening is done as these were necessary for the combinations.

The main advantage of using this technique is that it is fast and easy to implement. The samples are also easier to train on and are more similar to the samples in the test set. This should speed up training compared to the methods mentioned earlier. Additionally training the model to cope with noise should help it to recognize sounds in noisier environments and in harsher conditions such as during rain or storm. It could possibly even compensate for lower quality hardware.

However one should wonder if adding noise to the samples provides enough differentiation to avoid overfitting completely. When added to a waveform, noise seems to alter it dramatically, but if we look at the associated spectrograms, it only looks 'brighter' compared to the original sample–with details being washed out as a result. This is illustrated in Figure 4.7. This may compromise the quality of using this data augmentation type.

Finally this method does not generate multi-label samples and thus will not help in training a model to recognize multiple classes at the same time.

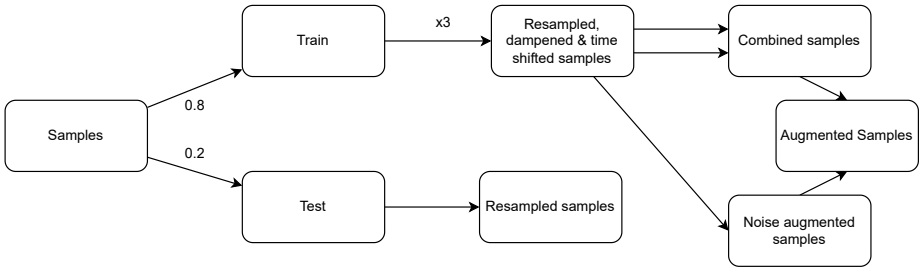


Figure 4.8: **Hybrid data augmentation. The augmented datasets obtained through improved sample combining and noise sample augmentation are combined into a new augmented dataset.**

	1	2	3	4	5	6	7	8	9
Gunshots (1)	1451	750	750	750	750	750	750	750	750
Mopeds (2)	750	1451	750	750	750	750	750	750	750
People noise (3)	750	750	1451	750	750	750	750	750	750
Terrace noise (4)	750	750	750	1451	750	750	750	750	750
Car horns (5)	750	750	750	750	1451	750	750	750	750
Slamming car doors (6)	750	750	750	750	750	1451	750	750	750
Amplified music (7)	750	750	750	750	750	750	1451	750	750
Motors (8)	750	750	750	750	750	750	750	1451	750
Moped alarms (9)	750	750	750	750	750	750	750	750	1451
Total primary	7451	7451	7451	7451	7451	7451	7451	7451	7451
Total	13451	13451	13451	13451	13451	13451	13451	13451	13451

Table 4.5: **The amount of samples per class combination after hybrid data augmentation and the total amount of samples per class.**

4.4 Hybrid data augmentation

With different data augmentation techniques explored in the previous sections, we see that each technique has their own advantages and disadvantages. The best way to harness the advantages of both the improved cross-label sample combination in section 4.2 and noise augmentation section 4.3 is to combine the augmented datasets to get a new, more varied augmented dataset. As illustrated in Figure 4.8, both the combined samples and noise augmented samples are generated separately, and then joined in a new dataset.

This hybrid data augmentation technique should benefit from the advantages of both techniques mentioned. The improved sample combining ensures a model is prepared for multi-label input, and the noise sample augmentation creates balanced single-label samples. As shown in Table 4.5, the input dataset is almost perfectly balanced. It consists of single-label uncombined samples (original samples with pink noise added), single-label combined samples (samples gained by combining two samples with the same class) and multi-label combined samples (samples gained by combining two samples with different classes).

Model	DA technique	Training samples	DA duration
None	None	2734	0:00:24
Naive	Naive rebalance	6309	0:00:38
Serval	Serval cross-label sample combining	49898	3:01:53
Improved	Improved sample combining	60750	0:03:50
Noise	Noise sample augmentation	6310	0:01:35
Hybrid	Hybrid data augmentation	67060	0:04:43

Table 4.6: **All trained models with the amount of training data and the time taken by data augmentation. Models *None* and *Naive* are baseline models. *Serval* is trained on Serval cross-label sample combining as mentioned in section 4.1, *Improved* is trained with Improved sample combining from section 4.2, *Noise* was trained with the Noise sample augmentation described in section 4.3, and finally *Hybrid* was trained on Hybrid data augmentation as mentioned in section 4.4**

4.5 Evaluation

To test our hypotheses, all mentioned techniques were used to train a model. Additionally, as a baseline, a model was trained without any data augmentation, and a "naive rebalance" technique was used, wherein the samples are rebalanced without sample modification, thus serving identical samples multiple times. All techniques and their respective time taken on data augmentation and training can be found in Table 4.6.

From the table it can be derived that all techniques using sample combination take less time per sample than techniques that do not employ sample combination. Furthermore, we can see that the "Serval cross-label sample combining" technique takes dramatically longer than all other methods. This can be explained partially by implementation details and the choice of library: as shown in Figure 4.9, the Serval code for resampling files is dramatically slower compared to the new methods, taking around 60 times as long. Serval uses the Librosa[17] Python library for loading samples, the resampy library for resampling samples, and the soundfile library - based on libsndfile - for writing samples. The new methods on the other hand use the scipy[27] library for both loading and writing files, and the audioop library for resampling. Finally, the new methods are optimized to not do any resampling when the sample rate is already the target rate, however in the making of Figure 4.9, it was ensured that resampling is happening.

All models are trained using the Serval training code. This is a good baseline as other parts of Serval will be looked at in other chapters–if those parts are changed as well we cannot compare just the data augmentation results anymore. The model trained is a default Serval model with 10 long short-term memory (LSTM) cells.

Unfortunately Serval does not do any validation, only testing. This makes it impossible to know during training whether a model is overfitting or underfitting or if it is training just right. This can be estimated after training by comparing the model predictions on both the validation and the training sets. Furthermore Serval does not report training accuracy either, making it even harder to rate how well a model in training is performing.

Finally, some augmentation parameters still need to be decided. Serval's ori-

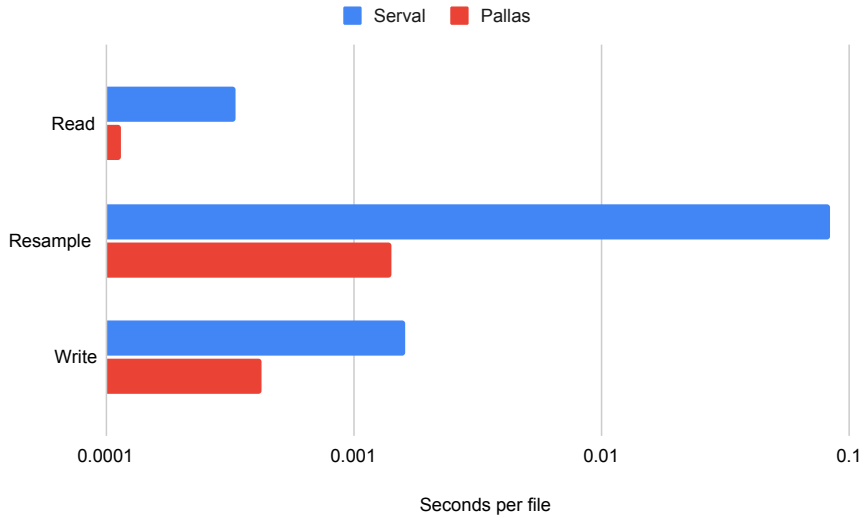


Figure 4.9: **Time taken for read, resample, and write operations for Serval cross-label sample combining and for all other data augmentation methods**

ginal parameters will not be changed, so in Serval cross-label sample combining, samples are resampled in 16000 Hz as required by VGGish, and dampened by 0dB, -6dB, and -12dB compared to the original samples. For each valid combination, up to 500 samples are created–sometimes the same random samples are picked and a new sample is not created, and some classes feature so few samples that 500 unique combinations cannot be created.

Improved sample combining uses the same resampling and dampening parameters, but creates only 250 samples per valid combination. This is because twice as many combinations are considered valid, so this results in roughly the same total number of samples.

In Noise sample augmentation, samples are also resampled to 16000 Hz, but no dampening takes place. Instead, pink noise is added to the sample with a signal-to-noise ratio of a random value between 40 and 60. For each class 701 new samples are created–the maximum amount of training samples in a class.

Finally in Hybrid data augmentation, the data sets generated by Improved sample combining and Noise sample augmentation are combined together in one dataset. The parameters used here are the same as mentioned above in the respective paragraphs.

Now, the accuracy of the models needs to be measured. As mentioned before, Serval does not test during its training loop, so the test samples will be used for evaluation. Recall that the goal of the network is not only to predict classes in single-label samples, and we have only single-label samples in the evaluation set, as no data augmentation has been run on these samples. In order to still measure multi-label capabilities of the networks, a second validation set is generated in a similar fashion to the data augmentation in section 4.2, however, no time-shift is done and the volumes of the samples are unchanged.

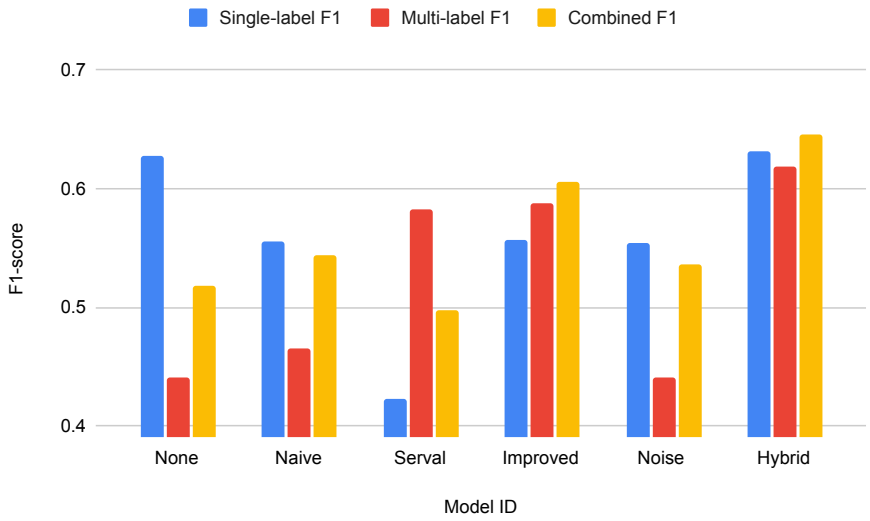


Figure 4.10: **F_1 scores for each model and each validation set**

For each of the 36 class combinations, a number of multi-label test sample are generated so that the total amount of multi-label samples matches the amount of single-label samples. For the new methods this means 19 samples are generated for each combination, while for Serval 146 samples are generated. For Serval, more test samples are generated as Serval has more test samples due to the early dampening and the 50%/50%/ train/test split, however this also means that these test samples are modified and not necessarily unique: some of them may have been used for training.

