Deep Learning 101 for Audio-based MIR

Geoffroy Peeters, Gabriel Meseguer-Brocal, Alain Riou, Stefan Lattner

version: 1.0.0

This is a web book written for a tutorial session of the 25th International Society for Music Information Retrieval Conference, Nov 10-14, 2024 in San Francisco, CA, USA. The ISMIR conference is the world's leading research forum on processing, searching, organising and accessing music-related data.

Citing this book

```
@book{deeplearning-101-audiomir:book,
        author = {Peeters, Geoffroy and Meseguer-Brocal, Gabriel and Riou, Alain an
        title = {Deep Learning 101 for Audio-based MIR, ISMIR 2024 Tutorial},
        url = {https://geoffroypeeters.github.io/deeplearning-101-audiomir_book},
        address = {San Francisco, USA},
        year = {2024},
        month = November,
        doi = {10.5281/zenodo.14049461},
}
```

Abstract

Context. Audio-based MIR (MIR based on the processing of audio signals) covers a broad range of tasks, including

- music audio analysis (pitch/chord, beats, tagging), retrieval (similarity, cover, fingerprint),
- music audio processing (source separation, music translation)

Skip to main content

A wide range of techniques can be employed for solving each of these tasks, spanning

- from conventional signal processing and machine learning algorithms
- to the whole zoo of deep learning techniques.

Objective. This tutorial aims to review the various elements of this deep learning zoo which are commonly applied in Audio-based MIR tasks. We review typical

- inputs: such as waveform, Log-Mel-Spectrogram, CQT, HCQT, Chroma
- front-ends: such as Dilated-Conv, TCN, SincNet
- projections: such as 1D-Conv, 2D-Conv, U-Net, RNN, LSTM, Transformer
- bottleneck: AE, VAE quantization using VQ-VAE, RVQ
- training paradigms: such as supervised, unsupervised (encoder-decoder), self-supervised, metric-learning, adversarial, denoising/latent diffusion

Method. Rather than providing an exhaustive list of all of these elements, we illustrate their use within a subset of (commonly studied) Audio-based MIR tasks such as

- analysis: multi-pitch, cover-detection, auto-tagging,
- processing: source separation
- generation: auto-regressive/LLM, diffusion

This subset of Audio-based MIR tasks is designed to encompass a wide range of deep learning elements.

The objective is to provide a 101 lecture (introductory lecture) on deep learning techniques for Audio-based MIR. It does not aim at being exhaustive in terms of Audio-based MIR tasks neither on deep learning techniques but to provide an overview for newcomers to Audio-Based MIR on how to solve the most common tasks using deep learning. It will provide a portfolio of codes (Colab notebooks and Jupyter book) to help newcomers achieve the various Audio-based MIR Tasks.

This tutorial can be considered as a follow-up of the tutorial "Deep Learning for MIR" by Alexander Schindler, Thomas Lidy and Sebastian Böck, held at ISMIR-2018.

Skip to main content

Introduction

Organisation of the book

The first part of the book, **"Tasks"**, describes a subset of typical audio-based MIR tasks.
To facilitate the reading of the book, we follow a similar structure to describe each of the audio-based MIR tasks we consider. We describe in turn:

- the **goal** of the task
- the performance measures used to **evaluate** the task
- the popular **datasets** used for the task
  (Datasets can be used to train system or evaluate the performances of a system.)
- how we can solve the task using **deep learning**.
  (This part refers to bricks that are described individually in the second part of the book.)

The second part of the book, **"Deep Learning Bricks"**, describes each brick individually.
We have chosen to separate the description of the bricks from the tasks in which they can be used in order to emphasise the fact that the same brick can be used for several tasks.

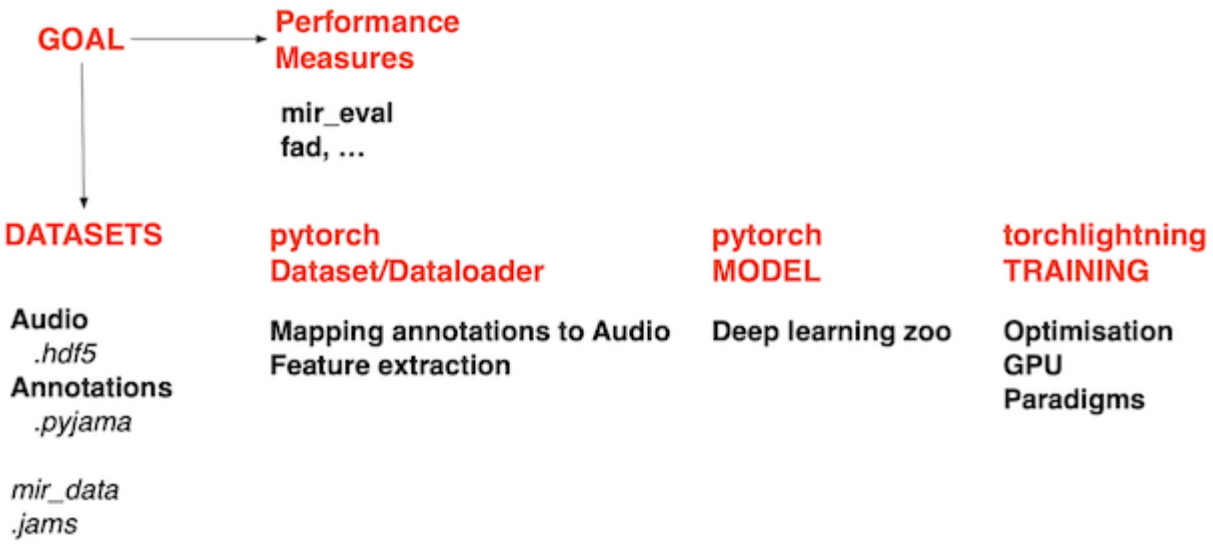


Fig. 1 Overall description of tasks in terms of goal/evaluation/datasets/model

Skip to main content

Simplifying the development

To make our life easier and **facilitate the reading of the code of the notebooks** we will rely on the following elements.

- for **datasets** (audio and annotations): .hdf5 (for audio) and .pyjama (for annotations), described below
- for **deep learning**: pytorch (a python library for deep learning)
  - for the dataset/dataloader
  - for the models, the losses, the optimizers
- for **training**: pytorch-lightning (a library added to pytorch that makes it easier/faster to train and deploy models)

Evaluation metrics

In the notebooks, we will rely most of the time on

- `mir_eval` which provides most MIR specific evaluation metrics,
- `scikit-learn` which provides the standard machine-learning evaluation metrics.

In summary

We summarize the various elements of code to be written below.

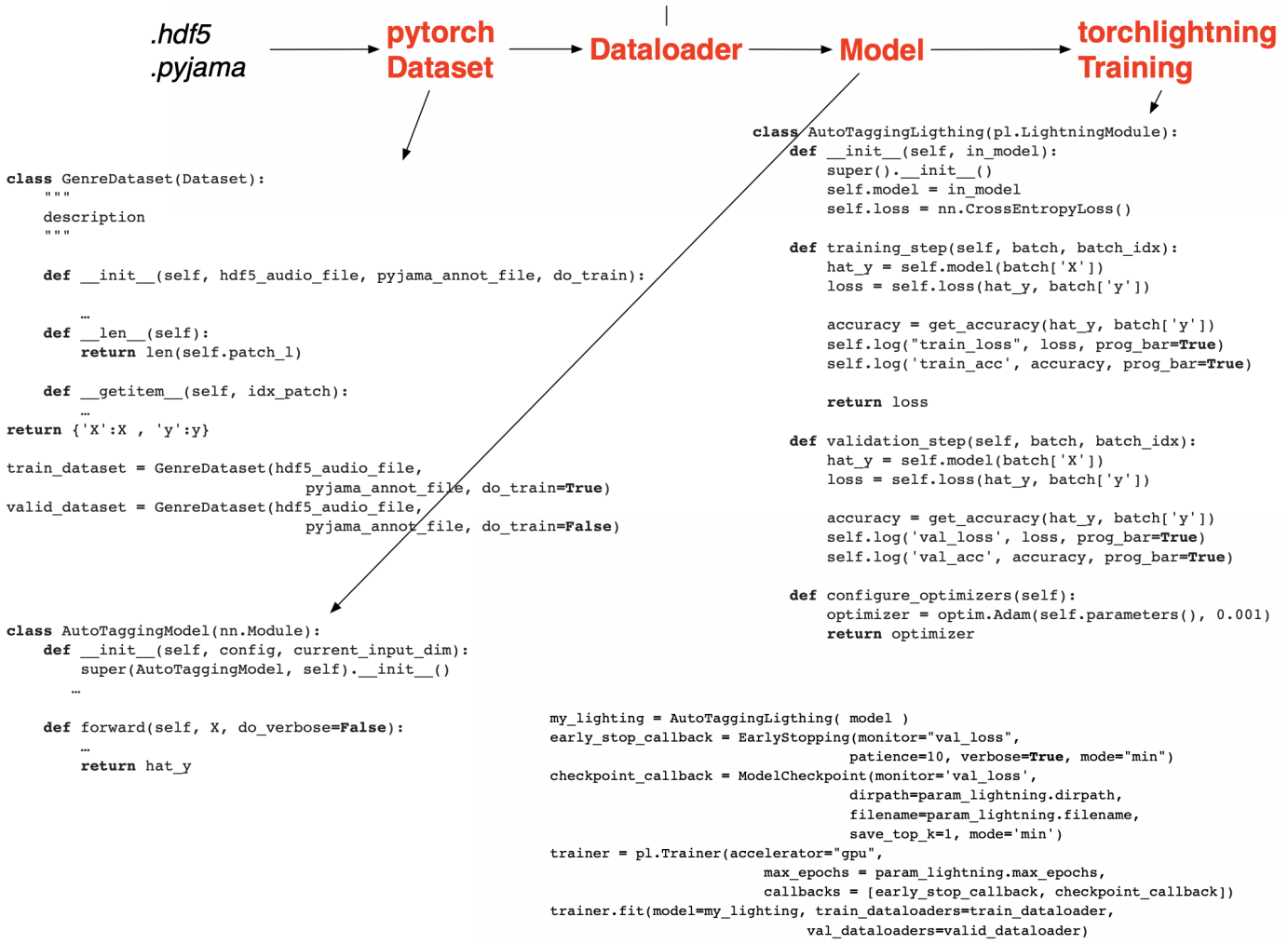


Fig. 2 Code to be written

Datasets .hdf5/.pyjama

In the first part of this tutorial, each dataset will be saved as a pair of files:

- one in .hdf5 format for the audio and
- one in .pyjama format for the annotations.

.hdf5 (Hierarchical Data Format version 5) is a file format and set of tools for managing and storing large amounts of data. It's widely used for handling complex data structures, such as multidimensional arrays, and allows efficient storage and retrieval of large datasets.

In our case, a single .hdf5 file contains all the audio data of a dataset. Each `key` corresponds to an

Skip to main content

- Its array contains the audio waveform.
- Its attribute `sr_hz` provides the sampling rate of the audio waveform.

```
with h5py.File(hdf5_audio_file, 'r') as hdf5_fid:
    audiofile_l = [key for key in hdf5_fid['/'].keys()]
    key = audiofile_l[0]
    pp.pprint(f"audio shape: {hdf5_fid[key][:].shape}")
    pp.pprint(f"audio sample-rate: {hdf5_fid[key].attrs['sr_hz']}")
```

.pyjama is a file format based on JSON which allows storing all the annotations (of potentially different types) of all files of a dataset. It is self-described.

The values of the `filepath` field of the .pyjama file correspond to the `key` values of the .hdf5 file.

```
with open(pyjama_annot_file, encoding = "utf-8") as json_fid:
    data_d = json.load(json_fid)
audiofile_l = [entry['filepath'][0]['value'] for entry in entry_l]
entry_l = data_d['collection']['entry']
pp.pprint(entry_l[0:2])
```

```
{'collection': {'descriptiondefinition': {'album': ...,
                                          'artist': ...,
                                          'filepath': ...,
                                          'original_url': {...,
                                          'tag': ...,
                                          'title': ...,
                                          'pitchmidi': ...},
                'entry': [

                           'artist': [{'value': 'American Bach Soloists'}],
                           'filepath': [{'value': '0+++american_bach_soloists-j_s__
                           'original_url': [{'value': 'http://he3.magnatune.com/all
                           'tag': [{'value': 'classical'}, {'value': 'violin'}],
                           'title': [{'value': 'Gleichwie der Regen und Schnee vom
                          },
                         {
                           'album': [{'value': 'J.S. Bach - Cantatas Volume V'}],
                           'artist': [{'value': 'American Bach Soloists'}],
                           'filepath': [{'value': '0+++american_bach_soloists-j_s__
                           'original_url': [{'value': 'http://he3.magnatune.com/all
                           'tag': [{'value': 'classical'}, {'value': 'violin'}],
                           'title': [{'value': '-Weinen Klagen Sorgen Zagen BWV 12_
                           'pitchmidi': [
                             {
```

Skip to main content

```
                            'duration': 0.26785899999999996
                          },
                          {
                            'value': 71,
                            'time': 0.500004,
                            'duration': 0.26785899999999996
                          }],
                        }
                        ]},
 'schemaversion': 1.31}
```

Using those, a dataset is described by only two files: a .hdf5 for the audio, a .pyjama for the annotations.

We provide a set of datasets (each with its .hdf5 and .pyjama file) for this tutorial here.

```
Index of /gpeeters/tuto_DL101forMIR
[ICO]   Name    Last modified   Size    Description
[PARENTDIR]     Parent Directory                -
[   ] bach10.pyjama                   2024-10-19 12:21  19M
[   ] bach10_audio.hdf5.zip           2024-10-02 07:51  129M
[   ] cover1000.pyjama                2024-10-19 12:21  1.0M
[   ] cover1000_feat.hdf5.zip         2024-10-02 07:52  101M
[   ] datacos-benchmark.pyjama        2024-10-19 12:21  6.3M
[   ] datacos-benchmark_feat.hdf5.zip 2024-10-14 12:31  1.5G
[   ] gtzan-genre.pyjama              2024-10-19 12:21  306K
[   ] gtzan-genre_audio.hdf5.zip      2024-10-02 09:59  1.5G
[   ] maps.pyjama                     2024-10-19 12:21  51M
[   ] maps_audio.hdf5.zip             2024-10-14 12:12  2.3G
[   ] mtt.pyjama                      2024-10-19 12:21  1.7M
[   ] mtt_audio.hdf5.zip              2024-10-14 12:15  2.3G
[   ] rwc-pop_chord.pyjama            2024-10-22 12:23  10M
[   ] rwc-pop_chord_audio.hdf5.zip    2024-10-22 12:25  1.8G
```

Pytorch

dataset/dataloader

To help understanding what a dataloader should provide, we use a top-down approach.
We start from the central part of the training of a deep-learning model.
It consists in a loop over `epochs` and for each an iteration over `batches` of data:

```
for n_epoch in range(epochs):
  for batch in train_dataloader:
    hat_y = my_model(batch['X'])
    loss = my_loss(hat_y, batch['y'])
    loss.backward()
    ...
```

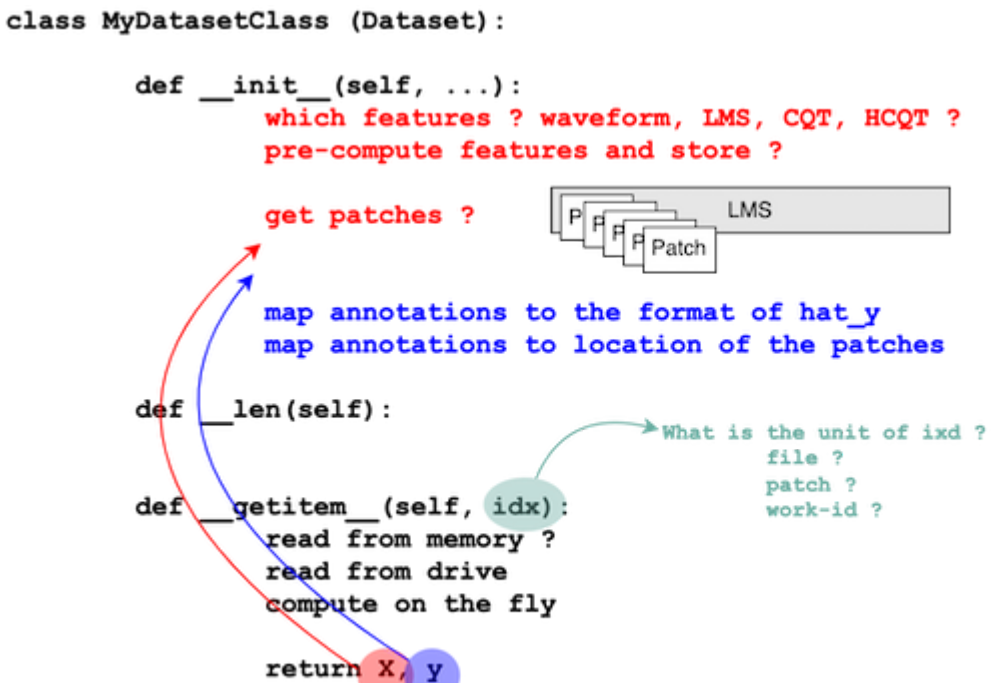
In this, `train_dataloader` is an instance of the pytorch-class `Dataloader` which goal is to encapsulate a set of `batch_size` paris of input `X`/output `y` which are each provided by `train_dataset`.

```
train_dataloader = torch.utils.data.DataLoader(dataset = train_dataset,
                                               batch_size = batch_size,
                                               shuffle = True)
```

`train_dataset` is the one responsible for providing the `X` and the `y`.
It is an instance of a class written by the user (which inherits from the pytorch-class `Dataset`).

