Thesis

Kilian Callebaut

July 19, 2021

Abstract

Abstract

Contents

Chapter 1

Introduction

TODO: Introduce the context of multi-task deep learning audio frameworks

1.1 Example: Variating audio task interaction in Multi-Task Research

TODO: Introduce original experiment set-ups as a basis for explaining what kind of multi-task development can be done, what the structure is and what it has to deal with

1.2 Multi-Task Research

TODO: Go further in depth about the general state of audio multi-task research and why this system is needed in that. Why is a speed up in development needed in the field?

Multi-task learning (MTL) is a machine learning paradigm where multiple different tasks are learned at the same time, exploiting underlying task relationships, to arrive at a shared representation. While the principle goal was to improve generalization accuracy of a machine learning system [Caruana, 1997], over the years multitask learning has found other uses, including speed of learning, improved intelligibility of learned models [Caruana, 1997], classification fairness [Oneto et al., 2019] and as a means to compress multiple parallel models [Georgiev, 2017]. This led to the paradigm finding its usage in multiple fields, including audio recognition.

The field of audio recognition is varied and ever expanding, due to a growing number of large public and non-publicly available datasets (e.g. AudioSet [Gemmeke et al., 2017]) each with their own variations like sources, lengths and subjects. The tasks in the field can roughly be divided into three categories: Speech recognition tasks, Environmental Sound recognition tasks and Music recognition tasks, along with tasks that combine multiple domains [Duan et al., 2014]. These domains inherently have a different structure from each other, which requires different processing and classification schemes. Speech for example, is inherently built up out of elementary phonemes that are internally dependent, the tasks linked to which have to deal with the exact differentiation and characterization of these, to varying degrees. Environmental sounds in contrast, do not have such substructures and cover a larger range of frequencies. Music then has its own stationary patterns like melody and rhythm [BoreGowda, 2018]. A general purpose audio classification system, dealing with real life audio, would have to deal with the presence of each of these types of audio though, regardless if its task is only in one of the domains.

Usually, in order to achieve high performance, it is necessary to construct a focused detector, which targets a few classes per task. Only focusing on one set of targets with a fitting dataset however, ignores the wealth of information available in other task-specific datasets, as well as failing to leverage the fact that they might be calculating the same features, especially in the lower levels of the architecture [Tagliasacchi et al., 2020]. This does not only entail a possible waste of information (and thus performance) but also entails a waste of computational resources, as each task might not require its own dedicated model to achieve the same level of performance. Originally conventional methods like Gaussian Mixture Models (GMM) and State Vector Machines (SVM) were the main focus, but due to the impressive results in visual tasks deep learning architectures have seen a lot of attention. The emergence of deep learning MTL set-ups is still fairly recent in audio recognition. While it has seen both successful [Tonami et al., 2019] applications and less successful [Sakti et al., 2016] when combining different tasks, very little is known about the exact circumstances when MTL works in audio recognition.

1.3 Developing Deep Learning Multi-Task Setups

TODO: Outline the steps in developing deep learning Multi-Task Set ups and how shortcuts can be made to speed up/improve the process. I.e. which problems have to be answered in the system. What developmental problems are you addressing?

Issues to face:

Data Reading

- Developing valid input for loading and training for different datasets takes time and is error prone, while a lot of the processes are repetitive =¿ DataReader to TaskDataset
- While developing and testing different set ups, intermediate parts (e.g. the feature extraction method, file reading method, resampling method) as well as additional parts (e.g. resampling) often have to be varied and replaced, which might be a complex and time consuming process depending on the amount of rewrites and datasets required =¿ Easily interchangeable pipeline pieces
- Developing read/write functionalities per dataset is time consuming and potentially chaotic if done differently every time. Add to that the possibility of testing different set-ups for the same dataset which would require good file management. =¿ Standardizing dataset read/write and automatic abstraction of reading when files are present
- loading in multiple datasets might be too memory intensive for a lot of systems
- Running the code on a different system requires good datamanagement and changeable path locations
- While some datasets have predefined train/test sets, others do not, which would require different handling of both cases which might be time consuming and error prone (===¿ actually a consequence of standardizing in this way, i.e. engineering problem)
- Some Datasets can have multiple tasks on the same inputs (===¿ actually a consequence of standardizing in this way, i.e. engineering problem)