For each model and each validation set, we now predict the labels, and we take the average of the F_1 scores of all classes – the harmonic mean of precision and recall as shown in Equation 4.1. This is a good metric for imbalanced datasets. Furthermore we measure the individual F_1 scores for each class.

$$F_1 = \frac{2PR}{P+R} \tag{4.1}$$

4.6 Results

In Figure 4.10 we can see the results of the experiments mentioned in section 4.5. In terms of single-label F_1 scores, the model trained with hybrid data augmentation seems to work best, closely followed by the model trained without any data augmentation. One would expect the naive and noise data augmentations to show improved single-label F_1 scores as well however this does not seem to happen. Interestingly, the naive data augmentation does improve the multi-label F_1 score compared to doing no augmentation. This does make sense however if we consider that the single-label test dataset is as imbalanced as the original dataset, while the multi-label dataset is perfectly balanced. This causes the single-label F_1 score to decrease when the model is more balanced

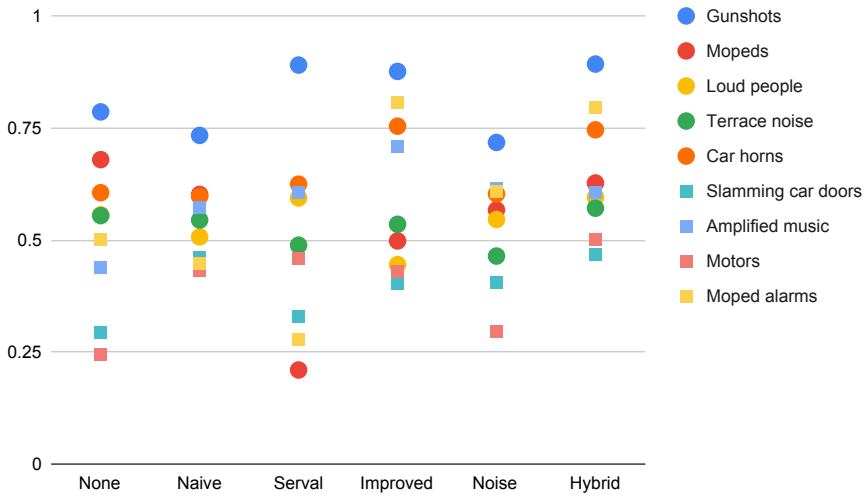


Figure 4.11: **F_1 scores for each model per sample with the combined single and multi-label test dataset**

– causing over represented classes to achieve a lower F_1 and under represented classes to achieve a higher F_1. In the (balanced) multi-label test dataset, the F_1 may increase depending on how much the under represented classes accuracy increase.

Furthermore we see that Serval's data augmentation apparently does not do well with single-label samples, but scores a lot higher with multi-label samples. This can be attributed to the heavy focus of the data augmenatation on the combinations. This result causes the F_1 score for the combined dataset to be rather low as well. The same could be expected from the improved sample combining, but this technique scores much higher on single-label samples and comparable on multi-label samples. I cannot explain why this happens.

Finally, when looking at all scores overall, the hybrid data augmentation appears to yield the best scores over all metrics. This is remarkable as the noise data augmentation by itself yields rather low results, but this combination shows no weakness whatsoever. This underscores the hypotheses stated in section 4.4.

Next, Figure 4.11 shows the F_1 scores achieved by each model with the combined test dataset. It shows clearly that indeed the naive data augmentation is much more balanced compared to doing no data augmentation, however the gain of using noise augmentation is much more subtle. Furthermore Serval appears to somehow only increase the imbalance between classes. Finally the improved sample combining data augmentation scores better on most classes compared to doing no data augmentation and the Serval data augmentation, and the hybrid data mostly improves upon this with the lower-accuracy classes. Only amplified music appears to be much harder to recognize compared to other methods. This could partly be blamed by this being a rather messy class, with lots of samples featuring speech and car noises that are not labelled as such.

Chapter 5

Embedding

Even with an augmented dataset, training a model on a complex task such as acoustic event detection requires a large network and a lot of processing power. This type of training may take multiple hundreds of hours using multiple high-end GPUs[13, 10]–both of which I don't have available.

One solution is to avoid training the convolution layers of the network by using existing weights from an existing model. This is essentially what an embedding is: we use the convolution layers of a deep convolutional neural network (DCNN) to extract the features, which facilitate the final decision making process of the model. In this chapter we look at two embeddings that can be used to train and run a neural network in much more reasonable time and with much less expensive hardware.

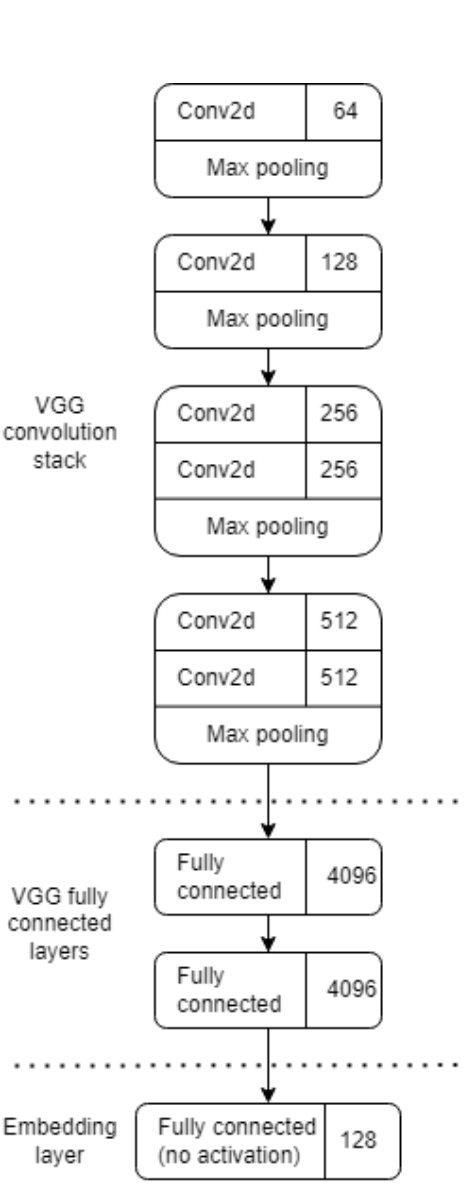
5.1 VGGish

As explained in section 2.1, Serval uses the embedding VGGish to facilitate its training and execution. VGGish is an embedding developed by researchers at Google[13], based on the image recognition network VGG[24]. This model was obtained by training a DCNN based on VGG with log-mel spectrograms derived from audio samples.

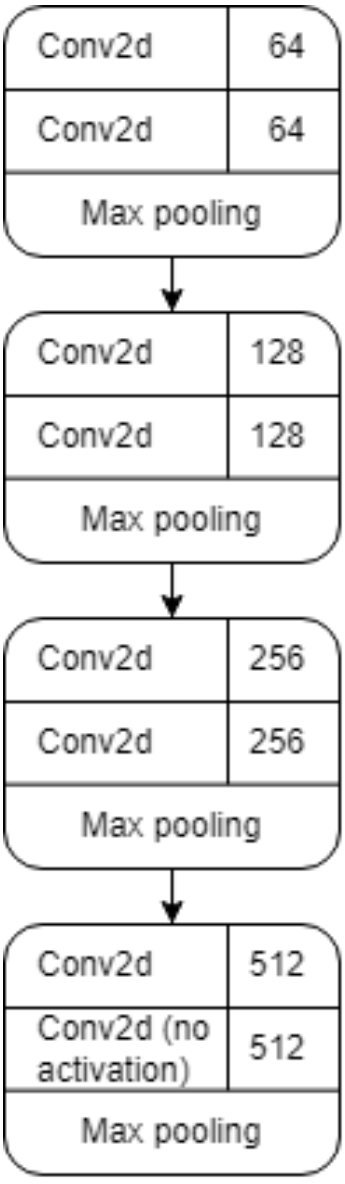
VGGish is a DCNN trained using supervised training on the Youtube-8M dataset. Just like VGG it uses 6 sets of a series of convolutional layers and max pooling layers, as shown in Figure 5.1a. Next are three fully connected layers and finally a soft-max layer. The model produces an embedding of 128 features.

In Figure 5.2 we can see what the data used and generated by VGGish looks like. Due to the low sample rate, we see in Figure 5.2a that the highest frequency visible in the spectrogram is only 8 KHz. Compare this to the nominal maximum audible frequency in humans: 20 KHz. In Figure 5.2b one of the 10 spectrogram windows created by VGGish' preprocessing is shown. Note that this is not a linear spectrogram but a mel spectrogram window. Mel spectrograms are further explained in section 5.2. These mel-spectrogram windows are then used as input for the actual VGGish model, resulting in the embedding shown in Figure 5.4c. For each window, 128 features are generated as shown in the figure.

