Thesis

Kilian Callebaut

November 10, 2021

Abstract

Abstract

Contents

Chapter 1

Introduction

With the huge expansion of machine learning research and applications over the last years, comes a matching hunger for new data to drive the research forward. Seemingly on a yearly basis, datasets are enlarged or created which allow new recognition systems to function. Furthermore, new ways are continuously devised to combine information, be it from different datasets or from different tasks. Some machine learning goals are simply inherently too niche to build the extensive dataset for, which it would get in an ideal world. Researchers often have to get creative when it comes to utilising the information they have in order to build the systems that have the performances they need in a real life context.

Audio recognition is a field where researchers especially have to get creative. Tasks like speech detection for example have to deal with an incredible amount of variation in terms of voices, genders, accents, background noise and language. Building datasets that can firmly cover these potential real world variations are nearly impossible. On top of that is it very hard to get an extensive, strongly labelled dataset for audio, even more so if the annotations are for certain sections in time only.

For this reason, Multi-Task learning has very recently gained more and more attention as a way to enlarge the information available for performing a task. This has the potential to build further reaching recognition systems by combining recognition systems performing smaller, focused tasks which share their information. The objectives for using this techniques have grown quickly beyond simple performance improvement.

However, implementing combinations of tasks and datasets can quickly scale in developmental complexity, as each one can have its own structure and getting the combined data to fit in a processable form for the systems often lead to complex solutions. On top of that, research is spurred by experimentation through varying parameters and subsystems later in the structure, which can in turn require developers to make changes back in the individual source structures. Problems should only be dealt with once. The process that a developer creates of extracting data from a dataset, applying various transformations, using it to train a recognition system and then evaluating that system is what will be referred to as the **Deep Learning Pipeline** and if that process involves the combination of multiple sets of data the **Multi Task Pipeline**.

Building pipelines that draw together different sets of information can quickly become clutered with rigid functions and classes as it requires an immense amount of foresight to anticipate required functionalities. That goes even more in case the system being implemented is not set in stone beforehand and requires iterative design decisions based on intermediate evaluations. Research and development implementations usually is not simply about executing pre-made plans, which requires that their implementations are open for these changes and additions. Considering the growing amount of datasets, being able to quickly add them to built pipelines would be a big step in facilitating future work to be performed.

This work tries to offer the tools necessary to efficiently implement combinatory multi-task pipelines, with attention to the variation of intermediate parts.

1.1 Example: General purpose multi-task classifier

As an illustrating use case, imagine a scenario where a general purpose classifier has to be built using a multi-task network. It has to be able to output multiple annotations from an audio fragment at once. The developer has to

decide which smaller tasks to use, combine multiple task specific datasets, figure out the best features to represent the audio, pre-process the data and of course develop a functioning neural network. General purpose classifiers could contain various annotation goals like:

- Speech Activity Detection (SAD): Automatic detection of the presence of speech in an audio frame along with the exact moments it happens.
- Acoustic Event Detection (AED): Detection, Identification and localization of specific sound events happening within an audio fragment.
- Acoustic Scene Classification (ASC): Recognition of tye type of environment the audio fragment takes place in.

Differences between tasks reveal its presence through these examples. AED and ASC for example are different in that ASC is a task dealing with analysing the background noise patterns of audio while AED needs to pinpoint the beginning and end of its subject. Combining these sorts of differences of tasks certainly have been used to improve the performance of one task [Imoto et al., 2020], but in this example the goal would be to achieve good performance on all tasks involved. If that is the goal, researching a functioning general purpose classifier would likely involve comparison to single task models performances and different levels of combination. The Multi-Task Set-up might also be brought in as a way to compress computational requirements of having multiple single task models in place, which can be looked at and compared. In some cases one could also define all tasks as one single task and compare how it fares to the multi-task options.

In essence if the goal is this open ended in terms of approaches as well as potential trade-offs, one would have to be able to create, test and compare the different task combinations as well as swap out and find the best working parameters for a number of the intermediate parts of the pipeline. Using pytorch, there certainly is support for creating models with numerous outputs, but the framework in terms of data encapsulation and manipulation is rather focused on single datasets. While there are structures which make it possible to combine different datasets, but creating batches of inputs and targets - especially ones where the sizes can vary - require either a lot of foresight or a lot of added development time. This would go even more for anticipating

the variations which were mentioned. A lot of decisions in the pipeline for things like pre-processing and transforming the data would possibly have to be made for all tasks involved.

Investigating the effectiveness of various multi-task set-ups quickly thus introduces a lot of cumbersome development overhead, which hampers the time available to actually develop the best conceivable systems. A lot of time from designing a system to the implementation would just go to waste dealing with the combinatorial aspects and making solutions applicable for multiple datasets and tasks at the same time. A lot of these issues can be anticipated and solved before a developer even starts. This work envisions to do exactly that, philosophising that a developer should only be worried about one part of the pipeline at a time, without having to worry that any other breaks down the line. This way, developers can develop and optimize pipelines as a whole. Adding datasets, tasks, manipulations as well as running and testing their work through singular lines could not only clear a lot of the road blocks for pure multi-task learning but grant opportunities for expanding datasets, easily offer common research functionalities and ready to go multi aspect evaluation of developed systems for any pytorch implementation.

1.2 Multi-Task Research

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

The process of developing multi-task set-ups depends on the context, use and goals of the system, but there are a number of steps that will almost certainly be present. In this section the intention is to outline the developmental steps with their correlated issues which will factor in how shortcuts can be made to improve the process. In essence, the job that needs to be done in both multi task as well as single task situations, is the construction of a pipeline going from raw datasets to fully trained and evaluated models. It is not very likely that this pipeline will be constructed statically. In stead, the final, best performing methodology will likely result from a process of constructing, replacing and tweaking parts in the pipeline until reaching the most satisfactory result.

This work splits the pipeline up in three distinct phases. One is the Data Reading phase, where the data is extracted from datasets to forms which are processable by the models. The next is the Data Loading phase, where these formed inputs are further refined, combined and loaded to serve as input for the models to predict as well as update in the training phase. The last is then the actual Training and Evaluating phase, where the models get updated and various metrics are calculated measuring the performance of the process.

PyTorch offers abstract classes which can consequently be inserted in its data loader functions, but extending these in a way to work with different forms of datasets can be quite the hassle. On top of that does it lack any dataset wide transformation functionalities, requiring the developer to implement those as well. While this is an annoying but manageable lacking aspect for single task problems, it becomes loathsome when trying to implement it for multiple datasets. Especially when the methodology is not set in stone beforehand and will be subject to potential, uncertain changes can this lead to a lot more debugging.

This framework therefore offers to standardize the dataset form and with it bring a whole catalogue of functionalities, while taking care of the combinatorial issues. Through this, what used to be blocks of code for functionalities which possibly had to be adapted for individual datasets, get reduced to

singular lines that add or replace new parts on the pipeline. Where the developer often had to go back and rework multiple parts to implement a new variation, they can be replaced at runtime.

For every one of the described phases though, the specific issues that pop up need to be identified. The following is a summarizing overview of the identified hurdles:

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.
- 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.
- 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.
- 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
- Some Datasets can have multiple tasks on the same inputs

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).

- 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.
- Manipulations are often dependent on the dataset and when a new dataset needs to be formed and manipulated after previous ones happened, the performed alterations need to be rewinded.

Training

- Combining datasets from tasks can be done in numerous ways, which can impact performance on training.
- 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
- In general for multi-task research, lots of parameters and parts should be varied
- 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
- Calculating, storing and visualizing results in an efficient way for comparison is crucial and can take up valuable development/debugging time
- 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

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

Providing a solution for

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 Recognition

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

Audio contains a rich amount of information. As one of the more important senses, audio provides humans with perceptual information about their environment and its occupants. Machines can analyse audio in the same way, in search of performing numerous functions humans do simultaneously. These tasks include for example deriving semantic information like recognizing speech TODO: REFERENCE, contextual information like recognizing the scene TODO: REFERENCE or entity identification like recognizing a piece of music. Audio classification is important for the field of pattern recognition, finding an ever expanding amount of applications.

Audio data and its relating tasks have been grouped and divided using numerous definitions, which change the focus and structure of the systems built. For one, the sound data can be classified according to domain groups. Duan et al. [2014] does this by subdividing them into human voice sounds, artificial sounds and natural sounds. Human voice includes speech, coughs and singing. Artificial sounds refer to human activity based sounds like traffic, aircraft and music. Natural sounds then include animals, weather and nature sounds. However within the same document, they utilise another distinction between sound data namely music, speech and environmental sounds. The point is that applications have been developed which target different defini-

tions of sound collections that are targeted. Grouping music together in the same domain with other human activity noise makes sense for applications that only need to identify a sound in an audio fragment as music as in Park et al. [2020], but not when the exact music piece needs to be identified in the presence of background noise TODO: REFERENCE.

Audio recognition tasks thus refer to the automated annotating of audio data for various sounds and/or sound groups present within. A typical **pipeline** for audio recognition tasks goes as follows:

1. Microphone records raw audio and stores it as a sampled time series.
2. Preprocessing is applied to accentuate certain properties in the audio signals, the choice of which is often dependant on the eventual application and what the system needs to be able to differentiate within the signals [Georgiev, 2017].
3. These time series then are divided into either overlapping or non-overlapping frames, for which each frame a feature gets calculated. The features of all the frames are then stored as a collection which represents a single data input instance. This is also called the **feature matrix**. A comprehensive overview of features are given in Mitrović et al. [2010]
4. The feature matrices are transformed in context of the other matrices, augmenting the extracted feature representations or scaling the values to the same specified value range.
5. The collections of data instances then form inputs for classification or regression problems. A number of instances are used for training a neural network which learns a representation of the data that it can optimally use for deriving the correct labels.

There are two distinctions in audio based learning tasks: instance-level and frame-level tasks. The difference between these is if the task needs to produce labels for the whole audio fragment or for events present within an audio fragment. The latter is a complexer task than the former. Instance level tasks include Acoustic Scene Recognition, Audio Tagging, Speaker Identification and Emotion Recognition. Frame level tasks include Acoustic Event

Detection and Automatic Speech Recognition.

Labels in instance based recognition work pretty straight forward. The data contains audio clips for which one or multiple labels are linked, depending if multiple classes can be present at once. A system thus takes a whole sound clip, extracts its features and outputs the corresponding labels the clip is an instance of. Labels in frame based recognition are different. As stated, clips are subdivided in frames and the output of the system is a series of labels, one for each frame which corresponds to a specific time step.

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
TODO: Explain multi task learning with unimportant auxiliary task for performance improvement

Multi-task Learning is a learning paradigm that performs inductive transfer by sharing representational knowledge from multiple related tasks. The principle goal of Multitask Learning is stated as being the improvement of generalization performance [Caruana, 1997]. To explain the meaning of this description and objective, the explanation of the paradigm will be given incrementally.

To begin, the term task must be defined, in order to understand what will be combined. A task is a collection of data instances and corresponding target labels, combined with a function mapping those instances to targets. Target labels can be represented as one-hot vectors for classification or continuous valued vectors for regression purposes [Meyer, 2019]. Learning a task infers learning the parameters for the mapping function to optimally link the instances to the known true labels. Usually, mapping functions are learned by only utilising the task's data and targets, which is also referred to as learning a representation. Large problems are broken into small independent subproblems that are learned this way separately in parallel and recombined afterwards. This is counterproductive as in real world problems, information from different task representations can not be used by the other tasks [Caruana, 1997]. The result might be that the representation is overfitted to

the utilised data, which falters when applied in the real world due to factors that weren't present in the data the representation is trained on. It is often found that task systems' performances degrade significantly when there is a discrepancy between the training conditions and testing conditions Lu et al. [2004].

The response to these shortcomings is multi-task learning, which learns a shared representation on multiple tasks at the same time to a degree. The mapping function's parameters are optimized for multiple objectives at once. Only one model is produced for which there are task specific parameters and shared task parameters. The specific parameters are updated using the error signals from a single task, while the shared ones are updated using the signals from all tasks Meyer [2019]. Multi-task models following this definition, can be built in a huge amount of ways, with the main concern being what parameters are shared which relates to what level of abstraction two tasks need to share in order to build a better representation for both. There are also two forms of sharing parameters: hard-sharing and soft-sharing. In the former, the shared model parameters for all tasks are the exact same. In soft-sharing settings, parameters are shared more loosely, with parameter updates happening only for one task, but the distance between the different tasks' parameters being regularized for their distance. This forces the parameters to be similar [Ruder, 2017]. This work mainly focuses on the hard parameter sharing setting.

For deep learning neural networks, like convolutional neural networks and recurrent neural networks, this parameter sharing set-up takes the form of sharing layers. The output of each layer is treated as a shared feature representation for the subsequent ones [Zhang and Yang, 2018]. This can be taken as a direct transformation from the input or combine hidden representation from multiple tasks to form more powerful hidden representations when different data sources require different sources.