Writing this class is probably the most complex part.
It involves

- defining what should the `__getitem__` return (the `X` and `y` for the model) and
- provides in the `__init__` all the necessary information so that `__getitem__` can do its job.



Skip to main content

Fig. 3 Writting a Dataset class

1. It involves defining `X`

- **what is the input representation** of the model ? (`X` can be waveform, Log-Mel-Spectrogram, Harmonic-CQT),
- **where to compute it**
  - compute those one-the-fly in the `__getitem__` ?
  - pre-compute those in the `__init__` and read them on-the-fly from drive/memory in the `__getitem__` ?

In the first notebooks, we define a set of features in the `feature.py` package (`feature.f_get_waveform`, `feature.f_get_lms`, `feature.f_get_hcqt`).

We also define the output of `__getitem__`/`X` as a **patch** (a segment/achunk) of a specific time duration.
The patches are extracted from the features which are represented as a tensor (Channel, Dimension/Frequency, Time).
In the case of `waveform` the tensor is (1,Time), of `LMS` it is (1,128,Time), of `H-CQT` it is (6,92,Time).
To define the patch, we do a **frame analysis** (with a specific window lenght and hope size) over the tensor.
We pre-compute the list of all possible patches for a given audio in the `feature.f_get_patches`.

2. It involves **mapping the annotations** contained in the .pyjama file (such as pitch, genre or work-id annotations)
   - to the format of the output of the pytorch-model, `hat_y`, (scalar, matrix or one-hot-encoding) and
   - to the time position and extent of the patches `X`.
3. It involves **defining what is the unit of** `idx` in the `__getitem__(idx)`.
   For example it can refer to
   - a patch number,
   - a file number (in this case `X` represents all the patches a given file) or
   - a work-id (in this case `X` provides the features of all the files with the same work-id).

```
class TagDataset(Dataset):
    def __init__(self, hdf5_audio_file, pyjama_annot_file, do_train):

        with open(pyjama_annot_file, encoding = "utf-8") as json_fid: data_d = json
        entry_l = data_d['collection']['entry']

        # --- get the dictionary of all labels (before splitting into train/valid)
        self.labelname_dict_l = f_get_labelname_dict(data_d, config.dataset.annot_k

        self.do_train = do_train
        # --- The split can be improved by filtering different artist, albums, ...
        if self.do_train:   entry_l = [entry_l[idx] for idx in range(len(entry_l))
        else:               entry_l = [entry_l[idx] for idx in range(len(entry_l))

        self.audio_file_l =  [entry['filepath'][0]['value'] for entry in entry_l]

        self.data_d, self.patch_l = {}, []

        with h5py.File(hdf5_audio_file, 'r') as audio_fid:
            for idx_entry, entry in enumerate(tqdm(entry_l)):
                audio_file= entry['filepath'][0]['value']

                audio_v, sr_hz =  audio_fid[audio_file][:], audio_fid[audio_file].a
                # --- get features
                if config.feature.type == 'waveform': feat_value_m, time_sec_v = fe
                elif config.feature.type == 'lms':    feat_value_m, time_sec_v = fe
                elif config.feature.type == 'hcqt':   feat_value_m, time_sec_v, fre

                # --- map annotations
                idx_label = f_get_groundtruth_item(entry, config.dataset.annot_key,

                # --- store for later use
                self.data_d[audio_file] = {'X': torch.tensor(feat_value_m).float(),

                # --- create list of patches and associate information
                localpatch_l = feature.f_get_patches(feat_value_m.shape[-1], config
                for localpatch in localpatch_l:
                    self.patch_l.append({'audiofile': audio_file, 'start_frame': lo

    def __len__(self):
        return len(self.patch_l)
```

We then pick up the information corresponding to an `idx`, here the unit is a `patch`.

```
    def __getitem__(self, idx_patch):
        audiofile = self.patch_l[idx_patch]['audiofile']
        s = self.patch_l[idx_patch]['start_frame']
        e = self.patch_l[idx_patch]['end_frame']
```

```
            X = self.data_d[ audiofile ]['X'][:,s:e]
        elif config.feature.type in ['lms', 'hcqt']:
            # --- X is (C, nb_dim, nb_time)
            X = self.data_d[ audiofile ]['X'][:,:,s:e]

        if config.dataset.problem in ['multiclass', 'multilabel']:
            # --- We suppose the same annotation for the whole file
            y = self.data_d[ audiofile ]['y']
        else:
            # --- We take the corresponding segment of the annotation
            y = self.data_d[ audiofile ]['y'][s:e]
        return {'X':X , 'y':y}

train_dataset = TagDataset(hdf5_audio_file, pyjama_annot_file, do_train=True)
valid_dataset = TagDataset(hdf5_audio_file, pyjama_annot_file, do_train=False)
```

models

Models in pytorch are usually written as classes which inherits from the pytorch-class `nn.Module`. Such a class should have

- a `__init__` method defining the parameters (layers) to be trained and
- a `__forward__` method describing how to do the forward with the layers defined in `__init__` (for example how to go from `X` to `hat_y`).

```
class NetModel(nn.Module):

    def __init__(self, config, current_input_dim):
        super().__init__()
          self.layer_l = []
          self.layer_l.append(nn.Sequential(...))
          ...
          self.model = nn.ModuleList(self.layer_l)

    def forward(self, X, do_verbose=False):
        hat_y = self.model(X)
        return hat_y
```

In practice, it is common to specify the hyper-parameters of the model (such as number of layers, feature-maps, activations) in a dedicated `.yaml`.

In the first notebooks, we go one step further and specify the entire model in a `.yaml` file.

- The class `model_factory.NetModel` allows to dynamically create model classes by parsing this file

Below is an example of such a `.yaml` file.

```
model:
    name: UNet
    block_l:
    - sequential_l: # --- encoder
        - layer_l:
            - [BatchNorm2d, {'num_features': -1}]
            - [Conv2d, {'in_channels': -1, 'out_channels': 64, 'kernel_size': [3,3]
            - [Conv2d, {'in_channels': -1, 'out_channels': 64, 'kernel_size': [3,3]
            - [Activation, ReLU]
        - layer_l:
            - [StoreAs, E64]
        - layer_l:
            - [BatchNorm2d, {'num_features': -1}]
            - [Conv2d, {'in_channels': -1, 'out_channels': 128, 'kernel_size': [3,3
            - [Conv2d, {'in_channels': -1, 'out_channels': 128, 'kernel_size': [3,3
            - [MaxPool2d, {'kernel_size': [2,2]}]
            - [Activation, ReLU]
        - layer_l:
            - [StoreAs, E128]
        - layer_l:
            - [BatchNorm2d, {'num_features': -1}]
            - [Conv2d, {'in_channels': -1, 'out_channels': 256, 'kernel_size': [3,3
            - [Conv2d, {'in_channels': -1, 'out_channels': 256, 'kernel_size': [3,3
            - [MaxPool2d, {'kernel_size': [2,2]}]
            - [Activation, ReLU]
        - layer_l:
            - [StoreAs, E256]
        - layer_l:
            - [BatchNorm2d, {'num_features': -1}]
            - [Conv2d, {'in_channels': -1, 'out_channels': 512, 'kernel_size': [3,3
            - [Conv2d, {'in_channels': -1, 'out_channels': 512, 'kernel_size': [3,3
            - [MaxPool2d, {'kernel_size': [2,2]}]
            - [Activation, ReLU]
    - sequential_l: # --- decoder
        - layer_l:
            - [BatchNorm2d, {'num_features': -1}]
            - [ConvTranspose2d, {'in_channels': -1, 'out_channels': 256, 'kernel_si
            - [Conv2d, {'in_channels': -1, 'out_channels': 256, 'kernel_size': [3,3
            - [Conv2d, {'in_channels': -1, 'out_channels': 256, 'kernel_size': [3,3
            - [Activation, ReLU]
        - layer_l:
            - [CatWith, E256]
        - layer_l:
            - [DoubleChannel, empty]
            - [BatchNorm2d, {'num_features': -1}]
```

```
            - [Conv2d, {'in_channels': -1, 'out_channels': 128, 'kernel_size': [3,3
            - [Activation, ReLU]
        - layer_l:
            - [CatWith, E128]
        - layer_l:
            - [DoubleChannel, empty]
            - [BatchNorm2d, {'num_features': -1}]
            - [ConvTranspose2d, {'in_channels': -1, 'out_channels': 64, 'kernel_siz
            - [Conv2d, {'in_channels': -1, 'out_channels': 64, 'kernel_size': [3,3]
            - [Conv2d, {'in_channels': -1, 'out_channels': 64, 'kernel_size': [3,3]
            - [Activation, ReLU]
        - layer_l:
            - [CatWith, E64]
        - layer_l:
            - [DoubleChannel, empty]
            - [BatchNorm2d, {'num_features': -1}]
            - [Conv2d, {'in_channels': -1, 'out_channels': 1, 'kernel_size': [3,3],
            - [Conv2d, {'in_channels': -1, 'out_channels': 1, 'kernel_size': [3,3],
```

TorchLightning training

Pytorch Lightning

- is a high-level wrapper for PyTorch that **simplifies** the process of organizing, training, and scaling models.
- **structures** PyTorch code with best practices, making it easier to implement, debug, and accelerate models across different hardware with minimal boilerplate code.
- allows to **bypass the tedious writing** of training and validation loop over epoch and over mini-batch.

The writing of the Lightning class is very standard and almost the same for all tasks. It involves indicating

- which `model` to use, `loss` to minimize and the `optimizer` to use
- what is a step of forward pass for training (`training_step`) and validation (`validation_step`)

```
class AutoTaggingLigthing(pl.LightningModule):

    def __init__(self, in_model):
        super().__init__()
```

```
    def training_step(self, batch, batch_idx):
        hat_y = self.model(batch['X'])
        loss = self.loss(hat_y, batch['y'])
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def validation_step(self, batch, batch_idx):
        hat_y = self.model(batch['X'])
        loss = self.loss(hat_y, batch['y'])
        self.log('val_loss', loss, prog_bar=True)

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), 0.001)
        return optimizer
```

The training code is then extremely simple: `trainer.fit`.

Pytorch Lightning also allows to define **CallBack** using predefined methods such as

- `EarlyStopping` to avoid over-fitting or
- `ModelCheckpoint` for saving the best model

```
my_lighting = AutoTaggingLigthing( model )

early_stop_callback = EarlyStopping(monitor="val_loss",
                                    patience=10,
                                    verbose=True,
                                    mode="min")
checkpoint_callback = ModelCheckpoint(monitor='val_loss',
                                      dirpath=param_lightning.dirpath,
                                      filename=param_lightning.filename,
                                      save_top_k=1,
                                      mode='min')

trainer = pl.Trainer(accelerator="gpu",
                     max_epochs = param_lightning.max_epochs,
                     callbacks = [early_stop_callback, checkpoint_callback])
trainer.fit(model=my_lighting,
            train_dataloaders=train_dataloader,
            val_dataloaders=valid_dataloader)
```

Notebooks in Colab

We describe here how you can run in Google Colab the notebooks we provide for this tutorial and

Google Colab, short for Collaboratory, is a free, cloud-based platform by Google that allows users to write and execute Python code in a Jupyter Notebook environment. It's especially popular for machine learning and data science tasks because it provides access to powerful hardware like GPUs and TPUs at no cost.

From Colab, choose `File`, `Import`, `Github`, and indicate the following repository `https://github.com/geoffroypeeters/deeplearning-101-audiomir_notebook`.

