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.

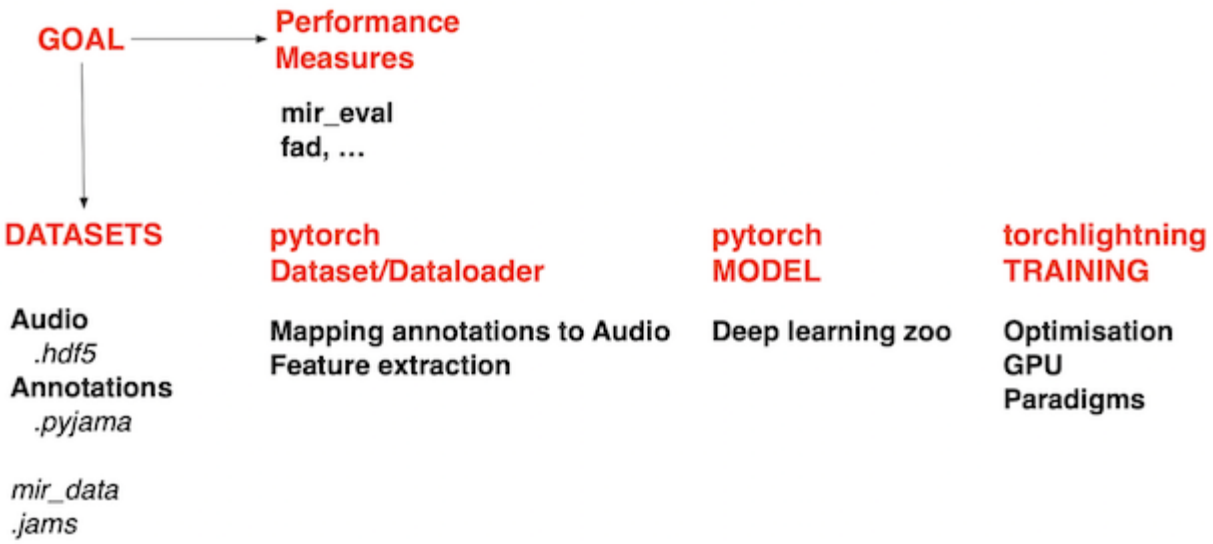


Fig. 1 Overall description of tasks in terms of goal/evaluation/datasets/model

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.

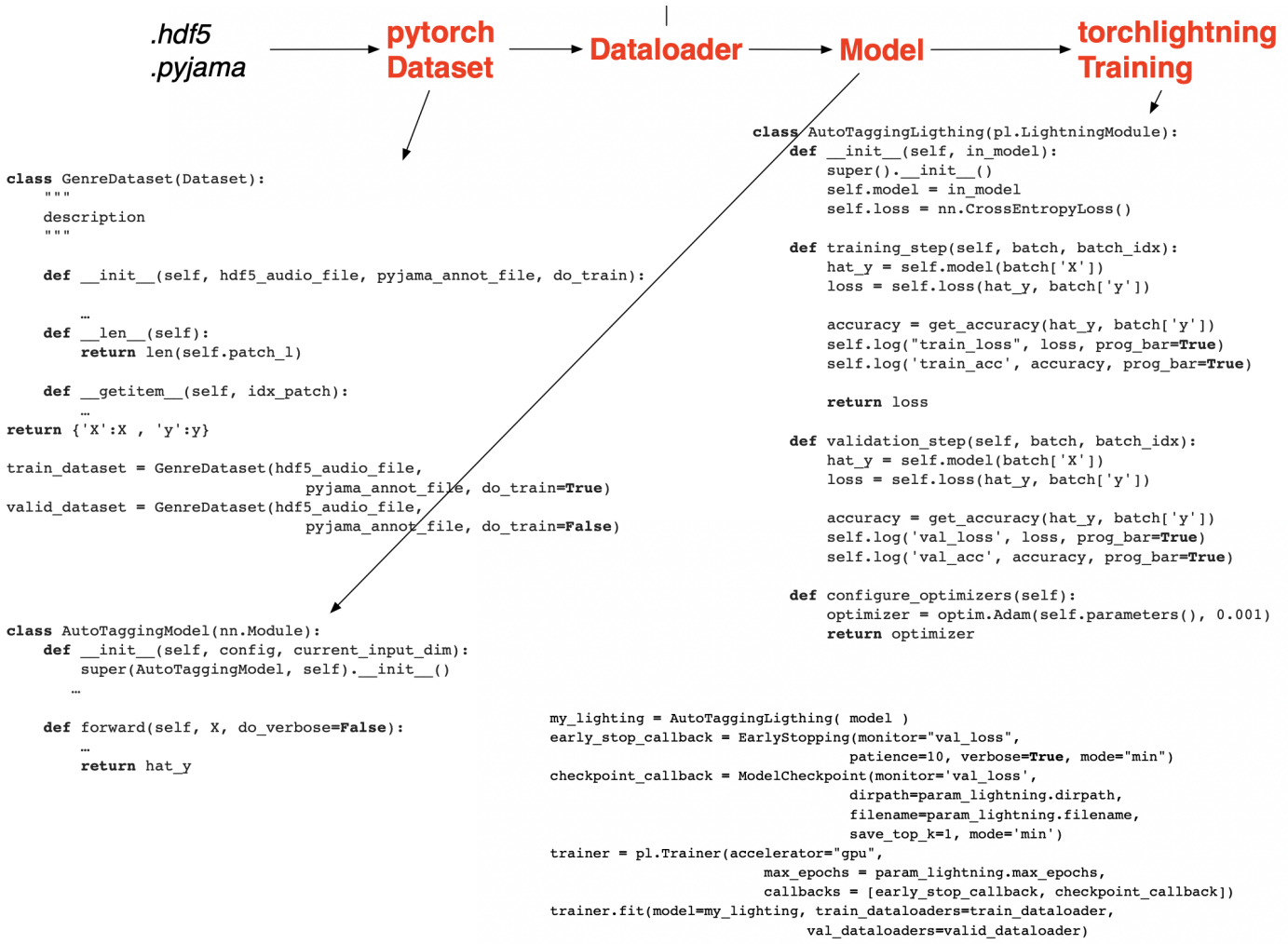


Fig. 2 Code to be written

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()
    ...
```

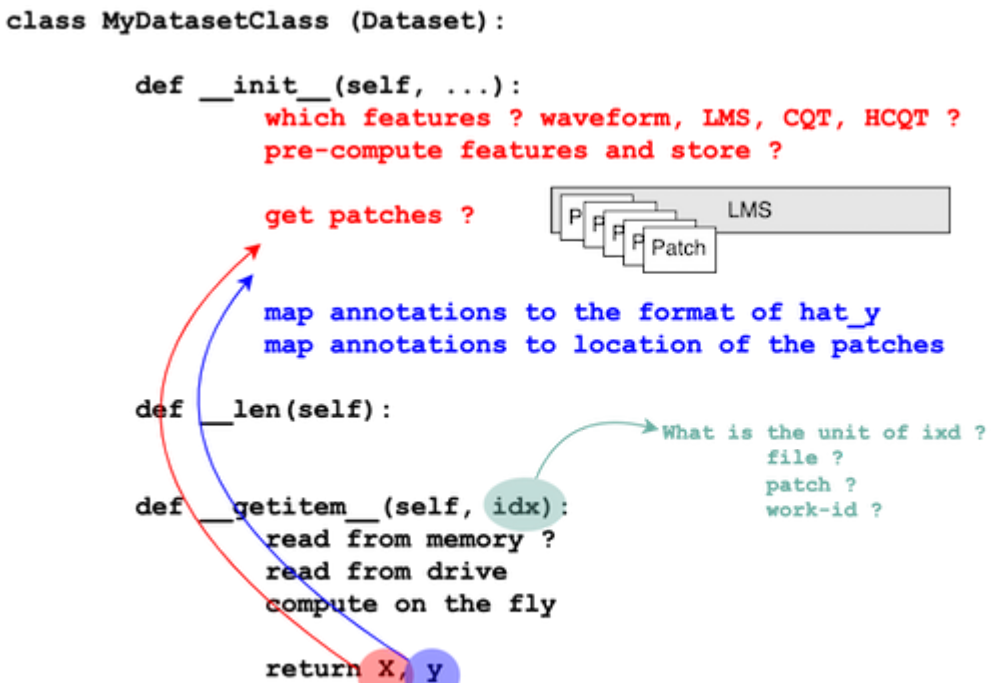
In this, `train_dataloader` is an instance of the pytorch-class `Dataloader` which goal is to encapsulate a set of `batch_size` paris of input `X`/output `y` which are each provided by `train_dataset`.

```
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`.

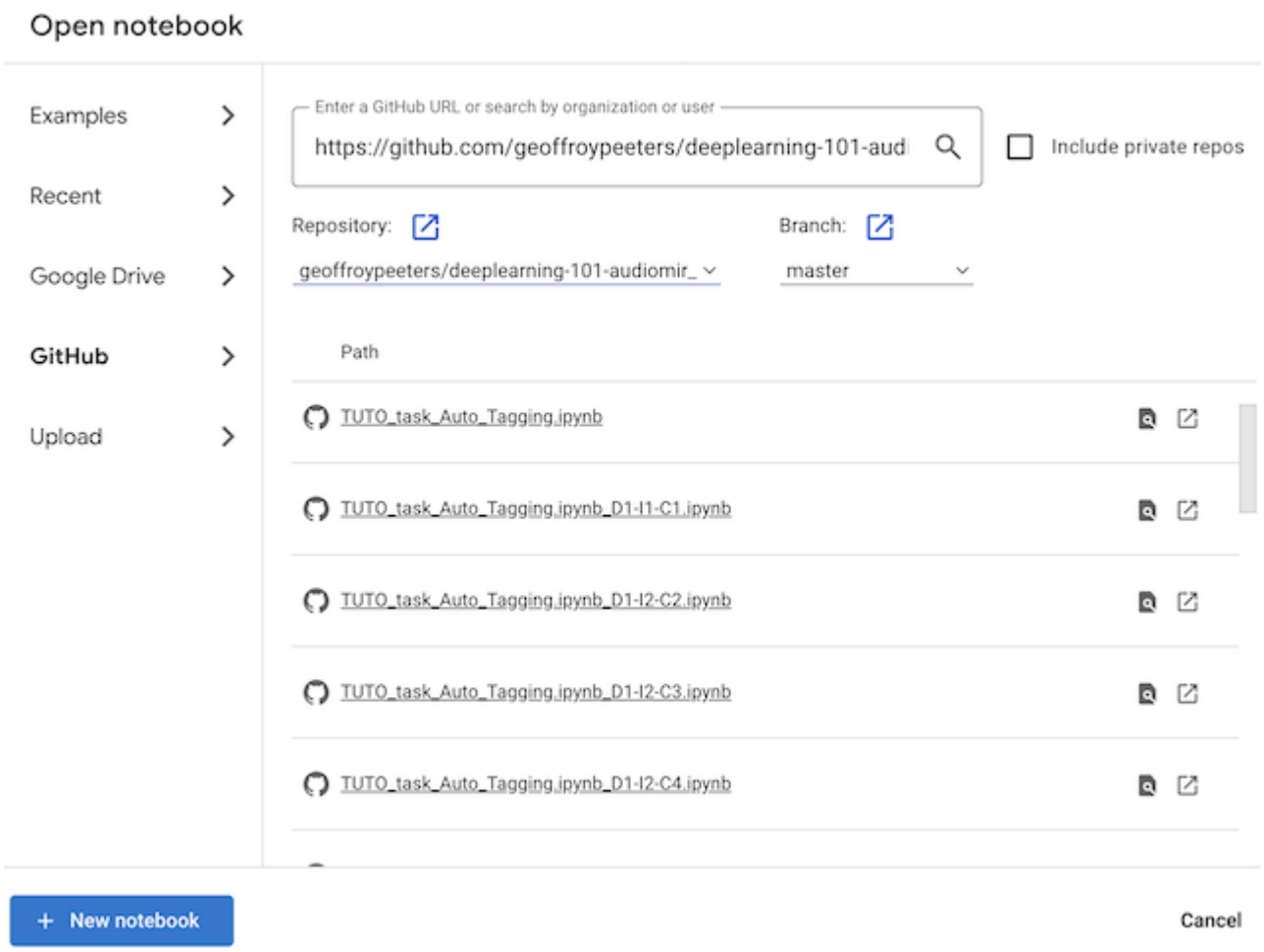


Fig. 4 Importing github files in Google Colab

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.

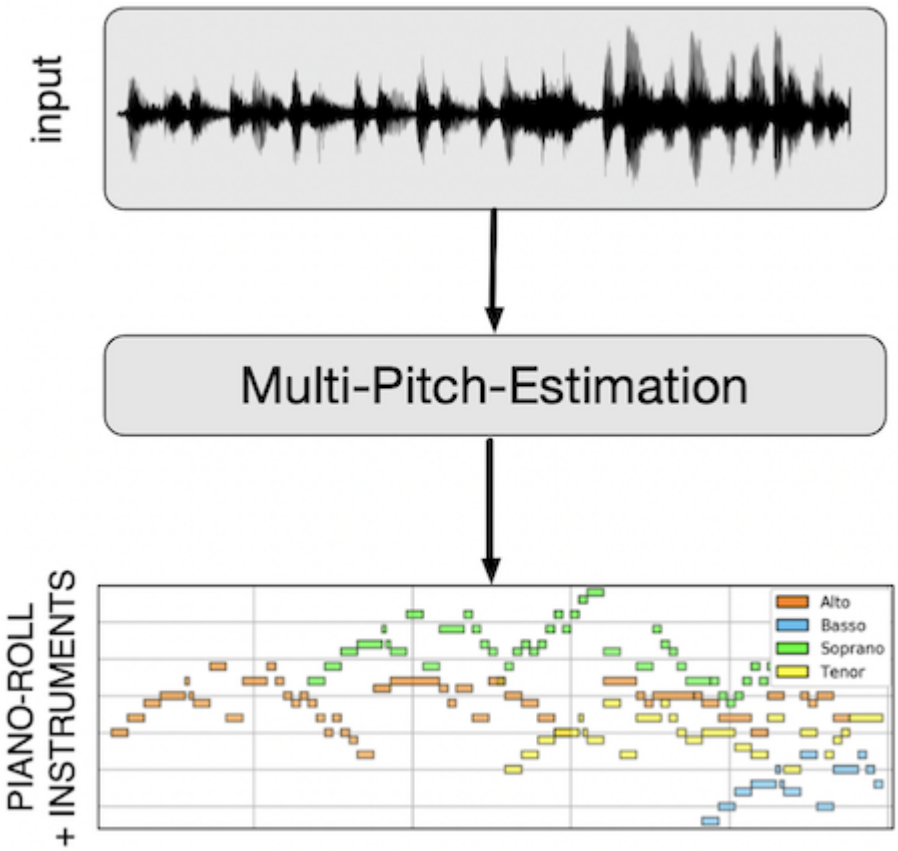
MIREX (Music Information Retrieval Evaluation eXchange) provides an extensive lists of such tasks.

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

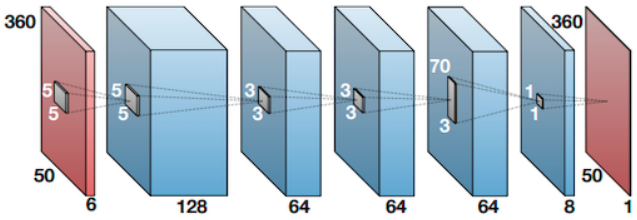


Figure *proposed by [BMS+17]*

U-Net model for MPE

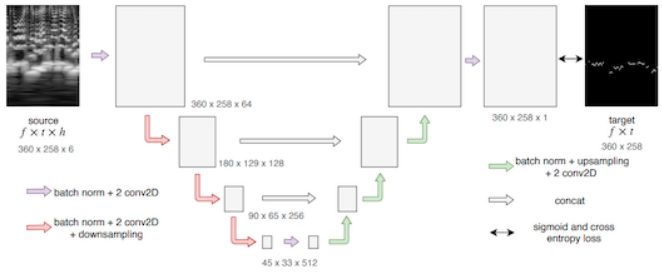
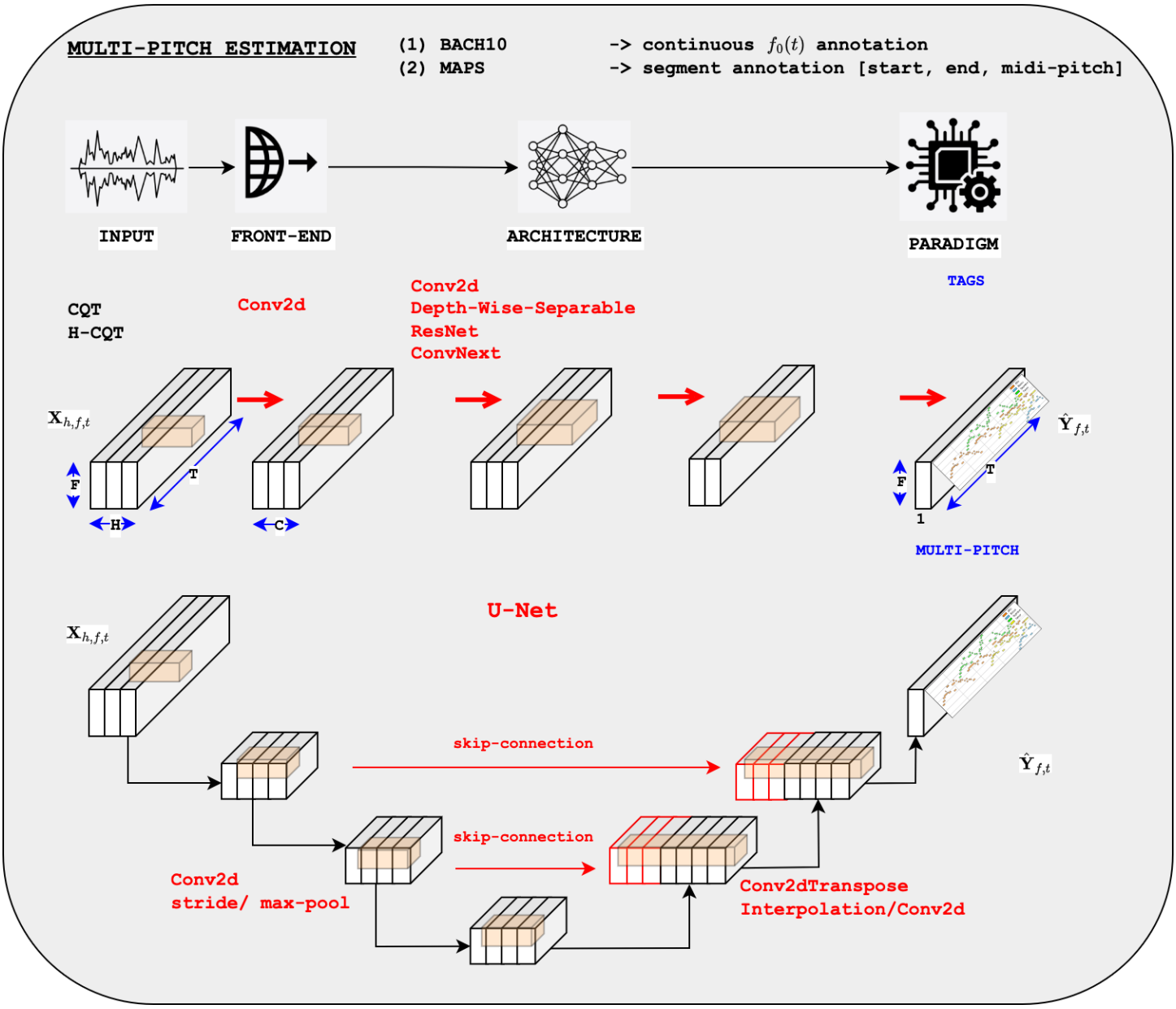


Figure *proposed by [DEP19, WP22]*

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)

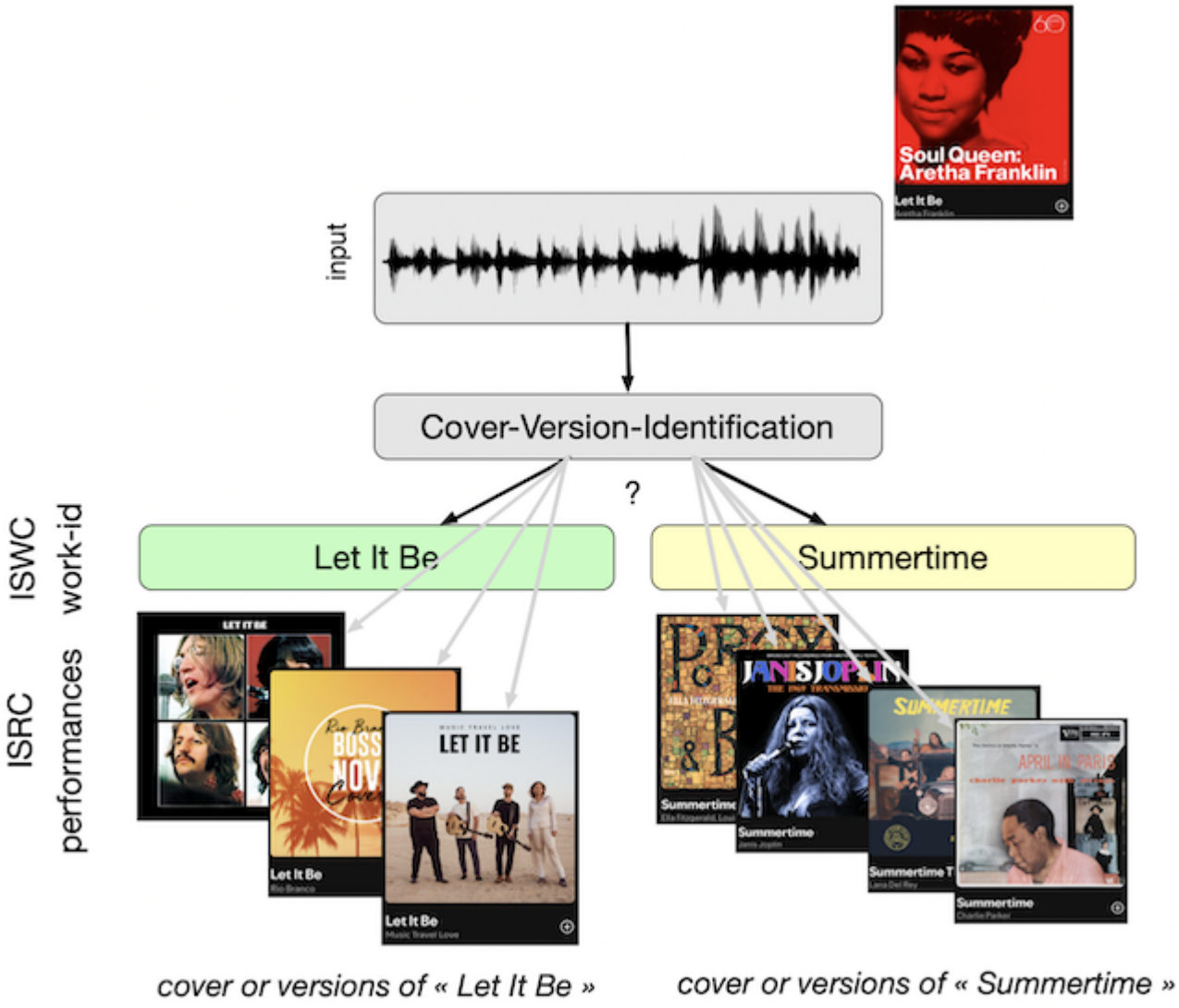
Goal of CSI ?

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

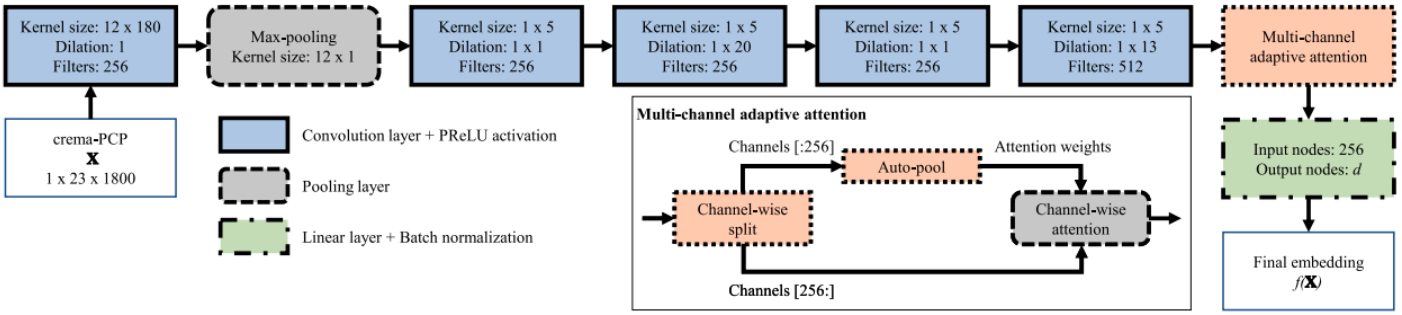
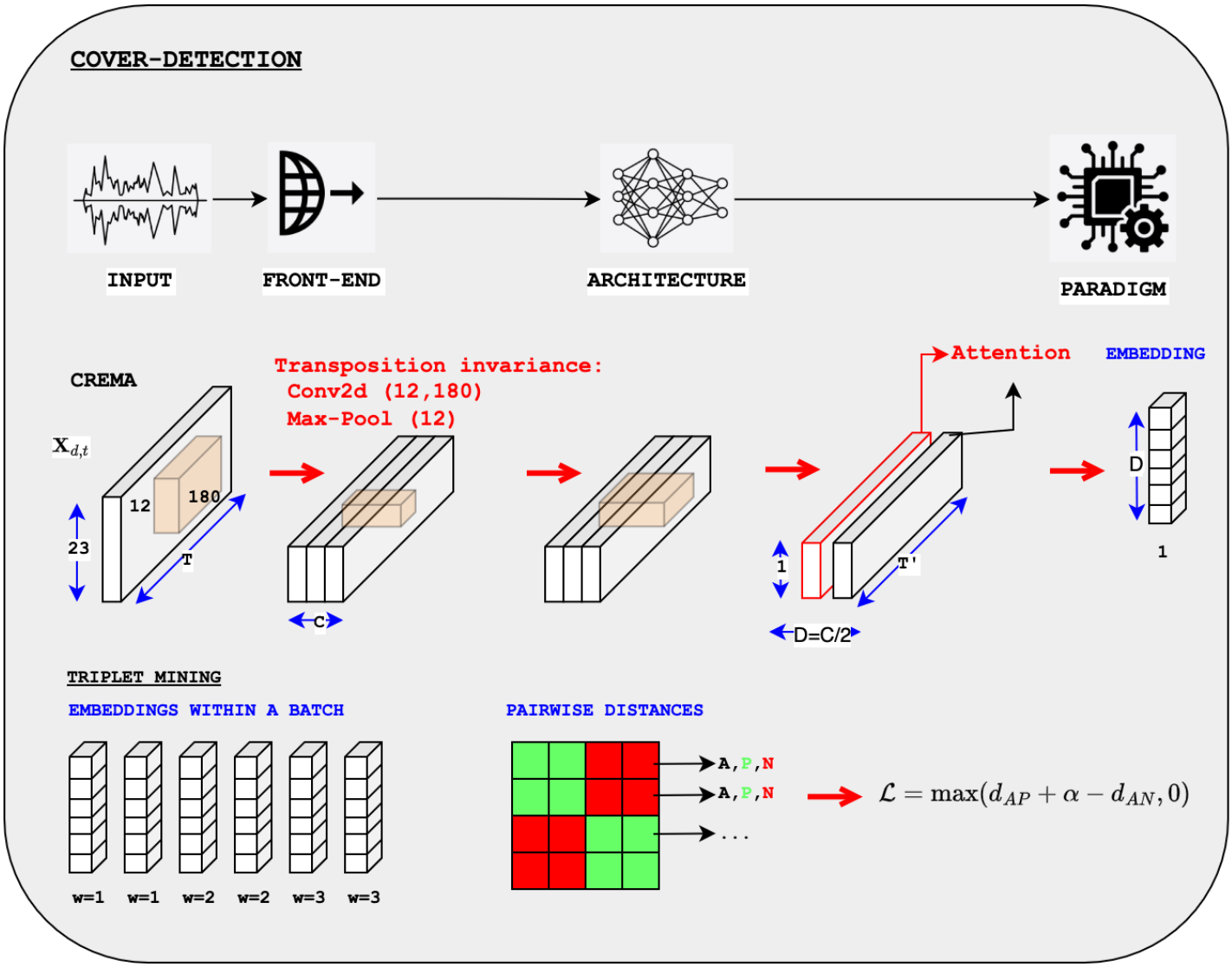


Figure *MOVE model for CSI proposed by [YSerraGomez20]*

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 $\mathbf{e}_i$.