To illustrate what this looks like, the simplest model structure, the multi-head model, is given as an example in figure 2.1. This demonstrates the situation where shared layers build a robust internal hidden representation utilising knowledge from multiple tasks in a feed-forward deep neural net. Combining dataset representations to a shared layer can be used to extend a single task's data sources. Multiple output heads can be defined for one

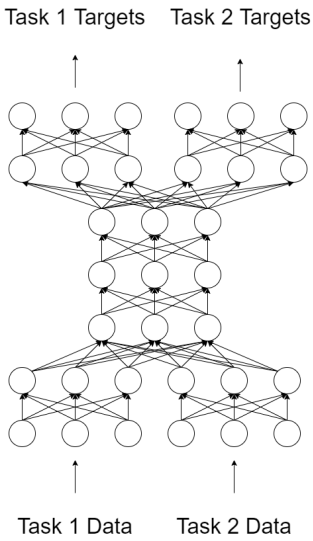

Figure 2.1: Example multi-head neural network model

dataset for a representation optimized for multiple purposes. The illustration performs both. Learning wise this achieves inductive transfer of knowledge and provides an inductive bias. Bias makes the model prefer some hypotheses over others Caruana [1997]. Defining extra targets can provide more control over the working of the model. Traditionally additional tasks are added in order to improve robustness, but recently more applications are investigated for additional improvements, like fairness [Oneto et al., 2019], compression [Georgiev, 2017] and expanding limited strong labelled data with weaker data Lee et al. [2019].

2.3 Multi-task Deep Learning Audio Tasks

Audio recognition tasks have only recently seen the adoption of multi-task frameworks, but for a quickly expanding set of reasons. This section maps out the key applications where, why and how deep learning systems where created to address problems. Single task focused systems were the way to go as instinctively it made sense that the best performances would be obtained by focusing on improving solely on building a system addressing that problem. Two factors however changed this notion. For one, systems in real

life contexts often have to perform multiple related tasks at once TODO: REFERENCE. For another, sharing information between related tasks have been shown to improve performances as it averages noise patterns and creates a more robust representation of the original data for the final classifier(s) TODO: REFERENCE. Here, it will be demonstrated how these factors influenced the recent evolution towards trying to build systems that optimise for multiple objectives at once. An overview of the used papers is given in tables 2.1, 2.2 and 2.3

Non-Parallel Multiple Tasks. What will be done first is demonstrate the context where multiple tasks need to be performed, but were not optimized at the same time in a multi-task setting. For painting a clearer picture of contexts where multiple tasks have to be learned for the same data, this work will first delve into the context where this has been addressed without a multi-task framework. In TODO:REFERENCE, a lifelogging system is designed which tries to annotate the naturalistic audio from a sensor device with different types of labels to create a contextual summary of a person's day. This takes an audio stream and performs speech activity detection, the results of which then in turn uses for its other tasks. It estimates the amount of speakers in an audio segment; uses that estimate to then recognize the speakers and determine the primary and secondary speakers. At the same time it also performs the task of "environmental sniffing" or detecting the current environment of the device. The results of this set-up are illustrated in TODO: GET DIAGRAM. This example is given to illustrate the need and the types of tasks that would be learned at the same time in a context. Task information is shared here, but segmentally in stead of parallel. The thing is however that this does not share information to improve the representation, which ignores potentially useful information. This also requires every task to be performed individually, a set-up which might not always be allowed in terms of time and resource constraints.

To clarify how this singular focus forms the basis for problems in naturalistic audio contexts, the work done by Park et al. [2020] is addressed. Here, the context of an audio device is being detected in a short time frame of seconds, by recognizing events that are tied together in wat they call contexts. First of all is the obvious need in this case for fast detection. Second point lies in their results. They find that including the detection of speech events significantly reduces the performance of the general classifier's performance.

Speech audio is different in structure from other background audio events and likely requires its own model for reliable detection.

With the first example being one of tasks connected segmentially and the second being unified in the same classifier, the stage is set for exploring the compromise between the two: separated, yet parallel task inference. While Multi-Task settings had demonstrated their use in visual tasks TODO: REFERENCE, the adoption in audio tasks was (similar to the trajectory of deep learning algorithms) a bit slower to develop. The first domain that showed its potential were speech recognition tasks.

Multi-Task Speech Recognition Tasks. For Automatic Speech Recognition, multi-task learning has been around for a while. These tasks accept windows of audio features as input and return posterior probability distributions over phonetic targets [Meyer, 2019]. The targets can range from whole words to phonetic parts and even simply characters. The multi-task model itself either serves as an end-to-end recognition model or performs a modeling function in a hybrid model.

Lu et al. [2004] proposed an architecture where noisy speech audio is fed into a Recurrent Neural Network that has a number of shared layers with three output heads. One for the prediction of words, one for enhancing speech and one for gender detection. With adding these auxiliary tasks on the same dataset, they succesfully managed to improve the original task's - Speech recognition - performance. Their reasoning for utilising multi-task learning was that performance degrades dramatically for when there is a mismatch between train and testing conditions. Multi-Task learning added robustness to classification performance.

A lot of the work done in multi-task speech recognition focuses on this sort of improvement of the original signal's representation through adding additional optimization goals. Seltzer and Droppo [2013] focus on the improvement of phoneme recognition by adding the prediction of the phone labels, state contexts and phone contexts - context here being predicting the label in the previous and next time frame.

General purpose audio The techniques that were found for Speech Recognition purposes eventually found their way for general purpose audio

tasks. There are a few trends within acoustic multi-task definitions that will be discussed here, which will be important bases for the system to cover. The trends deal with how the additional task was utilised in the same model. Four of these trends are addressed in this section: Auxiliary tasks on the same dataset, combining tasks from different datasets, splitting a task and finally multi-task learning for non-performance related issues.

Auxiliary tasks on the same dataset. The first strain of set-ups concern audio tasks that receive an additional task on the same dataset. This is usually with the aim to improve the original task's performance. These in turn come in two forms: supervised tasks [Kim et al., 2017] [Zeng et al., 2019] [Abrol and Sharma, 2020] [Fernando et al., 2020] [Wu et al., 2020] [Sun et al., 2017] [López-Espejo et al., 2019] [Panchapagesan et al., 2016] and self-supervised tasks [Lee et al., 2019] [Deshmukh et al., 2020] [Pankajakshan et al., 2019] [Lu et al., 2004].

Often, the auxiliary task is not important and only serves to help the neural network create a representation of the data that is formed for its qualities the additional tasks require. With a supervised task as an auxiliary task, this takes a more semantic form. The task is semantically related. In Zeng et al. [2019], the multi-task set-up results are explored for combining speaker identification and accent recognition in one case and emotion recognition in speech and emotion recognition in song in another. In both these instances, the semantic connections for the targets are clear. Abrol and Sharma [2020] does this concept slightly different and learns basically the same task but at different abstraction levels, in order to improve learn leveraging hierarchical relation structures. The set-up for Fernando et al. [2020] contains detecting speech activity along with predicting the next audio segment, using layers of LSTM. Here layers are shared at different levels, with a generative adversarial network that learns the loss function.

For the other scenario where the set-up adds a self-supervised task, this usually comes in the form of transforming the input signal according to different qualities, which will be transferred to the task at hand. Lee et al. [2019] investigates the effect of adding next-step prediction, noise reduction and upsampling of an audio signal as an auxiliary task to audio tagging, speaker identification and keyword detection tasks. Each possible subgroup of auxiliary tasks are tried. With this it successfully tries to get more effi-

ciency out of its labelled data. Deshmukh et al. [2020] recreates the Time Frequency representation of audio as its auxiliary task to event detection. This aims to reduce the noise of the internal representation used for classification which makes it function better as a classifier based on noisy recordings (like in real life contexts). It also utilises Multiple Instance Learning, which is a learning mechanism where instances get put into positive and negative bags. Positive bags do not only contain instances that are positive for that label - at least one for sure -, but negative bags do. Lu et al. [2004], which has been discussed before, is a hybrid of these two forms. One of its auxiliary task is gender detection (first scenario) and the other is speech enhancement (second scenario).

For the system to be created this signifies a lot of different possibilities in terms of possible targets, beyond simple labels as well as a possibly inexhaustible range for models and training requirements. Any abstraction made in this regard must be fully adaptable. Tasks should also freely be addable or retractable in order to investigate its combinations to the main task. Furthermore, grouping instances should be possible, as seen in the example for Multiple Instance Learning.

Different datasets, different tasks. Another utilisation of multi-task set-ups is to bring together two different tasks with - usually - two different datasets. The idea is to bring two tasks together that could prove to be beneficial for each other as successfully performing their combined task is directly beneficial for the task at hand. The improvement is either aimed for one task like in the previous case, or it is to benefit the performance of both tasks at once.

One of the clearest and more popular examples of this set-up is in set-ups that combine the Acoustic Event Detection (AED) task - which detects which sound events are present at each time frame - and the Acoustic Scene Recognition (ASR) - which detects in which background scene a sound sample takes place in [Tonami et al., 2019] [Xu et al., 2019] [Imoto et al., 2020] [Jung et al., 2020] [Komatsu et al., 2020]. The idea is that information on detecting specific scenes will help in detecting events, either by learning the noise pattern related to that scene or because certain events are inherently linked to certain scenes. Tonami et al. [2019] does exactly this, building a simple multi-head model that shares three layers before venturing off in task

specific networks. The labels which improved in this context are investigated, finding that labels which are only connected to each other but not other labels in the parallel task improve significantly in accurate detection. Xu et al. [2019] performs this but with a sole focus on improving AED. Imoto et al. [2020] builds on this but reforms the ASR to output soft labels (percentages in stead of deterministic labels). For training there is a separate independent network for ASR that teaches the ASR task in the multi-head model in what is a called a teacher-student learning framework. Finally [Jung et al., 2020] and [Komatsu et al., 2020] adopt a model, where the different task results are directly combined afterwards to only output better labels for one task.

Other examples are limited and underdeveloped. The work done by Sakti et al. [2016] might give a hint for the reason. They tried to bring together speech recognition with environmental sound detection, yet have to end up limiting the shared layers in both tasks, mainly finding that combined features improve the set-up slightly. These two tasks might have been too unrelated for the multi-task set-up to offer improvement, but very little investigation into this aspect was performed. The final example is found in the work by Huang et al. [2020a] and Huang et al. [2020b]. This is interesting as it finds a way to improve its intended task (AED) with weaker labeled data. It combines its model at different layers with splitting branches, adding an optional side branch for stronger labeled data that improves overal performance if available. This potential for extending original datasets with weaker datasets but with likely more instances is immense, which was proven by winning first place in the 2019 DCASE Task 4 competition. This shows how the multi-task framework has been gaining momentum over recent years, with the capacity to improve available task information, whether it is by simply providing more contextual information or finding a way to provide more valuable training data.

Same task split. Present in the last example Huang et al. [2020a] is the idea of using a multi-task framework to split a complex task in two separate tasks and combining the results for a single improved prediction. This mostly happens for AED tasks, by redefining this task as two tasks: determining the event type and determining the time . This either happens through splitting the task into a classification task for the type and a sound activity detection task that simply outputs whether any sound event is present in a time frame [Morfi and Stowell, 2018] [Pankajakshan et al., 2019]. The other way it has

been performed was by adding a regression task for the time offset of events to add more exact information on when the time exactly starts. These always involve some sort of result fusion after (probabilistic) prediction output.

In Phan et al. [2017] and Phan et al. [2019] also resembles this model, but is only a multi-task framework right at the end, as it optimizes for different criteria. One is the detection error, one is the distance error of events and the final is the confidence in the first two predictions. This does not completely defines the task as different ones, simply optimizes for different criteria, but still requires fusion after the fact.

The final interesting case is that in Nwe et al. [2017]. This splits up ASR into multiple tasks, which are all still ASR but in different recording conditions. The labels are split up into three groups: Indoor scenes, sparse outdoor scenes and crowded outdoor scenes, each group being defined as its own task.

In essence tasks on datasets can be reconstructed in an infinite amount of ways. The system should offer this level of control over datasets and tasks. With the idea in Huang et al. [2020a], it should also take in account parallel models that influence the training of the multi-task framework that was built.

Multi-Task Learning For Other Reasons. Finally, there are the cases where the multi-task set-up is used for reasons that do not relate to performance improvements of any kind. This idea is still limited, but illustrates how much further the multi-task framework's applicability goes. These investigate the capabilities of combining datasets and tasks that are unrelated and possibly require combining a large amount of heterogeous tasks to be combined together, compared to the previous trends.

In Georgiev [2017], a multi-task framework is used as a means to compress deep learning models that have to perform different tasks. Even in the instances where independent models are technically more optimal, it might be preferable to combine their network layers. This is useful if the model has to be deployed on computationally constrained devices. Multiple independent models can require too high resource costs, so the argument is that using a multi-task framework that does not sacrifice too much in terms of performance compresses the amount of complexity for execution of the same tasks.

For the same reasons, the work in Tagliasacchi et al. [2020] is performed, but offers more adaptation to the different tasks. Both of these examples bring together numerous tasks and datasets. These require a lot of work on combining dataset differences for execution in the same network. Also take in account that there is a process to arrive at these models, in which design and parameter differences have to be varied (often for each dataset in the same way), evaluated and compared with the variations before.

What is taken away from each of these trends, is that there are numerous opportunities in acoustic multi-task learning. An huge amounts of variations and possibilities still have to be explored. Facilitating free experimentation with the differences in tasks and the way to combine these would be crucial for promoting further research in these fields. A platform could be built that can dynamically handle these cases while offering abstractions that quickly deal with physical problems that can arise from these cases.