Data Loading

- Each training procedure needs a train and test set, which for some datasets need to be created using k-fold validation set-ups and for some don't. When quickly trying to execute multiple set-ups this requires a lot of repetitive work. It's also error prone, as creating train/test sets the wrong way can cause data leaking and thus weaken the evaluation. (e.g. if the normalization is wrongfully calculated (-¿ the mean and stdev) on both the train and test set, the system will use information it shouldn't have and will perform unforeseenly worse on unseen data). =¿ Abstraction to train/test set generation and handling

- Additional features like transforming or filtering the data again take up development time to specify for each separate dataset as well as can be a gruesome process to apply after the data is read into matrices. =¿ abstraction to dataset functions that don't rely on knowledge of the matrix structures

Training

- Combining datasets from tasks can be done in numerous ways, which can impact performance on training. =¿ Allow multiple and extendible ways to combine tasks in the training batches

- In multi-task training, loss calculation is done by combining separate losses from tasks which can be done in numerous ways and might be interesting to explore =¿ Allow multiple and extendible ways to combine losses in training

- In general for multi-task research, lots of parameters and parts should be varied =¿ Allow replacability of each part in training, without jeopardizing the training function

- There are three types of task output structures in classification: binary, multi-class and multi-label outputs which each have to be handled uniquely while still being able to be combined =¿ Abstraction of task type handling

- Calculating, storing and visualizing results in an efficient way for comparison is crucial and can take up valuable development/debugging time =¿ abstraction to calculating, storing and saving results that allows for easy comparison between runs

- Interrupted learning - the process of interrupting an ongoing training loop and restarting it later - requires good data management and saving of parameters to be loaded up again later, which is both error prone and time consuming =¿ quick and easy way to restart an old run from a certain point

Extra issues to be solved

- Figuring out the pipeline for multi-task deep learning set ups can be difficult, considering there are numerous types of and variations in multi-task learning schemes and not a lot of documentation on how to approach these

- Multi-task set-ups are most likely going to be compared to single task set-ups, meaning the code should already take this in account or handle the two cases separately

1.4 Challenges

TODO: Define the technological challenges in answering those problems. What problems/challenges do you face or have to take in account in developing such a system?

1.5 Contributions

TODO: Outline what new your thesis works contributes.

1.6 Outline

TODO: Summarize the rest of the thesis' structure

Chapter 2

Related Work

2.1 Audio Classification

TODO: Explain the field of audio classification, how it normally works, what kind of tasks there are and what kind of things are researched

2.2 Multi-task Learning

TODO: Explain the field of multi-task learning, where it came from, what the paradigm brought in improvements and what kind of things are researched

2.3 Multi-task Deep Learning Audio Tasks

TODO: Explain the merging of these fields and what (little) work has been done there so far. Also what it requires for more work to be done and what the current work lacks.

2.4 Development Frameworks

TODO: Get examples in from other development frameworks, how they answered the needs in their fields and why they are needed

Chapter 3

Problem Statement

TODO: Explain that this chapter is about defining the problem and what the solving system should be

3.1 Use Cases

TODO: Explain the need for requirements by clarifying examples

3.2 Developers

3.2.1 Researchers

TODO: Who need to vary different parts of the pipeline and report on their effect

3.2.2 Optimizers

TODO: Who need to be able to build and train the best performing model

3.3 Design Principles

TODO: Outline the assumptions you make that the system is built on and the objectives the framework has to achieve to offer better developmental support

3.3.1 Fast Prototyping

Deliver a framework which users can use to quickly build, train and test a deep learning multi-task pipeline without compromise.

3.3.2 Dynamic Handling Of Deviating Cases

The framework should be capable of handling a wide range of variations in datasets and task structures, without requiring adjustments to be made to the overall structure.

3.3.3 Easy Extendibility

Every part of the pipeline that has variations, should be variable in a modular way.

3.3.4 Abstracted File Management

Functionalities that require a system to write or read files on the system, should be abstracted to the point that users should not be forced to input more than the desired location of the files.

3.3.5 Guided Development

Developers getting familiar with the system should both be in possession of working examples, as well as guiding functions they have to implement in order to build a functioning pipeline.

