**Encoder, Synthesizer and Vocoder Real‑Time Voice Cloning**

1. **Install Anaconda & Python**
   * Download and install Anaconda (recommended Python 3.7–3.9).
   * Create a new environment:

conda create -n voiceclone python=3.9

conda activate voiceclone

conda install -c conda forge ffmpeg

1. **Install Dependencies**

* inflect==5.3.0
* librosa==0.8.1
* matplotlib==3.5.1
* numpy==1.20.3
* Pillow==8.4.0
* PyQt5==5.15.6
* scikit-learn==1.0.2
* scipy==1.7.3
* sounddevice==0.4.3
* SoundFile==0.10.3.post1
* tqdm==4.62.3
* umap-learn==0.5.2
* Unidecode==1.3.2
* urllib3==1.26.7
* visdom==0.1.8.9
* webrtcvad==2.0.10
* Use conda install -c conda-forge ffmpeg for audio encoding/decoding support.
* Install PyTorch

1. **Clone the Toolbox Repository**
   * Download the GitHub repo: https://github.com/CorentinJ/Real-Time-Voice-Cloning
   * Clone or unzip it to a local directory.
2. **Download Pre-trained Models**
   * Obtain encoder, synthesizer, and vocoder .pt files from the official release and place them in:

encoder/saved\_models/pretrained.pt

synthesizer/saved\_models/pretrained/pretrained.pt

vocoder/saved\_models/pretrained/pretrained.pt

1. **Launch the Toolbox**
   * Run the demo:

cd Real-Time-Voice-Cloning

python demo\_toolbox.py

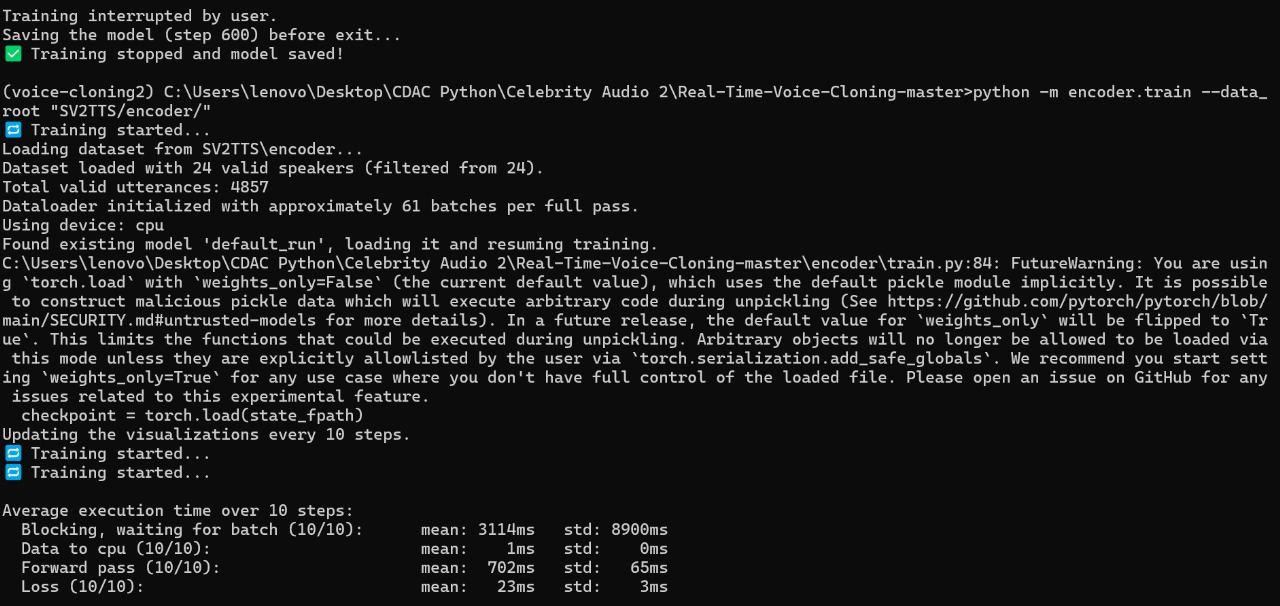
1. **Uploading of Audio**

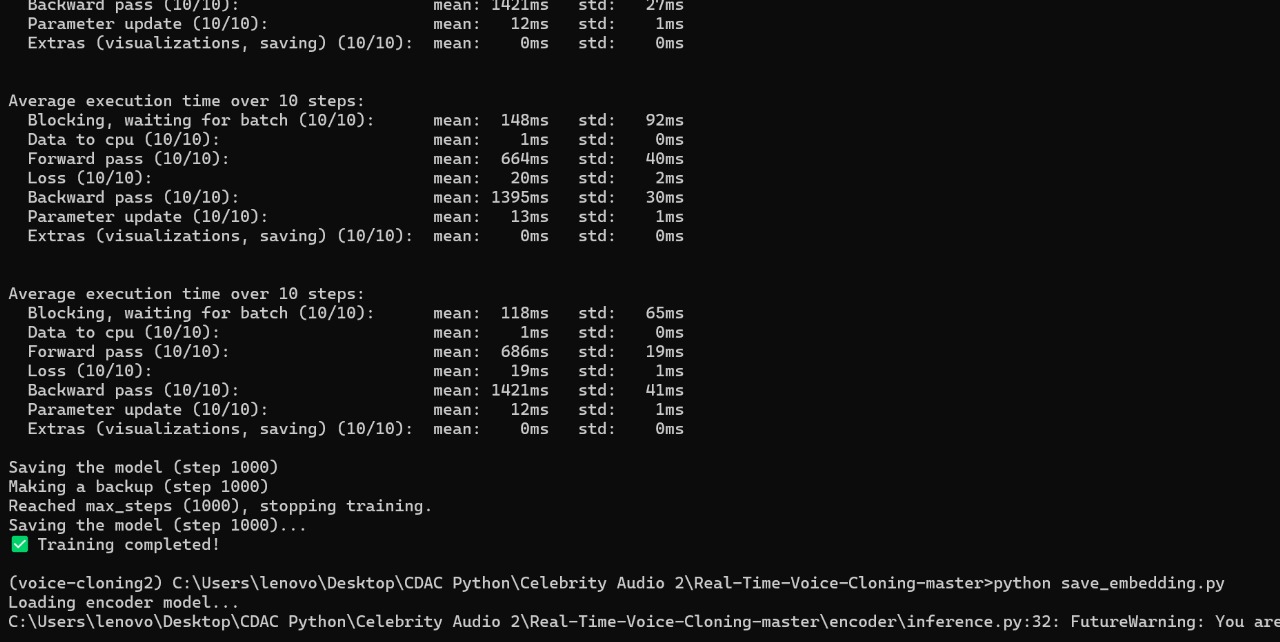
* The toolbox GUI allows:
  + Uploading or selecting a voice sample.
  + Entering custom text to synthesize.
  + Real-time cloning of one’s own voice or using provided samples.
* Synthesized speech generated via the loaded models.

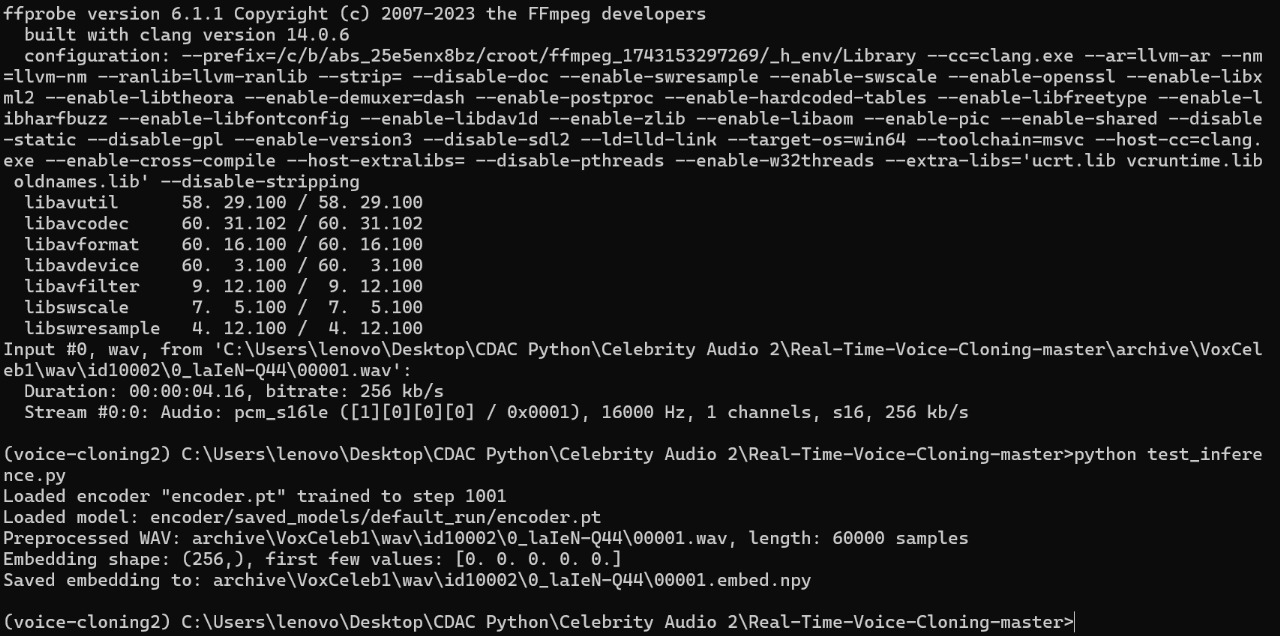
1. We can test with LibriSpeech/train-clean-100

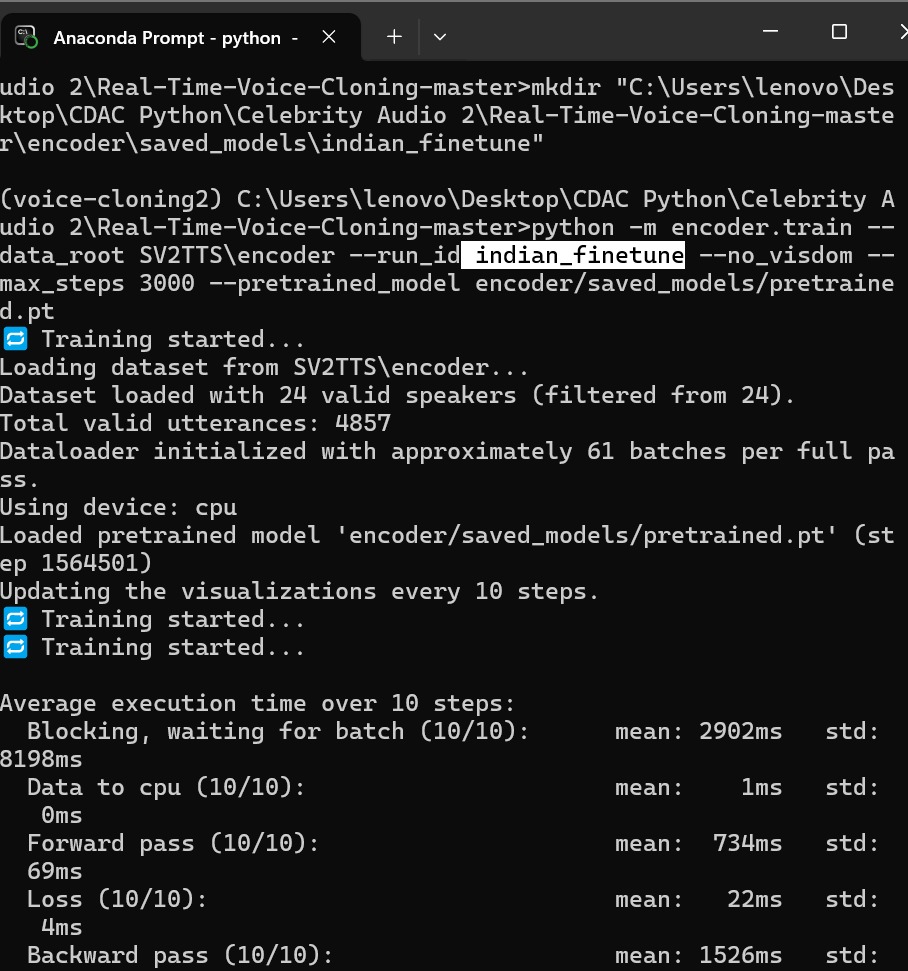
python demo\_toolbox.py -d <datasets\_root>

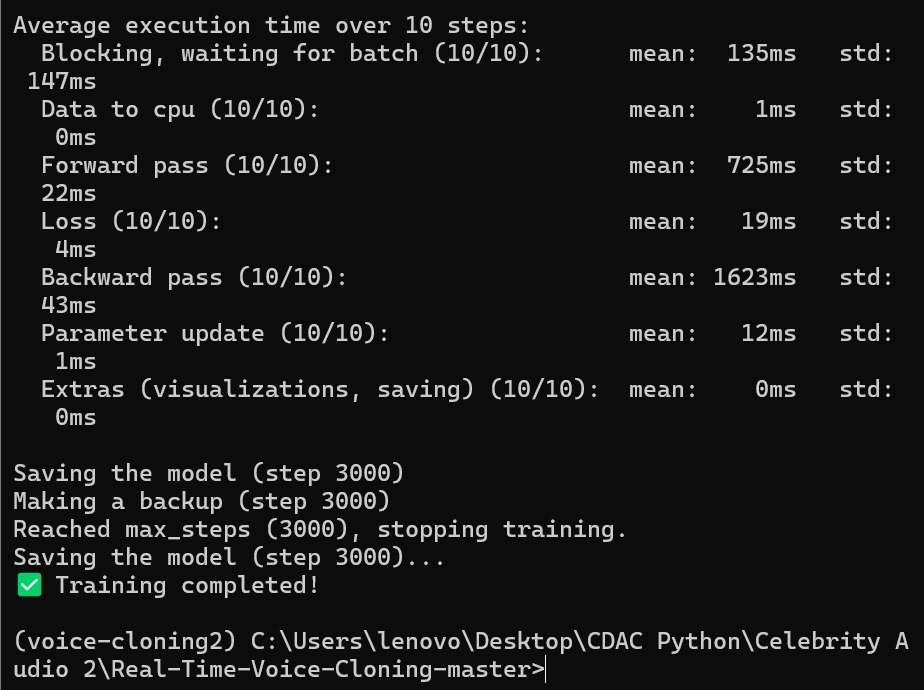
**Commands used and screenshots:**Encoder Training:

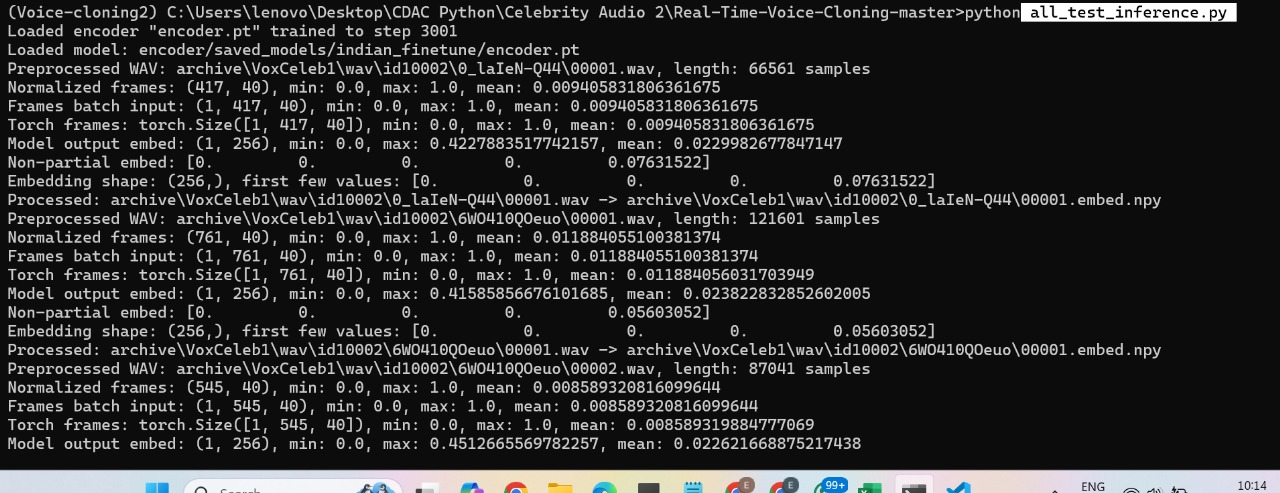
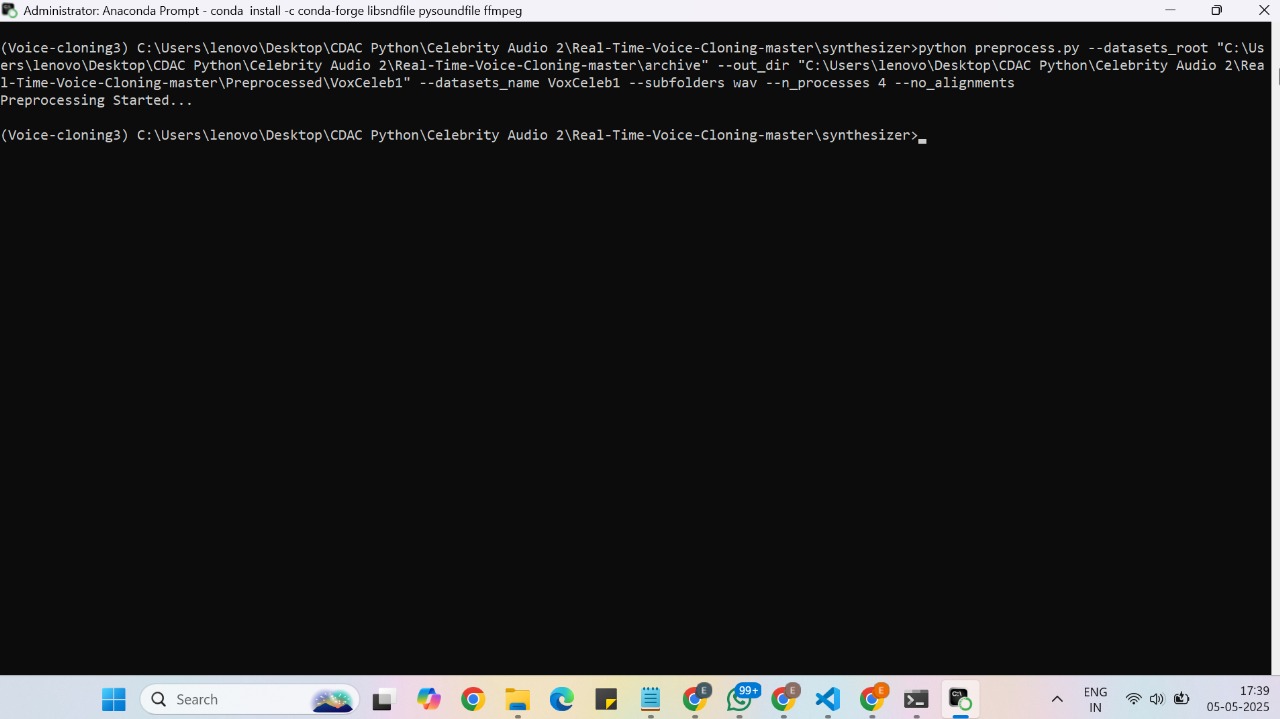


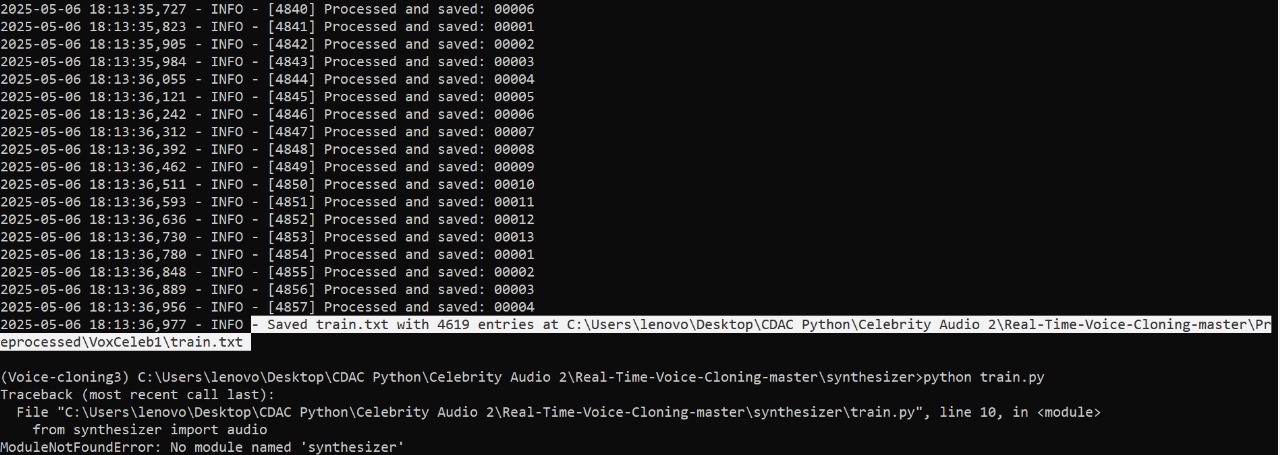


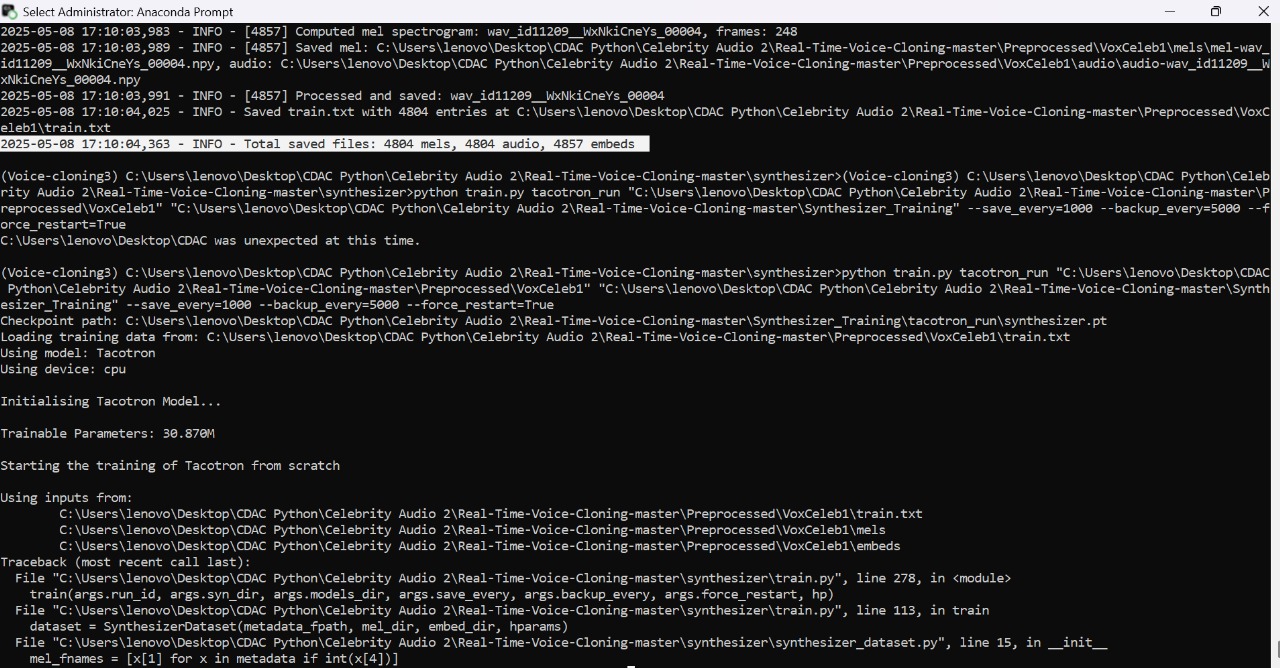
Encoder Inference:  
  


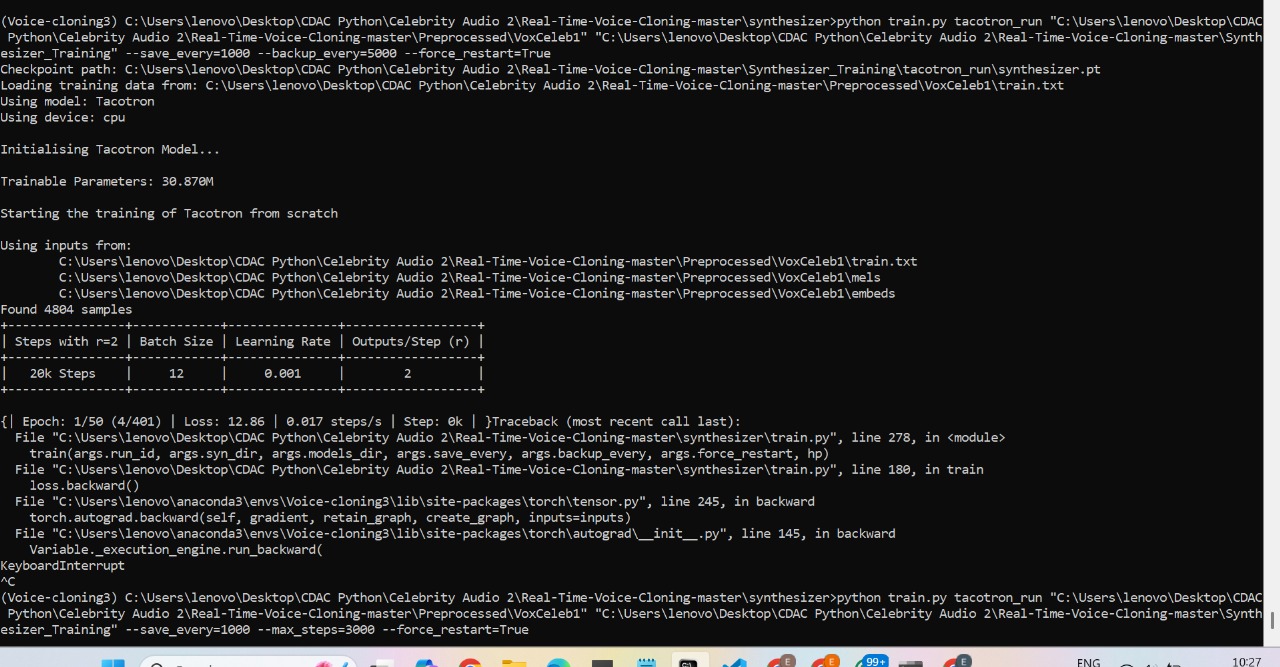
Encoder Indian Fine Tune:  
  


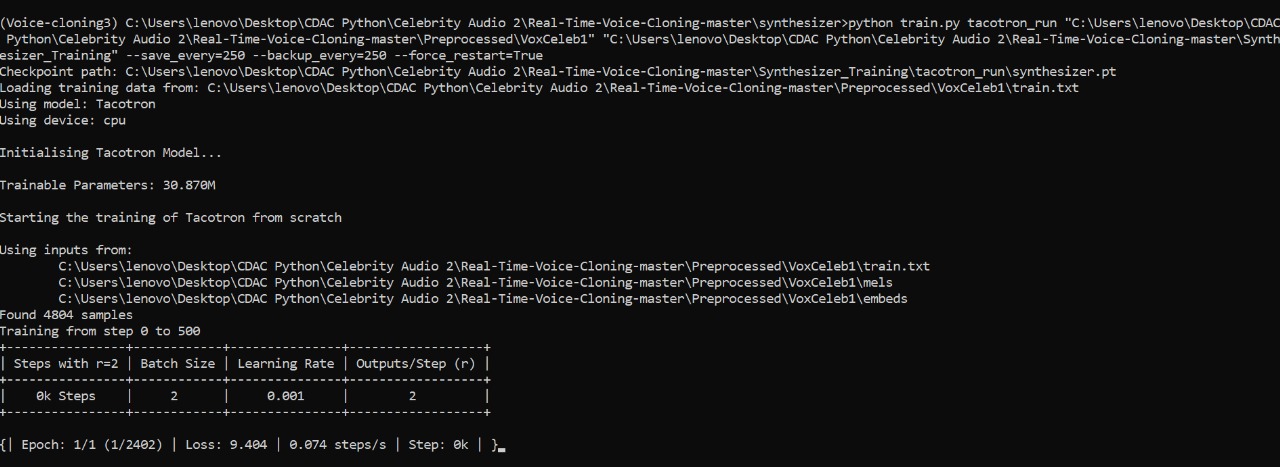


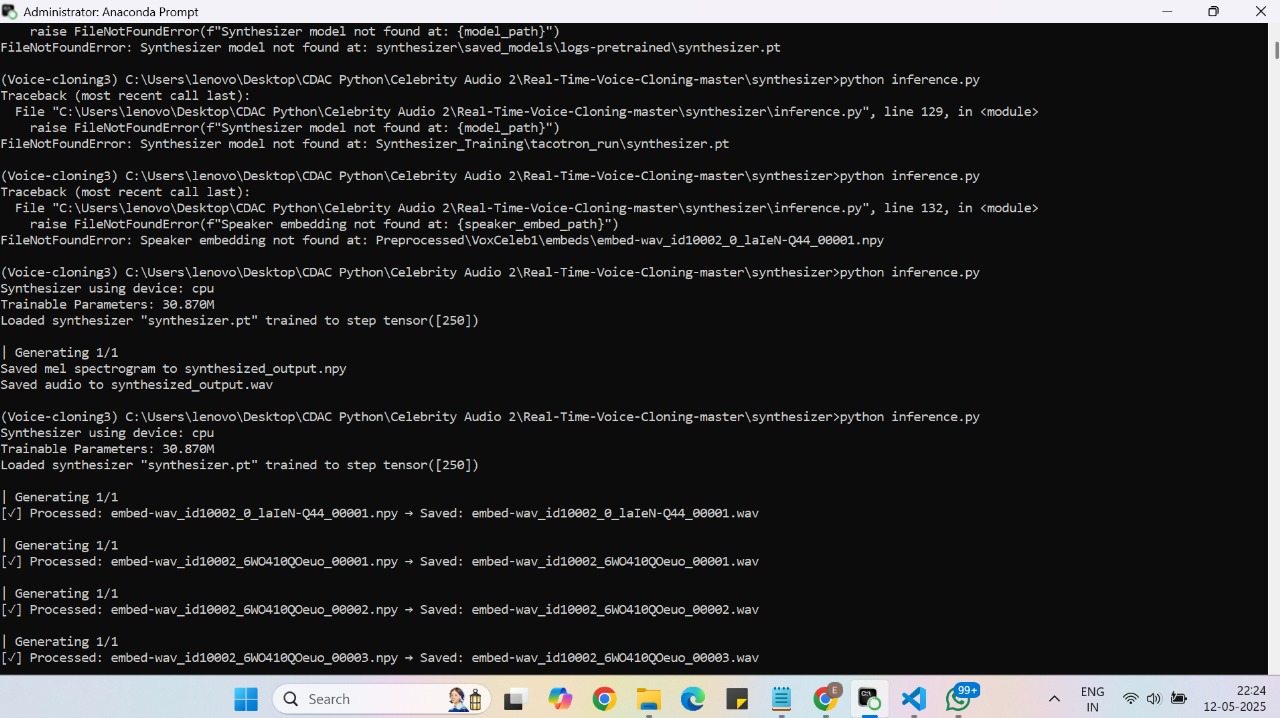
Encoder Final Inferencing:  
  