Summarizing the take aways from observing these trends goes as follows. In the first case, datasets can have multiple tasks defined on them. Extra tasks come in different forms themselves, even as aggregated forms from other tasks (multiple instance learning). In the second case, different tasks and different datasets get combined. This of course necessitates dealing with dataset differences while the different ones can be seen as one big dataset or not. Models outside the multi-head model can also be brought in to affect training. The performance effect on the datasets and the output labels also needs options to be evaluated closely. The third case clarifies that developers need a lot of control over both tasks and datasets. The final case demonstrates that multi-task networks are implemented for more than performance improvements and a possibly huge amount of datasets and tasks can be combined together. Handling of these are as likely to be different for each case as they are to be the same for all.

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 TODO: Apparently

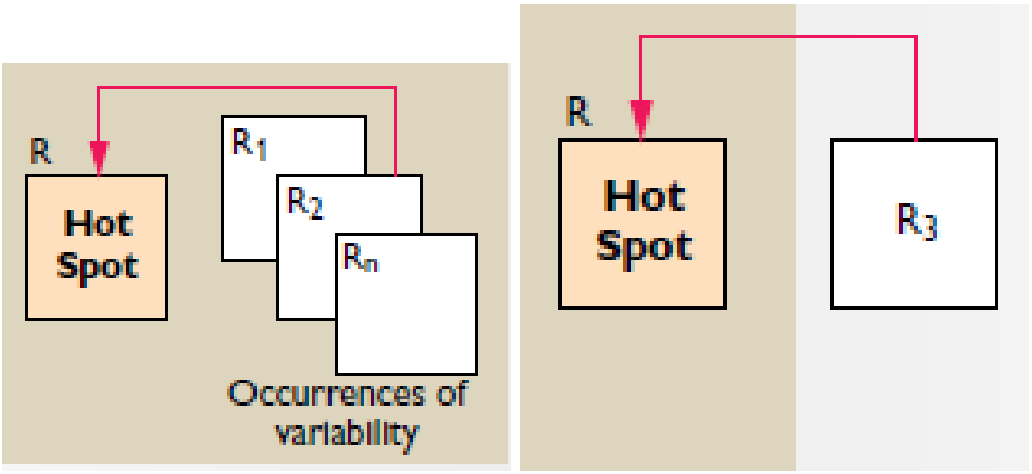

Figure 2.2: Black box hot-spot (left) and White box hot-spot (right)

there

Frameworks offer designs and pieces of code that are reusable and functionally allow the creation of different applications within its domain. These code structures are generic, intended to reduce the cost of development. The flexibility of frameworks are hard to design compared to specific applications as a lot of possibilities and abstractions have to be planned for beforehand. Framework design is its own subject that has a lengthy research history already.

Schmid [1997] and Roberts et al. [1996] are early works discussing the characteristics of framework development. The main element that has to be designed for is variability. Software patterns have to be put in to place that are organized in two parts [Ben-Abdallah et al., 2004]: hot-spots and the core. Hot-spots are places in a system where implementing applications have their own specific adaptation in place. The core is common to all applications derived from the framework.

Hot-spots come in two forms: black box and white box. Black box hot-spots have their variations predefined and implementations can only pick one. White box hot-spots require a developer's own implementation by programming a class or subsystem. These are illustrated in figure 2.2.

Variability itself comes with a few characteristics [Schmid, 1997].

- The common responsibility which overcouples different alternatives.
- different alternatives that realize the common responsibility
- The variability type that depends on the subjects structure.
- The multiplicity of alternatives and the structure of alternatives
- The point in time where the alternative is picked and implemented (fixed or at run-time)

UML design of a framework has to visualize the flexibility and points of variability clearly. This means that some extensions are necessary compared to the usual standards for normal applications. This is a subject which has been researched in a number of papers [Bouassida et al., 2001] [Ben-Abdallah et al., 2004]. The main takeaway is that the points of extension should be clear for developers immediately. Designing of a framework requires abstraction from specific implementations and how this happened in this work will be expanded upon in section 4.1.

No prior work is found on offering a framework for multi-task deep learning, let alone for audio purposes. Closely related is Dcase-Repo [2021], a tool released by the time behind the yearly dcase Dcase competition for audio recognition tasks. This is a standalone library which offers tools for processing audio and metadata files. The designs are mostly based around offering containers which come with functionalities for processing audio and metadata. However, this adds a lot of extra layers to the data which is a problem for verifying different datasets are processed in the same way. This also separates target handling from the inputs, which forms a problem for operations that rely on their connection (e.g. filtering). Training functionalities are present but need to fit in the predefined mold and do not allow multiple different datasets (without having to redefine them as exactly the same dataset). Finally, this requires that data is stored in similar ways to their own datasets which is not always the case. All these elements contribute to the overlying problem for multi-task set-ups that deals with collections of datasets, for which this implementation the handling of data is too individualistic.

One framework that this work did take inspiration from however was De Rainville et al. [2012], a development framework for evolutionary learning. This is a white-box framework that has a lot of parallel necessities in functionalities while also having to take in account the amount of possibilities that still need to be allowed by extension.

Finally, this work is an extension of PyTorch [Paszke et al., 2019], an imperative stile deep learning library. Building on their groundworks, this work offers abstractions to their deep learning modeling and training functionalities to better the development of multi-task networks alongside it. A number of their design principles are kept in place like the pythonic style, the focus on researchers and pragmatic performance.

Table 2.1: Tried Combinations

Title	Tasks	Classifier
Lu et al. [2004]	Automatic Speech Recognition; Speech Enhancement; Gender Detection	RNN
Seltzer and Droppo [2013]	Phone Label Recognition; State context recognition; Phone context recognition	DNN-HMM
Panchapagesan et al. [2016]	Keyword Spotting; Large Vocabulary Continuous Speech Recognition Senones Targets Recognition	DNN
Sakti et al. [2016]	Automatic Speech Recognition; Acoustic Event Detection	DNN
Georgiev [2017] Georgiev et al. [2017]	Speaker Identification; Emotion Detection; Stress Detection; Acoustic Scene Classification	DNN
Kim et al. [2017]	Emotion Detection; Auxiliary tasks: Arousal Level; Valence Level; Gender Detection	CNN
Nwe et al. [2017]	Acoustic Scene Classification (Grouped scenes as different tasks)	CNN
Phan et al. [2017]	Detection error; distance error; optimization confidence	DNN
Sun et al. [2017]	Keyword Spotting; Large Vocabulary Continuous Speech Recognition Phone Targets Recognition	DNN-HMM
Kremer et al. [2018]	Word Error Rate and Character-Level Automatic Speech Recognition	CNN
Morfi and Stowell [2018]	Audio Tagging; Event Activity Detection	DNN
Lee et al. [2019]	Main Tasks: Audio Tagging; Speaker Identification; Speech Command Recognition (Keyword Spotting); Auxiliary Tasks: Next-Step prediction; Noise Reduction; Upsampling	DNN
López-Espejo et al. [2019]	Keyword Spotting; Own-voice/External Speaker Detection	DNN
Meyer [2019]	Speech/Noise detection; Language Identification	CNN + LSTM

Table 2.2: Tried Combinations (Continued)

Title	Tasks	Classifier
Pankajakshan et al. [2019]	Sound Activity Detection (Event Activity Detection); Sound Event Detection (Audio Tagging)	CRNN
Phan et al. [2019]	Detection error; distance error; optimization confidence	CNN
Tonami et al. [2019]	Acoustic Event Detection; Acoustic Scene Classification	CRNN for AED, CNN for ASC
Xia et al. [2019]	Acoustic Event Type Detection (Audio Tagging); Predict frame position information (Event Activity Detection)	CNN
Xu et al. [2019]	Acoustic Event Detection; Acoustic Scene Classification	
Zeng et al. [2019] (1)	Emotion Detection; Music/Speech Classification	DNN
Zeng et al. [2019] (2)	Accent Recognition; Speaker Identification	DNN
Abrol and Sharma [2020]	Fine and Coarse Labels Acoustic Scene Classification	DNN
Deshmukh et al. [2020]	Acoustic Event Detection; Reconstruct Time Frequency Representation of Audio	CNN
Fernando et al. [2020]	Acoustic Event Type Detection (Audio Tagging); Predict Frame Position Information (Event Activity Detection)	LSTM
Huang et al. [2020a]	Audio Tagging; Temporal Detection (Event Activity Detection)	CNN PT/PS model
Huang et al. [2020b]	Audio Tagging; Event Boundary Detection (Event Activity Detection)	CNN
Imoto et al. [2020]	Acoustic Event Detection; Acoustic Scene Recognition	CNN
Jung et al. [2020]	Acoustic Scene Recognition; Audio Tagging	DNN
Komatsu et al. [2020]	Acoustic Event Detection; Acoustic Scene Recognition	CNN

Table 2.3: Tried Combinations (Continued)

Title	Tasks	Classifier
Tagliasacchi et al. [2020]	Keyword Spotting; Speaker Identification; Language Identification; Music/Speech Classification; Bird Audio Detection; Urban Acoustic Scene Classification; Music Instrument Pitch Detection; Music Instrument Detection	CNN
Wu et al. [2020]	Keyword Spotting; Domain Prediction	CNN + LSTM

Chapter 3

Problem Statement

TODO: Explain that this chapter is about defining the problem and what the solving system should be TODO: clarify the analysis performed to identify the requirements

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.
- **Flexible:** The framework should be able to dynamically handle possible differences in input and desired pipeline functionalities.

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. Methods must be available for aid in the 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.

- Create multiple input objects from the same dataset: It is possible that inputs can be created from the same dataset, but require different processing or require different subsets of the information.

- Tasks and datasets are a many to many relationship: Dataset objects can have multiple tasks and the same tasks can be present in multiple dataset objects.

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.This goes for example for example for creating the same train/test splits in a k-fold cross validation set-up. Another case is when the data is scaled based on metrics from the

training data. When a new dataset is then brought in to test a trained model, this data should be scaled using the same metrics.

- Scalable Manipulation: Manipulations executed on the feature matrices from one dataset must easily be able to be performed on all the datasets.

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 calculation specifiable: The user should be able to define how the loss is calculated and utilise it to update the model variables.
- 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.

- Direct comparison of different runs: Grant the ability to visually compare the results of different train/test runs which relate to different variables, design choices, ...

- Variable training paradigms: Offer the ability to train the model parameters using a desired training paradigm set by the user.

Chapter 4

System Design

TODO: What are the key decisions that were made and why?

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

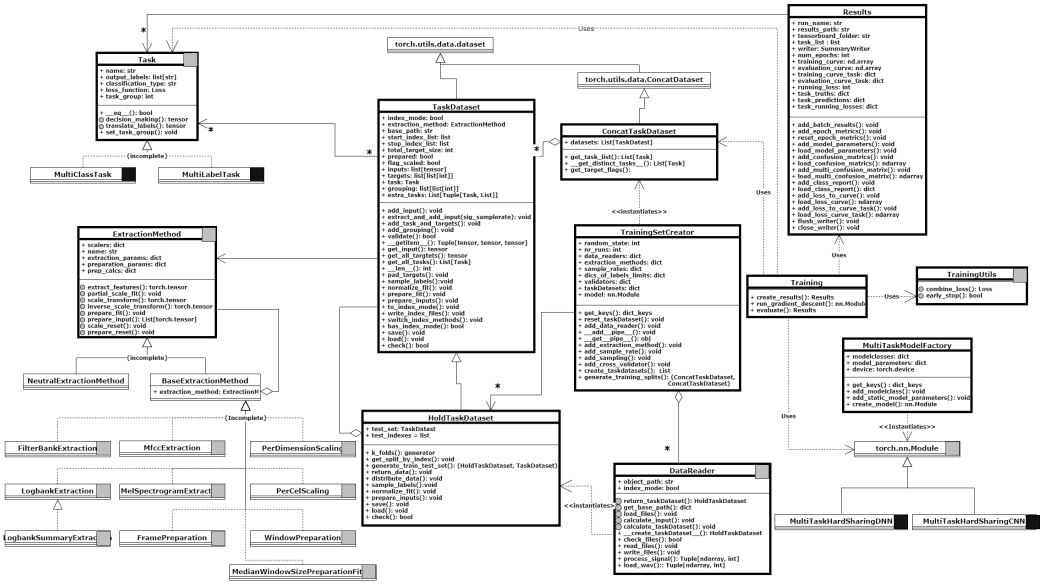

Figure 4.1: Class Diagram

a number of pre-made implementations addressing commonly available features in deep learning.

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.

TODO: Class diagram is pretty detailed, include higher level explanation

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.

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.