3.4 Non-functional Requirements

3.5 Functional Requirements

Chapter 4

System Design

TODO: Explain how the system was modeled based on the design requirements

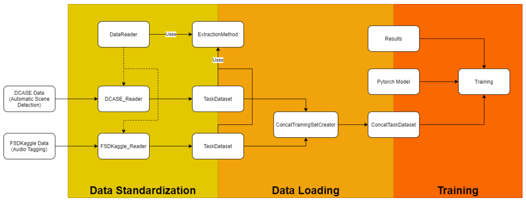

Figure 4.1: Simplified System overview

4.1 Easy changeable variables

4.1.1 Different datasets

4.1.2 Different Sample Rate

4.1.3 Different Feature Extraction

4.1.4 Different Data Transformation

4.1.5 Different Dataloading

4.1.6 Different DL Model

4.1.7 Different Optimizer

4.1.8 Different loss calculation

4.1.9 Different loss combination

4.1.10 Different Stopping Criteria

4.1.11 Different Saving Locations

4.2 Easy expansions

4.2.1 Adding Datasets

4.2.2 Adding Tasks to datasets

4.3 Simplifying abstractions

4.3.1 Saving/Reading Extracted Datasets

4.3.2 Index Mode

4.3.3 Combination of different datasets

TODO: The ConcatTaskDataset function

4.3.4 Train/test generation

4.3.5 Evaluation

4.3.6 Result Saving and Visualizing

4.3.7 Interrupted Learning

4.4 Developmental side rails

4.4.1 Abstract Data Reader

4.4.2 Abstract Extraction Method

4.4.3 Standardized valid input

4.4.4 Centralized Train/test Operations

Chapter 5

Implementation

5.1 Technology

The implementation is built in python and relies on pytorch for deep learning modeling and training. PyTorch is one of the biggest and most accessible frameworks for developing neural networks. This framework is designed to utilise its objects as to minimize an extra learning curve, as well as lighten any developmental work that it still requires.

5.2 System Architecture

TODO: Reiterate the design principles and a description of what functionally has been built

5.3 High Level Description

TODO: Simplified overview of the pipeline

5.4 Data Reading

The first part of the pipeline is responsible for reading the audio data from datasets, extracting their features and storing it along their targets in valid objects. This also includes abstractions for reading and writing of files that store these objects for later use. As every dataset has their own structure and storage method, the implementation for every data reader has to be specified by the developer. The structure therefore is built around following the pattern layed out in the **DataReader** class and extending its functions with dataset specific ones. The pattern goes as follows.

5.4.1 Structure

First, a **DataReader** object is instantiated with an **ExtractionMethod** object and relevant parameters. The **ExtractionMethod** is the tool used to transform individual data instances. Since specific data transformations can often rely on the specific extraction method used (TODO: Give an example of this), it is opted to group multiple transformation functions this way, which will be explained further later. When the features still have to be extracted, the datareader will first read the structure (e.g. list of locations, list of read signals, tensorflow dataset, ...) in memory, through the *load_files* function. Then, the standardized object, called a **TaskDataset**, is made in the *calculate_taskDataset* function. This requires a list of input tensors, a list of targets, the **ExtractionMethod** object, a unique name and the list of target names. The idea is that using the **ExtractionMethod** object, the list of inputs is created by iterating over the structure, getting the read wav form and extracting the desired features per audio instance in a PyTorch tensor object. When the **TaskDataset** object is correctly created, the next step is then to write the extracted features to files in the *write_files* method. While this method can be extended if it is desired to write additional files, it is not necessary as the **TaskDataset** object already has its own file management functionalities. Because the created **TaskDataset**object also received the **ExtractionMethod** object, it handles its files depending on the specified extraction method, thus nullifying any need for further adaptations to be made if the developer wants to extract different features for the same dataset. If everything is written once already, the DataReader is able to detect this using its *check_files* method and will automatically read in the **TaskDataset** instead using the *read_files* method.

Having given the general overview of how to go from audio data to standardized objects through the framework, it's also important to note what it is designed to be invariant to. More specifically, the **TaskDataset** object has a number of functionalities which do not require any additional handling when utilized. The biggest one is the so called index mode, which automatically distributes the data over files that are read when needed. This only requires to be activated at the initialization of the TaskDataset, after which the necessary functionalities will be switched out for index based ones.