The main advantage of using VGGish is that it is relatively small, compared

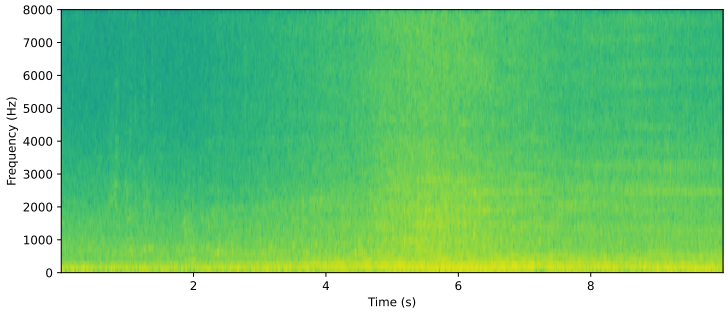


(a) **VGGish network. Derived from the source code, which does not describe the same model as the paper**

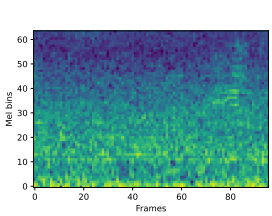


(b) **OpenL3 network**

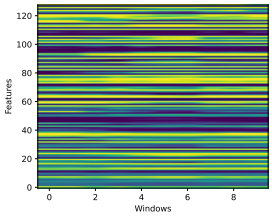
Figure 5.1: **Diagrams of the VGGish and OpenL3 networks. If no activation function is mentioned, a ReLU (Rectified Linear Unit) is used**



(a) **A complete spectrogram of a 16 KHz sample**



(b) **One of the 10 spectrogram windows that VGGish uses**



(c) **A complete VGGish embedding of a sample**

Figure 5.2: **Samples of the data that VGGish generates and (internally) uses**

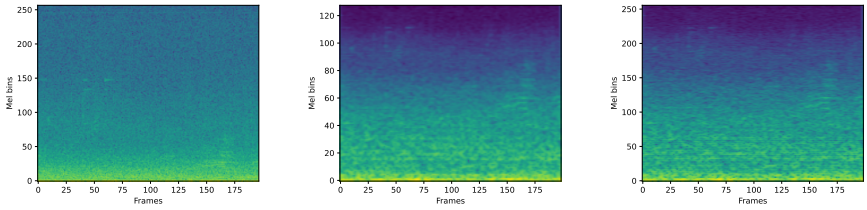
Option	Possible values		
Input representation	**linear**	**mel128**	**mel256**
Training content type	**music**	**environmental**	
Frontend	**librosa**	kapre	
Embedding size	**512**	6144	

Table 5.1: **Options to configure OpenL3 and their possible values**

to alternatives, and thus easier to run on simple hardware, such as a raspberry pi. It is also the embedding employed by Serval already, so we know that it is able to run on the hardware. However as Serval's accuracy is insufficient, changing the embedding to a more modern variant could improve the model accuracy.

5.2 OpenL3

An alternative proposed by Sensemakers is an embedding called OpenL3[10]. OpenL3 is a research based on L^3-net[3], evaluating the impact of a number of design choices and alternatives, such as the choice of input representation and training domain. An overview of these options can be found in Table 5.1. An eponymous library was released allowing these alternatives to be set as configurations. OpenL3 should be able to boost the accuracy by 4 to 6 percent point[3] compared to VGGish, depending on the dataset and task difficulty.



(a) **linear spectrogram** (b) **mel128 spectrogram** (c) **mel256 spectrogram**

Figure 5.3: **Spectrograms of the different types of input representations**

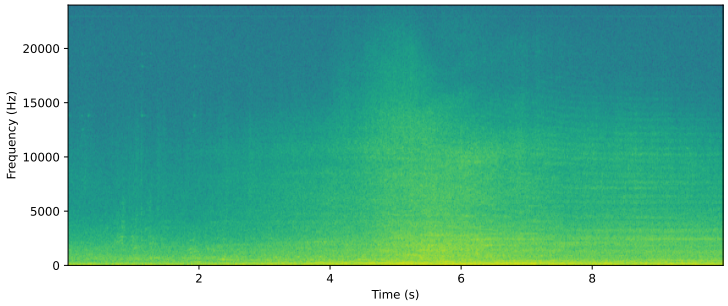
L^3-net was trained by exploiting the natural Audio-Visual Correspondence (AVC) that can be observed in most video's: Because the audio and the visuals are produced by the same source, thus events are often observed at the same moment in both data types. For example an explosion in a video normally is accompanied by both a bright flash in the visuals, and a loud noise at roughly the same time in the audio. Arandjelovic et al. trained a model to determine whether an audio and a muted video fragment are from the same video fragment.

The OpenL3 model itself is a DCNN, much like the VGGish model. Figure 5.1 shows the VGGish and OpenL3 models next to eachother. As shown in Figure 5.1b, the OpenL3 model is much simpler compared to the VGGish network.

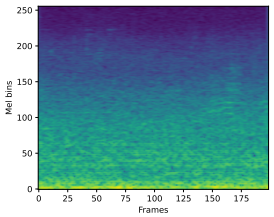
With the input representation (`input_rep`) one can choose how the input spectrogram has been formatted. The `linear` spectrogram–the input used by the original L^3-net–is a basic spectrogram with both a linear time and a linear frequency domain. The `mel128` and `mel256` options represent a Mel-frequency log-magnitude spectrogram with respectively 128 and 256 Mel bands. This type of spectrogram was designed[25] to better and more efficiently relay the frequencies that are most audible to humans. While it makes sense that using this spectrogram improves the performance of speech recognition or models designed for music, it remains to be seen if this applies to our urban environment as well. In Figure 5.3 we can see a spectrogram of each input representation.

A second option that was evaluated was the domain of the training dataset (`content_type`). The original L^3-net was trained on a musical dataset under the assumption that musical videos are easier to train on through Audio-Visual Correspondence. Cramer et al.[10] hypothesized that using an environmental dataset should improve the model performance, considering the downstream task is focused on environmental sounds as well. While the choice of training domain didn't seem to positively impact the model performance in their case, this may be different in our specific dataset and use-case.

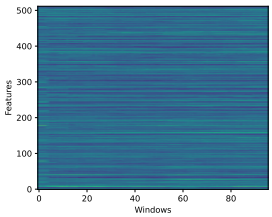
It is also possible to change which library OpenL3 uses to do the preprocessing. One can choose between the librosa library[17]–a python package for audio analysis–and Kapre (Keras Audio Preprocessors)[8], a library that provides certain often used audio processing functions in the form of Keras layers. This includes the STFT and Melspectrogram functions. As these two methods produce functionally equal spectrograms, one should not expect any difference in performance from this choice, though a difference in timing could be noticeable. As Librosa is easier to use and does not require TensorFlow 2 to run, it is used throughout the rest of the thesis.



(a) **A complete spectrogram of a 48 KHz sample**



(b) **One of the 96 spectrogram windows that Openl3 uses**



(c) **A complete Openl3 embedding of a sample**

Figure 5.4: **Samples of the data that Openl3 generates and (internally) uses. The OpenL3 parameters used are a mel256 input representation, environmental content type, librosa frontend and an embedding size of 512 features. The same audio sample was used as for Figure 5.2.**

Finally OpenL3 allows an alternative embedding size. L^3-net uses a $32 * 24$ pooling layer as its last layer, producing an embedding of 512 dimensions–4 times as many features as a VGGish embedding. OpenL3 allows the use of an alternative $8 * 8$ pooling layer instead, producing a 6144-D embedding, which is 48 times as many features as a VGGish embedding. While this allows for a much more detailed feature set, it also dramatically increases the size of the embedded dataset and of the downstream model. Preliminary tests shows that the training of a VGGish model takes about 2 hours and of a 512-D OpenL3 model around 6 hours, so it is expected that a 6144-D training session would take 96 hours–4 days–in total. Furthermore, with the large size of the 512-D embeddings on the disk in mind (See Table 5.2), the 6144-D embeddings would take over 4 TB to store, not even taking into account the extra experiment(s) needed to test these embeddings. For these reasons, this option is not evaluated.

The method of using OpenL3 and VGGish is essentially the same, however the model and preprocessing parameters differ a lot. Figure 5.4 shows the data used and generated by OpenL3, in a similar fashion to Figure 5.2. OpenL3 and L^3-net are designed to use 48 KHz samples, compare this to the 16 KHz samples used by VGGish. This means that a higher sound fidelity is achieved along with a 4 times as high maximum frequency, as shown in Figure 5.4a. From Figure 5.4b it is clear that OpenL3's preprocessing creates 4 times as many frequency bins as VGGish does, although this does depend on the input representation chosen.

Finally running the OpenL3 model with these spectrogram windows as input produces the embedding. As shown in Figure 5.4c, 512 features are produced for all of the 96 spectrogram windows. Compare this to the VGGish embedding which consists of 128 features over 10 windows: OpenL3 produces almost 40 times as much data for training or testing on.

Using OpenL3 has multiple obvious advantages. First of all, as it is a larger and more recent embedding compared to VGGish, it allows for a more precise output and thus a more accurate network. Remember that depending on the embedding size used, OpenL3 produces 40 to 1920 times as many features as VGGish does, allowing for higher detail for the downstream model. Additionally, the configurable options shown in Table 5.1 allow it to be configured such that it fits this domain best.

The source of its main advantage may also be its disadvantage: it is larger and heavier than VGGish. While not a big problem in itself, it may prove to be hard or impossible to run on the Raspberry Pi. The larger amount of features also increases the size of the downstream model dramatically, increasing memory usage, as well as training and prediction times. This problem will be addressed in chapter 6.

5.3 Pallas

Testing the OpenL3 embeddings will be not done with Serval, as it seems to fail to train on OpenL3 embeddings: no convergence was witnessed. To this end I present Pallas. Pallas is a neural network much like Serval, albeit depending on TensorFlow 2 and allowing for more configuration and different embeddings.

The Pallas model is similar to the Serval model as described in section 2.1: it features a long short-term memory (LSTM) layer as its first layer. However recall that while Serval uses a MOE model as its following layer, Pallas uses a simple fully connected layer.

Remember from section 4.5 that Serval does not do validation. Pallas is able to do validation, however because of the small amount of samples for some of the classes in the dataset, it was chosen not to use a separate validation set. Instead, Pallas uses its test set to get its test accuracy after every epoch. When the train accuracy is still increasing but the test accuracy starts to decrease, the model is considered to be overfitting. At this stage, the training will be aborted to avoid overfitting. Thus no hyperparameter tuning is done with this test set.

5.4 Evaluation

To compare the embeddings, the dataset obtained with Combined data augmentation as described in section 4.4 is embedded with each chosen configuration. These embeddings are then used to train a long short-term memory (LSTM) network. Two VGGish embeddings are created as well as a baseline. One VGGish embedding is created using the Serval codebase, and the second one using the Pallas codebase. These baselines allow for determining the difference in performance between Serval and Pallas and between VGGish and OpenL3. In the case of an OpenL3 embedding, a Pallas network is trained instead. This is done because the OpenL3 embedding does not seem to work properly with the

Serval code, producing garbage networks with very low ($\sim 30\%$) accuracy, while the Pallas codebase is able to train proper networks with both the VGGish and the OpenL3 embedding. All OpenL3 configuration combinations mentioned in section 5.2 and displayed in bold in Table 5.1 are tested, along with a Serval network trained using the VGGish embedding, which should yield the same results as in chapter 4 and a Pallas network trained using the VGGish embedding. All tested embeddings and their configurations can be found in Table 5.2.