  
  
Synthesizer Preprocessing:  
  




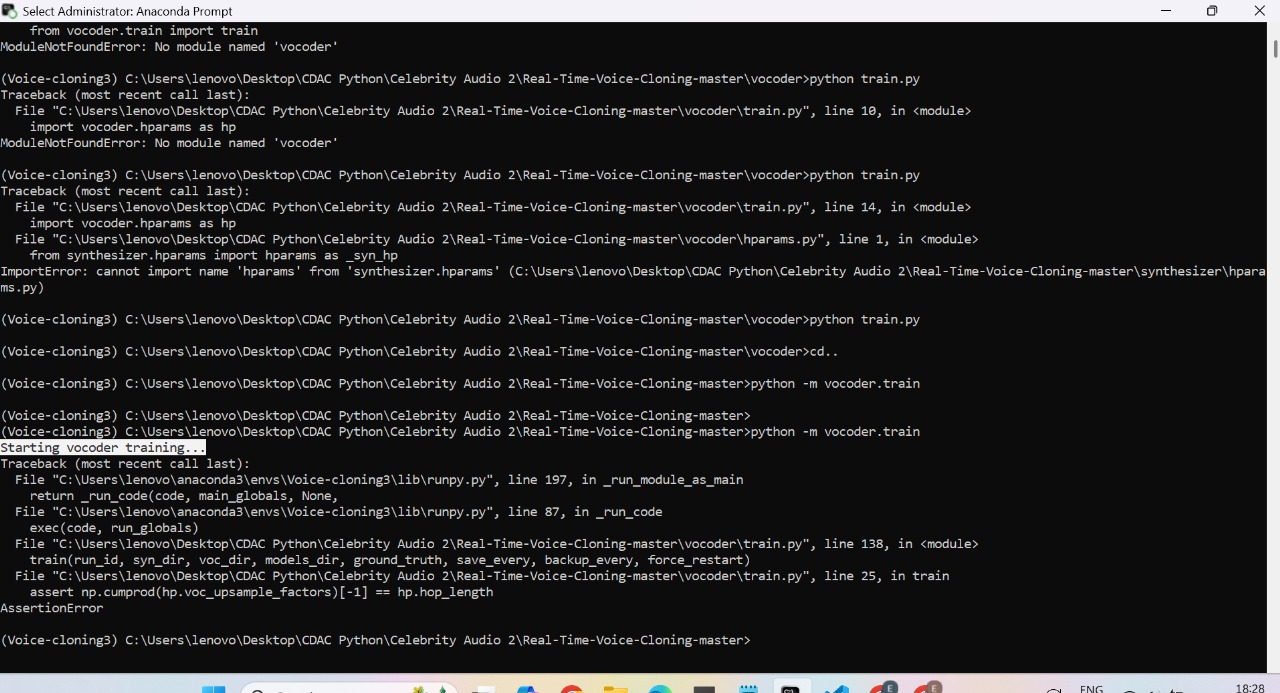
Synthesizer Training:  
  


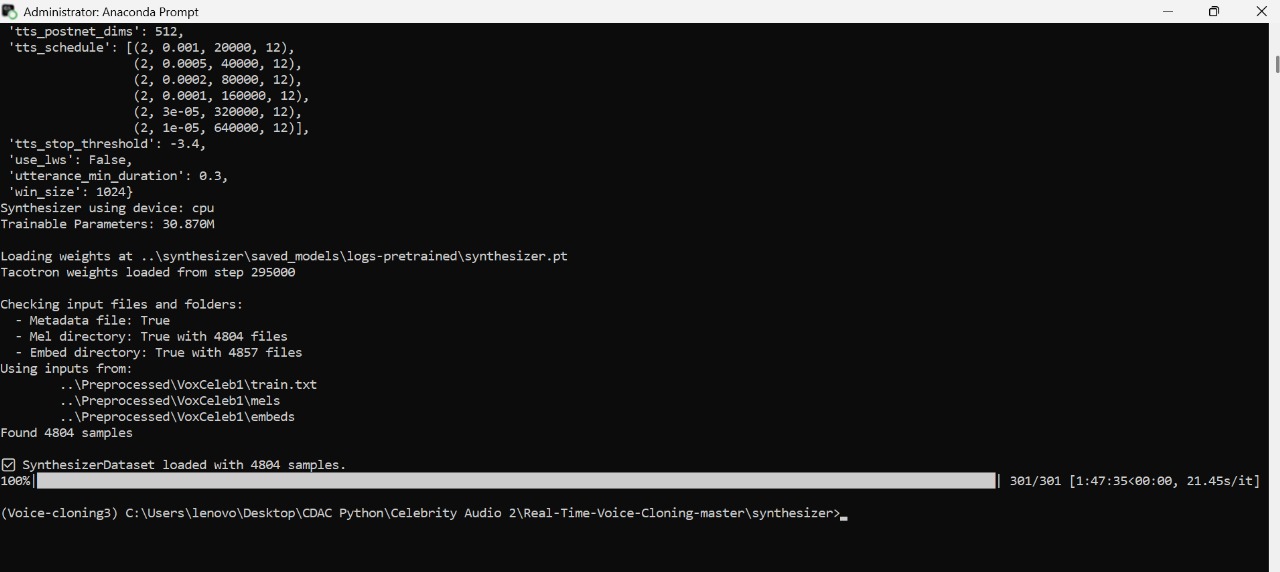


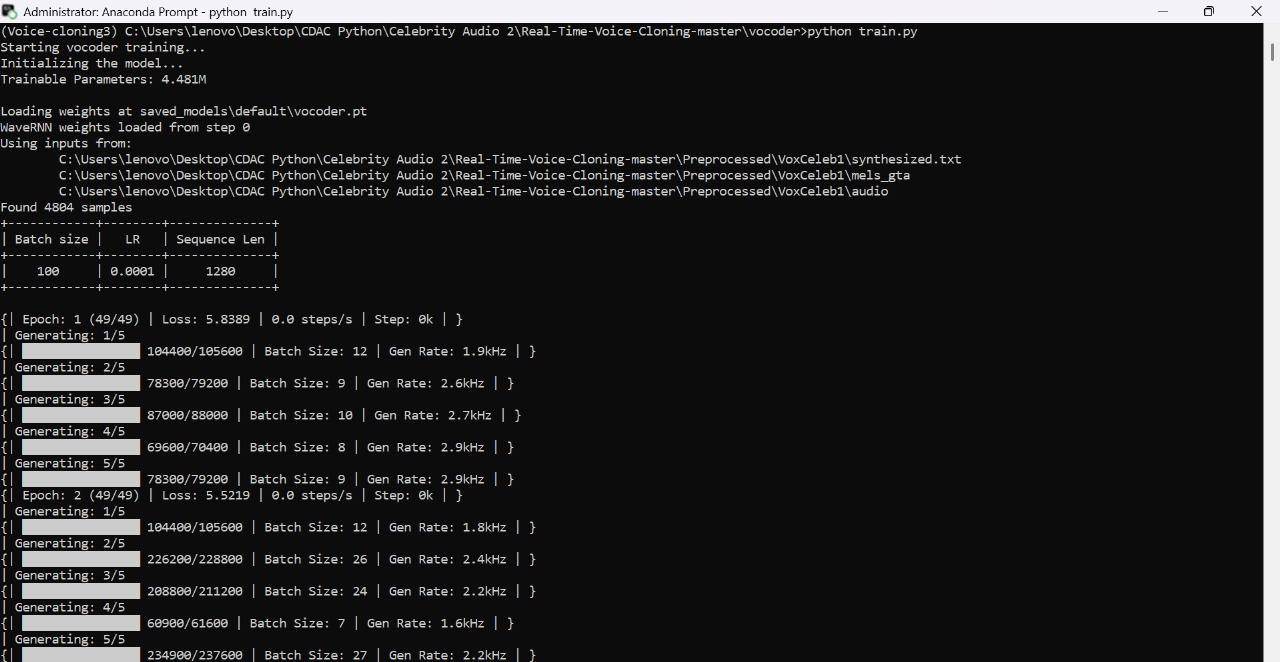


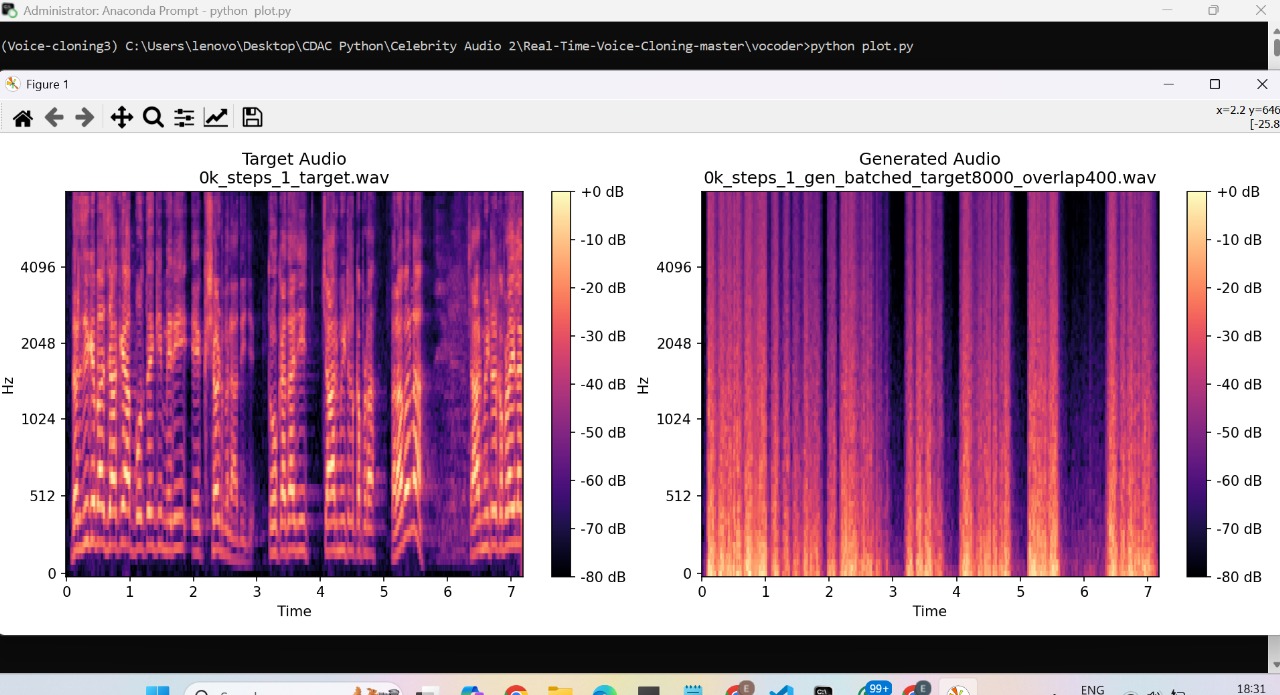
Synthesizer Inference:  
  


Vocoder Training:

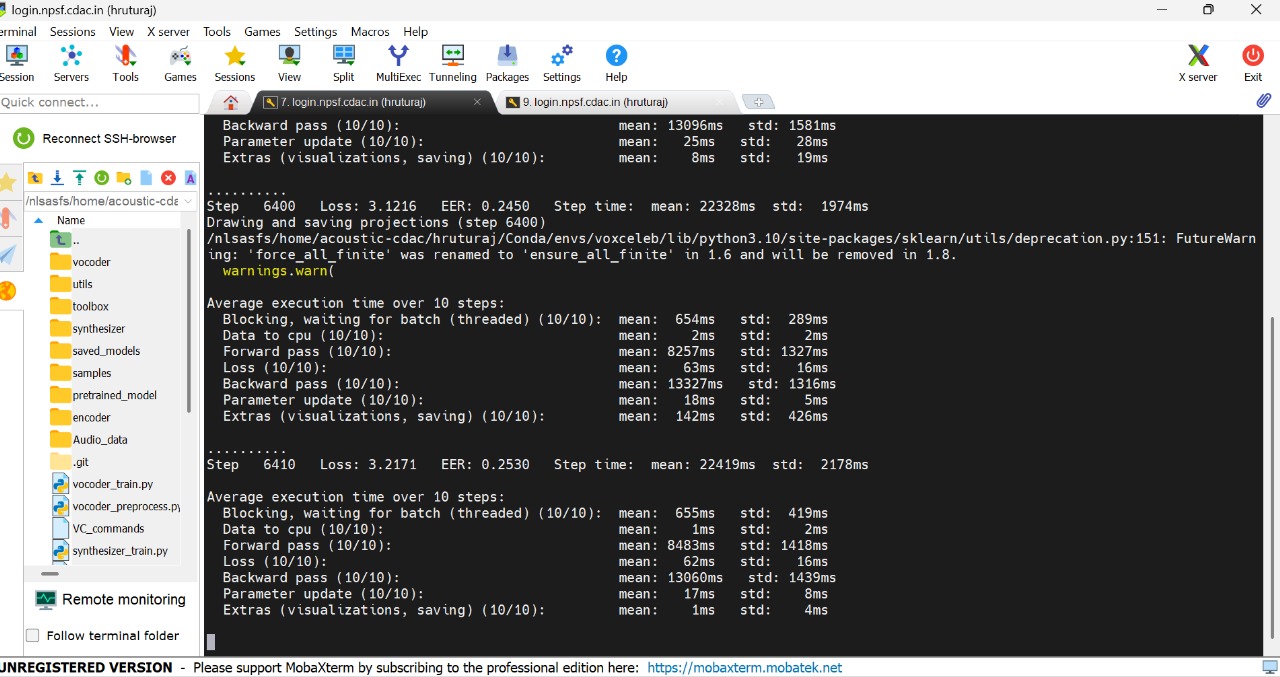


Running Synthesizer.py for vocoder training:  
  