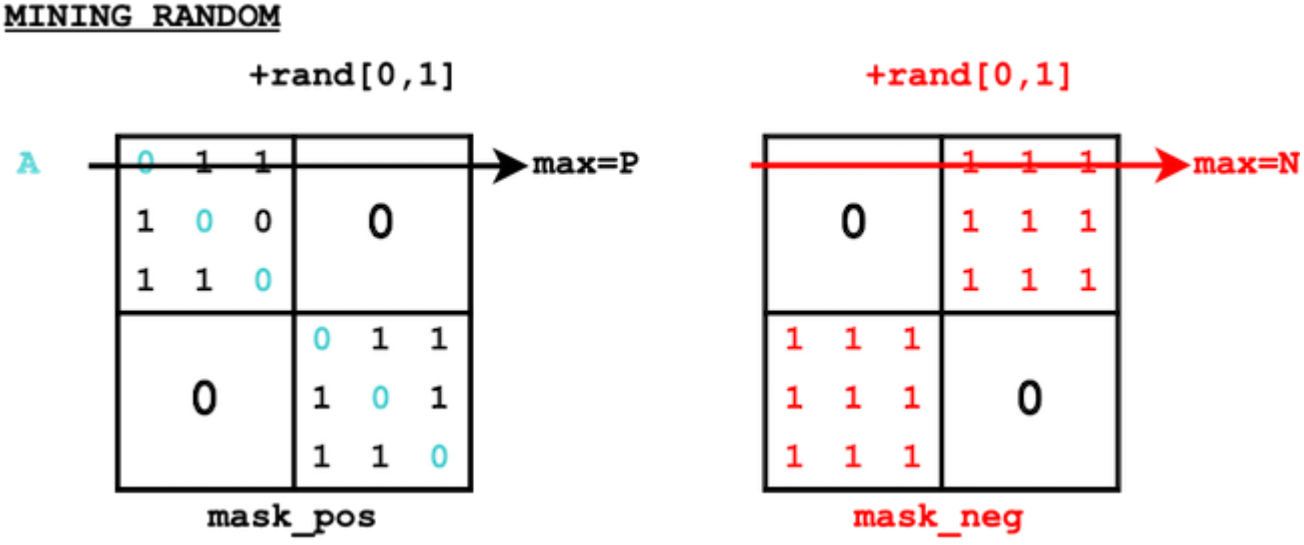
- The "cliques" are grouped together in the matrix, i.e. the performances $\{i-1, i, i+1\}$ belong to the same work-id.

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).



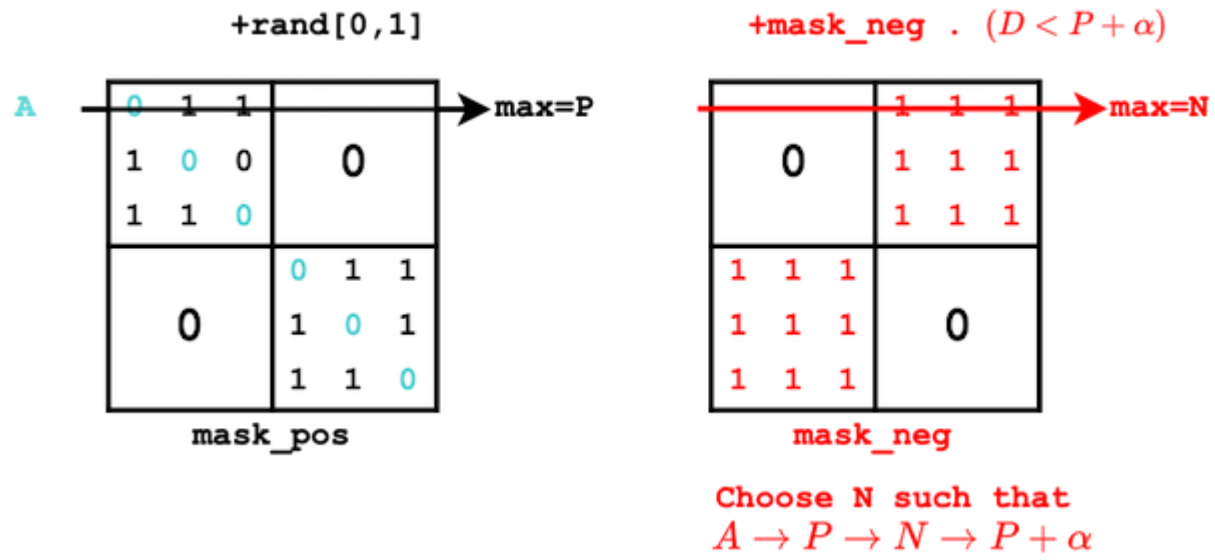

```
def triplet_mining_random(dist_all, mask_pos, mask_neg):
    """
    Performs online random triplet mining
    """
```

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).