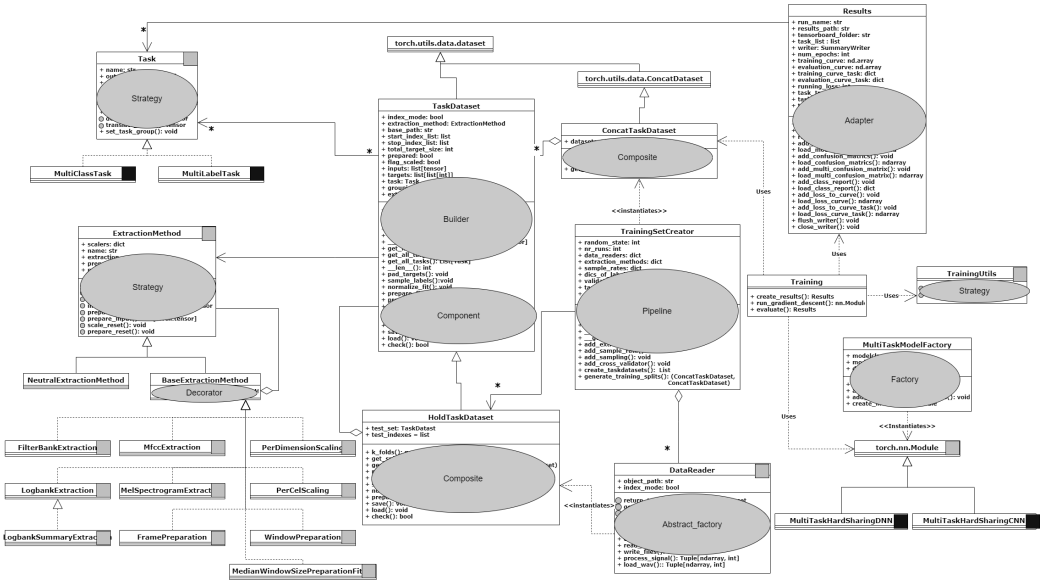

Figure 4.2: The framework's pattern diagram

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 appropriately 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 require 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

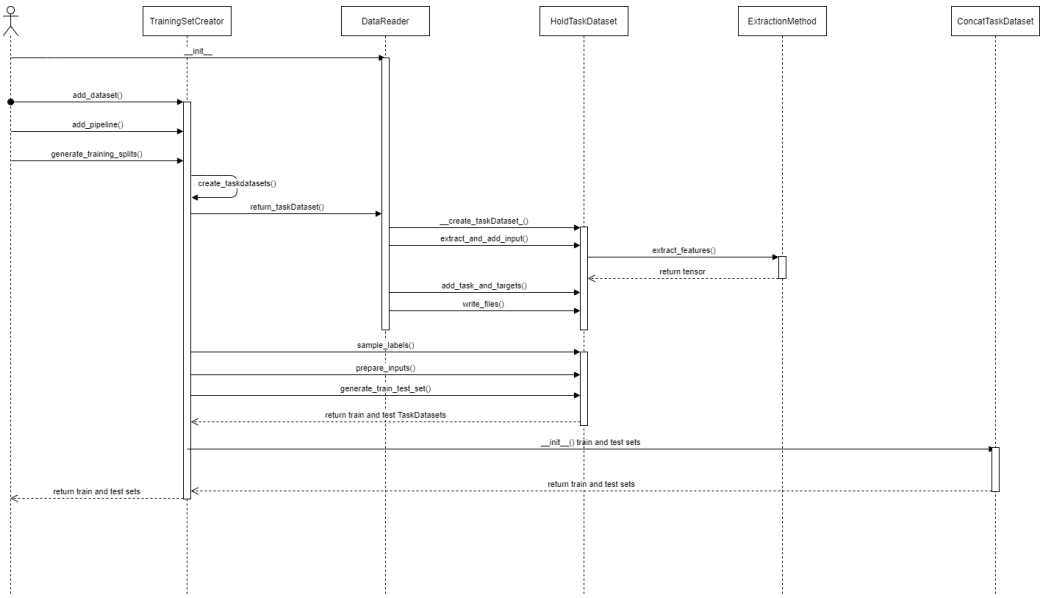

Figure 4.3: Sequence diagram illustrating how the TrainingSetCreator extracts the data from datasets to training and test sets

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 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

TODO: Create visual representation of dev workflow 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

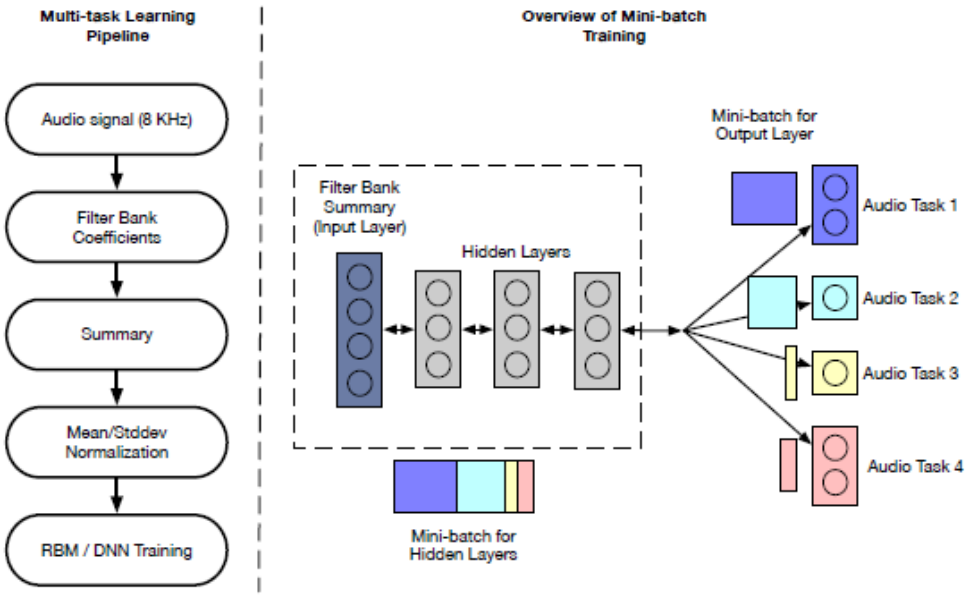

Figure 4.4: The Multi-Task pipeline and Model used in Georgiev et al. [2017]

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 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

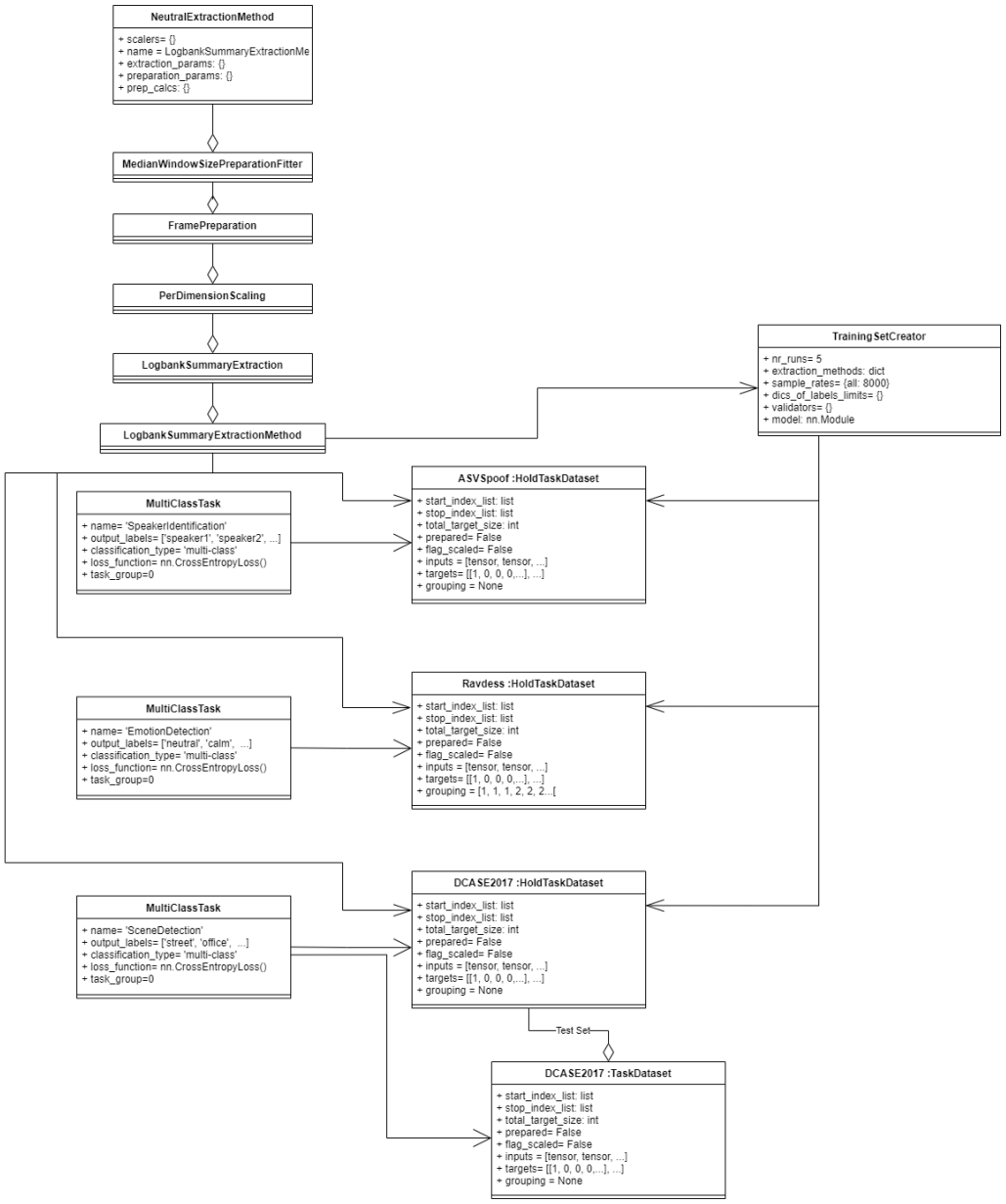

Figure 4.5: Object model instantiating the set-up from Georgiev et al. [2017] after the TaskDatasets are created

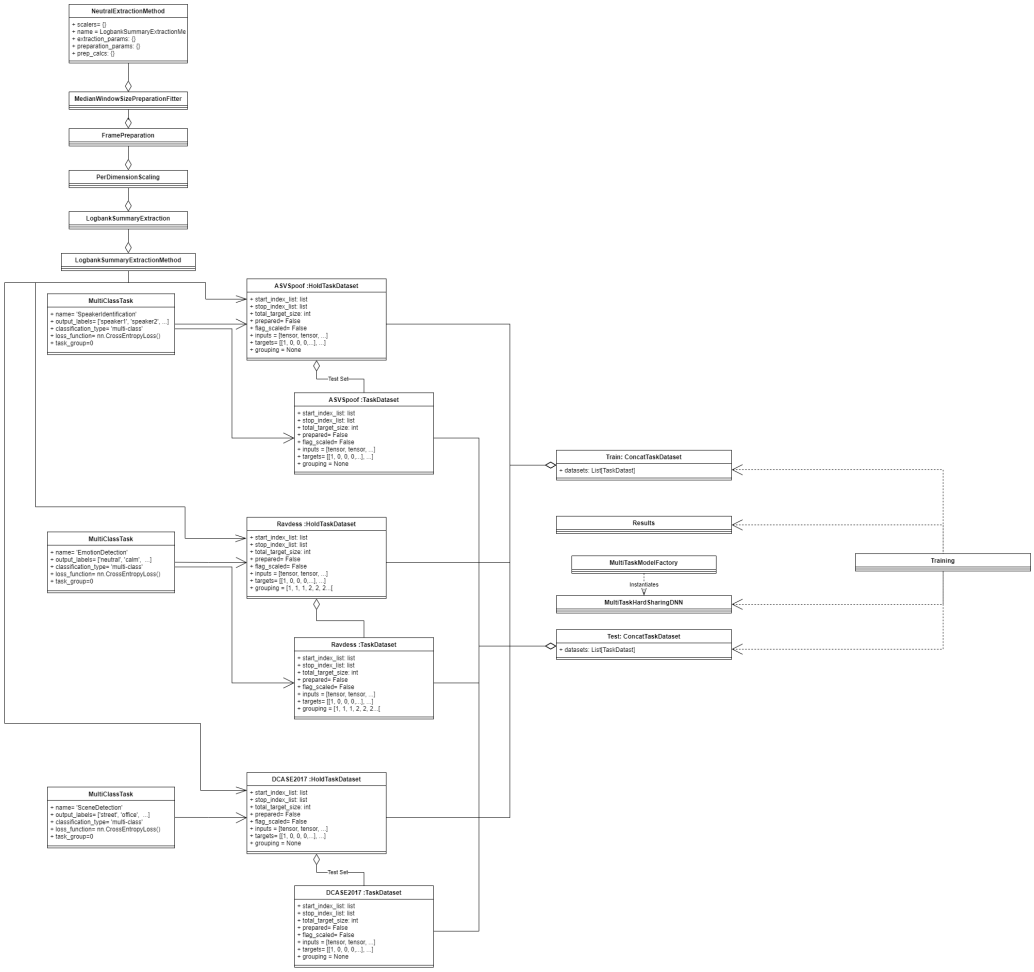

Figure 4.6: Object model instantiating the set-up from Georgiev et al. [2017] after the train and test sets are created along with the input objects for the Training

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 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

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.