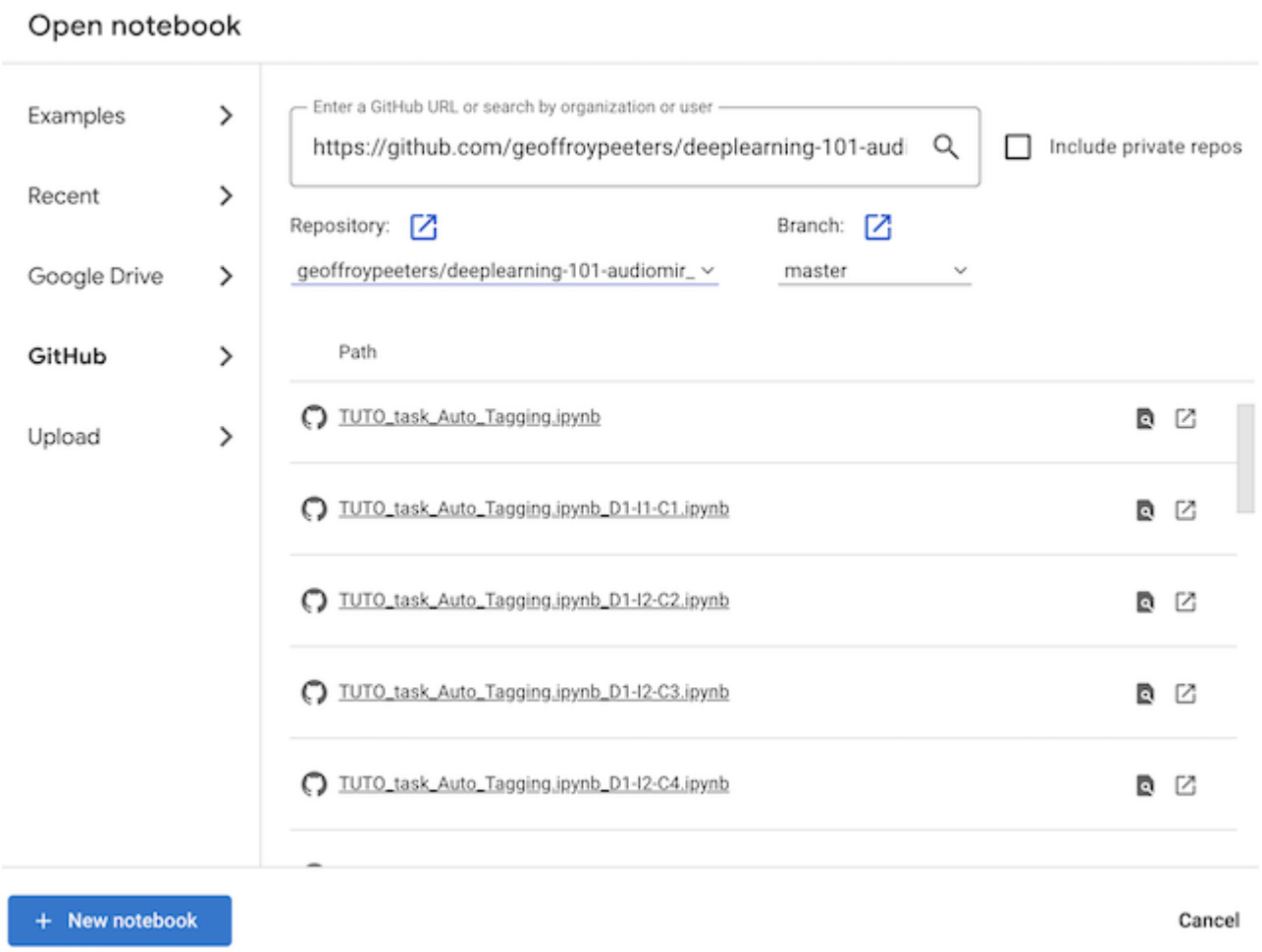


Fig. 4 Importing github files in Google Colab

Within the imported notebook, change `do_deploy` to `True`.
If you then run the notebook, it will automatically

1. **install/import** all necessary packages,
2. **git clone** the code of this tutorial in your local (temporary) Colab space

Skip to main content

3. **download/unzip** the necessary datasets (audio in .hdf5 and annotations in .pyjama) in your local (temporary) Colab space

```
do_deploy = True

if do_deploy:
    !git clone https://github.com/geoffroypeeters/deeplearning-101-audiomir_notebook.git
    %cd deeplearning-101-audiomir_notebook
    !ls

    import urllib.request
    import shutil
    ROOT = 'https://perso.telecom-paristech.fr/gpeeters/tuto_DL101forMIR/'

    hdf5_audio_file, pyjama_annot_file = 'gtzan-genre_audio.hdf5.zip', 'gtzan-genre.pyjama'
    hdf5_audio_file, pyjama_annot_file = 'mtt_audio.hdf5.zip', 'mtt.pyjama'
    hdf5_audio_file, pyjama_annot_file = 'rwc-pop_chord_audio.hdf5.zip', 'rwc-pop_chord_audio.pyjama'

    urllib.request.urlretrieve(ROOT + hdf5_audio_file, hdf5_audio_file)
    if hdf5_audio_file.endswith('.zip'): shutil.unpack_archive(hdf5_audio_file, './')
    urllib.request.urlretrieve(ROOT + pyjama_annot_file, pyjama_annot_file)

    ROOT = './'
```

Fig. 5 Deploying package and dataset within Google Colab

Actions:

We show that

- import a notebook on colab

Music Audio Analysis

Music Audio Analysis refers to the subset of MIR research which targets the extraction of information from the music content using audio analysis.

Such information can relate to

- musical notation (onset, **pitches**, **chords**, key, beats/downbeats, tempo, meter, instrumentation, lyrics)
- the labelling of tracks for searching in catalogues (into **genre**, **tags**, style, moods, activity)

Skip to main content

We study here a subset of these tasks.

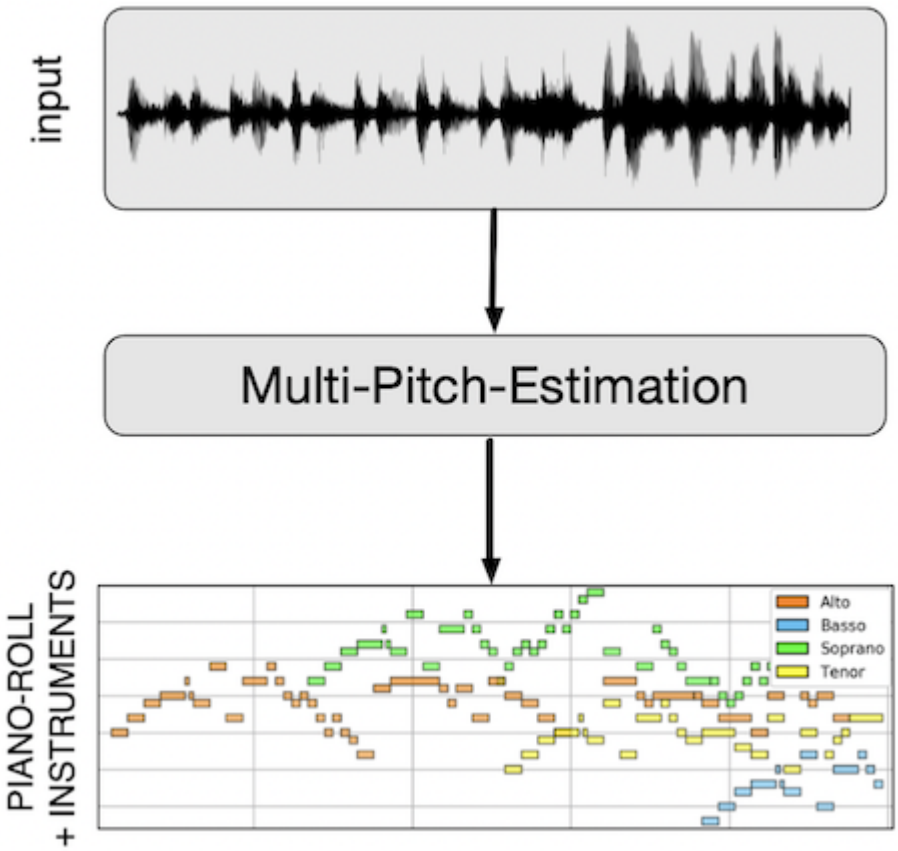
MIREX (Music Information Retrieval Evaluation eXchange) provides an extensive lists of such tasks.

Multi-Pitch-Estimation (MPE)

Goal of MPE ?

Multi-Pitch-Estimation aims at extracting information related to the simultaneously occuring pitches over time within an audio file. The task can either consists in:

1. estimating at each time frame the existing continuous fundamental frequencies (in Hz): $f_0(t)$
2. estimating the [start_time, end_time, pitch] of each musical note (expressed as MIDI note)
3. assigning an instrument-name (source) to the above(see illustration)



Skip to main content

A very short history of MPE

The task has a long history.

- Early approaches focused on single pitch estimation (SPE) using a signal-based method, such as the YIN [DK02] algorithm.
- Next, the difficult case of multiple pitch estimation (MPE) (overlapping harmonics, ambiguous number of simultaneous pitches) was addressed using iterative estimation, as in Klapuri et al [Kla03].
- Subsequently, the main trend has been to use unsupervised methods aiming at reconstructing the signal using a mixture of templates (with non-negative matrix factorisation NMF, probabilistic latent component analysis PLCA or shift-invariant SI-PLCA) [FBR13].

Deep learning era.

- We review here one of the most famous approaches proposed by Bittner et al [BMS+17]
- We show how we can extend it with the same front-end (Harmonic-CQT) using a U-Net [DEP19, WP22].

The task is still very active today, especially using unsupervised learning approaches, more specifically the "equivariance" property, such as in SPICE [GFR+20] or PESTO [RLHP23]

Fore more details, see the very good tutorials "Fundamental Frequency Estimation in Music" and "Programming MIR Baselines from Scratch: Three Case Studies".

How is MPE evaluated ?

To evaluate the performances of an MPE algorithm we rely on the metrics defined in [BED09] and implemented in the mir_eval package. By default, an estimated frequency is considered "correct" if it is within 0.5 semitones of a reference frequency.

Using this, we compute at each time frame t:

- "True Positives" TP(t): the number of f_0's detected that correctly correspond to the ground-truth f_0's

Skip to main content

- "False Negatives" FN(t): represent the number of active sources in the ground-truth that are not reported

From this, one can compute

- Precision= $\frac{\sum_t TP(t)}{\sum_t TP(t)+FP(t)}$
- Recall= $\frac{\sum_t TP}{\sum_t TP(t)+FN(t)}$
- Accuracy= $\frac{\sum_t TP(t)}{\sum_t TP(t)+FP(t)+FN(t)}$

We can also compute the same metrics but considering only the chroma associated to the estimated pitch (independently of the octave estimated).
This leads to the Chroma Precision, Accuracy, Recall.

Example:

```
freq = lambda midi : 440*2**((midi-69)/12)

ref_time = np.array([0.1, 0.2, 0.3])
ref_freqs = [np.array([freq(70), freq(72)]), np.array([freq(70), freq(72)]), np.arr

est_time = np.array([0.1, 0.2, 0.3])
est_freqs = [np.array([freq(70.4+12)]), np.array([freq(70), freq(72), freq(74)]), n

mir_eval.multipitch.evaluate(ref_time, ref_freqs, est_time, est_freqs)

OrderedDict([('Precision', 0.6666666666666666),
             ('Recall', 0.6666666666666666),
             ('Accuracy', 0.5),
             ('Substitution Error', 0.16666666666666666),
             ('Miss Error', 0.16666666666666666),
             ('False Alarm Error', 0.16666666666666666),
             ('Total Error', 0.5),
             ('Chroma Precision', 0.8333333333333334),
             ('Chroma Recall', 0.8333333333333334),
             ('Chroma Accuracy', 0.7142857142857143),
             ('Chroma Substitution Error', 0.0),
             ('Chroma Miss Error', 0.16666666666666666),
             ('Chroma False Alarm Error', 0.16666666666666666),
             ('Chroma Total Error', 0.3333333333333333)])
```

Some popular datasets for MPE

Skip to main content

MPE datasets can be obtained in several ways:

1. annotating manually the full-tracks,
2. annotating (manually or automatically using SPE) the individual stems of a full-track: such as Bach10 or MedleyDB
3. using a MIDI-fied piano: such as ENST MAPS, MAESTRO
4. using audio to score synchronization: such as MusicNet, SMD or SWD

We have chosen the two following datasets since they represent two different types of annotations:

Bach10

Bach10 [DPZ10] is a small (ten tracks) but multi-track datasets in which each track is annotated in pitch (time, continuous f0-value) over time-frames.


```
"entry": [
            {
                "filepath": [
                    {"value": "01-AchGottundHerr-violin.wav"}
                ],
                "f0": [
                    {"value": [
                            [
                                72.00969707905834,
                                72.00969707905834,
                                72.00763743216136,
                                72.00763743216136,
                                72.00763743216136,
                                72.03300373636725,
                                72.04641885597061,
                                ...
                              ]
                            ]}
                    {"time": [
                     0.023,
                     0.033,
                     0.043,
                     0.053,
                     0.063,
                     0.073,
                     0.083,
                     ...
                     ]}
```


Skip to main content

ENST MAPS

ENST MAPS (MIDI Aligned Piano Sounds) [EBD10] is a large (31 Go) piano dataset. Four categories of sounds are provided: isolated notes, random chords, usual chords, pieces of music. We only use the later for our experiment. It is annotated as a sequence of notes (start,stop,midi-value) over time.

This dataset has been made available online with Valentin Emiya's permission specifically for this tutorial. For any other use, please contact Valentin Emiya to obtain authorization.


```
"entry": [
            {
                "filepath": [
                    {"value": "MAPS_MUS-alb_se3_AkPnBcht.wav"}
                ],
                "pitchmidi": [
                    {"value": 67, "time": 0.500004, "duration": 0.26785899999999996
                    {"value": 71, "time": 0.500004, "duration": 0.26785899999999996
                    {"value": 43, "time": 0.500004, "duration": 1.0524360000000001}
                    ...
                ]
            }
```


How can we solve MPE using deep learning ?

We propose here a solution for the MPE task using supervised learning, i.e. with known output `y`.

- Rather than estimating the continuous f_0 by regression, we consider the classification problem into pitch-classes (f_0 are quantized to their nearest semi-tone or $\frac{1}{5}^{th}$ of semi-tone)
- The output `y` to be predicted is a binary matrix $\mathbf{Y} \in \{0,1\}^{(P,T)}$ indicating the presence of all possible pitch-classes $p \in P$ over time $t \in T$
- The problem is then a supervised multi-label problem
    - $\Rightarrow$ We use a softmax and a set of Binary-Cross-Entropy

For the input `X`, we study various choices

Skip to main content

- the Harmonic-CQT proposed by [BMS+17].

For the model f_θ, we study various designs

- the Conv-2D model proposed by [BMS+17] (see Figure below)
- variations of its **blocks**: Depthwise Separable Convolution, ResNet, ConvNext
- the U-Net model proposed by [DEP19, WP22] (see Figure below)