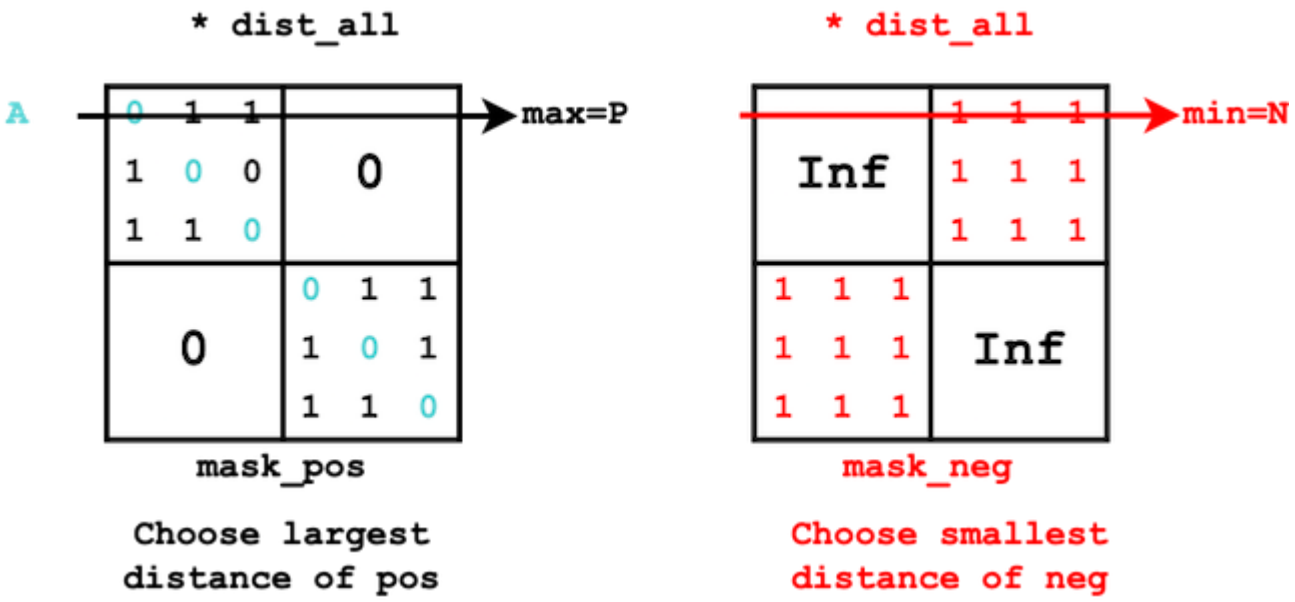



```
def triplet_mining_semihard(dist_all, mask_pos, mask_neg, margin):
    """
    Performs online semi-hard triplet mining (a random positive, a semi-hard negati
    """

    # --- 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)

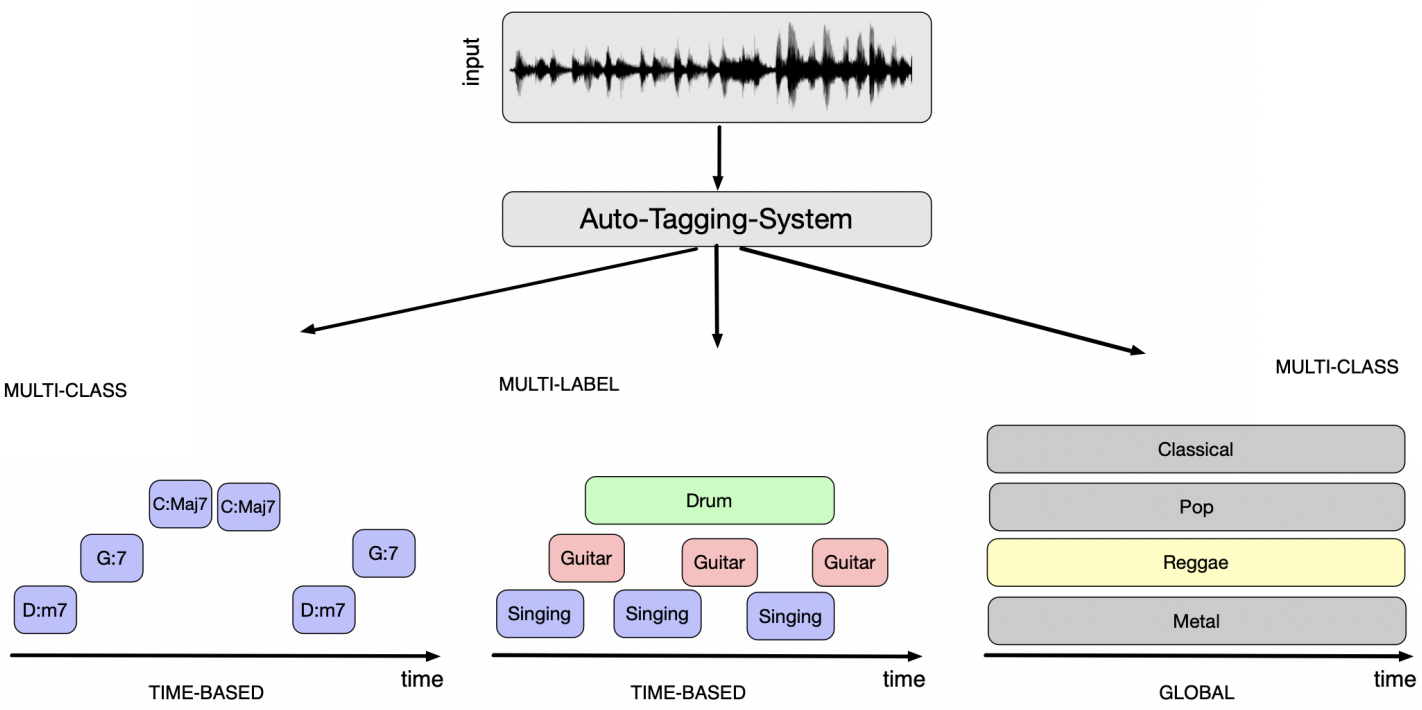
Goal of Auto-Tagging ?

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 $c \in \{1, \dots, C\}$.

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,


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.

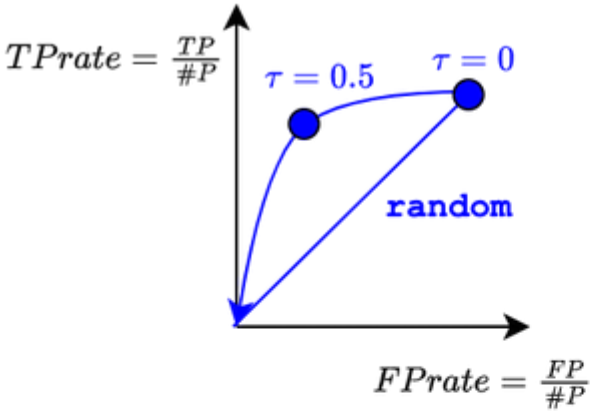
Using a default threshold ($\tau = 0.5$) of course allows to use the afore-mentioned metrics (Accuracy, Recall, Precision, F-measure).
However, in practice, we want to measure the performances independently of the choice of a given threshold.

This can be using either

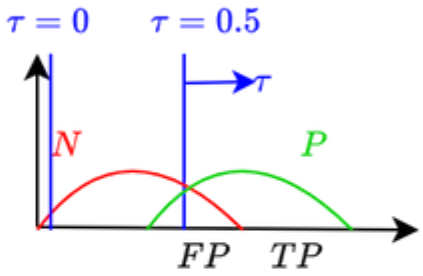
- the AUC (Area Under the Curve) of the ROC. The ROC curve represents the values of TPrate versus FPrate for all possible choices of a threshold τ. The larger the AUC-ROC is (maximum of 1) the more discrimination is between the Positive and Negative classes. A value of 0.5 indicates no discrimination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

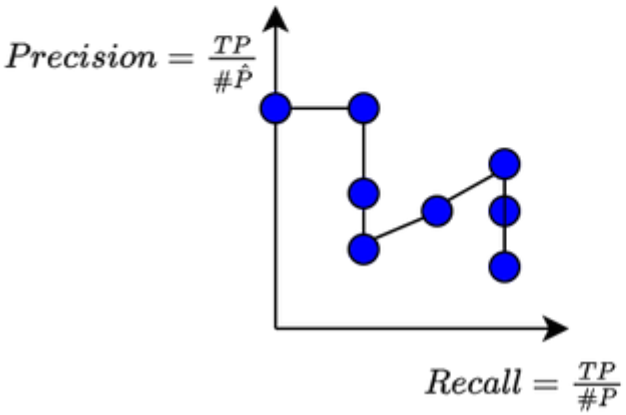
AUC ROC



	$\hat{P}$	$\hat{N}$	
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.

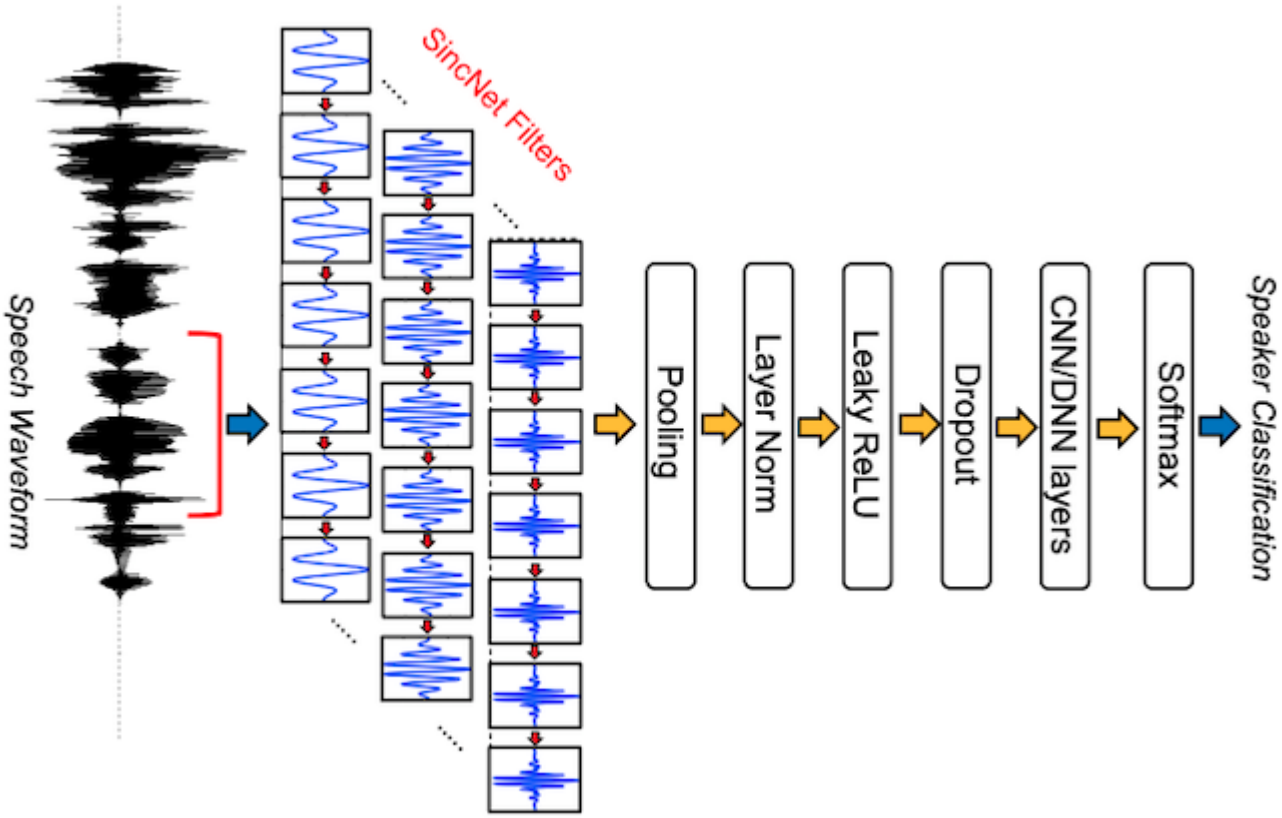
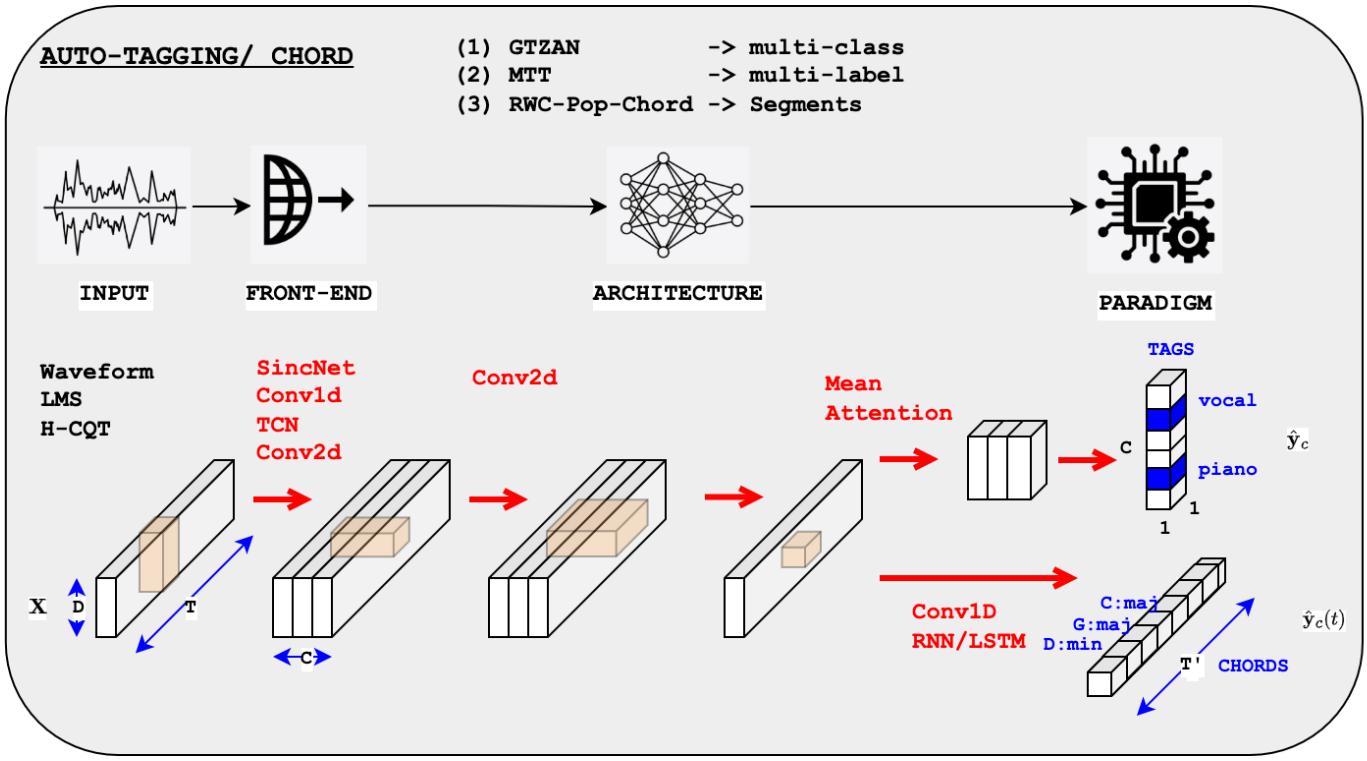


Figure. *SincNet model. image source: SincNet [RB18]*

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 (SSL)

- author: Gabriel, Alain

Music Audio Processing

Source Separation

- author: Gabriel

Musical Audio Generation

Goal of the Task

Musical audio generation aims to create various musical content, from individual notes [EAC+19] to instrumental accompaniments/arrangements [NPA+24] and complete songs [ECT+24]. In the early days of audio generation research, methods often focused on producing audio directly in the time or time-frequency domain. Recent approaches, however, work with compressed representations, often using neural audio codecs.

The most widely used models today are autoregressive (Transformer) architectures and diffusion models. Autoregressive architectures are particularly effective for discrete codecs, while diffusion models are better suited for continuous representations.

Popular Datasets

- **NSynth**: NSynth was once the go-to dataset for musical audio generation and can be regarded the "MNIST" in audio. It contains short, synthetic, single-note samples from different

- **GTZAN**: The GTZAN dataset is often used for genre classification and can serve as a starting point for more complex audio generation tasks involving diverse genres.
- **MusicNet**: Contains recordings of classical music with aligned annotations, suitable for tasks involving complex musical structures.
- **MAESTRO**: The MAESTRO dataset features piano performances, providing MIDI and corresponding audio recordings. This makes it particularly useful for training models of high-quality piano music generation.
- **MagnaTagATune**: Offers a large collection of music with tags, useful for genre classification and multi-label tasks.

How is the Task Evaluated?

Evaluation of generation tasks is difficult. In other ML tasks, specific targets (e.g., labels, data points) are available in a given evaluation set, allowing precision estimation for a given model. In contrast, in audio generation, the goal is to sample from the distribution of the training set without directly reproducing any training data.

As a result, indirect, distribution-based evaluation metrics are commonly used rather than relying on one-to-one comparisons, as in autoencoders or classification tasks.

Frechet Audio Distance (FAD)

Nowadays, the most commonly used metric in assessing the quality of generated audio is the Frechet Audio Distance (FAD) [KZRS19]. It compares the statistics of generated audio to those of real, high-quality reference samples in the embedding space of a pre-trained model. The idea is to assess the “closeness” of the two distributions: one for the generated samples and one for real samples.

Origins and Motivation

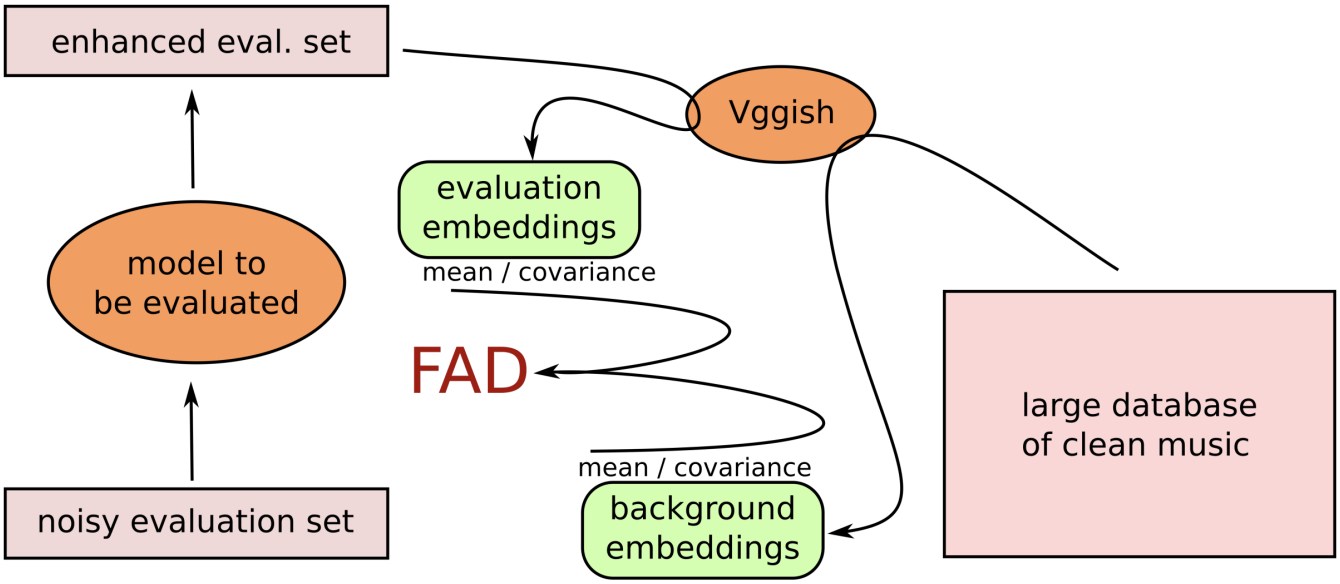


Figure 1: FAD computation overview for a music enhancement system as initially proposed (image source: [KZRS19]).

Fréchet Audio Distance was initially developed to evaluate music enhancement algorithms (see Figure 1). It filled a gap in objective audio quality evaluation, especially in generative tasks like music synthesis, audio inpainting, and speech generation. Before FAD, audio evaluations often relied on metrics like mean squared error (for reconstruction-based approaches) or subjective listening tests. While subjective tests remain essential for evaluating the perceptual quality of audio, FAD offers a more automated, quantifiable approach that aligns with perceptual quality.

Calculation of FAD

The FAD metric works by embedding audio signals into a perceptual feature space using a pre-trained deep neural network model. Initially a VGGish model was proposed, but it has been shown that LAION CLAP embeddings [WCZ+23] or a specific PANN model [KCI+20] align better with perceived audio quality [GGBE24, TLL+24]. Once embedded, it treats these embeddings as multidimensional distributions and calculates the Fréchet Distance (also known as the Wasserstein-2 distance) between the two distributions.

Mathematically, it involves comparing the means and covariances of these distributions:

$$\mathrm{FAD} = \|\mu_r - \mu_g\|^2 + \mathrm{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$$

Skip to main content

where μ_r and Σ_r are the mean and covariance of the reference (real) distribution, and μ_g and Σ_g are those of the generated distribution.

Applications of FAD

FAD is widely used in research on generative audio models, including:

1. **Music and Sound Generation**: Evaluating GANs or VAEs that generate music, sound effects, or synthetic soundscapes.
2. **Speech Synthesis**: Benchmarking TTS (Text-to-Speech) systems to gauge how close generated speech is to natural human voices.
3. **Audio Super-Resolution**: Comparing high-resolution generated audio with real high-resolution samples.
4. **Denoising and Enhancement**: Assessing the quality of denoised or enhanced audio by comparing it to clean reference audio.

Since its introduction, FAD has become a standard metric for evaluating the realism and quality of generated or processed audio.

Certainly! Here's a compact explanation of the **Inception Score (IS)** and **Kernel Inception Distance (KID)**, based on typical usage in evaluating generative models:

Inception Methods

The following methods apply if class labels are available for the training data of a generative model. They rely on a so-called *Inception network* – a classifier trained to predict the class labels from the data.

Inception Score (IS)

The **Inception Score** is a metric that evaluates the quality and diversity of generated samples by measuring two key properties: (1) **high confidence** in classifications (suggesting realistic, distinct samples) and (2) **diversity** across classes (indicating a wide variety of generated outputs).

It uses a pre-trained Inception network to classify generated samples and calculates the score as follows:

$$\text{IS} = \exp\left(\mathbb{E}_x\left[D_{\text{KL}}(p(y|x)\|p(y))\right]\right)$$

In this formula:

- $p(y|x)$ represents the conditional class distribution given a generated sample x, where higher confidence corresponds to more realistic samples.
- $p(y)$ is the marginal class distribution across samples, promoting diversity if samples cover a wide range of classes.
- The **Kullback-Leibler divergence** D_{KL} between $p(y|x)$ and $p(y)$ is averaged over all generated samples, capturing the balance between realism and diversity. The exponential of this average yields the final Inception Score, with higher values indicating better quality.

IS is commonly applied in image generation but can be adapted for use in generative audio and other domains.

Kernel Inception Distance (KID)

Kernel Inception Distance (KID) is a metric that evaluates the similarity between real and generated samples. Unlike IS, which only uses generated samples, KID compares the distributions of real and generated data using features extracted by a pre-trained Inception network. KID is calculated by computing the **squared Maximum Mean Discrepancy (MMD)** between the embeddings of real and generated samples, with a polynomial kernel for smoothing:

$$\text{KID} = \|\mathbb{E}[\phi(x_r)] - \mathbb{E}[\phi(x_g)]\|^2$$

Here:

- $\phi(x_r)$ and $\phi(x_g)$ represent the feature embeddings of real and generated samples, respectively.
- MMD measures the difference in means of these embeddings, with KID summing the squared differences to capture distributional similarity.

Unlike FAD, which uses covariance matrices, KID operates without assumptions about distribution shape and generally provides unbiased estimates, especially in small sample sizes. Lower KID values suggest that generated samples closely match the distribution of real samples, making it suitable for generative tasks requiring high-quality and realistic outputs.

Subjective Evaluation

Objective evaluation metrics cannot capture all the details people care about when listening to audio. Therefore, it is very common (and important) in audio generation works to perform user studies where human listeners assess the perceived quality of audio, focusing on aspects like naturalness, clarity, and overall fidelity. For completeness, we also include methods in this section that are used to compare the audio quality of two or more audio files, typically used in audio enhancement, super resolution or restoration.

For reliable results in all methods, it is crucial to conduct tests in controlled listening environments, ideally with high-quality audio equipment. Statistical analysis (like t-test) is often applied afterward to ensure the results are significant and unbiased.

Without Reference Samples

The following metrics are used in cases where an absolute reference isn't available, which typically applies to musical audio generation.

One of the most widely used methods is the **Mean Opinion Score (MOS)**, where listeners rate each audio sample on a numerical scale, typically from 1 to 5 (i.e., Likert Scale), with higher scores indicating better quality. MOS is popular because it gives a straightforward average score for quality, often applied in areas like audio generation, speech synthesis and audio enhancement.

Single Stimulus Rating (SSR) allows listeners to rate each sample individually. This method is helpful when comparing samples of widely differing qualities, such as generated audio, without needing a reference sample.

Attribute-Specific Rating (ASR) asks listeners to rate audio on specific qualities, like brightness, clarity, or naturalness, giving a more nuanced evaluation across multiple dimensions. This approach is particularly useful when certain attributes are especially important, like naturalness in speech

With Reference Samples

In **Comparison Category Rating (CCR)**, listeners are presented with two audio samples, often a high-quality reference and a processed version, and rate the difference in quality between them. This method helps detect subtle quality changes by making a direct comparison, which is useful for assessing codecs and noise reduction techniques.

For more precise distinctions, **ABX Testing** is a go-to approach. In this test, listeners hear three samples—two known (A and B) and one unknown (X) that matches either A or B. They must identify which one X corresponds to, revealing subtle perceptual differences. ABX tests are commonly used to test the transparency of audio processing techniques.

Another evaluation approach is the **Degradation Category Rating (DCR)**, which involves rating the perceived degradation of an audio sample relative to a high-quality reference. Listeners rate how much the quality has deteriorated, ranging from "imperceptible" to "very annoying," making DCR effective in testing the negative impacts of audio processing and compression methods.

In the **MUSHRA (MUltiple Stimuli with Hidden Reference and Anchor)** test, listeners rate multiple versions of the same audio on a scale from 0 to 100, with both high-quality and low-quality references included. MUSHRA provides detailed insight across various conditions, making it useful for codec testing and audio enhancement evaluations.

Basics of Generative Modeling

In generative tasks, it is necessary to inject *some form of stochasticity* into the generation process. In this regard, two general approaches can be distinguished: **Autoregressive** generation of *discrete sequences* and **Non-Autoregressive** (or parallel/latent variable) generation of *continuous-valued data*. In this section, we will have a brief look into the two paradigms and give some examples of how they are modeled.

Autoregressive Generation

For **discrete sequences**, models such as Recurrent Neural Networks (RNNs) [Elm90], Causal

entropy loss to output a probability distribution over discrete random variables in a deterministic manner. The stochasticity is then "injected" by sampling from that distribution.

At each time step t, the model outputs a probability distribution $P(y_t \mid y_{<t})$ over the vocabulary V, conditioned on the previous tokens $y_{<t}$. The cross-entropy loss used during training can be expressed as:

$$\mathcal{L}_{\text{CE}} = -\sum_{t} \log P(y_t^* \mid y_{<t})$$

where y_t^* is the true token at time t.

Note: In this case, we can primarily deal with **one-hot encoded sequences**, selecting one token per time step, as we don't have a simple way to sample **N-hot vectors** (where $N > 1$ tokens are selected simultaneously) from the model's output distribution. Sampling multiple tokens at once would require modeling the joint probability of combinations of tokens, which significantly increases complexity and is not commonly addressed in standard sequence generation models.

Non-Autoregressive/Latent Variable Generation

For generating **continuous-valued data**, the stochasticity usually comes from some form of noise injection into the neural network. Mathematically, this is typically defined as transforming a simple (usually Gaussian) distribution into the data distribution. In the following, a brief (architecture-agnostic) introduction in the most common training paradigms is given.

Generative Adversarial Networks (GANs)

For example, Generative Adversarial Networks (GANs) [GPougetAbadieM+14] in their basic form inject noise by inputting a high-dimensional noise vector $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (sampled from an independent Gaussian distribution) into the generator G. The generator transforms this noise vector into a data sample: $\mathbf{x} = G(\mathbf{z})$. Thus, the task can be described as learning to transform an independent Gaussian distribution into the data distribution.

The generator is trained by playing an adversarial game with a discriminator D. The discriminator aims to distinguish between real samples from the dataset and fake samples generated by G. The

generator is trained to produce samples that maximize the likelihood of fooling the discriminator. This can be formalized by the minimax game between G and D

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

where $p(\mathbf{z})$ represents the distribution of noise input. This adversarial setup ensures that as D improves in distinguishing real from fake data, G improves in generating more realistic samples, ultimately leading to convergence when the generated data becomes indistinguishable from the real data.

Variational Autoencoders (VAEs)

Similarly, in Variational Autoencoders (VAEs, composed of encoder and decoder) [KW14], the decoder receives as input a sample from an independent Gaussian prior distribution (a "standard normal distribution"). The model is trained so that the encoder learns to approximate the prior using a mixture of Gaussian posteriors, one for each data point: $\boldsymbol{\mu}, \boldsymbol{\sigma} = E(\mathbf{x})$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are multi-dimensional mean and variance vectors. From this posterior, we sample a latent variable $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$ during training. The decoder then reconstructs the input by transforming $\mathbf{z}$ into a data point $\hat{\mathbf{x}} = D(\mathbf{z})$.

Training involves minimizing two objectives: the reconstruction loss between $\mathbf{x}$ and $\hat{\mathbf{x}}$, and the Kullback-Leibler (KL) divergence between the learned posterior $q_\phi(\mathbf{z} \mid \mathbf{x})$ and the prior distribution $p(\mathbf{z}) \sim \mathcal{N}(0, I)$. The two objectives are adversarial because the KL term pushes the posteriors towards a zero mean and unit variance, while the reconstruction term encourages the posteriors to adopt distinct means and reduced variances, allowing each data point to have its own distribution. Together, they make it possible to sample from the prior $p(\mathbf{z}) \sim \mathcal{N}(0, I)$ at inference and decoding it into a plausible data sample: $\hat{\mathbf{x}} = D(\mathbf{z})$.

Diffusion Models

In Diffusion Models [HJA20], the noise input has the same dimensionality as the data point that should be generated. The model gradually transforms noise into data through a series of steps. Like before, the goal is to transform a Gaussian prior distribution into the data distribution through the learned denoising steps. In its initial form it is defined as a Markov chain with learned Gaussian

transitions starting at $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$. The model learns to reverse the noising process by estimating $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \sigma_t^2 \mathbf{I}).$$

Early Works

Before the rise of Transformers and Diffusion Models, models like Causal Convolutional Networks, Recurrent Neural Networks (RNNs), and Generative Adversarial Networks (GANs) were used for musical audio generation. At the time, it was common to generate in a low-level representation space, either directly in the signal domain (WaveNet, SampleRNN) or in the spectral domain (GANs). Not least, due to their generation in such a high-dimensional space, CNNs/RNNs struggled with long-term dependencies, leading to repetitive or incoherent results without higher-level structure.

GANs were used to generate audio in the signal or frequency domain but faced challenges with training instability and producing high-quality, diverse outputs. Through the usage of neural audio codecs and the resulting reduction in dimensionality, the problem became simpler. Nowadays, through a combination of more efficient/simpler to-train generative models with generation in a compressed space, it is possible to generate high-quality, full-length music tracks.

WaveNet

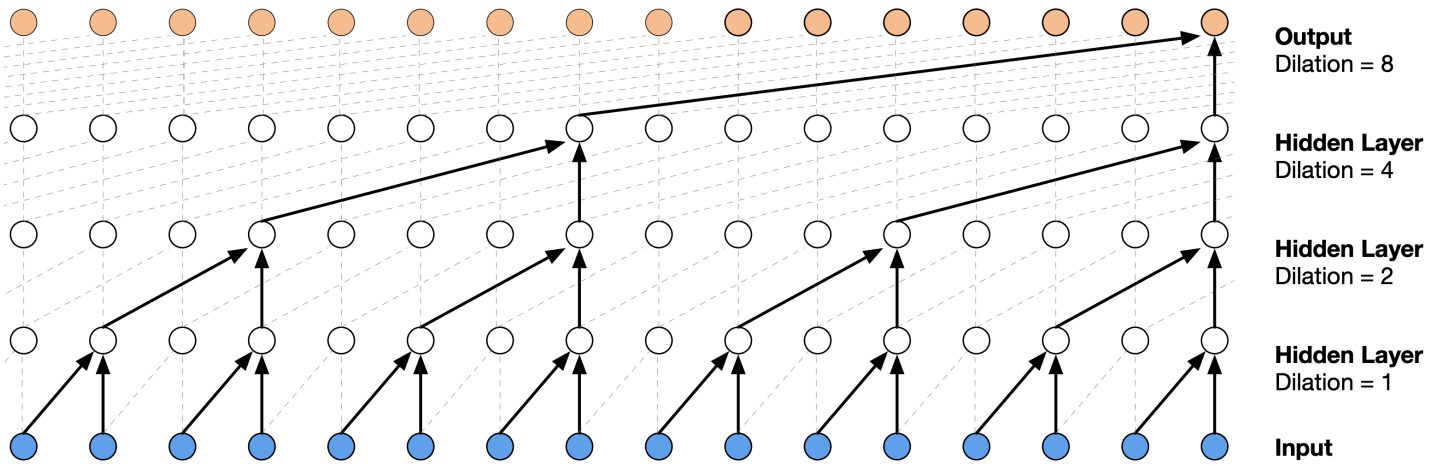


Figure 1: WaveNet architecture showing causal, dilated convolutions (image source: [vdODZ+16])

Skip to main content

WaveNet [vdODZ+16] can be seen as the first successful attempt to directly generate audio using a Neural Network. Important components in WaveNet are dilated convolutions [YK16] that enable an exponentially growing receptive field with linearly increasing numbers of layers. A big receptive field is critical in WaveNet because it operates directly in the signal domain with 16k samples/second. In addition, causal convolutions are used to prevent the model from looking into the future during training, resulting in a generative autoregressive sequence model.

Autoregressive sequence models are typically trained with cross-entropy loss that requires one-hot encoded sequences. As raw audio is usually 16-bit, a naive transformation into one-hot vectors would result in 65,536 dimensions per time step. To keep the problem tractable, in WaveNet, each sample is non-linearly scaled and quantized to obtain 256-dimensional vectors. The non-linear scaling function (ITU-T, 1988) is defined as

$$f(x_t) = \text{sign}(x_t) \frac{\ln(1 + \mu|x_t|)}{\ln(1 + \mu)}.$$

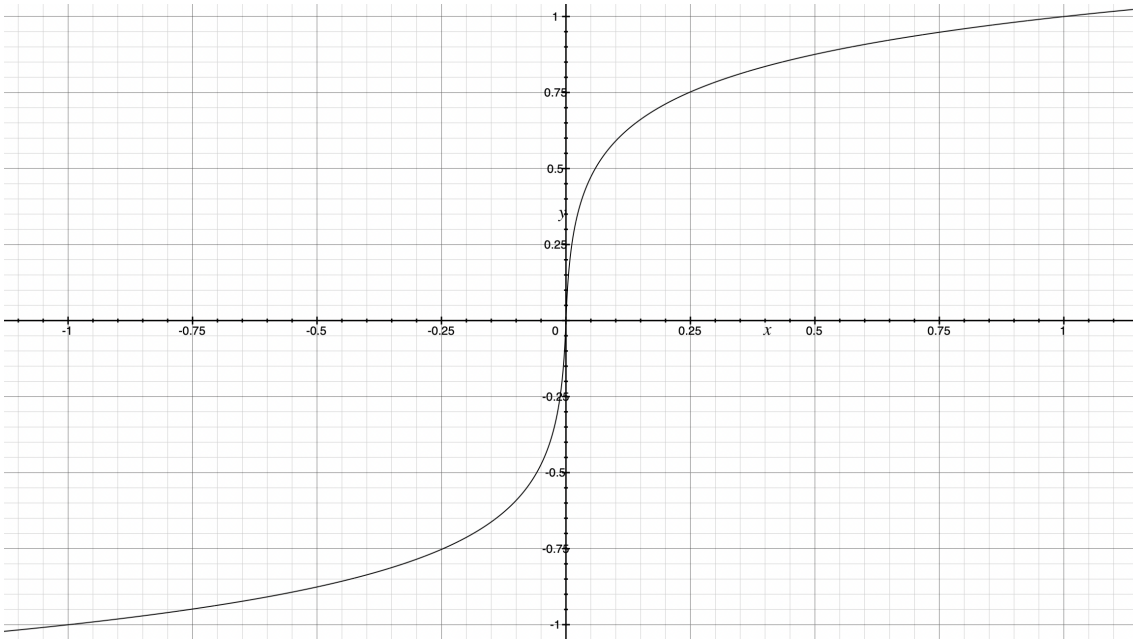


Figure 2: Non-linear scaling of audio samples in WaveNet for $\mu = 255$ (in practice, $-1 < x_t < 1$).

Usage Example: WaveNet was used by Google for text-to-speech (TTS) applications.

Skip to main content

SampleRNN

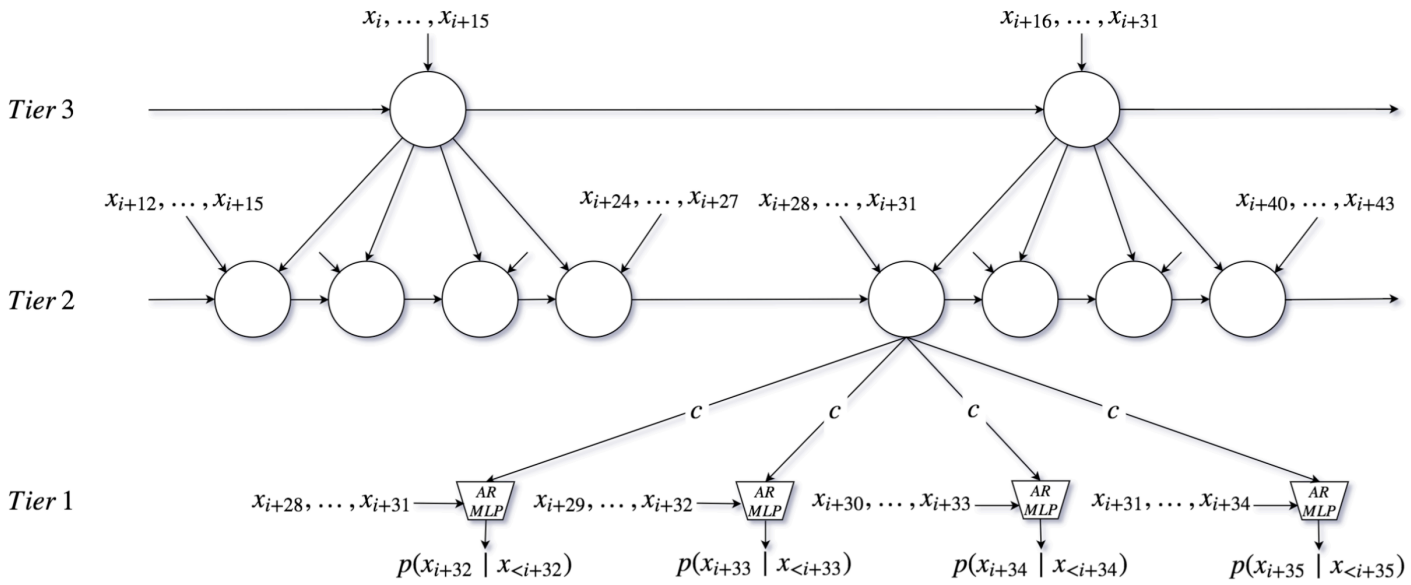


Figure 3: Snapshot of the unrolled SampleRNN model at timestep i with 3 tiers. As a simplification, only one RNN and up-sampling ratio $r = 4$ is used for all tiers (image source: [MKG+17]).

SampleRNN [MKG+17] was the first RNN-based neural audio synthesizer that had an impact in the community. It can effectively learn to generate long-form audio at a sample rate of 16kHz. While **WaveNet** builds hierarchical representations of audio by its built-in sub-sampling through dilated convolutional layers, SampleRNN builds such a hierarchy through multiple tiers of RNNs that operate in different "clock rates". This approach enables representations at varying temporal resolutions, where lower tiers (faster rates) are conditioned on higher tiers. This encourages higher tiers to generate higher-level signal representations that help predict lower-level details.

Similarly to WaveNet, in order to keep the task tractable, the sample values are quantized to 256 bins—but without prior, non-linear scaling. As the memory of RNNs can be updated iteratively without the need to reconsider past inputs, they tend to need less compute at inference time than non-recurrent autoregressive models (like causal convolutions or transformers).

Usage Example: Different artists used SampleRNN for music generation. Notably, a livestream (by Dadabots) with Technical Death Metal music is ongoing with hardly any interruptions since March 2019 [CZ18].

Skip to main content

Generative Adversarial Networks

For several years, Generative Adversarial Networks (GANs) [GPougetAbadieM+14] were among the most influential generative models. Their ability to implicitly model multi-dimensional *continuous-valued* distributions made them a compelling tool for image and audio generation. This enabled the use of spectrogram (or spectrogram-like) representations in audio generation, which is a natural modality for 2D convolutional networks. Another motivation for using image-like spectrogram representations with GANs for audio generation was the ability to leverage insights from the broader image-processing community.

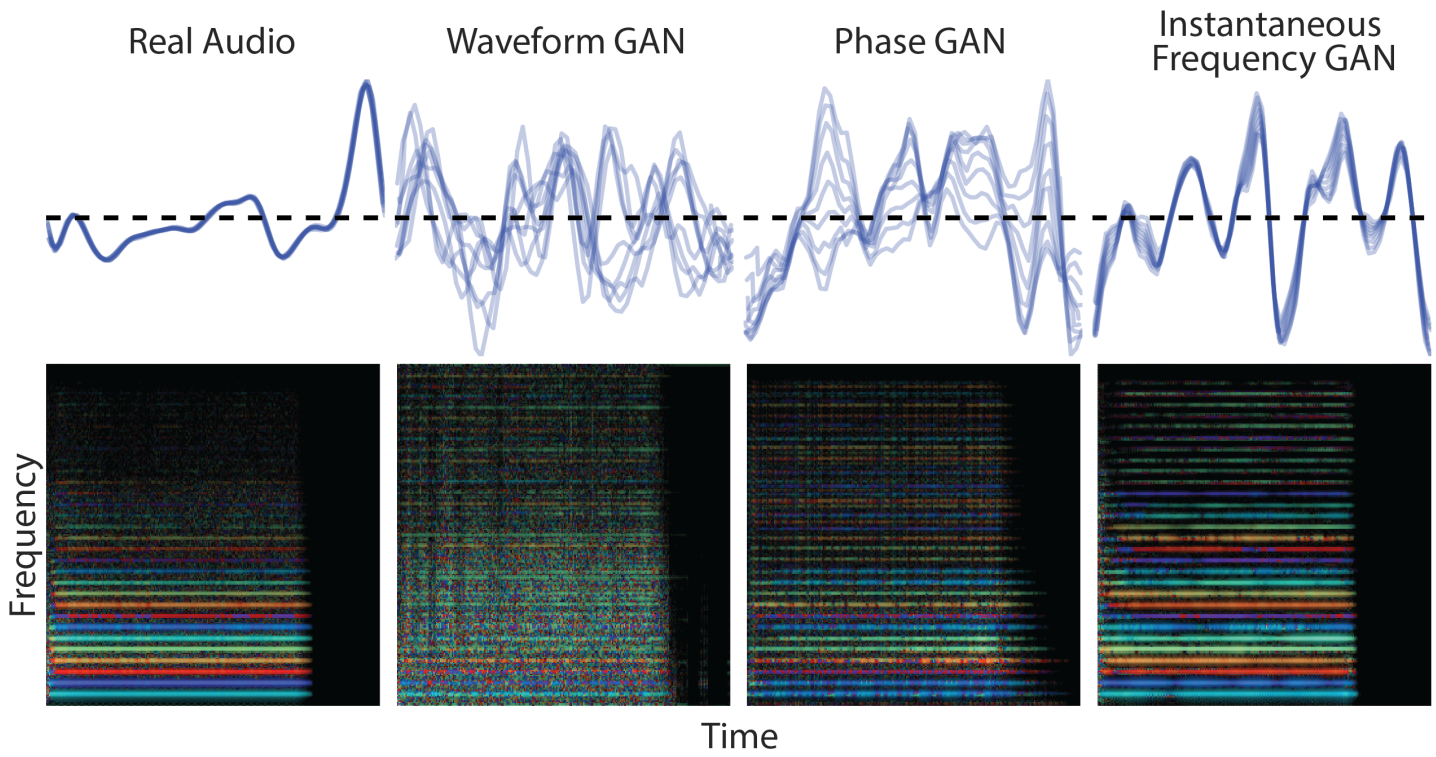


Figure 4: GANSynth rainbowgrams to showcase the influence of different audio representations (image source: [EAC+19]).

While WaveGAN [DMP19] was an influential work on using GANs directly for raw musical audio waveform generation, most works focussed on spectrogram-like representations. Examples for that are GANSynth [EAC+19], SpecGAN [DMP19], DrumGAN [NAVL22, NLR20], and DarkGAN [NLR21], omitting those only applied to speech. For simplicity reasons, the listed works can generate fixed-size outputs only. Some (later) examples of variable-size musical audio generation using GANs are VQCPC-GAN [NALR21] and Musica! [PSchluter22].

Presently, GANs are widely replaced by Diffusion Models, which are more stable in training, less prone to mode-collapse, and have a simpler architecture, resulting in higher-quality outputs.
A concept of GANs that could remain in the mid-term is the usage of adversarial losses from auxiliary networks, for example, for domain confusion or as additional loss in reconstruction-based training (e.g., in neural audio codecs, like DAC [KSL+23]).

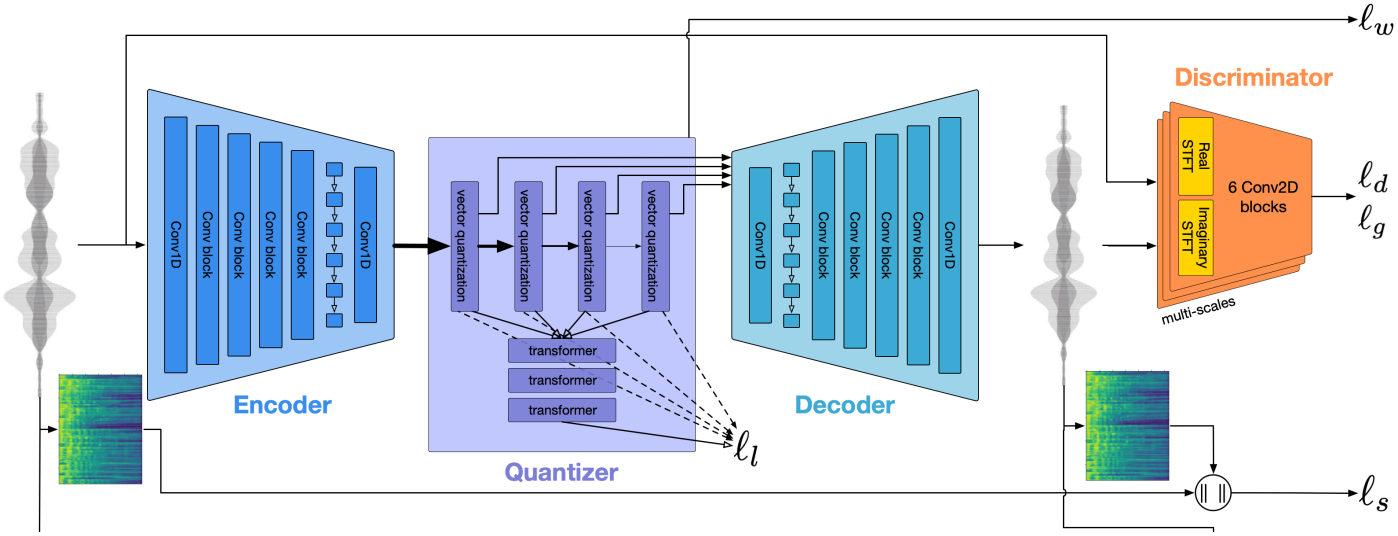
Usage Example: DrumGAN was the first commercially available neural audio synthesizer for music integrated into Steinberg's Backbone. It is now available for free as an online app.

Autoregressive Generation

In this tutorial, we demonstrate a practical implementation of musical audio generation using an autoregressive Transformer model and discrete audio representations obtained from a neural audio codec. We focus on the guitar subset of the NSynth dataset and use EnCodec representations to train our model.

EnCodec Neural Audio Codec

EnCodec [DefossezCSA23] is a neural audio codec that compresses audio signals into discrete latent codes, achieving high compression rates while maintaining audio quality. It encodes audio into a sequence of discrete tokens using quantized latent vectors from multiple codebooks. This transforms the continuous audio generation problem into a discrete sequence modeling task, suitable for autoregressive models like Transformers.



Skip to main content

Figure 1: Overview of the EnCodec architecture and training. The input audio is encoded into discrete tokens using residual vector quantization (image source: [DefossezCSA23]).

To use EnCodec for our task, we first encode our dataset into discrete token sequences.

```
!pip install encodec

from encodec import EncodecModel
from encodec.utils import convert_audio

# Load the EnCodec model
codec = EncodecModel.encodec_model_24khz()
codec.set_target_bandwidth(1.5)  # Set target bandwidth in kbps
LEVELS = 2 # 2 for bandwidth 1.5

# Function to encode audio into discrete tokens (schematic from DiscreteAudioRepData
def encode_audio(waveform, sample_rate):
    # Add batch dimension
    waveform = torch.tensor(waveform, dtype=torch.float32).unsqueeze(0)
    waveform = convert_audio(waveform, sample_rate, codec.sample_rate, codec.channe
    with torch.no_grad():
        encoded_frames = codec.encode(waveform)

    # we linearize the codebook
    codes = encoded_frames[0][0].contiguous().permute(0, 2, 1).reshape(-1)
    return codes.flatten()
```

Preparing the Dataset

We use the guitar subset of the NSynth dataset, encoding each audio file into discrete token sequences by initializing the `DiscreteAudioRepDataset` and the `DataLoader`.

```
# Download the NSynth guitar dataset
!git clone https://github.com/SonyCSLParis/test-lfs.git
!bash ./test-lfs/download.sh NSYNTH_GUITAR_MP3

# Load the NSYNTH dataset and prepare DataLoader for training and validation.
audio_folder_train = "./NSYNTH_GUITAR_MP3/nsynth-guitar-train"
audio_folder_val = "./NSYNTH_GUITAR_MP3/nsynth-guitar-valid"

dataset = DiscreteAudioRepDataset(root_dir=audio_folder_train, encoder=codec,
                                  lazy_encode=False, max_samples=-1)

dataset_val = DiscreteAudioRepDataset(root_dir=audio_folder_val, encoder=codec,
```

Skip to main content

```
# Create Dataloaders for training and validation.
dataloader = DataLoader(dataset, batch_size=125, shuffle=True)
dataloader_val = DataLoader(dataset_val, batch_size=125, shuffle=True)
```

Transformer Model Architecture

We use a classic Transformer decoder with *rotary positional embeddings* from x_transformers to model the sequence of discrete tokens autoregressively.

The model predicts the next token given the previous tokens.

