Thesis

Kilian Callebaut

August 25, 2021

Abstract

Abstract

Contents

1 Introduction **5**
- 1.1 Example: Variating audio task interaction in Multi-Task Research . . . 5
- 1.2 Multi-Task Research . . . 5
- 1.3 Developing Deep Learning Multi-Task Set-ups . . . 7
- 1.4 Challenges . . . 9
- 1.5 Contributions . . . 9
- 1.6 Outline . . . 9

2 Related Work **10**
- 2.1 Audio Classification . . . 10
- 2.2 Multi-task Learning . . . 10
- 2.3 Multi-task Deep Learning Audio Tasks . . . 10
- 2.4 Development Frameworks . . . 10

3 Problem Statement **11**
- 3.1 Use Cases . . . 11
  - 3.1.1 Developing General Purpose Classifiers . . . 11
  - 3.1.2 Researching Multi-Task Set-ups . . . 12
  - 3.1.3 New Datasets . . . 13
- 3.2 Stakeholders . . . 14
  - 3.2.1 Researchers . . . 14
  - 3.2.2 Developers . . . 14
  - 3.2.3 Newcomers . . . 14
- 3.3 Design Principles . . . 15
- 3.4 Non-functional Requirements . . . 17
- 3.5 Functional Requirements . . . 17
  - 3.5.1 Data Reading . . . 18

3.5.2 Data Loading 19
3.5.3 Training 21

4 System Design **22**
4.1 Framework Design 22
4.2 Assumptions 26
4.3 Creating standardized objects 27
4.3.1 Variations 28
4.4 Manipulating datasets 29
4.5 Combining Datasets 31
4.6 Creating Model 33
4.7 Training and Evaluating Model 33
4.8 Three Implementations 34
4.9 Easy changeable variables 43
4.9.1 Different datasets 43
4.9.2 Different Sample Rate 43
4.9.3 Different Feature Extraction 43
4.9.4 Different Data Transformation 43
4.9.5 Different Dataloading 43
4.9.6 Different DL Model 43
4.9.7 Different Optimizer 43
4.9.8 Different loss calculation 43
4.9.9 Different loss combination 43
4.9.10 Different Stopping Criteria 43
4.9.11 Different Saving Locations 43
4.10 Easy expansions 43
4.10.1 Adding Datasets 43
4.10.2 Adding Tasks to datasets 43
4.11 Simplifying abstractions 43
4.11.1 Saving/Reading Extracted Datasets 43
4.11.2 Index Mode 43
4.11.3 Combination of different datasets 43
4.11.4 Train/test generation 44
4.11.5 Training 44
4.11.6 Evaluation 44
4.11.7 Result Saving and Visualizing 44
4.11.8 Interrupted Learning 44
4.12 Developmental side rails 44

4.12.1 Abstract Data Reader 44
4.12.2 Abstract Extraction Method 44
4.12.3 Standardized valid input 44
4.12.4 Centralized Train/test Operations 44

5 Implementation **45**
5.1 Technology 45
5.2 High Level Description 45
5.3 Data Reading 45
5.3.1 Structure 46
5.3.2 DataReader 48
5.3.3 ExtractionMethod 48
5.3.4 TaskDataset 49
5.3.5 Examples To Get List 51
5.4 Data Loading 51
5.4.1 Combining TaskDatasets 51
5.4.2 Generating Train and Test Sets 53
5.4.3 Preparing Inputs to same size 54
5.4.4 Scaling Inputs 54
5.4.5 Filtering Inputs 55
5.4.6 Loading Data 56
5.5 Training 57
5.5.1 Model Creation 57
5.5.2 Results Handling 57
5.5.3 Training Updating 59
5.5.4 Evaluation 62
5.6 Complementary tools 63
5.7 Extendibility 63
5.7.1 Classes that are meant to be extended 63
5.7.2 Classes that can be extended 65
5.7.3 Classes that should be extended from outside libraries . 66
5.7.4 Classes that should not be extended 66
5.8 Three Implementations 68

6 Evaluation **69**
6.1 Goals and Results 69
6.2 Discussion on the implementation 69
6.3 Memory Saving (and such) 69

6.4 Requirements 69

7 Conclusion **70**
7.1 Future Work 70

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 Set-ups

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

This section will examine some hypothetical use cases to serve as a basis for drawing up requirements for the framework. Each case examines a situation where a multi-task pipeline - from the raw datasets to the trained and evaluated models - needs to be created.

3.1.1 Developing General Purpose Classifiers

A lifelogging system that tracks and annotates its user's day through audio can be really useful for purposes like memory augmentation [citation needed] or safety [citation needed]. Imagine for example an on-person security system that can detect the environment its in as well as automatically alert when a noisy threat is present. In order to provide multiple, robust annotations for a piece of audio which provide possibilities for comprehensively browsing through the events in a day or statistical analysis, one has to build a classification system capable of this. While the usual approach for this is to build separate classifiers per task. A big implication of this, is that the raw audio is sent to a server, able to house and execute the classifiers, but this can cause several issues in this case. For one, there is the privacy and security issue of

continuously recording audio and sending it to a remote system. One has no assurance that this data is safe and not being misused for other purposes. Another one is that this centralization of data possibly causes issues for scalability both in terms of time and devices. Lastly, it requires sending a lot of data continuously which might be subject to network outages and traffic bottlenecks. A different approach to sending all that data to the server is performing the classification on the device and only sending the resulting output.

It is however rather unlikely that the device which records the lifelogging audio is able to fit and execute multiple trained neural network models at the same time. This is where the multi-task set-up comes in, which shares its network layers over multiple tasks and thus reduces the resources required for general purpose classification. Not only that, but multi-task classification has proven that it can achieve more robust representations of audio data which can be of serious benefit for the noisy, continuous real life data.

Developing such a system thus inquires a trained singular model which can perform multiple varied tasks reliably. It is unlikely that one dataset can be found suitable to train all tasks, but each task is likely to have at least one dataset. The trained model needs to be evaluated on real life audio using the same preprocessing as the training data.

3.1.2 Researching Multi-Task Set-ups

The Multi-Task learning paradigm has successfully been implemented to improve accuracy and robustness [citation needed]. Research into utilising multi-task learning can have great benefits for developing optimal audio recognition systems. For example, there is a yearly competition for developing audio classification systems with numerous objectives, called DCASE [citation needed]. Multi-task systems have seen more success [citation needed] here recently, both for tasks that have multiple objectives [citation needed] as well as for improving performance in singular tasks [citation needed].

In order for a researcher to find and compare working solutions to the posed challenge, they need to be able to vary multiple elements in the deep learning pipeline easily as well as compare their results to the baseline easily. However, compared to singular tasks, changing elements - e.g. the feature

extraction method, data transformations, loss calculation - for multiple tasks can lead to repetitive and error prone code modifications. Multi-task set-ups also bring more elements in the process, as the data from different tasks can require different handling, have to be combined and the shared system has to be updated. This makes researching multi-task set-ups more time consuming, which puts a strain on new developments in the field.

3.1.3 New Datasets

New datasets continuously become available, with different purposes in mind and different sources. These might suddenly make it possible to develop different kind of systems, but also improve existing ones, even if the data is of comparatively lower quality for the task at hand. Research has been done proving that weakly labeled data can be used to improve the performance of a system trained on strongly labeled data [citation needed]. Access to a new dataset opens up all sorts of opportunities for new goals or improving old systems.

Take for example the case where a dataset from google was extended with more fine grained labels in order to develop a recognition system that needed mere seconds for inference [citation needed, park]. In this case, older classifiers need to be tested for performance. Another thing is that this dataset was a subset of the larger dataset, with more fine grained annotations, but its parent set still contains valuable information for training the system. While this is essentially the same task set-up, the developer would still likely have to spend time making adjustments to how the original dataset is handled, the training loop functions and/or the results are calculated. A system which would take the combination of tasks in account beforehand and only require the correct handling of the new dataset would go a long way on cutting development time. This goes as well for being able to bring in older classifiers in a modular way and testing the difference in performance without any adjustments. The need is not only for the ability to develop working systems statically, but open the development up for additions, which do not require reworking the rest of the pipeline.

3.2 Stakeholders

The developmental framework needs to take in account different stakeholders that are concerned with building and training a multi-task Neural Network. These have different objectives which leads to different necessities the framework has to provide.

3.2.1 Researchers

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

Researchers are the people that, in this intended use of the word, would use the framework to figure out cause and effect relationships concerning multi-task neural networks. What this means in practice is that these people need to be able to vary the changeable variables in the system and evaluate the results. This framework should make this easier to vary one parameter modularly and provide quick and easy ways to visualize the results. Another thing is that the framework should provide opportunity for reproducibility of results.

3.2.2 Developers

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

Developers are the group of people that need to be able to build solutions for a given task as quick and easy as possible, but with opportunity to extend the framework in the places that likely can change. This necessitates that often used features - which take up needless time to develop - in a deep learning pipeline are already available and easy to use. This group likely models a system like this with the intension of deploying or executing on a different device, while their working device might be resource constrained. This both means that the framework needs to take usual resource bottlenecks in account as well as facilitate transitioning the execution environment to a different system.

3.2.3 Newcomers

The final group is the crowd for whom the framework and possibly the multi-task learning paradigm are new. For them, the framework structure needs to

be comprehensible as well as offer some developmental railings to help them avoid problems. The framework should have a clear workflow and provide enough guiding for implementing new pipelines correctly through providing type checks and examples. While this framework's intended purpose is not to educate the user on how multi-task learning works, it should provide assistance to lessen or track potential issues.

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

Previous work done by Roberts et al. [1996] informally describes the requirement for frameworks being "simple enough to be learned, yet must provide enough features that it can be used quickly and hooks for the features that are likely to change". The goal of this platform is to facilitate research and development of deep learning multi-task algorithms for audio recognition purposes. This frameworks is built on top of the PyTorch library that already offers comprehensible and easy to use tools for developing deep learning models. However, this extension looks to alleviate the pain points the multi-task paradigm brings with it: extracting multiple datasets and tasks and combining them to train and test a single model.

The truth of the matter is that performing research requires changing a lot of variables in the process of deep learning and reporting on the outcomes. For multi-task learning however, the work required for implementing the changes can quickly scale with the amount of datasets, the amount of tasks and the amount of elements that need to be varied. Not only does the extra amount of input cause a lot of unnecessary double work, but each difference in the individual tasks and datasets can cause problems further down the line when combining. Thus the main idea is to offer a pipeline pattern where each individual step can be filled in and tinkered with, without having to worry about previous or next parts in the pipeline breaking. For this purpose, it builds on the groundworks from Pytorch to provide the deep learning tools, while focusing on standardizing input data, anticipated handling of possible variations and offer often used features in acoustic deep learning.

As a basis for developing the framework, following the example set in a different framework De Rainville et al. [2012], a number of hypotheses were made about the usage of the system. These are as follows:

Hypothesis 1. *The user will need to vary parts of the pipeline. These parts should be easily interchangeable and cause little to no problems in the rest of the system when changed. Furthermore, the framework should be ready for quick iteration on top of previous results, as well as the need to compare these iterations.*

Hypothesis 2. *Every dataset is different, while every model needs similar inputs. No assumptions should be made about the structure of the datasets, but the user should be able to store the data in a structure that is guaranteed to be valid. The structure should be robust enough to deal with variations in the dataset, without having to alter its behaviour. The user knows best how to navigate the dataset's structure in order to extract the necessary information.*

Hypothesis 3. *Speed of developing pipelines is more important than speed of execution of the result. Clarity and simplicity are important for designing the framework. This tool is meant to help developers create the best model. The creation can be reimplemented in another language for optimal resource efficiency.*

Hypothesis 4. *Not every possible feature can be covered beforehand. If the user is in need of a different functionality in a certain part, they should be able to implement their own solution and plug it in easily.*

Hypothesis 5. *Optimal resource usage is not required, but the system should be executable. Concatenating multiple datasets means more space is required and more time will be needed to execute. The framework should assume the entire concatenated dataset possibly can not fit in memory and device failures can happen while executing, which should not automatically require a restart of the entire process.*

3.4 Non-functional Requirements

From the observations made in the previous sections, a number of non-functional requirements have been drawn up. These requirements are goals for the design of the software framework. The requirements are as follows:

- **Modular:** The framework aims to provide a helpful tool to build deep learning multi-task pipelines, for which the individual parts of the pipeline are likely to be tinkered with in order to develop optimal solutions. The different components should be modifiable and be interchangeable independently from the rest of the components. A developer should only have to worry about one part of the pipeline at a time, without having to worry about disruptions further down.
- **Extendible:** The framework should provide open hooks for features and functionalities that likely require change.
- **Fast prototyping:** Developers using the framework should be met with an environment that provides them with the tools necessary to develop their own multi task pipelines fast.
- **Cutting Double Work:** Anticipate that multi-task models will be designed through iterated variations and that the system can be run with largely the same variables without having to recalculate the same things as before.
- **Developmental Side-rails:** New developers using the framework should get a clear development structure and guiding through examples, warnings and hints.
- **Robust:** The framework should be able to dynamically handle possible differences in input and desired pipeline functionalities.
- **Portable:** The developed code should be transferable to different devices with possibly more constrained resources.