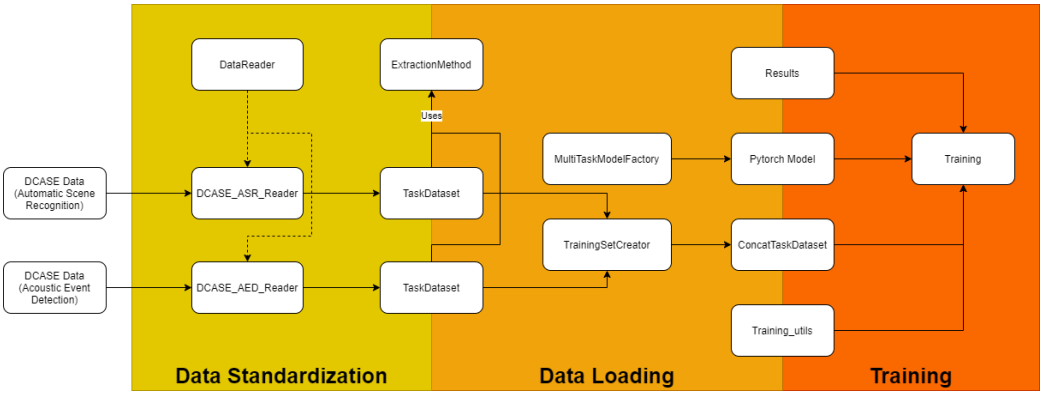

Figure 5.1: Simplified Pipeline Overview

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

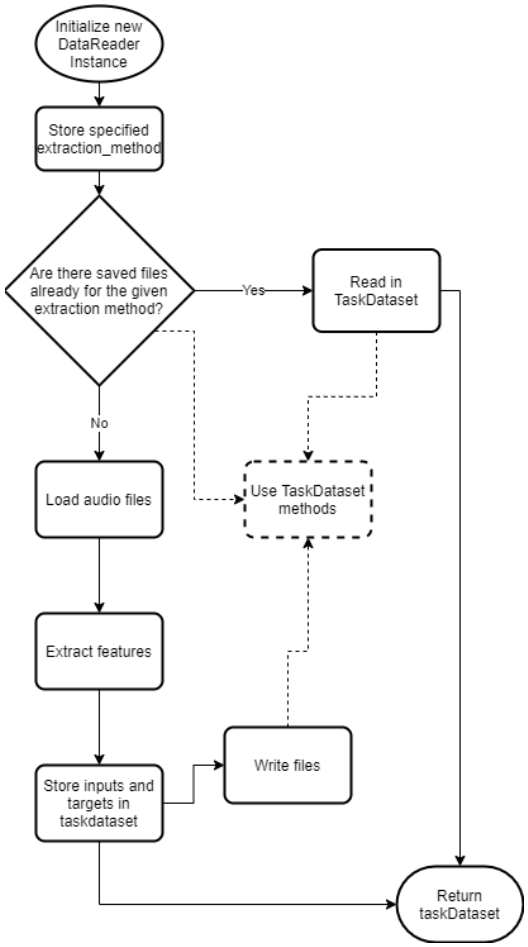
Initialize new DataReader Instance
Store specified extraction_method
Are there saved files already for the given extraction method?
Yes
Read in TaskDataset
No
Load audio files
Use TaskDataset methods
Extract features
Write files
Store inputs and targets in taskdataset
Return taskDataset

Figure 5.2:

```
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.

Chapter 6

Evaluation

6.1 Demonstrate Implementations

Table 6.1: Implementations LOC comparisons

Title	Total LOC	Data Reading LOC	Data Loading LOC	Training LOC
Park et al. [2020] Without	177	75	24 + 2	25 + 10
Park et al. [2020] With	101	1+10+78 = 89	8	3
Georgiev et al. [2017] Without	211	105	46	32+30
Georgiev et al. [2017] With	159	4 +46 +52+40=142	11	3
Xu et al. [2019] Without		50	35	43
Xu et al. [2019] With	92	2 + 46 + 33	8	3
Own Experiments	364	8 + 52 + 78 + 46 + 33 + 49 + 40 + 33 = 339	15	8

As a way of demonstrating the way the framework offers the tools for rapid prototyping as well as reusability and extensibility, a number of imple-

mentation tasks have been made. For evaluating these aspects, the number of lines of code (LOC) for each implementation is shown, compared to implementations that have been made using pytorch, without the framework. The implementations that were chosen were multi-task set-ups described in published works, along with an implementation which is set-up to variate and analyse a number of different elements in the multi-task pipeline. Since this framework has a focus to be utilised in research, it is important to demonstrate that its results align with those reported in published results as well. The framework implementations shouldn't differ significantly from the reported results, even if they deviate due to numerous small implementation details that would unavoidably differ from the original work.

The results of the implementations with their LOC are given in table 6.1. For each implementation, things like imports, empty lines and debugging logic are ignored. Another thing that is not counted are the LOC for the actual models, as they are solely part of the PyTorch framework and their implementation would be the same without the extending framework. Each implementation without the framework covers the basic training and testing of models, which imply the three stages of the multitask deep learning pipeline mentioned earlier, namely Data Reading, Data Loading and Training. Each stage is mentioned separately in terms of LOC, to demonstrate the amount of work cutting that happens. Aside from training and evaluating models, metric calculation and visualization is handled in the same way as the framework does itself, which means calculation through sklearns metrics toolbox and visualization through TensorBoard. However, additional visualizations that are present in the framework, like those of the confusion matrices and the loss, as well as the additional storing and checkpointing that happens are not covered in those implementations. Solely the work required to replicate the original work is implemented. LOC in Data Reading are split up for each experiment that uses the framework, with the first element being the calling of the DataReader class and the subsequent elements being the separate implementations. What was included for each section goes as follows. Data Reading covers iterating over the dataset and extracting the data to a form which functions as a readable collection that can later be in turn iterated over and fed to the model. Data Loading is taking that extracted data, applying the necessary transformations so that the Training stage can simply receive and process the instances. Training then includes creating and training the model with unaltered data from the previous stage.

First thing which can be noted from looking at the results is the fact that Data Reading consistently comes out higher than without the framework. The reason for this is that the DataReaders do not really offer a lot of abstractions for simplifying reading datasets into usable forms for input and training of models. What they mainly do offer are quick extra quality of life features like quick reading, as well as functionalities that allow variations, which in turn can significantly reduce later work. This reduction does not only go for reusing the Data Reading structures, but is shown apparent in the significant decrease for the data loading and training sections. The extra lines of code required for the Data Reading almost exclusively come in the form of the function definitions and outputs, with the exception of having to add getter functions for the task name and the storage location that the quick reading functionalities use.

Park et al. [2020] The first implementation Park et al. [2020] targets a paper which does not actually describe a multi-task framework, but a single task one. The LOC comparisons are given in table 6.1 and the comparison between the reported results and the framework results in **??**. The feature extraction - vectors outputted by the VGGish autoencoder TODO: REFERENCE - was not present yet in the framework, so had to be defined as an ExtractionMethod object following the framework. This only takes three more lines, which are function definitions. To explain the sum of LOC: the first 1 is how much LOC is required to call the complete data reading functionalities in the eventual experiment. The 10 LOC is for implementing the ExtractionMethod object, which was not covered yet by the base framework. Last LOC are for the actual implementation of extracting the data from datasets, which come close to the original required amount of LOC.

[Park et al., 2020] describes its results for two cases. In one, it has a label of leftovers which it limits to 500 instances, as to not be disproportionately present in training. In the other, it additionally removes the label 'speech' from its instances. These fall under the Data Loading section of the process, which is a feature covered by the framework. The data has to be split into a train and test set, which is also covered in the framework. These two elements explain how even single task set-ups can get significant reductions in required lines in its Data Loading process as seen in the table.

Explaining the reduction for training is pretty simple, as the training loop - loading batches of data and updating the model - is covered in one function in the framework, which also covers visualization of the results. A training loop becomes as simple as creating the model, creating the Results object and inserting the necessary information in the training loop.

Georgiev et al. [2017] In the next implementation from [Georgiev et al., 2017], 4 tasks are taken from 3 datasets, but a different clip length has to be taken for the two tasks that come from the same dataset. Essentially, two different datasets must be extracted from the same dataset, with one being a subset of the other. The change in clip length can be added on the fly, using the DataReader's time_split_signal function, but otherwise, the same applies as before, with the framework offering little in the way of line cutting abstractions. What can be noted in the other sections, it that they don't really require more LOC even with the increase in datasets. The data loading section only requires more lines than the previous, due to the fact that the data must be scaled and every possible combination of the datasets must be created and compared. These results demonstrate the power of the TrainingSetCreator and the training functions, which easily scales operations in terms of added datasets and tasks. Training implementation becomes more complex from the previous case, due to the padding required to combine multiple targets of different lengths in the same batch. The correct task losses must be updated according to the instances that where given in. Additionally, the DCASE Dataset already has a test set defined which must be connected with the other datasets that require splitting. Normalization of the data must also be done after the train/test splits are made, as they shouldn't be normalized using test data which is not seen. All these complicating factors are handled by the framework automatically, where the TrainingSetCreator handles the correct execution order of defined transformations, reducing their call to singular lines.

Xu et al. [2019] Following that is the case of [Xu et al., 2019]. This connects a multi-label and multi-class task, which require different ways of handling of the model output for loss and metric calculation. In the table, the LOC in the Data Reading table are split up according to the dataset that they are handling. The 46 lines are from the same DCASE dataset used in the previous implementation. These can effectively be reused in the scenario that the previous implementation was already made, with the differences

in extraction be achieved through given inputs, adding no additional LOC whatsoever. In that scenario, jumping from the previous scenario would be impeded by the fact that the extraction method - melspectrogram features - does not automatically result in same size feature matrices, which infers that some sort of framing or windowing mechanism is required. On top of that foresight was required to reutilize the previous implementation's code to allow for a different extraction technique. For the framework, this is as simple as giving in a different ExtractionMethod object - decorated with the desired preparation functions - and calling the preparation operation on the TrainingSetCreator. Again, it can be seen that the training LOC stays consistent and the Data Loading doesn't necessitate further additions compared to previous cases. This also demonstrates the reusability of the code once it is implemented.

Variation Experiment In this work, the different combinations of a large set of datasets and tasks are tested, variating models and extraction methods. The experiment recreates a scenario that research might face, requiring large combinations, extensive variations and heterogenic task types. The aim here is not to provide new insights into the multi-task set-up or the results, but to evaluate the model's ability for combination and variation demonstrated by how the coding requirements scale compared to the previous cases. Each of the previous cases' datasets and tasks are included in this experiment. In essence this would also provide with a concept for how new research can easily be built of previous work.

In total 7 datasets were used in creating this set-up, with 8 tasks. The dataset linked to two tasks is the same used earlier for [Georgiev et al., 2017]. The tasks contain both multi-class and multi-label tasks. Each combination is tested for 2 extraction methods and 2 models. For each run, the feature matrices should be framed in the same size, based on the average feature matrix size and the data normalized. As can be seen in table 6.1, this is the first time the required Training LOC makes a significant jump in the Training stage, simply due to the fact that it varies models. The models are loaded for the first time through the MultiTaskModel factory which mainly functions as a way to concentrate static and dynamic model parameters for creation and isn't absolutely necessary. Otherwise, the training is performed using the same three lines as before: model creation, results creation and training loop instantiation. The Data Loading stage also takes a jump, but is in no

```
csc = ConcatTrainingSetCreator(random_state=123,
                               nr_runs=4,
                               index_mode=False,
                               recalculate=False)
csc.add_data_reader(ASVspoof2015(object_path=os.path.join(data_base, 'ASVspoof2015_{}'),
 ...

csc.add_sample_rate(sample_rate=32000)
csc.add_signal_preprocessing(preprocess_dict=dict(mono=True))
for ex in range(2):
    csc.add_extraction_method(
        MelSpectrogramExtractionMethod(**extraction_params)) if ex == 0 else csc.add_extraction_method(
        MFCCExtractionMethod(**extraction_params))
    csc.add_transformation_call('prepare_fit')
    csc.add_transformation_call('prepare_inputs')
    csc.add_transformation_call('normalize_fit')
    csc.add_transformation_call('normalize_inputs')

    keys = list(csc.get_keys())
    comb_iterator = itertools.chain(*map(lambda x: itertools.combinations(keys, x), range(1, len(keys) + 1)))

    for combo in comb_iterator:
        key_list = list(combo)
        for train, test, fold in csc.generate_training_splits(key_list):
            for i in range(2):
                model = mtmf.create_model(MultiTaskHardSharing.__name__,
                                          input_size=train.datasets[0].get_input(0).flatten().shape[0],
                                          task_list=train.get_task_list()) if i == 1 else mtmf.create_model(
                    MultiTaskHardSharingConvolutional.__name__,
                    task_list=train.get_task_list())
```

Figure 6.1: Data Loading in the variation experiments

way related to the high and diverse amount of datasets, simply the variation of elements in the pipeline. There are operations performed in the pipeline: resampling, conversion to mono, calculating and framing the feature matrices, normalizing the data. The Data loading code can be found in figure 6.1, which makes it apparent that the extra lines outside the transformation calls are simply due to iterating over the required variations. The actual amount of LOC relating to direct data loading operations is 8.

This example makes it clear that all the work concerning datasets comes beforehand in the Data Reading stage, with little extra effort on the developer's side, while the combinatorial aspects are handled by the framework in the background. Nothing has to be explicitly reloaded or recalculated, as seen when adding the ExtractionMethod, by the developer as variations will be handled by the framework and necessary data recalculated when required. The framework thus reduces research variations to singular line changes.

To build on the last point of reducing work for research variations, especially in future work, it should be noted that given that the previous implementations would have been made as was the case in this scenario, only two extra datasets were added: FSD Kaggle 2018 TODO: REFERENCE and the Speech Commands dataset TODO: Reference. Given that, the Data Reading would be reduced to 8 + 83 LOC in this implementation. DataReader objects are merely paths for extracting the data, while the specifics of how can be given later. Vanilla implementations would either require large code changes if e.g. other extraction methods or signal preprocessing functions were required or have to take these in account beforehand and likely end up with similar structures.

What these efforts demonstrate is that the LOC only directly scale with added required operations and do not spill over in other stages. To clarify the last part, figures 6.2 illustrate how without the framework, train and test set generation - which falls under the data loading stage - scales directly depending on the amount of datasets used. Compare that to figure 6.3, where it can be seen that train and test generation is actually reduced to one simple line before the three Training stage lines in the end. This also demonstrates how the framework grants flexibility in which datasets to actually load and split with the key_list input, which grants further work reduction as in practice it is possible that not all datasets are required which were planned beforehand.