Further factors the data structure can automatically deal with are datasets which have predefined train and test sets. This is possible through the hold - train - test set-up which allows for the train and test set to be defined and linked through the holding TaskDataset. If this is not the case, then the data is directly inserted in the holding TaskDataset and the splits can be made later. Separate Train and Test TaskDatasets can have their own storage locations, which the file management automatically handles as if it's the unseparated case.

The last one are multiple tasks for the same dataset, which can simply be inserted without any limit into the same TaskDataset, after which the getter functions will automatically take all targets for all tasks at the specified index.

5.4.2 DataReader

TODO: insert Data Reader Model

The **DataReader** class is meant as a parent class to be extended by specific implementations for each dataset. As previously mentioned, it has a number of abstract functions which require to be extended. Besides those, it also contains an automatic parser for ExtractionMethod objects from text, in case the input is directly read from files e.g. json. Alongside that, it also contains a function to read in wav files at a specific location, using the Librosa library and a separate resampling function, in case the signal is already read. The ability to resample signals is used often in multi-task learning, which makes it the extra parameter in the *calculate_input* function.

5.4.3 ExtractionMethod

TODO: insert ExtractionMethod Model

If the DataReader is the workbench to transform audio datasets to TaskDatasets, then the **ExtractionMethod** class is the hammer. The functionality of this class is instance based, but groups together a number of transformations. The main one of course being feature extraction. This class works similarly to the DataReader class as it has a number of abstract methods to be extended if one wants to make their own implementation. However, a number of them are already available, like the MFCC, the Melspectrogram and the LogbankSummary (TODO: Refer to papers using and explaining these) features. At instantiation, this class should receive extraction parameters and preparation parameters. The extraction parameters should be a dictionary with parameters which can fit in the utilised extraction method. Since these are stored in the object, the same object can easily be reused on different datasets for consistency and easy scalability.

The other functionalities that were referred to, to possibly be dependent on the extraction method used are data transformations. One is the normalization of data. This requires scalers to be fit on the data to then transform each instance according to the scalers (typically infers calculating the mean and the variance of the whole dataset and then scaling these so that the mean of all instances is 0 and the variance is 1). Aside from scaling the data, the ExtractionMethod object also includes a function for other transformations. A typical use for this is cutting the matrices into same sized frames, as audio data can have varying lengths. This function is already included, along with a slight alternative, where the input matrices are not cut but windowed, meaning one input matrix result in multiple windows of the same size with overlap, so no data is lost. Standard methods for fitting, scaling, inverse scaling entire 2D inputs and 2D inputs per row are also already available and are implemented using the sklearn preprocessing toolbox.

(TODO: Explain the example of the Logbank summary and the MelSpectrogram requiring different handling)

5.4.4 TaskDataset

The TaskDataset structure is how the framework manages to standardize inputs and targets in one valid object for training. It extends PyTorch's Dataset class to allow for integration with its dataloader objects. This class is responsible for containing the data with functionalities for getting data, storage and transformation. This class is however only a parent class to the Hold-train-test structure, which is set up to deal with functionalities related to generating and handling valid train and test sets. The entire TaskDataset structure is designed to be customizable, but invariant when handling from the outside. There are 5 parts to this that have their own strategies: File