Because OpenL3 embeddings are many times larger compared to VGGish embeddings, the downstream model and its training parameters have to be scaled to be able to use this abundance of data. Both the Serval and the Pallas model contain an LSTM layer as the first layer. The VGGish-based models contain 1 layer of 10 LSTM cells, which is Serval's default setting. In contrast the OpenL3-based models contain 1 layer of 100 LSTM cells. Furthermore testing proved that using OpenL3 instead of VGGish requires changing the learning rate as well: the OpenL3-based models use a learning rate of 0.0001 instead of Serval's default learning rate of 0.001.

In order to compare the different embeddings, we first need to define the metrics used to compare them. The first metric is the most obvious one–like in section 4.5, the average of the F_1 scores per class. Also the time taken by each step in the prediction pipeline is measured: the loading time (time taken by reading samples), preprocessing time, embedding time, predicting time, and saving time (time taken by saving the predicted logits on the drive).

Furthermore we are interested in the size and speed difference between the two embeddings. To gain an insight in this the total file size of the produced embeddings, and the time taken to produce them is reported. Finally the difference in size and speed between Serval and Pallas, and likewise between VGGish-based and OpenL3-based models is reported, as well as the speed difference in predicting with these models. These data can be read from the file system as well.

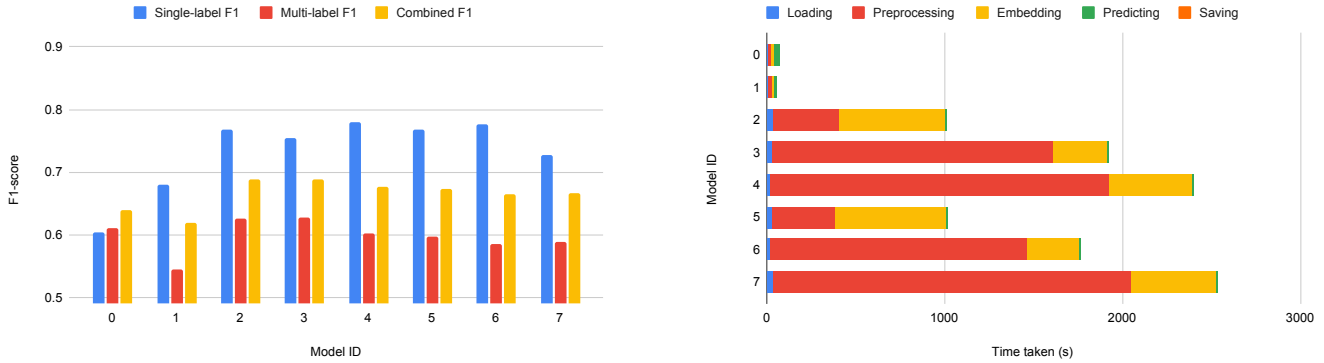
To test the performance of the resulting networks, all networks are used to predict the labels on identical test sets. The testing code and the test sets used are identical to the code and samples mentioned in section 4.5.

We expect the following results in the next section:

1. All OpenL3-based models perform better than the VGGish-based models.
2. The OpenL3 embeddings are around 40 times as massive and take around 40 times as long to embed compared to the VGGish embeddings.
3. The same size and speed difference can be observed in both predicting with and training the OpenL3-based and VGGish-based networks.
4. Networks trained with an environmental embedding perform similar or worse compared to networks trained with a musical embedding.
5. Networks trained with a mel embedding perform better than networks trained with a linear embedding.

Embedding ID	Codebase	Embedding	OpenL3 configuration		Embedding		Training		Testing
			Input rep	Content type	Size	Duration	Size	Duration	Duration
0	Serval	VGGish			96.4 MB	5:03:56	1.44 MB	4:19:28	0:00:37
1	Pallas	VGGish			342.4 MB	1:22:24	793 KB	7:20:22	0:00:32
2	Pallas	OpenL3	Linear	Music	14.0 GB	18:43:14	3.5 MB	6:18:12	0:13:49
3	Pallas	OpenL3	Mel128	Music	14.0 GB	1:2:48:53	3.5 MB	6:20:31	0:29:15
4	Pallas	OpenL3	Mel256	Music	14.0 GB	1:7:36:46	3.5 MB	6:12:21	0:40:36
5	Pallas	OpenL3	Linear	Env	14.0 GB	11:22:47	3.5 MB	6:05:18	0:15:31
6	Pallas	OpenL3	Mel128	Env	14.0 GB	1:3:35:36	3.5 MB	6:11:27	0:38:46
7	Pallas	OpenL3	Mel256	Env	14.0 GB	1:7:00:52	3.5 MB	6:13:42	0:56:02

Table 5.2: **Tested embeddings and their configurations**



(a) **F_1 metrics for each embedding on each test set. The model ID's refer to the same model ID's found in Table 5.2**

(b) **Time taken by each model in each step in the prediction timeline. The model ID's refer to the same model ID's found in Table 5.2**

Figure 5.5: **F_1 and timing metrics for tested embeddings**

5.5 Results

The embedding metrics in Table 5.2 clearly show that OpenL3 is a much larger embedding. At 14 GB's the embedding is much larger than VGGish' 342 MB's. This means the OpenL3 embeddings are over 40 times as large as the VGGish embeddings. This seems to align with the expected difference. The difference between Serval's VGGish size and Pallas' VGGish size can be explained by VGGish quantizing the data to 8-bit integers while OpenL3 does not do this.

Creating an OpenL3 embedding also takes a lot longer compared to a VGGish embedding, and the input representation used seems to have a large impact on the duration of the embedding as well. This shows that converting the spectrograms to mel spectrograms requires a lot more computation time than running the OpenL3 model itself.

The size of the OpenL3-based models appears to be about 4.5 times larger compared to the VGGish-based models. This can be explained by the OpenL3 embeddings having an embedding size 4 times as large as the VGGish embeddings. LSTM models are designed to have a list as input so the 9.6 times increase in the time dimension is not reflected in the model size. However do note that the amount of LSTM cells is also 10 times as large in the OpenL3-based models compared to the VGGish-based models. I cannot explain why this change does not seem to impact the model size any further, nor why the OpenL3-based models take less time to train compared to the VGGish-based models.

Figure 5.5a shows the embedding metrics such as embedding duration and embedding size. All OpenL3 models outscore the VGGish results in single-label and combined F_1-scores, but the difference between the embeddings at the multi-label F_1-score is less clear. When looking at which OpenL3-based model performs best, the model trained on embedding number 2 – Pallas with an OpenL3, music-based embedding with a linear spectrogram as input – has the highest combined F_1-score, which is the most important metric given that any sample may contain any amount of labels.

It is interesting to note that the embeddings trained on a musical dataset fare better than the embeddings trained on an environmental dataset, even if the downstream task is an environmental one. Also interesting is the fact that the embeddings using a mel-spectrogram as its input seem to fare slightly worse compared to their linear spectrogram counterparts.

The testing duration appears to be heavily dependent on the embedding used and the chosen embedding parameters. This is evident in Figure 5.5b. From this graph two things are immediately evident: firstly it is obvious that the VGGish embedding is magnitudes faster to compute. Both the preprocessing and the embedding itself take much longer when using the OpenL3 embedding. Secondly preprocessing and embedding time in OpenL3 heavily depend on the embedding parameters used. Specifically using a mel-spectrogram as input type may multiply the time taken by the preprocessing by 4 to 6 times, but reduce the embedding time up to 50%.

Finally when looking at the VGGish-based models, Pallas seems to fare worse on multi-label and better on single-label samples compared to Serval. This can be explained by the difference in model layout compared to the Serval model. Furthermore, Pallas does not do the quantization and the principal component analysis that Serval does, in which a dataset is transformed and scaled based on its means and variances.

Looking back on the hypotheses in section 5.4 we can say the following about each hypothesis:

1. *All OpenL3-based models perform better than the VGGish-based models.*

   This appears to be the case. Looking at Figure 5.5a we can safely say that this hypothesis has been met.

2. *The OpenL3 embeddings are around 40 times as massive and take around 40 times as long to embed compared to the VGGish embeddings.*

   This appears to be true for the embedding size, but not for the embedding duration: embedding duration takes only between 8 and 23 times as long as VGGish. This is probably due to the overhead of running a lot of smaller operations, which is what happens with VGGish.

3. *The same size and speed difference can be observed in both predicting with and training the OpenL3-based and VGGish-based networks.*

   This is partly true. The size difference can be seen in Table 5.2, but the speed difference was not as apparent. This can probably be attributed to tensorflow and memory overhead.

4. *Networks trained with an environmental embedding perform similar or worse compared to networks trained with a musical embedding.*

   Networks trained with an environmental embedding seemed to perform slightly worse compared to networks trained with a musical embedding, so this hypothesis appears to be true.

5. *Networks trained with a mel embedding perform better than networks trained with a linear embedding.*

   Finally this hypothesis appears to be false. Networks trained with an embedding using a mel-spectrogram seem to perform worse when considering final network accuracy and take a lot more time compared to networks trained with an embedding using linear spectrograms. This is interesting as it contradicts the findings by Cramer et al.[10]. This could be due to the dataset's focus on environmental sounds that are often not designed for human listening or alerting, thus these sounds do not benefit from a transformation that highlights human-audible frequency bands. Also, Cotton and Ellis [9] found that mel-frequency features perform very poorly when samples contain a lot of noise. This is of course the case for many samples and classes in our urban-centered dataset.

Chapter 6

Neural Networks in Embedded Systems

When using a neural network in an embedded system, new problems arise. Neural networks may require large amounts of memory and processing capacity. Most embedded systems do not have these specifications as they are designed to be as inexpensive and energy efficient as possible.

One solution to this problem is running as much as possible in the cloud, however this requires sending large amounts of identifiable information to a remote server. This data could be partially anonymized by running the pre-processing on the edge, but with audio networks one may find that the pre-processing takes more time than running the neural network itself. Some solutions have been devised to train and run networks with obfuscated data in the cloud; however, these typically diminish accuracy of the models. More recent solutions with noise injection however produce near-perfect models[12] compared to regular models.

Another solution is to use TensorFlow Lite[1] to run the models on the device instead of full TensorFlow. This library is adapted for use on mobile and edge devices and may prove more feasible to run thanks to its lower memory and computational footprint.

A final solution is to accelerate the embedded system with a Tensor Processing Unit (TPU). This would mean adding a peripheral computation device optimized for working with tensors. Tensors are the building blocks of neural networks and the namesake of TensorFlow. These TPU's are able to compute tensor calculations magnitudes faster and more efficient compared to CPU's and GPU's. The downside of using a TPU is that it requires adding this potentially costly device to each deployed device.