```
kf = GroupKFold(n_splits=5)
ravdess_splitter = kf.split(inputs['ravdess_em'], groups=groupings['ravdess_em'])
asv_splitter = kf.split(inputs['asv'], groups=groupings['asv'])
for train_idx, test_idx in ravdess_splitter:
    ravdess_em_dataset_train = Data(
        inputs=[inputs['ravdess_em'][i] for i in range(len(inputs['ravdess_em'])) if i in train_idx],
        targets=[targets['ravdess_em'][i] for i in range(len(inputs['ravdess_em'])) if i in train_idx])
    ravdess_em_dataset_test = Data(
        inputs=[inputs['ravdess_em'][i] for i in range(len(inputs['ravdess_em'])) if i in test_idx],
        targets=[targets['ravdess_em'][i] for i in range(len(inputs['ravdess_em'])) if i in test_idx])
    ravdess_s_dataset_train = Data(
        inputs=[inputs['ravdess_s'][i] for i in range(len(inputs['ravdess_s'])) if i in train_idx],
        targets=[targets['ravdess_s'][i] for i in range(len(inputs['ravdess_s'])) if i in train_idx])
    ravdess_s_dataset_test = Data(
        inputs=[inputs['ravdess_s'][i] for i in range(len(inputs['ravdess_s'])) if i in test_idx],
        targets=[targets['ravdess_s'][i] for i in range(len(inputs['ravdess_s'])) if i in test_idx])
    train_idx_asv, test_idx_asv = next(asv_splitter)
    asv_dataset_train = Data(
        inputs=[inputs['asv'][i] for i in range(len(inputs['asv'])) if i in train_idx_asv],
        targets=[targets['asv'][i] for i in range(len(inputs['asv'])) if i in train_idx_asv])
    asv_dataset_test = Data(
        inputs=[inputs['asv'][i] for i in range(len(inputs['asv'])) if i in test_idx_asv],
        targets=[targets['asv'][i] for i in range(len(inputs['asv'])) if i in test_idx_asv])
    training = ConcatDataset(
        datasets=[dcase_train_dataset, ravdess_em_dataset_train, ravdess_s_dataset_train, asv_dataset_train])
    testing = ConcatDataset(
        datasets=[dcase_test_dataset, ravdess_em_dataset_test, ravdess_s_dataset_test, asv_dataset_test])
```

Figure 6.2: Train and Test set creation without framework for the Georgiev et al. [2017] implementation

```
csc = ConcatTrainingSetCreator(nr_runs=5)
csc.add_data_reader(DCASE2017_SS(object_path=drive + r'Thesis_Results\Data_Readers\DCASE2017_SS_{}',
                                 data_path=drive + r'Thesis_Datasets\DCASE2017'))
csc.add_data_reader(Ravdess(object_path=drive + r'Thesis_Results\Data_Readers\Ravdess',
                            data_path=drive + r'Thesis_Datasets\Ravdess'))
csc.add_data_reader(Ravdess(object_path=drive + r'Thesis_Results\Data_Readers\Ravdess',
                            data_path=drive + r'Thesis_Datasets\Ravdess',
                            mode=1))
csc.add_data_reader(ASVspoof2015(object_path=os.path.join(drive, r'Thesis_Results\Data_Readers\ASVspoof2015'),
                                 data_path=drive + r'Thesis_Datasets\Automatic Speaker Verification Spoofing and Co
csc.add_sample_rate(8000)
csc.add_extraction_method(extraction_method=LogbankSummaryExtraction(
    PerCelScaling(NeutralExtractionMethod()),
    extraction_params=dict(winlen=0.03, winstep=0.01, nfilt=24, nfft=512)), multiply=False)
csc.add_transformation_call('normalize_fit')
csc.add_transformation_call('normalize_inputs')

mtmf = MultiTaskModelFactory()
mtmf.add_modelclass(MultiTaskHardSharing)
mtmf.add_static_model_parameters(MultiTaskHardSharing.__name__, **{"hidden_size": 512, "n_hidden": 4})
keys = list(csc.get_keys())
comb_iterator = itertools.chain(*map(lambda x: itertools.combinations(keys, x), range(1, len(keys) + 1)))