3.5 Functional Requirements

The framework is a tool for building Deep Learning Multi-task pipelines, which this work distributes in three steps. The first step is the data reading.

In this step, the raw datasets are read, features extracted from the instances and the results stored in objects which will serve as the input for the rest of the system. Then, the data loading happens, in which the multiple separate objects are prepared for the specific training set-up, combined and then served to train and evaluate the model. In the last step, training, the model is created, the data instances go through the model, the loss for each task gets calculated and combined which is used to optimize the model and the optimized model gets evaluated.

Further note should be made of the difference between datasets and tasks. A dataset can have multiple tasks and a task can have multiple datasets. A task is in essence the learning objective for the input and comes in different forms for the target labels. A data instance can belong to only one of two classes (binary tasks), only one of multiple classes (multi-class tasks), multiple classes at once (multi-label tasks) or have a continuous value for a class (regression tasks). The dataset is the collection of data instances that can be linked to the targets. The framework must deal with the fact that there can be multiple datasets and multiple tasks in a many to many relationship and that each can require different handling.

The functional requirements will be grouped along the steps in the pipeline, keeping the non-functional requirements in mind and determining what is necessary to allow multiple different datasets and tasks.

3.5.1 Data Reading

The functional requirements for reading the data to standardized objects are as follows:

- Standardizing input: The developer must be able to read the data from the datasets to standardized objects which will always be valid and function in the rest of the process, so that they only need to worry about extracting the data. These objects must be versatile enough to deal with any dataset and be able to be combined with other standardized inputs. Furthermore, the standardized object must be able to contain multiple tasks and those tasks must be able to be contained in multiple dataset objects. Finally methods must be available for aid in the extraction of features and creation of valid objects.

- Handling dataset differences: Datasets can come in various structures and storage forms. The developer must have the power to navigate the dataset structure and extract the data to the required form on their own, but the system must have the capability of dealing with different ways the data is stringed together. Datasets can have predefined train and test sets, which have to be combinable with datasets that have to be split later. The system thus must get these two cases in a unified form to achieve modularity where other parts handling the standardized objects don't have to differentiate between the two cases once the standardized object is made. Same goes for datasets that have pre-split audio segments. These belong together and should not be separated later on.

- Scalable preprocessing: It is often the case that input data must enter the system as if they are the same input. This means having the same preproccesing as well as sample rate for audio. To cut on useless double work, the system must provide with easy possibility to replicate the same preprocessing for each dataset.

- File storage abstraction: Saving, reading and checking files require repetitive work for multiple datsets, especially if it's the case that datasets must be extracted multiple times to vary for research, which requires saving to different files. The system can take workload this off the developer's hands for the standardized objects as well as further files that need to be written and read.

- Quick Reading: In order to vary quickly and not have to either extract the entire dataset each time or have to enter the location of the desired stored dataset, the framework must provide a function that reads the correct file automatically when the data is read.

3.5.2 Data Loading

Combining and loading the data for training has the following functional requirements:

- Combining datasets: The framework must provide a way to combine different datasets in standardized objects that the training function can take instances from and derive predictions for the multiple tasks.

- Not requiring the combined datasets in memory: Computer memory on numerous devices is likely not large enough to hold multiple datasets at the same time. In this case, the framework must provide away around this in order to make multi-task learning possible, without having to treat the standardized object differently.

- Train and test set generation: Train and test sets most likely have to be created from the original dataset. The framework must provide an easy way to generate these for the combined datasets for datasets that both have and don't have predefined sets.

- Transforming data: Scaling and windowing functions for the input matrices must be available so the developer should only have to deal with the specifics of what methods to use for these and the parameters.

- Filtering data: Research in deep learning often deals with adjusting the distribution of instances with certain labels in a dataset, for which the framework should be able to provide a filtering method.

- Reusing data: Data is likely to be reused and reiterated over with different transformations and such applied. The system must be prepared for this and only store the base extracted feature matrices without any of the subsequent adjustments.

- Batching multiple tasks: Batches of input are done matrices that append inputs from different datasets and targets from different tasks together. These inputs and targets must have the same shape in order to be able to fit together in a matrix. The framework must provide for this instance automatic functions that make this possible, for the task targets. For inputs however, they either have to be cut or padded to the same shape, so the developer must have the tools available to achieve this.

- Replicability: An important part in research is the ability to replicate the results. Any randomness based operations the system adds must come with pseudo random number generators that make it possible to receive the same output every time.

3.5.3 Training

Training and evaluation based on the batches of input data deal with predicting results from the model and using those predictions to calculate the error margin, optimizing the model parameters and outputting metrics. The training part of the pipeline has these functional requirements:

- Predicting multiple tasks: The framework must be able to predict the targets of multiple tasks for each data instance.
- Task specific output handling: The developer must have the ability to define the handling of the prediction output for each task. This should be easily integrated and extendible for the desired handling. This includes the loss calculation but also any other task-specific metric calculations.
- Loss combination specifiable: The way the different losses are then combined to one single loss by which to update the system should be definable by the developer.
- Metric calculation, storage and visualization: Metrics are different evaluation criteria based on the predicted output labels compared to the true output labels. Calculating and inspecting these are a crucial of research, so the framework must provide an easy way of doing so in which it is also possible to compare the results to previous ones. Furthermore, the developer must also be able to extend these with their own implementation and additions.
- Interrupted Learning: Sometimes a training run can fail or be stopped in the middle of its execution. This is more likely when the running time is longer due to the increase in inputs from combining multiple datasets. To deal with this the framework must provide a feature called interrupted learning, in which a training run can restart where it left last time around.
- Separate evaluation: To follow the line of modularity, the system must not assume that training and evaluation will always happen together, but a developer can use the system to simply evaluate a model or a previous training run. Therefore it must be able to evaluate models and historical runs without much hassle.

Chapter 4

System Design

4.1 Framework Design

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

Development frameworks have been defined by Roberts et al. [1996] as "reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate". The bread and butter of framework design is developing abstractions that cut usual required work, while identifying the points where an implementing application likely requires to change. These variable points in an application domain are called hot-spots Schmid [1997]. In the context of a framework, these are the points where the software allows to plug in application specific classes or subsystems. These hot-spots come in two forms: white-box and black-box hot-spots. White-box hot-spots require programming the plugged in blocks of code, while black-box hot-spots give the option to select from pre-implemented solutions.

Expanding existing ground work for deep learning systems with abstractions for developing and researching multi-task learning set-ups, means that the focus will be on providing abstractions cutting effort of combining datasets and tasks, but also that providing exhaustive possibilities are nearly impossible. Therefore, the design will only offer white-box solutions, but with a number of pre-made implementations addressing commonly available features in deep learning.

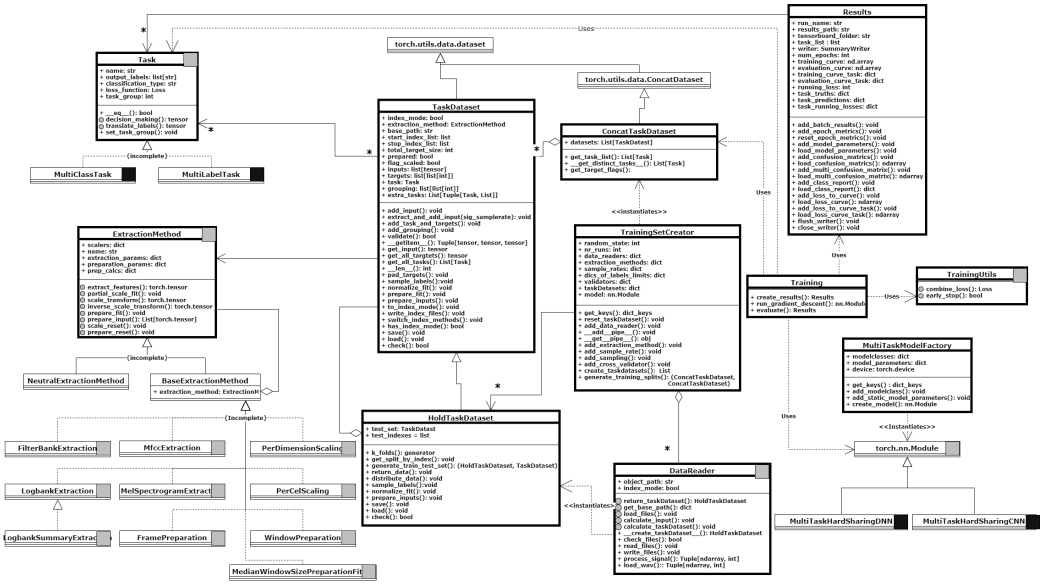


Figure 4.1: Class Diagram

The nature of variability in the envisioned use is not static for a developed application. Researching multi-task set-ups requires implementing and executing multiple variations and comparing their results. The hot-spots thus have to be variable at run-time while ensuring the validity of the pipeline and differentiability of the different instantiations. Furthermore, in the interest of fast development, every variation in dataset handling must both be able to be implemented for one specific dataset or implemented for all at once.

With these characteristics of variations in the framework in mind, the design UML is drawn up following the design principles from Bouassida et al. [2001]. The definitions of white-box and black-box hot-spots differ slightly, in that they do not go for all descendants here. In stead black-box hot-spots cover classes with predefined code that can change their functionality based on allowed input variables for the class, yet should not be modified in code. The reason for this change is that in the original definitions complete class hierarchies where either white-box or black-box hot-spots where the classes and all of its inheritants either contained default code -and the desired implementation should be chosen from available options- or it did not. This is

too rigid as it only allows extra extensions in a class if there is no default code.

In figure 4.1 is the framework's extended UML class diagram. The explanation of the extra annotations goes as follows:

1. Classes with a grey box in the top indicate that these classes compose a white-box hot-spot. These classes require an extending implementation from the developer that can expand the original methods defined in the class.

2. Classes with a black box in the top indicate that these classes are black-box hot-spots. These have default code present in their implementation and should not be modified.

3. Grey circles in front of methods signify functions that change from one implementation to another. These are abstract methods that express variation points.

4. Generalization relationships marked incomplete have base implementations that already have pre-made inheritants, but can be extended with extra classes.

5. Highlighted borders signify that a class belongs to the program's core. The core are the 'frozen' parts of the system, which will remain unchanged in implementing applications made with the framework.

Going into the model overview, without going into the details and inner workings, a structural explanation will be given. First thing to note is the central class TaskDatset, which is the encompassing object that standardizes the data and ensures its validity as input for the rest of the system. This is connected to possibly multiple Task objects which keeps task related information, separated from the dataset itself. Datasets can have multiple tasks and the same tasks can be found in multiple datasets. It also has an ExtractionMethod object which decouples all data transformation implementations.

On top of the TaskDatasets are two classes where it functions as a component in a composition structure. These are the HoldTaskDataset - which adds training set splitting and managing functionalities - and the ConcatTaskDataset - which adds the possibility to combine other datasets.