6.1 OpenEars

The hardware used by Serval is called OpenEars[2]. OpenEars is an open source project based on devicehive[3] and is being developed by SensingClues and Sense-

[1] `https://www.tensorflow.org/lite`
[2] `https://github.com/sensemakersamsterdam/OpenEars`
[3] `https://github.com/devicehive/devicehive-audio-analysis`

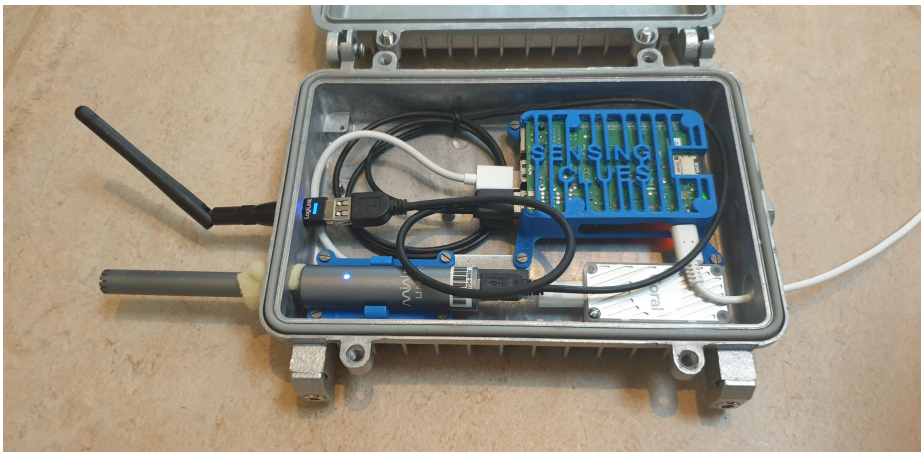


Figure 6.1: **An opened OpenEars device with the Coral TPU added**

makers AMS. Its hardware consists of a Raspberry PI 4b and USB microphone – a UMIK-1[4]. Furthermore, MQTT is used to communicate with a Grafana Dashboard server. A Coral TPU[5] can be connected to improve its tensor computational abilities. An OpenEars device with such a Coral TPU added is shown in Figure 6.1.

6.2 Pallas device code

Pallas uses the same hardware as Serval, however, all device code is rewritten to improve coding standards and to upgrade its Python and Tensorflow versions. As shown in section 6.6, the pre-processing step and the embedding step take roughly the same amount of time. Especially if the embedding is done on a TPU, these two steps can very well be run in parallel. This is why Pallas uses a multi-threaded pipeline to improve prediction speed. A flow chart of this pipeline can be found in Figure 6.2.

From this flowchart is it clear that the Pallas codebase is designed with backwards compatibility and modularity in mind. For the input a file system loading thread can be used, while the microphone thread uses live recorded sound data as its input. Both threads allow setting the target sample rate, and produce a waveform and its sample rate. The loading thread optionally allows retrieving the true label if a true label is given.

Preprocessing for Pallas can be done with the Librosa library and with Kapre. As described in section 5.2, OpenL3 can use both Librosa and Kapre as its front-end. VGGish requires its own specific preprocessing implemented by the VGGish preprocess thread.

As for the embedding, Pallas allows creating VGGish and OpenL3 embeddings. The VGGish embedding can then be used with both Serval and Pallas, while the OpenL3 embedding can only be used with Pallas. These prediction threads then produce a logit (also called log-odds) for each trained class: the

[4] `https://www.minidsp.com/images/documents/Product%20Brief%20-%20Umik.pdf`
[5] `https://coral.ai/static/files/Coral-USB-Accelerator-datasheet.pdf`

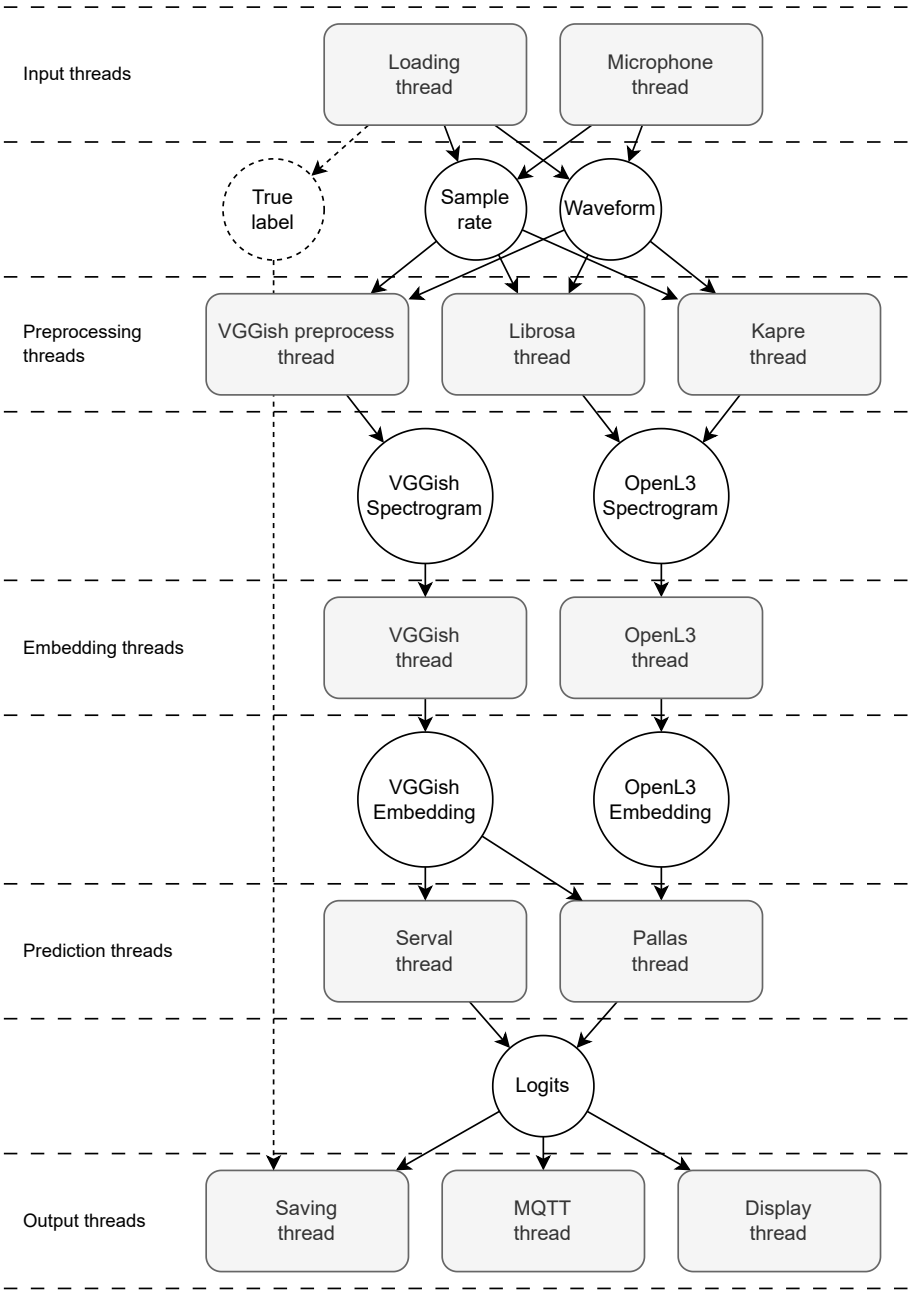


Figure 6.2: **The pipeline pallas uses with the threads (in the boxes) and the data exchanged between threads (in the circles). Note that most threads are interchangable with alternative threads, like the loading thread and the microphone thread.**

logarithm of the odds that a sample contains a given class. These logits are finally used in the output threads. The exact behavior depends on the thread: The saving thread saves the logits and optionally the true labels and system performance statistics to the file system. The MQTT thread sends the logits and some system performance statistics to the configured MQTT host. Finally the display thread calculates the predicted classes using an optimal threshold attained through testing with labelled samples. These predicted classes and some system performance statistics are displayed in a console and/or saved to a log file for offline usage.

6.3 TensorFlow Lite

As mentioned, one option for improving operation speed and power usage is to use TensorFlow Lite models. To do this, the models first need to be converted using the built-in TensorFlow Lite converter. Additionally, this converter allows for a number of optimizations such as quantization, pruning, and clustering. Quantization reduces the precision and size of the model's variables, allowing for a smaller network size and faster computation, while compromising accuracy. Pruning and clustering only reduce model download size with no benefit to computation speed, so these optimizations are left aside.

TensorFlow Lite's quantization has a few configuration options, of which some can be used concurrently. However, due to TensorFlow Lite's complexity and many experimental options, a TensorFlow Lite guide[6] is used to determine some types of quantization that can be tried. The most obvious optimization is to change the datatype to a smaller and simpler datatype. By default TensorFlow models use 32-bit floating point number to store weights and biases. Using the TensorFlow Lite Converter, the variables can be converted to 16-bit floating point, 16-bit integer, and 8-bit integer. Finally an experimental option is present that allows storing activations as 16-bit integers, weights as 8-bit integers and biases as 64-bit integers. This method should improves accuracy compared to other optimizations and only slightly increase the size of the model.

Converting 32-bit floating point values to 16-bit floating point is a trivial operation: nominal values found in our neural networks are well within the range of both 32-bit and 16-bit floats. However this conversion will negatively affect the precision of the model. Converting to integer numbers is not as trivial, as converting floating point numbers to integers in general comes with a penalty in accuracy. To mitigate this by using the full available range of integers, the value is normalized to the full range of the target datatype. In order to do this, TensorFlow determines the range of the variables dynamically using sample inputs. Integer operations are much faster on most hardware, so this conversion should yield a model that is both smaller and faster.

Running a TensorFlow Lite model is straightforward: the model is loaded into a TensorFlow Lite Interpreter. This object manages the tensors and the operation of the model. If the input of the model is quantized, one has to quantize the input before calling the interpreter.

[6] `https://www.tensorflow.org/lite/performance/post_training_quantization`

6.4 Edge TPU

Another option involves using a Tensor Processing Unit (TPU). These devices considerably speed up the operation of a neural network, but before a model can be deployed on one, it needs to be compiled first. Google's Coral TPU offers a compiler for TensorFlow Lite models. Not all TensorFlow Lite models can be compiled for the Coral TPU, as it only supports 8-bit integer operations. In section 6.3 it was described how an 8-bit integer quantized model can be created. This model can then be compiled to yield a Coral-compatible TensorFlow Lite model.