Vocoder training:  
  


Vocoder sample voice plot for an audio:  
  
  
  
Doing Encoder training in MOBAXTERM:  
  
python encoder\_preprocess.py Audio\_data --out\_dir=Audio\_data/SV2TTS/encoder/ -s true

python encoder\_train.py encoder\_run\_1 Audio\_data/SV2TTS/encoder/



**Encoder:**

**Params\_data.py**

hparams = {

    ## Mel-filterbank

    "mel\_window\_length": 25,  # In milliseconds

    "mel\_window\_step": 10,    # In milliseconds

    "mel\_n\_channels": 40,

    "num\_mels": 40,

    ## Audio

    "sampling\_rate": 16000,

    "partials\_n\_frames": 160,     # 1600 ms

    "inference\_n\_frames": 80,     #  800 ms

    ## Voice Activation Detection (VAD)

    "vad\_window\_length": 30,  # In milliseconds

    "vad\_moving\_average\_width": 8,

    "vad\_max\_silence\_length": 6,

    ## Audio volume normalization

    "audio\_norm\_target\_dBFS": -30,

    ## Add these derived values 👇

    "fft\_size": 512,

    "win\_length": int(25 / 1000 \* 16000),  # 25 ms window -> 400 samples

    "hop\_length": int(10 / 1000 \* 16000),  # 10 ms hop -> 160 samples

}

# Expose needed values for import

partials\_n\_frames = hparams["partials\_n\_frames"]

inference\_n\_frames = hparams["inference\_n\_frames"]

**Params\_model.py**

## Model parameters

model\_hidden\_size = 256 #Size of hidden layers in the neural network.

model\_embedding\_size = 256 #Size of the output embedding vector representing speaker voice identity.

model\_num\_layers = 3

## Training parameters

learning\_rate\_init = 1e-4

speakers\_per\_batch = 64

utterances\_per\_speaker = 10

**config.py**  
  
librispeech\_datasets = {

    "train": {

        "clean": ["LibriSpeech/train-clean-100", "LibriSpeech/train-clean-360"], #higher-quality recordings for training.

        "other": ["LibriSpeech/train-other-500"] #noisier or more challenging audio samples.

    },

    "test": {

        "clean": ["LibriSpeech/test-clean"],

        "other": ["LibriSpeech/test-other"]

    },

    "dev": { #used for validation during training.

        "clean": ["LibriSpeech/dev-clean"],

        "other": ["LibriSpeech/dev-other"]

    },

}

libritts\_datasets = {

    "train": {

        "clean": ["LibriTTS/train-clean-100", "LibriTTS/train-clean-360"],

        "other": ["LibriTTS/train-other-500"]

    },

    "test": {

        "clean": ["LibriTTS/test-clean"],

        "other": ["LibriTTS/test-other"]

    },

    "dev": {

        "clean": ["LibriTTS/dev-clean"],

        "other": ["LibriTTS/dev-other"]

    },

}

voxceleb\_datasets = {

    "voxceleb1" : {

        "train": ["VoxCeleb1/wav"],

        "test": ["VoxCeleb1/test\_wav"]

    },

    "voxceleb2" : {

        "train": ["VoxCeleb2/dev/aac"],

        "test": ["VoxCeleb2/test\_wav"]

    }

}

other\_datasets = [

    "LJSpeech-1.1",

    "VCTK-Corpus/wav48",

]

anglophone\_nationalites = ["australia", "canada", "ireland", "uk", "usa"]

**Audio.py**from scipy.ndimage.morphology import binary\_dilation

from pathlib import Path

from typing import Optional, Union

from warnings import warn

import numpy as np

import librosa

import struct

try:

    import webrtcvad

except:

    warn("Unable to import 'webrtcvad'. This package enables noise removal and is recommended.")

    webrtcvad = None

int16\_max = (2 \*\* 15) - 1

audio\_norm\_target\_dBFS = -30

def preprocess\_wav(fpath\_or\_wav: Union[str, Path, np.ndarray],

                   source\_sr: Optional[int] = None,

                   normalize: Optional[bool] = True,

                   trim\_silence: Optional[bool] = True,

                   hparams: dict = None):

    """

    Applies the preprocessing operations used in training the Speaker Encoder to a waveform

    either on disk or in memory. The waveform will be resampled to match the data hyperparameters.

    :param fpath\_or\_wav: either a filepath to an audio file (many extensions are supported, not

    just .wav), or the waveform as a numpy array of floats.

    :param source\_sr: if passing an audio waveform, the sampling rate of the waveform before

    preprocessing. After preprocessing, the waveform's sampling rate will match the hparams.

    :param hparams: dictionary containing sampling\_rate and other parameters.

    """

    if hparams is None:

        raise ValueError("hparams must be provided with sampling\_rate defined.")

    # Load the wav from disk if needed

    if isinstance(fpath\_or\_wav, str) or isinstance(fpath\_or\_wav, Path):

        wav, source\_sr = librosa.load(str(fpath\_or\_wav), sr=None)

    else:

        wav = fpath\_or\_wav

    # Resample the wav if needed

    if source\_sr is not None and source\_sr != hparams["sampling\_rate"]:

        wav = librosa.resample(wav, source\_sr, hparams["sampling\_rate"])

    # Apply the preprocessing: normalize volume and shorten long silences

    if normalize:

        wav = normalize\_volume(wav, audio\_norm\_target\_dBFS, increase\_only=True)

    if webrtcvad and trim\_silence:

        wav = trim\_long\_silences(wav, hparams)

    return wav

def wav\_to\_mel\_spectrogram(wav, hparams: dict):

    """

    Derives a mel spectrogram ready to be used by the encoder from a preprocessed audio waveform.

    Note: this is not a log-mel spectrogram.

    """

    frames = librosa.feature.melspectrogram(

        y=wav,

        sr=hparams["sampling\_rate"],

        n\_fft=hparams["fft\_size"],

        hop\_length=hparams["hop\_length"],

        win\_length=hparams["win\_length"],

        n\_mels=hparams["num\_mels"]

    )

    return frames.astype(np.float32).T

def trim\_long\_silences(wav, hparams: dict):

    """

    Ensures that segments without voice in the waveform remain no longer than a

    threshold determined by the VAD parameters in hparams.

    :param wav: the raw waveform as a numpy array of floats

    :param hparams: dictionary containing VAD parameters and sampling\_rate

    :return: the same waveform with silences trimmed away (length <= original wav length)

    """

    # Compute the voice detection window size

    samples\_per\_window = (hparams["vad\_window\_length"] \* hparams["sampling\_rate"]) // 1000

    # Trim the end of the audio to have a multiple of the window size

    wav = wav[:len(wav) - (len(wav) % samples\_per\_window)]

    # Convert the float waveform to 16-bit mono PCM

    pcm\_wave = struct.pack("%dh" % len(wav), \*(np.round(wav \* int16\_max)).astype(np.int16))

    # Perform voice activation detection

    voice\_flags = []

    vad = webrtcvad.Vad(mode=3)

    for window\_start in range(0, len(wav), samples\_per\_window):

        window\_end = window\_start + samples\_per\_window

        voice\_flags.append(vad.is\_speech(pcm\_wave[window\_start \* 2:window\_end \* 2],

                                        sample\_rate=hparams["sampling\_rate"]))

    voice\_flags = np.array(voice\_flags)

    # Smooth the voice detection with a moving average

    def moving\_average(array, width):

        array\_padded = np.concatenate((np.zeros((width - 1) // 2), array, np.zeros(width // 2)))

        ret = np.cumsum(array\_padded, dtype=float)

        ret[width:] = ret[width:] - ret[:-width]

        return ret[width - 1:] / width

    audio\_mask = moving\_average(voice\_flags, hparams["vad\_moving\_average\_width"])

    audio\_mask = np.round(audio\_mask).astype(np.bool\_)

    # Dilate the voiced regions

    audio\_mask = binary\_dilation(audio\_mask, np.ones(hparams["vad\_max\_silence\_length"] + 1))

    audio\_mask = np.repeat(audio\_mask, samples\_per\_window)

    return wav[audio\_mask == True]

def normalize\_volume(wav, target\_dBFS, increase\_only=False, decrease\_only=False):

    if increase\_only and decrease\_only:

        raise ValueError("Both increase only and decrease only are set")

    dBFS\_change = target\_dBFS - 10 \* np.log10(np.mean(wav \*\* 2))

    if (dBFS\_change < 0 and increase\_only) or (dBFS\_change > 0 and decrease\_only):

        return wav

    return wav \* (10 \*\* (dBFS\_change / 20))

**Preprocess.py**

print("Starting preprocessing script...")

import sys

import os

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

from datetime import datetime

from functools import partial

from multiprocessing import Pool

from pathlib import Path

import argparse

import numpy as np

from tqdm import tqdm

from encoder import audio

from encoder.config import librispeech\_datasets, anglophone\_nationalites

from encoder.params\_data import hparams  # Import hparams directly

\_AUDIO\_EXTENSIONS = ("wav", "flac", "m4a", "mp3")

class DatasetLog:

    def \_\_init\_\_(self, root, name):

        self.text\_file = open(Path(root, "Log\_%s.txt" % name.replace("/", "\_")), "w")

        self.sample\_data = dict()

        start\_time = str(datetime.now().strftime("%A %d %B %Y at %H:%M"))

        self.write\_line("Creating dataset %s on %s" % (name, start\_time))

        self.write\_line("-----")

        self.\_log\_params()

    def \_log\_params(self):

        self.write\_line("Parameter values:")

        for param\_name, value in hparams.items():

            self.write\_line("\t%s: %s" % (param\_name, value))

        self.write\_line("-----")

    def write\_line(self, line):

        self.text\_file.write("%s\n" % line)

    def add\_sample(self, \*\*kwargs):

        for param\_name, value in kwargs.items():

            if param\_name not in self.sample\_data:

                self.sample\_data[param\_name] = []

            self.sample\_data[param\_name].append(value)

    def finalize(self):

        self.write\_line("Statistics:")

        for param\_name, values in self.sample\_data.items():

            self.write\_line("\t%s:" % param\_name)

            self.write\_line("\t\tmin %.3f, max %.3f" % (np.min(values), np.max(values)))

            self.write\_line("\t\tmean %.3f, median %.3f" % (np.mean(values), np.median(values)))

        self.write\_line("-----")

        end\_time = str(datetime.now().strftime("%A %d %B %Y at %H:%M"))

        self.write\_line("Finished on %s" % end\_time)

        self.text\_file.close()

def \_init\_preprocess\_dataset(dataset\_name, datasets\_root, out\_dir) -> (Path, DatasetLog):

    dataset\_root = datasets\_root.joinpath(dataset\_name)

    if not dataset\_root.exists():

        print("Couldn\'t find %s, skipping this dataset." % dataset\_root)

        return None, None

    return dataset\_root, DatasetLog(out\_dir, dataset\_name)

def \_preprocess\_speaker(speaker\_dir: Path, datasets\_root: Path, out\_dir: Path, skip\_existing: bool):

    speaker\_name = "\_".join(speaker\_dir.relative\_to(datasets\_root).parts)

    speaker\_out\_dir = out\_dir.joinpath(speaker\_name)

    speaker\_out\_dir.mkdir(exist\_ok=True)

    sources\_fpath = speaker\_out\_dir.joinpath("\_sources.txt")

    if sources\_fpath.exists():

        try:

            with sources\_fpath.open("r") as sources\_file:

                existing\_fnames = {line.split(",")[0] for line in sources\_file}

        except:

            existing\_fnames = {}

    else:

        existing\_fnames = {}

    sources\_file = sources\_fpath.open("a" if skip\_existing else "w")

    audio\_durs = []

    for extension in \_AUDIO\_EXTENSIONS:

        for yt\_id\_dir in speaker\_dir.glob("\*"):

            if not yt\_id\_dir.is\_dir():

                continue

            for in\_fpath in yt\_id\_dir.glob("\*\*/\*.%s" % extension):

                out\_fname = "\_".join(in\_fpath.relative\_to(speaker\_dir).parts)

                out\_fname = out\_fname.replace(".%s" % extension, ".npy")

                if skip\_existing and out\_fname in existing\_fnames:

                    continue

                # Pass hparams instead of sampling\_rate

                wav = audio.preprocess\_wav(in\_fpath, hparams=hparams)

                if len(wav) == 0:

                    continue

                frames = audio.wav\_to\_mel\_spectrogram(wav, hparams=hparams)

                if len(frames) < hparams["partials\_n\_frames"]:

                    continue

                out\_fpath = speaker\_out\_dir.joinpath(out\_fname)

                np.save(out\_fpath, frames)

                sources\_file.write("%s,%s\n" % (out\_fname, in\_fpath))

                audio\_durs.append(len(wav) / hparams["sampling\_rate"])

    sources\_file.close()

    return audio\_durs

def \_preprocess\_speaker\_dirs(speaker\_dirs, dataset\_name, datasets\_root, out\_dir, skip\_existing, logger, target\_sampling\_rate):

    print("%s: Preprocessing data for %d speakers with sampling rate %d Hz." % (dataset\_name, len(speaker\_dirs), target\_sampling\_rate))

    work\_fn = partial(\_preprocess\_speaker, datasets\_root=datasets\_root, out\_dir=out\_dir, skip\_existing=skip\_existing)

    with Pool(4) as pool:

        tasks = pool.imap(work\_fn, speaker\_dirs)

        for sample\_durs in tqdm(tasks, dataset\_name, len(speaker\_dirs), unit="speakers"):

            for sample\_dur in sample\_durs:

                logger.add\_sample(duration=sample\_dur)

    logger.finalize()

    print("Done preprocessing %s.\n" % dataset\_name)

def preprocess\_librispeech(datasets\_root: Path, out\_dir: Path, skip\_existing=False, sampling\_rate=16000):

    for dataset\_name in librispeech\_datasets["train"]["other"]:

        dataset\_root, logger = \_init\_preprocess\_dataset(dataset\_name, datasets\_root, out\_dir)

        if not dataset\_root:

            return

        speaker\_dirs = list(dataset\_root.glob("\*"))