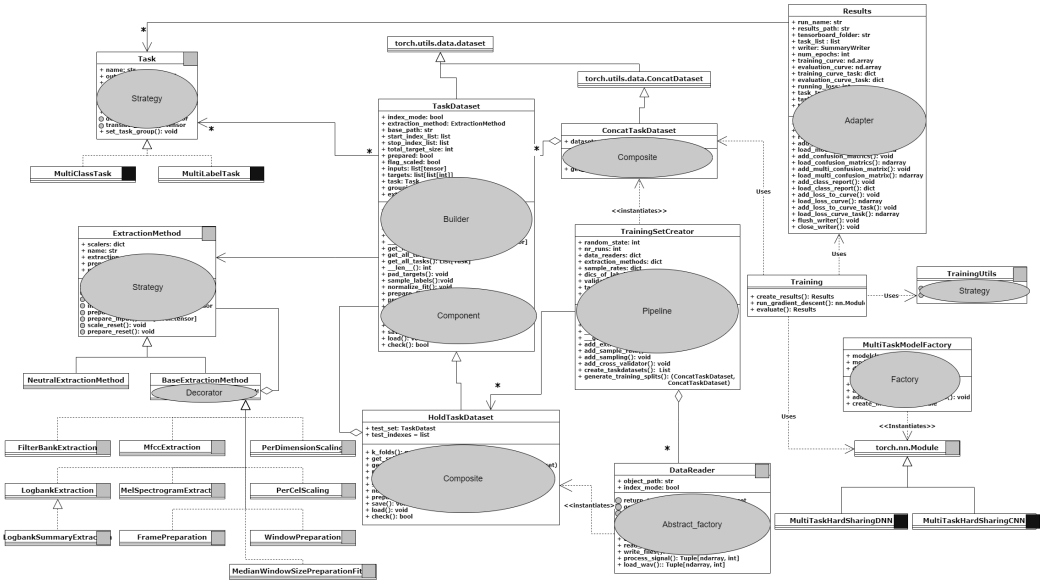


Figure 4.2: The framework's pattern diagram

Alongside these, two creating classes can be found: the DataReader and the TrainingSetCreator. The DataReader has a grey square as it should be extended for every dataset implemented. This transitions a raw dataset to a standardized object. The TrainingSetCreator then take the different standardized objects, transforms and combines them into valid input for the training and evaluation functionalities.

This leads to the final section of the model, which is centered around the Training class. This in turn requires 4 classes to function: The ConcatTaskDataset for its input data, torch's nn Module for the Multi-Task model, a Results object for calculating and storing results and a TrainingUtils object for decoupling some of its methods and allowing their variation. While Task objects are contained in the TaskDataset, there is still a line drawn from the Training class, as these objects contain data that can change Task dependent functionalities in the Training class.

To further clarify the roles played by the classes in the network, a pattern diagram following the example in Bouassida et al. [2001]. Two patterns - the composite patterns the ConcatTaskDataset and the HoldTaskDataset creates

with the TaskDataset - are already mentioned above. The Task and the ExtractionMethod classes are also obvious strategy patterns as they decouple implementations from the original classes. The ExtractionMethod accumulates a few data transformations, which can be matched together through its implementing Decorator pattern.

How these classes function and why these methodologies were chosen will be explained further in the section, but two classes should be still highlighted in the overview. One is the TaskDataset which also has a builder role. This includes adding partmental creator functionalities, which are used here to ensure that a TaskDataset is correctly implemented as well as facilitating the process of building such complex object. The other is the Pipeline pattern of the TrainingSetCreator. This creates a sequence of operations that lead from the raw datasets to combined inputs for a specific training set-up. The Pipeline pattern is what allows every step up to training to be scalable, modularly variable and correctly executed.

One guiding design rule for this framework was that no class should assume anything about the underlying implementation of another. This is to ensure modularity in the system. Given that the framework is a white-box implementation, most developmental speed-up is made by letting the developer only focus on one class per responsibility and giving the ability to execute the same parallel operations as if they were performed on one object. The overall design envisions the whole pipeline from datasets to trained models, but these can be picked apart according to the developer's needs. A TaskDataset will always be valid input for training and evaluation, no matter if it came from a DataReader. A Results object can always write data to and read data from files, no matter if it happens in training.

4.2 Assumptions

Before resuming to the detailed explanations of the operation designs, a small summary of the design assumptions made is given:

- The device is unlikely to hold many datasets in memory at once, yet for systems that can, this might still be desirable for the speed-up it can provide.

- The model creation is central and developers should be able to build pytorch models unrestrained.
- The training and evaluation loops should not required modification, as they are static for all (gradient descent based) applications, but they should take variable functionalities.
- Hot spots for variation will likely have to be varied at each point within the runtime itself
- Preprocessing will have to be both easily made the same for multiple datasets as have an individual process for each one

4.3 Creating standardized objects

One of the most important aspects of this framework is the addition of standardized objects to encapsulate the input. These allow for easy manipulation and combination of different datasets. The objective is that once created, the developer should not have to worry about them further down the pipeline.

The standardized object is called the TaskDataset. This contains the feature matrices extracted from the audio files in the dataset, their target labels which form the ground truths and their corresponding task information. Creating these objects from the raw datasets is a complex task which requires the navigation of their storage structure, that is specific for each dataset. Developers should thus be responsible for getting the correct data out of the dataset but should get proper assurance the object was created correctly.

The design choice was thus made to create an abstract factory the developer has to extend, loaded with the necessary tools to insert the data correctly. In order to provide the necessary assurances the object is created correctly as well as facilitate the process, the TaskDataset is loaded with builder functions. Through these the TaskDataset can be filled step-by-step. This further achieves two things. One, because the functions are in the TaskDataset itself, *no outside classes have to make any assumptions about internal data structure or workings of the TaskDataset.* This ensures modularity and extendibility of the TaskDataset and will be a constraint held for

the rest of the design. Two, because datasets can be too large to be loaded in the system's working memory, this way each instance can be inserted one by one in the data object.

Back to the abstract factory - the DataReader - where the abstract methods are called in a predefined pattern, so the developer only has to extend the required methods and adhere to their required responsibilities. To aid with extraction and processing the sound files from different datasets in the same way, this class also has a processing function for the numerical time series representing the audio. Included here are abilities like resample audio to the same rate and converting multi-channel audio to a single channel. The developer themselves can choose to use this or not. Along with creating the standardized object, the factory also includes quick saving and reading of the created object for the specific method of extraction. This makes it more feasible to quickly reload the dataset without having to redo the extraction.

The structure for storing data in the TaskDataset itself then is as follows. Since in audio every instance likely has a different size due to varying audio lengths, the collection of input matrices are stored as tensors in a Python list. Each target then is stored as a list of integers. Usually they are encoded as binary strings with a 1 in each column number that an instance has a corresponding label for. The target labels in the order which relates to the column numbers and the rest of the task information is recorded in a Task object which is stored in the TaskDataset as well.

4.3.1 Variations

The default assumption is that a dataset has a single list of unrelated inputs with a target label for each instance. This section will explain how variations on this can also be contained in the TaskDataset. First variation are datasets which have separated predefined training and test sets. These will have to be combined with datasets that have to be split afterwards. Both training and test dataset have to receive the same preprocessing, so to enforce this their objects have to be somehow linked. This is done through making the a composite object that is a TaskDataset which contains a test TaskDataset. The test data can then be stored in this test object inside the composite object, while sharing the same preprocessing functions as well as appropri-

ately handling called functions on it to allow it to be treated uniformly to a dataset without presplit data.

Next variation are datasets that have target data for multiple tasks. While one task with according targets is required, the rest can be added in a list of tuples of task objects and a list of targets same as for the 'main' task.

The last one are datasets that have "groupings". What is meant here are datasets consisting of audio files that are already split, with each part having their own ground truth, but cannot be divided later (e.g. parts of the same split going to the training set and the test set). For this instance, the same formality is used as in sklearn's splitting functions and a grouping list can be stored where the index corresponding to each instance belonging together contains the same number.

The objective is to treat the resulting TaskDataset uniformly regardless of variation. The strategy for handling each of these cases will be discussed in the appropriate sections later.

4.4 Manipulating datasets

As mentioned before the created standardized objects are also responsible for providing easy manipulation abilities for datasets. The choice was made to include dataset changing handles on the object itself, which can be called by other classes, without having them know the underlying representation of the object itself. These handles fall into two categories: functions that transform data instances and functions that change the collection itself.

The functions that transform instances are the most likely to change, as these have to be tweaked based on their effect on performance of the resulting recognition system. What we understand under these are data scaling methods, windowing functions, but also feature extraction as this transforms the original time series representation to one fit for the required learning task. The data scaling and other possibly required preparation functions will depend on the feature extraction method used. For example, a feature extraction method which outputs a time dependent representation will re-

quire a windowing function that cuts the feature matrices to the same shape in order to fit them in the same batches, while one that outputs matrices of the same shape will not. To adress this along with the likelihood of change, the instance transformation functions are encapsulated in an object called the Extraction_Method.

This is an abstract template that developers have to extend in order to define their required implementation. Parameters for these implementations can be given on the fly as input to instantiate these objects. The whole object is then stored in return as input in the TaskDataset which will then use it when the call is made to transform the data. The extraction_method object has a decorator inheritant, with a number of premade classes that already implement numerous transformation functions, as can be seen in figure 4.1.

The extracion_method object accumulates a number of transformations that depend on which extraction method is used. These can be categorized as feature extractors, scalers, preparation fitters and preparation executors. Feature extractors create a feature matrix tensor from a given time series representing an audio file. Scalers include calculating metrics for scaling and then using those to execute the scaling of the data. Preparation fitters and executors are separated. Executors mainly concise of framing and segmenting operations which transform the data in matrices of the same size. Fitters calculate the parameters for these operations, but these operations can always be executed with parameters given by the user.

The Extraction_Method object has another hidden function which lies in its name variable. The TaskDataset uses this object's name to store its extracted feature matrices to files. This happens in the interest of being able to quickly reiterate and compare different variations. The extraction method itself also gets stored, to allow for recreating the input at a different time than the original extraction and transformations were done.

Functions that change the collection itself then are ones that filter and split the data. For these ones it does not matter what is contained in the actual instances, but simply how the collection is composed. Filtering the data instances - based on their associated labels - is present to offer the ability of adjusting the label distribution. This happens through taking a (pseudo)random sample of data instances with the associated label of the

defined size the developer wants the (maximum) amount of labels to be. The rest then gets filtered out of the current dataset object.

Splitting the dataset then in a train and test set requires a more sophisticated strategy. Because both require that the same transformations are performed, with some additional constraints (see further), it is opted for creating a composite object which both is a TaskDataset and contains a TaskDataset that is the test set. Splitting the dataset into a train and test set is performed then by filling this test set with data split from the original, while both share the same object - with the same parameters - for performing transformations. As mentioned before, there are datasets which have test sets defined beforehand. The difference between these two situations is thus that the developer fills this test dataset beforehand. At the point where a dataset has a filled test set, both of these situations thus become the same again for the rest of the process.

Creating training and test sets themselves do not simply happen at random either. The default case is that these get a stratified collection, meaning that the resulting folds will try to stick as close to the original label distribution as possible. Mentioning folds gives away that the splitting is not simply meant for one time use. The splitting into train and test sets actually contains three operations. First is splitting the data into k equally sized folds based on their indices. Second takes indices and transforms them into a train and test TaskDataset, while returning previous data to the original set. Third simply handles the difference between predefined sets and sets where the splits still needs to happen, calling the previous functions in the latter case.

4.5 Combining Datasets

When the datasets are created and prepared for training, they need to be combined as a single dataset, but from which the instances can be processed differently per task in training and evaluation. The solution for this is another composite object called the ConcatTaskDataset. Built on the groundwork from PyTorch's ConcatDataset, this class contains a list of TaskDatasets and builds a front for the data loading in training making it seem like one

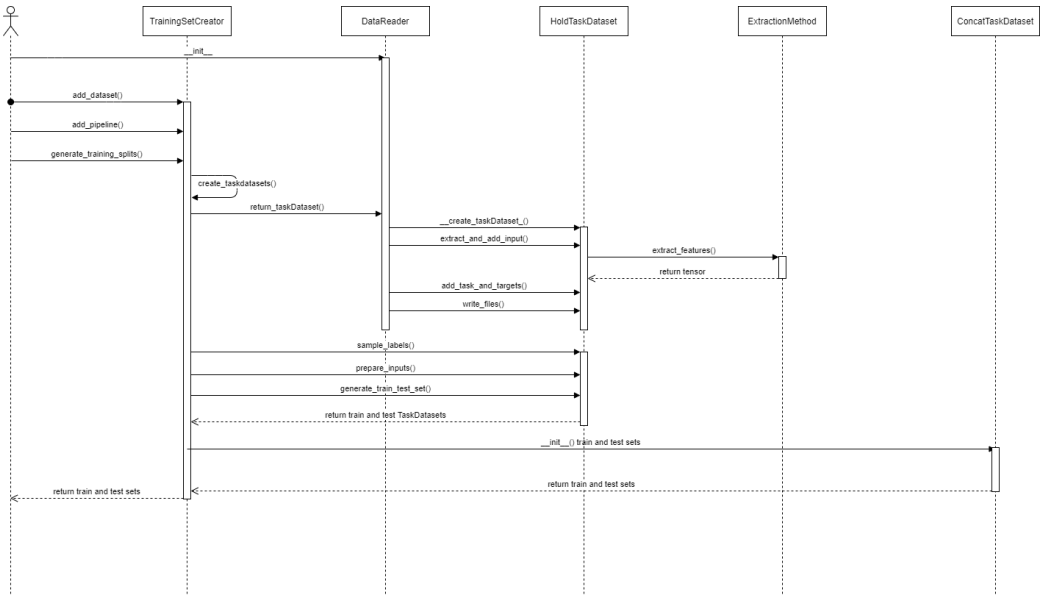


Figure 4.3: Sequence diagram illustrating how the TrainingSetCreator extracts the data from datasets to training and test sets

big dataset. Alongside this, the class contains methods for differentiating the different tasks afterwards. At instantiation every individual dataset gets loaded with information to combine the target vectors into one. More detailed information on the exact problems that need to be addressed and how it achieves this are given in section 5.4.

