Thesis

Kilian Callebaut

August 1, 2021

Abstract

Abstract

Contents

Chapter 1

Introduction

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

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

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

1.2 Multi-Task Research

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

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

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

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

1.3 Developing Deep Learning Multi-Task Set-ups

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

Issues to face:

Data Reading

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

Data Loading

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

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

Training

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

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

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

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

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

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

Extra issues to be solved

- Figuring out the pipeline for multi-task deep learning set ups can be difficult, considering there are numerous types of and variations in multi-task learning schemes and not a lot of documentation on how to approach these
- Multi-task set-ups are most likely going to be compared to single task set-ups, meaning the code should already take this in account or handle the two cases separately

1.4 Challenges

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

1.5 Contributions

TODO: Outline what new your thesis works contributes.

1.6 Outline

TODO: Summarize the rest of the thesis' structure

Chapter 2

Related Work

2.1 Audio Classification

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

2.2 Multi-task Learning

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

2.3 Multi-task Deep Learning Audio Tasks

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

2.4 Development Frameworks

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

Chapter 3

Problem Statement

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

3.1 Use Cases

TODO: Explain the need for requirements by clarifying examples

3.2 Developers

3.2.1 Researchers

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

3.2.2 Optimizers

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

3.3 Design Principles

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

3.3.1 Fast Prototyping

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

3.3.2 Dynamic Handling Of Deviating Cases

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

3.3.3 Easy Extendibility

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

3.3.4 Abstracted File Management

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

3.3.5 Guided Development

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

3.4 Non-functional Requirements

3.5 Functional Requirements

Chapter 4

System Design

4.1 Assumptions

- For feature extraction, the device can hold each entire extracted dataset separately in memory, but no
-

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

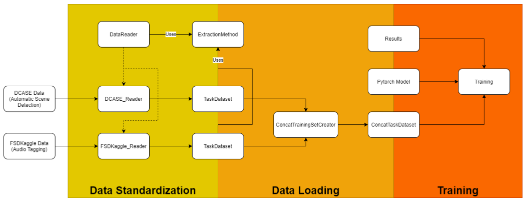

Figure 4.1: Simplified System overview

4.2 Easy changeable variables

4.2.1 Different datasets

4.2.2 Different Sample Rate

4.2.3 Different Feature Extraction

4.2.4 Different Data Transformation

4.2.5 Different Dataloading

4.2.6 Different DL Model

4.2.7 Different Optimizer

4.2.8 Different loss calculation

4.2.9 Different loss combination

4.2.10 Different Stopping Criteria

4.2.11 Different Saving Locations

4.3 Easy expansions

4.3.1 Adding Datasets

4.3.2 Adding Tasks to datasets

4.4 Simplifying abstractions

4.4.1 Saving/Reading Extracted Datasets

4.4.2 Index Mode

4.4.3 Combination of different datasets

TODO: The ConcatTaskDataset function

4.4.4 Train/test generation

4.4.5 Evaluation

4.4.6 Result Saving and Visualizing

4.4.7 Interrupted Learning

4.5 Developmental side rails

4.5.1 Abstract Data Reader

4.5.2 Abstract Extraction Method

4.5.3 Standardized valid input

4.5.4 Centralized Train/test Operations

Chapter 5

Implementation

5.1 Technology

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

5.2 System Architecture

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

5.3 High Level Description

TODO: Simplified overview of the pipeline

5.4 Data Reading

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

5.4.1 Structure

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

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

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

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

5.4.2 DataReader

TODO: insert Data Reader Model

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

5.4.3 ExtractionMethod

TODO: insert ExtractionMethod Model

If the DataReader is the workbench to transform audio datasets to 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 overlap, so no data is lost. Standard methods for fitting, scaling, inverse scaling entire 2D inputs and 2D inputs per row are also already available and are implemented using the sklearn preprocessing toolbox.

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

5.4.4 TaskDataset

The TaskDataset structure is how the framework manages to standardize inputs and targets in one valid object for training. It extends PyTorch's Dataset class to allow for integration with its dataloader objects. This class is responsible for containing the data with functionalities for getting data, storage and transformation. This class is however only a parent class to the Hold-train-test structure, which is set up to deal with functionalities related to generating and handling valid train and test sets. The entire TaskDataset structure is designed to be customizable, but invariant when handling from the outside. There are 3 parts to this that have their own strategies: File