Conv-2D model for MPE

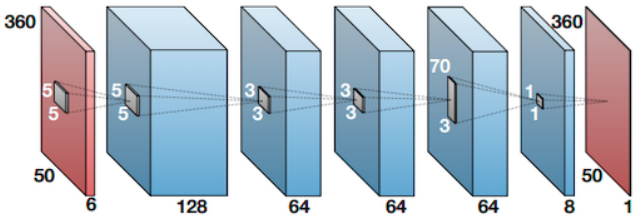


Figure *proposed by [BMS+17]*

U-Net model for MPE

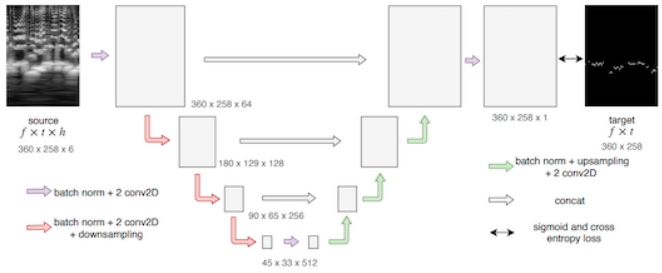
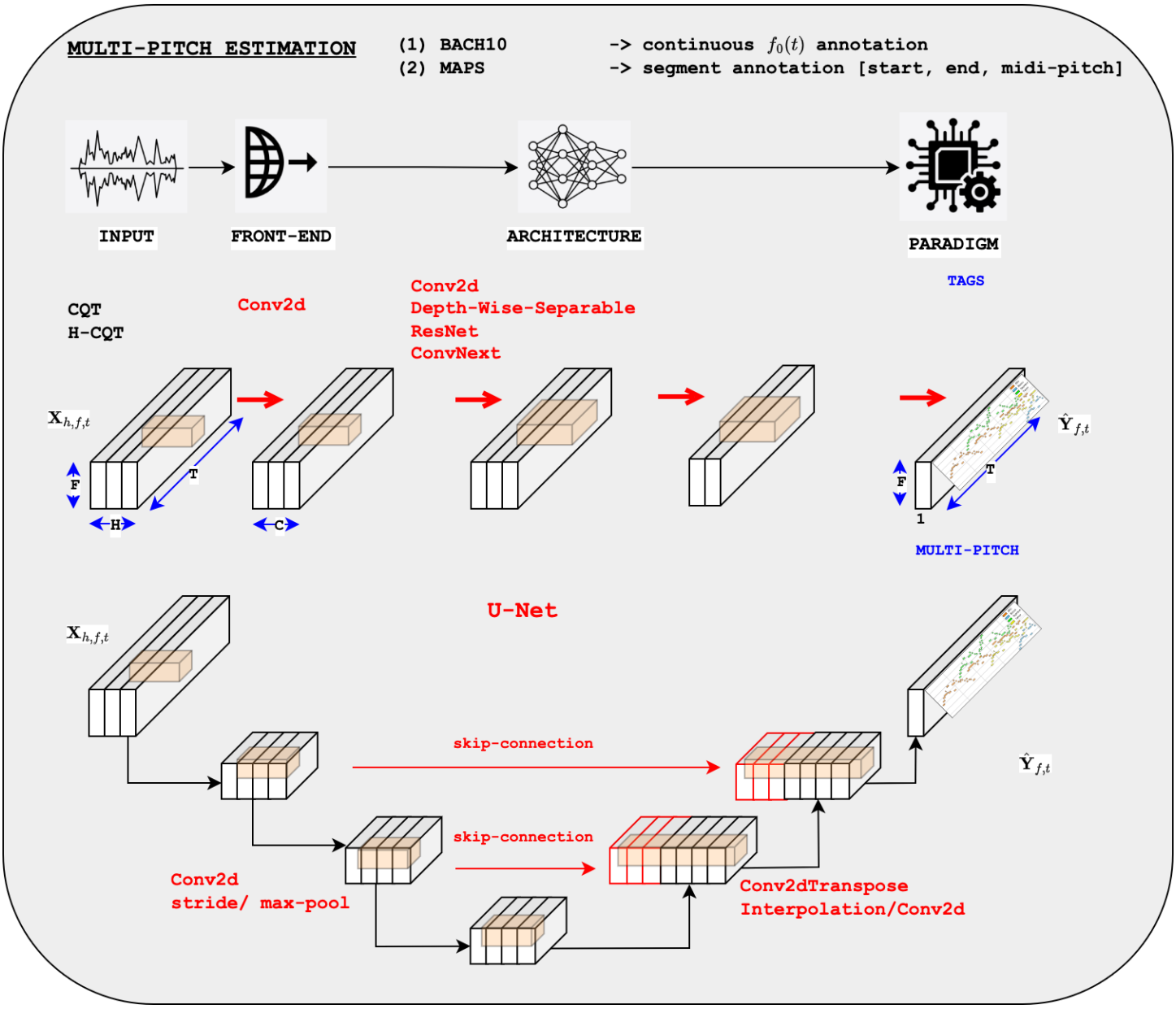


Figure *proposed by [DEP19, WP22]*

We test the results on two datasets:

- a small one (Bach10 with continous f0 annotation)
- a large one (MAPS with segments annotated in MIDI-pitch)

Skip to main content



Experiments

The code is available here:

- (Main notebook)(geoffroypeeters/deeplearning-101-audiomir_notebook)
- (Config Conv2D)[geoffroypeeters/deeplearning-101-audiomir_notebook]
- (Config U-Net)[geoffroypeeters/deeplearning-101-audiomir_notebook]

Skip to main content

Dataset	Input	Frontend	Results	Code
Bach10	CQT(H=1)	Conv2D	P=0.84, R=0.71, Acc=0.63	LINK
Bach10	HCQT(H=6)	Conv2D	P=0.92, R=0.79, Acc=0.74	LINK
Bach10	HCQT(H=6)	Conv2D/DepthSep	P=0.92, R=0.78, Acc=0.74	LINK
Bach10	HCQT(H=6)	Conv2D/ResNet	P=0.93, R=0.80, Acc=0.75	LINK
Bach10	HCQT(H=6)	Conv2D/ConvNext	P=0.92, R=0.80, Acc=0.75	LINK
Bach10	HCQT(H=6)	U-Net	P=0.91, R=0.78, Acc=0.73	LINK
–	–	–	–	–
MAPS	HCQT(H=6)	Conv2D	P=0.86, R=0.75, Acc=0.67	LINK
MAPS	HCQT(H=6)	Conv2D/ResNet	P=0.83, R=0.83, Acc=0.71	LINK
MAPS	HCQT(H=6)	U-Net	P=0.84, R=0.81, Acc=0.70	LINK

Code:

Illustrations of

- show config Conv2D
- show code Bach10/HCQT/Conv2D
  - f_parrallel
  - f_map_annot_frame_based
  - PitchDataset
  - Check the model
  - PitchLigthing, EarlyStopping, ModelCheckpoint, trainer.fit
  - Evaluation: load_from_checkpoint, illustration, mir_eval, plot np.argmin
- show config U-Net
- show code ResNet on MAP

Skip to main content

Cover Song Identification (CSI)

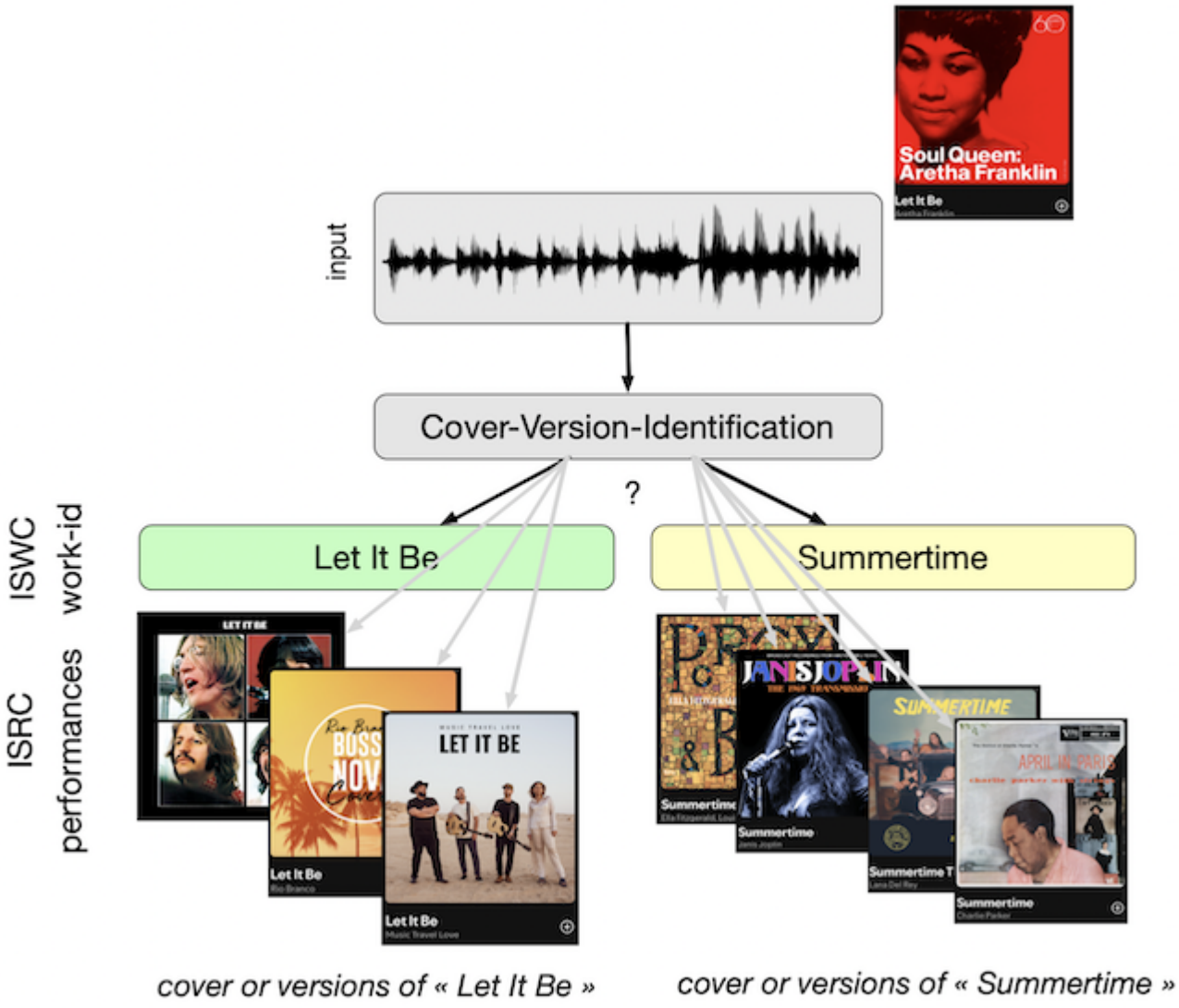
Goal of CSI ?

Cover(Version) Song Identification(Detection) is the task aiming at detecting if a given music track is a cover/version of an existing composition. .
For example detecting that

- this **performance** by Aretha Franklin
- is a **cover/version** of the **work-id** “Let It Be”, composed by the Beatles and also performed in Beatles.

They are covers/versions of the same composition.
We said that they are performances of the same work-id (or ISWC).
The group of songs that are identified as cover versions of each other is often denoted as a **“clique”** .

Skip to main content



Common approach

Considering the very large number of possible work-id (there exist millions of compositions), it is not possible to solve this as a classification (multi-class) problem (too many classes).
To solve this, the approach commonly used is to

- have a a reference dataset R, containing tracks $\{r_i\}$ with known work-id,
- compare the query track q to each track r_i of the reference dataset.

If q is similar to one track of the dataset (i.e. the distance $d(q, r_i)$ is small),
$\Rightarrow$ we decide that q is a cover of r_i and they share the same work-id.

This involves setting a **threshold** τ on $d(q, r_i)$. If $d(q, r_i) < \tau$ we decide they are cover of each

Skip to main content

A very short history of CSI

- The story starts with Ellis et al. [EP07] who proposed to compute $d(q, r_i)$ as the cross-correlation between processed Chroma/PCP features of q and r_i.
- Later on, Serra et al. [SerraGomez08] proposed to improve the features (Harmonic-PCP) and the comparison algorithm (DTW, Dynamic Time Warping). This has lead to the standard approach for years. However, computing the DTW for every pair of tracks is **computationally expensive**.
- To reduce the cost, d should simplified to a simple Euclidean distance between trained features/embedding extracted from q and r_i. Such an approach have been tested in the linear case (using 2D-DFT, PCA, ..) by [HNB13]. However, the results were largely below those of Serra.

Deep learning era.

The solution will come from Computer Vision and the face recognition problem in which deep learning is used to perform metric learning [SKP15].

This method will be transferred to do the cover-song-identification case by [DP19] and [YSerraGomez20].

Fore more details, see the very good tutorial "Version Identification in the 20s".

How is CSI evaluated ?

In practice to evaluate the task, another problem is considered.

The distances between q and all $r_i \in R$ are computed and ranked (from the smallest to the largest) $\Rightarrow A$

- we denote by $w(.)$ the function that gives the work-id of a track,
- we check at which position in the ranked list A we have $w(r_i \in A) == w(q)$.

We can then use the standard ranking/recommendation performance metrics.

Skip to main content

- A the ranked list (of length K) corresponding to a query q
- a_i its i^{th} element,
- $A^k = \{a_i\}_{i \in 1...k}$ the k first ranked items,
- $rel(q, a_i)$ the relevance of items a_i, i.e. whether the item a_i has the same work-id than q: $w(a_i) == w(q)$.

We then compute the usual ranking metrics:

- **MR1: Mean Rank** (lower better): it is the mean (average over queries) of the rank of the first correct result
$$MR1 = \mathbb{E}_{q \in Q} \arg\min_i \{rel(q, a_i) = 1\}$$
- **MRR1: Mean Reciprocal Rank** (higher better): it is the mean (...) of 1/rank of the first correct result
$$MRR1 = \mathbb{E}_{q \in Q} \arg\max_i \frac{1}{rel(q,a_i)=1}$$
- **Precision @ k** (higher better): the number of correct results in the first k elements of the ranked list
$$P(k) = \frac{1}{k} \sum_{i=1}^{k} rel(q, a_i)$$
- **mAP: mean Average Precision** (higher better): same as for multi-label classification
$$AP^q = \frac{1}{K} \sum_{k=1}^{K} P(k)\ rel(q, a_k)$$


```
def F_mean_rank(relevance):
    return relevance.nonzero()[0][0]+1

def F_mean_reciprocal_rank(relevance):
    return 1./ F_mean_rank(relevance)

def F_precision_at_k(relevance, k):
    return np.mean(relevance[:k] != 0)

def F_average_precision(relevance):
    out = [F_precision_at_k(relevance, k + 1) for k in range(relevance.size) if rel
    return np.mean(out)
```

Skip to main content

Other metrics are also commonly used such as the Cumulative Gain, (CG) Discounted Cumulative Gain (DCG), Normalised DCG.

Some popular datasets for CSI

A (close to) exhaustive list of MIR datasets is available in the ismir.net web site.

The first dataset proposed for this task was the cover80 datasets containing 80 different work-id (or cliques) with 2 versions each.

Since then, **much larger datasets** have been created mostly relying on the data provided by the collaborative website SecondHandSongs.

For our implementations, we will consider the two following datasets:

- Cover-1000: 996 performances of 395 different works
- DA-TACOS: 15.000 performances of 3000 different works

Notes that those do not provide access to the audio but to the already extracted **CREMA** features [MB17] (12-dimensional).

How can we solve CSI using deep learning ?

The usual deep learning technique is based on metric learning:

- we train a neural network f_θ such that the projections of q, $f_\theta(q)$ can be directly compared (using Euclidean distance) to the projections of r_i, $f_\theta(r_i)$.
- the distance should relates to their "cover-ness" (how much q and r_i are two performances of the same work-id).
- only the projections (named embedding) of the elements of the reference-set R are stored

Various approaches can be used for metric learning, but the most common is the triplet loss.

For the proposal code we will used the MOVE model [YSerraGomez20] and follow its implementation.