        \_preprocess\_speaker\_dirs(speaker\_dirs, dataset\_name, datasets\_root, out\_dir, skip\_existing, logger, sampling\_rate)

def preprocess\_voxceleb1(datasets\_root: Path, out\_dir: Path, skip\_existing=False, sampling\_rate=22050):

    dataset\_name = "VoxCeleb1"

    dataset\_root, logger = \_init\_preprocess\_dataset(dataset\_name, datasets\_root, out\_dir)

    if not dataset\_root:

        return

    meta\_file\_path = dataset\_root.joinpath("vox1\_meta.csv")

    if meta\_file\_path.exists():

        with meta\_file\_path.open("r") as metafile:

            metadata = [line.split("\t") for line in metafile][1:]

            nationalities = {line[0]: line[3] for line in metadata}

            keep\_speaker\_ids = [speaker\_id for speaker\_id, nationality in nationalities.items() if

                                nationality.lower() in anglophone\_nationalites]

            print("VoxCeleb1: using samples from %d (presumed anglophone) speakers out of %d." %

                  (len(keep\_speaker\_ids), len(nationalities)))

    else:

        print("VoxCeleb1: No meta file found, processing all speakers.")

        keep\_speaker\_ids = None

    wav\_path = dataset\_root.joinpath("wav")

    speaker\_dirs = list(wav\_path.glob("\*"))

    if keep\_speaker\_ids:

        speaker\_dirs = [d for d in speaker\_dirs if d.name in keep\_speaker\_ids]

    print("VoxCeleb1: found %d speakers on the disk." % len(speaker\_dirs))

    \_preprocess\_speaker\_dirs(speaker\_dirs, dataset\_name, datasets\_root, out\_dir, skip\_existing, logger, sampling\_rate)

def preprocess\_voxceleb2(datasets\_root: Path, out\_dir: Path, skip\_existing=False, sampling\_rate=16000):

    dataset\_name = "VoxCeleb2"

    dataset\_root, logger = \_init\_preprocess\_dataset(dataset\_name, datasets\_root, out\_dir)

    if not dataset\_root:

        return

    speaker\_dirs = list(dataset\_root.joinpath("dev", "aac").glob("\*"))

    \_preprocess\_speaker\_dirs(speaker\_dirs, dataset\_name, datasets\_root, out\_dir, skip\_existing, logger, sampling\_rate)

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Preprocesses audio files for encoder training.")

    parser.add\_argument("datasets\_root", type=str, help="Path to the dataset root")

    parser.add\_argument("--datasets", type=str, nargs="+", default=["VoxCeleb1"], help="List of datasets to process")

    parser.add\_argument("--sampling\_rate", type=int, choices=[16000, 22050], default=22050, help="Target sampling rate (16000 or 22050 Hz)")

    parser.add\_argument("--skip\_existing", action="store\_true", help="Skip existing processed files")

    parser.add\_argument("--no\_cuda", action="store\_true", help="Disable CUDA")

    args = parser.parse\_args()

    # Update hparams with the chosen sampling rate

    hparams["sampling\_rate"] = args.sampling\_rate

    hparams["win\_length"] = int(hparams["mel\_window\_length"] / 1000 \* args.sampling\_rate)

    hparams["hop\_length"] = int(hparams["mel\_window\_step"] / 1000 \* args.sampling\_rate)

    out\_dir = Path("SV2TTS", "encoder")

    out\_dir.mkdir(parents=True, exist\_ok=True)

    for dataset in args.datasets:

        if dataset == "VoxCeleb1":

            preprocess\_voxceleb1(Path(args.datasets\_root), out\_dir, args.skip\_existing, args.sampling\_rate)

        elif dataset == "VoxCeleb2":

            preprocess\_voxceleb2(Path(args.datasets\_root), out\_dir, args.skip\_existing, args.sampling\_rate)

        elif dataset in librispeech\_datasets["train"]["other"]:

            preprocess\_librispeech(Path(args.datasets\_root), out\_dir, args.skip\_existing, args.sampling\_rate)

        else:

            print(f"Dataset {dataset} not recognized, skipping.")

**Model.py**from encoder.params\_model import \*

from encoder.params\_data import \*

from encoder.params\_data import hparams

from scipy.interpolate import interp1d

from sklearn.metrics import roc\_curve

from torch.nn.utils import clip\_grad\_norm\_

from scipy.optimize import brentq

from torch import nn

import numpy as np

import torch

class SpeakerEncoder(nn.Module):

    def \_\_init\_\_(self, device, loss\_device):

        super().\_\_init\_\_()

        self.loss\_device = loss\_device

        # Network defition

        self.lstm = nn.LSTM(input\_size=hparams["mel\_n\_channels"],

                            hidden\_size=model\_hidden\_size,

                            num\_layers=model\_num\_layers,

                            batch\_first=True).to(device)

        self.linear = nn.Linear(in\_features=model\_hidden\_size,

                                out\_features=model\_embedding\_size).to(device)

        self.relu = torch.nn.ReLU().to(device)

        # Cosine similarity scaling (with fixed initial parameter values)

        self.similarity\_weight = nn.Parameter(torch.tensor([10.])).to(loss\_device)

        self.similarity\_bias = nn.Parameter(torch.tensor([-5.])).to(loss\_device)

        # Loss

        self.loss\_fn = nn.CrossEntropyLoss().to(loss\_device)

    def do\_gradient\_ops(self):

        # Gradient scale

        self.similarity\_weight.grad \*= 0.01

        self.similarity\_bias.grad \*= 0.01

        # Gradient clipping

        clip\_grad\_norm\_(self.parameters(), 3, norm\_type=2)

    def forward(self, utterances, hidden\_init=None):

        """

        Computes the embeddings of a batch of utterance spectrograms.

        :param utterances: batch of mel-scale filterbanks of same duration as a tensor of shape

        (batch\_size, n\_frames, n\_channels)

        :param hidden\_init: initial hidden state of the LSTM as a tensor of shape (num\_layers,

        batch\_size, hidden\_size). Will default to a tensor of zeros if None.

        :return: the embeddings as a tensor of shape (batch\_size, embedding\_size)

        """

        # Pass the input through the LSTM layers and retrieve all outputs, the final hidden state

        # and the final cell state.

        out, (hidden, cell) = self.lstm(utterances, hidden\_init)

        # We take only the hidden state of the last layer

        embeds\_raw = self.relu(self.linear(hidden[-1]))

        # L2-normalize it

        embeds = embeds\_raw / (torch.norm(embeds\_raw, dim=1, keepdim=True) + 1e-5)

        return embeds

    def similarity\_matrix(self, embeds):

        """

        Computes the similarity matrix according the section 2.1 of GE2E.

        :param embeds: the embeddings as a tensor of shape (speakers\_per\_batch,

        utterances\_per\_speaker, embedding\_size)

        :return: the similarity matrix as a tensor of shape (speakers\_per\_batch,

        utterances\_per\_speaker, speakers\_per\_batch)

        """

        speakers\_per\_batch, utterances\_per\_speaker = embeds.shape[:2]

        # Inclusive centroids (1 per speaker). Cloning is needed for reverse differentiation

        centroids\_incl = torch.mean(embeds, dim=1, keepdim=True)

        centroids\_incl = centroids\_incl.clone() / (torch.norm(centroids\_incl, dim=2, keepdim=True) + 1e-5)

        # Exclusive centroids (1 per utterance)

        centroids\_excl = (torch.sum(embeds, dim=1, keepdim=True) - embeds)

        centroids\_excl /= (utterances\_per\_speaker - 1)

        centroids\_excl = centroids\_excl.clone() / (torch.norm(centroids\_excl, dim=2, keepdim=True) + 1e-5)

        # Similarity matrix. The cosine similarity of already 2-normed vectors is simply the dot

        # product of these vectors (which is just an element-wise multiplication reduced by a sum).

        # We vectorize the computation for efficiency.

        sim\_matrix = torch.zeros(speakers\_per\_batch, utterances\_per\_speaker,

                                 speakers\_per\_batch).to(self.loss\_device)

        mask\_matrix = 1 - np.eye(speakers\_per\_batch, dtype=np.int)

        for j in range(speakers\_per\_batch):

            mask = np.where(mask\_matrix[j])[0]

            sim\_matrix[mask, :, j] = (embeds[mask] \* centroids\_incl[j]).sum(dim=2)

            sim\_matrix[j, :, j] = (embeds[j] \* centroids\_excl[j]).sum(dim=1)

        ## Even more vectorized version (slower maybe because of transpose)

        # sim\_matrix2 = torch.zeros(speakers\_per\_batch, speakers\_per\_batch, utterances\_per\_speaker

        #                           ).to(self.loss\_device)

        # eye = np.eye(speakers\_per\_batch, dtype=np.int)

        # mask = np.where(1 - eye)

        # sim\_matrix2[mask] = (embeds[mask[0]] \* centroids\_incl[mask[1]]).sum(dim=2)

        # mask = np.where(eye)

        # sim\_matrix2[mask] = (embeds \* centroids\_excl).sum(dim=2)

        # sim\_matrix2 = sim\_matrix2.transpose(1, 2)

        sim\_matrix = sim\_matrix \* self.similarity\_weight + self.similarity\_bias

        return sim\_matrix

    def loss(self, embeds):

        """

        Computes the softmax loss according the section 2.1 of GE2E.

        :param embeds: the embeddings as a tensor of shape (speakers\_per\_batch,

        utterances\_per\_speaker, embedding\_size)

        :return: the loss and the EER for this batch of embeddings.

        """

        speakers\_per\_batch, utterances\_per\_speaker = embeds.shape[:2]

        # Loss

        sim\_matrix = self.similarity\_matrix(embeds)

        sim\_matrix = sim\_matrix.reshape((speakers\_per\_batch \* utterances\_per\_speaker,

                                         speakers\_per\_batch))

        ground\_truth = np.repeat(np.arange(speakers\_per\_batch), utterances\_per\_speaker)

        target = torch.from\_numpy(ground\_truth).long().to(self.loss\_device)

        loss = self.loss\_fn(sim\_matrix, target)

        # EER (not backpropagated)

        with torch.no\_grad():

            inv\_argmax = lambda i: np.eye(1, speakers\_per\_batch, i, dtype=np.int)[0]

            labels = np.array([inv\_argmax(i) for i in ground\_truth])

            preds = sim\_matrix.detach().cpu().numpy()

            # Snippet from https://yangcha.github.io/EER-ROC/

            fpr, tpr, thresholds = roc\_curve(labels.flatten(), preds.flatten())

            eer = brentq(lambda x: 1. - x - interp1d(fpr, tpr)(x), 0., 1.)

        return loss, eer

**Train.py**from pathlib import Path

import argparse

import os

import torch

from encoder.data\_objects import SpeakerVerificationDataLoader, SpeakerVerificationDataset

from encoder.model import SpeakerEncoder

from encoder.params\_model import \*

from encoder.visualizations import Visualizations

from utils.profiler import Profiler

# Print only from the main process

if os.environ.get('RANK', '0') == '0':

    print("🔁 Training started...")

def sync(device: torch.device):

    if device.type == "cuda":

        torch.cuda.synchronize(device)

def train(run\_id: str, clean\_data\_root: Path, models\_dir: Path, umap\_every: int, save\_every: int,

          backup\_every: int, vis\_every: int, force\_restart: bool, visdom\_server: str,

          no\_visdom: bool, max\_steps: int, pretrained\_model: str):

    main\_process = os.environ.get('RANK', '0') == '0'

    if main\_process:

        print(f"Loading dataset from {clean\_data\_root}...")

    # Create dataset and filter speakers

    try:

        dataset = SpeakerVerificationDataset(clean\_data\_root)

        speaker\_dirs = [d for d in clean\_data\_root.iterdir() if d.is\_dir()]

        speaker\_utterances = {d.name: len(list(d.glob("\*.npy"))) for d in speaker\_dirs}

        min\_utterances = 5

        valid\_speakers = [s for s, count in speaker\_utterances.items() if count >= min\_utterances]

        dataset.speakers = valid\_speakers

        if not valid\_speakers:

            if main\_process:

                print(f"No speakers found with at least {min\_utterances} utterances. Training aborted.")

            return

        total\_utterances = sum(speaker\_utterances[s] for s in valid\_speakers)

        if main\_process:

            print(f"Dataset loaded with {len(valid\_speakers)} valid speakers (filtered from {len(speaker\_dirs)}).")

            print(f"Total valid utterances: {total\_utterances}")

    except Exception as e:

        if main\_process:

            print(f"Failed to load dataset: {e}")

        return

    # Adjust batch sizes

    speakers\_per\_batch = min(16, len(valid\_speakers))

    utterances\_per\_speaker = 5

    loader = SpeakerVerificationDataLoader(

        dataset,

        speakers\_per\_batch=speakers\_per\_batch,

        utterances\_per\_speaker=utterances\_per\_speaker,

        num\_workers=2,

    )

    if main\_process:

        num\_batches = (total\_utterances + speakers\_per\_batch \* utterances\_per\_speaker - 1) // (speakers\_per\_batch \* utterances\_per\_speaker)

        print(f"Dataloader initialized with approximately {num\_batches} batches per full pass.")

    # Setup device

    device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

    loss\_device = torch.device("cpu")

    if main\_process:

        print(f"Using device: {device}")

    # Create model and optimizer

    model = SpeakerEncoder(device, loss\_device)

    optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate\_init)

    init\_step = 1

    # Load pretrained model if specified

    state\_fpath = models\_dir / run\_id / "encoder.pt"

    if pretrained\_model:

        checkpoint = torch.load(pretrained\_model, map\_location=device, weights\_only=True)

        model.load\_state\_dict(checkpoint["model\_state"])

        init\_step = 1

        if main\_process:

            print(f"Loaded pretrained model '{pretrained\_model}' (step {checkpoint['step']})")

    elif not force\_restart and state\_fpath.exists():

        if main\_process:

            print(f"Found existing model '{run\_id}', loading it and resuming training.")

        checkpoint = torch.load(state\_fpath)

        init\_step = checkpoint["step"]

        model.load\_state\_dict(checkpoint["model\_state"])

        optimizer.load\_state\_dict(checkpoint["optimizer\_state"])

        optimizer.param\_groups[0]["lr"] = learning\_rate\_init

    else:

        if main\_process:

            print(f"No model '{run\_id}' found or force\_restart is True, starting from scratch.")

    model.train()

    # Initialize visualizations

    vis = Visualizations(run\_id, vis\_every, server=visdom\_server, disabled=no\_visdom)

    if not no\_visdom and main\_process:

        vis.log\_dataset(dataset)

        vis.log\_params()

        device\_name = str(torch.cuda.get\_device\_name(0) if torch.cuda.is\_available() else "CPU")

        vis.log\_implementation({"Device": device\_name})

    # Training loop

    profiler = Profiler(summarize\_every=10, disabled=False)

    try:

        for step, speaker\_batch in enumerate(loader, init\_step):

            if step > max\_steps:

                if main\_process:

                    print(f"Reached max\_steps ({max\_steps}), stopping training.")

                    print(f"Saving the model (step {step - 1})...")

                    torch.save({

                        "step": step,

                        "model\_state": model.state\_dict(),

                        "optimizer\_state": optimizer.state\_dict(),

                    }, state\_fpath)

                break

            profiler.tick("Blocking, waiting for batch")