Running such a Coral-compiled model is slightly more cumbersome than running a regular TensorFlow Lite model. First there are some additional software dependencies that must be installed. The compiled TensorFlow Lite model can then be loaded into the interpreter. To ensure this interpreter attempts to run on the Coral TPU, the Coral TPU library is passed to the interpreter as a so-called "delegate". These delegates allow the interpreter to use hardware acceleration through internal or external libraries.

6.5 Evaluation

To find out what optimizations are necessary to run Pallas on the embedded device, we need to measure the performance of all optimization options. As some optimizations may incur a penalty in model accuracy, both the model accuracy and the time and resource performance must be measured.

As shown in section 5.5, running the Pallas model takes a negligible amount of time compared to the preprocessing and embedding steps. As the preprocessing happens in a secondary library, it cannot be accelerated using TensorFlow Lite, so only the OpenL3 model will be converted to TensorFlow Lite.

Running OpenL3 on the Raspberry Pi takes much longer compared to running it on a PC, at almost two minutes per 10-second sample, compared to half a second on a PC. This renders running the full test dataset infeasible, considering it consists of 1363 such samples, bringing the total time taken for running the entire test dataset up to around two days. Thus, running multiple experiments would take too long. To avoid this, a smaller dataset was created consisting of only 54 samples. If an experiment proves to run fast enough, it can be repeated with the full dataset to yield a better comparison with earlier experiments.

All experiments can be seen in Table 6.1. These experiments will be ran and timed. The same timing statistics and F_1-scores as in section 5.4 are produced. All experiments that run fast enough to run in real-time, i.e. that finish in at most 540 seconds will be run again with the complete test dataset to compare the accuracy of all models. These models will then be compared against the accuracy of Serval and the accuracy of Pallas when using an unoptimized OpenL3 model.

Finally for both Serval and Pallas the power consumption is measured. This is done using a BASETech EM-3000[7]. The full setup used can be seen in Figure 6.3. The BASETech EM-3000 has a power consumption of up to 1W and a precision of 2%. This means it is not the highest precision device available for

[7]Datasheet downloaded at 7 February 2024 from `https://asset.conrad.com/media10/add/160267/c1/-/gl/001611632ML01/gebruiksaanwijzing-1611632-basetech-em-3000-energiekostenmeter-kostenprognose.pdf`

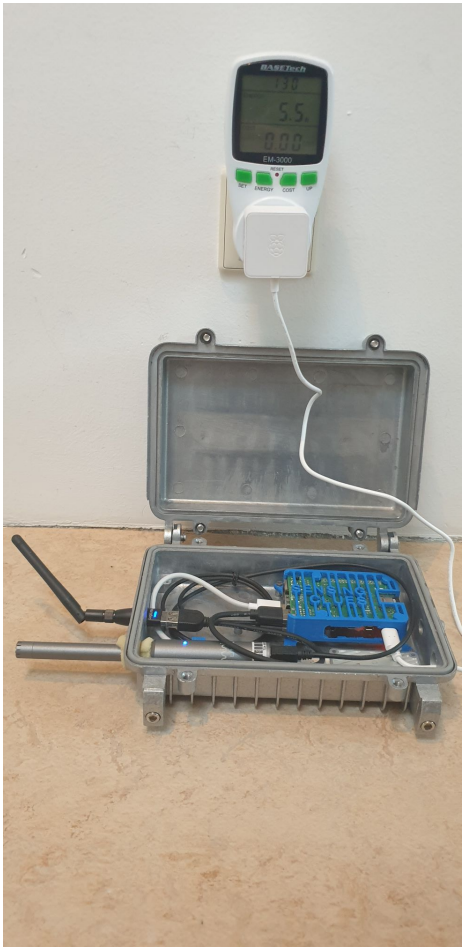


Figure 6.3: **The setup for measuring power consumption. A BASETech EM-3000 is placed between the power outlet and the Raspberry PI power adapter, measuring power usage of the device.**

this scenario. To remedy this, power consumption is measured over at least 24 hours. Because the own power consumption of the BASETech EM-3000 is not clear, it is included in the measured values.

We can formulate the following hypotheses:

1. Integer models are faster than Floating point models.

   We expect that the TensorFlow Lite models using integer variables are faster than the TensorFlow Lite models using floating point variables. Furthermore, the TensorFlow Lite models using floating point variables are expected to be faster than the original model by a small margin.

2. The TPU-compiled model is faster than all other OpenL3 models.

   Models running on the TPU may both be faster and slower compared to their original models running on the CPU. OpenL3 is not the largest

Model ID	Optimization	Model size	Optimization duration
0	None	17.9 MB	N/A
1	TFLite	17.8 MB	4s
2	Dynamic range	4.5 MB	4s
3	Full integer	4.5 MB	3317s
4	16-bit float	8.9 MB	3s
5	16x8 int	4.5 MB	6637s
6	Full integer TPU	4.5 MB	3310s

Table 6.1: **Experiments and their optimization types, compiled model size and time taken by the optimization process itself**

model so we expect the TPU-compiled TensorFlow Lite model to be faster than all other OpenL3 optimizations and the original.

3. The accuracy of TensorFlow Lite optimized models (models 2 to 6) is less than the original model by a small margin.

   The trade-off for faster models is less accurate models, so it is to be expected that the faster a model is, the less accurate it is.

4. TensorFlow Lite optimized models require less computational resources compared to the original model.

   TensorFlow Lite is optimized to use fewer resources compared to the original model. Especially the TPU-compiled model is expected to use less resources as the model will run on the TPU, freeing up resources on the CPU and in the RAM.

5. Running with a TPU requires less power than running without TPU.

   As the TPU-compiled model requires less resources on the CPU, and the TPU requires a fraction of the power a CPU or a GPU requires for running neural networks, it is expected that running a large model on the TPU is much more energy efficient compared to running the same model - or even a smaller model - on the CPU.

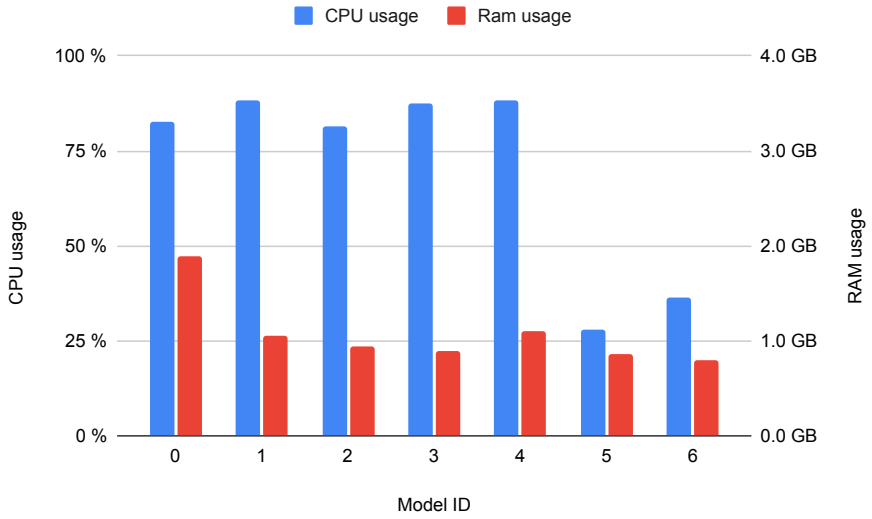


Figure 6.4: **CPU and RAM usage for each model. The CPU usage measured is the average combined core usage, so 100% means all cores have 100% usage. The RAM usage is the maximum RAM usage seen during execution for the relevant python processes. The total amount of RAM available on the OpenEars device is 4 GB.**

6.6 Results

From Figure 6.4 it appears that while most models still show high CPU-usage, RAM usage is reduced in all optimized models. This can partially be explained by the original OpenL3 model requiring the full TensorFlow library, while the optimized models only require the TensorFlow Lite runtime. Also note that the 5th model and the 6th model do not fully use the CPU. In the case of the 6th model this makes sense, as this model mostly runs on the TPU instead of the CPU. However, for the 5th model, this could only be attributed to how TensorFlow Lite runs this model. As the 5th method uses an experimental optimization it may be the case that its execution was not optimized yet, or it was unable to run multi-threaded (its slightly-more-than 25% CPU usage corresponds to a single core of the four available cores being used).

In Figure 6.5 it is clear that single-label performance did not suffer from the optimizations, however the multi-label F_1-scores appear to be drastically reduced for the integer-based models (3, 5 and 6) while the float-based models (0, 1, 2, and 4) do not show any difference. This shows that integer quantization indeed impacts the model accuracy, while using reduced-accuracy floating-point variables barely impacts accuracy at all. Interestingly enough the performance hit only seems to be apparent for single-label samples. Possibly these single-label samples are easier to recognize and thus do not suffer from the integer quantization.

Regarding the timings found in Figure 6.6, it appears that the integer-quantized models (3, 4, and 6) are much faster than the floating point models (0, 1, and 2),

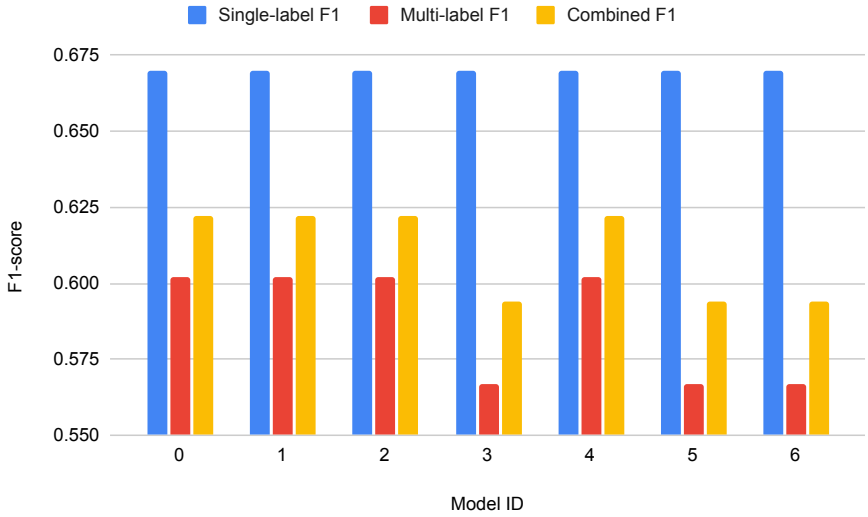


Figure 6.5: **Final metrics for each model on each test set in experiment 1. The model ID's correspond to the same model ID's found in Table 6.1. Do note that these results were made with the slim dataset, so they cannot be compared to other F_1-score results.**