Skip to main content

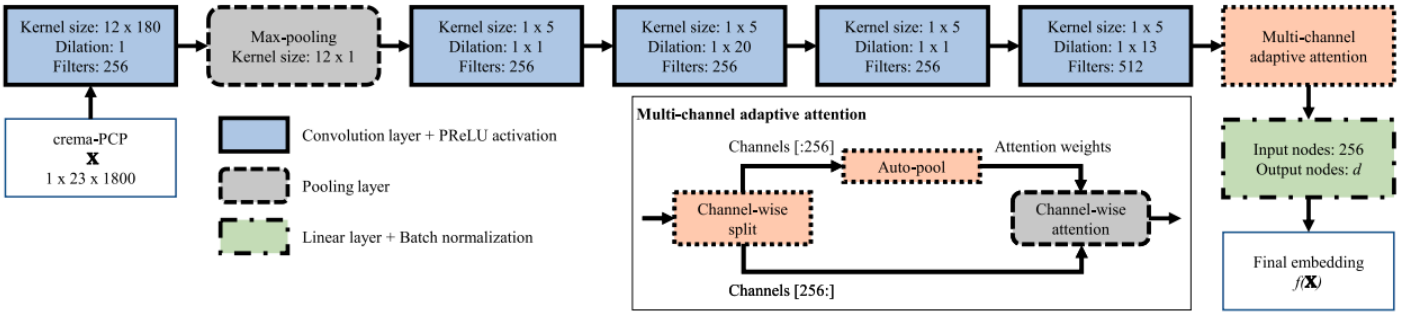
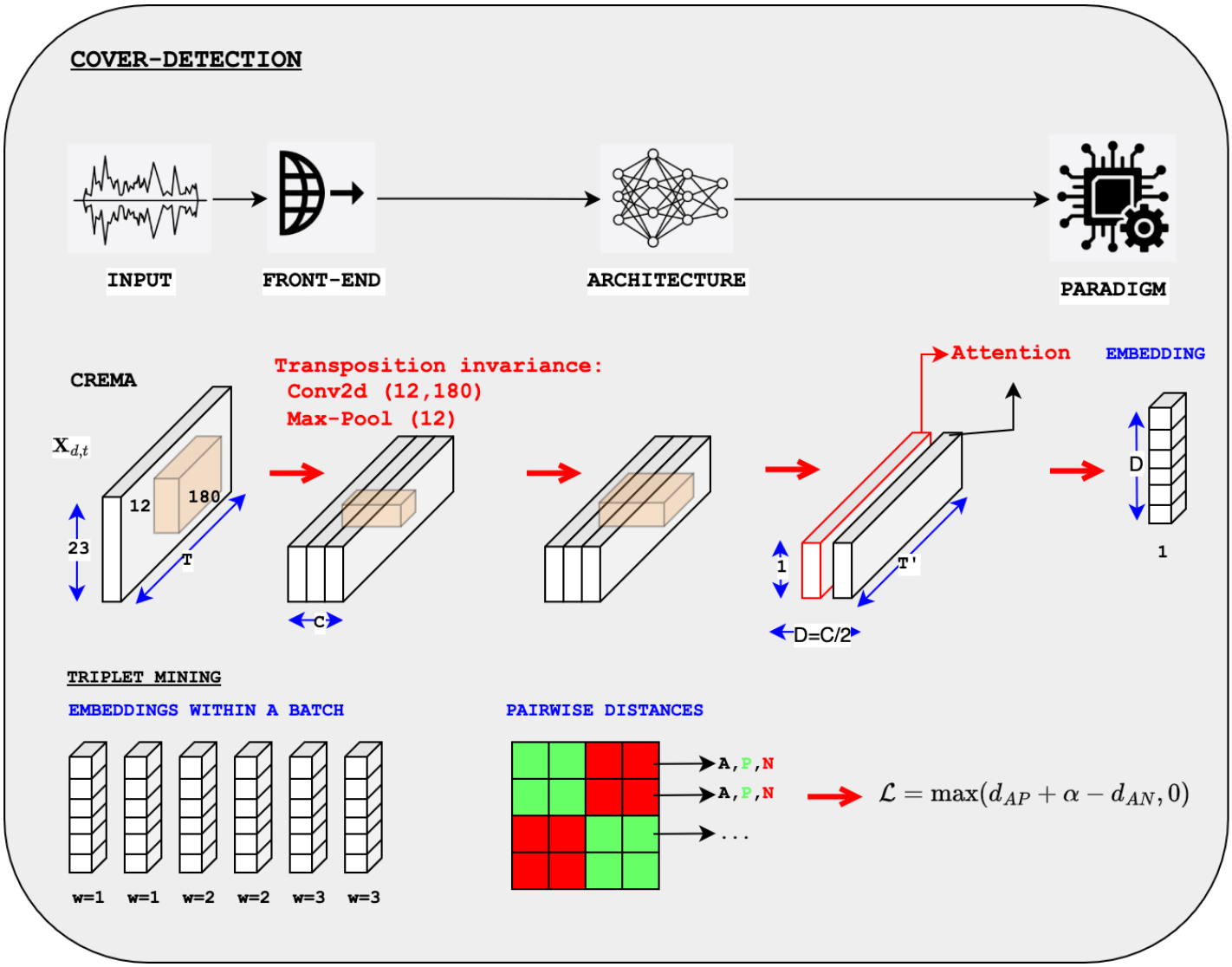


Figure *MOVE model for CSI proposed by [YSerraGomez20]*

We test the results on two datasets:

- a small one (Cover-1000)
- a large one (DA-TACOS-benchmark)



Skip to main content

Experiments

The code is available here:

- (Main notebook)(geoffroypeeters/deeplearning-101-audiomir_notebook)
- (Config Cover)[geoffroypeeters/deeplearning-101-audiomir_notebook]

Dataset	Input	Frontend	Results		Code
Cover1000	CREMA	Move	meanRank=11.2, P@1=0.44, mAP=0.11	meanRecRank=0.551,	LINK
Datacos-benchmark	CREMA	Move	meanRank=465.3, P@1=0.13, mAP=0.073	meanRecRank=0.201,	LINK

Code:

Illustrations of

- show config Model, explain invariance to transposition (maxpool-12)
- show `CoverDataset`, explain `__getitem__` provides a click, `train_dataloader` provides a set of work-id
- explain AutoPoolWeightSplit
- explain Online Triplet Mining, `triplet_loss_mining`
- show performance measures, evaluation of number of OK triplets

Online Triplet mining explained

The online mining of the triplets is actually not a mining of the best data to be fed to the model since all data are actually fed into the model to obtain the embeddings:

$$\mathbf{e}_i = f_\theta(\mathbf{x}_i), i \in \{1, \ldots, \text{batch\_size}\}$$

Skip to main content

- Online mining select the $\{(\mathbf{e}_i\}$ that will form the triplets `{A,P,N}` which are then used to compute the loss (which is to be minimized by SGD).
- Only those selected are used for the loss.

Distance matrix

We first compute a pair-wise distance matrix `dist_all` between all the embeddings $\mathbf{e}_i$.

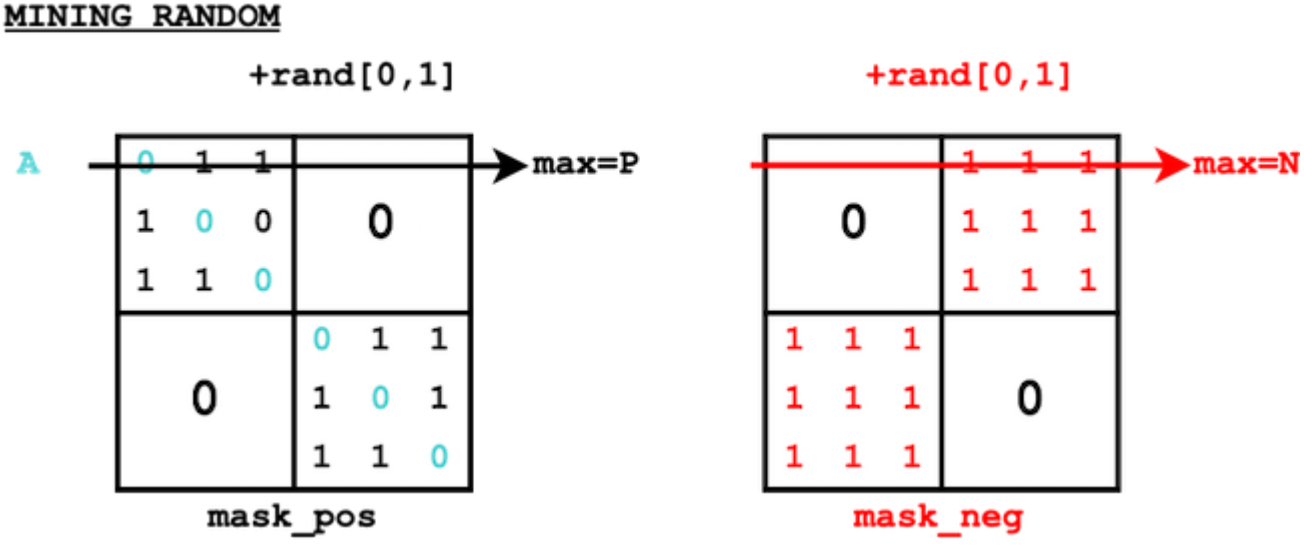
- The "cliques" are grouped together in the matrix, i.e. the performances $\{i-1, i, i+1\}$ belong to the same work-id.

We can then create

- a `mask_pos`: all the distances of similar work-id
- a `mask_neg`: all the distances of different work-id

Random mining

For each anchor A (row), we select randomly a positive (among the mask_pos) and a negative (among the mask_neg).



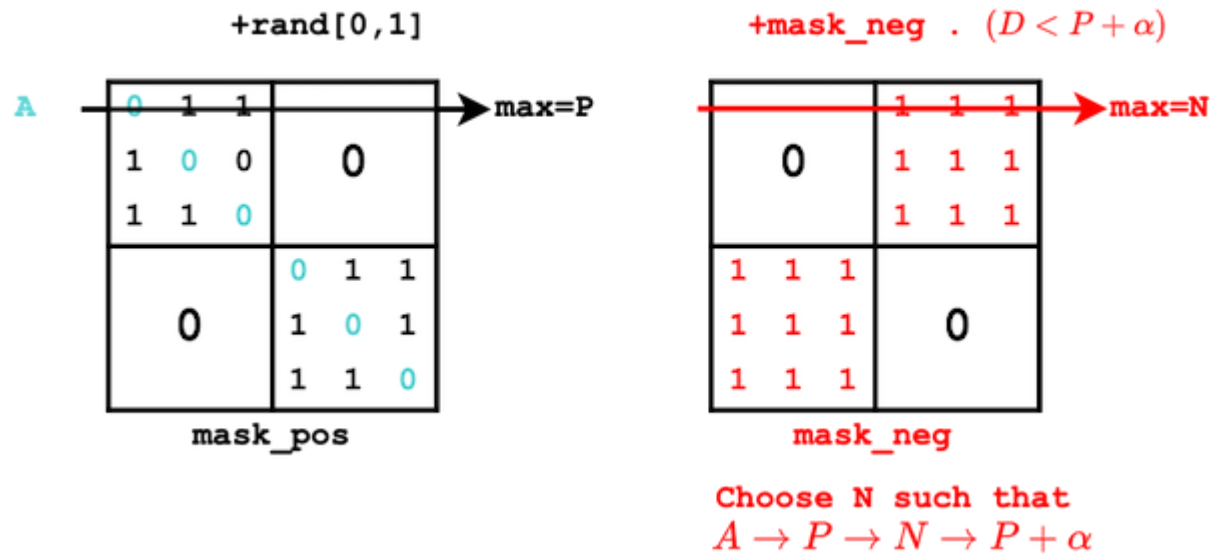

```
def triplet_mining_random(dist_all, mask_pos, mask_neg):
    """
    Performs online random triplet mining
    """
```

Skip to main content

```
    # we consider each row as an anchor and takes the maximum of the masked row (mas
    _, sel_pos = torch.max(mask_pos.float() + torch.rand_like(dist_all), dim=1)
    dists_pos = torch.gather(input=dist_all, dim=1, index=sel_pos.view(-1, 1))

    # selecting the negative elements of triplets
    # we consider each row as an anchor and takes the maximum of the masked row (mas
    _, sel_neg = torch.max(mask_neg.float() + torch.rand_like(dist_all), dim=1)
    dists_neg = torch.gather(input=dist_all, dim=1, index=sel_neg.view(-1, 1))

    return dists_pos, dists_neg
```

Semi-hard mining

For each anchor A (row), we select randomly a positive (among the mask_pos) and a negative (among the mask_neg that statisfy D_neg < D_pos + margin).

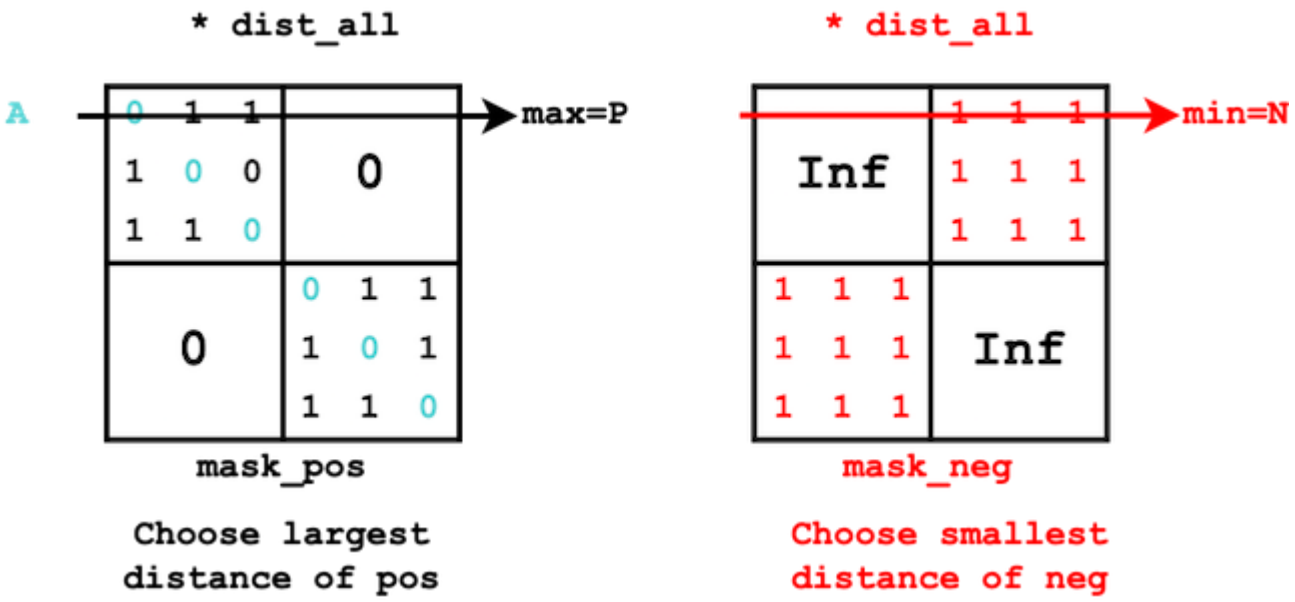



```
def triplet_mining_semihard(dist_all, mask_pos, mask_neg, margin):
    """
    Performs online semi-hard triplet mining (a random positive, a semi-hard negati
    """

    # --- the code below seems wrong
    # --- need criteria
    # 1) should be negative (should be from a different work-id)
    # 2) should be P < N < P+margin

    # selecting the positive elements of triplets
    # we consider each row as an anchor and takes the maximum of the masked row (mas
     , sel_pos = torch.max(mask_pos.float() + torch.rand_like(dist_all), dim=1)
```

```
    # selecting the negative elements of triplets
    _, sel_neg = torch.max(
                            (mask_neg
                            + mask_neg * (dist_all < (dists_pos.expand_as(dist_all)
                            + torch.rand_like(dist_all),
                           dim=1)

    dists_neg = torch.gather(input=dist_all, dim=1, index=sel_neg.view(-1, 1))

    return dists_pos, dists_neg
```

Hard mining

For each anchor A (row), we select (among the mask_pos) the positive with the largest distance and the negative (among the mask_neg) with the smallest distance.




```
def triplet_mining_hard(dist_all, mask_pos, mask_neg, device):
    """
    Performs online hard triplet mining (both positive and negative)
    """

    # --- the code below seems wrong
    # --- need criteria
    # 1) should be negative (from a different work-id)
    # 2) should be N < P

    # selecting the positive elements of triplets
    # --- for each anchor (row) we take the positive with the largest distance
    _, sel_pos = torch.max(dist_all * mask_pos.float(), 1)
```

Skip to main content

```
    # modifying the negative mask for hard mining (because we will use the min)
    # --- if mask_neg==0 then inf
    # --- if mask_neg==1 then 1
    true_value = torch.tensor(float('inf'), device=device)
    false_value = torch.tensor(1., device=device)
    mask_neg = torch.where(mask_neg == 0, true_value, false_value)
    # selecting the negative elements of triplets
    # --- for each anchor (row) we take the negative with the smallest distance
    _, sel_neg = torch.min(dist_all + mask_neg.float(), dim=1)
    dists_neg = torch.gather(input=dist_all, dim=1, index=sel_neg.view(-1, 1))

    return dists_pos, dists_neg
```