To illustrate the execution flow of creating the input for training and evaluation a sequence diagram is given in figure 4.3. In the above sections the designs are detailed for the individual handling of the datasets. This diagram illustrates how the TrainingSetCreator brings the operations together by functioning the handles present on the above classes. As mentioned in 4.1, this class is a pipeline pattern where dataset and manipulations can be optionally added and scaled to individual or all TaskDatasets at the same time.

Not only does this add modularly replaceable sections in the handling of data, the TaskDataset objects are reset every time a new pipeline piece gets added. This allows modifying the creation of a TaskDataset while keeping

the ones that do not need to be altered. Imagine the situation where multiple TaskDatasets are created and combined, when the developer wants to alter the creation of one of them. In stead of having to recreate all the objects, only one of them is reset and remade.

4.6 Creating Model

This framework is aimed at providing a facilitating extension to PyTorch for training and testing multi-head hard parameter sharing models. Model creation thus puts as little requirements on designing the actual models as possible. The rest of the system builds on Pytorch's classes as to ensure that the input would be valid for any PyTorch module. The only presumption the system has to make is that the output is a tuple with each entry being the predicted results for each task in the order of the task list from the input dataset. While no class was made that ensures the developer does this correctly - with the model creation being central, the developer should have the freedom to create whatever they want, adjusting the functionalities around it if necessary - there are two base multi-head networks, a CNN and a DNN, available which has variable layers based on the input. These also already include adaptations based on the type of task. In this area the mantra is that the developer knows best.

4.7 Training and Evaluating Model

When the model is designed and the input dataset is created, they will be inserted into the method which trains and/or evaluates a model. These two are similar, with the only exception that the training function updates the model parameters every loop. Training (and evaluating) a model happens through a method which should remain static, but can vary its functionalities based on input. The reason for this is that the framework can only ensure the validity of the created pipeline if it controls how inputs are received and outputs are processed. In other words if it knows what it is going to do with the received data.

The handling of objects and data which the rest of the system does not rely on however, can be overwritten in numerous ways, depending on the required change. These are all discussed in section 5.7. Every batch statistics are calculated from comparing the model's results to the ground truths. This happens through an adapter class that takes the predicted results and is responsible for writing them to the correct files. This class, the Results object, is wired to write results already to TensorBoard. This library provides easy visualization for deep learning metrics. This happens live, both in the training and evaluation loop, so that developer can follow the progress and intervene during the loop if necessary. The developer can easily extend this class and define their own desired metric calculations, as the results only receive the ground truths, predictions and the loss.

Alongside metric calculation and storage, the Results object also provides checkpointing function for the model's parameters. This makes it possible to retrieve a model's previous states. The Results object thus acts as a unique adapter for each training run, managing and distributing files resulting from training and evaluation. Because of this design choice, it is possible to continue or restart previous runs by instantiating the same Results object and giving it to the "blind" training and evaluation functions.

Especially the evaluation function dynamically uses this, as it can either take a model object or load up the model from each epoch if none was given. This simple aspect allows a developer to use the system as a testing framework for trained created models if they need to.

Changing non-metric related functions - like early stopping - can be done by another extra object which can be given as input, namely the Training_Utils object. This simply contains functions, for which the developer can write an extending object and give as input.

4.8 Three Implementations

With the base strategies in the pipeline designed, it might seem complex without a clear reason why its abstractions were built. This section will give insight to how the framework was developed alongside giving clear examples

of how the previous designs are instantiated.

The designing process for a framework is broken down by Roberts et al. [1996] as follows. Frameworks are reusable software patterns that facilitate the development of applications. Determining the correct abstractions must come from concrete examples, as it is nearly impossible to have to foresight to address the required functionalities from simply the domain. A number of abstractions do not become apparent until the framework has been reused. Generalizable solutions can only come from actually building the applications.

What [Roberts et al., 1996] propose as a simple step by step plan is to build three applications in the same problem domain which differ from each other each time. While the more applications get developed lead to a more generalizable framework, there has to be a cut off point as too many applications can make it impossible to actually finish the work. That being said, a framework is likely to continue to evolve after the three applications are made. What follows now is an explanation of the three applications that shaped the design of the framework, ending with conclusions drawn from them and further extensions.

Choice

First however, the choice of projects must be decided. Each of them will be acoustic multi-task classifiers with different requirements for the system. The focus for this platform is research and iteratively designing the best performing multi-task system. Therefore, the first two implementations are following two research papers. This choice also adds an opportunity to evaluate the system by comparing to the reported results. Generalizing for multi-task purposes means that the choices must cover enough datasets, tasks and data processing features. Implementing existing research can be helpful here for discovering possible set-ups, one must take in account that this is completed work. The process of discovery and improvement of systems is not usually covered in the resulting paper. Therefore, the last implementation actually went into trying to develop new functioning set-up for multi-task research .

Low-resource Multi-task Audio Sensing for Mobile

and Embedded Devices via Shared Deep Neural Network Representations

In the work done by [Georgiev et al., 2017], a Deep Neural Network is developed which tries to create a general purpose audio task model that addresses computational limitations of mobile, embedded and IOT devices. The approach is to bring 4 separate audio tasks together in one deep learning framework, for which they try out different configurations. Every task is tested combined in a multi-head network as well as separated after which they evaluate the effect on performance for each of them. The model that combined each in a single multi-head model was not significantly worse than the best performing one, making the multi-task set-up a viable way to reduce resource requirements.

The chosen tasks here were Speaker Identification, Emotion Recognition, Stress Detection and Acoustic Scene Recognition. These are three background identification tasks, meaning they don't actually require to know what happens in an audio fragment and can take a longer time frame for labeling. To rebuild the actual set-up this work utilised the ASVSpoof database TODO: REFERENCE for the Speaker Identification task, the Ravdess database TODO: REFERENCE for emotional speech recognition task and the DCASE 2017 Acoustic Scene dataset TODO: Reference for the Acoustic Scene Recognition task. However, the authors used a self made dataset for the stress detection for which no suitable replacement was found.

For extracting features they utilised both MFCC TODO: REFERENCE and their own created summary of filter banks. This summary consists of creating different statistical aggregation metrics (e.g. the mean, the standard deviation, the median, ...) per coefficient from extracted log filter banks TODO: Reference. This creates a representation from an audio sample independent of time and requiring less space.

For design purposes, the multi-task pipeline is examined. The system is built for low quality audio with a sample rate of 8 kHZ. From the audio MFCC or the logbank summaries are extracted. Each task has a different dataset but all the inputs require the same extraction. The resulting matrices are then normalized using the mean and the standard deviation from the numerical data. This is then used as input for training the model, which is

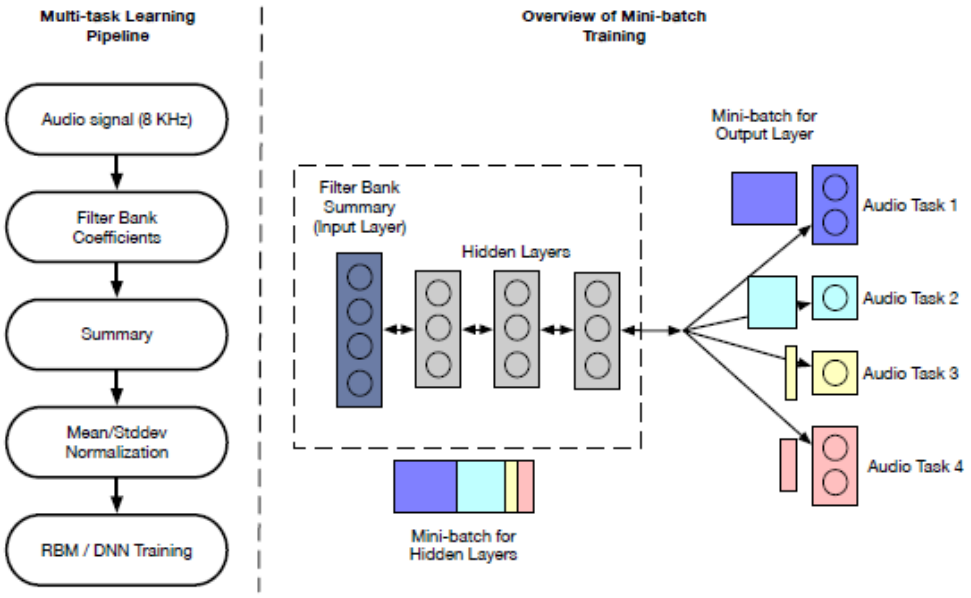


Figure 4.4: The Multi-Task pipeline and Model used in Georgiev et al. [2017]

done using gradient descent. The data is fed to the training module in stratified batches. In this context, it means that a batch contains samples from every dataset, with the amount of samples from every dataset matching internal size ratios of the datasets. The summary of this can be found in figure 4.4

A few lessons have been made from recreating this exact set-up.

- Every audio sample should be able to be resampled which will likely happen at all the datasets at once
- Feature extraction as well as any other preprocessing will have to be easily replicable for every dataset at once
- Features can be time related or not. If the dataset has varying time lengths then the resulting feature matrices will be of varying lengths. These can not be batched together for data loading. The solution for this is to include an operation that can transform feature matrices to the same size in a windowing function.

- Depending on the desired batching method, it is possible that data instances from all datasets and tasks have to go in the same batch. This does not only require that input instances have to have the same size, but also that target vectors can go in a unified matrix structure.
- Normalization using the mean and the standard deviation happens differently for these two feature extraction methods. In MFCC, the mean and standard deviation is calculated per one dimension in the feature matrix: the coefficients. This makes sense as the second dimension are time steps, so the numerical distribution will be from the same domain. However for the statistical summary matrices, every value in the second dimension is a different metric. Mean and standard deviation calculation must happen in this instance per cell basis. In essence for the system, this means that both these scaling methods need to be covered, but also that normalization (and possibly any other transformation operation like windowing) depend on the extraction method used.
- The DCASE dataset has a predefined train and test set. The other two do not. The system has to be able to store both of these instances in the same standardized object as well as combine them easily and take them in account when generating test sets for the other case.

In figures 4.5 and 4.6 two object models depicting instantiations of the set-up. In the first image is the situation where the TrainingSetCreator executed the TaskDataset generation method in each DataReader class. This illustrates how a Dataset which has a predefined test set is combined with the others which do not, while all of them have the same transformations in the ExtractionMethod object. The ExtractionMethod object then is composed of a LogbankSummaryExtraction, a PerCellScaling, a FramePreparation and a MedianWindowSizePreparationFitter object, accumulated in the LogbankSummaryExtractionMethod. A step-by-step explanation goes as follows:

The LogbankSummaryMethod extracts summaries of logbank filters as explained in the paper. PerCellScaling standardizes the data by calculating the mean and standard deviation of every separate cell aggregated from every feature matrix in the dataset. These are then used to achieve a per cell data distribution with a mean of 0 and a deviation of 1. FramePreparation cuts