```
from x_transformers import TransformerWrapper, Decoder

model = TransformerWrapper(
    num_tokens=1024,  # Vocabulary size from EnCodec
    max_seq_len=250,  # Maximum sequence length
    attn_layers=Decoder(
        dim=256,
        depth=6,
        heads=4,
        rotary_pos_emb=True
    )
)
```

Training and Inference

Training Objective

The model is trained to minimize the **cross-entropy loss** between the predicted token distribution and the true next token in the sequence:

$$\mathcal{L} = -\sum_{t} \log P(y_t^* \mid y_{<t})$$

where y_t^* is the true token at time t, and $y_{<t}$ are the previous tokens.

Skip to main content

Training Loop

We train the model using teacher forcing, where the true previous tokens are provided as input during training.

```
import torch.nn as nn

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
    for batch in dataloader:
        start_tokens = torch.zeros((batch.shape[0], 1))
        batch = torch.cat([start_tokens, batch], dim=1)

        logits = model(batch)
        logits = logits.permute(0, 2, 1)

        inputs = logits[..., :-1]  # All tokens except the last
        targets = batch[..., 1:]  # All tokens except the first

        loss = criterion(inputs, targets)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Inference and Generation

During inference, we generate new sequences by sampling tokens one at a time from the model's output distribution.

```
    LEVELS = 2 # 2 for bandwidth 1.5
    generated = [start_token]
    model.eval()
    for _ in range(seq_length):
        input_seq = torch.tensor([generated]).long()
        logits = model(input_seq)[:, -1, :]
        probs = torch.softmax(logits, dim=-1)
        next_token = torch.multinomial(probs, num_samples=1).item()
        generated.append(next_token)

    generated_sequence = torch.tensor(generated[1:]) # Remove start_token
```

```
    # Decode into waveform
    decoded_audio = codec.decode([(codes, None)])
```

Conclusion

In this tutorial, we demonstrated a simple implementation of musical audio generation using an autoregressive Transformer model and EnCodec representations. By encoding audio data into discrete token sequences, we transformed the audio generation problem into a sequence modeling task suitable for training with cross-entropy loss and sampling from a multinomial distribution.

Key Points:

- **Neural Audio Codec (EnCodec)**: Compresses audio into discrete tokens, enabling training using cross-entropy loss and sampling from a discrete distribution.
- **Autoregressive Transformer**: Models the probability distribution of the next token given previous tokens.
- **Training with Cross-Entropy Loss**: Trains the model to predict the next token in the sequence.
- **Sequence Generation**: Generates new audio samples by sampling tokens from the model.

This approach leverages the strengths of both neural audio codecs and autoregressive sequence models, enabling efficient audio generation in a compressed latent space.

Generation with Latent Diffusion

For non-autoregressive generation, Latent Diffusion is currently one of the most effective and popular approaches. It focuses on creating high-quality samples by modeling data in a compressed, latent space rather than the spectrogram or waveform level. This approach reduces computational cost and time by generating compact latent representations that, compared to raw data, are already simplified and organized.

Here, we provide a simple example of musical audio generation using the guitar subset of the NSynth dataset, Music2Latent representations, and a classic U-Net architecture. In the corresponding notebook we implement a diffusion model that adopts a **Rectified Flow** method

probabilistic models (DDPMs) and **normalizing flows**, resulting in a continuous-time framework for generative modeling.

Music2Latent Codec

Music2Latent (M2L) [PLF24] provides highly compressed, continuous audio representations with ~ 11×64-dimensional vectors per second (for 44.1kHz sample rate). A consistency autoencoder facilitates a *generative decoder* that makes up for potentially lost information, enabling high-quality reconstructions. The M2L representations serve as the data space in which the diffusion process operates. The diffusion model learns to generate M2L latent vectors, which are then decoded back into audio signals.

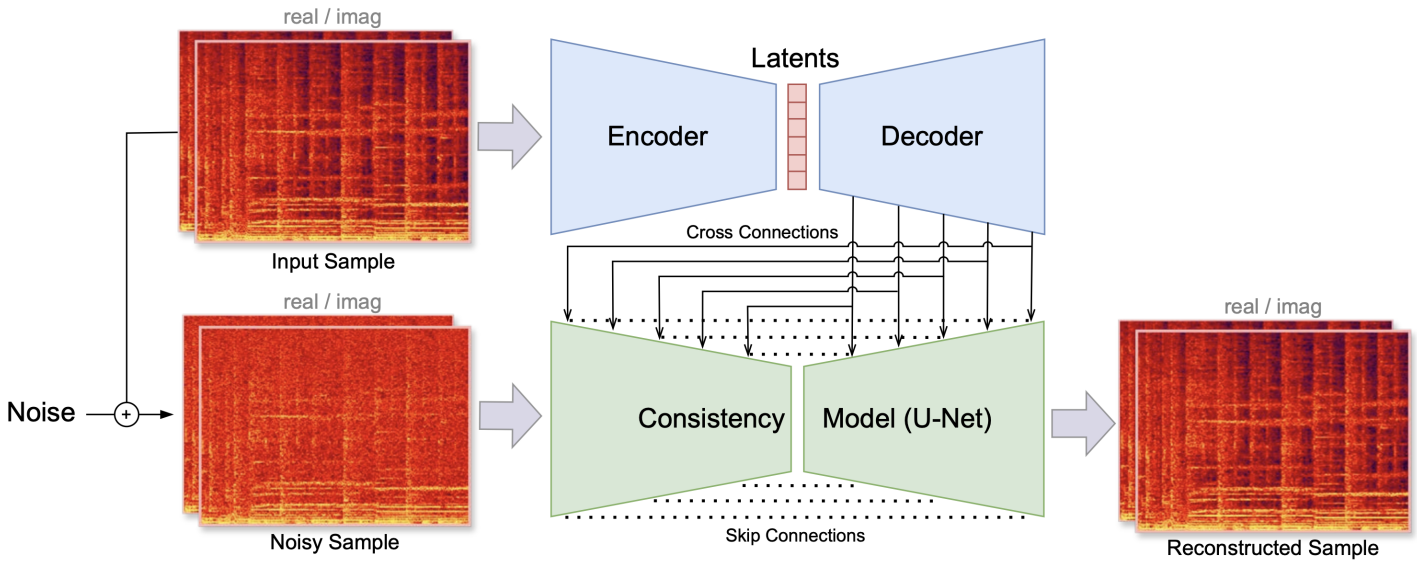


Figure 1: Training process of Music2Latent. The input sample is first encoded into a sequence of latent vectors. The latents are then upsampled with a decoder model. The consistency model is trained via consistency training, conditioned on the information coming from the cross connections (image source: [PLF24]).

In our example, we first transform the dataset into M2L representations on which we then train our diffusion model. This involves the following steps:

```
!pip install music2latent

import soundfile as sf
```

Skip to main content

```
# Initialize the encoder/decoder
encdec = EncoderDecoder()

# Load the audio file using soundfile
waveform, _ = sf.read("sample.wav")

# Encode using music2latent
latent = EncoderDecoder().encode(waveform)
```

Preparing the dataset

In the following code, we download the dataset, intitialize the `MusicLatentDataset` and the `DataLoader`:

```
# Download the dataset using git-lfs
!git clone https://github.com/SonyCSLParis/test-lfs.git
!bash ./test-lfs/download.sh NSYNTH_GUITAR_MP3

# Initialize the dataset and dataloader
audio_folder_train = "./NSYNTH_GUITAR_MP3/nsynth-guitar-train"
audio_folder_val = "./NSYNTH_GUITAR_MP3/nsynth-guitar-valid"

# When initializing the dataset, the data gets compressed using the M2L encoder
dataset = MusicLatentDataset(root_dir=audio_folder_train, encoder=encdec)
dataset_val = MusicLatentDataset(root_dir=audio_folder_val, encoder=encdec)

dataloader = DataLoader(dataset, batch_size=500, shuffle=True)
dataloader_val = DataLoader(dataset_val, batch_size=500, shuffle=False)
```

Diffusion Model Architecture

The `DiffusionUnet` class defines a classic U-Net architecture with time conditioning. The time embedding (`self.time_mlp`) is a multi-layer perceptron (MLP) that embeds the time variable t into a higher-dimensional space to condition the model on the diffusion time step. It is added to the deepest layer, allowing the model to adjust its predictions based on the amount of noise present.

```
def forward(self, x, t):
    # ...

    # Bottleneck
```

Skip to main content

```
    # Time embedding
    t_emb = self.time_mlp(t.unsqueeze(-1))  # [batch_size, channels]
    t_emb = t_emb.unsqueeze(-1)             # [batch_size, channels, 1]
    h = h + t_emb                           # Broadcast addition
    # ...
```

Training and Inference

In this section, we show the

1. **Noise Addition via Linear Interpolation**: The model adds noise to the data by linearly interpolating between the clean data $\mathbf{x}_0$ and pure noise $\mathbf{z}$, controlled by a time variable t as

$$\mathbf{x}_t = (1-t)\mathbf{x}_0 + t \cdot \mathbf{z}, \quad t \in [0,1],$$

   where t is the time step indicating the level of noise added, with $t = 0$ being no noise and $t = 1$ being full noise. This formulation is commonly used in Rectified Flow models and allows a simple, linear interpolation between the data and pure noise.

2. **Training Objective**: The model is trained to predict the residual $\mathbf{v} = \mathbf{x}_t - \mathbf{z}$ from the noisy samples and the time t. This residual guides the denoising process. In the case of Rectified Flow with ODE-based sampling, the reverse process can be formulated as:

$$\frac{d\mathbf{x}}{dt} = f(\mathbf{x}_t, t),$$

   where $f(\mathbf{x}_t, t)$ is the learned denoising function that predicts the residual (difference between noisy data and clean data):

$$f(\mathbf{x}_t, t) \approx \mathbf{x}_t - \mathbf{z}.$$

   As the model is trained to learn the difference between the data and the *unscaled* noise, it effectively learns a vector field with vectors pointing towards positions of high density (high probability).

3. **ODE-Based Sampling**: During inference, the model uses an Ordinary Differential Equation (ODE) solver (the trained U-Net) to integrate over time from $t = 1$ (pure noise) to $t = 0$ (clean

$\Delta t = \frac{1}{\text{num\_steps}}$ (the `step_size`). Note that, given the initial noise $\mathbf{z}$, this process is deterministic. This is in contrast to Denoising Diffusion Probabilistic Models (DDPMs) based on Stochastic Differential Equations (SDEs), where the noise is sampled at every step of the reverse diffusion process.

Noise Addition and Training Objective

The `RectifiedFlows` class handles the noise addition and defines the training loss:

- **Noise Addition (`add_noise` method)**:

```
def add_noise(self, x, noise, times):
    return (1. - times) * x + times * noise
```

`add_noise` performs a linear interpolation between clean data `x` and random noise `noise` based on the time variable `times`.

- **Time Variable Sampling**:

```
times = torch.nn.functional.sigmoid(
    torch.randn(x.shape[0]) * self.P_std + self.P_mean
)
```

The sigmoid non-linearity ensures `times` lies between 0 and 1.

- **Training Objective (`forward` method)**:

```
def forward(self, model, x, sigma=None, return_loss=True, **model_kwargs):
    # ...
    noises = torch.randn_like(x)
    v = x - noises
    noisy_samples = self.add_noise(x, noises, times)
    fv = model(noisy_samples, times, **model_kwargs)
    loss = mse(v, fv)
    # ...
```

The model calculates the residual by subtracting the noise from the data, expressed as $v = x - \text{noise}$. It then predicts fv, an approximation of the residual v, based on the noisy samples

Skip to main content

the true residual v and the predicted residual fv, which is given by $\text{Loss} = \|v - \text{fv}\|^2$. Predicting the residual instead of the noise or the data directly is a characteristic of the **Rectified Flow** method, while in **Denoising Diffusion Probabilistic Models (DDPMs)**, the model typically predicts the noise.

3. Inference and Sampling Process (`inference` function)

During inference, the model generates new samples by solving an ODE:

```
def inference(rectified_flows, net, latents_shape, num_steps):
    # Initialize with pure noise
    current_sample = torch.randn(latents_shape)
    times = torch.ones(latents_shape[0])
    step_size = 1 / num_steps
    # Integrate over time
    for i in range(num_steps):
        v = net(current_sample, times)
        current_sample = current_sample + step_size * v
        times = times - step_size
    return current_sample / sigma_data
```

- **Initialization**:
  - `current_sample`: Starts as pure noise.
  - `times`: Begins at $t = 1$, representing the highest noise level.
- **Integration Loop**:
  - **Time Step (`step_size`)**: Calculated as $\Delta t = \frac{1}{\text{num\_steps}}$.
  - **Euler's Method**: Updates the sample by moving in the direction of `v` predicted by the model:

    $$\text{current\_sample} = \text{current\_sample} + \Delta t \cdot v$$

  - **Time Update**: $t = t - \Delta t$
- **Result**: After integrating from $t = 1$ to $t = 0$, the `current_sample` approximates a sample from the data distribution.

Conclusion

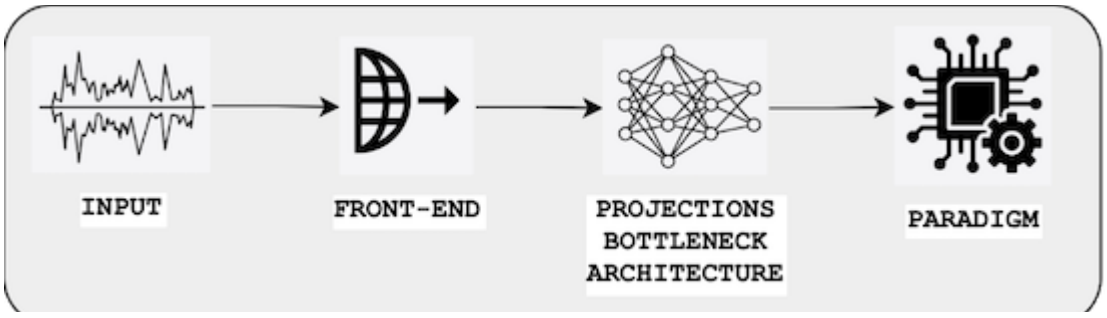
In this tutorial, we demonstrated a practical implementation of musical audio generation using Latent Diffusion with the guitar subset of the NSynth dataset and Music2Latent (M2L) representations. By encoding audio data into compressed latent vectors through M2L, we significantly reduced computational complexity.

We employed a diffusion model that adopts the **Rectified Flow** method with **ODE-based deterministic sampling**. This approach merges concepts from both denoising diffusion probabilistic models (DDPMs) and normalizing flows into a continuous-time generative modeling framework.

Specifically, the model:

- **Trains to predict residuals** between the data and noise, effectively learning a vector field pointing toward regions of high data density. This residual prediction guides the denoising process during generation.
- **Utilizes ODE-based deterministic sampling** during inference, integrating over time from pure noise ($t = 1$) to clean data ($t = 0$) without introducing additional randomness at each step. This results in a more efficient and stable generation process compared to stochastic methods.
- **Incorporates a time-conditioned U-Net architecture** that adapts its predictions based on the noise level at each time step. The time embedding allows the model to handle varying noise levels.

Inputs



Skip to main content

We denote by `input` the $\mathbf{X}$ fed to a neural network. We describe in the following the usual type of inputs in the case of audio.

Waveform

It is possible to use directly the audio waveform $\mathbf{x}_n$ as input to a model. In this case, the input is a 1-dimensional sequence over time. Such a system is often denoted by end-to-end (E2E). The first layer of the models then act as a learnable feature extractor. It is often either a 1D-convolution, a TCN or a parametric front-end such as SincNet.

More details can be found in the following "Waveform-based music processing with deep learning" by Jongpil Lee, Jordi Pons, Sander Dieleman ISMIR-2019 tutorial.

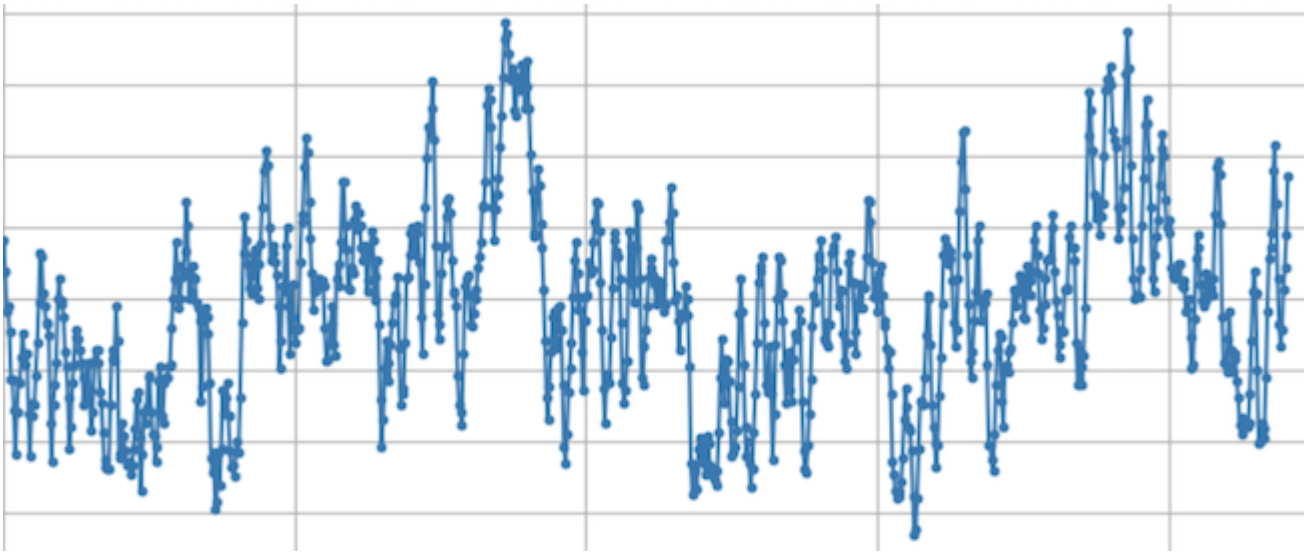


Figure *Waveform of an audio signal with sample values.*

Log-Mel-Spectrogram (LMS)

The spectrogram $|\mathbf{X}_{k,m}|$ is the magnitude of the Short Time Fourier Transform of the signal $\mathbf{x}_n$ with

- discrete frequencies k and
- time frames m

Its frequencies can be converted to the Mel perceptual scale. This allows

- to reduce the dimensionality of the data
- to mimic the decomposition of the frequencies performed by the cochlea into critical-bands
- to allows performing (some) invariance over small pitch modifications (
    - hence LMS are invariant to the pitch and only represent the so-called timbre.

Its amplitude can be concerted to the Log-scale. This allows

- to map the recording level of the audio to a constant: $\alpha \mathbf{x}_n \rightarrow \log(\alpha) + \log(\mathbf{X}_{k,m})$
- to mimic the compression of the amplitude performed by the inner-cell of the cochlea
- to change the distribution of the input

Usually, a $\log(1 + Cx)$ (with $C = 10.000$) is used instead of a $\log(x)$ to avoid singularity in $x = 0$.

Another explanation of the LMS, is to consider that those are equivalent to the MFCC but without the last DCT performed for the MFCC. Indeed this DCT was necessary to decorrelate the dimensions and then allows covariance matrix in GMM-based systems but is not necessary for deep learning models.

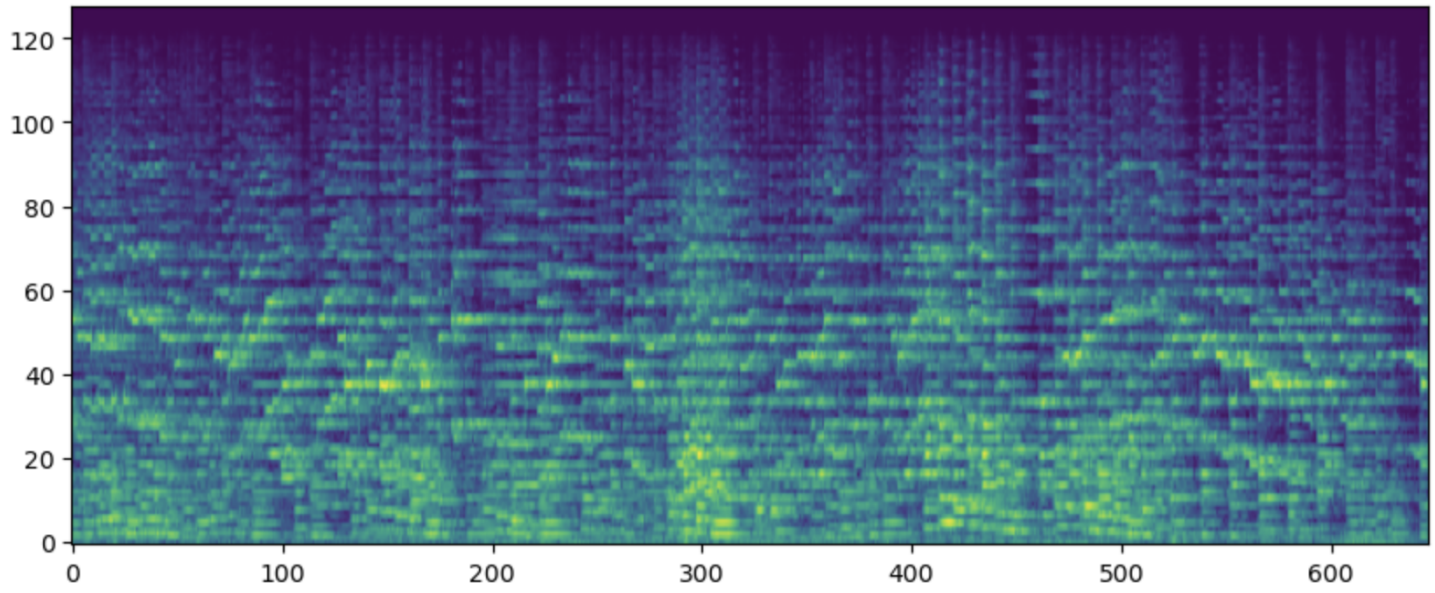


Figure *Log-Mel-Spectrogram of an audio signal with 128 Mel bands.*

```
def f_get_lms(audio_v, sr_hz, param_lms):
    """
    description:
        compute Log-Mel-Sepctrogram audio features
    inputs:
```

Skip to main content

```
        - sr_hz
    outputs:
        - data_m (nb_dim, nb_frame): Log-Mel-Spectrogram matrix
        - time_sec_v (nb_frame): corresponding time [in sec] of analysis windows
    """
    # --- data (nb_dim, nb_frames)
    mel_data_m = librosa.feature.melspectrogram(y=audio_v, sr=sr_hz,
                                                n_mels=param_lms.nb_band,
                                                win_length=param_lms.L_n,
                                                hop_length=param_lms.STEP_n)
    data_m = f_log(mel_data_m)
    nb_frame = data_m.shape[1]
    time_sec_v = librosa.frames_to_time(frames=np.arange(nb_frame),
                                        sr=sr_hz,
                                        hop_length=param_lms.STEP_n)

    return data_m, time_sec_v
```

Constant-Q-Transform (CQT)

Constant-Q-Transform was proposed in [Bro91].

- The CQT divides the frequency axis into bins where the ratio between adjacent frequencies is constant (i.e., logarithmically spaced): $r = f_{k+1}/f_k = cst$
  - This is different from the Discrete Fourier Transform (DFT), where frequency bins are linearly spaced $f_{k+1} - f_k = cst$.
  - In music $r = 2^{1/12}$ for adjacent musical pitches (semitones).
  - It is possible to increase the number of bins for each semitone (if 5 bins per semitone the ratio is $r = 2^{1/(5 \cdot 12)}$)
- The CQT adapts the duration of the analysis window $\mathbf{w}_k$ for each frequency f_k, in order to be able to spectrally separate adjacent frequencies f_k, f_{k+1}.
  - This is different from the DFT, which uses a single analysis window $\mathbf{w}$ (hence a single duration) for all its frequencies f_k (hence with a fixed spectral resolution),

Because of the frequencies are expressed in log-scale in the CQT, pitch-shifting of a musical instrument correspond to a vertical translation of the corresponding CQT ($\alpha f \rightarrow \log(\alpha) + \log(f)$. This property has been used in some works such as Shift-Invariant PLCA or in 2D-Convolution.

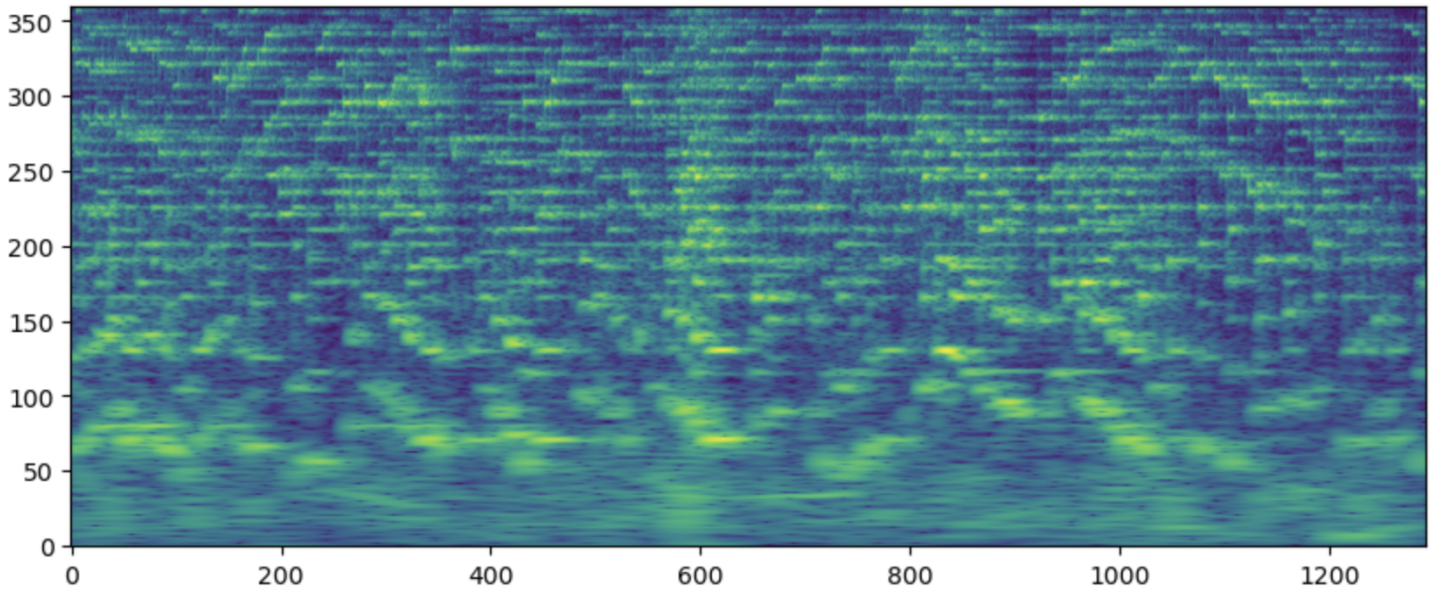


Figure *Constant-Q-Transform of an audio signal with 6 octaves and 5 bins per semi-tone (* $6 \times 12 \times 5$ *frequencies).*

Harmonic-CQT (HCQT)

The Harmonic-CQT has been proposed in [BMS+17].

- The usual "local correlation of the pixels" assumption underlying 2D-Convolution does not hold for musical sound. This is because the harmonics hf_0 of a given sound are spread over the whole spectrum, hence the content at frequencies f_k are not correlated.
- To allow highlighing this correlation of harmonics, the Harmonic-CQT represent the harmonics of a frequency f_k in a new depth/channel dimension
  - In this, a channel h represent the frequencies $f_k^{(h)} = hf_k$
    - Using channels $h \in \{1, 2, 3, 4, 5\}$ allows to represent in the depth the first five harmonics of a potentially pitched sound at $f_0 = f_k$: $f_k^{(1)} = f_k, f_k^{(2)} = 2f_k, f_k^{(3)} = 3f_k, \dots$
  - To compute it, we use CQTs with different starting frequencies hf_{min}
  - The HCQT is obtained by stacking the various downsampled CQTs in depth/channel dimension
  - The resulting Harmonic-CQT is a 3D-tensor of dimensions (harmonic, time, frequencies).