            # Forward pass

            inputs = torch.from\_numpy(speaker\_batch.data).to(device)

            sync(device)

            profiler.tick("Data to %s" % device)

            embeds = model(inputs)

            sync(device)

            profiler.tick("Forward pass")

            embeds\_loss = embeds.view((speakers\_per\_batch, utterances\_per\_speaker, -1)).to(loss\_device)

            loss, eer = model.loss(embeds\_loss)

            sync(loss\_device)

            profiler.tick("Loss")

            # Backward pass

            model.zero\_grad()

            loss.backward()

            profiler.tick("Backward pass")

            model.do\_gradient\_ops()

            optimizer.step()

            profiler.tick("Parameter update")

            # Update visualizations

            if not no\_visdom and main\_process:

                vis.update(loss.item(), eer, step)

                if umap\_every != 0 and step % umap\_every == 0:

                    print(f"Drawing and saving projections (step {step})")

                    projection\_fpath = models\_dir / run\_id / f"umap\_{step:06d}.png"

                    embeds = embeds.detach().cpu().numpy()

                    vis.draw\_projections(embeds, utterances\_per\_speaker, step, projection\_fpath)

                    vis.save()

            # Save model

            if save\_every != 0 and step % save\_every == 0 and main\_process:

                print(f"Saving the model (step {step})")

                torch.save({

                    "step": step + 1,

                    "model\_state": model.state\_dict(),

                    "optimizer\_state": optimizer.state\_dict(),

                }, state\_fpath)

            # Backup

            if backup\_every != 0 and step % backup\_every == 0 and main\_process:

                print(f"Making a backup (step {step})")

                backup\_fpath = models\_dir / run\_id / f"encoder\_{step:06d}.bak"

                torch.save({

                    "step": step + 1,

                    "model\_state": model.state\_dict(),

                    "optimizer\_state": optimizer.state\_dict(),

                }, backup\_fpath)

            profiler.tick("Extras (visualizations, saving)")

        if main\_process and step <= max\_steps:

            print("✅ Training completed (reached end of data before max\_steps)!")

            print(f"Saving final model (step {step})...")

            torch.save({

                "step": step + 1,

                "model\_state": model.state\_dict(),

                "optimizer\_state": optimizer.state\_dict(),

            }, state\_fpath)

    except KeyboardInterrupt:

        if main\_process:

            print("\nTraining interrupted by user.")

            print(f"Saving the model (step {step}) before exit...")

            torch.save({

                "step": step + 1,

                "model\_state": model.state\_dict(),

                "optimizer\_state": optimizer.state\_dict(),

            }, state\_fpath)

            print("✅ Training stopped and model saved!")

    if main\_process:

        print("✅ Training completed!")

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Trains the speaker encoder.")

    parser.add\_argument("--data\_root", type=str, required=True, help="Path to preprocessed data")

    parser.add\_argument("--run\_id", type=str, default="default\_run", help="Run identifier")

    parser.add\_argument("--models\_dir", type=Path, default=Path("encoder/saved\_models"), help="Directory for saved models")

    parser.add\_argument("--umap\_every", type=int, default=10, help="Steps between UMAP projections")

    parser.add\_argument("--save\_every", type=int, default=100, help="Steps between model saves")

    parser.add\_argument("--backup\_every", type=int, default=500, help="Steps between backups")

    parser.add\_argument("--vis\_every", type=int, default=10, help="Steps between visualization updates")

    parser.add\_argument("--force\_restart", action="store\_true", help="Force training from scratch")

    parser.add\_argument("--visdom\_server", type=str, default="http://localhost", help="Visdom server address")

    parser.add\_argument("--no\_visdom", action="store\_true", help="Disable Visdom visualizations", default=True)

    parser.add\_argument("--max\_steps", type=int, default=3000, help="Maximum training steps")

    parser.add\_argument("--pretrained\_model", type=str, default=None, help="Path to pretrained model for fine-tuning")

    args = parser.parse\_args()

    train(

        run\_id=args.run\_id,

        clean\_data\_root=Path(args.data\_root),

        models\_dir=args.models\_dir,

        umap\_every=args.umap\_every,

        save\_every=args.save\_every,

        backup\_every=args.backup\_every,

        vis\_every=args.vis\_every,

        force\_restart=args.force\_restart,

        visdom\_server=args.visdom\_server,

        no\_visdom=args.no\_visdom,

        max\_steps=args.max\_steps,

        pretrained\_model=args.pretrained\_model

    )

**Synthesizer:  
  
Hparams.py**

import ast

import pprint

class HParams(object):

    def \_\_init\_\_(self, \*\*kwargs): self.\_\_dict\_\_.update(kwargs)

    def \_\_setitem\_\_(self, key, value): setattr(self, key, value)

    def \_\_getitem\_\_(self, key): return getattr(self, key)

    def \_\_repr\_\_(self): return pprint.pformat(self.\_\_dict\_\_)

    def parse(self, string):

        # Overrides hparams from a comma-separated string of name=value pairs

        if len(string) > 0:

            overrides = [s.split("=") for s in string.split(",")]

            keys, values = zip(\*overrides)

            keys = list(map(str.strip, keys))

            values = list(map(str.strip, values))

            for k in keys:

                self.\_\_dict\_\_[k] = ast.literal\_eval(values[keys.index(k)])

        return self

hparams = HParams(

        ### Signal Processing (used in both synthesizer and vocoder)

        sample\_rate = 16000,

        n\_fft = 800,

        num\_mels = 80,

        hop\_size = 200,                             # Tacotron uses 12.5 ms frame shift (set to sample\_rate \* 0.0125)

        win\_size = 800,                             # Tacotron uses 50 ms frame length (set to sample\_rate \* 0.050)

        fmin = 55,

        min\_level\_db = -100,

        ref\_level\_db = 20,

        max\_abs\_value = 4.,                         # Gradient explodes if too big, premature convergence if too small.

        preemphasis = 0.97,                         # Filter coefficient to use if preemphasize is True

        preemphasize = True,

        ### Tacotron Text-to-Speech (TTS)

        tts\_embed\_dims = 512,                       # Embedding dimension for the graphemes/phoneme inputs

        tts\_encoder\_dims = 256,

        tts\_decoder\_dims = 128,

        tts\_postnet\_dims = 512,

        tts\_encoder\_K = 5,

        tts\_lstm\_dims = 1024,

        tts\_postnet\_K = 5,

        tts\_num\_highways = 4,

        tts\_dropout = 0.5,

        tts\_cleaner\_names = ["english\_cleaners"],

        tts\_stop\_threshold = -3.4,                  # Value below which audio generation ends.

                                                    # For example, for a range of [-4, 4], this

                                                    # will terminate the sequence at the first

                                                    # frame that has all values < -3.4

        ### Tacotron Training

        tts\_schedule = [(2,  1e-3,  20\_000,  12),   # Progressive training schedule

                        (2,  5e-4,  40\_000,  12),   # (r, lr, step, batch\_size)

                        (2,  2e-4,  80\_000,  12),   #

                        (2,  1e-4, 160\_000,  12),   # r = reduction factor (# of mel frames

                        (2,  3e-5, 320\_000,  12),   #     synthesized for each decoder iteration)

                        (2,  1e-5, 640\_000,  12)],  # lr = learning rate

        tts\_clip\_grad\_norm = 1.0,                   # clips the gradient norm to prevent explosion - set to None if not needed

        tts\_eval\_interval = 500,                    # Number of steps between model evaluation (sample generation)

                                                    # Set to -1 to generate after completing epoch, or 0 to disable

        tts\_eval\_num\_samples = 1,                   # Makes this number of samples

        ### Data Preprocessing

        max\_mel\_frames = 900,

        rescale = True,

        rescaling\_max = 0.9,

        synthesis\_batch\_size = 16,                  # For vocoder preprocessing and inference.

        ### Mel Visualization and Griffin-Lim

        signal\_normalization = True,

        power = 1.5,

        griffin\_lim\_iters = 60,

        ### Audio processing options

        fmax = 7600,                                # Should not exceed (sample\_rate // 2)

        allow\_clipping\_in\_normalization = True,     # Used when signal\_normalization = True

        clip\_mels\_length = True,                    # If true, discards samples exceeding max\_mel\_frames

        use\_lws = False,                            # "Fast spectrogram phase recovery using local weighted sums"

        symmetric\_mels = True,                      # Sets mel range to [-max\_abs\_value, max\_abs\_value] if True,

                                                    #               and [0, max\_abs\_value] if False

        trim\_silence = True,                        # Use with sample\_rate of 16000 for best results

        ### SV2TTS

        speaker\_embedding\_size = 256,               # Dimension for the speaker embedding

        silence\_min\_duration\_split = 0.4,           # Duration in seconds of a silence for an utterance to be split

        utterance\_min\_duration = 1.6,               # Duration in seconds below which utterances are discarded

        )

def hparams\_debug\_string():

    return str(hparams)

**Audio.py**

import librosa

import librosa.filters

import numpy as np

from scipy import signal

from scipy.io import wavfile

import soundfile as sf

def load\_wav(path, sr):

    return librosa.core.load(path, sr=sr)[0]

def save\_wav(wav, path, sr):

    wav \*= 32767 / max(0.01, np.max(np.abs(wav)))

    #proposed by @dsmiller

    wavfile.write(path, sr, wav.astype(np.int16))

def save\_wavenet\_wav(wav, path, sr):

    sf.write(path, wav.astype(np.float32), sr)

def preemphasis(wav, k, preemphasize=True):

    if preemphasize:

        return signal.lfilter([1, -k], [1], wav)

    return wav

def inv\_preemphasis(wav, k, inv\_preemphasize=True):

    if inv\_preemphasize:

        return signal.lfilter([1], [1, -k], wav)

    return wav

#From https://github.com/r9y9/wavenet\_vocoder/blob/master/audio.py

def start\_and\_end\_indices(quantized, silence\_threshold=2):

    for start in range(quantized.size):

        if abs(quantized[start] - 127) > silence\_threshold:

            break

    for end in range(quantized.size - 1, 1, -1):

        if abs(quantized[end] - 127) > silence\_threshold:

            break

    assert abs(quantized[start] - 127) > silence\_threshold

    assert abs(quantized[end] - 127) > silence\_threshold

    return start, end

def get\_hop\_size(hparams):

    hop\_size = hparams.hop\_size

    if hop\_size is None:

        assert hparams.frame\_shift\_ms is not None

        hop\_size = int(hparams.frame\_shift\_ms / 1000 \* hparams.sample\_rate)

    return hop\_size

def linearspectrogram(wav, hparams):

    D = \_stft(preemphasis(wav, hparams.preemphasis, hparams.preemphasize), hparams)

    S = \_amp\_to\_db(np.abs(D), hparams) - hparams.ref\_level\_db

    if hparams.signal\_normalization:

        return \_normalize(S, hparams)

    return S

def melspectrogram(wav, hparams):

    D = \_stft(preemphasis(wav, hparams.preemphasis, hparams.preemphasize), hparams)

    S = \_amp\_to\_db(\_linear\_to\_mel(np.abs(D), hparams), hparams) - hparams.ref\_level\_db

    if hparams.signal\_normalization:

        return \_normalize(S, hparams)

    return S

def inv\_linear\_spectrogram(linear\_spectrogram, hparams):

    """Converts linear spectrogram to waveform using librosa"""

    if hparams.signal\_normalization:

        D = \_denormalize(linear\_spectrogram, hparams)

    else:

        D = linear\_spectrogram

    S = \_db\_to\_amp(D + hparams.ref\_level\_db) #Convert back to linear

    if hparams.use\_lws:

        processor = \_lws\_processor(hparams)

        D = processor.run\_lws(S.astype(np.float64).T \*\* hparams.power)

        y = processor.istft(D).astype(np.float32)

        return inv\_preemphasis(y, hparams.preemphasis, hparams.preemphasize)

    else:

        return inv\_preemphasis(\_griffin\_lim(S \*\* hparams.power, hparams), hparams.preemphasis, hparams.preemphasize)

def inv\_mel\_spectrogram(mel\_spectrogram, hparams):

    """Converts mel spectrogram to waveform using librosa"""

    if hparams.signal\_normalization:

        D = \_denormalize(mel\_spectrogram, hparams)

    else:

        D = mel\_spectrogram

    S = \_mel\_to\_linear(\_db\_to\_amp(D + hparams.ref\_level\_db), hparams)  # Convert back to linear

    if hparams.use\_lws:

        processor = \_lws\_processor(hparams)

        D = processor.run\_lws(S.astype(np.float64).T \*\* hparams.power)

        y = processor.istft(D).astype(np.float32)

        return inv\_preemphasis(y, hparams.preemphasis, hparams.preemphasize)

    else:

        return inv\_preemphasis(\_griffin\_lim(S \*\* hparams.power, hparams), hparams.preemphasis, hparams.preemphasize)

def \_lws\_processor(hparams):

    import lws

    return lws.lws(hparams.n\_fft, get\_hop\_size(hparams), fftsize=hparams.win\_size, mode="speech")

def \_griffin\_lim(S, hparams):

    """librosa implementation of Griffin-Lim

    Based on https://github.com/librosa/librosa/issues/434

    """

    angles = np.exp(2j \* np.pi \* np.random.rand(\*S.shape))

    S\_complex = np.abs(S).astype(np.complex)

    y = \_istft(S\_complex \* angles, hparams)

    for i in range(hparams.griffin\_lim\_iters):

        angles = np.exp(1j \* np.angle(\_stft(y, hparams)))

        y = \_istft(S\_complex \* angles, hparams)

    return y

def \_stft(y, hparams):

    if hparams.use\_lws:

        return \_lws\_processor(hparams).stft(y).T

    else:

        return librosa.stft(y=y, n\_fft=hparams.n\_fft, hop\_length=get\_hop\_size(hparams), win\_length=hparams.win\_size)

def \_istft(y, hparams):

    return librosa.istft(y, hop\_length=get\_hop\_size(hparams), win\_length=hparams.win\_size)

##########################################################

#Those are only correct when using lws!!! (This was messing with Wavenet quality for a long time!)

def num\_frames(length, fsize, fshift):

    """Compute number of time frames of spectrogram

    """

    pad = (fsize - fshift)

    if length % fshift == 0:

        M = (length + pad \* 2 - fsize) // fshift + 1

    else:

        M = (length + pad \* 2 - fsize) // fshift + 2

    return M

def pad\_lr(x, fsize, fshift):

    """Compute left and right padding

    """

    M = num\_frames(len(x), fsize, fshift)

    pad = (fsize - fshift)

    T = len(x) + 2 \* pad

    r = (M - 1) \* fshift + fsize - T

    return pad, pad + r

##########################################################

#Librosa correct padding

def librosa\_pad\_lr(x, fsize, fshift):

    return 0, (x.shape[0] // fshift + 1) \* fshift - x.shape[0]

# Conversions

\_mel\_basis = None

\_inv\_mel\_basis = None

def \_linear\_to\_mel(spectogram, hparams):

    global \_mel\_basis

    if \_mel\_basis is None:

        \_mel\_basis = \_build\_mel\_basis(hparams)

    return np.dot(\_mel\_basis, spectogram)

def \_mel\_to\_linear(mel\_spectrogram, hparams):

    global \_inv\_mel\_basis

    if \_inv\_mel\_basis is None:

        \_inv\_mel\_basis = np.linalg.pinv(\_build\_mel\_basis(hparams))

    return np.maximum(1e-10, np.dot(\_inv\_mel\_basis, mel\_spectrogram))

def \_build\_mel\_basis(hparams):

    assert hparams.fmax <= hparams.sample\_rate // 2

    return librosa.filters.mel(hparams.sample\_rate, hparams.n\_fft, n\_mels=hparams.num\_mels,

                               fmin=hparams.fmin, fmax=hparams.fmax)

def \_amp\_to\_db(x, hparams):

    min\_level = np.exp(hparams.min\_level\_db / 20 \* np.log(10))

    return 20 \* np.log10(np.maximum(min\_level, x))

def \_db\_to\_amp(x):

    return np.power(10.0, (x) \* 0.05)

def \_normalize(S, hparams):

    if hparams.allow\_clipping\_in\_normalization:

        if hparams.symmetric\_mels:

            return np.clip((2 \* hparams.max\_abs\_value) \* ((S - hparams.min\_level\_db) / (-hparams.min\_level\_db)) - hparams.max\_abs\_value,

                           -hparams.max\_abs\_value, hparams.max\_abs\_value)

        else:

            return np.clip(hparams.max\_abs\_value \* ((S - hparams.min\_level\_db) / (-hparams.min\_level\_db)), 0, hparams.max\_abs\_value)

    assert S.max() <= 0 and S.min() - hparams.min\_level\_db >= 0

    if hparams.symmetric\_mels:

        return (2 \* hparams.max\_abs\_value) \* ((S - hparams.min\_level\_db) / (-hparams.min\_level\_db)) - hparams.max\_abs\_value

    else:

        return hparams.max\_abs\_value \* ((S - hparams.min\_level\_db) / (-hparams.min\_level\_db))

def \_denormalize(D, hparams):

    if hparams.allow\_clipping\_in\_normalization:

        if hparams.symmetric\_mels:

            return (((np.clip(D, -hparams.max\_abs\_value,

                              hparams.max\_abs\_value) + hparams.max\_abs\_value) \* -hparams.min\_level\_db / (2 \* hparams.max\_abs\_value))

                    + hparams.min\_level\_db)

        else:

            return ((np.clip(D, 0, hparams.max\_abs\_value) \* -hparams.min\_level\_db / hparams.max\_abs\_value) + hparams.min\_level\_db)

    if hparams.symmetric\_mels:

        return (((D + hparams.max\_abs\_value) \* -hparams.min\_level\_db / (2 \* hparams.max\_abs\_value)) + hparams.min\_level\_db)

    else:

        return ((D \* -hparams.min\_level\_db / hparams.max\_abs\_value) + hparams.min\_level\_db)

**Preprocess.py**

import sys

import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from multiprocessing.pool import Pool

from synthesizer import audio

from functools import partial

from itertools import chain

from encoder import inference as encoder

from pathlib import Path

from utils import logmmse

from tqdm import tqdm

import numpy as np

import librosa

def preprocess\_dataset(datasets\_root: Path, out\_dir: Path, n\_processes: int, skip\_existing: bool, hparams,

                       no\_alignments: bool, datasets\_name: str, subfolders: str):

    # Gather the input directories

    dataset\_root = datasets\_root.joinpath(datasets\_name)

    input\_dirs = [dataset\_root.joinpath(subfolder.strip()) for subfolder in subfolders.split(",")]

    print("\n    ".join(map(str, ["Using data from:"] + input\_dirs)))

    assert all(input\_dir.exists() for input\_dir in input\_dirs)

    # Create the output directories for each output file type

    out\_dir.joinpath("mels").mkdir(exist\_ok=True)

    out\_dir.joinpath("audio").mkdir(exist\_ok=True)

    # Create a metadata file

    metadata\_fpath = out\_dir.joinpath("train.txt")

    metadata\_file = metadata\_fpath.open("a" if skip\_existing else "w", encoding="utf-8")

    # Preprocess the dataset

    speaker\_dirs = list(chain.from\_iterable(input\_dir.glob("\*") for input\_dir in input\_dirs))

    func = partial(preprocess\_speaker, out\_dir=out\_dir, skip\_existing=skip\_existing,

                   hparams=hparams, no\_alignments=no\_alignments)

    job = Pool(n\_processes).imap(func, speaker\_dirs)

    for speaker\_metadata in tqdm(job, datasets\_name, len(speaker\_dirs), unit="speakers"):

        for metadatum in speaker\_metadata:

            metadata\_file.write("|".join(str(x) for x in metadatum) + "\n")

    metadata\_file.close()

    # Verify the contents of the metadata file

    with metadata\_fpath.open("r", encoding="utf-8") as metadata\_file:

        metadata = [line.split("|") for line in metadata\_file]

    mel\_frames = sum([int(m[4]) for m in metadata])

    timesteps = sum([int(m[3]) for m in metadata])

    sample\_rate = hparams.sample\_rate

    hours = (timesteps / sample\_rate) / 3600

    print("The dataset consists of %d utterances, %d mel frames, %d audio timesteps (%.2f hours)." %

          (len(metadata), mel\_frames, timesteps, hours))

    print("Max input length (text chars): %d" % max(len(m[5]) for m in metadata))

    print("Max mel frames length: %d" % max(int(m[4]) for m in metadata))

    print("Max audio timesteps length: %d" % max(int(m[3]) for m in metadata))

def preprocess\_speaker(speaker\_dir, out\_dir: Path, skip\_existing: bool, hparams, no\_alignments: bool):

    metadata = []

    for book\_dir in speaker\_dir.glob("\*"):

        if no\_alignments:

            # Gather the utterance audios and texts

            # LibriTTS uses .wav but we will include extensions for compatibility with other datasets

            extensions = ["\*.wav", "\*.flac", "\*.mp3"]

            for extension in extensions:

                wav\_fpaths = book\_dir.glob(extension)

                for wav\_fpath in wav\_fpaths:

                    # Load the audio waveform

                    wav, \_ = librosa.load(str(wav\_fpath), hparams.sample\_rate)

                    if hparams.rescale:

                        wav = wav / np.abs(wav).max() \* hparams.rescaling\_max

                    # Get the corresponding text

                    # Check for .txt (for compatibility with other datasets)

                    text\_fpath = wav\_fpath.with\_suffix(".txt")

                    if not text\_fpath.exists():

                        # Check for .normalized.txt (LibriTTS)

                        text\_fpath = wav\_fpath.with\_suffix(".normalized.txt")

                        assert text\_fpath.exists()

                    with text\_fpath.open("r") as text\_file:

                        text = "".join([line for line in text\_file])

                        text = text.replace("\"", "")

                        text = text.strip()

                    # Process the utterance

                    metadata.append(process\_utterance(wav, text, out\_dir, str(wav\_fpath.with\_suffix("").name),

                                                      skip\_existing, hparams))

        else:

            # Process alignment file (LibriSpeech support)

            # Gather the utterance audios and texts

            try:

                alignments\_fpath = next(book\_dir.glob("\*.alignment.txt"))

                with alignments\_fpath.open("r") as alignments\_file:

                    alignments = [line.rstrip().split(" ") for line in alignments\_file]

            except StopIteration:

                # A few alignment files will be missing

                continue

            # Iterate over each entry in the alignments file

            for wav\_fname, words, end\_times in alignments:

                wav\_fpath = book\_dir.joinpath(wav\_fname + ".flac")

                assert wav\_fpath.exists()

                words = words.replace("\"", "").split(",")

                end\_times = list(map(float, end\_times.replace("\"", "").split(",")))

                # Process each sub-utterance

                wavs, texts = split\_on\_silences(wav\_fpath, words, end\_times, hparams)

                for i, (wav, text) in enumerate(zip(wavs, texts)):

                    sub\_basename = "%s\_%02d" % (wav\_fname, i)

                    metadata.append(process\_utterance(wav, text, out\_dir, sub\_basename,

                                                      skip\_existing, hparams))

    return [m for m in metadata if m is not None]

def split\_on\_silences(wav\_fpath, words, end\_times, hparams):

    # Load the audio waveform

    wav, \_ = librosa.load(str(wav\_fpath), hparams.sample\_rate)

    if hparams.rescale:

        wav = wav / np.abs(wav).max() \* hparams.rescaling\_max

    words = np.array(words)

    start\_times = np.array([0.0] + end\_times[:-1])

    end\_times = np.array(end\_times)

    assert len(words) == len(end\_times) == len(start\_times)

    assert words[0] == "" and words[-1] == ""

    # Find pauses that are too long

    mask = (words == "") & (end\_times - start\_times >= hparams.silence\_min\_duration\_split)

    mask[0] = mask[-1] = True

    breaks = np.where(mask)[0]

    # Profile the noise from the silences and perform noise reduction on the waveform

    silence\_times = [[start\_times[i], end\_times[i]] for i in breaks]

    silence\_times = (np.array(silence\_times) \* hparams.sample\_rate).astype(np.int)

    noisy\_wav = np.concatenate([wav[stime[0]:stime[1]] for stime in silence\_times])

    if len(noisy\_wav) > hparams.sample\_rate \* 0.02:

        profile = logmmse.profile\_noise(noisy\_wav, hparams.sample\_rate)

        wav = logmmse.denoise(wav, profile, eta=0)

    # Re-attach segments that are too short

    segments = list(zip(breaks[:-1], breaks[1:]))

    segment\_durations = [start\_times[end] - end\_times[start] for start, end in segments]

    i = 0

    while i < len(segments) and len(segments) > 1:

        if segment\_durations[i] < hparams.utterance\_min\_duration:

            # See if the segment can be re-attached with the right or the left segment

            left\_duration = float("inf") if i == 0 else segment\_durations[i - 1]

            right\_duration = float("inf") if i == len(segments) - 1 else segment\_durations[i + 1]

            joined\_duration = segment\_durations[i] + min(left\_duration, right\_duration)

            # Do not re-attach if it causes the joined utterance to be too long

            if joined\_duration > hparams.hop\_size \* hparams.max\_mel\_frames / hparams.sample\_rate:

                i += 1

                continue

            # Re-attach the segment with the neighbour of shortest duration

            j = i - 1 if left\_duration <= right\_duration else i

            segments[j] = (segments[j][0], segments[j + 1][1])

            segment\_durations[j] = joined\_duration

            del segments[j + 1], segment\_durations[j + 1]

        else:

            i += 1

    # Split the utterance

    segment\_times = [[end\_times[start], start\_times[end]] for start, end in segments]

    segment\_times = (np.array(segment\_times) \* hparams.sample\_rate).astype(np.int)

    wavs = [wav[segment\_time[0]:segment\_time[1]] for segment\_time in segment\_times]

    texts = [" ".join(words[start + 1:end]).replace("  ", " ") for start, end in segments]

    # # DEBUG: play the audio segments (run with -n=1)

    # import sounddevice as sd

    # if len(wavs) > 1:

    #     print("This sentence was split in %d segments:" % len(wavs))

    # else:

    #     print("There are no silences long enough for this sentence to be split:")

    # for wav, text in zip(wavs, texts):

    #     # Pad the waveform with 1 second of silence because sounddevice tends to cut them early

    #     # when playing them. You shouldn't need to do that in your parsers.

    #     wav = np.concatenate((wav, [0] \* 16000))

    #     print("\t%s" % text)

    #     sd.play(wav, 16000, blocking=True)

    # print("")

    return wavs, texts

def process\_utterance(wav: np.ndarray, text: str, out\_dir: Path, basename: str,

                      skip\_existing: bool, hparams):

    ## FOR REFERENCE:

    # For you not to lose your head if you ever wish to change things here or implement your own

    # synthesizer.

    # - Both the audios and the mel spectrograms are saved as numpy arrays

    # - There is no processing done to the audios that will be saved to disk beyond volume

    #   normalization (in split\_on\_silences)

    # - However, pre-emphasis is applied to the audios before computing the mel spectrogram. This

    #   is why we re-apply it on the audio on the side of the vocoder.

    # - Librosa pads the waveform before computing the mel spectrogram. Here, the waveform is saved

    #   without extra padding. This means that you won't have an exact relation between the length

    #   of the wav and of the mel spectrogram. See the vocoder data loader.

    # Skip existing utterances if needed

    mel\_fpath = out\_dir.joinpath("mels", "mel-%s.npy" % basename)

    wav\_fpath = out\_dir.joinpath("audio", "audio-%s.npy" % basename)

    if skip\_existing and mel\_fpath.exists() and wav\_fpath.exists():

        return None

    # Trim silence

    if hparams.trim\_silence:

        wav = encoder.preprocess\_wav(wav, normalize=False, trim\_silence=True)

    # Skip utterances that are too short

    if len(wav) < hparams.utterance\_min\_duration \* hparams.sample\_rate:

        return None

    # Compute the mel spectrogram

    mel\_spectrogram = audio.melspectrogram(wav, hparams).astype(np.float32)

    mel\_frames = mel\_spectrogram.shape[1]

    # Skip utterances that are too long

    if mel\_frames > hparams.max\_mel\_frames and hparams.clip\_mels\_length:

        return None

    # Write the spectrogram, embed and audio to disk

    np.save(mel\_fpath, mel\_spectrogram.T, allow\_pickle=False)

    np.save(wav\_fpath, wav, allow\_pickle=False)

    # Return a tuple describing this training example

    return wav\_fpath.name, mel\_fpath.name, "embed-%s.npy" % basename, len(wav), mel\_frames, text

def embed\_utterance(fpaths, encoder\_model\_fpath):

    if not encoder.is\_loaded():

        encoder.load\_model(encoder\_model\_fpath)

    # Compute the speaker embedding of the utterance

    wav\_fpath, embed\_fpath = fpaths

    wav = np.load(wav\_fpath)

    wav = encoder.preprocess\_wav(wav)

    embed = encoder.embed\_utterance(wav)

    np.save(embed\_fpath, embed, allow\_pickle=False)

def create\_embeddings(synthesizer\_root: Path, encoder\_model\_fpath: Path, n\_processes: int):

    wav\_dir = synthesizer\_root.joinpath("audio")

    metadata\_fpath = synthesizer\_root.joinpath("train.txt")

    assert wav\_dir.exists() and metadata\_fpath.exists()

    embed\_dir = synthesizer\_root.joinpath("embeds")

    embed\_dir.mkdir(exist\_ok=True)

    # Gather the input wave filepath and the target output embed filepath

    with metadata\_fpath.open("r") as metadata\_file:

        metadata = [line.split("|") for line in metadata\_file]

        fpaths = [(wav\_dir.joinpath(m[0]), embed\_dir.joinpath(m[2])) for m in metadata]