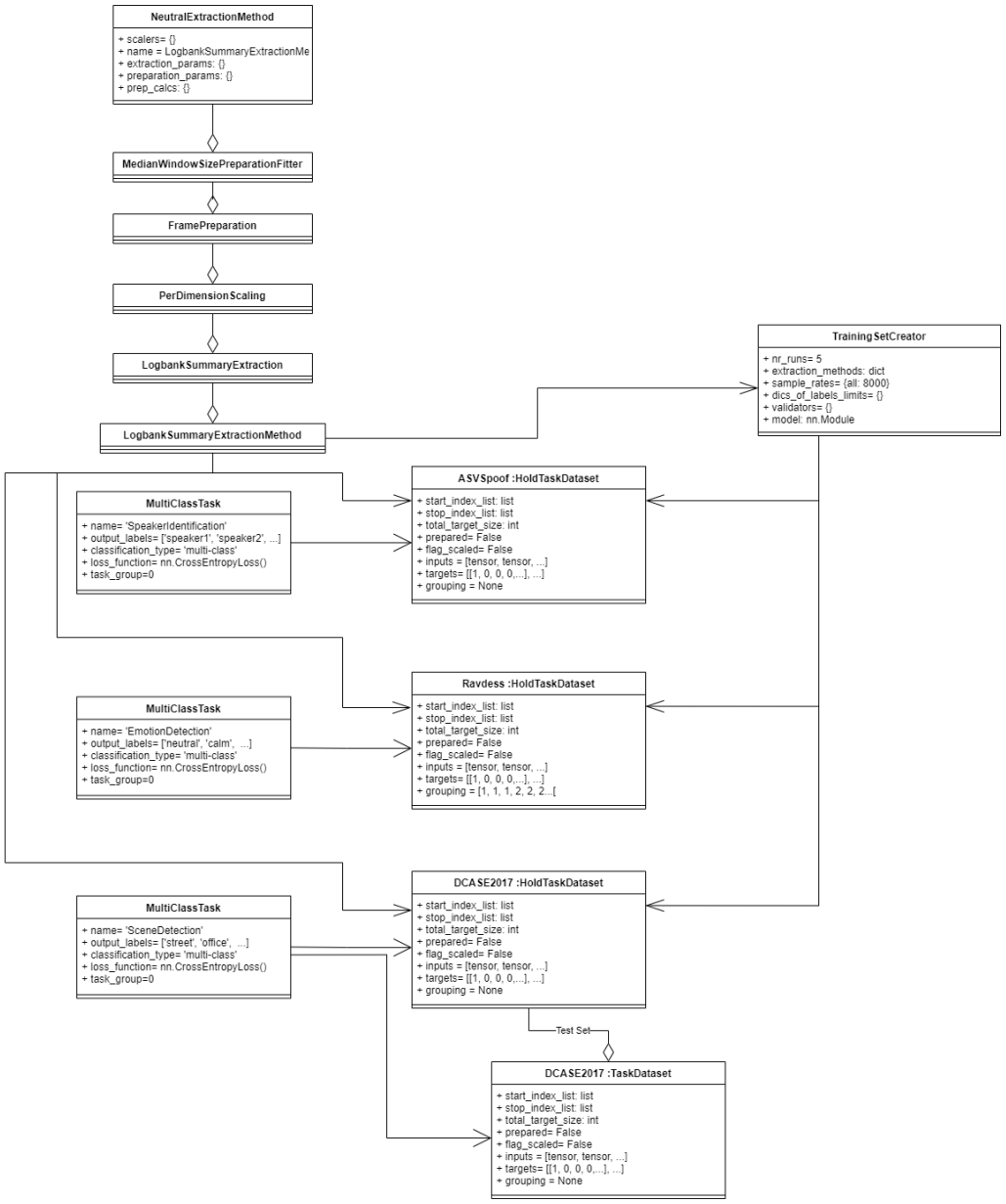


Figure 4.5: Object model instantiating the set-up from Georgiev et al. [2017] after the TaskDatasets are created

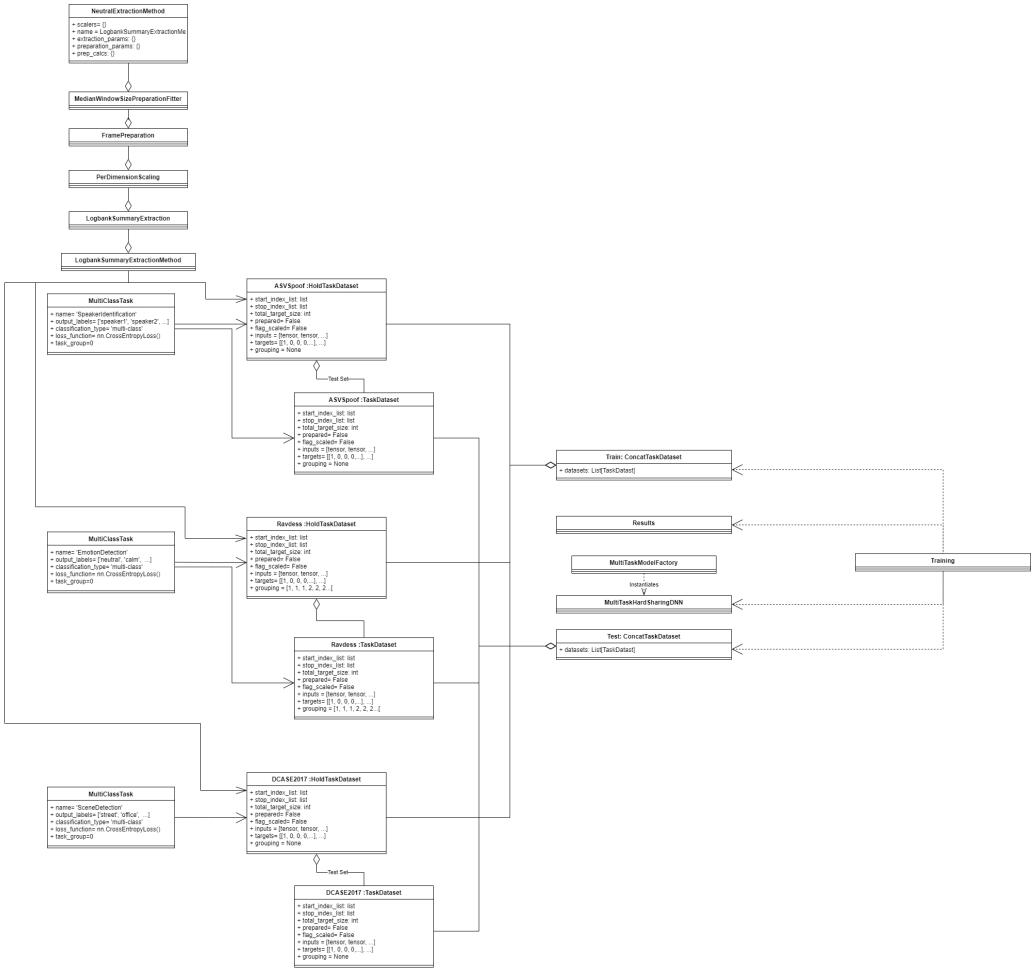


Figure 4.6: Object model instantiating the set-up from Georgiev et al. [2017] after the train and test sets are created along with the input objects for the Training

the matrices to frames of the same window size. This size is then decided by calculating the median matrix size in the dataset in the MedianWindowSizePreparationFitter.

In the next object model, it is demonstrated then what the situation is after the other datasets have their testDatasets split off and how these are accumulated in their concatenated train and test sets. These, along with a Results object and Multi-Task model form the input for training.

A Multi-task Learning Approach Based on Convolutional Neural Network for Acoustic Scene Classification

In this paper, by Xu et al. [2019], an Acoustic Event Detection (AED) task and a Acoustic Scene Recognition (ASR) task are put together in the same multi task framework. Here a Convolutional Neural Network is used for classification with AED being an auxiliary task for improving the ASR task. The training data for the ASR task comes from the DCASE 2017 acoustic scene dataset while for the AED task it comes from the DCASE 2017 sound event dataset. The evaluation data comes from their respective predefined evaluation datasets. The evaluation metric is not simply accuracy which was the case in the previous work but Unweighted Average Recall TODO: Reference.

Again, a number of lessons have been made from recreating this set-up:

- This work has two predefined test sets, yet only one is used for evaluation as the focus is only one of the two tasks. Developers need to simply be able to control what goes in the training set and the test set separately, so training and evaluation can happen independently. On the same note, the framework should not only be made for combined datasets but be as receptive for single datasets.
- ASR and AED function differently. ASR is a task where one singular label is predicted for a whole sound file. AED detects whether or not a sound event is present at every time frame within a sound file. Multiple sound events - and thus multiple labels - can be given to a single feature matrix. These are respectively called multi-class tasks and multi-label tasks and the system thus needs the ability to combine

both. They (often) require different loss calculation methods (in this case categorical cross entropy and binary cross entropy respectively), which thus should be decided in the system depending on the type of task. Following that the developer only should focus on one part of the pipeline at a time, the developer should be able to define the handling of the task when the task is defined (in other words in the Task object).

- Alongside handling the model output depending on the task definition, the model output head for each task itself should be adaptable on its definition. Multi-class tasks need only one output, which in this case is achieved by a SoftMax layer. Multi-label tasks need multiple output labels which happens here through a sigmoid layer defining the activation value for each label. The dynamic output of models is important when different task combinations are made for the same system.

- There are numerous ways to evaluate the output of the system. Standard evaluation metrics should automatically be readily available, but the developer needs to be able to define their own required implementations easily.

- The DCASE audio files are not mono audio files but stereo. This means that there are two time series in parallel for which individual feature matrices have to be made and appended. It's also possibly desired that the audio files are converted to mono files.

TODO: Model instantiation and explain

Finding the best

Explain further extensions

4.9 Easy changeable variables

4.9.1 Different datasets

4.9.2 Different Sample Rate

4.9.3 Different Feature Extraction

4.9.4 Different Data Transformation

4.9.5 Different Dataloading

4.9.6 Different DL Model

4.9.7 Different Optimizer

G

4.9.8 Different loss calculation

4.9.9 Different loss combination

4.9.10 Different Stopping Criteria

4.9.11 Different Saving Locations

4.10 Easy expansions

4.10.1 Adding Datasets

4.10.2 Adding Tasks to datasets

4.11 Simplifying abstractions

4.11.1 Saving/Reading Extracted Datasets

4.11.2 Index Mode

4.11.3 Combination of different datasets

TODO: The ConcatTaskDataset function

4.11.4 Train/test generation

4.11.5 Training

4.11.6 Evaluation

4.11.7 Result Saving and Visualizing

4.11.8 Interrupted Learning

4.12 Developmental side rails

4.12.1 Abstract Data Reader

4.12.2 Abstract Extraction Method

4.12.3 Standardized valid input

4.12.4 Centralized Train/test Operations

Chapter 5

Implementation

In this chapter, the implementation details are described for how building a multi-task learning pipeline is implemented.

5.1 Technology

The implementation is built in python and relies on pytorch [Paszke et al., 2019] 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 High Level Description

To illustrate what the resulting pipelines built with the application consist of, a simplified example is given in figure 5.1. Using the model designed in the previous chapter, a sequence of steps can be created resulting in a trained and evaluated multi-task model.

The pipeline can be split up into three parts: Data Standardization, Data Loading and Training. In the first section, Data Standardization, labeled audio datasets are transformed into standardized objects.

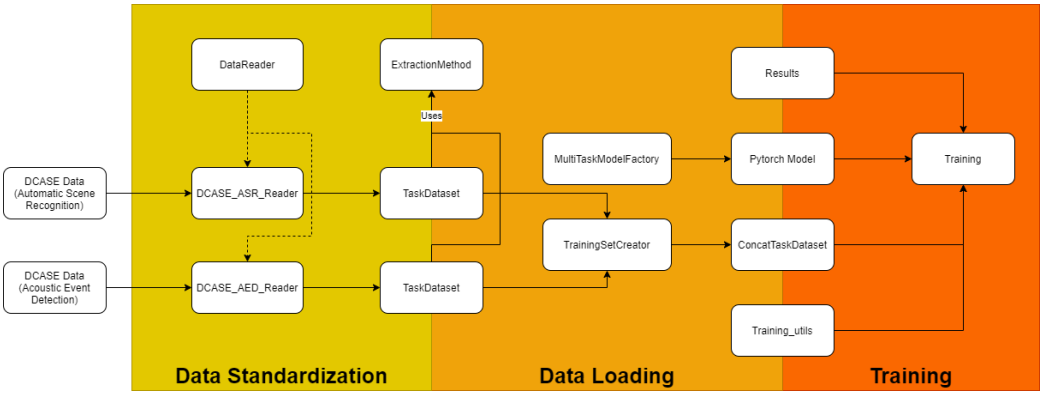


Figure 5.1: Simplified Pipeline Overview

5.3 Data Standardization

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.

5.3.1 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.