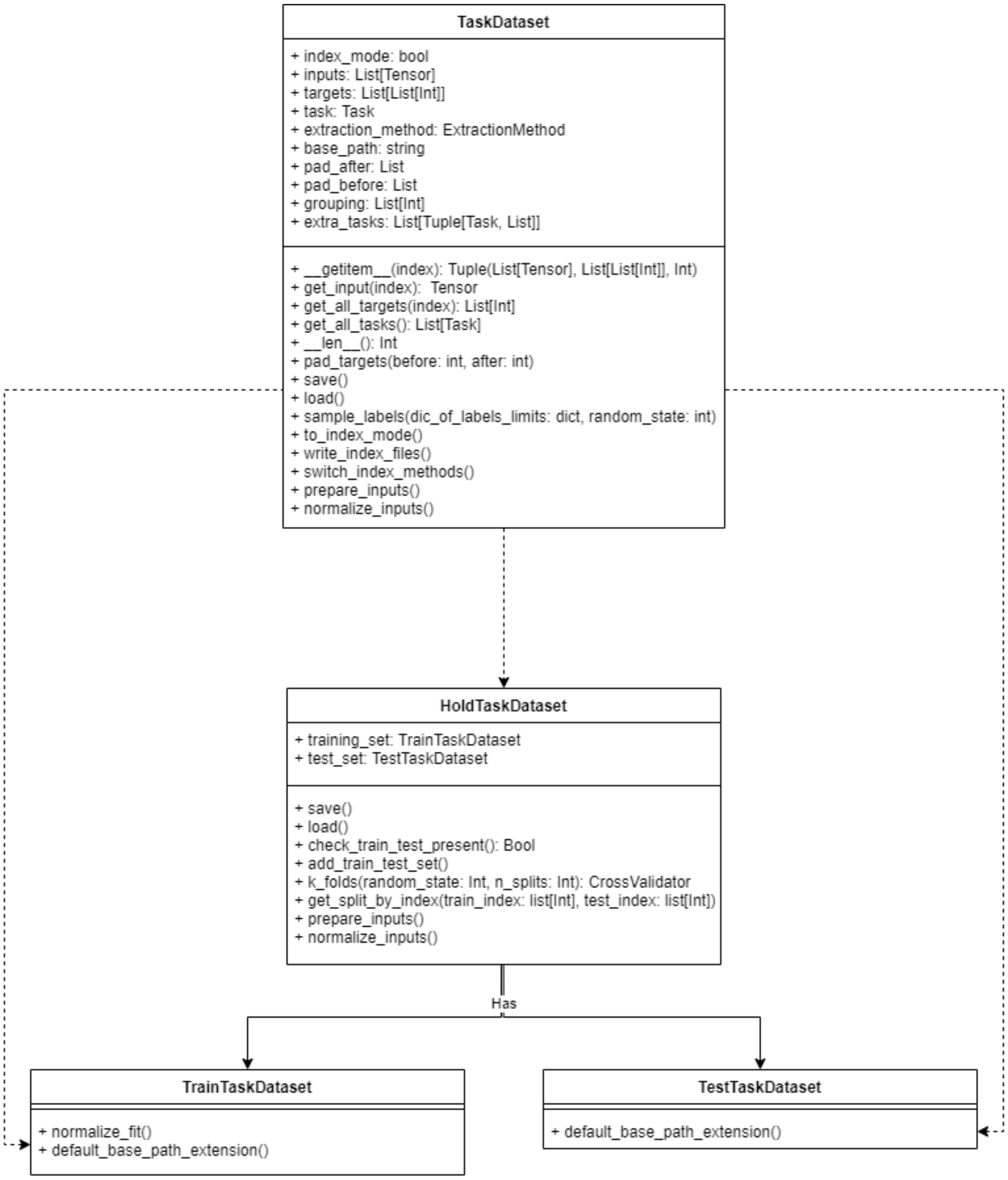
TaskDataset
+ index_mode: bool
+ inputs: List[Tensor]
+ targets: List[List[Int]]
+ task: Task
+ extraction_method: ExtractionMethod
+ base_path: string
+ pad_after: List
+ pad_before: List
+ grouping: List[Int]
+ extra_tasks: List[Tuple[Task, List]]
+ __getitem__(index): Tuple(List[Tensor], List[List[Int]], Int)
+ get_input(index): Tensor
+ get_all_targets(index): List[Int]
+ get_all_tasks(): List[Task]
+ __len__(): Int
+ pad_targets(before: int, after: int)
+ save()
+ load()
+ sample_labels(dic_of_labels_limits: dict, random_state: int)
+ to_index_mode()
+ write_index_files()
+ switch_index_methods()
+ prepare_inputs()
+ normalize_inputs()
HoldTaskDataset
+ training_set: TrainTaskDataset
+ test_set: TestTaskDataset
+ save()
+ load()
+ check_train_test_present(): Bool
+ add_train_test_set()
+ k_folds(random_state: Int, n_splits: Int): CrossValidator
+ get_split_by_index(train_index: list[Int], test_index: list[Int])
+ prepare_inputs()
+ normalize_inputs()
Has
TrainTaskDataset
+ normalize_fit()
+ default_base_path_extension()
TestTaskDataset
+ default_base_path_extension()

Figure 5.1: TaskDataset Structure

management, structure of data and the index mode.