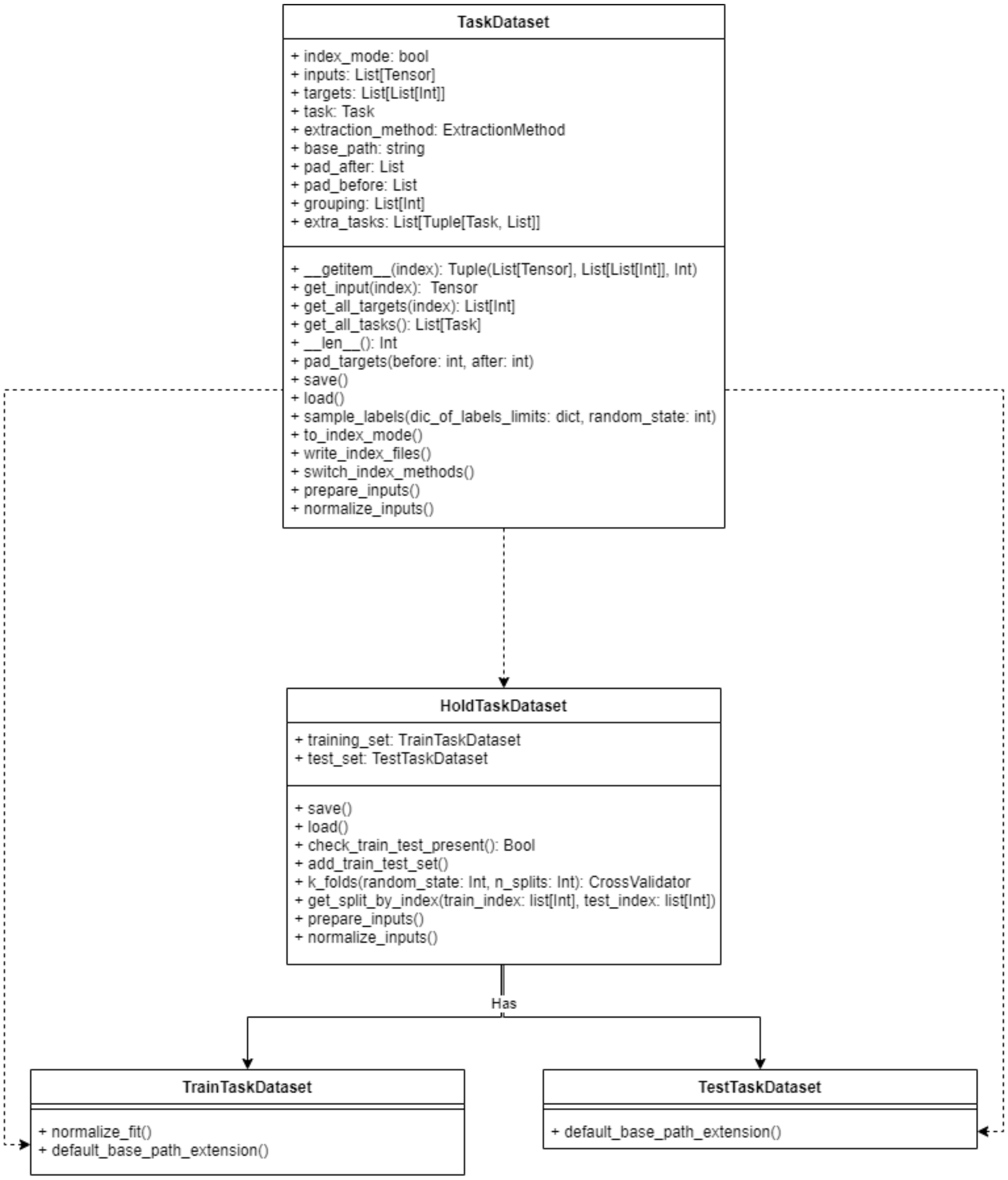
TaskDataset
+ index_mode: bool
+ inputs: List[Tensor]
+ targets: List[List[Int]]
+ task: Task
+ extraction_method: ExtractionMethod
+ base_path: string
+ pad_after: List
+ pad_before: List
+ grouping: List[Int]
+ extra_tasks: List[Tuple[Task, List]]
+ __getitem__(index): Tuple(List[Tensor], List[List[Int]],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

Figure 5.1: TaskDataset Structure

management, structure of data, 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 5.1, 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 data loading is discussed later, getting an item at an index from a TaskDataset infers getting three things: the feature matrix as input, the target list as correct output and the task_group. Getting the feature matrix is a simple indexing operation, after which the scaling transformation is applied. This transformation is applied every time in the get function, as the same data likely has to be rescaled multiple times - e.g. in a five fold cross validation training set-up - so there is no need to revert the transformation every time.

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

5.4.5 Examples To Get List

Example of creating a DataReader

Example of creating a TaskDataset in index mode

Example of creating a HoldTaskDataset with predefined train and test set

5.5 Data Loading

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.

5.5.1 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.

5.5.2 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 5.1. The **HoldTaskDataset** is what comes out the **DataReader** and holds both a **TrainTaskDataset** and a **TestTaskDataset**. Either the Train- and Test-TaskDataset 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.

5.5.3 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.

5.5.4 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.

5.5.5 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**.

5.5.6 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 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.

5.6 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.

5.6.1 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.

5.6.2 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.

5.6.3 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.

5.6.4 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.

5.7 Complementary tools

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

5.8 Extendibility

Things that are meant to be subclassed

DataReader ExtractionMethod Task TrainingUtils

Things that can be subclassed

Results TaskDataset

Things that should be subclassed from outside libraries Model Sampler Any PyTorch object that needs specific functions not available

Things that can vary functionality based on input but should not be changed Training & Evaluation -¿ TrainingUtils -¿ Task TrainingSetCreator -¿ ExtractionMethod

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.

5.8.1 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 5.4.2

Chapter 6

Evaluation

6.1 Goals and Results

6.2 Discussion on the implementation

6.3 Memory Saving (and such)

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

6.4 Requirements

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

Chapter 7

Conclusion

7.1 Future Work

Bibliography

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

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

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

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

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

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

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

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

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