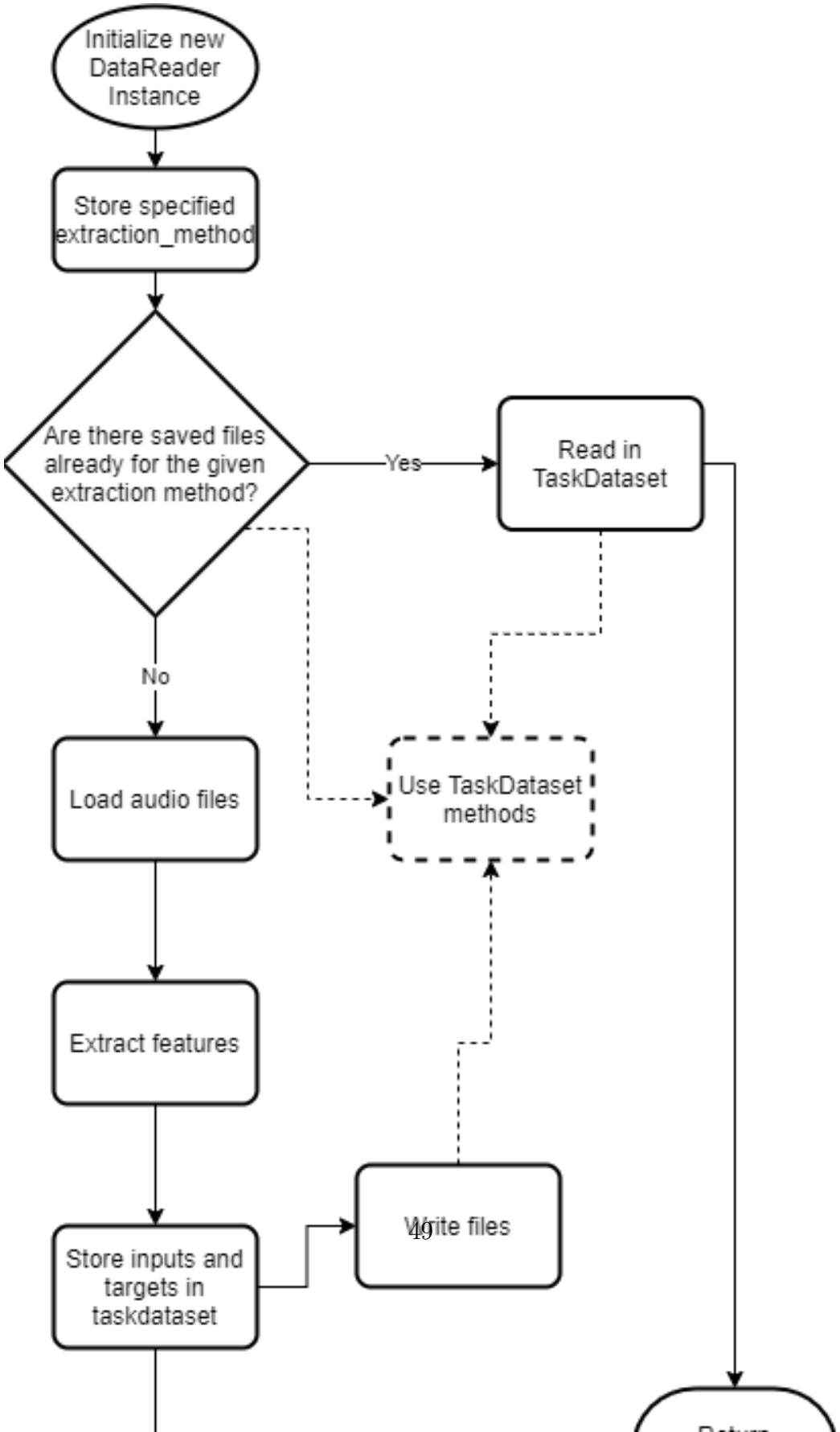
5.3.2 DataReader

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.

5.3.3 ExtractionMethod

If the DataReader is the workbench to transform audio datasets to Task-Datasets, 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


Initialize new DataReader Instance
Store specified extraction_method
Are there saved files already for the given extraction method?
Yes
Read in TaskDataset
No
Load audio files
Use TaskDataset methods
Extract features
Store inputs and targets in taskdataset
Write files

```
def return_taskDataset(self,
                       extraction_method: ExtractionMethod,
                       **preprocess_parameters) -> HoldTaskDataset:
    """
    Either reads or calculates the HoldTaskDataset object from the dataset
    :param extraction_method: The extraction method object to extract the inputs with
    :param preprocess_parameters: The preprocessing parameters see preprocess_signal
    :return: HoldTaskDataset: The standardized object
    """

    taskDataset = self.__create_taskDataset__(extraction_method)
    if self.check_files(extraction_method):
        print('reading')
        self.read_files(taskDataset)
    else:
        print('calculating')
        self.load_files()
        self.calculate_input(taskDataset=taskDataset,
                             preprocess_parameters=preprocess_parameters)

        self.calculate_taskDataset(taskDataset)
        taskDataset.validate()
        self.write_files(taskDataset)
    return taskDataset
```

Figure 5.3: Code which calls the abstract methods that need to be implemented in the DataReader

```
class LogbankSummaryExtractionMethod(BaseExtractionMethod):

    def __init__(self,
                 preparation_params: dict = None,
                 extraction_params: dict = None
                 ):
        super().__init__(
            name='LogbankExtractionMethod',
            extraction_method=LogbankSummaryExtraction(PerCelScaling(
                NeutralExtractionMethod(preparation_params=preparation_params, extraction_params=extraction_params))),

        )
```

Figure 5.4:

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.

5.3.4 TaskDataset

TODO: Revisit this, the initialization is now different and splits inserting the data from the rest of initialization

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 3 parts to this that have their own strategies: File management, structure of data, the index mode and combining separate train and test sets.

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.

Now, 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.

Lastly, the case is examined where a dataset has a predefined train and test set, possibly stored at different locations. As seen in figure **??**, a dataset is not simply stored as a TaskDataset, but in the hold-train-test structure. Every HoldTaskDataset has a Train- and TestTaskdataset, for which it is the administrating object. Separated Train and Test datasets can be made through the HoldTaskDataset and only require unique paths to save their data. At that point, they can just utilize the same functions for loading and saving defined in TaskDataset.

Getting a data instance - i.e. the feature matrix and targets - requires more than just plucking the corresponding elements from the list. While the

```
def calculate_input(self,
                    taskDataset: HoldTaskDataset,
                    preprocess_parameters: dict):
    print('training')
    perc = 0

    for audio_idx in range(len(self.files)):
        read_wav = self.preprocess_signal(self.load_wav(self.files[audio_idx] + '.wav'), **preprocess_parameters)
        taskDataset.extract_and_add_input(read_wav)
        if perc < (audio_idx / len(self.files)) * 100:
            print("Percentage done: {}".format(perc))
            perc += 1
```

Figure 5.5:

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.

5.3.5 Example

5.4 Data Loading

After the DataReaders created their individual **HoldTaskDatasets**, the data should be prepared for training, split in train and test sets, concatenated into 1 dataset and loaded in batches. This section will detail how the system turns the TaskDatasets into train and test sets and then loads them in batches for training.

```
def calculate_taskDataset(self, taskDataset: HoldTaskDataset, **kwargs):
    distinct_labels = self.truths.folder.unique()
    distinct_labels.sort()
    distinct_labels = np.append(distinct_labels, 'unknown')

    targets = []
    for i in range(self.truths.shape[0]):
        target = [int(distinct_labels[label_id] == self.truths.loc[i].folder)
                  if (self.truths.loc[i].method == 'genuine' or self.truths.loc[i].method == 'human')
                  else int(label_id == len(distinct_labels) - 1)
                  for label_id in range(len(distinct_labels))]
        targets.append(target)

    taskDataset.add_task_and_targets(task=MultiClassTask(name='ASVspoof2015',
                                                         output_labels=distinct_labels),
                                     targets=targets)
```

Figure 5.6:

5.4.1 Combining TaskDatasets

In order to combine multiple datasets into a structure which combines them all as one, while still preserving necessary task information and functions, PyTorch has a class called ConcatDataset which does exactly that. ConcatDatasets can be used as inputs for PyTorch's DataLoader classes, which creates batched inputs for training. This class is extended in the framework's **ConcatTaskDataset**. Intuitively, this class accumulates **TaskDataset**s, but also provides necessary functionalities for combining different datasets.

The aim is now to create a **ConcatTaskDataset** for training and testing, from the individual TaskDatasets and have its data be valid for loading and training in a multi-task manner. There are a few hurdles to this, as the framework should take in account a few different possible scenarios. Generating batched inputs requires that every input in the batch has the same shape. Specifically each feature matrix and each target matrix within a batch must have the same dimensions.

Getting an item from a **TaskDataset**, means getting a feature matrix and a target matrix at the specified index. When loading data batches, a number of feature matrices and target matrices are concatenated in their respective batch matrices. This function has to take in account the scenario when multiple tasks are present within the same batch. In order to be able

to quickly look up to which dataset an item belongs to, a dataset identifier is also returned. This dataset identifier is set on initialization in the **ConcatTaskDataset** per task, as it is simply the order id a **TaskDataset** has in the **ConcatTaskDataset**. After a batch of feature matrices and targets are made, the system can thus quickly identify which line belongs to which task, which is useful for updating different loss functions later.

Another addition is required in the **ConcatTaskDataset** for this scenario as well, namely padding of the targets. Because targets have to have the same dimensions to be loaded in the same batch matrix, they have to pad their vectors with zeros for to achieve the same vector size across all tasks in the **ConcatTaskDataset**. These tasks also include the extra tasks possibly present. Afterwards, the system has to be able to point out which indexes in the padded vector belong to which task, which is done through a function that generates a matrix of booleans per task, pointing out which columns are theirs.

This is the structure how **TaskDatasets** are combined, but before that, they have to be split into train and test sets and have their data processed for training. Now that the end goal is clarified, the road leading up to it is detailed.

5.4.2 Generating Train and Test Sets

To simplify the process of generating concatenated train and test sets combined from various datasets, the system has an abstraction to this process, named the ConcatTrainingSetCreator. This class can take the different datasets and generate training sets according to k-fold validation, with a variable amount of folds. In the process, it also applies further preparations for the data to be training ready.
TODO: show a code example

For every TaskDataset that gets added in the **ConcatTrainingSetCreator** a **TrainingSetCreator** is made, which deals with the individual TaskDatasets operations. This class is just an assembly line for calling the functions in the TaskDataset class. As already mentioned the TaskDataset objects exist in the Hold-Train-Test structure as seen in figure **??**. The **HoldTaskDataset** is what comes out the **DataReader** and holds both a

TrainTaskDataset and a **TestTaskDataset**. Either the Train- and TestTaskDataset are defined at the creation of the HoldTaskDataset, in which case, the HoldTaskDataset holds no actual data samples, or they are created using the k_folds method. This method returns a sklearn split generator, which generates stratified splits, meaning the original distribution in terms of target labels will be matched as much as possible in each split. The output of this generator are just two sets of indexes, which can be turned into the **HoldTaskDataset**'s train and test sets with the get_split_by_index method.

So back in the **TrainingSetCreator**, there is a generation function that automatically processes the data and returns a training and test set from the assigned **HoldTaskDataset**. When this TaskDataset has a predefined training and testing set, it will simply return this. If one has to be created, then it will automatically generate the kfold splits and return the next **TrainTaskDataset** and **TestTaskDataset** every time the function gets called. The number of k can be defined beforehand. However, if one TaskDataset that does have predefined train and test sets is combined with one that does not, it will automatically generate its train and test set k times. Calling the generate_concats method in the **ConcatTrainingSetCreator** then calls each individual **TrainingSetCreator**'s generate train test method and combines each train test and each test set in **ConcatTaskDataset**s. From the outside, one could just input all the TaskDatasets from the DataReaders and then iterate through the generate_concats method, which results in valid datasets for Training and Evaluation.

5.4.3 Preparing Inputs to same size

While the system can automatically make the target vectors the same size, for all the datasets, the developer should be in charge on how this happens for input feature matrices. Audio data can come in largely varying lengths and feature extraction methods who's output dimensions depend on the time domain, will have to either cut or pad their input matrices to the same size as the rest of the batch. The TaskDataset class has a function that transforms the feature matrices like this, which in turn calls the prepare_input method from its ExtractionMethod object for each matrix. Because of this reliance on the used feature extraction method, the preparation is also in this class.

The framework provides two preparation methods out of the box. The first simply cuts the matrix down or pads the matrix to the desired feature length. It only requires the desired window length as input. This also can be used alongside a window size calculation, which takes the median window length of all the feature matrices. The second one transforms each instance to possibly multiple windows of the desired length, with a given hop length. This way, no information is lost, but it can greatly increase the amount of data. If the developer wants to write their own function for this, they can just extend the **ExtractionMethod** class and insert that into the TaskDataset. The preparation is automatically called in the **TrainingSetCreator**. The preparation parameters are stored beforehand in the **ExtractionMethod** object.