for combo in comb_iterator:
    key_list = list(combo)
    for train, test, fold in csc.generate_training_splits(key_list):
        model = mtmf.create_model(MultiTaskHardSharing.__name__,
                                  input_size=train.datasets[0].get_input(0).flatten().shape[0],
                                  task_list=train.get_task_list())
        results = Training.create_results(modelname=model.name, task_list=train.get_task_list(), fold=fold,
                                          results_path=os.path.join(drive, 'Thesis_Results'), num_epochs=200)
        Training.run_gradient_descent(model=model, concat_dataset=train, test_dataset=test, results=results,
                                      learning_rate=0.001, batch_size=32, num_epochs=200)
```

Figure 6.3: Complete implementation of Georgiev et al. [2017] with the framework

In the first case, that would imply multiple lines of code change, while in the second, simply one line for the key list.

What should again be noted, is that the implementations without the framework did not include a lot of the extra features that are performed automatically, the main one being quick reading and writing of the data. To reiterate, this means that when the Data Reading stage is done, it is written to files on the disk, so that future runs would not unnecessarily have to reextract feature matrices. Including these would raise the LOC required for the vanilla implementations a lot and would either have specific implementations per dataset or end up constructing similar functions to the framework. These LOC comparisons are for the bare required necessities only.

6.2 Literature Evaluation

In line with the goal to provide a tool that can spur development of multi-task research, this section will examine the papers identified in tables 2.1, 2.2, 2.3 in order to evaluate the expressiveness of the framework as well as its limitations. Specifically, the examination will be made in terms of changes that are required to achieve some of the necessary features. The feature analysis of the framework is performed by analysing the papers as problem contexts for identifying model problems [Brown Alan and Wallnau Kurt, 1996]. The goal is to identify possible limitations to the architecture and possible future extensions that are required.

To do this, an examination of specific techniques that are present in the literature is made and discussed to what degree and facility they can be implemented by the system. Mainly techniques that haven't been clearly been addressed in previous sections, but have presence in the literature are brought up here. From this, conclusions are drawn towards what future development can focus on to provide a more complete framework for multi-task development and what the limitations are of the (current) design.

Input Techniques. The first set of techniques are related to input requirements. The framework's data structure for encapsulating inputs is a list of tensors that are, by index, related to one or multiple targets. Each input

can either be added directly as a tensor or as a signal from which the feature matrix is extracted on input using the TaskDataset's ExtractionMethod object. In any case, once the data is encapsulated within the TaskDataset, it is always a feature matrix which is ready to serve as input for the training function. There are (limited) functionalities available for transforming audio signals before extraction through the DataReader's process_signal function.

While most research utilize **mono channel** audio signals Seltzer and Droppo [2013] Panchapagesan et al. [2016] Kim et al. [2017], there are instances where **multi-channel signals** are used, even as much as four Nwe et al. [2017]. The framework's extraction methodology already takes this situation into account, creating parallel feature matrices and concatenating them into a singular input instance. In some instances though, stereo input signals are transformed to mono channel signals in order to reduce the feature matrix size. This situation is already covered as well as part of the process_signal function, where a multi-channel signal is averaged to a single channel at the command of an input boolean, meaning it can varied at runtime.

However, this brings up the way **signal processing** is handled in the framework. Its limitations become apparent when more signal transformations need to be performed. For example the data augmentations described in López-Espejo et al. [2019] would be a problem in the current setting. In this scenario, multiple randomized signal transformations are applied to a single fragment, before MFCC features are extracted. In order to scale this operation for multiple datasets, the developer has two options. One is to create an extension to the DataReader class which are in turn extended by each of their DataReaders for the individual datasets. The other is to store the signal as a tensor and encapsulate the data augmentation techniques in the preparation function of the ExtractionMethod class. Sadly, there are complicating factors which do not make the described augmentation method possible, as additional noise segments from random files in their dataset as well as the augmentation being recalculated every epoch for 30% of the dataset.

This case thus highlights two problems in the signal preprocessing approach: it is messy to create extra pre-processing functionalities and it is impossible or very complicated to vary steps in creation of the input between epochs. At first sight, creating a standardized approach for addressing this

specific situation would be complicated due to the reliance on an additional dataset for noise fragments. The TaskDataset structures are not designed with access to the original dataset in mind, nor do they currently offer any storage of additional information.

Moving on to examining what singular **input instances** actually are. In pretty much all the research, an input instance is a single feature matrix, which is the result of an extraction methodology applied to an audio signal. However, two works challenge this form.

The first one is Fernando et al. [2020], where the **input for a single audio fragment comes in three forms**: The time series, the MFCC features and the MFCC deltas. These three representations are combined into a single LSTM encoder which results in the actual input embedding. This embedding is then used along with a random noise vector as an input for a Generative Adverserial Network (GAN). However this network is trained (encoder LSTM trained separately or together with the rest of the Multi-Task network), there is no issue as tensors can be created which are concatenated from matrices with different dimensions. The one caveat however is that, in order to batch multiple instances, the individual tensor dimensions must be equal, but this is always the case anyway, which is why windowing transformations are standard available. The current implementations however only take two dimensional matrices in account. If every instance does have variable shapes, a custom DataLoader must be created, which would be the case anyway without the framework. No extra complications are introduced due to the option to input the DataLoader in the training and evaluation functions.

A more difficult situation however is found in Komatsu et al. [2020], where **target labels from one task are served as input** for the additional task. Targets are not given as input for the model in training, which means that these should be included as part of the input matrix. The issues that this causes however are minimal, as it means that target vectors have to be stored twice, which wastes space.

These cases impose less severe issues, but do require forms of extraction and forming the classifier in a way which would inhibit easy addition of new datasets. Besides, the models would also have to include steps to split up

these concatenated tensors, which possibly introduce too many undesired calculations. More flexibility to deal with these situations could be provided by allowing multiple input instances per fragment, similar to how multiple targets are possible per instance.

Target Techniques. The next category includes techniques related to how the targets are formed. Targets are assumed to be vectors and are stored as lists of integers. In the current setting, targets are labels which are numeric, either discrete, representing categories or continuous for regression tasks. In short, the targets are made for supervised labeled learning tasks.

Of course this makes the current framework not optimised to store any other **target structure**. A first example of this comes in Lu et al. [2004]. In this work, speech enhancement is implemented as an additional task to improve the automatic speech recognition task. Speech enhancement requires **clean speech signals as targets** on which a loss function is calculated to measure the discrepancy between noisy and clean signals. Creating an extension to the TaskDataset to allow targets other than labels doesn't sound too complicated. The issue comes when combining multiple taskdatasets.

In order to allow for batching operations, which combines instances of targets in a singular matrix. However, in order to allow an unspecified amount of tasks - coming from tasks which may or may not be present within the TaskDataset at hand - target labels are combined within a single target vector, with dummy **padding for the targets** that are not present. When targets become multidimensional, this structure consequently falls short. This signals a requirement more sophisticated ways to encapsulate multiple instances for input matrices and targets, but has to pay attention not to introduce unnecessary computational complexity for the simpler cases.

Mentioned above is the fact that the system is currently based around supervised, labeled learning tasks. **Self-supervised tasks** like in Lee et al. [2019] are not that much of a problem. As these tasks are based on loss functions which calculate on internal parameters or outputs (e.g. distance metrics), these can be either added in the form of a task object with dummy labels or directly extended in the Training_Util's loss calculation function. Tasks have to have target labels, which do add small unnecessary extra memory usage. The loss functions themselves can be formed as normal PyTorch loss functions, which means no extra work load is introduced. An extra op-

tion is to include it in the model itself.

Adaptation mechanisms like the gating mechanism in Tagliasacchi et al. [2020] are in the same vein as self-supervised tasks. In Tagliasacchi et al. [2020], an internal gating mechanism for prediction outputs is created, which adapts based on a cost function of utilising more channels. This cost function is combined with the normal loss calculation and the model is optimized through the regular back propagation method used. Each task has its own cost function calculated, but also can either be implemented within the model itself or added as an extra task with dummy labels.

The final case is that presented in Wu et al. [2020]. The work presents a multi-task model, with two parallel models that in one model uses the output of an internal layer from the parallel classifier. The catch is that the parallel **classifier is pre-trained**, so its output would not be required to be utilised in updating the multi-task model, nor would its layers be updated. The framework would not even require any extension for this case, as the training set can simply not contain the extra task, but the testing set would. The model creation is entirely modularly separated from the rest of the framework, so different subsections and combinations of the dataset can be used for the model training and evaluation. As long as it arrives in the form of a TaskDataset, the model will be fed (batched) instances.

Loss Functions. In this part, methodologies concerning loss functions are examined. The current methodology places one loss function per task. These are calculated for each task in each batch separately, after which they are combined. The combined loss is used for the backpropagation update of the network. Both the separate loss calculations and combination of those losses are part of the functions in the Training_Utils object. Remember that earlier it was stated the training and evaluation loops were designed not to be touched, but the functions included in the Training_Utils object do offer modification options which can be varied at runtime, by instantiating a different Training_Utils object.

With the current systems in place, the target structure from earlier is assumed and the different tasks' outputs are taken by breaking up the target vectors. Any amount of different supervised is thus not a problem, with the instances of the batch they apply to also being taken care of. The combina-

tion simply sums up the losses. The separate calculations and combination of those losses are two separate functions, so a small adaptation like Park et al. [2020] of **introducing weights** can simply be added in that extension. Loss functions, contained in the Task objects, are given at instantiation and can be varied at runtime.

This structure of doing things, does however require the implementations of those functions to work with what was received in the training loop. Neither receive the input feature matrices. This can be limiting to **teacher student training models** like in Imoto et al. [2020]. This methodology is similar to the parallel models in Wu et al. [2020], except that the parallel model is only used in the training phase as a way to train one of the task heads. The loss is not calculated on predefined ground truth labels but labels outputted in the parallel pre-trained model. This is only a problem however if the parallel model's outputs have to be calculated at runtime for some reason however, in stead of pre-predicting the instances and storing them as normal targets in a TaskDataset.

Finally, it has been mentioned often, but the training loop is only targeted at **gradient descent** based training, following PyTorch's methodology for performing these. However, forms of training that can not be written this way would require their own training implementation. This also limits some forms of NN to utilize the current training loop. In Georgiev [2017] for example, a RBM network is additionally created which requires greedy layer wise pre-training, which would not be possible through the framework.

Georgiev [2017] also shows the last technique which would not be possible in the current state, which is **mini-batch SGD**. There is a reason the training loop function is called run_gradient_descent as it does implement a standard gradient descent loop. In many ways this also illustrates why the choice was made to require the developer to make their own implementation of the training loop. The loop consists of the standard gradient descent training loop which elements can be individually interchanged. Other training procedures could be and possibly bloated to create standardized solutions for, while the developer can still utilize the same building blocks that offer its variability qualities.

Conclusions

After examining the literature, some conclusions can be drawn regarding what the framework lacks or can be improved in. In general, the main limiting factors regarding the optimal implementation of research methodologies can be found in the data structures for encapsulating the inputs and targets and the rigid structure of the training mechanism.

The signal pre-processing, which happens in the DataReader classes, by utilizing their shared preprocessing function could use a more scalable, extendible format. Currently either a new extension to the DataReader class has to be made. Furthermore, the addition of allowing multiple feature matrices for one data instance should be considered. These input related issues do not prohibit any implementation to be made but do require circuitous solutions due to the framework.

In the target structures are more severe issues for implementing certain techniques. Allowing other target forms from labels are crucial for certain tasks even within supervised learning problems. This in turn would also require the way the system combines targets for batching to change from the padding that currently. Less severe is that tasks with loss functions should become possible without targets which would allow for more straight forward multi-task implementations which optimize for non label based loss functions as well, but there are workarounds as discussed.

In the sense of the training function, the rigid structure which requires the developers own implementation if changes are required that can not be achieved by extending the Training_Utils class. Adding small elements like mini-batching requires rewriting the training algorithm completely. Also does the training mechanism only function for gradient descent based training.

6.3 Fulfilment of the Requirements

This section contains a point by point discussion of the designated functional and non-functional requirements

6.3.1 Non-functional Requirements

- **Modular:** The framework is largely modular as a central design point. The framework presents developers with the loose tools to design and implement a multi-task pipeline at different levels. The DataReader gives a loose structure to extract a dataset's inputs, targets and additional task information. The DataReader's open design merely gives a common structure and tools to create a TaskDataset, as well as implement the quick reading mechanism, yet has no obligation to output a TaskDataset object. The TaskDataset and related structures offer a unified way to encapsulate the extracted dataset. The training and evaluation loop offer the ability to load batches of inputs and targets from a PyTorch Dataset, utilise them to predict the outcome using the neural network. To compose the pipeline, the TrainingSetCreator structure is in place, which creates the path from the raw datasets to the processed and transformed inputs for the training and evaluation mechanisms. The TaskDataset's use in the taining mechanisms and the TrainingSetCreator is not a full on dependency but does require specific requirements for an alternative to be met. For the Training and Evaluation mechanism, this is nothing more than the fact that an input and target batch are outputted in tuple form when getting an individual instance. If so, then it doesn't matter what type of Pytorch's Dataset is given as its input. The TrainingSetCreator however does utilise the added functions of the TaskDataset and the HoldTaskDataset. While this does add a dependency on the TaskDataset structure, the different phases - Data Reading, Data Loading and Training - are still completely independent from each other. Whithin each of those phases, there are numerous different elements that have been shown to be easily variable either at implementation level or at runtime, depending of the necessity.

- **Extendible:** The design's extensibility is discussed in its own implementation section 5.7. The framework builds atop of PyTorch's basics for the static data structures, which mimics its own extensibility. Every implementing class has constraints which would ensure its applicability in the rest of the system in case a developer needs to develop an extending implementation. The non-static datastructures, which refer to functions like the training loop and the TrainingSetCreator, make sure

to keep its dependencies on a function basis, meaning that every function deals with one type of data structure and thus only those functions need to be extended.

The Framework's extensibility is mostly open due to its modularity. As the few requirements for the usage of objects in other classes are met, every functionality and more can be changed or added. Mainly the training and evaluation functions are defined pretty rigidly in the sense that they have to be completely reimplemented in certain cases. Otherwise, the extraction methods, the data transformations, the tasks, the loss functions and many more described above can be extended and introduced, without having to touch any other part of the framework. Section 6.2 offers insight to the limits of the extensibility compared to the requirements that were posed in the literature. Especially the transformations of Datasets are completely open for extensibility even with the TrainingSetCreator, as they are called through their naming convention.

- **Fast prototyping:** The speed of prototyping is achieved in two ways in the framework. The first is presented in section 6.1, where it shows how a lot of papers' described methodologies can be shortened through usage of the system's functions. The second lies in the system's TrainingSetCreator structure, which automatically will form defined dataset pipelines and recalculate them if new parts are introduced. When fine tuning a new system, the developer can thus change variables and functionalities while the TrainingSetCreator will now what parts to recalculate on the fly. This way, both set implementations as well as iteratively designed and fine-tuned implementations of networks have reduced work loads.

- **Cutting Double Work:** In section 6.1 as well as the previous item, it is discussed how the framework handles pipelines to avoid double recalculations, with section 6.2 discussing it limits. However, finding the right meta-parameters and optimizing the process steps get cut through the TrainingSetCreator's management of pipeline parts. Furthermore, the quick reading functionality standardly offered through the DataReader prevents unnecessary recalculation of features which can take up a lot of computational resources.

- **Flexible:** The flexibility as well as its limits have been thoroughly discussed in secion 6.2 and the previous chapters. To summarize, the framework optimizes for numerous supervised learning cases, but still has some blind spots, especially for non-supervised training set-ups. When the data is encapsulated as a TaskDataset, the training and evaluation loops will have no trouble processing regardless of what it encapsulates.

6.3.2 Funtional Requirements

Data Reading

- Standardizing Input - The TaskDataset object is developed for assuring that the data is valid throughout the rest of the process. Its extension of PyTorch's Dataset class ensures that it can be utilised by the PyTorch framework. The builder pattern allows the TaskDataset to be built incrementally and valid along the way, with each step including various validity checks. The exception where the TaskDataset can't check for validity is in terms of the input feature matrix size. The matrix sizes might not be compatible with the developed PyTorch Model. The responsibility for this is up to the developer.