Auto-Tagging (front-ends)

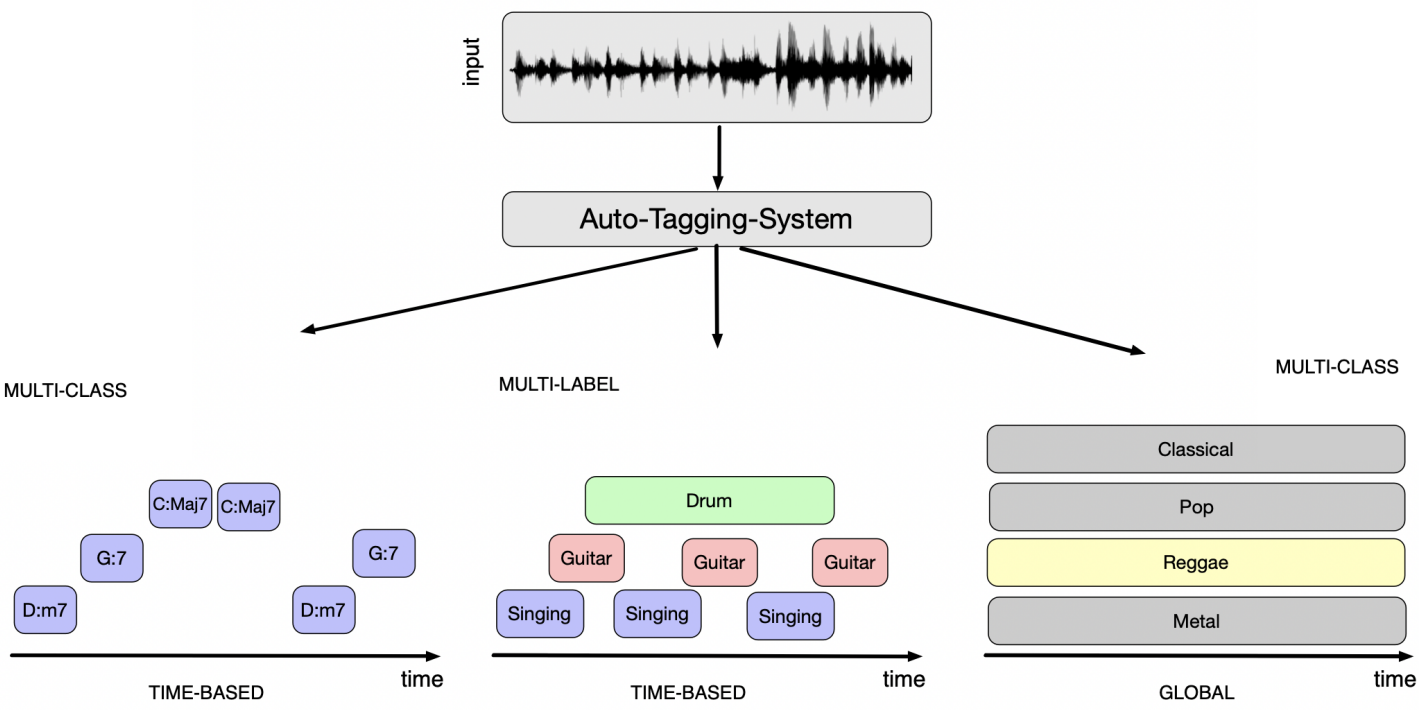
Goal of Auto-Tagging ?

Music auto-tagging is the task of assigning tags (such as genre, style, moods, instrumentation, chords) to a music track.

Tags can be

	Mutually exclusive (multi-class)	**Non-mutually exclusive (multi-label)**
Global in time	Music-genre	User-tags
Time-based	Chord-segments	Instrument-segments

Skip to main content



A very short history of Auto-Tagging

The task has a long history in MIR.

- As soon as 2002 Tzanetakis et al. [TC02] demonstrated that it is possible to estimate the `genre` using a set of low-level (hand-crafted) audio features (such as MFCC) and simple machine-learning models (such as Gaussian-Mixture-Models).
- Over years, the considered audio features improved [Pee04], including block-features [Sey10] or speech-inspired features (Universal-Background-Models and Super-Vector [CTP11]), as well as the machine-learning models (moving to Random forest or Support-Vector-Machine).
- It also quickly appeared that the same feature/ML system could be trained to solve many tasks of tagging or segmentation (genre, mood, speech/music) [Pee07], [BP09].

Deep learning era.

- We start the story with Dieleman [Die14] who proposes to use a Conv2d applied to a Log-Mel-Spectrogram with kernel extending over the whole frequency range, therefore performing only convolution over time.
  The rational for this, is that, as opposed to natural images, sources in a T/F representation are not invariant by translation over frequencies and the adjacent frequencies are not necesseraly correlated (spacing between harmonics)

Skip to main content

- Despite this, Choi et al. [CFS16] proposed (with success) to apply Computer Vision VGG-like architecture to a time-frequency representation.
- Later on, Pons et al. [PLS16] proposed to design kernel shapes using musical consideration (with kernel extending over frequencies to represent timbre, over time to represent rhythm).
- In order to avoid having to choose the kernel shape and STFT parameters, it is been proposed to use directly the audio waveform as input, the "End-to-End" systems of Dieleman et al. [DS14] or Lee et al. [LPKN17].
- The task of auto-tagging has also close relationship with their equivalent task in Speech.

We will develop here a model developed initially for speaker recognition by Ravanelli et al. [RB18].

The task is still very active today. For example

- in the supervised case,
  - MULE [MKO+22] which uses a more sophisticated ConvNet architecture (Short-Fast-Normalizer-Free Net F0) and training paradigm (contrastive learning)
  - PaSST [KMS+21] which uses Vit with tokenized (set of patches) spectrograms fed to a Transformer
- in the Self-Supervised-Learning case
  - with the so-called foundation models such as MERT [LYZ+24] (see second part of this tutorial).

Fore more details, see the very good tutorial "musical classification".

A very short history of Chord Estimation.

Chord estimation can be considered as a specific tagging application: it involves applying mutually exclusive labels (of chords) over segments of time.
However, it has (at least) two specificities:

- chord transition follow musical rules which can be represented by a language model.
- some chord are equivalent, their spelling depends on the choice of the level of detail, and their choice on the

Therefore, ASR (Automatic Speech Recognition) inspired techniques has been developed at first

- an acoustic model representing $p(\text{chord}|\text{chroma})$ and
- a language model, often a Hidden Markov Model, representing $p(\text{chord}_t|\text{chord}_{t-1})$.

Deep learning era. In the case of chord estimation, deep learning is also now commonly used. One seminal paper for this is McFee at al. [MB17]

- the model is a RCNN (a ConvNet followed by a bi-directional RNN, here GRU)
- the model is trained to use an inner representation which relates to the `root`, `bass` and `pitches` (the CREMA)
  - this allows learning representation which brings together close (but different) chords

We will develop here a similar model based on the combination of Conv2d and Bi-LSTM but without the multi-task approach.

How is the task evaluated ?

We consider a set of classes $c \in \{1, \ldots, C\}$.

Multi-class

In a **multi-class** problem, the classes are **mutually exclusive**.

- The outputs of the (neural network) model o_c therefore go to a softmax function.
- The outputs of the softmax, p_c, then represent the probability $P(Y = c|X)$.
- The predicted class is then chosen as $\arg\max_c p_c$.

We evaluate the performances by computing the standard

- Accuracy, Recall, Precision, F-measure for each class c and then take the average over classes c.

```
from sklearn.metrics import classification_report, confusion_matrix
classification_reports = classification_report(labels_idx,
                                               labels_pred_idx,
                                               output_dict=True)
cm = confusion_matrix(labels_idx, labels_pred_idx)
```

Skip to main content

Multi-label

In the **multi-label** problem, the classes are **NOT mutually exclusive**.

- Each o_c therefore goes individually to a sigmoid function (multi-label is processed as a set of parallel independent binary classification problems).
- The outputs of the sigmoids p_c then represent $P(Y_c = 1|X)$.
- We then need to set a threshold τ on each p_c to decide wether class c exist or not.

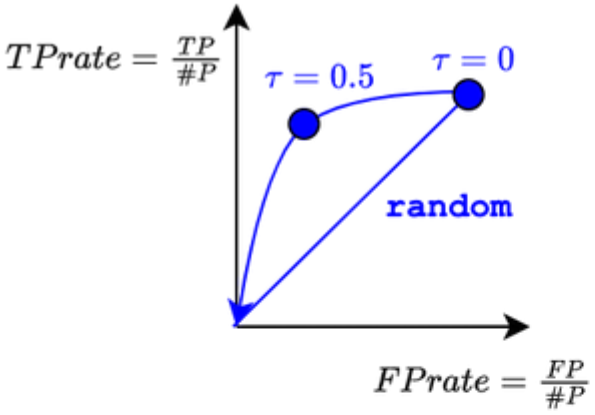
Using a default threshold ($\tau = 0.5$) of course allows to use the afore-mentioned metrics (Accuracy, Recall, Precision, F-measure).
However, in practice, we want to measure the performances independently of the choice of a given threshold.

This can be using either

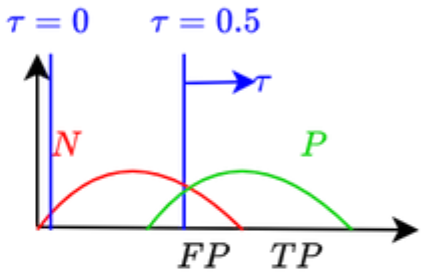
- the AUC (Area Under the Curve) of the ROC. The ROC curve represents the values of TPrate versus FPrate for all possible choices of a threshold τ. The larger the AUC-ROC is (maximum of 1) the more discrimination is between the Positive and Negative classes. A value of 0.5 indicates no discrimination (random system).
- the mean-Average-Precision (mAP). The mAP measures the AUC of the Precision versus Recall curve for all possible choices of a threshold τ).

Note: The AUC-ROC is known to be sensitive to class imbalancing (in case of multi-label, negative examples are usually more numerous than positive ones, hence the FPrate is artificially low leading to good AUC of ROC). In the opposite, mAP which relies on the Precision is less sensitive to class imbalancing and is therefore preferred.

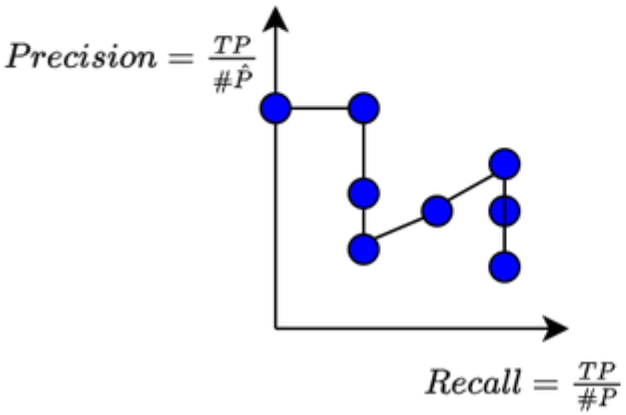
AUC ROC



	$\hat{P}$	$\hat{N}$	
P	TP	FN	$\#P$
N	FP	TN	$\#N$
	$\#\hat{P}$	$\#\hat{N}$	



mean Average Precision



	$\hat{P}$	$\hat{N}$	
P	TP	FN	$\#P$
N	FP	TN	$\#N$
	$\#\hat{P}$	$\#\hat{N}$	

rank ⟶

	T	N	N	T	T	N	N
P=	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{2}{4}$	$\frac{3}{5}$	$\frac{3}{6}$	$\frac{3}{7}$
R=	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$	$\frac{3}{3}$	$\frac{3}{3}$

```
from sklearn.metrics import roc_auc_score, average_precision_score
roc_auc_score(labels_idx, labels_pred_prob, average="macro")
average_precision_score(labels_idx, labels_pred_prob, average="macro")
```

About the averages in `scikit-learn`:

- `macro` **average**: computes the metric independently for each class and then takes the average (i.e., all classes are treated equally, regardless of their frequency).
- `micro` **average**: aggregates the contributions of all classes before calculating the overall metric, essentially treating the problem as a single binary classification task across all samples

Skip to main content

Chord Estimation

In the following (for the sake of simplicity) we will evaluate our chord estimation system as a multi-class problem.

However, chord are not simple labels. Indeed, chord annotation is partly subjective, some chords are equivalent, and the spelling of a chord depends on the choice of the level of detail (the choice of a dictionary).
For this reason, `mir_eval` [RMH+14] or Pauwels et al. [PP13] proposed metrics that allows measuring the correctness of the `root`, the `major/minor` component, the `bass` or the constitution in terms of `chroma`.

Some popular datasets

A (close to) exhaustive list of MIR datasets is available in the ismir.net web site.

We have chosen the following ones since they are often used, they represent the multi-class, multi-label and chord estimation problems, and their audio is easely accessible.

GTZAN

GTZAN contains 1000 audio files of 30s duration, each with a single (**multi-class**) genre label

- among 10 classes: 'blues','classical','country','disco','hiphop','jazz','metal','pop', 'reggae','rock'

Note that GTZAN has been criticized for the quality of its genre label [Stu13]; so results should be considered with cares.

```
"entry": [
            {
                "filepath": [
                    {"value": "blues+++blues.00000.wav"}
                ],
                "genre": [
                    {"value": "blues"}
                ]
```

```
                "filepath": [
                    {"value": "blues+++blues.00001.wav"}
                ],
                "genre": [
                    {"value": "blues"}
                ]
            }
        ]
```


Magna-Tag-A-Tune

Magna-Tag-A-Tune (MTT) is a **multi-label** large-scale dataset of 25,000 30-second music clips from various genres, each annotated with

- multiple tags describing genre, mood, instrumentation, and other musical attributes such as ('guitar', 'classical', 'slow', 'techno', 'strings', 'drums', 'electronic', 'rock', 'fast', 'piano', ...)

We only use a subset of this dataset by only selecting the most 50 used tags and further reducing the number of audio by 20.


```
"entry": [
            {
                "filepath": [
                    {"value": "0+++american_bach_soloists-j_s__bach__cantatas_volume
                ],
                "tag": [
                    {"value": "classical"},
                    {"value": "violin"}
                ],
                "artist": [
                    {"value": "American Bach Soloists"}
                ],
                "album": [
                    {"value": "J.S. Bach - Cantatas Volume V"}
                ],
                "track_number": [
                    {"value": 1}
                ],
                "title": [
                    {"value": "Gleichwie der Regen und Schnee vom Himmel fallt BWV 1
                ],
                "clip_id": [
                    {"value": 29}
                ],
                "original_url": [
```


Skip to main content

```
                "segmentEnd": [
                    {"value": 146}
                ],
                "segmentStart": [
                    {"value": 117}
                ]
            },
        ]
```


RWC-Popular-Chord (AIST-Annotations)

RWC-Popular-Chord (AIST-Annotations)[GHNO02], [Got06] is one of the earliest and remains one of the most comprehensive datasets, featuring annotations for genre, structure, beat, chords, and multiple pitches.

We use the subset of 100 tracks named `Popular-Music-Dataset` and the **chord segments annotations** which we map to a simplified 25 elements dictionary `maj/min/N`.