5.4.4 Scaling Inputs

In order to produce successful results in deep learning, the numerical inputs often need to be scaled. This happens because input variables may have different scales, which in turn can make training unstable. Scaling normally happens based on statistics calculated from the dataset, which are then used to change the distribution of the data. One example is Standardization, for which the data's mean and variance are taken and the data transformed so that the new mean and variance are 0 and 1 respectively.

The statistics have to be calculated on only the training set - otherwise this would result in data leakage which would give a skewed result for evaluation - and then used to scale both the training and test set. Therefore, calculating the statistics are only a function in the **TrainTaskDataset** class. Because of the Hold-Train-Test structure, every set shares the same ExtractionMethod object, so when the statistics are saved in the TrainTaskDataset, they can be used in the TestTaskDataset as well.

The default scaling uses sklearn's StandardScaler, which performs standardization. Already included are two ways of performing scaling. One is where each matrix is scaled per feature. This is the normal case if the feature extraction depends on the time domain. The statistics are taken from all the rows per column for every data sample and the scaling is applied for every column. The other one scales per feature and row, for situations where each

row is another feature. The statistics are thus taken and applied per cell of each feature matrix.

When the feature statistics are calculated, which happens in the **TrainingSetCreator**, the transformation only happens at the getter level of the TaskDataset. This way, the transformed data is not stored every time and shouldn't be reversed every new train/test set gets generated. It also allows the system to run in index mode in largely the same way as normal mode.

5.4.5 Filtering Inputs

In order to examine the effect of the distribution of data samples with specific labels, the framework adds an easy way to filter/limit the amount of samples per label. The **TaskDataset** class includes a sample_labels function, for which a dictionary can be inserted where the key is the label and the value its maximum amount of samples. This operation does remove samples from the TaskDataset object, so in case the filtering needs to change, the object has to be reread from memory. This filtering happens before the train/test sets are created in the **TrainingSetCreator**.

5.4.6 Loading Data

When everything is prepared and the cumulative train and test sets are created, the datasets can finally be loaded for training. The train and test generation, including every preparation step leading up to it can simply be done by inserting all the created TaskDatasets into a **ConcatTrainingSetCreator** and iterating over its generate_concats function. At that point, the train and test set can be inserted into the training and evaluation functions, which will be detailed later.

However, what is still explained here, is the data loader and what it returns, as this is important for how everything before it functions. The training uses PyTorch's DataLoader which takes a PyTorch Dataset and PyTorch BatchSampler. The BatchSampler is intuitively used for creating batches of input items and defining how they are assembled. The standard BatchSampler does this randomly, meaning every batch can have an item

```
def create_taskdatasets(self, class_list: List[str] = None):
    self.reset_taskDatasets(class_list)
    for dr in self.data_readers.keys():
        if (class_list and dr not in class_list) or dr in self.taskdatasets.keys():
            continue
        tsk = self.data_readers[dr].return_taskDataset(
            extraction_method=self.__get__pipe__(key=dr, dictionary=self.extraction_methods),
            resample_to=self.__get__pipe__(key=dr,
                                           dictionary=self.sample_rates),
            **self.__get__pipe__(key=dr,
                                 dictionary=self.preproccesing)


        )
        if dr in self.dics_of_label_limits:
            tsk.sample_labels(self.__get__pipe__(key=dr, dictionary=self.dics_of_label_limits[dr]))

        tsk.prepare_fit()
        tsk.prepare_inputs()
        self.taskdatasets[dr] = tsk
    return self.taskdatasets
```

Figure 5.7:

from any dataset. The framework also provides an extra BatchSampler, that keeps every batch from the same TaskDataset, but switches from which one randomly. The BatchSampler does its operations based on indexes from the dataset, which the dataloader then uses to call the __getitem__ function from the Dataset. As mentioned before, each matrix in a batch must have the same dimension. Ergo in the first sampler, all the datasets must have feature matrices with equal dimensions, while in the second, the matrices only have to have the same dimensions within the same dataset. In other words, the dataset preparations depend on which BatchSampler will be utilized for training.

In **TaskDatasets**, this function returns three things: the feature matrix at the specified index, the (cummulative) target vector at the specified index and an identifier for which dataset the item belongs to. The difference between index_mode and without, is solely how it returns the feature matrix. When iterating over the DataLoader, these will thus be returned in 3 separate matrices of the specified batch size.

```
csc = ConcatTrainingSetCreator(random_state=123, nr_runs=5)
csc.add_data_reader(DCASE2017_SS(object_path=os.path.join(data_base, 'DCASE2017_SS_{}'), index_mode=True))
csc.add_data_reader(DCASE2017_SE(object_path=os.path.join(data_base, 'DCASE2017_SE_{}'), index_mode=True))
csc.add_sample_rate(sample_rate=32000)
csc.add_signal_preprocessing(preprocess_dict=dict(mono=True))
csc.add_extraction_method(MelSpectrogramExtractionMethod(**extraction_params))
for train, test in csc.generate_training_splits():
```

Figure 5.8:

5.4.7 TrainingSetCreator

5.5 Training

When the train and test sets are created, it is time for the training loop. Training and evaluation are designed to not require any modification, as they include hooks for multiple possible extensions. At this point, the only inputs that are necessary for training are the PyTorch model, a **Results** object, the training set and any additional training parameters. This simplicity yet extendibility for training tries to allow developers to easily change variables anywhere in the process, as quickly as possible, without having to adjust other parts of the pipeline. There are four components to this stage: Model creation, results handling, training updating and evaluation.

5.5.1 Model Creation

Model creation does not have any additional functionalities and simply requires PyTorch Modules. It is up to the developer to create models using PyTorch that can handle their Multi-task requirements. This also allows external models to be plugged into the framework and easily tested. The standard assumption the system does make in training however is that the different classification results are returned in tuples.

In order to provide a helpful basis, the framework already has two simple, adjustable models available: A DNN and a CNN. Both consist of an adjustable number of shared layers, with an adjustable number of nodes that branch into different output layers per task that have an activation function, depending on the type of task. Multi-class tasks have a log softmax activation function, while multi-label tasks get a sigmoid activation function.

```
csc = ConcatTrainingSetCreator(random_state=123, nr_runs=5)

csc.add_data_reader(DCASE2017_SS(object_path=os.path.join(data_base, 'DCASE2017_SS_{}'), index_mode=True))
csc.add_data_reader(DCASE2017_SE(object_path=os.path.join(data_base, 'DCASE2017_SE_{}'), index_mode=True))
csc.add_sample_rate(sample_rate=32000)
csc.add_signal_preprocessing(preprocess_dict=dict(mono=True))
csc.add_extraction_method(MelSpectrogramExtractionMethod(**extraction_params))

mtmf = MultiTaskModelFactory()

mtmf.add_modelclass(MultiTaskHardSharingConvolutional)
mtmf.add_static_model_parameters(MultiTaskHardSharingConvolutional.__name__,
                                 **read_config('model_params_dnn'))

for train, test, fold in csc.generate_training_splits():
    model = mtmf.create_model(MultiTaskHardSharingConvolutional.__name__,
                              task_list=train.get_task_list())
    print('Model Created')

    results = Training.create_results(modelname=model.name,
                                      task_list=train.get_task_list(),
                                      fold=fold,
                                      results_path=drive + r":\Thesis_Results",
                                      num_epochs=meta_params['num_epochs'])

    Training.run_gradient_descent(model=model,
                                  concat_dataset=train,
                                  results=results,
                                  batch_size=meta_params['batch_size'],
                                  num_epochs=meta_params['num_epochs'],
                                  learning_rate=meta_params['learning_rate'],
                                  test_dataset=test)
    results.close_writer()
```

Figure 5.9:

5.5.2 Results Handling

To evaluate the system's performance efficiently as well as creating an abstraction layer for reading/writing intermediate results, the framework utilizes the **Results** class. This object is responsible for storing and recalling calculated data during training and evaluation. Furthermore, it also provides an easy way to visualize data through TensorBoard.

The results object has to be created beforehand with a unique name for the training run. This gets used for the file locations, as well as identifiers to compare runs in TensorBoard. Developers can create their own Results

object or use the *create_results* method in the Training class, which handles the unique name creation. After it is created, the training function and evaluation function require this object and automatically write their results after every batch and after every epoch. After every batch, the overall loss gets saved as well as the true labels, the predicted labels and the loss of each individual task. After every epoch, this information is used to calculate the learning curve, the evaluation metrics and the confusion matrix. Each of these then both gets written to files using the Joblib library, as well as visualized using the TensorBoard Library. Developers can see the metrics develop during training, which can also help them anticipate problems in their models.

The evaluation metrics are calculated using the sklearn's metrics library. These return the precision, recall, f1-score and support metrics for all individual labels as well as for different aggregation forms. For multi-class and binary class tasks, these aggregation forms are macro average and weighted average, along with the aggregate score for accuracy. For multi-label tasks, these are micro average, macro average and weighted average aggregations. These reports get written in full to files for every epoch. The visualization of evolutions of these numbers can be seen and downloaded through TensorBoard's UI.

Also the confusion matrix in every epoch gets calculated and written to TensorBoard, so one can follow the evolution of how samples are classified. Same goes for the learning curve or the loss curve which is calculated from the overall combined loss of all tasks.

Not only do the metrics get written every epoch through the Results object, a copy of the model that is being trained its parameters get written as well. This thus creates a checkpoint for the model at every epoch that can be used later. Every writing function of the previously mentioned data comes with a straight forward loading function as well. Every path and name used is set during initialization. Therefore, one would only need to initialize a **Results** object in the same way as it was done previously, in order to load up all written data related, using the same additional information (e.g. epoch number, the task, ...) that was used to write. The Results object can easily be recreated through a static method in the class called the *create_model_loader* which takes the run name and any custom paths that are

needed to recreate the same results object. This simplifies the data loading process when a developer would need it, but in combination with saving the model parameters also allows for something more.

Being able to easily load every model state in training, means that each of these states can be reintroduced into the training or evaluation function. This allows for what is called interrupted training, meaning the training loop can simply continue from a certain epoch's model state if the loop was somehow stopped. To facilitate this, both the training and evaluation functions include a start epoch parameter, from which the loops can then continue until the end. The evaluation function goes even further and includes an automatic model parameter function if the inserted model is blank. This way, evaluation of the different states can happen at any time, as long as the **Results** object it received is correctly initialised. If the object was created using the default settings before, this only requires the correct name of the run.

5.5.3 Training Updating

Training a Neural Network happens by multiple times iterating through the data, each time inserting a batch of feature matrices, predicting their labels, calculating a loss function from the predictions and the correct labels and then updating the model's parameters based on the loss function using a - usually gradient descent based - optimizer. In a multi-task setting, this can get trickier as one has to calculate each task's loss separately, possibly with different loss functions, based on only the inputs from that task's dataset and then combine the losses to a single result, with which to update the model. A platform that is able to handle all sorts of task and set-up variations has to thus dynamically deal with various scenarios as well as open the opportunity for the developer to customize for possible other variations. The training function *run_gradient_descent* is designed to not require any code adaptations, meaning most of its functionalities have a default way of working which can be overwritten. Each step of the training loop will be explained and discussed what scenarios it can take on.

Initializing

From the start, the developer can submit their own optimizer, data loader,

device and **TrainingUtils**. The optimizer just needs to be one of the PyTorch optim objects, or extends it, with the default being the ADAM optimizer. The data loader also should either come from or extend PyTorch's DataLoader. This is by default the standard PyTorch DataLoader, but with the previously mentioned **MultiTaskSampler**, which alternates for each batch between tasks. The device is also an element from the PyTorch library, which is responsible for defining where the deep learning calculations are made. By default this gets set to the system's GPU if available, else the CPU.

The **TrainingUtils** object is a collection of different functions which a developer possibly wants to alter in the training loop. This includes the *combine_loss* function - responsible for defining how the losses from different tasks are added - and the method for defining early stopping. By extending this class, a developer can define their own definitions for these functions without problem.

After this, the training starts looping over the epochs wherein it loops over all the data in the dataloader.

Prediction

Inside an epoch, the training loop goes over every batch of data in the DataLoader - which holds the **ConcatTaskDataset** from earlier. Every batch includes a batch of feature matrices, a batch of correct target labels and a list of identifiers to which dataset each sample in a batch belongs. The input feature matrices are then sent to the specified device and inserted into the PyTorch model to acquire predictions. While this can be adjusted if the corresponding functions are changed in the **TrainingUtils** object, the system's only requirement from the PyTorch model is that the prediction for each task is a separate tensor, combined in a tuple. This part requires no further adjustments for the multi-task setting.

loss calculation

After the prediction is made, the loss has to be calculated for each task and combined. The list of identifiers from the data loader is there to take in account the scenario where multiple tasks from different datasets are present within the same batch. A list of booleans is created for each task that indicates for each sample in the batch whether it belongs to that task. A similar list, indicating which target columns belong to which task, is received from

the **ConcatTaskDataset**. To reiterate, every target vector has a column for every label in its own dataset, as well as zero padding for all the labels in the other datasets, which allows samples from different tasks to be present in the same batch. In order to calculate the loss for each task, the matrix of predicted labels and the matrix of ground truth labels, are filtered so that only the samples and columns for the task at hand remain, when its loss is calculated. When only one task is present per batch, this doesn't do anything and nothing is filtered out. When multiple tasks are present in one batch, but only one dataset, then no samples are filtered out, but the other target columns still are for calculating the loss.