- Handling dataset differences - The DataReader class is an abstract class that the developer must extend to deal with the peculiarities of navigating each dataset structure to extract the correct information. This corresponds to it being a white box hot-spot. Predefined train/test splits can be stored through the HoldTaskDataset structure and pre-split audio segments can be kept together by defining the grouping.

- Scalable preprocessing - Preprocessing audio signals and preprocessing feature matrices happen in different places, as TaskDatasets should only contain valid input instances at any point. Preprocessing signals can utilise an (optional) function from the DataReader class with parameters that are received when the TaskDataset is extracted. Reusing the method can thus hand developers easy replicability of the signal preprocessing. These can be further scaled by using the TrainingSetCreator. In this class, any preprocessing or transformation can be added 'on the fly'. This means that if a functionality (e.g. resampling) is added, any previous

- File storage abstraction: There are handles on the TaskDataset which can be called to store, load or check the TaskDataset to or from files, which are specific for the currently used extraction method and task.
- Quick Reading: The DataReader automatically checks if there is a stored TaskDataset available for the given extraction_method and task and loads it if so.
- Create multiple input objects from the same dataset: The framework is open ended in how the TaskDataset object is extracted from the data and allows extra parameters for the DataReader to be given at initialization. TaskDatasets are stored using the ExtractionMethod object's name and the (main) Task's name, so for every new variation of these will be automatically linked to different files.
- Tasks and datasets are a many to many relationship: Tasks can be present in multiple datasets. The tasks need to have the same name, output labels and classification type in order to be seen as the same. When combined in the ConcatTaskDataset, the target vectors will automatically be placed in the same positions, which will make them be seen as the same task in the training function. Datasets can have multiple tasks to an unlimited degree in its list of extra tasks, which always combines them with a list of targets of the same amount of input instances.

Data Loading

- Combining datasets: The ConcatTaskDataset can hold and present multiple datasets as a single bigger dataset, while helper functions allow multiple data instances to be batched. Furthermore, a data loader is provided which can randomly load batches from alternating tasks, which would forego the need to have a unified matrix size across tasks. The loss calculation present in the training and evaluation functions automatically pick apart the prediction outputs and only use the correct data for the related task's loss function.
- Not requiring the combined datasets in memory: Index mode implemented which forms a streaming context for the data to be stored and read from disk. The index mode can be automatically given as a parameter for initializing a TaskDataset, which makes it very easy to

switch to. The more datasets are added, the more likely this scenario is, so a runtime switch between holding the data in memory or reading it from disk can both permit implementations to be executed and cut complicated redevelopment work.

- Train and test set generation: The HoldTaskDataset is responsible for all train and test set generation functionalities added to a regular TaskDataset. TaskDataset functions can be called which will automatically be called accordingly on the Test set as well. The TrainingSetCreator can also be called which operates the training set generation and functions called to refine the TaskDataset in the correct order. Furthermore, there is support for k-fold validation in both these classes as well. Every new fold generated also rewinds normalizations which were based on training set metrics.

- Transforming data: There are two functions present in the TaskDataset for transforming the data, one for scaling the input instances and one for further transformations like cutting the matrices into same sized frames. Both of these operations are implemented on an instance basis in the ExtractionMethod object given to the TaskDataset. The functions on the TaskDataset themselves simply call these to transform every individual input matrix. The TrainingSetCreator can dynamically add new transformations through the add_transformation function, which operates based on names, making it open to extended implementations of the TaskDataset that have more functions. Also available is multiple TaskDatasets sharing the same ExtractionMethods, which would let them share the same metric calculations and thus perform the same transformations as if they were one unified dataset. The TrainingSetCreator can do this automatically at the command of one boolean at input. This is also the reason that the transformation and their calculation functions work instance based in stead of the whole dataset at once.

- Filtering data: Data can be filtered through the sample labels function present on the TaskDataset. The developer can even use this to change the task itself by removing all instances with a specified label and remove its mention in the Task object automatically.

- Reusing data: The TrainingSetCreator is the class which manages the

multi-dataset pipeline. Here, the DataReaders are given as input, while steps in the pipeline (e.g. pre-processing, filtering, transformation,...) can be dynamically added to one specific or all datasets. Only when the dataset creation function is called, do the actual TaskDatasets get created and stored. However, if any step is replaced after creation, the TrainingSetCreator will only reload or recalculate the affected dataset. To minimize unnecessary data usage, which exact present TaskDatasets get created can be determined by the developer, while all other unnecessary TaskDatasets get removed from memory.

- Batching multiple tasks: Multiple tasks can be combined in multiple datasets. As mentioned before tasks and datasets are many-to-many relationships and the tricky issue is to fit a variable amount of tasks, connected target labels and the fact that tasks can be present in multiple TaskDatasets in batches which could include instances from any dataset. Therefore the padding system was developed which also has its limitations concerning targets that are not labels as identified in section 6.2.

- Replicability: Extraction method objects are stored, including their calculations in order to recreate the extraction and the subsequent transformations. Beyond that, the system's randomness based functions like filtering make sure to include optional keys for pseudo random number generation to replicate the same pipelines as previous runs. The training mechanism also stores the model's state at every epoch, through the Results object. Recreating a Results object with the same name, pointing at the same folder location gives automated access for reloading old checkpoints and written metrics. Even more is the fact that the evaluation function further automates this system by including the opportunity to insert a blank model, which in cooperation with the Results object would load and evaluate these old models for all or a subset of the epochs.

- Scalable Manipulation: In the TrainingSetCreator, manipulations can be added for one specified dataset or all at once. These manipulations do not only include the TaskDataset's methods, but also pre-processing parameters and ExtractionMethod objects, which can be replaced at runtime.

Training

- Predicting multiple tasks: Each dataset can have multiple tasks linked to their targets. There are automatic filters for the output to isolate the task specific predictions.

- Task specific output handling: Handling of the task functions, like loss calculation and decision making of the eventual classes from the probabilities are stored in the Task objects

- Loss calculation specifiable: The calculation of the Loss of each task is definable in the Task object. However, currently, losses are always tied to tasks, which have target labels. This is not always the case, as in the research [Tagliasacchi et al., 2020] [Wu et al., 2020], losses have also been calculated based on cost functions from internal model parameters. Since these loss functions are not linked to datasets, but to the models themselves, for which the framework does not offer modules which can be used in the training function for specific handling, it is up to the developer to implement these in the Training_Utils object.

- Loss combination specifiable: Implemented in the Training_Utils object

- Metric calculation, storage and visualization: Gives predictions, true labels and losses to the Results object which calculates the metrics, stores them and writes them to tensorboard where they can easily be compared to other results

- Interrupted Learning: (DEMONSTRATE) Implemented by recreating the Results object and starting the training loop from the given epoch.

- Separate evaluation: (DEMONSTRATE) The evaluation function is separate from the training loop. Training parameters for transformations and such can be reloaded from the stored extraction_method object as well as the model parameters at every epoch in the training function.

- Direct comparison of different runs: Every run has a unique name and TensorBoard has the ability to place the results from different files side-by-side

- Variable training paradigms: In this state, the only training paradigm available is Gradient Descent. Implementing a different paradigm requires foregoing the current training loop implementation.

Chapter 7

Conclusion

The increasing number of audio tasks, datasets, methodologies and reasons to combine these, continuously opens up new ideas researchers can investigate. Promoting new developments in multi-task and multi-dataset research requires as much road blocks to be cleared as possible. This work focuses on clearing the combinatorial issues as well as offer features that can target multiple datasets at once. A literature survey was performed from which an analysis of the concerning fields were made as well as looked for similar frameworks. No frameworks were found to deal with multi-task issues, nor the problematic aspects of multi-dataset problems. From the analysis of the fields, key conclusions were made as to what structural qualities the framework needed to have.

After the requirements were outlined, three in depth implementations were made of different problems that required alternating approaches. Staring from these, generalized solutions were made which would cut development time in the future significantly as well as offering the ability to freely vary different elements and parameters. Comparisons were made by making implementations that do not utilise the framework, which show significant reduction in the data loading and training phases for LOC. The literature was then examined on a per case basis, to investigate what mechanisms could not be implemented through the system or required a degree of work around. However, for supervised, labeled deep learning tasks of different kinds, which were the most popular in the examined papers, the framework already demonstrates the ability to reduce a lot of leg work.

7.1 Future Work

As mentioned in Roberts et al. [1996], frameworks in essence can rarely be considered finished and will likely continue to evolve over time. In this section, some prominent challenges for future expansion are laid out.

7.1.1 More pre-made implementations

In order to be interesting for researchers to use the system, it is important that a lot of the existing extraction methods, transformations and other deep learning features are already available, which would cut a lot of precious development time. The current implementation's features are mostly the ones used in the implemented problems. Future work should round these features out more.

7.1.2 Additional Support for Data Reading and Model creation

The DataReader and model creation were intentionally left open ended. For the DataReader, it was already addressed in section 6.2 that the framework could use a more scalable way of implementing and expanding pre-processing of signals which in its current state is offered through an optional function. Additionally more tools can also be provided to cut the work necessary to implement Data Reading structures which was demonstrated by the slightly higher LOC requirement in section 6.1.

Model creation was also left to remain purely in PyTorch, in order to refrain from imposing unwanted limits on the developer for designing these. However, builder tools could still be provided in order to help create dynamically adapting models for the uncertain number and types of tasks that can be inserted.

7.1.3 Debugging tools and statistics

Building on the last point, additional tools could be provided for debugging parts of the pipeline. As features get scaled to multiple datasets, it can occur

that one dataset's inputs get rendered unusable through execution of transformations. The problem gets bigger due to the higher number of datasets that can be added. Health checks of the pipeline, which would include the models themselves, would also be of help. In case some sequence of pipeline parts causes the input to be unusable further up the pipeline, it is best that this is found before the whole dataset is extracted.

Statistics are already present in evaluation, but could also be of use in the Datasets themselves. This would go hand in hand with debugging as easy evaluation of the matrix sizes, label distributions etc. would prevent mistakes to be made without being noticed. The success of deep learning often depends on the makeup of the dataset, including its balancing in terms of labels.

7.1.4 Expanding Task Types

Another point that was made in 6.2, which was that the framework is mainly optimised for supervised labelled tasks, while other target structures might be more difficult to implement. Not only for targets is there limitation in the design, but the training and evaluation loop are rigid in their implementations and can quickly require the developer to reimplement them all together (e.g. when a mini-batching procedure is needed). These need to be addressed in the future, to allow for more multi-task implementations to be built using the framework.

7.1.5 Optimizing implementation

The system mainly focuses on cutting development time and offering tools that allow iterative research of multi-task systems. The mantra was mainly to enable more than to perfect. What hasn't been looked at in detail is optimizing the developed pipelines in computational resources and execution time. Especially in the TrainingSetCreator, there is a great opportunity to optimize the calculations, after the pipeline is defined. Parallelization is an option when multiple datasets are present. Also in terms of optimal storage structure there are potentially better solutions to be found.

7.1.6 Evolving beyond audio data

The framework is built around audio data, but in theory can easily be expanded to other kinds of data like images. The main thing holding it back currently is how the transformations and extraction methods are all audio based, along with the DataReader's offered tools. Expanding the mediums would however also possibly lead to a wider range of input structures to be necessary which have to be investigated on a case-by-case basis. Still, a lot of the groundwork is already there and the shift would mainly require building more extensions to existing classes and creating divides for tools for audio and other forms of input.

Bibliography

Vinayak Abrol and Pulkit Sharma. Learning hierarchy aware embedding from raw audio for acoustic scene classification. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28:1964–1973, 2020.

Hanêne Ben-Abdallah, Nadia Bouassida, Faiez Gargouri, and Abdelmajid Ben Hamadou. A uml based framework design method. *J. Object Technol.*, 3(8):97–120, 2004.

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.

W Brown Alan and C Wallnau Kurt. A framework for systematic evaluation of software technologies. *IEEE Software, September*, 1996.

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

Dcase. Detection and classification of acoustic scenes and events. URL `http://dcase.community/`.

Dcase-Repo. Dcase-repo/dcase$_u til$: $Acollectionofutilitiesfordetectionandclassificationofacousti$

Soham Deshmukh, Bhiksha Raj, and Rita Singh. Multi-task learning for interpretable weakly labelled sound event detection. *arXiv preprint arXiv:2008.07085*, 2020.

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.

Tharindu Fernando, Sridha Sridharan, Mitchell McLaren, Darshana Priyasad, Simon Denman, and Clinton Fookes. Temporarily-aware context modeling using generative adversarial networks for speech activity detection. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28: 1159–1169, 2020.

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.

Yuxin Huang, Liwei Lin, Shuo Ma, Xiangdong Wang, Hong Liu, Yueliang Qian, Min Liu, and Kazushige Ouch. Guided multi-branch learning systems for dcase 2020 task 4. *arXiv preprint arXiv:2007.10638*, 2020a.

Yuxin Huang, Xiangdong Wang, Liwei Lin, Hong Liu, and Yueliang Qian. Multi-branch learning for weakly-labeled sound event detection. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 641–645. IEEE, 2020b.

Keisuke Imoto, Noriyuki Tonami, Yuma Koizumi, Masahiro Yasuda, Ryosuke Yamanishi, and Yoichi Yamashita. Sound event detection by multitask learning of sound events and scenes with soft scene labels. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 621–625. IEEE, 2020.

Jee-weon Jung, Hye-jin Shim, Ju-ho Kim, Seung-bin Kim, and Ha-Jin Yu. Acoustic scene classification using audio tagging. *arXiv preprint arXiv:2003.09164*, 2020.

Nam Kyun Kim, Jiwon Lee, Hun Kyu Ha, Geon Woo Lee, Jung Hyuk Lee, and Hong Kook Kim. Speech emotion recognition based on multi-task learning

using a convolutional neural network. In *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 704–707. IEEE, 2017.

Tatsuya Komatsu, Keisuke Imoto, and Masahito Togami. Scene-dependent acoustic event detection with scene conditioning and fake-scene-conditioned loss. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 646–650. IEEE, 2020.

Jan Kremer, Lasse Borgholt, and Lars Maaløe. On the inductive bias of word-character-level multi-task learning for speech recognition. *arXiv preprint arXiv:1812.02308*, 2018.

Tyler Lee, Ting Gong, Suchismita Padhy, Andrew Rouditchenko, and Anthony Ndirango. Label-efficient audio classification through multitask learning and self-supervision. *arXiv preprint arXiv:1910.12587*, 2019.

Iván López-Espejo, Zheng-Hua Tan, and Jesper Jensen. Keyword spotting for hearing assistive devices robust to external speakers. *arXiv preprint arXiv:1906.09417*, 2019.

Youyi Lu, Fei Lu, Siddharth Sehgal, Swati Gupta, Jingsheng Du, Chee Hong Tham, Phil Green, and Vincent Wan. Multitask learning in connectionist speech recognition. In *Proceedings of the Australian International Conference on Speech Science and Technology*. Citeseer, 2004.

Josh Meyer. *Multi-task and transfer learning in low-resource speech recognition*. PhD thesis, The University of Arizona, 2019.

Dalibor Mitrović, Matthias Zeppelzauer, and Christian Breiteneder. Features for content-based audio retrieval. In *Advances in computers*, volume 78, pages 71–150. Elsevier, 2010.

Veronica Morfi and Dan Stowell. Deep learning for audio event detection and tagging on low-resource datasets. *Applied Sciences*, 8(8):1397, 2018.

Tin Lay Nwe, Tran Huy Dat, and Bin Ma. Convolutional neural network with multi-task learning scheme for acoustic scene classification. In *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1347–1350. IEEE, 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.

Sankaran Panchapagesan, Ming Sun, Aparna Khare, Spyros Matsoukas, Arindam Mandal, Björn Hoffmeister, and Shiv Vitaladevuni. Multi-task learning and weighted cross-entropy for dnn-based keyword spotting. In *Interspeech*, volume 9, pages 760–764, 2016.

Arjun Pankajakshan, Helen L Bear, and Emmanouil Benetos. Polyphonic sound event and sound activity detection: A multi-task approach. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 323–327. IEEE, 2019.

Chunjong Park, Chulhong Min, Sourav Bhattacharya, and Fahim Kawsar. Augmenting conversational agents with ambient acoustic contexts. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–9, 2020.

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.

Huy Phan, Martin Krawczyk-Becker, Timo Gerkmann, and Alfred Mertins. Dnn and cnn with weighted and multi-task loss functions for audio event detection. *arXiv preprint arXiv:1708.03211*, 2017.

Huy Phan, Oliver Y Chén, Philipp Koch, Lam Pham, Ian McLoughlin, Alfred Mertins, and Maarten De Vos. Unifying isolated and overlapping audio event detection with multi-label multi-task convolutional recurrent neural networks. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 51–55. IEEE, 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.

Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

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.

Michael L Seltzer and Jasha Droppo. Multi-task learning in deep neural networks for improved phoneme recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6965–6969. IEEE, 2013.

Ming Sun, David Snyder, Yixin Gao, Varun K Nagaraja, Mike Rodehorst, Sankaran Panchapagesan, Nikko Strom, Spyros Matsoukas, and Shiv Vitaladevuni. Compressed time delay neural network for small-footprint keyword spotting. In *Interspeech*, pages 3607–3611, 2017.

Marco Tagliasacchi, Félix de Chaumont Quitry, and Dominik Roblek. Multi-task 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.

Haiwei Wu, Yan Jia, Yuanfei Nie, and Ming Li. Domain aware training for far-field small-footprint keyword spotting. *arXiv preprint arXiv:2005.03633*, 2020.

Xianjun Xia, Roberto Togneri, Ferdous Sohel, Yuanjun Zhao, and Defeng Huang. Multi-task learning for acoustic event detection using event and frame position information. *IEEE Transactions on Multimedia*, 22(3):569–578, 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.

Yuni Zeng, Hua Mao, Dezhong Peng, and Zhang Yi. Spectrogram based multi-task audio classification. *Multimedia Tools and Applications*, 78(3):3705–3722, 2019.

Yu Zhang and Qiang Yang. An overview of multi-task learning. *National Science Review*, 5(1):30–43, 2018.