Skip to main content

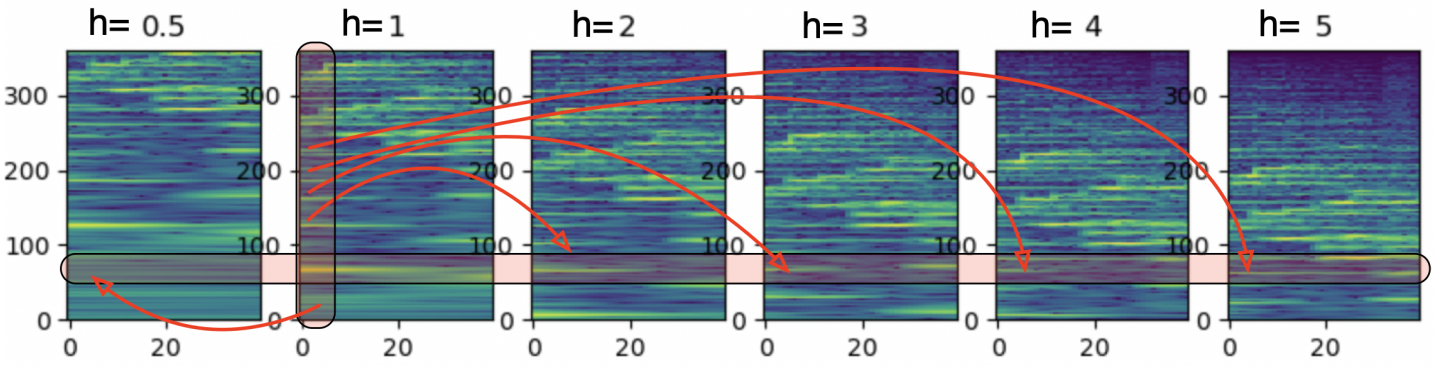


Figure *Set of CQT of the same audio signal with various starting frequencies f_{min}. The default CQT is referred as "h=1". The red vertical stripe highlight the fundamental frequency of a sound. In $h = 1$, the stripe highlights f_0. If we downsample the CQT by a factor 2 (indicated in "h=2"), the stripe now highlight $2f_0$. If we downsample the CQT by a factor 3 (indicated in "h=3"), the stripe now highlight $3f_0$. The various harmonics hf_0 are now aligned vertically across downsampled versions.*

The HCQT is often used as input to a 2D-Convolution layer with small kernels (5×5) which extend over the whole depth of the HCQT. When used for Multi-Pitch-Estimation, the kernels should therefore learn the specific relationship among harmonics specific to harmonics versus non-harmonics. An extra component $h = 0.5$ is added to avoid octave errors.

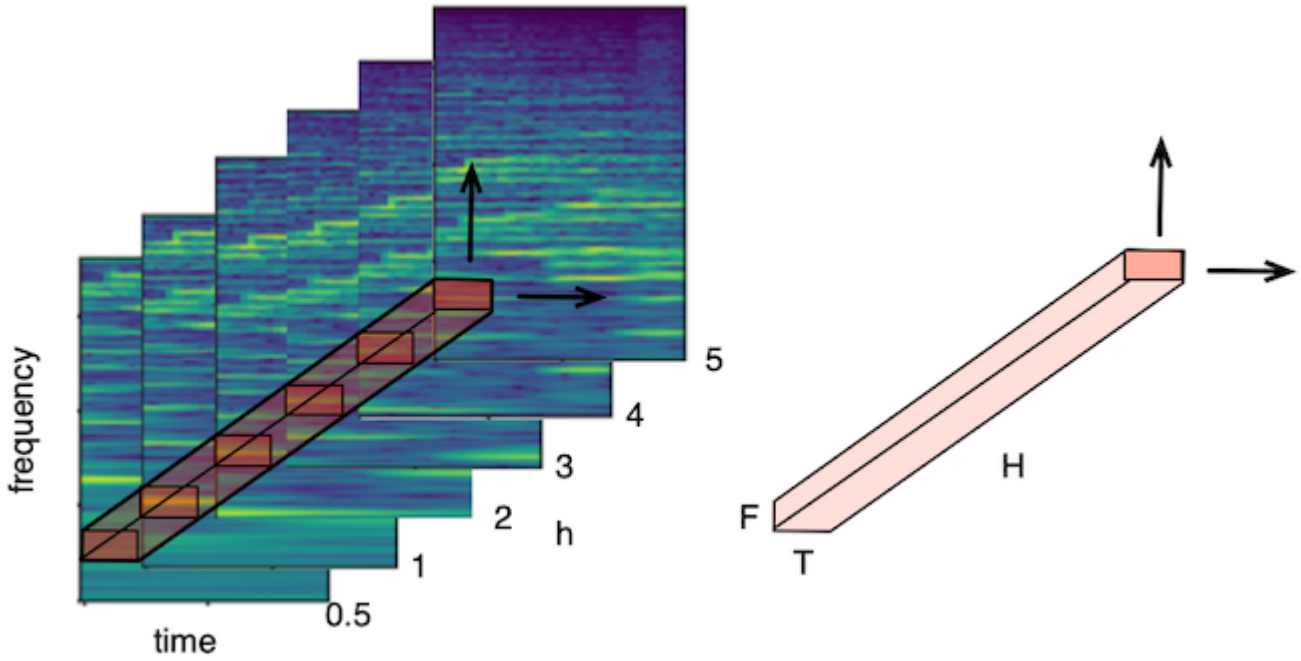


Figure *2D-Convolution (over time and frequencies) of the HCQT with (5×5) kernels which extend over the whole depth.*

```
    description:
        compute Harmonic CQT
    inputs:
        - audio_v
        - sr_hz
    outputs:
        - data_3m (H, nb_dim, nb_frame): Harmonic CQT
        - time_sec_v (nb_frame): corresponding time [in sec] of analysis windows
        - frequency_hz_v (nb_dim): corresponding frequency [in Hz] of CQT channels
    """
    for idx, h in enumerate(param_hcqt.h_l):
        A_m = np.abs(librosa.cqt(y=audio_v, sr=sr_hz,
                                 fmin=h*param_hcqt.FMIN,
                                 hop_length=param_hcqt.HOP_LENGTH,
                                 bins_per_octave=param_hcqt.BINS_PER_OCTAVE,
                                 n_bins=param_hcqt.N_BINS))
        if idx==0:
            data_3m = np.zeros((len(param_hcqt.h_l), A_m.shape[0], A_m.shape[1]))
        data_3m[idx,:,:] = A_m

    n_times = data_3m.shape[2]
    time_sec_v = librosa.frames_to_time(np.arange(n_times),
                                        sr=sr_hz,
                                        hop_length=param_hcqt.HOP_LENGTH)
    frequency_hz_v = librosa.cqt_frequencies(n_bins=param_hcqt.N_BINS,
                                             fmin=param_hcqt.FMIN,
                                             bins_per_octave=param_hcqt.BINS_

    return data_3m, time_sec_v, frequency_hz_v
```

Chroma/ Pitch-Class-Profile

Chroma (or Pitch-Class-Profile) [Fuj99] [Wak99] is a compact (12-dimensions) representation of the harmonic content over time of a music track.

- Its dimensions correspond to the pitch-classes (hence independently of their octave): C, C#, D, D#, E, ...
- Chroma can be obtained by mapping the content of the spectrogram (or the CQT) to the pitch-classes (summing the content of all frequency bands corresponding to the C0, C1, C2, ... to obtain the Chroma C, ...).

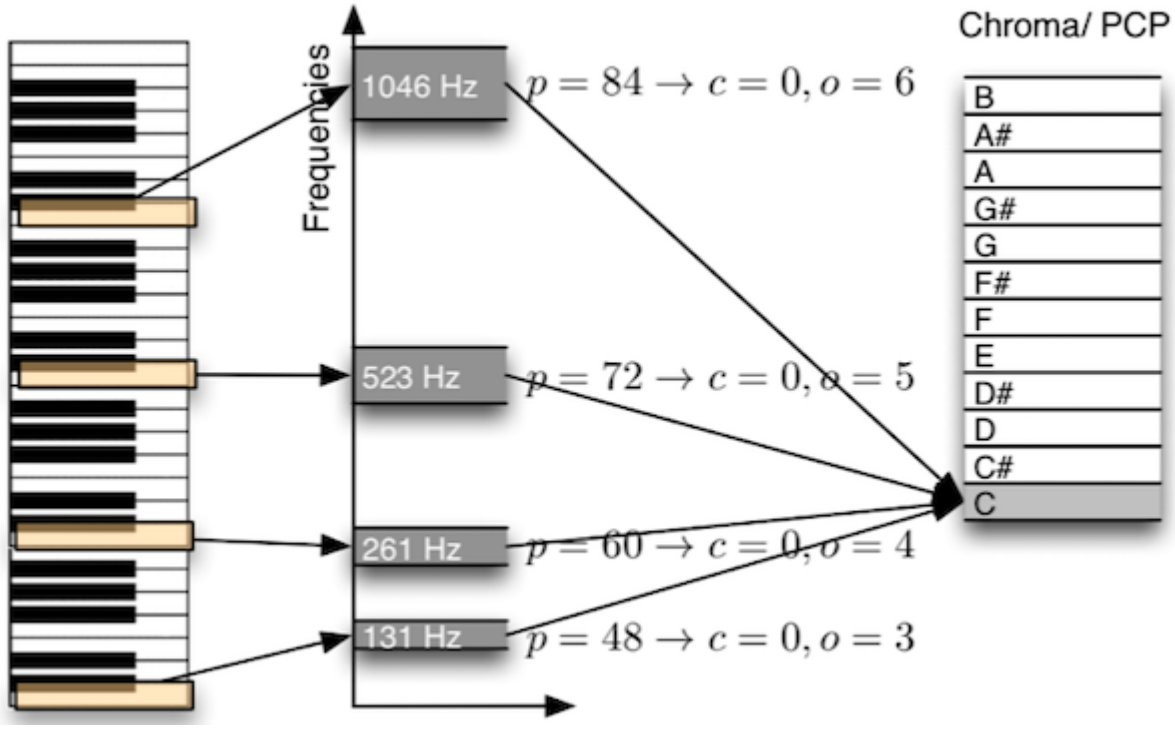


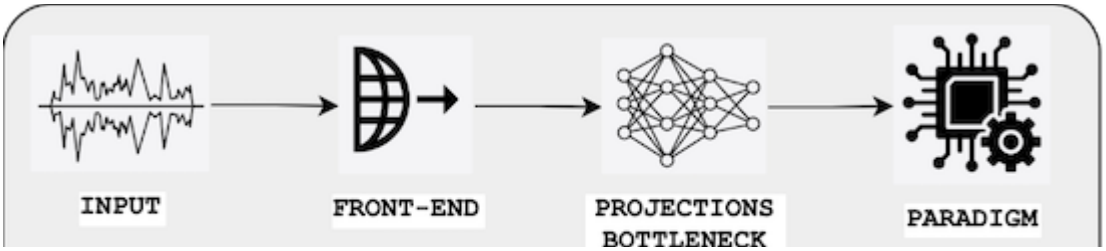
Figure *Chroma map the content of the DFT frequencies to the 12 semi-tones pitch-classes*

```
librosa.feature.chroma_stft(*, y=None, sr=22050, S=None, norm=inf, n_fft=2048, hop_
librosa.feature.chroma_cqt(*, y=None, sr=22050, C=None, hop_length=512, fmin=None,
```

Since the direct mapping from spectram/CQT suffers from artifacts (fifth harmonics, noise, percussive instruments), it has been proposed to learn a cleaner chroma representation using deep learning models, the so-called deep-chroma [KW16] [MB17] [WP21].

Chroma are often as input for applications such as Automatic-Chord-Recogniton (ACR), key-detection or Cover-Song-Identification (CSI). We use here for CSI the deep-chroma of [MB17] named *crema-PCP*.

Front-ends



Skip to main content

We denote by `front-end` the first projections of a neural network directly applied to the input $\mathbf{X}$. Those therefore depends on the type of the input.

Conv-1D

- Dieleman et al. [DS14] were probably among the first to attempt replacing the spectrogram input by a learnable front-end, here 1D-convolution.
  - They replicated the spectrogram parameters (window length, hop size) in the 1D-convolution parameters (kernel-length and stride).
- A major difference between the spectrogram (magnitude of the STFT) and learnable 1D-kernels, is the phase-shift invariance provided by the former (to perform this with the later will necessitates a kernel for all possible phase-shift of a given frequency).
  - To facilitate the learning of such kernels, smaller kernel (hence with less possibilities for phase-shifts) has been proposed, such as in Sample-CNN (a cascade, as in VGG-net, or small 1D filters) [LPKN17].

1D-Convolution is very popular for source separation front-ends: as in Wav-U-Net [SED18], ConvTasNet [LM19] or Demucs [DefossezUBB19].

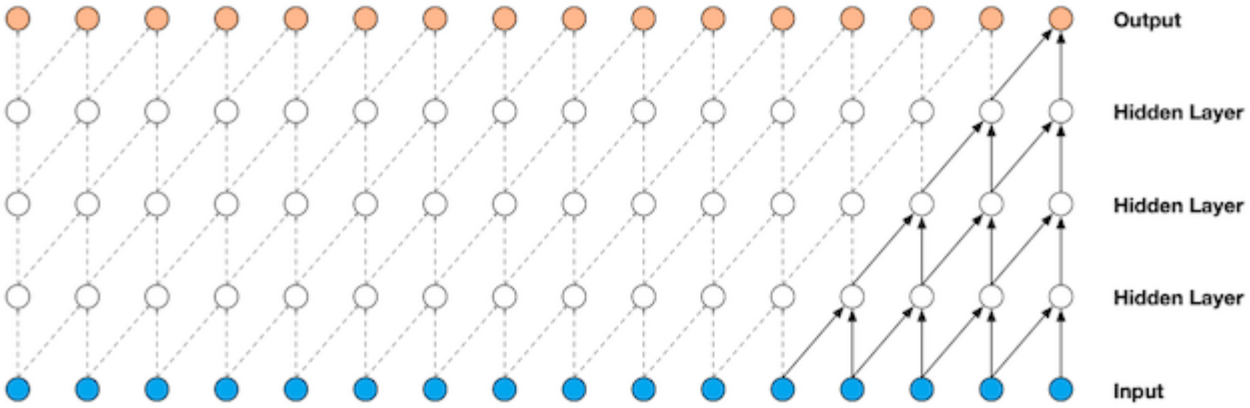


Figure *1D-Convolution; image source: WaveNet [vdODZ+16]*

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilati
```

Dilated-Conv-1D

The 1D-dilated convolution was proposed in the paper WaveNet was proposed in [vdODZ+16]. The

Skip to main content

- Indeed, since audio signals has a large dimensionality (16.000 values for 1 second of audio at a sampling rate of 16Khz), one would need a very large kernel, or a very large number of layers in order the receptive field to capture the whole signal.
- Dilated convolution consists in skipping 1 sample over 2 (over 4, over 8, ...) when computing convolution (or equivalently adding holes in the kernel).

For a 1D-filter w of size l and a sequence x_n,

- the usual convolution is written $(x \circledast w)_n = \sum_{i=0}^{l-1} w_i x_{n-i}$
- the dilated convolution with a dilatation factor d is written $(x \circledast_d w)_n = \sum_{i=0}^{l-1} w_i x_{n-(d \times i)}$,
  - the filter is convolved with the signal only considering one over d values.

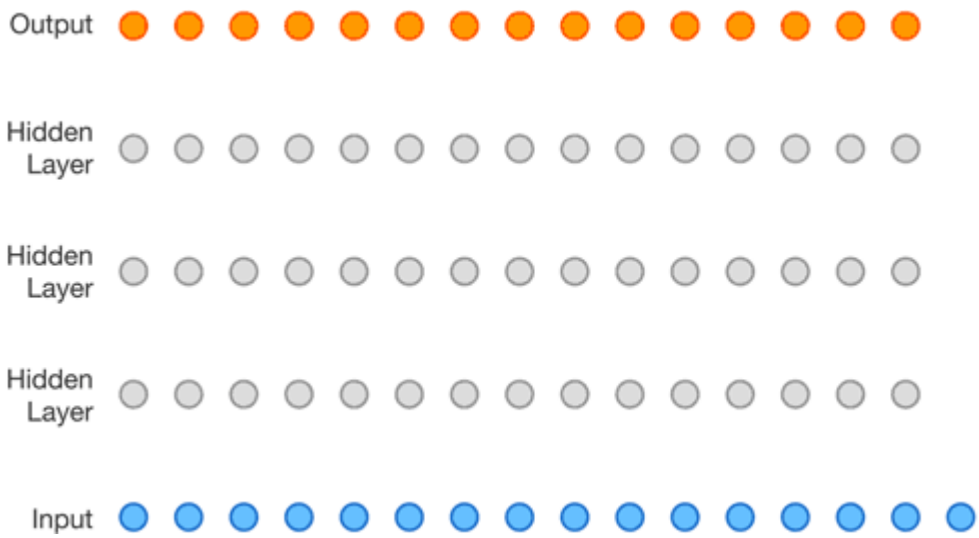


Figure *Dilated 1D-Convolution: the first projection uses $d = 1$, the second $d = 2$, the third $d = 4$. Note that the stride remains equal to 1 so that all values over time and layers are processes; image source: WaveNet [vdODZ+16]*

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilati
```

Temporal Convolution Network (TCN)

Temporal Convolution Network was proposed in [BKK18]. It is a simplification of the computational block underlying WaveNet. It is made of

- Two blocks of

Skip to main content

- Weight normalization
- ReLU
- Dropout

- A residual/skip connnection

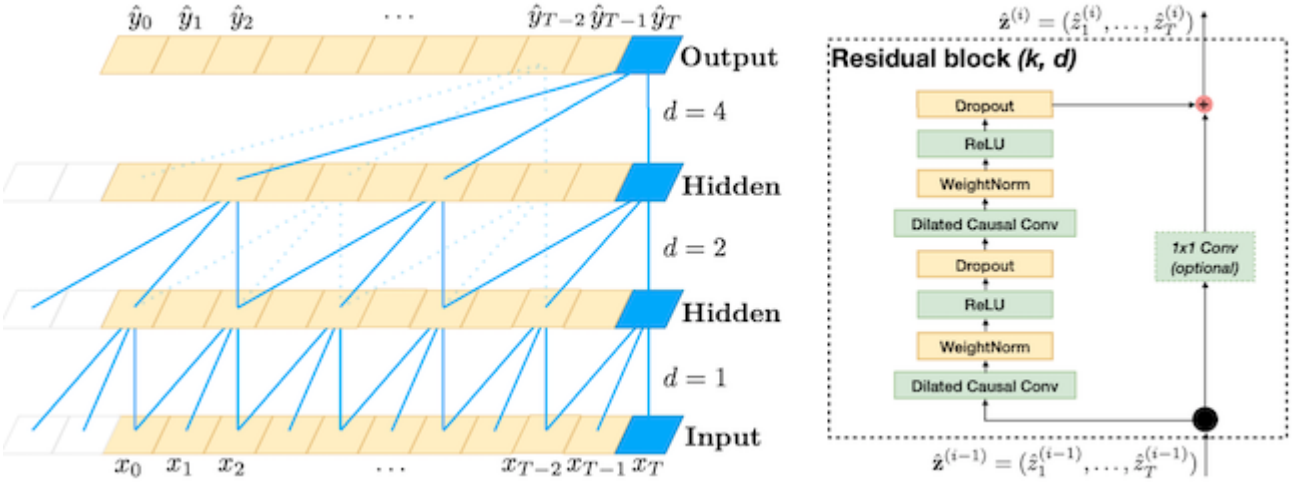


Figure *Temporal Convolution Network Block; image source: [BKK18]*

```
# TCN code: https://github.com/locuslab/TCN
class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size
    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class TemporalBlock(nn.Module):
    def __init__(self, n_inputs, n_outputs, kernel_size, stride, dilation, padding,
        super(TemporalBlock, self).__init__()
        self.conv1 = weight_norm(nn.Conv1d(n_inputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=
        self.chomp1 = Chomp1d(padding)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(dropout)

        self.conv2 = weight_norm(nn.Conv1d(n_outputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=
        self.chomp2 = Chomp1d(padding)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)

        self.net = nn.Sequential(self.conv1, self.chomp1, self.relu1, self.dropout1
                                 self.conv2, self.chomp2, self.relu2, self.dropout2
        self.downsample = nn.Conv1d(n_inputs, n_outputs, 1) if n_inputs != n_output
```

Skip to main content

```
    def init_weights(self):
        self.conv1.weight.data.normal_(0, 0.01)
        self.conv2.weight.data.normal_(0, 0.01)
        if self.downsample is not None:
            self.downsample.weight.data.normal_(0, 0.01)

    def forward(self, x):
        out = self.net(x)
        res = x if self.downsample is None else self.downsample(x)
        return self.relu(out + res)


class TemporalConvNet(nn.Module):
    def __init__(self, num_inputs, num_channels, kernel_size=2, dropout=0.2):
        super(TemporalConvNet, self).__init__()
        layers = []
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i-1]
            out_channels = num_channels[i]
            layers += [TemporalBlock(in_channels, out_channels, kernel_size, stride=
                                     padding=(kernel_size-1) * dilation_size, dropou

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

Parametric front-end: SincNet

SincNet was proposed in [RB18]. It is one of the first **parametric front-end**.

Parametric kernel:

- SincNet defines the value of a 1D-kernel w_n (to be used for 1D-convolution) as the results of a parametric function evaluated at the points $n \in \{0, \dots, N-1\}$: $w_n = f_\theta(n)$
  - where θ are the parameters of the function, to be trained

Training:

- Training a normal 1D-convolution kernel of length $N \Rightarrow$ one has to learn each of the N filter values.

Skip to main content

SincNet

- aims at designing kernels which frequency response is a band-pass filter $[f_1, f_2]$.
- band-pass filters $[f_1, f_2]$ = subtracting two low-pass filters at frequency f_1 and f_1,
- low-pass filters are expressed as SinC function in time: $SinC(x) = \frac{\sin(x)}{x}$

$$w_n^{f_1,f_2} = 2f_2 sinc(2\pi f_2 n) - 2f_1 sinc(2\pi f_1 n)$$

Only two parameters to be trained: $\theta = \{f_1, f_2\}$. How to train ? we can compute the derivative of the Loss w.r.t. f_1 and f_2, we can then optimize ...

SincNet is the first of a series of front-ends which rely on differentiable implementation of signal processing:

- Analytic filters [PCDV20],
- Complex Gabor (CG-CNN) [NoePM20], or
- LEAF (Learnable Audio Front-End) [ZTdCQT21].
- It can be considered as the early stages of what would lead to the well-known DDSP [EHGR20].

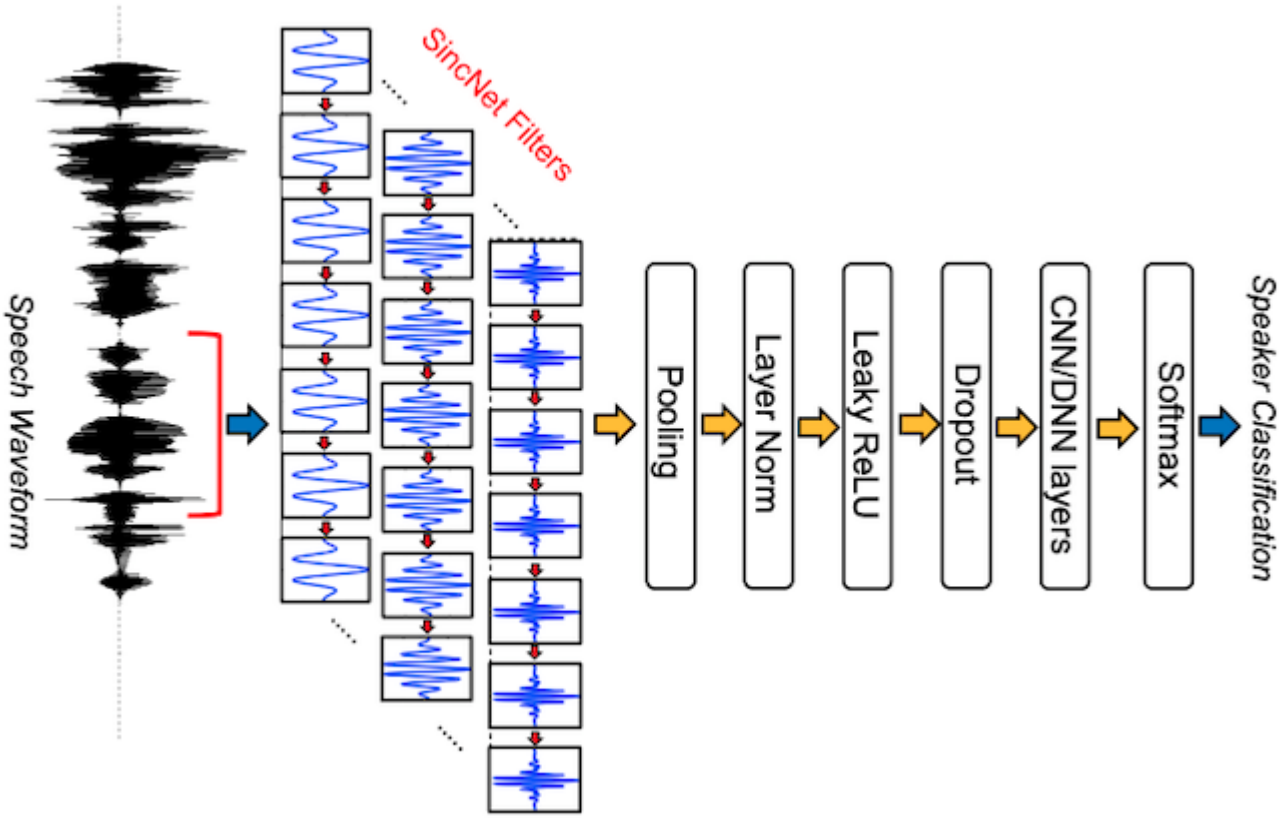


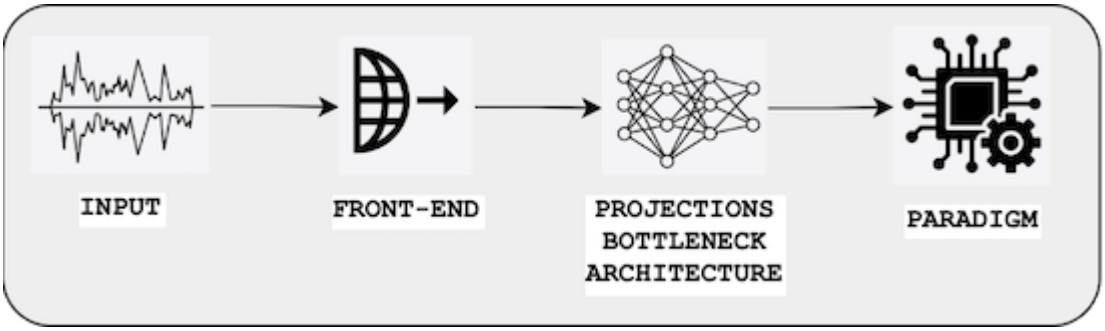
Figure *SincNet architcture; image source: [RB18]*

```
class SincConv_fast(nn.Module):
    """Sinc-based convolution
    Parameters
    ----------
    in_channels : `int`
        Number of input channels. Must be 1.
    out_channels : `int`
        Number of filters.
    kernel_size : `int`
        Filter length.
    sample_rate : `int`, optional
        Sample rate. Defaults to 16000.
    Usage
    -----
    See `torch.nn.Conv1d`
    Reference
    ---------
    Mirco Ravanelli, Yoshua Bengio,
    "Speaker Recognition from raw waveform with SincNet".
    https://arxiv.org/abs/1808.00158
    """

    @staticmethod
    def to_mel(hz):
```

```
    @staticmethod
    def to_hz(mel):
        return 700 * (10 ** (mel / 2595) - 1)

    def __init__(self, out_channels, kernel_size, sample_rate=16000, in_channels=1,
                 stride=1, padding=0, dilation=1, bias=False, groups=1, min_low_hz=

        super(SincConv_fast,self).__init__()

        if in_channels != 1:
            #msg = (f'SincConv only support one input channel '
            #       f'(here, in_channels = {in_channels:d}).')
            msg = "SincConv only support one input channel (here, in_channels = {%i
            raise ValueError(msg)

        self.out_channels = out_channels
        self.kernel_size = kernel_size

        # Forcing the filters to be odd (i.e, perfectly symmetrics)
        if kernel_size%2==0: self.kernel_size=self.kernel_size+1

        self.stride = stride
        self.padding = padding
        self.dilation = dilation

        if bias: raise ValueError('SincConv does not support bias.')
        if groups > 1: raise ValueError('SincConv does not support groups.')

        self.sample_rate = sample_rate
        self.min_low_hz = min_low_hz
        self.min_band_hz = min_band_hz

        # initialize filterbanks such that they are equally spaced in Mel scale
        low_hz = 30
        high_hz = self.sample_rate / 2 - (self.min_low_hz + self.min_band_hz)
        mel_v = np.linspace(self.to_mel(low_hz), self.to_mel(high_hz), self.out_cha
        hz_v = self.to_hz(mel_v)


        # filter lower frequency (out_channels, 1)
        self.low_hz_v_ = nn.Parameter(torch.Tensor(hz_v[:-1]).view(-1, 1))

        # filter frequency band (out_channels, 1)
        self.band_hz_v_ = nn.Parameter(torch.Tensor(np.diff(hz_v)).view(-1, 1))

        # Hamming window
        #self.window_ = torch.hamming_window(self.kernel_size)
        n_lin = torch.linspace(0, (self.kernel_size/2)-1, steps=int((self.kernel_si
        self.window_ = 0.54-0.46*torch.cos(2*np.pi*n_lin/self.kernel_size);


        # (1, kernel_size/2)
```

```
    def forward(self, waveforms):
        """
        Parameters
        ----------
        waveforms : `torch.Tensor` (batch_size, 1, n_samples)
            Batch of waveforms.
        Returns
        -------
        features : `torch.Tensor` (batch_size, out_channels, n_samples_out)
            Batch of sinc filters activations.
        """

        self.n_ = self.n_.to(waveforms.device)
        self.window_ = self.window_.to(waveforms.device)

        low_v = self.min_low_hz  + torch.abs(self.low_hz_v_)
        high_v = torch.clamp(low_v + self.min_band_hz + torch.abs(self.band_hz_v_),
                             self.min_low_hz,
                             self.sample_rate/2)
        band_v = (high_v - low_v)[:,0]

        f_times_t_low = torch.matmul(low_v, self.n_)
        f_times_t_high = torch.matmul(high_v, self.n_)

        band_pass_left = ((torch.sin(f_times_t_high) - torch.sin(f_times_t_low)) /
        band_pass_center = 2 * band_v.view(-1,1)
        band_pass_right= torch.flip(band_pass_left, dims=[1])

        band_pass=torch.cat([band_pass_left,
                             band_pass_center,
                             band_pass_right],dim=1)

        band_pass = band_pass / (2*band_v[:,None])

        self.filters = (band_pass).view(self.out_channels, 1, self.kernel_size)

        return F.conv1d(waveforms, self.filters, stride=self.stride, padding=self.pa


model = SincConv_fast(out_channels=80, kernel_size=251, sample_rate=16000, in_channe
X = torch.randn(2, 1, 16000)
model(X);
filter_m = model.filters[:,0,:].detach().numpy()
fft_filter_m = np.abs(np.fft.rfft(filter_m, 4096))

plt.figure(figsize=(14,4)); plt.plot(fft_filter_m.T);
```

Projections



We denote by `projection` the type of operations performed within a neural network. Those sometimes overlap with the ones used in the front-end.

Fully-Connected

The Fully-Connected (FC) projection is the core projection of Multi-Layer-Perceptron (MLP) or Feed-Forward (FF) neural network.

- MLP or FF is a sandwich of two FC with a non-linearity in between
- FC are generally used as output of classification network, used within RNN/LSTM or Transformer.
- When applied to all spatial (temporal) elements, it can be replaced by a Convolution with a (1x1) kernel.