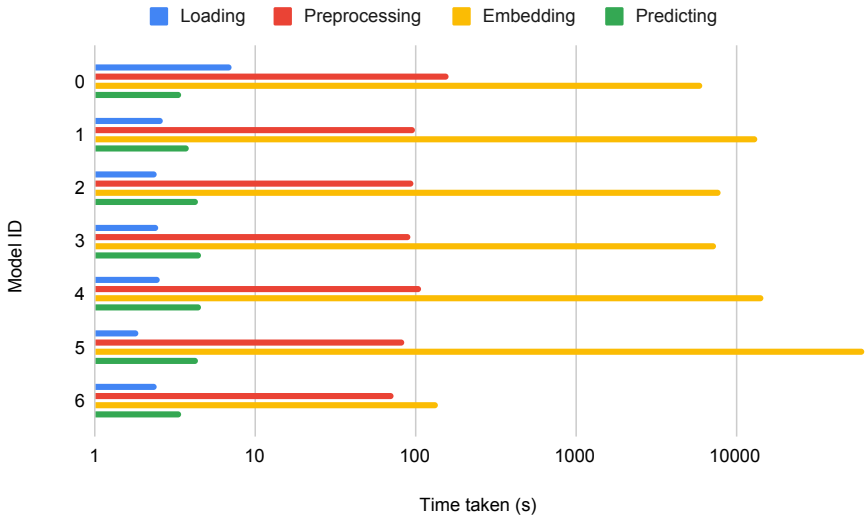


Figure 6.6: **Time taken by each model in each step in the prediction timeline during experiment 1. The model ID's refer to the same model ID's found in Table 6.1**

except for the 16x8 integer model (model 5). Furthermore, the TPU-compiled model (model 6) is much faster than its alternatives and is able to run in real time. This means this is the only model that will actually be tested in the next experiment with the large dataset.

Running the Serval model with VGGish, Pallas with the original OpenL3 model and Pallas with the TPU-compiled OpenL3 model on the full dataset mentioned in section 5.4 produces the results shown in Figure 6.7. As shown in this graph, the TPU-compiled model actually outperforms both the Pallas model with the original OpenL3 and the Serval model with VGGish.

Finally as shown in Figure 6.8, the power usage of the different experiments does not differ much. Serval uses only slightly more power in parallel than it does in serial usage, however in Pallas we see a larger difference. This is probably due to the smaller embedding used by Serval and thus less overhead due to queuing and serialization necessary for parallel execution. Also clear is that Pallas in serial usage uses less power than Serval in serial. This can be attributed to Pallas running its embedding on the optimized TPU device instead of on the CPU.

Looking back at the hypotheses formulated in section 6.5 we can conclude the following about them:

1. Integer models are faster than Floating point models.

   From Figure 6.5 we can conclude that indeed most integer models are faster than floating point models. Only the 16x8 integer model fails to run in reasonable time. Considering that this is an experimental optimization, we can conclude that it was probably not designed yet to run efficiently.

2. The TPU-compiled OpenL3 model is faster than all other OpenL3 variants.

   This model indeed outperforms all other variants based on OpenL3. Only the Serval model, which uses VGGish, is still a lot faster.

3. The accuracy of TensorFlow Lite optimized models (models 2 to 6) is less than the original model by a small margin.

   In Figure 6.5 it is shown that the models using TensorFlow Lite's integer quantization indeed score lower F_1-scores compared to their floating point quantization counterparts, however in Figure 6.7 this dip in accuracy cannot be seen. On the contrary, the TPU-compiled OpenL3 model appears to score a slightly higher F_1 score compared to the original OpenL3 model.

   This difference could be explained by the size of the smaller dataset: in such a small set, artifacts are more probable due to the random selection of samples from the full set, the difference shows that the selection may happen to contain mostly more difficult samples.

4. TensorFlow Lite optimized models require less computational resources compared to the original model.

   As shown in Figure 6.4, the integer quantized models are much faster and use less ram compared to the float-quantized models. However, the 5th model is an exception to this: It appears that it was unable to fully use the computational resources the device has to offer, causing it to be much slower compared to all other models.

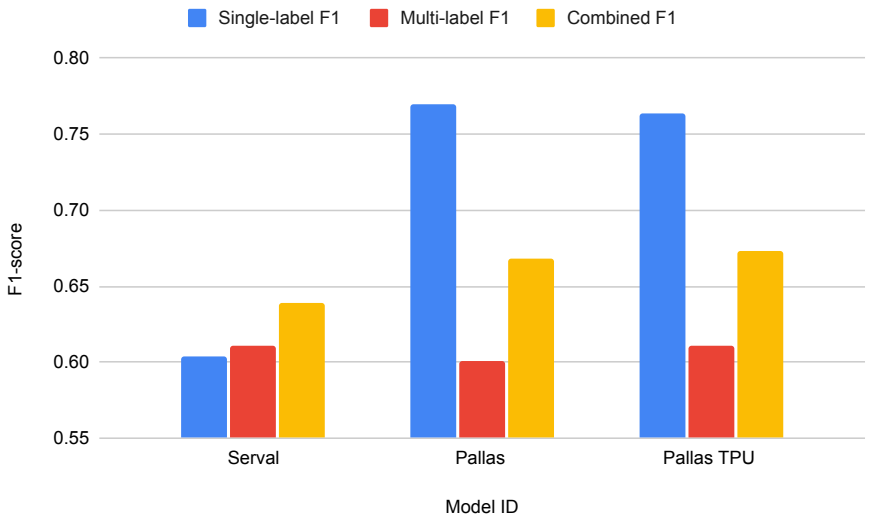


Figure 6.7: **Final metrics for each model on each test set in the second experiment. Note that these results were obtained from testing the full test dataset.**

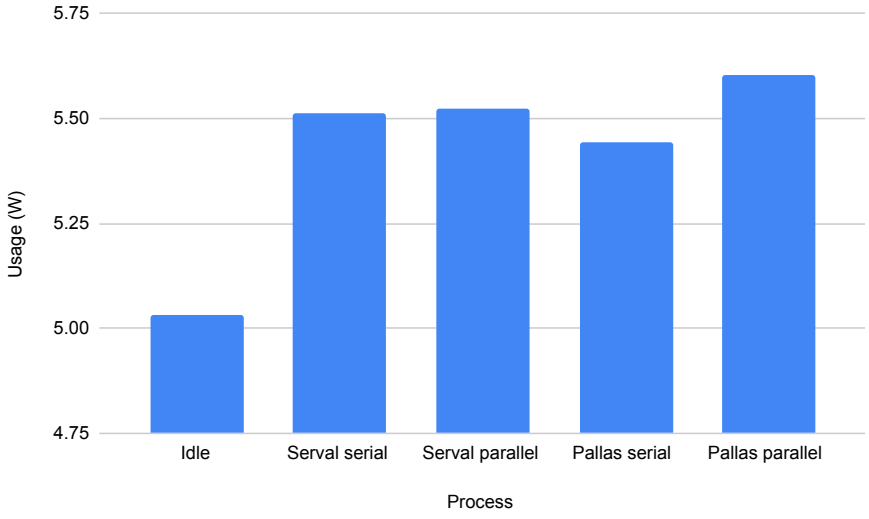


Figure 6.8: **Power usage of the Raspberry Pi in different usage scenarios**

The float-quantized models do not show much better performance compared to the original model.

Regarding RAM usage, all optimized models perform much better than the original model, sometimes requiring less than half the RAM usage the original model does.

5. Running with a TPU requires less power than running without TPU.

   It is clear from Figure 6.8 that Pallas when running in serial and avoiding parallelization overhead is much more efficient compared to Serval doing the same thing.

Chapter 7

Conclusion

This thesis describes the process of improving an existing sound recognition sensor, while exploring and applying the state of the art. As described in section 2.2, Serval was based and built on rather outdated code. Furthermore the data augmentation technique used is not a common method and does not produce a well balanced dataset.

This is why first the data augmentation technique was revised in chapter 4. As shown in section 4.1, the method used shows some flaws. Multiple methods were tested of which a hybrid data augmentation technique using both an improved method based on Serval's, and a novel data augmentation technique using pink noise to differentiate samples are combined. As shown in section 4.6, this hybrid method achieves the best F_1 scores for each test set: single-label samples, multi-label samples and the combined single and multi-label samples dataset.

In chapter 5, Serval's embedding, VVGish, was tested against multiple configurations of a more modern embedding, OpenL3. As shown in section 5.5, not all configurations have the same impact. It is clear that using an OpenL3 embedding that was trained on a musical dataset outperforms OpenL3 embeddings trained on environmental data, which is surprising given our environmental dataset. The difference in performance between embeddings with different input representations is not as clear. However when considering the amount of extra time required for the mel-spectrogram embeddings, the linear spectrogram embeddings outperform the mel-embeddings. Thus the best OpenL3 configuration to use would be one trained on a musical dataset with linear spectrograms.

Finally chapter 6 shows us that running OpenL3 is not as easy as running VGGish. On my PC, an experiment using OpenL3 takes about 31 times as long to run as the same experiment with VGGish. On the Raspberry Pi, this ratio is as high as 110 times as long. Unfortunately this makes it impossible to run the sensor with OpenL3 in real-time. To remedy this, this chapter explores different options to speed up OpenL3 execution. As shown in section 6.6, when running an embedding that was optimized for usage on an auxiliary device, the time required to run is reduced significantly to a point where running in real-time is possible. Furthermore power usage is decreased and the F_1 scores are slightly increased.

In section 1.1, I posed several research questions. Here we reiterate them and try to answer them.

- Can we reduce the class imbalance and improve the accuracy of the Serval model without changing the dataset?

  In chapter 4 we learn that different data augmentation techniques may improve both the accuracy and the balance of the model. In particular the hybrid data augmentation shows higher accuracy for almost all classes and lower variance between class accuracies.

- Is changing the embedding enough to reduce imbalance or improve the accuracy of a model?

  We saw that switching VGGish to OpenL3 alone increases the accuracy of the downstream model considerably. Unfortunately this could not be a full comparison as Serval does not train on OpenL3 embeddings, but it appears that a Pallas model trained on an OpenL3 embedding performs much better than a Pallas model trained on a VGGish embedding.

- Can we get a model better than the Serval model while staying within computational and timing restraints?

  After changing the embedding used to OpenL3, it required a number of optimizations as detailed in chapter 6, but in the end Pallas runs with better accuracy and lower power usage compared to Serval. This did however require a hardware upgrade.