This dataset has been made available online with Masataka Goto's permission specifically for this tutorial. For any other use, please contact Masataka Goto to obtain authorization.


```
"entry": [
            {
                "filepath": [
                    {"value": "001"}
                ],
                "chord": [
                    {"value": "N:-", "time": 0.0, "duration": 0.104},
                    {"value": "G#:min", "time": 0.104, "duration": 1.754},
                    {"value": "F#:maj", "time": 1.858,"duration": 1.7879999999999998
                    {"value": "E:maj","time": 3.646,"duration": 1.7409999999999997}
                    {"value": "F#:maj", "time": 5.387, "duration": 3.6800000000000008
                ]
            }
        ]
```


How we can solve it using deep learning

Our goal is to show that we can solve the three tasks (multi-class GTZAN, multi-label MTT and chord segment estimation RWC-Pop) with a single code. Depending on the task, we of course adapt the model (defined in the `.yaml` files).

Skip to main content

- GTZAN and RWC-Pop-Chord are **multi-class** problems ⇒ softmax and categorial-CE
- MTT is **multi-label** ⇒ sigmoids and BCEs

global/local:

- GTZAN and MTT have **global** annotations ⇒ we reduce the time axis using AutoPoolWeightSplit
- RWC-Pop-Chord have **local** annotations with a language model ⇒ we use a RNN/bi-LSTM.

For GTZAN and MTT our core model is the SincNet model illustrated below.

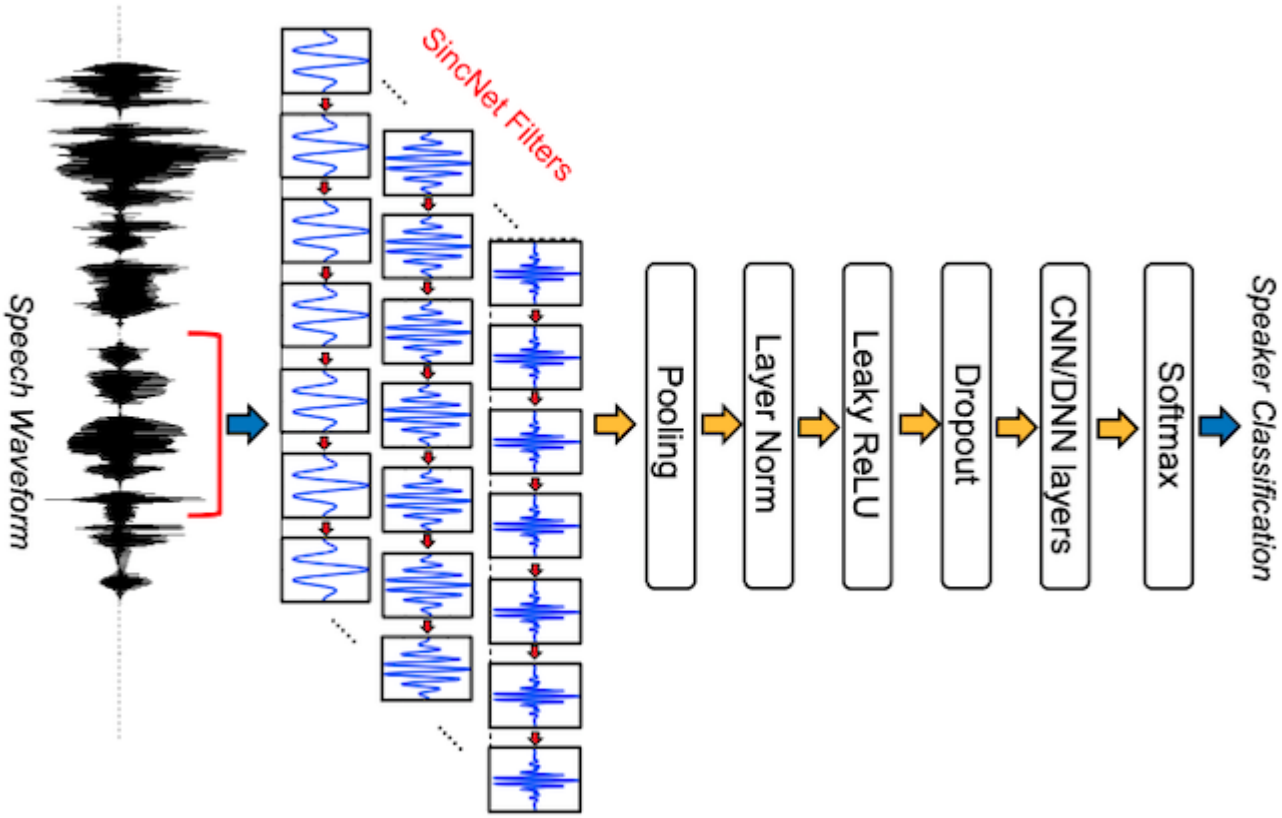
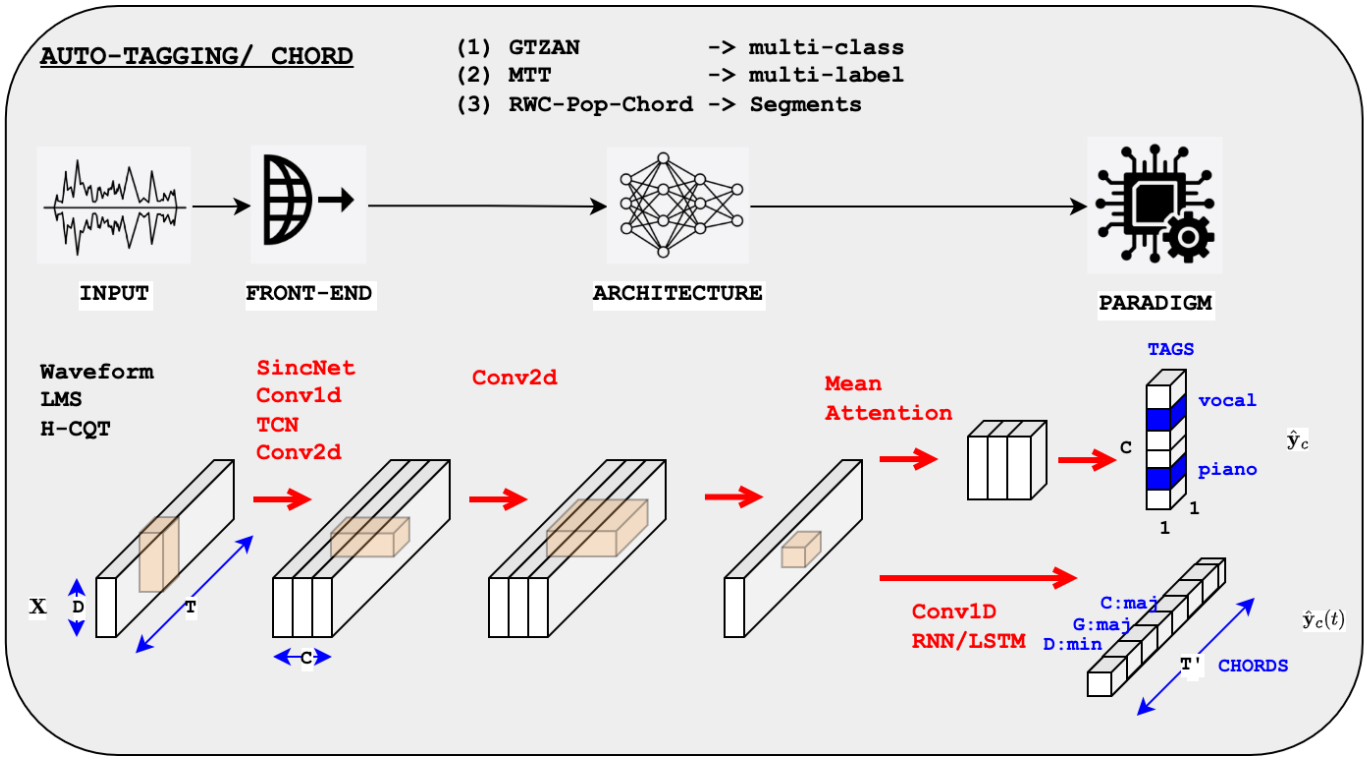


Figure. *SincNet model. image source: SincNet [RB18]*

We will vary in turn

- the **inputs** ⇒ **front-end**:
  - Input: waveform ⇒ Front-end: Conv-1D, TCN, SincNet,
  - Input: Log-Mel-Spectrogram, CQT ⇒ Front-end: Conv-2d
- the model **blocks**:
  - Conv-1d, Linear and AutoPoolWeightSplit for multi-class, multi-label

Skip to main content



Experiments:

The code is available here:

- (Main notebook)(geoffroypeeters/deeplearning-101-audiomir_notebook)
- (Config Auto-Tagging)[geoffroypeeters/deeplearning-101-audiomir_notebook]
- (Config Chord)[geoffroypeeters/deeplearning-101-audiomir_notebook]

Skip to main content

Dataset	Input	Frontend	Model	Results	Code
GTZAN	LMS	Conv2d(128,5)	Conv1d/Linear/AutoPoolWeightSplit	macroRecall: 0.56	LINK
GTZAN	Waveform	Conv1D	Conv1d/Linear/AutoPoolWeightSplit	macroRecall: 0.54	LINK
GTZAN	Waveform	TCN	Conv1d/Linear/AutoPoolWeightSplit	macroRecall: 0.46	LINK
GTZAN	Waveform	SincNet/Abs	Conv1d/Linear/AutoPoolWeightSplit	macroRecall: 0.56	LINK
–	–	–	–	–	
MTT	LMS	Conv2d(128,5)	Conv1d/Linear/AutoPoolWeightSplit	AUC: 0.81, avgPrec: 0.29	LINK
–	–	–	–	–	
RWC-Pop-Chord	CQT	Conv2D(1,5) (5,1)*	Conv1D/LSTM/Linear	macroRecall: 0.54	LINK

Code:

Illustrations of

- autotagging config file
- multi-class: results, CM and Tag-O-Gram
- multi-class:: learned filters SincNet, code SincNet
- multi-class: learned filters Conv1d
- multi-label: results, tag-o-gram:
- chord config file

Skip to main content

- chord: results, tag-o-gram

Auto-Tagging (self-supervised-learning)

Geoffroy Peeters, Gabriel Meseguer-Brocal, Alain Riou, Stefan Lattner

Self-supervised learning (SSL) is a paradigm in machine learning that aims to learn meaningful representations from data without relying on labels. The goal is to leverage the natural structure in data to define tasks that can guide the model's learning process. This approach has grown in popularity for applications in computer vision, natural language processing, and audio processing, where labeled data is often scarce or costly to obtain.

In this tutorial, we'll focus on **Siamese architectures**, which are particularly well-suited to SSL. Siamese networks learn by comparing **positive pairs** of inputs, effectively constructing a task from data itself to learn useful representations.

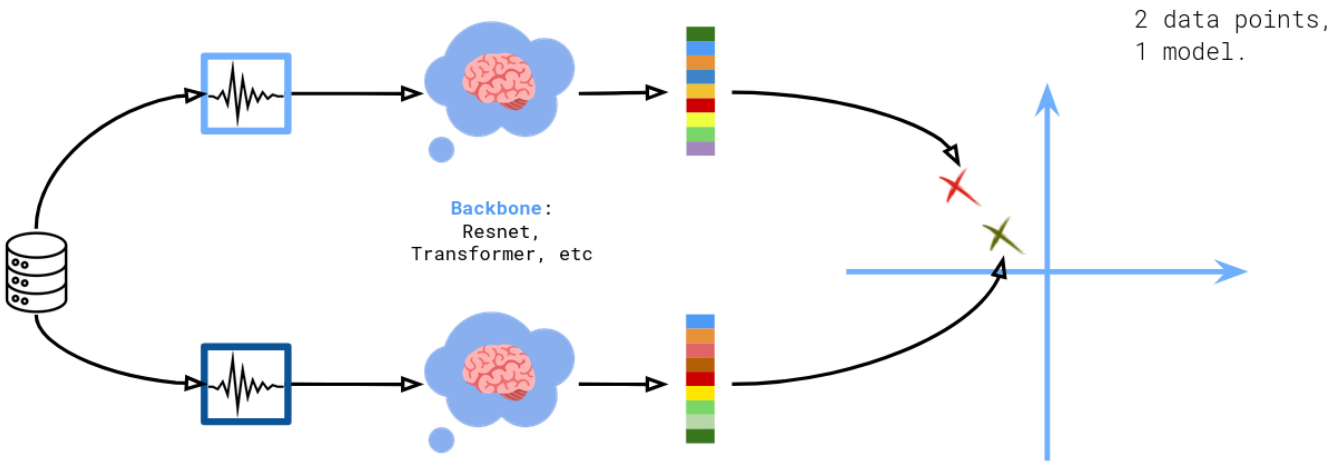
Disclaimer: This book incorporates bricks of Python code for the main components described. The full runnable code for training and evaluating a SSL model is provided in this Jupyter notebook.

Key Concepts in Siamese Architectures for SSL

A Siamese network takes two input samples x_1 and x_2 and projects them into a shared latent space using a neural network f_θ with parameters θ. The underlying idea is simple yet powerful:

1. **Positive pairs**: Each input pair (x_1, x_2) is chosen such that x_1 and x_2 are related in some way (e.g., two different views of the same object or nearby audio chunks in a music track). Since these inputs are "similar," we want their representations in the latent space to be close.
2. **Learning Objective**: To achieve this, we optimize the parameters θ of f_θ by minimizing the distance between $f_\theta(x_1)$ and $f_\theta(x_2)$ in the latent space.

Training the model in this way encourages it to focus on commonalities between similar samples, capturing meaningful and invariant features in the data. The resulting learned representations are useful for downstream tasks, even when labeled data is unavailable.



1. Training a SSL model

In this part, we will implement the different components of a Siamese architecture, namely its structure, the underlying backbone architecture and the criterion to optimize. In addition, we implement the mechanism to create the positive pairs.

1.1. Building a dataset of positive pairs

The first step is to define what are the positive pairs that our model to process, and how to build them. In practice, this mechanism will be directly implemented in a standard PyTorch `Dataset`.

When dealing with music, common strategies to create pairs of similar points without supervision are:

- extracting different chunks from the same song
- applying audio effects to the audio chunk

Recall that the network will learn to find what the elements of the pair have **in common**. For example, if you use chunks of the same song, it will probably capture info such as tonality, genre, tempo, but not chords or timbre. On the contrary, if you transpose your audio to create pairs, it will learn to discard pitch information.

In other words, you control what your model captures by choosing how you compute pairs. And of course, you can combine different techniques!

Let's start by implementing a simple dataset that extracts chunks of audio and randomly apply several transforms. The convenient thing when training a SSL model is that we do not need any label, so we can recursively explore folders and use any audio data we find, which makes the `__init__` function super simple.