```
torch.nn.Lineartorch.nn.Linear(in_features, out_features)
```

Conv-1D

see link

Skip to main content

Conv-2D

The Conv-2D is the operator underlying all ConvNet archiectures (often combined with a Max-Pooling operator). It implements the following convolution with a kernel W:

$$(X \circledast W)_{x,y} = \sum_{i,j} W_{i,j} X_{x-i,y-j}$$

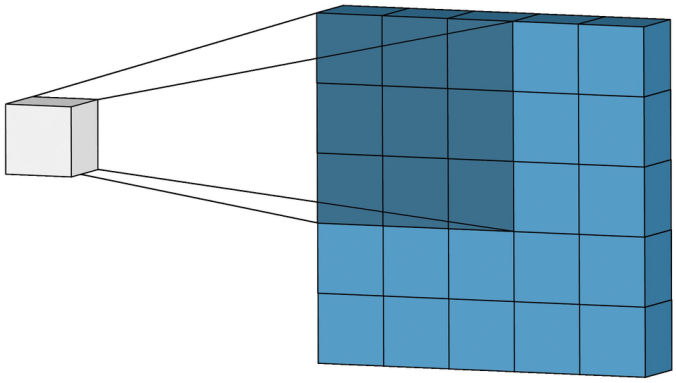


Figure *Animated illustration of 2D-convolution; image source Link*

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilati
```

Depthwise Separable Convolution

Depthwise Separable Convolution (DSC) was proposed in [Cho17] in the continuation of the Inception network.

- It is the concatenation of
  - a Depthwise Convolution (channel-wise convolution)
  - a Pointwise (1x1) convolution.

Skip to main content

- A normal convolution uses C_{out} kernels each of shape $(C_{in}, 3, 3)$.
  - It necessitates $C_{out} \times (C_{in} \times 3 \times 3)$ weights.
- A DSC uses C_{in} kernels each of shape $(3, 3)$, then C_{out} kernels of shape $(C_{in}, 1, 1)$
  - A DSC will only necessitates $C_{in} \times (3 \times 3)$ and $C_{in} \times C_{out}$ weights. The number of parameters to be learnt and of multiplications to be performed is largely reduced.

Because of this, it is largely used in model for IoT such as MobileNet [HZC+17].

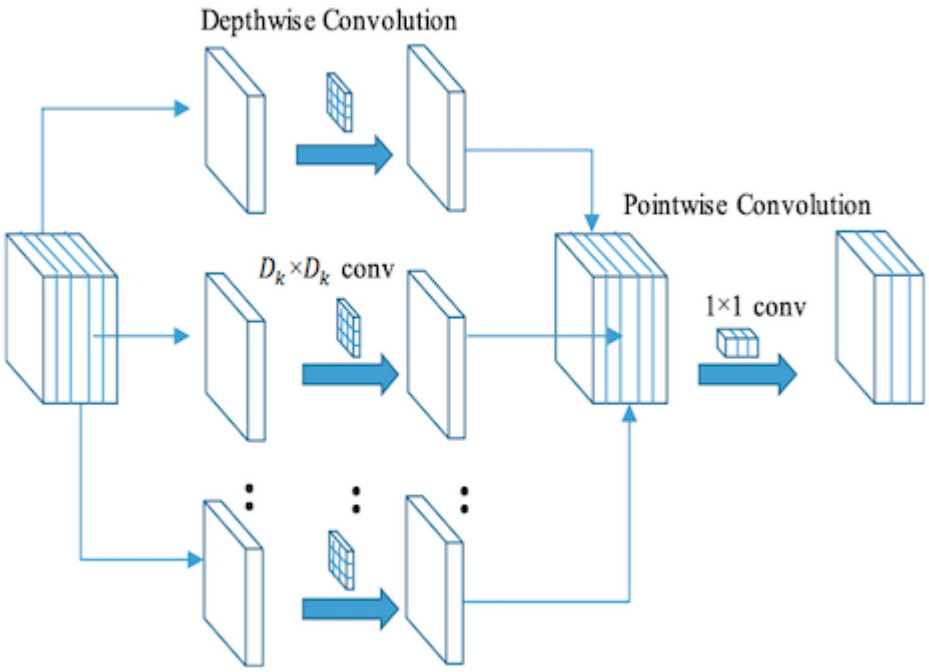


Figure *Depthwise Separable Convolution; image source: Link*

```
# Code: https://github.com/seungjunlee96/Depthwise-Separable-Convolution_Pytorch/bl
class depthwise_separable_conv(nn.Module):
    def __init__(self, nin, kernels_per_layer, nout, kernel_size=3, padding=1):
        super(depthwise_separable_conv, self).__init__()
        self.depthwise = nn.Conv2d(nin, nin * kernels_per_layer, kernel_size=kernel
        self.pointwise = nn.Conv2d(nin * kernels_per_layer, nout, kernel_size=1)
    def forward(self, x):
        out = self.depthwise(x)
        out = self.pointwise(out)
        return out

model = depthwise_separable_conv(16, 1, 32)
X = torch.randn(2,16,23,23)
model(X).size()
```

Skip to main content

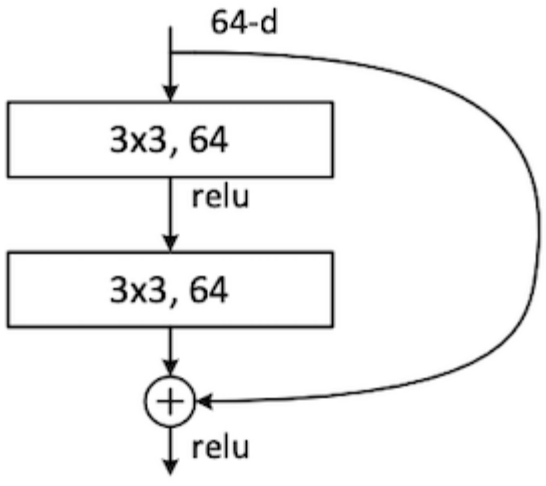
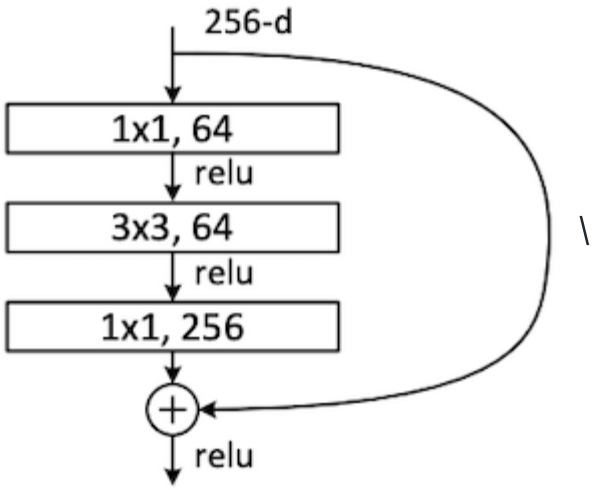
ResNet

ResNet has been proposed by [HZRS16] in the framework of image recognition. A ResNet is made of a

- large number of blocks each containing a residual connection (skip-connection).
- the residual connection allows to
  - bypass blocks during forward, and
  - backward easily the gradients during training hence allows constructing very deep models (152 in the original papers).

We are interested here in the two building blocks of ResNet:

- the **building block**:
  It is a stack of - a first 2D-Convolution, - a ReLU, - a second 2D-Convolution, - the residual connection ($\mathcal{F}(x) + x$), - a ReLU
- the **"bottleneck" block**:
  It is a stack of 3 layers instead of 2.
  The three layers are 1×1, 3×3, and 1×1 convolutions, where
  - the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions,
  - the 3×3 layer is operates in a smaller input/output dimensions.

ResNet "building" block	ResNet "bottleneck" block
	

\

Skip to main content

```
# https://www.digitalocean.com/community/tutorials/writing-resnet-from-scratch-in-py
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size = 3, stride = 1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, out_channels, kernel_size = kernel_s
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, kernel_size = kernel_
                        nn.BatchNorm2d(out_channels))
        self.downsample = False

        if in_channels != out_channels:
            self.downsample = True
            self.conv_ds = nn.Conv2d(in_channels, out_channels, kernel_size = 1, st
        self.relu = nn.ReLU()
        self.out_channels = out_channels

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.conv2(out)
        if self.downsample: residual = self.conv_ds(x)
        out += residual
        out = self.relu(out)
        return out
```

```
# https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py
class Bottleneck(nn.Module):
    def __init__(self, inplanes, planes, stride, downsample, groups, base_width, di
        super().__init__()
        if norm_layer is None: norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.0)) * groups
        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
```

```
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

ConvNeXt

ConvNeXT has been proposed in "A ConvNet for the 2020s" [LMW+22] with the goal of modernizing ResNet architecture and to remains competitive with Vision Transformers (ViTs).

It especially bases its design on the Swin Transformers:

- larger kernel size (kernels are (7×7)),
  - as in Transfomer (Transfomer has a non-local self-attention, which enables each layer to have a global receptive field)
- ResNeXt-ify
  - instead of grouped convolution use dephtwise convolution
- inverted bottleneck (from 96 channels to 384 then back to 96),
  - as in Transformer (the hidden dimension of the MLP block is four times wider than the input dimension)
- various layer-wise micro designs (use of layer normalization)
- Fewer activation functions
  - Replacing ReLU with GELU (Gaussian Error Linear Unit)
  - as in Transformer which has fewer activation functions (only one activation function present in the MLP block)

In [LMW+22], it has been shown to achieve better performances than Transformer-based architecture.

Skip to main content

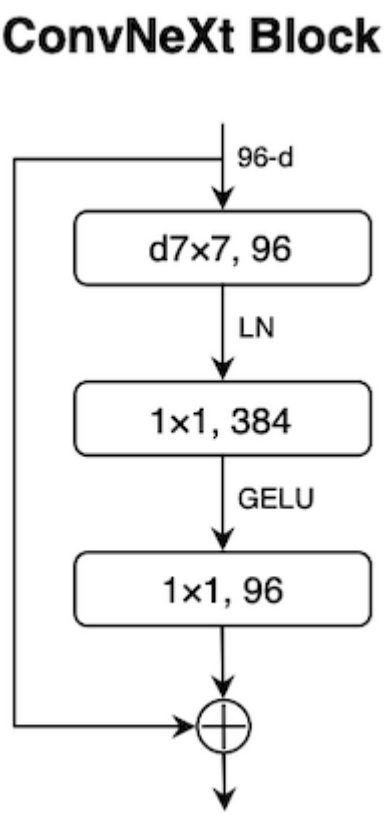
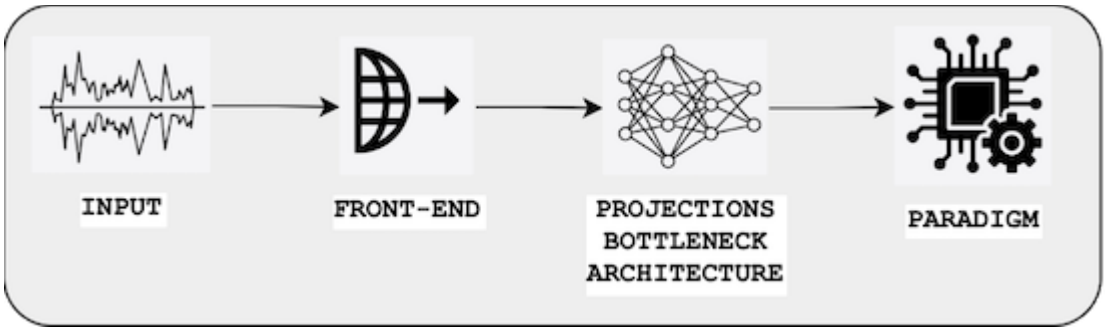


Figure *ConvNeXt block; image source: Link*

```
# ConvNeXt CODE: https://github.com/facebookresearch/ConvNeXt/blob/main/models/convn
class ConvNeXtBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=7, drop_path=0.0):
        super(ConvNeXtBlock, self).__init__()

        # 1. Depthwise convolution (spatial convolution with large kernel)
        self.dwconv = nn.Conv2d(in_channels, in_channels, kernel_size=kernel_size, p

        # 2. Layer normalization applied across channels
        self.norm = nn.LayerNorm(in_channels, eps=1e-6)  # LayerNorm is applied afte

        # 3. Pointwise convolution to project to higher dimensions (expanding and co
        self.pwconv1 = nn.Linear(in_channels, 4 * in_channels)  # expand channels by
        self.act = nn.GELU()  # GELU activation
        self.pwconv2 = nn.Linear(4 * in_channels, out_channels)  # project back to o

        # 4. Stochastic depth (optional) for better regularization
        self.drop_path = nn.Identity() if drop_path == 0 else StochasticDepth(drop_p

    def forward(self, x):
        # Input: (B, C, H, W)
        residual = x

        # 1. Depthwise convolution
        x = self.dwconv(x)
```

```
        x = x.permute(0, 2, 3, 1)  # (B, C, H, W) -> (B, H, W, C)
        x = self.norm(x)

        # 3. Pointwise convolutions + GELU
        x = self.pwconv1(x)
        x = self.act(x)
        x = self.pwconv2(x)

        # 4. Drop path (if applicable) and residual connection
        x = x.permute(0, 3, 1, 2)  # (B, H, W, C) -> (B, C, H, W)
        x = self.drop_path(x) + residual  # Add residual connection

        return x

class StochasticDepth(nn.Module):
    """Drop paths (stochastic depth) per sample (when applied in the main path of r
    ...
```

Bottleneck



We denote by `bottleneck` the type of representation $\mathbf{z}$ (and paradigm used) in an auto-encoder $\hat{\mathbf{x}} = g_\phi(\mathbf{z}), \mathbf{z} = f_\theta(\mathbf{x})$: *such as continuous deterministic, continuous probabilist, quantized, residual*.

Auto-encoder

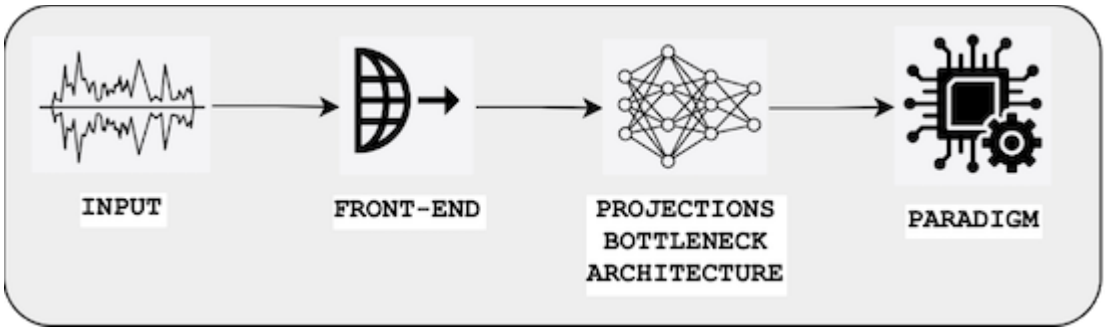
Variational auto-encoder (VAE)

Vector Quantised-Variational AutoEncoder (VQ-VAE)

Residual Vector Quantizers (RVQ)

Soundstream

Architectures



We denote by `architecture` the overall design of a neural network, i.e. the way front-end and projections are used together.

U-Net

The U-Net was proposed in [RFB15] in the framework of biomedical image segmentation and made popular in MIR by [JHM+17] for singing voice separation.

The U-Net is an auto-encoder with skip-connections.

Skip to main content

- The **decoder** (right part) upsample the spatial dimensions and decrease the depth.

Skip connections are added between equivalent layers of the encoder and decoder. Their goals are:

- to bring back details of the original images to the decoder
  (the bottleneck being to compressed to represent detailed information)
- to facilitate the back-propagation of the gradient.

The **upsampling** (decoder) part can be done either: using Transposed Convolution (hence a well-known checkerboard artefact may appears) or Interpolation followed by Normal Conv-2d.

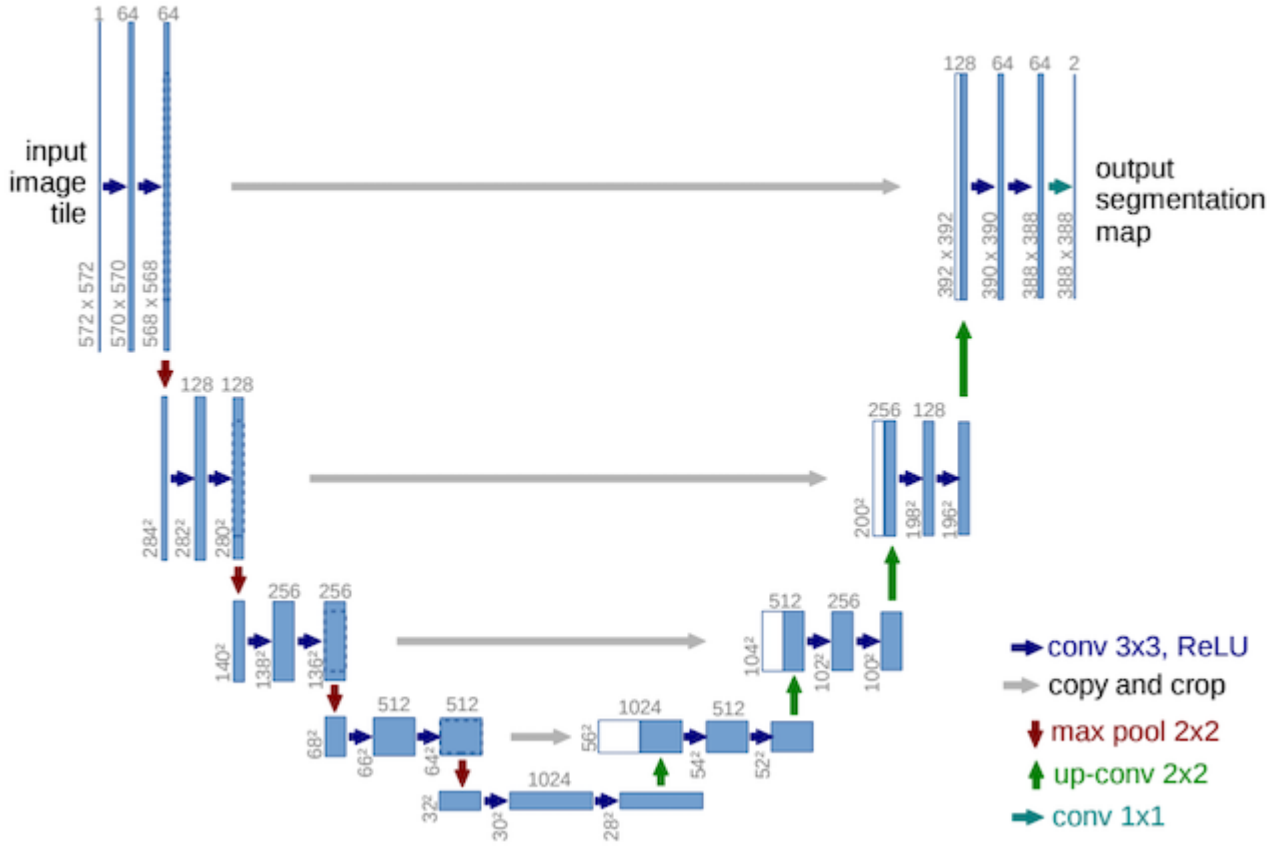


Figure *U-Net architecture for biomedical image segmentation; image source: [RFB15]*

Many to One: reducing the time dimensions

Objective: reduce a (temporal) sequence of embeddings $\{\mathbf{e}_1, \dots \mathbf{e}_{T_x}\}$ to a single embedding $\mathbf{e}$

Usage: map the (temporal) sequence of embeddings (provided by the last layer of a network) to a

Acoustic Scene Classification).

Pooling: The most simple way to achieve this is to use the Mean/Average value (Average Pooling) or Maximum value (Max Pooling) of the $\mathbf{e}_t$ over time t (as done for example in [Die14]).

Attention weighting

Compute a weighted sum of the values $\mathbf{e}_t$ where the weights a_t are **attention** parameters:

$$\mathbf{e} = \sum_{t=0}^{T_x-1} a_t \mathbf{e}_t$$

In [GSL19], it is proposed to compute the attention weights a_t either

1. by computing a new projection of the $\mathbf{e}_t$ and then normalize them: $a_t = \mathrm{softmax}_t(\sigma(\mathbf{v}^T h(\mathbf{e}_t))$
   - with h a learnable embedding, $\mathbf{v}$ the learned parameters of the attention layer
2. doing the same after splitting $\mathbf{e}_t$ in two (along the channel dimensions):
   - the first part $\mathbf{e}_t^{(1)}$ being used to compute "prediction",
   - the second $\mathbf{e}_t^{(2)}$ to compute attention "weights"

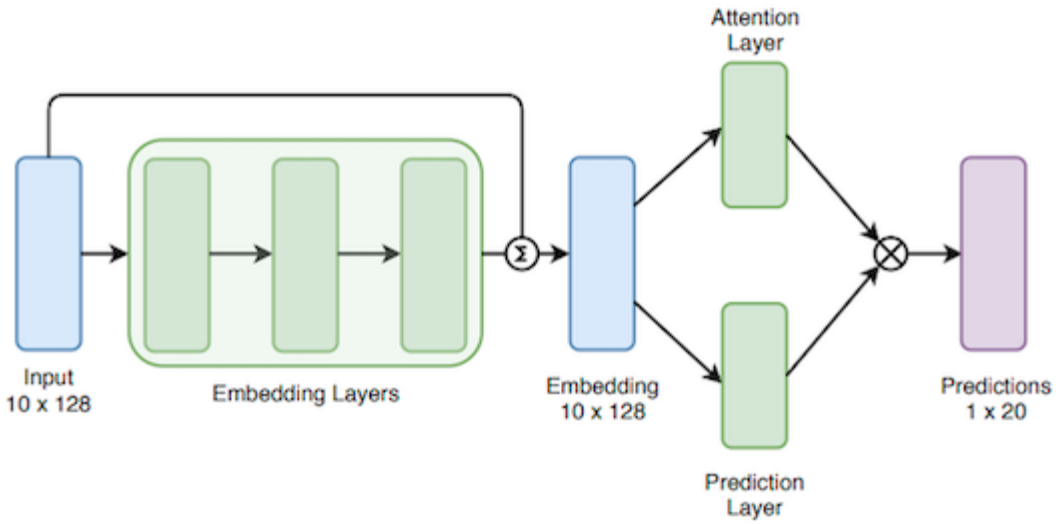


Figure Attention weighting, *image source: [GSL19]*

Skip to main content

```
    Perform attention weighing based on softmax with channel splitting
    Code from https://github.com/furkanyesiler/move
    """
    def __init__(self, nb_channel):
        super().__init__()
        self.nb_channel = nb_channel
    def forward(self, X):
        weights = torch.nn.functional.softmax(X[:, int(self.nb_channel/2):], dim=3)
        X = torch.sum(X[:, :int(self.nb_channel/2)] * weights, dim=3, keepdim=True)
        return X
```

Auto-Pool

The above attention mechanism use a `softmax` for normalizing $\mathbf{a}_t$. We can replace it by the auto-pool operators proposed by [MSB18]defined as

$$\tilde{\mathbf{a}}_t = \frac{\exp(\alpha \cdot \mathbf{a}_t)}{\sum_\tau \exp(\alpha \cdot \mathbf{a}_\tau)}$$

It introduces a training parameter α (also optimized by SGD) which allows to range from

- $\alpha = 0$ (unweighted, a.k.a. average pooling): $\tilde{\mathbf{a}}_t = 1/T_x$
- $\alpha = 1$ (softmax weighted mean): $\tilde{\mathbf{a}}_t = \mathrm{softmax}_t(a_t)$
- $\alpha = \infty$: (a.k.a. max pooling): $\tilde{\mathbf{a}}_t = \max_t(a_t)$

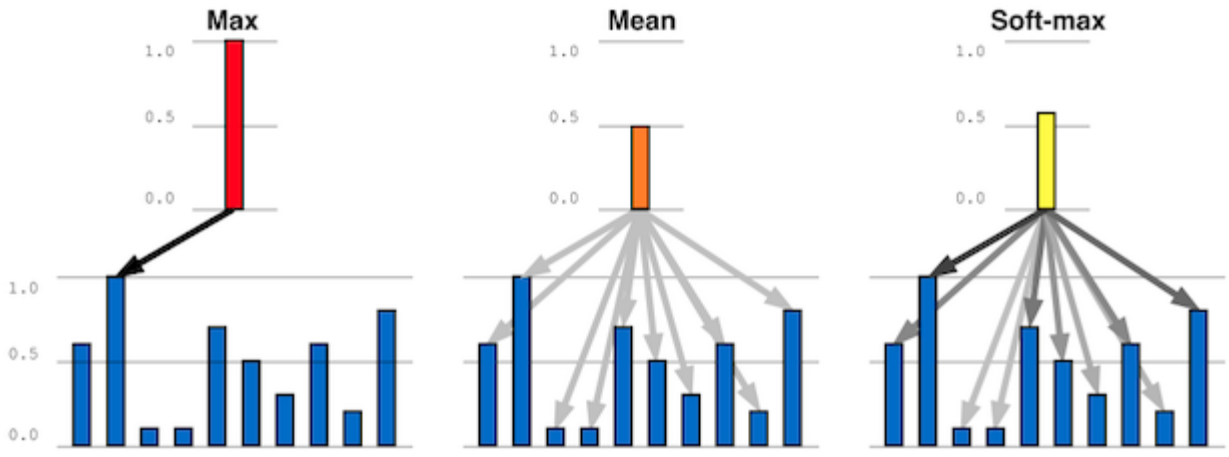


Figure Auto-pool operator *image source: [MSB18]*