In conclusion, I think this thesis answers the posed research questions well and shows a good insight in which methods were evaluated and why these were chosen. Some chapters could benefit from extended research, such as evaluating more data augmentation techniques and combinations in chapter 4 and more embeddings in chapter 5. Because of uncertainties in the workload of the latter parts and lack of time, and the chapters being sufficiently clear, they were wrapped up in their current state.

7.1 Process

When I started working on this thesis, I started researching the Serval codebase. I quickly found that there were lots of inconsistencies and design choices that made the codebase hard to understand. Furthermore, my relative lack of a machine learning background did not help in understanding Serval and in developing the Pallas.

After developing a basic framework for Pallas, containing alternative data augmentation methods, support for embedding with VGGish and OpenL3, and training the alternative Pallas network, I started writing the paper. Concurrently I started work on upgrading the device code. I quickly found that OpenL3 could not run fast enough on the Raspberry Pi, and that the code could run much faster when running different parts in parallel. Rewriting OpenEars to a parallel version would take as much time as starting from scratch, so this is when I started work on the new multi-threaded Pallas device code.

While working on the paper I found that I did not explore enough data augmentation methods and OpenL3 configurations. After adding the new data

augmentation methods mentioned in chapter 4 and OpenL3 configurations mentioned in chapter 5, I found that my original methods were worse than these new methods. After running all steps again I got the final results as shown in the paper.

Remaining now is only the paper. During the last part of the thesis I only worked on the paper and multiple presentation opportunities. After some iterations I present to you the final version of this paper.

In hindsight the biggest problems were my lack of a solid machine learning background and the somewhat rash approach at the start of the thesis. This forced me multiple times to redo or rerun parts of previously done work. If I had started with this breadth-first approach starting with data augmentation and working from that on, instead of this depth-first approach of trying to get everything working and then fine-tuning every part, a lot of time could have been saved.

7.2 Future work

As stated before, some parts of the thesis are not perfect yet at the end. This section details which steps can be taken to improve upon this work.

First of all, a large part of this thesis is focused on working around class imbalance. Furthermore this class imbalance in the dataset makes it hard to use the usual 80%/10%/10% train/validation/test split, as 10% of the 23 moped alarms samples would come down to having only 2 test and 2 validation samples for this class. This does not give us a representative accuracy. These problems are solved when the class imbalance is not as apparent in the used dataset. Finally there are certain classes that may be added to avoid false positives or for further interest, such as fireworks (to avoid false positives with gunshots), wind or rain noise, airplanes and vehicle sirens.

Chapter 4 about data augmentation evaluates multiple data augmentation options, and concludes that a hybrid data augmentation technique using multiple data augmentations shows the best potential. However, many promising options could be evaluated, as well as more combinations of multiple techniques. Finally a pipeline approach was not tested, where, for example, noise is added to each combined sample. These approaches could be considered for future exploration.

In chapter 5, VGGish was tested against some configurations of OpenL3. No comparison was made with other embeddings due to time constraints. Testing an embedding is not that simple. These embeddings were chosen because they are readily available and easy to use, but for other embeddings this may not be the case. Nevertheless other embeddings may show promising results, and these could be taken in consideration.

As mentioned in section 5.3, the Pallas model is a simplified version of the Serval model. This approach was chosen partly because of my lack of a machine learning background and due to time considerations. However this model may be improved by using additional layers, for example the MOE layer that Serval uses. Furthermore layers can be added that assist in training, such as noise or dropout layers. Finally many loss and optimizer functions were tried. Better results may be achieved after applying more fitting loss, optimizer and activation functions.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. In *arXiv: Computer Vision and Pattern Recognition*, 2016.

[3] Relja Arandjelovic and Andrew Zisserman. Look, listen and learn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 609–617, 2017.

[4] Yusuf Aytar, Carl Vondrick, and Antonio Torralba. Soundnet: Learning sound representations from unlabeled video. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 892–900, Red Hook, NY, USA, 2016. Curran Associates Inc.

[5] Margaret Carol Bell and Fabio Galatioto. Novel wireless pervasive sensor network to improve the understanding of noise in street canyons. *Applied Acoustics*, 74(1):169–180, 2013.

[6] Juan P. Bello, Claudio Silva, Oded Nov, R. Luke Dubois, Anish Arora, Justin Salamon, Charles Mydlarz, and Harish Doraiswamy. Sonyc: A system for monitoring, analyzing, and mitigating urban noise pollution. *Commun. ACM*, 62(2):68–77, jan 2019.

[7] Dick Botteldooren, Timothy Van renterghem, Damiano Oldoni, Dauwe Samuel, Luc Dekoninck, Pieter Thomas, Weigang Wei, Michiel Boes, Bert De Coensel, Bernard De Baets, and Bart Dhoedt. The internet of sound observatories. *Proceedings of Meetings on Acoustics*, 19(1):040140, 06 2013.

[8] Keunwoo Choi, Deokjin Joo, and Juho Kim. Kapre: On-gpu audio preprocessing layers for a quick implementation of deep neural network models with keras. In *Machine Learning for Music Discovery Workshop at 34th International Conference on Machine Learning*. ICML, 2017.

[9] Courtenay V. Cotton and Daniel P. W. Ellis. Spectral vs. spectro-temporal features for acoustic event detection. In *2011 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 69–72, 2011.

[10] Aurora Linh Cramer, Ho-Hsiang Wu, Justin Salamon, and Juan Pablo Bello. Look, listen, and learn more: Design choices for deep audio embeddings. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3852–3856, 2019.

[11] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 776–780. IEEE Press, 2017.

[12] Rishabh Gupta, Ishu Gupta, Deepika Saxena, and Ashutosh Kumar Singh. A differential approach and deep neural network based data privacy-preserving model in cloud environment. *Journal of Ambient Intelligence and Humanized Computing*, 2022.

[13] Shawn Hershey, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, Devin Platt, Rif A. Saurous, Bryan Seybold, Malcolm Slaney, Ron J. Weiss, and Kevin Wilson. Cnn architectures for large-scale audio classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 131–135. IEEE Press, 2017.

[14] Chinnavat Jatturas, Sornsawan Chokkoedsakul, Pisitpong Devahasting Na Avudhva, Sukit Pankaew, Cherdkul Sopavanit, and Widhyakorn Asdornwised. Feature-based and deep learning-based classification of environmental sound. In *2019 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pages 126–130, 2019.

[15] M.I. Jordan and R.A. Jacobs. Hierarchical mixtures of experts and the em algorithm. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, volume 2, pages 1339–1344 vol.2, 1993.

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[17] Brian McFee, Matt McVicar, Daniel Faronbi, Iran Roman, Matan Gover, Stefan Balke, Scott Seyfarth, Ayoub Malek, Colin Raffel, Vincent Lostanlen, Benjamin van Niekirk, Dana Lee, Frank Cwitkowitz, Frank Zalkow, Oriol Nieto, Dan Ellis, Jack Mason, Kyungyun Lee, Bea Steers, Emily Halvachs, Carl Thomé, Fabian Robert-Stöter, Rachel Bittner, Ziyao Wei, Adam Weiss, Eric Battenberg, Keunwoo Choi, Ryuichi Yamamoto, CJ Carr, Alex Metsai, Stefan Sullivan, Pius Friesch, Asmitha Krishnakumar, Shunsuke Hidaka, Steve Kowalik, Fabian Keller, Dan Mazur, Alexandre Chabot-Leclerc, Curtis Hawthorne, Chandrashekhar Ramaprasad, Myungchul Keum, Juanita Gomez, Will Monroe, Viktor Andreevitch Morozov, Kian Eliasi, nullmightybofo, Paul Biberstein, N. Dorukhan Sergin, Romain Hennequin, Rimvydas Naktinis, beantowel, Taewoon Kim, Jon Petter Åsen, Joon Lim, Alex Malins, Darío Hereñú, Stef van der Struijk, Lorenz Nickel, Jackie Wu, Zhen Wang, Tim Gates, Matt Vollrath, Andy Sarroff, Xiao-Ming, Alastair Porter, Seth Kranzler, Voodoohop, Mattia Di Gangi, Helmi Jinoz, Connor Guerrero, Abduttayyeb Mazhar, toddrme2178, Zvi Baratz, Anton Kostin, Xinlu Zhuang, Cash TingHin Lo, Pavel Campr, Eric Semeniuc, Monsij Biswal, Shayenne Moura, Paul Brossier, Hojin Lee, and Waldir Pimenta. librosa/librosa: 0.10.0.post2, March 2023.

[18] G. Vincze P. Szendro and A. Szasz. Bio-response to white noise excitation. *Electro- and Magnetobiology*, 20(2):215–229, 2001.

[19] Giambattista Parascandolo, Heikki Huttunen, and Tuomas Virtanen. Recurrent neural networks for polyphonic sound event detection in real life recordings. *CoRR*, abs/1604.00861, 2016.

[20] Karol J. Piczak. Environmental sound classification with convolutional neural networks. In *2015 IEEE 25th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, 2015.

[21] Karol J. Piczak. Esc: Dataset for environmental sound classification. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, page 1015–1018, New York, NY, USA, 2015. Association for Computing Machinery.

[22] Justin Salamon and Juan Pablo Bello. Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal Processing Letters*, 24(3):279–283, 2017.

[23] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, Jul 2019.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[25] S. S. Stevens, J. Volkmann, and E. B. Newman. A Scale for the Measurement of the Psychological Magnitude Pitch. *The Journal of the Acoustical Society of America*, 8(3):185–190, 06 2005.

[26] Naoya Takahashi, Michael Gygli, Beat Pfister, and Luc Van Gool. Deep Convolutional Neural Networks and Data Augmentation for Acoustic Event Recognition. In *Proc. Interspeech 2016*, pages 2982–2986, 2016.

[27] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[28] Zhichao Zhang, Shugong Xu, Shunqing Zhang, Tianhao Qiao, and Shan Cao. Attention based convolutional recurrent neural network for environmental sound classification. *Neurocomputing*, 453:896–903, 2021.

List of terms and acronyms

Acronyms

AED acoustic event detection. 9, 11, 13, 25

DCNN deep convolutional neural network. 10–12, 25, 28

ESC environmental sound classification. 9, 10

LSTM long short-term memory. 5, 10, 21, 30

MOE Mixture of (Logistic) Experts. 5, 30, 49

RNN recurrent neural network. 10

TPU Tensor Processing Unit. 35, 39

Glossary

overfitting (In machine learning) This happens if the model is too large for the input data. Basically, the model does not learn to recognize the features that define a class, but it remembers the input data. 18

spectrogram (In audio) A visual representation of the amplitudes versus time and frequency. 19