After the unnecessary data is filtered, the predictions and ground truths can go through the loss calculation. The loss calculation functions again rely on PyTorch loss modules. These are different for different types of tasks, and the loss function, as well as the handling of the predictions and ground truths matrices are defined within the **Task** object. For example, CrossEntropyLoss cannot be used for Multi-Label classification tasks, as it requires a singular class as its target. Therefore, the system calls on a function to translate the ground truths beforehand, which in the case where CrossEntropyLoss is used for a Multi-Class classification task, would mean that the ground truths, which are encoded as binary sequences, first are converted to the class number to then be put through the loss function. The loss function itself is also stored in the **Task** object and can thus be easily be replaced by a different function through implementing one's own **Task** class extension and giving it to the **TaskDataset**. Also the final class choice of the prediction can be adjusted in the **Task** object, but this is only used for statistics, as the PyTorch loss functions can perform their own decision functions for predicted chances.

model updating

Combining the different losses then normally happens through a simple summation of all the losses. As mentioned before, this can easily be changed in the **TrainingUtils** object. This cumulative loss is then used to update the model's gradients and the model is updated using the optimizer performing gradient descent. All individual losses and the combined loss are saved through the **Results** object, and the next batch is loaded. After the batches, the training metrics are then calculated and saved, as well as the model's current parameters.

At the end of each epoch then, the system will call the **TrainingUtils**' *early_stop* function to assess whether it should quit training early or not. After training stops, everything is written to files and the function returns the trained model and the results object.

5.5.4 Evaluation

Finally, the evaluation is examined. The evaluation loop is in large parts the same as the training loop, just without updating the model's parameters. The same functionalities to deal with different multi-task scenarios are present in the evaluation function, so the same inputs can be used for both. The system thus loads a batch of data, predicts the labels, calculates a loss function and then writes the evaluation metrics to files. The same Results object should be used in training as well as testing. The metrics and their files are automatically differentiated into train and test results, but both can then be viewed and distinguished in the TensorBoard UI.

One notable thing about the evaluation function however is when it is used. This is open for two different cases: one where the evaluation happens during training and one where it happens after. If the developer wants this to happen during training, then all they need to do is include the test set in the training function as a variable. The training loop then automatically pushes the current model into the evaluation function for one epoch, with all the same variables as in training. This then just iterates through the whole test set once and writes the resulting metrics to files.

To allow for separate evaluation - e.g. if the developer wants to test a previously trained model - the evaluation function solely needs a Results object with the same data as during training and an indication that it is a blank model. When this happens, the evaluation loop automatically uses the **Results** object's function to load in a model's parameters from file, which it does for each epoch.

5.6 Complementary tools

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

5.7 Extendibility

In this section, there will be a deeper look at where the framework is open for customization and how the developer is meant to implement this. The framework aims to simplify development, but provide hooks for features that likely need to be modified. In this line of thinking, the framework has different categories to extend the base functionalities depending on the likeliness of change. Each will be examined based on their structure, what is required to introduce the extension and what it should take in account.

5.7.1 Classes that are meant to be extended

In this category are the classes that are basically abstract, for which the developer should build their own extension, unless it is already covered by the provided implementations. These contain the functionalities most likely to change depending on the specific case, but provide the required method definitions along with some basic functions to help the developer along. If the developer extends the abstract functions and follows its required output, the rest of the pipeline will not require any further adaptation.

The first one is the **DataReader** class. The structure and use of this class has already been discussed in section 5.3.2. In terms of extendibility, the parent DataReader class includes a basic structure for returning a TaskDataset, calling on a few abstract functions which should be implemented in the child class. These methods include output type hints, which if followed always lead to correct execution of the creation of a TaskDataset from the Reader, but Python cannot enforce these, so the developer should be aware of this. Aside from the abstract methods, the *return_taskDataset* method also calls on functions that do already have implementations to respectively check, read and write files to disk. These implementations simply call on the ones defined in the **TaskDataset** class and only deal with the file handling of TaskDataset objects. If a developer wants to check, read or write files

other than that, they should write their own implementations for these as well. One example scenario would be to write read-in audio files to one file, so that the code doesn't have to read in all individual files every time a new feature set is needed from the same data.
TODO: Show the return_taskDataset code

Next is the **ExtractionMethod** class. As explained in 5.3.3, this is an object used in the **DataReader**, **TaskDataset** and the **TrainingSetCreator** classes to transform individual data instances. In effect, this has three forms of transformation: feature extraction, feature scaling and preparation. The methods for all of these have to be overwritten in a child class, but there are a few predefined methods for them already available in the parent class. Also available are a few implementing child classes which the developer can use or take as an example. Even if no scaling or further preparation of the data is necessary, the methods for these will still be called in the **TrainingSetCreator**, so the developer should either not utilise this class or return the same objects in that case. Aside from the methods, it is important that each implementing class gives itself a unique name, for file storing purposes.
TODO: Show the abstract methods in ExtractionMethod

Following that is the **Task** class. This class, explained in sections 5.3.4 and 5.5.3, has two methods *decision_making* and *translate_labels* that a child class has to implement. Respectively, these are responsible for deciding how labels get assigned from probability based inputs and translating binary sequences to class numbers. These are used for loss calculation and metric calculation, which have to change depending on the classification type. It is not really expected that this class is overwritten if the developer is dealing with Multi-Class or Multi-Label type classification tasks as they already contain implementations. This object also holds the loss function for the task at hand, but is given at instantiation, making extending this class only necessary if the decision making and translation functions need to change.

Finally there is the **TrainingUtils** class. This class, discussed in section 5.5.3, is a collection of different functionalities used in the training function *run_gradient_descent*. This class is extended if the developer needs a different way to combine losses or criteria for early stopping, without having to adjust the code in the training and evaluation functions. The early stopping receives the current epoch and more importantly the **Results** object, from which it can take any previously written data in the training run.

5.7.2 Classes that can be extended

This category contains the classes that are open for extension, in case some functionality is required that is not covered in the implementation or needs to be performed differently. In this section will be explained which classes fall in this category and how they can be subclassed so that the rest of the system does not need further adjustments. These classes are objects that are handled in the rest of the system.

TaskDataset objects are itself extensions of the PyTorch Dataset class. The **HoldTaskDataset**, **TrainTaskDataset** and **TestTaskDataset** classes inherit from this class. Extending these classes is pretty straight forward and their functionalities are called in three different classes. In the **DataReader**, initialization of this class is called. In the **TrainingSetCreator**, the transformation functions from the base class are called and the train/test splitting functions from the **HoldTaskDataset** are called. For the dataloader, the getter and len methods are used. In the training and evaluation function then, only the **Task** object stored in the TaskDataset is utilised. So, in order to change any functionality related to these, one either has to follow the original output structure or simply take note of how it is used in the corresponding classes and of course its internal use. An example for something the developer would want to modify by extending the class is to change the input and target data structures. The only class the external code should change is the initialization in the user implemented DataReader, as they are responsible for correct initialization anyway, but otherwise any data manipulation is handled inside the class internally. No external class makes any assumptions about the internal nature of the TaskDataset, except for the return types of its functions.

The **ConcatTaskDataset** functions like this as well, but is only used in the **ConcatTrainingSetCreator** and the training and evaluation function for two simple getter methods. One is to return the list of all **Task** objects in the concatenated dataset and the other for returning the target flags matrix, or the indicators per task which columns belongs to it.

Result objects can be extended as well. This class is only used in the training and evaluation functions. Its use happens through adding predicted outputs and ground truths along with the losses for each individual task as

well as the combined total, every batch in the epoch. At the end of the epoch then, the *add_epoch_metrics* function is called, where the metrics of the epoch are calculated and the internal writing function for each metric type as well as the model parameters are called. Extending this class can thus be done for each individual write/load function. Another option is thus to extend the batch and/or epoch functions in order to change what metrics are calculated and written, without having to change anything about the training or evaluation function. The Results object is also used in the **Training_Utils** class to calculate the stopping criteria. Any additional functions can thus be added and called for this function by extending both this and the **Training_Utils** class as mentioned before.

5.7.3 Classes that should be extended from outside libraries

These are classes where the system relies on the original PyTorch implementation. These extensions should simply follow the original implementation's functions, that can be found in the PyTorch documentation. These are the PyTorch Module, the classifier used in the training and evaluation function, which should be extended anyway for every new implementation. Also the PyTorch DataLoader - which handles the creation of batches - and the PyTorch DataSampler - which handles how batches are sampled from the wider set. All of these are only used in the training and evaluation functions, for which they are optional inputs with default values.

5.7.4 Classes that should not be extended

Finally there are the classes that are not open for extension. In actuality, they can be extended of course, but the system is not designed for it and changing their code likely requires extensive rewrites in depending classes. In stead, these are designed so that their internal functionalities can be modified based on input as much as possible.

The first instance is the **TrainingSetCreator** and the **ConcatTrainingSetCreator**. The ConcatTrainingSetCreator just creates TrainingSetCreators for each dataset and forms the concatenated train and test sets from

their outputs. The TrainingSetCreator then is nothing more but an abstraction that calls the data manipulation functions in the **TaskDataset** objects. Every one of its operations thus only depends on the implementation inside the **TaskDataset** object that it handles, the rest is just organized calling of these functions, in order to create valid train and test sets. Aside from the data manipulation functions though, the preparing calculations for those manipulations are called here as well. Any code the developer thus makes to modify the behaviour of this class would need to take this in account, but the framework keeps the scenario in mind that these are not desired at all, so it won't make any assumptions for the implementation of its children.

The next instance is the Training class, which of course contains the training function *run_gradient_descent* and the evaluation function *evaluate*. The framework is designed so that these functions do not have to change at all, by standardizing all data structures beforehand and making functionalities modular and changeable by input. The extension of the classes described above mostly change the behaviour in this class. Here, a short overview will be given of how each functionality in these functions can be changed from the default behaviour.

- The DataLoader and its Sampler are PyTorch implementations can be given as input
- The device on which the deep learning is performed - the cpu or the gpu - is a PyTorch implementation and defaulted to the gpu if available, or can be defined as input
- The optimizer responsible for updating the model's parameters is a PyTorch implementation and can be given as input
- The **TrainingUtils** object can be given as input. This object's functions are called for combining the losses from different tasks and early stopping criteria. These functions can be changed by creating an object extending the **TrainingUtils** class.
- The PyTorch Model responsible for predicting the labels of an input can be given as input

- The data structure of the inputs and targets can be changed in the **TaskDataset**'s getter function
- The way a target vector from a task is translated to serve as input for the loss function relies on the *translate_labels* function of the **Task** object in the **TaskDataset**
- The loss function of a task is given as input in initialization of the **Task** object in the **TaskDataset**
- The decision function, translating class probabilities outputted by the PyTorch Model to actual classes, relies on the *decision_making* function of the **Task** object in the **TaskDataset**
- The **Results** can be given as input, which receives and stores the outputs, ground truths and losses every batch then calculates and writes the metrics based on these results every epoch. This also stores the model parameters.

5.8 Three Implementations

Chapter 6

Evaluation

IDEA: get paper examples and discuss how to implement them

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.

Nadia Bouassida, Hanêne Ben-Abdallah, and Faïez Gargouri. A uml based design language for framework reuse. In *OOIS 2001*, pages 211–221. Springer, 2001.

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

François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. Deap: A python framework for evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 85–92, 2012.

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.

Petko Georgiev, Sourav Bhattacharya, Nicholas D Lane, and Cecilia Mascolo. Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–19, 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.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Don Roberts, Ralph Johnson, et al. Evolving frameworks: A pattern language for developing object-oriented frameworks. *Pattern languages of program design*, 3:471–486, 1996.

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.

Han Albrecht Schmid. Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51, 1997.

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.

Kuilong Xu, Shilei Huang, Gang Cheng, and Xiao Song. A multi-task learning approach based on convolutional neural network for acoustic scene classification. In *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*, pages 23–27, 2019.