For the transforms, we can directly use the transforms implemented in the torchaudio-augmentations repository from Janne Spijkervet (who did a tutorial about SSL in ISMIR 2021, btw). Our `__getitem__` function then just picks an audio, extracts two chunks from it and randomly applies transforms before returning the pair.

```
class PairDataset(torch.utils.data.Dataset):
    """
    A custom dataset class that retrieves pairs of random audio chunks
    from WAV files in a specified directory.

    Args:
        data_dir (str): Path to the directory containing WAV files.
        chunk_duration (float): Duration of the audio chunks in seconds. Defaults t

    Attributes:
        paths (List[str]): List of paths to all WAV files in the dataset.
        chunk_duration (float): Duration of the audio chunks to be extracted.
        transforms (torch.nn.Module): Placeholder for audio augmentation/transforms
    """
    def __init__(self, data_dir: str, chunk_duration: float = 10.):
        self.paths = glob.glob(f"{data_dir}/**/*.mp3", recursive=True)
        self.chunk_duration = chunk_duration

        # Define the set of transforms here
        self.transforms = nn.Sequential(*[
            RandomApply([PolarityInversion()], p=0.8),
            RandomApply([Noise(min_snr=0.001, max_snr=0.005)], p=0.3),
            RandomApply([Gain()], p=0.2),
            HighLowPass(sample_rate=22050), # this augmentation will always be appl
            RandomApply([Delay(sample_rate=22050)], p=0.5),
            RandomApply([Reverb(sample_rate=22050)], p=0.3)
        ])

    def __len__(self) -> int:
        """
        Returns the number of audio files in the dataset.

        Returns:
            int: The total number of WAV files in the dataset.
        """
        return len(self.paths)
```

```
        Retrieves a pair of random audio chunks from a WAV file.

        Args:
            idx (int): Index of the file to retrieve.

        Returns:
            Tuple[torch.Tensor, torch.Tensor]: Two randomly extracted mono audio chu
            as tensors from the selected WAV file.
        """
        # Retrieve file path and audio info
        path = self.paths[idx]
        info = torchaudio.info(path)
        sr = info.sample_rate
        total_frames = info.num_frames

        # Calculate the number of frames per chunk
        num_frames = int(self.chunk_duration * sr)

        # Randomly select starting points for two chunks
        i1, i2 = torch.randint(0, total_frames - num_frames, (2,))

        # Load two audio chunks from the file
        x1, _ = torchaudio.load(path, frame_offset=i1, num_frames=num_frames)
        x2, _ = torchaudio.load(path, frame_offset=i2, num_frames=num_frames)

        # Apply transforms (if any) to both chunks
        x1 = self.transforms(x1)
        x2 = self.transforms(x2)

        # Convert stereo to mono by summing across the channel dimension
        x1 = x1.sum(dim=0)
        x2 = x2.sum(dim=0)

        return x1, x2
```

1.2. Implementing Siamese Networks as a `LightningModule`

Siamese Networks are a simple structure just composed of two... siamese networks, and each of them projects an element of the pair in the latent space. Then a criterion between the two projections is being optimized.

But why only two? Couldn't we use more than only two views? Actually yes, some recent works such as this article from ICLR 2024 by Shidani et al. suggest that Siamese networks can be generalized to more than pairs of two views. However, as 99% of the papers, we will stick to pairs in this tutorial.

Skip to main content

Let us build our Siamese networks as a `LightningModule`. Recall that it is just a training paradigm that does not depend on the underlying architectures, so we can make it quite modular. Our LightningModule just takes two main arguments:

- The architecture of the network itself
- The loss criterion that we want to optimize

That's it!

An interesting trick to improve the generalization abilities of the learned embeddings is not to use the outputs of the last layer of the network but a previous one after training. Doing so enables the mitigation of the misalignment between the training objective and the actual downstream applications. Some French researchers named this trick Guillotine Regularization and studied it in depth in this journal paper from Bordes et al..

In practice, we implement this trick by splitting our network in two successive parts, usually referred to as the **encoder** and the **projector**. In our case, as often, we will use a domain-specific architecture for the encoder and a simple MLP with 2 layers for the projector.

```
class SiameseNetwork(pl.LightningModule):
    """
    A Siamese Network implemented using PyTorch Lightning, designed to work with
    a backbone neural network and a loss function. The network projects two input
    samples into a latent space and optimizes their relationship via the provided l

    Args:
        encoder (torch.nn.Module): The feature extractor model that projects inputs
        loss_fn (torch.nn.Module): The loss function used to optimize the network b
                                   or dissimilarity between the two inputs.
    """
    def __init__(self,
                 encoder: torch.nn.Module,
                 loss_fn: torch.nn.Module,
                 in_channels: int = 512,
                 out_channels: int = 64):
        super(SiameseNetwork, self).__init__()

        self.encoder = encoder
        self.projector = nn.Sequential(
            nn.Linear(in_channels, in_channels),
            nn.ReLU(),
            nn.Linear(in_channels, out_channels)
        )
```

```
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Forward pass through the backbone network.

        Args:
            x (torch.Tensor): Input tensor to be projected into the latent space.

        Returns:
            torch.Tensor: Latent representation of the input.
        """
        y = self.encoder(x)
        return self.projector(y)

    def training_step(self, batch: Any, batch_idx: int) -> torch.Tensor:
        """
        Defines a single training step, including the forward pass and loss computat

        Args:
            batch (Any): A batch of data, expected to contain two input tensors.
            batch_idx (int): Index of the current batch.

        Returns:
            torch.Tensor: The computed loss for this step.
        """
        x1, x2 = batch

        # Project x1 and x2 into the latent space
        z1 = self.forward(x1)
        z2 = self.forward(x2)

        # Compute the loss based on z1 and z2
        loss = self.loss_fn(z1, z2)

        # Log the loss for visualization and monitoring
        self.log("loss", loss, prog_bar=True)

        return loss

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for training.

        Returns:
            torch.optim.Optimizer: Adam optimizer with a learning rate of 1e-4.
        """
        return torch.optim.Adam(self.parameters(), lr=1e-4)
```

1.3. Architecture of the model

Skip to main content

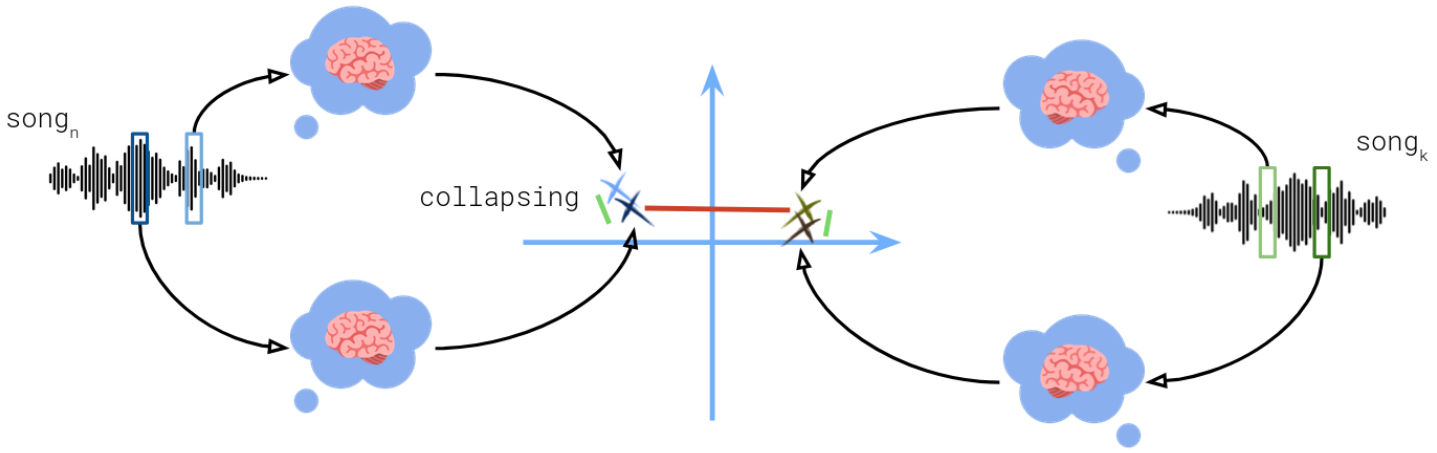
Now let us define those two components. In the notebook, we use SampleCNN for the architecture of the backbone and we define it as a simple PyTorch `nn.Module`. However, the choice of the architecture is independent from the paradigm: any neural architecuture can be used to do SSL, SampleCNN is just an example.

```
class SampleCNN(nn.Module):
    def __init__(self):
        super(SampleCNN, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv1d(1, 128, kernel_size=3, stride=3, padding=0),
            nn.BatchNorm1d(128),
            nn.ReLU())
        ...



    def forward(self, x):
        x = x.view(x.shape[0], 1, -1)
        out = self.conv1(x)
        ...

        # average along time axis to get a single embedding per audio
        return out.mean(dim=-1)
```

1.4. Loss function

Recall that Siamese networks consist in pushing together the representations of inputs that are similar. However, if we only train our model to minimize the Mean Squared Error between positive pairs, we observe an undesirable phenomenon: the network will project everything to the same point, discarding any information from the input.

This phenomenon is called **collapse**, and a lot of research has been about how to prevent this phenomenon to happen. In particular, the most widely technique is to use a contrastive learning. In other words, we want to have **positive pairs** that we push together, but also **negative pairs** that we push far away from each other.



We know how to choose the positive pairs, but how to choose the negative ones? Well, a simple yet effective is to say that everything that is not a positive pair is a negative pair. Practically speaking, since we anyway process batches of inputs, we use the other elements of a batch to create these negative pairs. Given a pair of two batches of size N, we concatenate both into a big matrix $Z = (z_1, \dots, z_{2N}) \in \mathbb{R}^{2N \times d}$. For $1 \leq i \leq 2N$, let i^+ be the index of the corresponding positive pair (i.e. $i^+ = i \pm N$). Overall, the formula looks like this:

$$
\begin{aligned}
\mathcal{L} &= -\frac{1}{2N} \sum_{i=1}^{2N} \log \left(\frac{\exp(\text{sim}(z_i, z_{i^+})/\tau)}{\sum_{j=1, i \neq j}^{2N} \exp(\text{sim}(z_i, z_j)/\tau)} \right) \\
&= -\frac{1}{2N} \sum_{i=1}^{2N} \text{sim}(z_i, z_{i^+})/\tau + \frac{1}{2N} \sum_{i=1}^{2N} \log \left(\sum_{j=1, i \neq j}^{2N} \exp(\text{sim}(z_i, z_j)/\tau) \right) \\
&= \underbrace{-\frac{1}{N} \sum_{i=1}^{N} \text{sim}(z_i, z_{i+N})/\tau}_{\text{positive pairs}} + \underbrace{\frac{1}{2N} \sum_{i=1}^{2N} \log \left(\sum_{j=1, i \neq j}^{2N} \exp(\text{sim}(z_i, z_j)/\tau) \right)}_{\text{negative pairs}}
\end{aligned}
$$

where $\text{sim}(z_i, z_j)$ denotes the cosine similarity between vectors z_i and z_j and τ is a fixed temperature hyperparameter.

```python
class ContrastiveLoss(nn.Module):
    """
    A contrastive loss function designed for self-supervised learning. It computes
    the similarity between two sets of embeddings (z1, z2) and aims to maximize the
    between positive pairs (same inputs) and minimize it between negative pairs (di

    Args:
        temperature (float): A scaling factor applied to the similarity scores. Defa
    """
    def __init__(self, temperature: float = 0.1):
```

Skip to main content

```
    def forward(self, z1: torch.Tensor, z2: torch.Tensor) -> torch.Tensor:
        """
        Computes the contrastive loss between two sets of embeddings.

        Args:
            z1 (torch.Tensor): A batch of embeddings from the first view (e.g., firs
            z2 (torch.Tensor): A batch of embeddings from the second view (e.g., se

        Returns:
            torch.Tensor: The contrastive loss computed from the similarity between
        """
        n = z1.size(0)

        # Concatenate z1 and z2 along the batch dimension and normalize them
        z = torch.nn.functional.normalize(torch.cat((z1, z2)))

        # Compute cosine similarity matrix scaled by temperature
        sim = torch.mm(z, z.t()).div_(self.temperature)

        # Positive loss: average of n-diagonal elements (corresponding to positive p
        pos_loss = -torch.diag(sim, diagonal=n).mean()

        # Negative loss: log of the sum of the exponentiated similarities for negati
        exp_sim = sim.exp_().clone()  # Avoid in-place ops that interfere with autog
        exp_sim.fill_diagonal_(0)     # Set diagonal elements (positive pairs) to 0

        neg_loss = exp_sim.sum(dim=1).log_().mean()

        # Return the combined loss (positive + negative)
        return pos_loss + neg_loss
```

1.5. Train!

We now have all the elements to train a model:

- A `Dataset` object that yields positive pairs of audio chunks
- The general Siamese Networks training pipeline (in a `LightningModule`)
- An appropriate architecture for the encoder (SampleCNN, in a `nn.Module`)
- An objective to minimize that pushes the projections of positive pairs together, but also prevents collapse

Here is a minimal code example to train a SSL model using Lightning's `Trainer` with all the elements we described above:

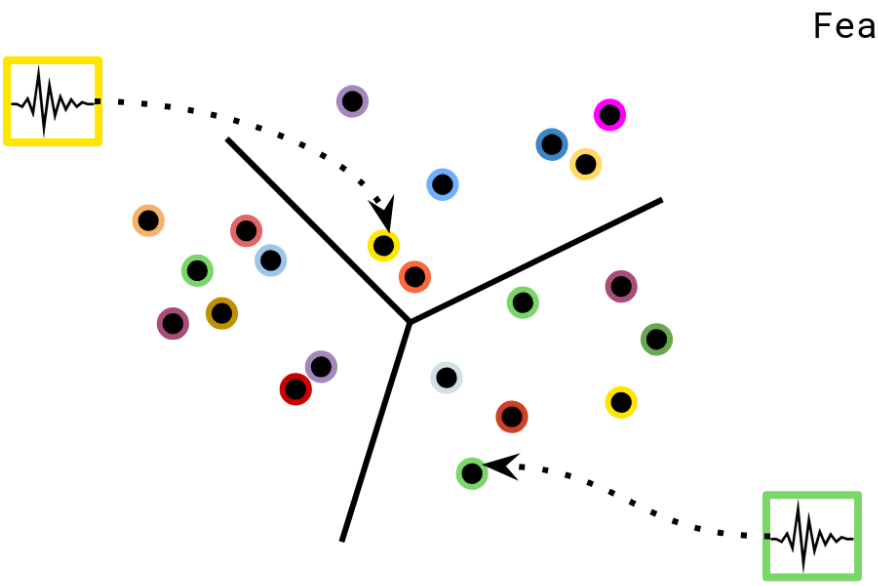
```
# build dataloader
dataloader = torch.utils.data.DataLoader(dataset,
                                         batch_size=96,              # use approxima
                                         num_workers=os.cpu_count(), # use one worke
                                         shuffle=True,               # elements of
                                         pin_memory=True,            # I never unde
                                         persistent_workers=True)    # reuse the ex


# Build the model
sample_cnn = SampleCNN()
model = SiameseNetwork(sample_cnn, loss_fn=ContrastiveLoss(temperature=0.5))

# Define the trainer. To speed-up training and reduce memory footprint, we use 16-b
trainer = pl.Trainer(accelerator='gpu', max_epochs=500)

# Train!
trainer.fit(model, dataloader)
```

Note how modular are the different blocks; the dataset implementation is completely independent from the loss which is independent from the architecture, etc.

In practice you can imagine alternative ways to sample pairs, use a different frontend/architecture, optimize another loss function, etc. or ***any combination of these!*** Actually, all of these design choices are interesting research directions that led to many publications.

2. Evaluation

To evaluate SSL models, we typically combine the encoder and a linear probe. First, the encoder is frozen to prevent any further updates to its parameters. This ensures that we are evaluating the representations learned during the self-supervised phase. Next, a simple linear classifier (linear probe) is trained on top of these frozen features using labeled data. The performance of this linear probe, typically measured through metrics like accuracy, mean Average Precision (mAP), or ROC-AUC, provides an indication of the quality of the learned representations. This evaluation method effectively assesses how well the SSL model has captured useful features from the data.



In this tutorial, we focus on a music tagging task on a subset of MagnaTagATune. It is modeled as a multilabel classification problem.

2.1. Loading the small annotated dataset

In this part, we rely on a small **annotated** dataset.

```
pd.read_csv("mtt_ssl/train/annotations.csv")
```