```
# Code: https://github.com/furkanyesiler/move
```

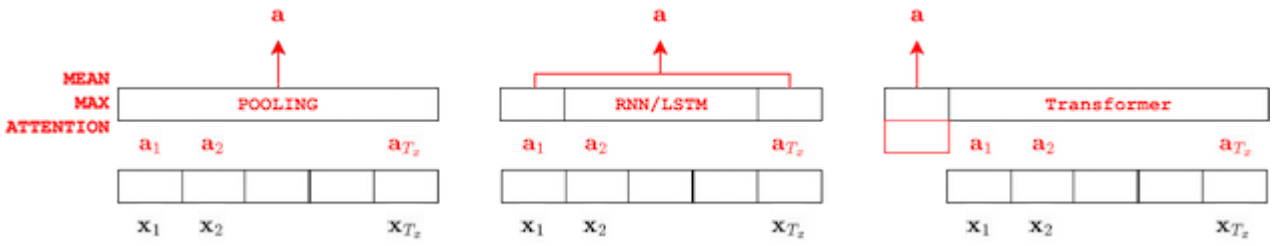
Skip to main content

```
...
def f_autopool_weights(data, autopool_param):
    """
    Calculating the autopool weights for a given tensor
    :param data: tensor for calculating the softmax weights with autopool
    :return: softmax weights with autopool

    see https://arxiv.org/pdf/1804.10070
    alpha=0: unweighted mean
    alpha=1: softmax
    alpha=inf: max-pooling
    """
    # --- x: (batch, 256, 1, T)
    x = data * autopool_param
    # --- max_values: (batch, 256, 1, 1)
    max_values = torch.max(x, dim=3, keepdim=True).values
    # --- softmax (batch, 256, 1, T)
    softmax = torch.exp(x - max_values)
    # --- weights (batch, 256, 1, T)
    weights = softmax / torch.sum(softmax, dim=3, keepdim=True)
    return weights
```

Using models

It is also possible to use a **RNN/LSTM in Many-to-One configuration** (only the last hidden state $\mathbf{x}_{T_x}$ is mapped to an output $\hat{y}$).

Finally it is possible to add an extra CLASS token to a Transformer architecture.



It should be noted that the term “Attention” encapsulates a large set of different paradigms.

- In the **encode-decoder architecture** [BCB15] it is used during decoding to define the correct context $\mathbf{c}_\tau$ to be used to generate the hidden state $\mathbf{s}_\tau$. For this it compares the decoder hidden state $\mathbf{s}_{\tau-1}$ to all the encoder hidden states $\mathbf{a}_t$.
- In the **transformer architecture** [VSP+17] it is used to compute a self-attention. For this, the $\mathbf{x}_t$ are mapped (using matrix projections) to query $\mathbf{q}_t$, key $\mathbf{k}_t$ and values $\mathbf{v}_t$. A given $\mathbf{q}_\tau$ is then

compared to all $\mathbf{k}_t$ to compute attention weights $\mathbf{a}_{t,\tau}$ which are used in the weighting sum of the $\mathbf{v}_t$: $\mathbf{e}_\tau = \sum_t \mathbf{a}_{t,\tau}\mathbf{v}_t$.

Recurrent Architectures

RNN

Recurrent Neural Networks (RNNs) are a type of neural network designed to work with sequential data (e.g., time series, text, etc.). They "remember" information from previous inputs by using hidden states, which allows them to model dependencies across time steps.

Their generic formulation for inputs $\mathbf{x}_t$ over time is:

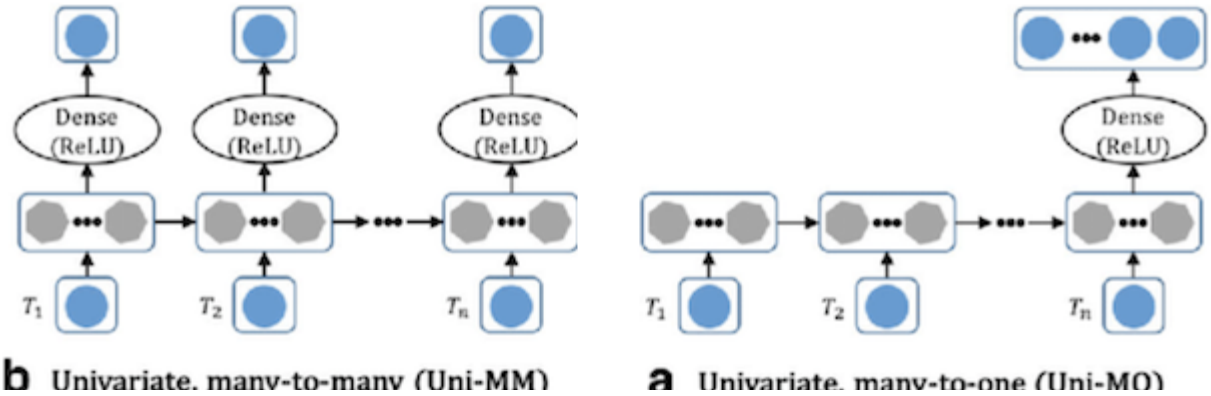
$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{b}_h)$$

where $\mathbf{h}_t$ is the hidden state of the RNN at time t.

A bi-directional-RNN, read the data in both directions (left-to-right and right-to-left). The goal is to make $\mathbf{h}_t$ both dependent on $\mathbf{h}_{t-1}$ and $\mathbf{h}_{t+1}$.

Two configurations are often used with RNNs:

- Many-to-Many: RNN can be used to model the evaluation over time of features (such as done in the past with Kalman filters or HMM). They are often used to represent a Language model.
- Many-to-One: One can also use the last hidden state of a RNN $\mathbf{h}_{T_x}$ where T_x is the length of the input sequence, to sum up the content of the input sequence (see picture below).



Skip to main content

```
torch.nn.RNN(input_size, hidden_size, num_layers=1, bidirectional=False)
```

LSTM

Long Short-Term Memory (LSTM) are a specialized type of RNN designed to handle long-term dependencies more effectively. LSTM use a more complex architecture with

- a memory $\mathbf{c}_t$ over time t,
- a hidden value $\mathbf{h}_t$ and
- a set of gates (input gate, forget gate, and output gate)
    - they allow to control the flow of information between the input $\mathbf{x}_t$, the previous hidden state $\mathbf{h}_{t-1}$ and memory $\mathbf{c}_{t-1}$ and their new values.

This allows them to retain relevant information over longer sequences while "forgetting" irrelevant information.

A critical reason why LSTMs work better than RNNs is that the memory cell provides a path for information to flow across time steps without repeatedly passing it through non-linearities (e.g., `torch.nn.Sigmoid` or `torch.nn.Tanh`). This principle mitigates the vanishing gradient problem and is similar to ResNets and the residual stream in Transformers, where skip connections allow information to bypass layers that add non-linearities.

As in RNNs, two configurations are often used with LSTMs:

- Many-to-many
- Many-to-one

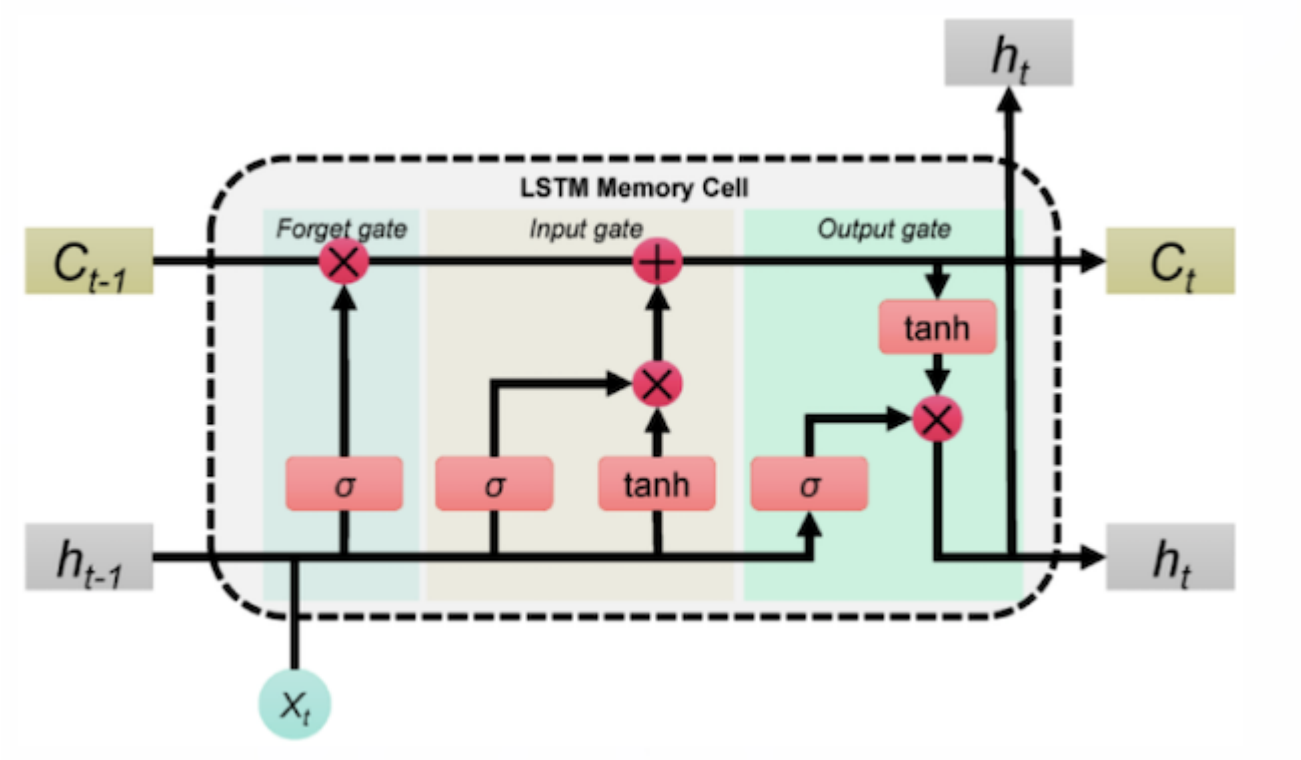


Figure Details of a LSTM cell *image source: Link*

```
torch.nn.LSTM(input_size, hidden_size, num_layers=1, bidirectional=False)
```

Transformer/Self-Attention

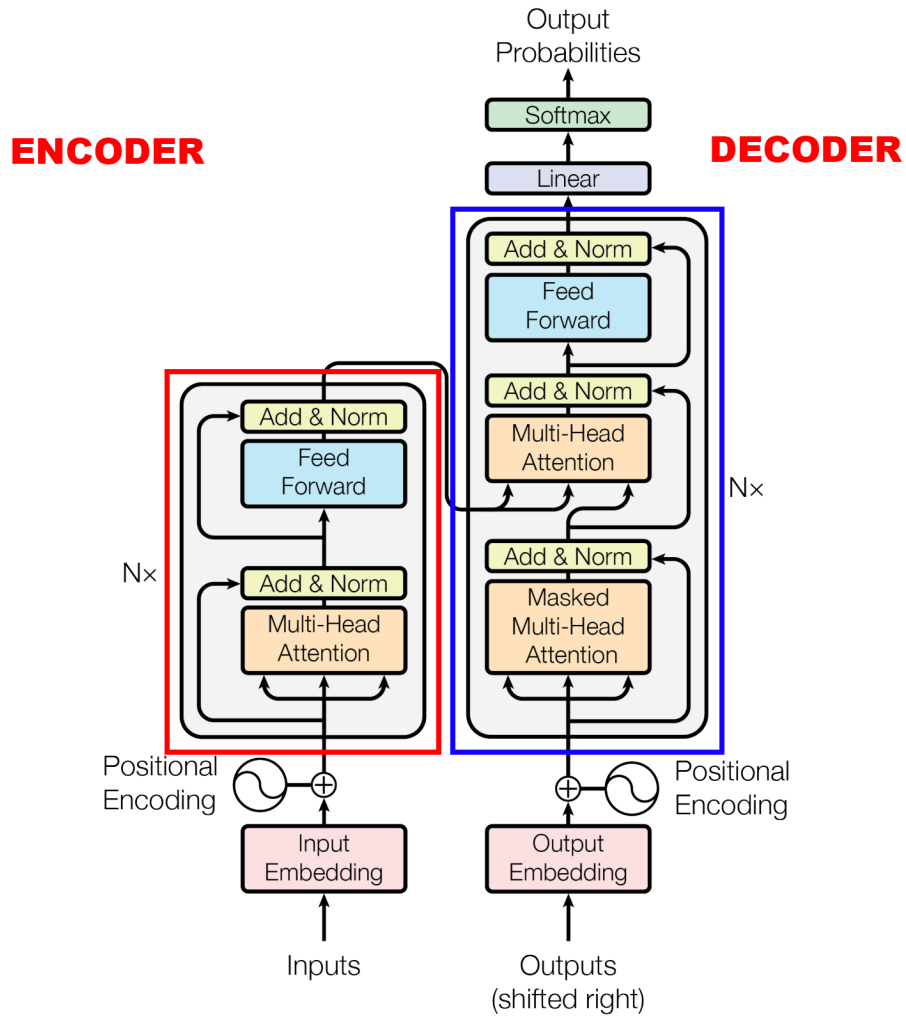


Figure: The Transformer - model architecture.

In recent years, Transformers [VSP+17] widely replaced recurrent architectures for sequence modeling tasks and are also increasingly used instead of convolutional architectures. Their signature component, the **attention mechanism**, gives them a unique advantage over previous architectures. There are several intuitive explanations for the attention mechanism (e.g., attending to important tokens, address-based memory access). Independent of how one thinks about the attention mechanism, its result is an attention matrix (resembling a weight matrix) that is input-dependent, while most other architectures employ weight matrices whose parameters are fixed at inference time.

For sequence modeling (cf. our autoregressive generation example), we usually employ a **causal**

use the **DECODER** part, while for non-causal tasks like *masked token prediction* [DCLT19], the **ENCODER** part is used. Using both, an encoder with cross-connections to the decoder, as proposed in the initial paper, is mainly used to inject conditioning information if needed.

Note that after every `Multi-Head Attention` or `Feed Forward` module, there is an `Add & Norm` operation. This means, the input to each module is added to its output, resulting in a "residual stream", where information is written into or retrieved from. From a simplified point of view, it is now understood that the `Mult-Head Attention` modules rather combine and shuffle information from the residual stream, while weights of the `Feed Forward` modules act as "memories" that inject new information into the residual stream.

Self-Attention Example

This section gives an explanation of self-attention that is illustrative but very simplified. In practice, tokens are not full words but rather word fragments. Keys, values and queries are continuous vectors whose meaning is not as simple and discrete as in the example below, and a token can attend to more than one value. However, the example is correct in how information is propagated through a self-attention layer and could theoretically happen as described.

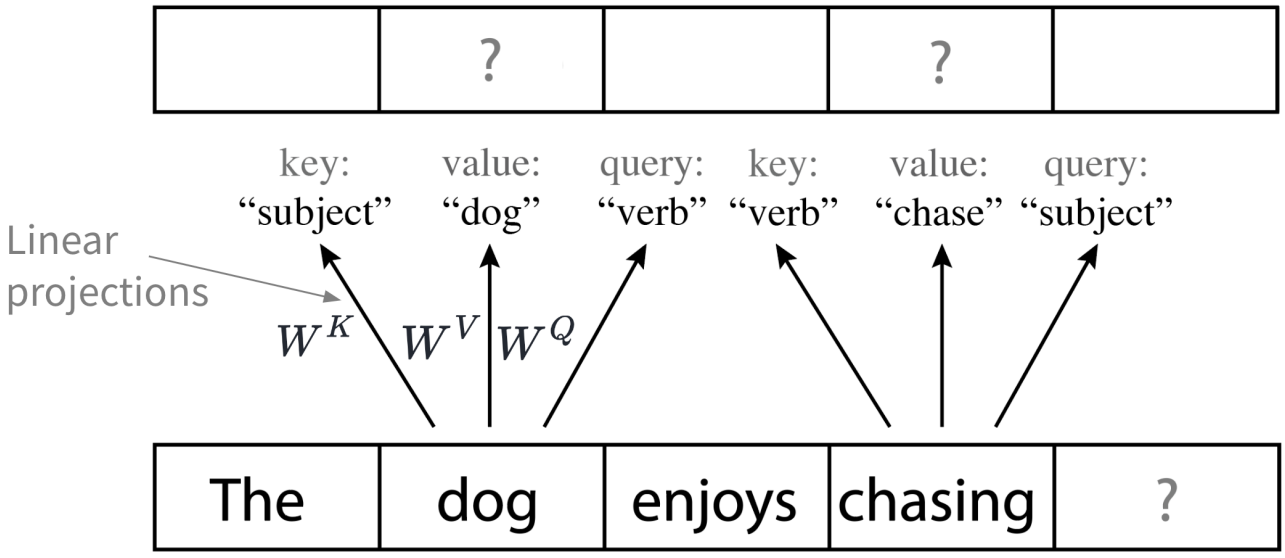


Figure: Simple self-attention example.

In self-attention, every *token* (every *word* in the example above), is represented by an embedding vector. By multiplying every such token embedding with three fixed matrices (W^K, W^V and W^Q) we obtain a key, value and query vector for every position.

Skip to main content

In our simplified example, the model may have learned to emit a *key* vector that stands for verb from a token embedding that stands for chasing, effectively saying "i am a verb!". For the dog embedding, it may ask "what is the dog doing?" and therefore emitting a query resembling the verb key. The result of the self-attention is then to copy the value to whereever a query fits the respective key:

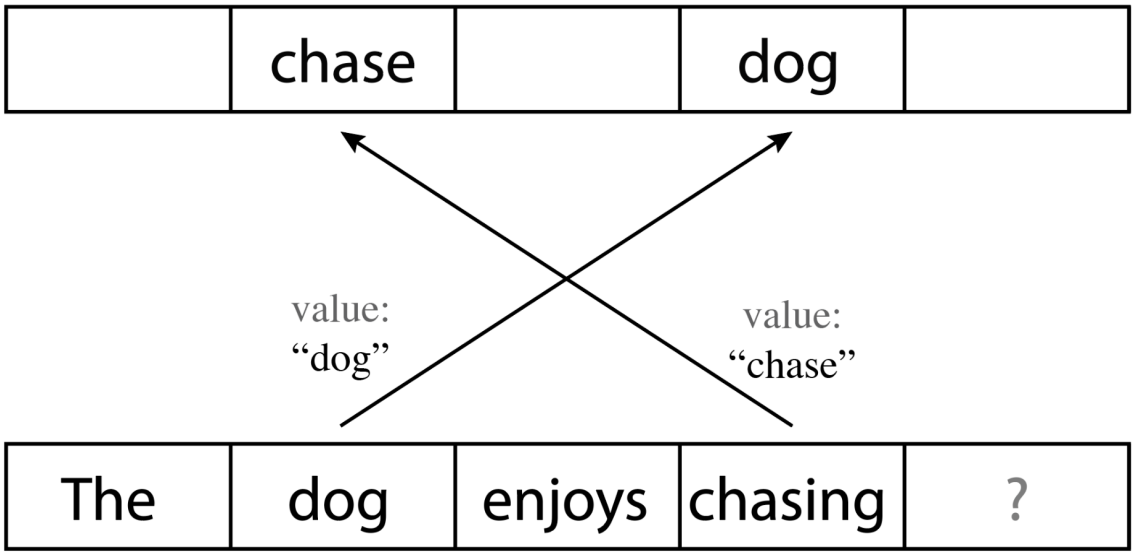


Figure: Result of the simple self-attention example.

As indicated in the architecture diagram, after every attention layer, there is an `Add & Norm` operation. In our example, we start from the dog embedding (i.e., the dog position in a semantic space), and add the chase vector, effectively augmenting dog by moving into the chase direction. As a result, we obtain a "chasing dog" that can then be further transformed in subsequent layers. Through iterative, relative transformations of such embeddings in a semantic space, we can thereby resolve complex relationships and perform precise, final predictions.

Positional Encoding

Note that in the example above, the results would occur the same way if the order of the input sequence would be shuffled (i.e., the chase vector would also be added to the dog position). Transformers do not process tokens sequentially and thus lack an inherent sense of order in the input data. Positional embeddings are a way to inject information about the position of each token within the sequence so that the model can still interpret the sequential nature of the data.

Skip to main content

Embedding Strategy: Each position in the sequence is assigned a unique vector (embedding). This positional embedding can be static (learned) or computed through mathematical functions. It usually has the dimensionality of the token embeddings so that it can be easily added to the token embedding through element-wise addition.

Different Positional Embeddings

Sinusoidal Positional Embeddings: In the original Transformer, positional embeddings are calculated using sine and cosine functions of varying frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Here:

- pos represents the position of the token in the sequence.
- i is the dimension index.
- d is the dimensionality of the embeddings.

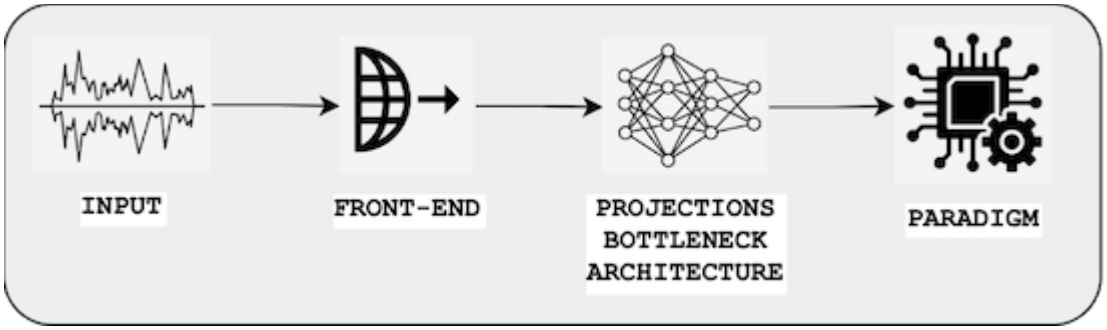
This allows the model to use position information consistently across different sequence lengths, making it invariant to input size.

Learned Positional Embeddings: Another possibility is to use learned positional embeddings, where each position in the sequence has an associated embedding vector that the model learns during training. This can sometimes be more flexible, but it is not as adaptable to longer sequences not seen in training.

Relative Positional Embeddings: In relative positional embeddings [SUV18], the model learns the relative distances between tokens, rather than absolute positions. This approach is more natural for some tasks (e.g., musical sequences), and is assumed to allow the model to generalize better to sequences of varying lengths.

Rotary Positional Embeddings: Rotary Positional Embeddings (RoPE) [SAL+24] provide a way to encode positional information by rotating each token embedding in vector space. This technique allows the model to capture relative positional information through rotation matrices applied to the embeddings at each position. Unlike traditional positional embeddings, RoPE enables better generalization over longer sequences, as the rotational encoding inherently supports extrapolation beyond the training context.

Paradigms



We denote by `paradigm` the overall problem that is used to train a neural network: *such as supervised, metric-learning, self-supervised, adversarial, encoder-decoder, ...*

Supervised

Supervised learning is the most standard paradigm in machine learning, hence in deep learning, in which one has access to both input data X and the corresponding ground-truth y.

The goal is then to define a function f (a specific neural network architecture) and optimize its parameters θ such that $\hat{\mathbf{y}} = f_\theta(\mathbf{x})$ best approximates $\mathbf{y}$. This is done by defining a loss $\mathcal{L}$ associated to the approximation of $\mathbf{y}$ by $\hat{\mathbf{y}}$. Such a loss can be

- binary cross entropy (for binary classification problems, i.e. $y \in \{0, 1\}$, or multi-label problems i.e. $\mathbf{y} \in \{0, 1\}^C$,
- categorical cross entropy (for multi-class problem, i.e. $y \in \{0, \dots, C-1\}$)
- mean square error (for regression problems, i.e. $y \in \mathbb{R}$)

Since we do not have access to the distribution $p(\mathbf{x}, \mathbf{y})$ but only to samples of it $\mathbf{x}^{(i)}$,

$$\theta^* = \arg\min_\theta \sum_{i=0}^{I-1} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

This minimization is usually done using one type of Steepest Gradient Descent (SDG, Momentum, AdaGrad, AdaDelta, ADAM) and using various cardinality for I (stochastic, mini-batch, batch GD).

Self-supervised

Metric Learning

Metric learning is a type of machine learning technique focused on learning a distance function or similarity measure between data points. The goal is to map input data into a space where

- similar examples are close together and
- dissimilar examples are far apart, based on a certain metric (e.g., Euclidean distance).

There exist several type of supervision to achieve this

- **Class** labels: $(\mathbf{x}, \mathbf{y})$
- **Pairwise** similarity/dissimilarity: $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \pm)$
- **Relative** comparisons (triplet): $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}) \Rightarrow d(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) < d(\mathbf{x}^{(1)}, \mathbf{x}^{(3)})$

There exist many algorithm to train such a model such as

- Contrastive Loss [HCL06] in which we optimize in turns (but not jointly) the model to minimize a distance for similar pairs and maximize (up to a margin) it for dissimilar pairs
- Triplet Loss [HA15] see below

The triplet loss can be extended to the multiple-loss with close relationship with Contrastive Learning (InfoNCE, NT-Xent losses).

Fore more details, see the very good tutorial "Metric Learning for Music Information Retrieval" by Brian McFee, Jongpil Lee and Juhan Nam

Triplet Loss

The goal is to train a network f_θ such that the projections of $\mathbf{x}_A$ (an **anchor**), $\mathbf{x}_P$ (a **positive** we consider close), $\mathbf{x}_N$ (a **negative** we consider distant), satisfy the following triplet constraint:

$$d(f_\theta(\mathbf{x}_A), f_\theta(\mathbf{x}_P)) + \alpha < d(f_\theta(\mathbf{x}_A), f_\theta(\mathbf{x}_N))$$
$$d_{AP} + \alpha < d_{AN}$$

In other words, we want d_{AP} to be smaller by a margin α than d_{AN}

We solve this by minimizing the so-called "triplet-loss":

$$\mathcal{L} = \max(d_{AP} + \alpha - d_{AN}, 0)$$

Note: It is usual to L2-normalized the output of f_θ (which then lives in the unit-hypersphere) to facilitate the setting of the α value.

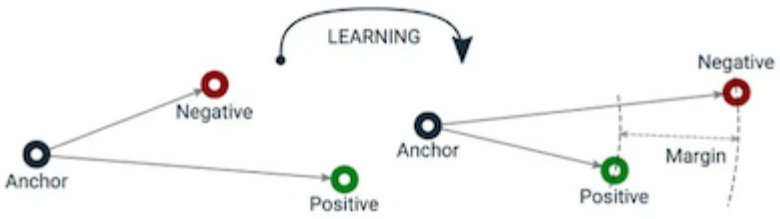


Figure *Triplet Loss, bringing A and P closer and A and N further appart; image source: Link*