First the file management will be detailed. The idea is simple: the save function writes the Taskdataset to files and the load function reads the files to a valid TaskDataset object. Using the joblib library, which allows files to easily be written and read in a parallelised manner, the inputs are stored separately from the targets and the other information. In order to create inputs that used different extraction methods easily, the storage takes includes the name of the stored ExtractionMethod.

Next is structure of the data. The input features are stored as PyTorch tensors in a python list. The targets are stored as lists of binary numbers. These two lists should have the same length. One data instance thus has a feature tensor at index i in the inputs list and a target list at index i in the targets list, where the number is 1 if the instance has the label at that position. The named labels and their order are stored in the **Task** object, which is also stored in the TaskDataset object. The **Task** object holds all information related to the Task as well as functionalities which depend on the type of task used. If more than one task should be available for the same dataset - without having to put multiple copies of the same data in the combined dataset - then these can be inserted and stored in the list of extra tasks, which consist of tuples of **Task** and list of targets pairs. The indexes in these lists of targets should still refer to the same data instance as the other indexes.

Lastly, the index mode is discussed. The index mode basically writes the feature matrices to individual files which are loaded when the getter function is called. This prevents that the whole dataset has to be loaded into memory. A TaskDataset object should not be handled differently from the outside when it is running in index mode or not. This is achieved by switching out the getter method for feature matrices, the save function and the load function to one specified for index mode. The list of input tensors is switched out for a list of integers that represent the indexes of the inputs. A input feature matrix is loaded and saved with this index in its file name. All other information is kept as usual.

Getting a data instance - i.e. the feature matrix and targets - requires more than just plucking the corresponding elements from the list. While the data loading is discussed later, getting an item at an index from a TaskDataset infers getting three things: the feature matrix as input, the target list as correct output and the task_group. Getting the feature matrix is a simple indexing operation, after which the scaling transformation is applied. This transformation is applied every time in the get function, as the same data likely has to be rescaled multiple times - e.g. in a five fold cross validation training set-up - so there is no need to revert the transformation every time.

Getting the target data has to take in account more factors though. First of all, it is required for creating batches that all returned items have the same shape, meaning that every returned input and target list must have the same dimensions. Correctly shaping the input matrices can be done using the prepare inputs functionalities beforehand, but the targets are different.

5.4.5 Examples To Get List

Example of creating a DataReader

Example of creating a TaskDataset in index mode

Example of creating a HoldTaskDataset with predefined train and test set

5.5 Data Loading

5.6 Training

5.7 Complementary tools

TODO: Describe things like the index mode, which answer additional needs outside of fast development.

5.8 Extendibility

Chapter 6

Evaluation

6.1 Goals and Results

6.2 Discussion on the implementation

6.3 Memory Saving (and such)

TODO: Any objective demonstration of the system's functionalities (like index mode)

6.4 Requirements

TODO: Going back to the (non-)functional requirements and how the system addresses them

Chapter 7

Conclusion

7.1 Future Work

Bibliography

HB BoreGowda. Environmental sound recognition: A survey. 2018.

Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

Shufei Duan, Jinglan Zhang, Paul Roe, and Michael Towsey. A survey of tagging techniques for music, speech and environmental sound. *Artificial Intelligence Review*, 42(4):637–661, 2014.

Jort F Gemmeke, Daniel PW 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)*, pages 776–780. IEEE, 2017.

Petko Georgiev. Heterogeneous resource mobile sensing: computational offloading, scheduling and algorithm optimisation. 2017.

Luca Oneto, Michele Doninini, Amon Elders, and Massimiliano Pontil. Taking advantage of multitask learning for fair classification. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 227–237, 2019.

Sakriani Sakti, Seiji Kawanishi, Graham Neubig, Koichiro Yoshino, and Satoshi Nakamura. Deep bottleneck features and sound-dependent i-vectors for simultaneous recognition of speech and environmental sounds. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 35–42. IEEE, 2016.

Marco Tagliasacchi, Félix de Chaumont Quitry, and Dominik Roblek. Multitask adapters for on-device audio inference. *IEEE Signal Processing Letters*, 27:630–634, 2020.

Noriyuki Tonami, Keisuke Imoto, Masahiro Niitsuma, Ryosuke Yamanishi, and Yoichi Yamashita. Joint analysis of acoustic events and scenes based on multitask learning. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 338–342. IEEE, 2019.