```
loss = F.relu(dist_pos + param_d.margin - dist_neg)
```

Triplet Mining

Triplet mining is the process of selecting the triplets for training using the triplet loss. The goal is to ensure the model learns effectively by choosing the right combination of examples.

Given the choice of an `A` and a `P`, we denotes by

- **Easy negatives**: the `N`s that are already far from `A`.
  - they do not provide much useful information (since the model already distinguishes them

Skip to main content

- **Hard negatives**: the `N`s that are very close to `A` (even closer than the `P`).
  - they are difficult for the model to separate
  - this might lead to instability or overfitting.
- **Semi-hard negatives**: the `N`s that are farther from `A` than the `P` but still relatively close.
  - they provide valuable information (because they are challenging without being as problematic as hard negatives).

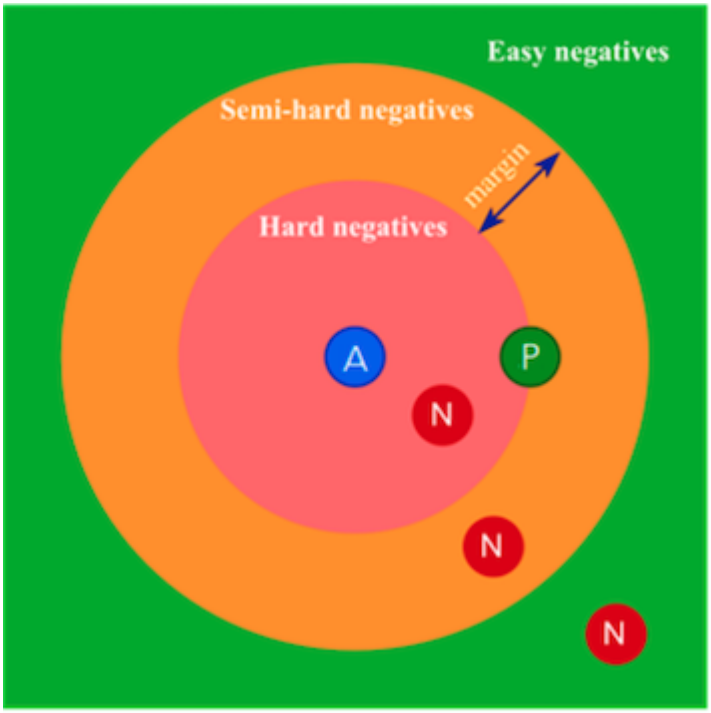


Figure *Triplet mining: easy, hard, semi-hard; image source: Link*

We also distinguish between

- **Offline mining**: triplets are selected prior to training
  - may not adapt to the evolving model during training.
- **Online mining**: triplets are selected during training (from the current mini-batch) using the already learned projection f_θ
  - allows selecting the most informative triplets

Skip to main content

Adversarial

Encoder-Decoder

Diffusion

About the authors

Geoffroy Peeters is a full professor in the Image-Data-Signal (IDS) department of Télécom Paris. Before that (from 2001 to 2018), he was Senior Researcher at IRCAM, leading research related to Music Information Retrieval. He received his Ph.D. in signal processing for speech processing in 2001 and his Habilitation (HDR) in Music Information Retrieval in 2013 from the University Paris VI. His research topics concern signal processing and machine learning (including deep learning) for audio processing, with a strong focus on music. He has participated in many national or European projects, published numerous articles and several patents in these areas, and co-authored the ISO MPEG-7 audio standard. He has been co-general-chair of the DAFx-2011 and ISMIR-2018 conferences, member and president of the ISMIR society, and is the current AASP review chair for ICASSP.

Gabriel Meseguer-Brocal is a research scientist at Deezer with over two years of experience at the company. Before joining Deezer, he completed postdoctoral research at Centre National de la

Skip to main content

Telecommunications, and Electronics with a focus on the Sciences & Technologies of Music and Sound at IRCAM. His research interests include signal processing and deep learning techniques for music processing, with a focus on areas such as source separation, dataset creation, multi-tagging, self-supervised learning, and multimodal analysis.

Alain Riou is a PhD student working on self-supervised learning of musical representations at Télécom-Paris and Sony CSL Paris, under the supervision of Stefan Lattner, Gaëtan Hadjeres and Geoffroy Peeters. Before that, he obtained a master degree in mathematics for machine learning at Ecole Normale Supérieure de Cachan (2020) and another one in signal processing and computer science applied to music at IRCAM (2021). His main research interests are related to deep representation learning, with a strong focus on self-supervised methods for music information retrieval and controllable music generation. His work “PESTO: Pitch Estimation with Self-supervised Transposition-equivariant Objective” received the Best Paper Award at ISMIR 2023.

Stefan Lattner serves as a researcher leader at the music team at Sony CSL Paris, where he focuses on generative AI for music production, music information retrieval, and computational music perception. He earned his PhD in 2019 from Johannes Kepler University (JKU) in Linz, Austria, following his research at the Austrian Research Institute for Artificial Intelligence in Vienna and the Institute of Computational Perception Linz. His studies centered on the modeling of musical structure, encompassing transformation learning and computational relative pitch perception. His current interests include human-computer interaction in music creation, live staging, and information theory in music. He specializes in generative sequence models, computational short-term memories, (self-supervised) representation learning and musical audio generation. In 2019, Lattner received the best paper award at ISMIR for his work, “Learning Complex Basis Functions for Invariant Representations of Audio.”

Skip to main content

Bibliography

BCB15 Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1409.0473.

BKK18 Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, 2018. URL: http://arxiv.org/abs/1803.01271, arXiv:1803.01271.

BED09 Mert Bay, Andreas F. Ehmann, and J. Stephen Downie. Evaluation of multiple-f0 estimation and tracking systems. In Keiji Hirata, George Tzanetakis, and Kazuyoshi Yoshii, editors, *Proceedings of the 10th International Society for Music Information Retrieval Conference, ISMIR 2009, Kobe International Conference Center, Kobe, Japan, October 26-30, 2009*, 315–320. International Society for Music Information Retrieval, 2009. URL: http://ismir2009.ismir.net/proceedings/PS2-21.pdf.

BMS+17 Rachel M. Bittner, Brian McFee, Justin Salamon, Peter Li, and Juan Pablo Bello. Deep salience representations for F0 estimation in polyphonic music. In Sally Jo Cunningham, Zhiyao Duan, Xiao Hu, and Douglas Turnbull, editors, *Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017, Suzhou, China, October 23-27, 2017*, 63–70. 2017. URL: https://brianmcfee.net/papers/ismir2017_salience.pdf.

Bro91 J. Brown. Calculation of a constant q spectral transform. *JASA (Journal of the Acoustical Society of America)*, 89(1):425–434, 1991.

BP09 Juan-Jo Burred and Geoffroy Peeters. An adaptive system for music classification and tagging. In *Proc. of LSAS (International Workshop on Learning the Semantics of Audio Signals)*. Graz, Austria, 2009.

CZ18 CJ Carr and Zack Zukowski. Generating albums with samplernn to imitate metal, rock, and punk bands. *CoRR*, 2018.

CTP11 Christophe Charbuillet, Damien Tardieu, and Geoffroy Peeters. Gmm supervector for

Effects), 425–428. Paris, France, September 2011.

CFS16 Keunwoo Choi, György Fazekas, and Mark B. Sandler. Automatic tagging using deep convolutional neural networks. In Michael I. Mandel, Johanna Devaney, Douglas Turnbull, and George Tzanetakis, editors, *Proceedings of the 17th International Society for Music Information Retrieval Conference, ISMIR 2016, New York City, United States, August 7-11, 2016*, 805–811. 2016. URL: https://wp.nyu.edu/ismir2016/wp-content/uploads/sites/2294/2016/07/009_Paper.pdf.

Cho17 François Chollet. Xception: deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, 1800–1807. IEEE Computer Society, 2017. URL: https://doi.org/10.1109/CVPR.2017.195, doi:10.1109/CVPR.2017.195.

DK02 A. DeCheveigne and H. Kawahara. Yin, a fundamental frequency estimator for speech and music. *JASA (Journal of the Acoustical Society of America)*, 111(4):1917–1930, 2002.

DCLT19 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 4171–4186. Association for Computational Linguistics, 2019.

Die14 Sander Dieleman. Recommending music on spotify with deep learning. Technical Report, Spotify, http://benanne.github.io/2014/08/05/spotify-cnns.html, 2014.

DS14 Sander Dieleman and Benjamin Schrauwen. End-to-end learning for music audio. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, Florence, Italy, May 4-9, 2014*, 6964–6968. IEEE, 2014. URL: https://doi.org/10.1109/ICASSP.2014.6854950, doi:10.1109/ICASSP.2014.6854950.

DMP19 Chris Donahue, Julian J. McAuley, and Miller S. Puckette. Adversarial audio synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=ByMVTsR5KQ.

DEP19 Guillaume Doras, Philippe Esling, and Geoffroy Peeters. On the use of u-net for dominant melody estimation in polyphonic music. In *Proc. of First International Workshop on Multilayer Music Representation and Processing (MMRP19)*. Milan, Italy, January , 24-25 2019. URL: https://hal.science/hal-02457728/document.

Skip to main content

Guillaume Doras and Geoffroy Peeters. Cover detection using dominant melody embeddings. In Arthur Flexer, Geoffroy Peeters, Julián Urbano, and Anja Volk, editors, *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019, Delft, The Netherlands, November 4-8, 2019*, 107–114. 2019. URL: http://archives.ismir.net/ismir2019/paper/000010.pdf.

DPZ10 Zhiyao Duan, Bryan Pardo, and Changshui Zhang. Multiple fundamental frequency estimation by modeling spectral peaks and non-peak regions. *IEEE Trans. Speech Audio Process.*, 18(8):2121–2133, 2010. URL: https://doi.org/10.1109/TASL.2010.2042119, doi:10.1109/TASL.2010.2042119.

DefossezCSA23 Alexandre Défossez, Jade Copet, Gabriel Synnaeve, and Yossi Adi. High fidelity neural audio compression. *Trans. Mach. Learn. Res.*, 2023.

DefossezUBB19 Alexandre Défossez, Nicolas Usunier, Léon Bottou, and Francis R. Bach. Demucs: deep extractor for music sources with extra unlabeled data remixed. *CoRR*, 2019. URL: http://arxiv.org/abs/1909.01174, arXiv:1909.01174.

EP07 Daniel P. W. Ellis and Graham E. Poliner. Identifying 'cover songs' with chroma features and dynamic programming beat tracking. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2007, Honolulu, Hawaii, USA, April 15-20, 2007*, 1429–1432. IEEE, 2007. URL: https://doi.org/10.1109/ICASSP.2007.367348, doi:10.1109/ICASSP.2007.367348.

Elm90 Jeffrey L. Elman. Finding structure in time. *Cogn. Sci.*, 14(2):179–211, 1990. URL: https://doi.org/10.1207/s15516709cog1402_1, doi:10.1207/S15516709COG1402_1.

EBD10 Valentin Emiya, Roland Badeau, and Bertrand David. Multipitch estimation of piano sounds using a new probabilistic spectral smoothness principle. *IEEE Trans. Speech Audio Process.*, 18(6):1643–1654, 2010. URL: https://doi.org/10.1109/TASL.2009.2038819, doi:10.1109/TASL.2009.2038819.

EAC+19 Jesse H. Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, and Adam Roberts. Gansynth: adversarial neural audio synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=H1xQVn09FX.

Skip to main content

Jesse H. Engel, Lamtharn Hantrakul, Chenjie Gu, and Adam Roberts. DDSP: differentiable digital signal processing. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=B1x1ma4tDr.

ECT+24 Zach Evans, CJ Carr, Josiah Taylor, Scott H. Hawley, and Jordi Pons. Fast timing-conditioned latent audio diffusion. In *ICML*. OpenReview.net, 2024.

FBR13 Benoit Fuentes, Roland Badeau, and Gaël Richard. Harmonic adaptive latent component analysis of audio and application to music transcription. *IEEE Trans. Speech Audio Process.*, 21(9):1854–1866, 2013. URL: https://doi.org/10.1109/TASL.2013.2260741, doi:10.1109/TASL.2013.2260741.

Fuj99 Takuya Fujishima. Realtime chord recognition of musical sound: a system using common lisp music. In *Proceedings of the 1999 International Computer Music Conference, ICMC 1999, Beijing, China, October 22-27, 1999*. Michigan Publishing, 1999. URL: https://hdl.handle.net/2027/spo.bbp2372.1999.446.

GFR+20 Beat Gfeller, Christian Havnø Frank, Dominik Roblek, Matthew Sharifi, Marco Tagliasacchi, and Mihajlo Velimirovic. SPICE: self-supervised pitch estimation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 28:1118–1128, 2020. URL: https://doi.org/10.1109/TASLP.2020.2982285, doi:10.1109/TASLP.2020.2982285.

GPougetAbadieM+14 Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *CoRR*, 2014. URL: http://arxiv.org/abs/1406.2661, arXiv:1406.2661.

Got06 Masataka Goto. AIST annotation for the RWC music database. In *ISMIR 2006, 7th International Conference on Music Information Retrieval, Victoria, Canada, 8-12 October 2006, Proceedings*, 359–360. 2006.

GHNO02 Masataka Goto, Hiroki Hashiguchi, Takuichi Nishimura, and Ryuichi Oka. RWC music database: popular, classical and jazz music databases. In *ISMIR 2002, 3rd International Conference on Music Information Retrieval, Paris, France, October 13-17, 2002, Proceedings*. 2002. URL: http://ismir2002.ismir.net/proceedings/03-SP04-1.pdf.

GGBE24 Azalea Gui, Hannes Gamper, Sebastian Braun, and Dimitra Emmanouilidou. Adapting

Skip to main content

GSL19 Siddharth Gururani, Mohit Sharma, and Alexander Lerch. An attention mechanism for musical instrument recognition. In Arthur Flexer, Geoffroy Peeters, Julián Urbano, and Anja Volk, editors, *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019, Delft, The Netherlands, November 4-8, 2019*, 83–90. 2019. URL: http://archives.ismir.net/ismir2019/paper/000007.pdf.

HCL06 Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006), 17-22 June 2006, New York, NY, USA*, 1735–1742. IEEE Computer Society, 2006. URL: https://doi.org/10.1109/CVPR.2006.100, doi:10.1109/CVPR.2006.100.

HZRS16 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 770–778. IEEE Computer Society, 2016. URL: https://doi.org/10.1109/CVPR.2016.90, doi:10.1109/CVPR.2016.90.

HJA20 Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html.

HA15 Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6622.

HZC+17 Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: efficient convolutional neural networks for mobile vision applications. *CoRR*, 2017. URL: http://arxiv.org/abs/1704.04861, arXiv:1704.04861.

HNB13 Eric J. Humphrey, Oriol Nieto, and Juan Pablo Bello. Data driven and discriminative projections for large-scale cover song identification. In Alceu de Souza Britto Jr., Fabien Gouyon, and Simon Dixon, editors, *Proceedings of the 14th International Society for Music*

Skip to main content

Information Retrieval Conference, ISMIR 2013, Curitiba, Brazil, November 4-8, 2013, 149–154. 2013. URL: http://www.ppgia.pucpr.br/ismir2013/wp-content/uploads/2013/09/246_Paper.pdf.

JHM+17 Andreas Jansson, Eric J. Humphrey, Nicola Montecchio, Rachel M. Bittner, Aparna Kumar, and Tillman Weyde. Singing voice separation with deep u-net convolutional networks. In Sally Jo Cunningham, Zhiyao Duan, Xiao Hu, and Douglas Turnbull, editors, *Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017, Suzhou, China, October 23-27, 2017*, 745–751. 2017. URL: https://ismir2017.smcnus.org/wp-content/uploads/2017/10/171_Paper.pdf.

KZRS19 Kevin Kilgour, Mauricio Zuluaga, Dominik Roblek, and Matthew Sharifi. Fréchet audio distance: A reference-free metric for evaluating music enhancement algorithms. In *INTERSPEECH*, 2350–2354. ISCA, 2019.

KW14 Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. URL: http://arxiv.org/abs/1312.6114.

Kla03 Anssi Klapuri. Multiple fundamental frequency estimation based on harmonicity and spectral smoothness. *Speech and Audio Processing, IEEE Transactions on*, 11(6):804–816, 2003.

KCI+20 Qiuqiang Kong, Yin Cao, Turab Iqbal, Yuxuan Wang, Wenwu Wang, and Mark D. Plumbley. Panns: large-scale pretrained audio neural networks for audio pattern recognition. *IEEE ACM Trans. Audio Speech Lang. Process.*, 28:2880–2894, 2020.

KW16 Filip Korzeniowski and Gerhard Widmer. Feature learning for chord recognition: the deep chroma extractor. In Michael I. Mandel, Johanna Devaney, Douglas Turnbull, and George Tzanetakis, editors, *Proceedings of the 17th International Society for Music Information Retrieval Conference, ISMIR 2016, New York City, United States, August 7-11, 2016*, 37–43. 2016. URL: https://wp.nyu.edu/ismir2016/wp-content/uploads/sites/2294/2016/07/178_Paper.pdf.

KMS+21 Khaled Koutini, Shahed Masoudian, Florian Schmid, Hamid Eghbal-zadeh, Jan Schlüter, and Gerhard Widmer. Learning general audio representations with large-scale training of patchout audio transformers. In Joseph Turian, Björn W. Schuller, Dorien Herremans, Katrin Kirchhoff, L. Paola Garc\'ıa-Perera, and Philippe Esling, editors, *HEAR: Holistic Evaluation of Audio Representations, Virtual Event, December 13-14, 2021*, volume 166 of Proceedings of

Skip to main content

Machine Learning Research, 65–89. PMLR, 2021. URL: https://proceedings.mlr.press/v166/koutini22a.html.

KSL+23 Rithesh Kumar, Prem Seetharaman, Alejandro Luebs, Ishaan Kumar, and Kundan Kumar. High-fidelity audio compression with improved RVQGAN. In *NeurIPS*. 2023.

LPKN17 Jongpil Lee, Jiyoung Park, Keunhyoung Luke Kim, and Juhan Nam. Sample-level deep convolutional neural networks for music auto-tagging using raw waveforms. *CoRR*, 2017. URL: http://arxiv.org/abs/1703.01789, arXiv:1703.01789.

LYZ+24 Yizhi Li, Ruibin Yuan, Ge Zhang, Yinghao Ma, Xingran Chen, Hanzhi Yin, Chenghao Xiao, Chenghua Lin, Anton Ragni, Emmanouil Benetos, Norbert Gyenge, Roger B. Dannenberg, Ruibo Liu, Wenhu Chen, Gus Xia, Yemin Shi, Wenhao Huang, Zili Wang, Yike Guo, and Jie Fu. MERT: acoustic music understanding model with large-scale self-supervised training. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=w3YZ9MSlBu.

LMW+22 Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, 11966–11976. IEEE, 2022. URL: https://doi.org/10.1109/CVPR52688.2022.01167, doi:10.1109/CVPR52688.2022.01167.

LM19 Yi Luo and Nima Mesgarani. Conv-tasnet: surpassing ideal time-frequency magnitude masking for speech separation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 27(8):1256–1266, 2019. URL: https://doi.org/10.1109/TASLP.2019.2915167, doi:10.1109/TASLP.2019.2915167.

MKO+22 Matthew C. McCallum, Filip Korzeniowski, Sergio Oramas, Fabien Gouyon, and Andreas F. Ehmann. Supervised and unsupervised learning of audio representations for music understanding. In Preeti Rao, Hema A. Murthy, Ajay Srinivasamurthy, Rachel M. Bittner, Rafael Caro Repetto, Masataka Goto, Xavier Serra, and Marius Miron, editors, *Proceedings of the 23rd International Society for Music Information Retrieval Conference, ISMIR 2022, Bengaluru, India, December 4-8, 2022*, 256–263. 2022. URL: https://archives.ismir.net/ismir2022/paper/000030.pdf.

MB17 Brian McFee and Juan Pablo Bello. Structured training for large-vocabulary chord recognition. In Sally Jo Cunningham, Zhiyao Duan, Xiao Hu, and Douglas Turnbull, editors, *Proceedings of the 18th International Society for Music Information Retrieval Conference*

ISMIR 2017, Suzhou, China, October 23-27, 2017, 188–194. 2017. URL: https://ismir2017.smcnus.org/wp-content/uploads/2017/10/77_Paper.pdf.

MSB18 Brian McFee, Justin Salamon, and Juan Pablo Bello. Adaptive pooling operators for weakly labeled sound event detection. *IEEE ACM Trans. Audio Speech Lang. Process.*, 26(11):2180–2193, 2018. URL: https://doi.org/10.1109/TASLP.2018.2858559, doi:10.1109/TASLP.2018.2858559.

MKG+17 Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron C. Courville, and Yoshua Bengio. Samplernn: an unconditional end-to-end neural audio generation model. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=SkxKPDv5xl.

NALR21 Javier Nistal, Cyran Aouameur, Stefan Lattner, and Gaël Richard. VQCPC-GAN: variable-length adversarial audio synthesis using vector-quantized contrastive predictive coding. In *WASPAA*, 116–120. IEEE, 2021.

NAVL22 Javier Nistal, Cyran Aouameur, Ithan Velarde, and Stefan Lattner. Drumgan VST: A plugin for drum sound analysis/synthesis with autoencoding generative adversarial networks. *Proc. of International Conference on Machine Learning ICML, Workshop on Machine Learning for Audio Synthesis, MLAS, 2022*, 2022.

NLR20 Javier Nistal, Stefan Lattner, and Gaël Richard. DRUMGAN: synthesis of drum sounds with timbral feature conditioning using generative adversarial networks. In *ISMIR*, 590–597. 2020.

NLR21 Javier Nistal, Stefan Lattner, and Gaël Richard. Darkgan: exploiting knowledge distillation for comprehensible audio synthesis with gans. In *ISMIR*, 484–492. 2021.

NPA+24 Javier Nistal, Marco Pasini, Cyran Aouameur, Maarten Grachten, and Stefan Lattner. Diff-a-riff: musical accompaniment co-creation via latent diffusion models. *CoRR*, 2024.

NoePM20 Paul-Gauthier Noé, Titouan Parcollet, and Mohamed Morchid. CGCNN: complex gabor convolutional neural network on raw speech. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, 7724–7728. IEEE, 2020. URL: https://doi.org/10.1109/ICASSP40776.2020.9054220, doi:10.1109/ICASSP40776.2020.9054220.

Skip to main content

Hélène Papadopoulos and Geoffroy Peeters. Large-scale study of chord estimation algorithms based on chroma representation. In *Proc. of IEEE CBMI (International Workshop on Content-Based Multimedia Indexing)*. Bordeaux, France, 2007.

PCDV20 Manuel Pariente, Samuele Cornell, Antoine Deleforge, and Emmanuel Vincent. Filterbank design for end-to-end speech separation. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, 6364–6368. IEEE, 2020. URL: https://doi.org/10.1109/ICASSP40776.2020.9053038, doi:10.1109/ICASSP40776.2020.9053038.

PLF24 Marco Pasini, Stefan Lattner, and George Fazekas. Music2latent: consistency autoencoders for latent audio compression. *CoRR*, 2024. URL: https://doi.org/10.48550/arXiv.2408.06500, arXiv:2408.06500, doi:10.48550/ARXIV.2408.06500.

PSchluter22 Marco Pasini and Jan Schlüter. Musika! fast infinite waveform music generation. In *ISMIR*, 543–550. 2022.

PP13 Johan Pauwels and Geoffroy Peeters. Evaluating automatically estimated chord sequences. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, 749–753. IEEE, 2013. URL: https://doi.org/10.1109/ICASSP.2013.6637748, doi:10.1109/ICASSP.2013.6637748.

Pee04 Geoffroy Peeters. A large set of audio features for sound description (similarity and classification) in the cuidado project. Cuidado Project Report, Ircam, 2004.

Pee07 Geoffroy Peeters. A generic system for audio indexing: application to speech/ music segmentation and music genre. In *In Proc. of DAFx (International Conference on Digital Audio Effects)*. Bordeaux, France, 2007.

PLS16 Jordi Pons, Thomas Lidy, and Xavier Serra. Experimenting with musically motivated convolutional neural networks. In *14th International Workshop on Content-Based Multimedia Indexing, CBMI 2016, Bucharest, Romania, June 15-17, 2016*, 1–6. IEEE, 2016. URL: https://doi.org/10.1109/CBMI.2016.7500246, doi:10.1109/CBMI.2016.7500246.

RMH+14 Colin Raffel, Brian McFee, Eric J. Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel P. W. Ellis. Mir_eval: A transparent implementation of common MIR metrics. In Hsin-Min Wang, Yi-Hsuan Yang, and Jin Ha Lee, editors, *Proceedings of the 15th International*

2014, 367–372. 2014. URL: http://www.terasoft.com.tw/conf/ismir2014/proceedings/T066_320_Paper.pdf.

RB18 Mirco Ravanelli and Yoshua Bengio. Speaker recognition from raw waveform with sincnet. In *2018 IEEE Spoken Language Technology Workshop, SLT 2018, Athens, Greece, December 18-21, 2018*, 1021–1028. IEEE, 2018. URL: https://doi.org/10.1109/SLT.2018.8639585, doi:10.1109/SLT.2018.8639585.

RLHP23 Alain Riou, Stefan Lattner, Gaëtan Hadjeres, and Geoffroy Peeters. PESTO: pitch estimation with self-supervised transposition-equivariant objective. In Augusto Sarti, Fabio Antonacci, Mark Sandler, Paolo Bestagini, Simon Dixon, Beici Liang, Gaël Richard, and Johan Pauwels, editors, *Proceedings of the 24th International Society for Music Information Retrieval Conference, ISMIR 2023, Milan, Italy, November 5-9, 2023*, 535–544. 2023. URL: https://doi.org/10.5281/zenodo.10265343, doi:10.5281/ZENODO.10265343.

RFB15 Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells III, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, volume 9351 of Lecture Notes in Computer Science, 234–241. Springer, 2015. URL: https://doi.org/10.1007/978-3-319-24574-4_28, doi:10.1007/978-3-319-24574-4_28.

SKP15 Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 815–823. IEEE Computer Society, 2015. URL: https://doi.org/10.1109/CVPR.2015.7298682, doi:10.1109/CVPR.2015.7298682.

SerraGomez08 Joan Serrà and Emilia Gómez. Audio cover song identification based on tonal sequence alignment. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2008, March 30 - April 4, 2008, Caesars Palace, Las Vegas, Nevada, USA*, 61–64. IEEE, 2008. URL: https://doi.org/10.1109/ICASSP.2008.4517546, doi:10.1109/ICASSP.2008.4517546.

Sey10 Klaus Seyerlehner. *Content-Based Music Recommender Systems: Beyond simple Frame-Level Audio Similarity*. PhD thesis, Johannes Kepler Universität, Linz, Austria, December 2010.

SHV18 Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position

Skip to main content

SE03 A. Sheh and Daniel P. W. Ellis. Chord segmentation and recognition using em-trained hidden markov models. In *Proc. of ISMIR (International Society for Music Information Retrieval)*, 183–189. Baltimore, Maryland, USA, 2003.

SED18 Daniel Stoller, Sebastian Ewert, and Simon Dixon. Wave-u-net: A multi-scale neural network for end-to-end audio source separation. In Emilia Gómez, Xiao Hu, Eric Humphrey, and Emmanouil Benetos, editors, *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, September 23-27, 2018*, 334–340. 2018. URL: http://ismir2018.ircam.fr/doc/pdfs/205_Paper.pdf.

Stu13 Bob L. Sturm. The GTZAN dataset: its contents, its faults, their effects on evaluation, and its future use. *CoRR*, 2013. URL: http://arxiv.org/abs/1306.1461, arXiv:1306.1461.

SAL+24 Jianlin Su, Murtadha H. M. Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

TLL+24 Modan Tailleur, Junwon Lee, Mathieu Lagrange, Keunwoo Choi, Laurie M. Heller, Keisuke Imoto, and Yuki Okamoto. Correlation of fréchet audio distance with human perception of environmental audio is embedding dependant. *CoRR*, 2024.

TC02 George Tzanetakis and Perry R. Cook. Musical genre classification of audio signals. *IEEE Trans. Speech Audio Process.*, 10(5):293–302, 2002. URL: https://doi.org/10.1109/TSA.2002.800560, doi:10.1109/TSA.2002.800560.

vdODZ+16 Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In Alan W. Black, editor, *The 9th ISCA Speech Synthesis Workshop, SSW 2016, Sunnyvale, CA, USA, September 13-15, 2016*, 125. ISCA, 2016. URL: https://www.isca-archive.org/ssw_2016/vandenoord16_ssw.html.

VSP+17 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 5998–6008. 2017. URL:

Skip to main content

https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Wak99 Gregory H. Wakefield. Mathematical representation of joint time-chroma distributions. In *Proc. of SPIE conference on Advanced Signal Processing Algorithms, Architecture and Implementations*, 637–645. Denver, Colorado, USA, 1999.

WP21 Christof Weiss and Geoffroy Peeters. Training deep pitch-class representations with a multi-label CTC loss. In Jin Ha Lee, Alexander Lerch, Zhiyao Duan, Juhan Nam, Preeti Rao, Peter van Kranenburg, and Ajay Srinivasamurthy, editors, *Proceedings of the 22nd International Society for Music Information Retrieval Conference, ISMIR 2021, Online, November 7-12, 2021*, 754–761. 2021. URL: https://archives.ismir.net/ismir2021/paper/000094.pdf.

WP22 Christof Weiss and Geoffroy Peeters. Comparing deep models and evaluation strategies for multi-pitch estimation in music recordings. *Speech and Audio Processing, IEEE Transactions on*, 2022. URL: https://arxiv.org/pdf/2202.09198.

WCZ+23 Yusong Wu, Ke Chen, Tianyu Zhang, Yuchen Hui, Taylor Berg-Kirkpatrick, and Shlomo Dubnov. Large-scale contrastive language-audio pretraining with feature fusion and keyword-to-caption augmentation. In *ICASSP*, 1–5. IEEE, 2023.

YSerraGomez20 Furkan Yesiler, Joan Serrà, and Emilia Gómez. Accurate and scalable version identification using musically-motivated embeddings. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, 21–25. IEEE, 2020. URL: https://doi.org/10.1109/ICASSP40776.2020.9053793, doi:10.1109/ICASSP40776.2020.9053793.

YK16 Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1511.07122.

ZTdCQT21 Neil Zeghidour, Olivier Teboul, Félix de Chaumont Quitry, and Marco Tagliasacchi. LEAF: A learnable frontend for audio classification. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=jM76BCb6F9m.