    # TODO: improve on the multiprocessing, it's terrible. Disk I/O is the bottleneck here.

    # Embed the utterances in separate threads

    func = partial(embed\_utterance, encoder\_model\_fpath=encoder\_model\_fpath)

    job = Pool(n\_processes).imap(func, fpaths)

    list(tqdm(job, "Embedding", len(fpaths), unit="utterances")) **Train.py**

from datetime import datetime

from functools import partial

from pathlib import Path

import torch

import torch.nn.functional as F

from torch import optim

from torch.utils.data import DataLoader

from synthesizer import audio

from synthesizer.models.tacotron import Tacotron

from synthesizer.synthesizer\_dataset import SynthesizerDataset, collate\_synthesizer

from synthesizer.utils import ValueWindow, data\_parallel\_workaround

from synthesizer.utils.plot import plot\_spectrogram

from synthesizer.utils.symbols import symbols

from synthesizer.utils.text import sequence\_to\_text

from vocoder.display import \*

def np\_now(x: torch.Tensor): return x.detach().cpu().numpy()

def time\_string():

    return datetime.now().strftime("%Y-%m-%d %H:%M")

def train(run\_id: str, syn\_dir: Path, models\_dir: Path, save\_every: int,  backup\_every: int, force\_restart: bool,

          hparams):

    models\_dir.mkdir(exist\_ok=True)

    model\_dir = models\_dir.joinpath(run\_id)

    plot\_dir = model\_dir.joinpath("plots")

    wav\_dir = model\_dir.joinpath("wavs")

    mel\_output\_dir = model\_dir.joinpath("mel-spectrograms")

    meta\_folder = model\_dir.joinpath("metas")

    model\_dir.mkdir(exist\_ok=True)

    plot\_dir.mkdir(exist\_ok=True)

    wav\_dir.mkdir(exist\_ok=True)

    mel\_output\_dir.mkdir(exist\_ok=True)

    meta\_folder.mkdir(exist\_ok=True)

    weights\_fpath = model\_dir / f"synthesizer.pt"

    metadata\_fpath = syn\_dir.joinpath("train.txt")

    print("Checkpoint path: {}".format(weights\_fpath))

    print("Loading training data from: {}".format(metadata\_fpath))

    print("Using model: Tacotron")

    # Bookkeeping

    time\_window = ValueWindow(100)

    loss\_window = ValueWindow(100)

    # From WaveRNN/train\_tacotron.py

    if torch.cuda.is\_available():

        device = torch.device("cuda")

        for session in hparams.tts\_schedule:

            \_, \_, \_, batch\_size = session

            if batch\_size % torch.cuda.device\_count() != 0:

                raise ValueError("`batch\_size` must be evenly divisible by n\_gpus!")

    else:

        device = torch.device("cpu")

    print("Using device:", device)

    # Instantiate Tacotron Model

    print("\nInitialising Tacotron Model...\n")

    model = Tacotron(embed\_dims=hparams.tts\_embed\_dims,

                     num\_chars=len(symbols),

                     encoder\_dims=hparams.tts\_encoder\_dims,

                     decoder\_dims=hparams.tts\_decoder\_dims,

                     n\_mels=hparams.num\_mels,

                     fft\_bins=hparams.num\_mels,

                     postnet\_dims=hparams.tts\_postnet\_dims,

                     encoder\_K=hparams.tts\_encoder\_K,

                     lstm\_dims=hparams.tts\_lstm\_dims,

                     postnet\_K=hparams.tts\_postnet\_K,

                     num\_highways=hparams.tts\_num\_highways,

                     dropout=hparams.tts\_dropout,

                     stop\_threshold=hparams.tts\_stop\_threshold,

                     speaker\_embedding\_size=hparams.speaker\_embedding\_size).to(device)

    # Initialize the optimizer

    optimizer = optim.Adam(model.parameters())

    # Load the weights

    if force\_restart or not weights\_fpath.exists():

        print("\nStarting the training of Tacotron from scratch\n")

        model.save(weights\_fpath)

        # Embeddings metadata

        char\_embedding\_fpath = meta\_folder.joinpath("CharacterEmbeddings.tsv")

        with open(char\_embedding\_fpath, "w", encoding="utf-8") as f:

            for symbol in symbols:

                if symbol == " ":

                    symbol = "\\s"  # For visual purposes, swap space with \s

                f.write("{}\n".format(symbol))

    else:

        print("\nLoading weights at %s" % weights\_fpath)

        model.load(weights\_fpath, optimizer)

        print("Tacotron weights loaded from step %d" % model.step)

    # Initialize the dataset

    metadata\_fpath = syn\_dir.joinpath("train.txt")

    mel\_dir = syn\_dir.joinpath("mels")

    embed\_dir = syn\_dir.joinpath("embeds")

    dataset = SynthesizerDataset(metadata\_fpath, mel\_dir, embed\_dir, hparams)

    for i, session in enumerate(hparams.tts\_schedule):

        current\_step = model.get\_step()

        r, lr, max\_step, batch\_size = session

        training\_steps = max\_step - current\_step

        # Do we need to change to the next session?

        if current\_step >= max\_step:

            # Are there no further sessions than the current one?

            if i == len(hparams.tts\_schedule) - 1:

                # We have completed training. Save the model and exit

                model.save(weights\_fpath, optimizer)

                break

            else:

                # There is a following session, go to it

                continue

        model.r = r

        # Begin the training

        simple\_table([(f"Steps with r={r}", str(training\_steps // 1000) + "k Steps"),

                      ("Batch Size", batch\_size),

                      ("Learning Rate", lr),

                      ("Outputs/Step (r)", model.r)])

        for p in optimizer.param\_groups:

            p["lr"] = lr

        collate\_fn = partial(collate\_synthesizer, r=r, hparams=hparams)

        data\_loader = DataLoader(dataset, batch\_size, shuffle=True, num\_workers=2, collate\_fn=collate\_fn)

        total\_iters = len(dataset)

        steps\_per\_epoch = np.ceil(total\_iters / batch\_size).astype(np.int32)

        epochs = np.ceil(training\_steps / steps\_per\_epoch).astype(np.int32)

        for epoch in range(1, epochs+1):

            for i, (texts, mels, embeds, idx) in enumerate(data\_loader, 1):

                start\_time = time.time()

                # Generate stop tokens for training

                stop = torch.ones(mels.shape[0], mels.shape[2])

                for j, k in enumerate(idx):

                    stop[j, :int(dataset.metadata[k][4])-1] = 0

                texts = texts.to(device)

                mels = mels.to(device)

                embeds = embeds.to(device)

                stop = stop.to(device)

                # Forward pass

                # Parallelize model onto GPUS using workaround due to python bug

                if device.type == "cuda" and torch.cuda.device\_count() > 1:

                    m1\_hat, m2\_hat, attention, stop\_pred = data\_parallel\_workaround(model, texts, mels, embeds)

                else:

                    m1\_hat, m2\_hat, attention, stop\_pred = model(texts, mels, embeds)

                # Backward pass

                m1\_loss = F.mse\_loss(m1\_hat, mels) + F.l1\_loss(m1\_hat, mels)

                m2\_loss = F.mse\_loss(m2\_hat, mels)

                stop\_loss = F.binary\_cross\_entropy(stop\_pred, stop)

                loss = m1\_loss + m2\_loss + stop\_loss

                optimizer.zero\_grad()

                loss.backward()

                if hparams.tts\_clip\_grad\_norm is not None:

                    grad\_norm = torch.nn.utils.clip\_grad\_norm\_(model.parameters(), hparams.tts\_clip\_grad\_norm)

                    if np.isnan(grad\_norm.cpu()):

                        print("grad\_norm was NaN!")

                optimizer.step()

                time\_window.append(time.time() - start\_time)

                loss\_window.append(loss.item())

                step = model.get\_step()

                k = step // 1000

                msg = f"| Epoch: {epoch}/{epochs} ({i}/{steps\_per\_epoch}) | Loss: {loss\_window.average:#.4} | " \

                      f"{1./time\_window.average:#.2} steps/s | Step: {k}k | "

                stream(msg)

                # Backup or save model as appropriate

                if backup\_every != 0 and step % backup\_every == 0 :

                    backup\_fpath = weights\_fpath.parent / f"synthesizer\_{k:06d}.pt"

                    model.save(backup\_fpath, optimizer)

                if save\_every != 0 and step % save\_every == 0 :

                    # Must save latest optimizer state to ensure that resuming training

                    # doesn't produce artifacts

                    model.save(weights\_fpath, optimizer)

                # Evaluate model to generate samples

                epoch\_eval = hparams.tts\_eval\_interval == -1 and i == steps\_per\_epoch  # If epoch is done

                step\_eval = hparams.tts\_eval\_interval > 0 and step % hparams.tts\_eval\_interval == 0  # Every N steps

                if epoch\_eval or step\_eval:

                    for sample\_idx in range(hparams.tts\_eval\_num\_samples):

                        # At most, generate samples equal to number in the batch

                        if sample\_idx + 1 <= len(texts):

                            # Remove padding from mels using frame length in metadata

                            mel\_length = int(dataset.metadata[idx[sample\_idx]][4])

                            mel\_prediction = np\_now(m2\_hat[sample\_idx]).T[:mel\_length]

                            target\_spectrogram = np\_now(mels[sample\_idx]).T[:mel\_length]

                            attention\_len = mel\_length // model.r

                            eval\_model(attention=np\_now(attention[sample\_idx][:, :attention\_len]),

                                       mel\_prediction=mel\_prediction,

                                       target\_spectrogram=target\_spectrogram,

                                       input\_seq=np\_now(texts[sample\_idx]),

                                       step=step,

                                       plot\_dir=plot\_dir,

                                       mel\_output\_dir=mel\_output\_dir,

                                       wav\_dir=wav\_dir,

                                       sample\_num=sample\_idx + 1,

                                       loss=loss,

                                       hparams=hparams)

                # Break out of loop to update training schedule

                if step >= max\_step:

                    break

            # Add line break after every epoch

            print("")

def eval\_model(attention, mel\_prediction, target\_spectrogram, input\_seq, step,

               plot\_dir, mel\_output\_dir, wav\_dir, sample\_num, loss, hparams):

    # Save some results for evaluation

    attention\_path = str(plot\_dir.joinpath("attention\_step\_{}\_sample\_{}".format(step, sample\_num)))

    save\_attention(attention, attention\_path)

    # save predicted mel spectrogram to disk (debug)

    mel\_output\_fpath = mel\_output\_dir.joinpath("mel-prediction-step-{}\_sample\_{}.npy".format(step, sample\_num))

    np.save(str(mel\_output\_fpath), mel\_prediction, allow\_pickle=False)

    # save griffin lim inverted wav for debug (mel -> wav)

    wav = audio.inv\_mel\_spectrogram(mel\_prediction.T, hparams)

    wav\_fpath = wav\_dir.joinpath("step-{}-wave-from-mel\_sample\_{}.wav".format(step, sample\_num))

    audio.save\_wav(wav, str(wav\_fpath), sr=hparams.sample\_rate)

    # save real and predicted mel-spectrogram plot to disk (control purposes)

    spec\_fpath = plot\_dir.joinpath("step-{}-mel-spectrogram\_sample\_{}.png".format(step, sample\_num))

    title\_str = "{}, {}, step={}, loss={:.5f}".format("Tacotron", time\_string(), step, loss)

    plot\_spectrogram(mel\_prediction, str(spec\_fpath), title=title\_str,

                     target\_spectrogram=target\_spectrogram,

                     max\_len=target\_spectrogram.size // hparams.num\_mels)

    print("Input at step {}: {}".format(step, sequence\_to\_text(input\_seq)))

**Inference.py**

import torch

from synthesizer import audio

from synthesizer.hparams import hparams

from synthesizer.models.tacotron import Tacotron

from synthesizer.utils.symbols import symbols

from synthesizer.utils.text import text\_to\_sequence

from vocoder.display import simple\_table

from pathlib import Path

from typing import Union, List

import numpy as np

import librosa

class Synthesizer:

    sample\_rate = hparams.sample\_rate

    hparams = hparams

    def \_\_init\_\_(self, model\_fpath: Path, verbose=True):

        """

        The model isn't instantiated and loaded in memory until needed or until load() is called.

        :param model\_fpath: path to the trained model file

        :param verbose: if False, prints less information when using the model

        """

        self.model\_fpath = model\_fpath

        self.verbose = verbose

        # Check for GPU

        if torch.cuda.is\_available():

            self.device = torch.device("cuda")

        else:

            self.device = torch.device("cpu")

        if self.verbose:

            print("Synthesizer using device:", self.device)

        # Tacotron model will be instantiated later on first use.

        self.\_model = None

    def is\_loaded(self):

        """

        Whether the model is loaded in memory.

        """

        return self.\_model is not None

    def load(self):

        """

        Instantiates and loads the model given the weights file that was passed in the constructor.

        """

        self.\_model = Tacotron(embed\_dims=hparams.tts\_embed\_dims,

                               num\_chars=len(symbols),

                               encoder\_dims=hparams.tts\_encoder\_dims,

                               decoder\_dims=hparams.tts\_decoder\_dims,

                               n\_mels=hparams.num\_mels,

                               fft\_bins=hparams.num\_mels,

                               postnet\_dims=hparams.tts\_postnet\_dims,

                               encoder\_K=hparams.tts\_encoder\_K,

                               lstm\_dims=hparams.tts\_lstm\_dims,

                               postnet\_K=hparams.tts\_postnet\_K,

                               num\_highways=hparams.tts\_num\_highways,

                               dropout=hparams.tts\_dropout,

                               stop\_threshold=hparams.tts\_stop\_threshold,

                               speaker\_embedding\_size=hparams.speaker\_embedding\_size).to(self.device)

        self.\_model.load(self.model\_fpath)

        self.\_model.eval()

        if self.verbose:

            print("Loaded synthesizer \"%s\" trained to step %d" % (self.model\_fpath.name, self.\_model.state\_dict()["step"]))

    def synthesize\_spectrograms(self, texts: List[str],

                                embeddings: Union[np.ndarray, List[np.ndarray]],

                                return\_alignments=False):

        """

        Synthesizes mel spectrograms from texts and speaker embeddings.

        :param texts: a list of N text prompts to be synthesized

        :param embeddings: a numpy array or list of speaker embeddings of shape (N, 256)

        :param return\_alignments: if True, a matrix representing the alignments between the

        characters

        and each decoder output step will be returned for each spectrogram

        :return: a list of N melspectrograms as numpy arrays of shape (80, Mi), where Mi is the

        sequence length of spectrogram i, and possibly the alignments.

        """

        # Load the model on the first request.

        if not self.is\_loaded():

            self.load()

        # Preprocess text inputs

        inputs = [text\_to\_sequence(text.strip(), hparams.tts\_cleaner\_names) for text in texts]

        if not isinstance(embeddings, list):

            embeddings = [embeddings]

        # Batch inputs

        batched\_inputs = [inputs[i:i+hparams.synthesis\_batch\_size]

                             for i in range(0, len(inputs), hparams.synthesis\_batch\_size)]

        batched\_embeds = [embeddings[i:i+hparams.synthesis\_batch\_size]

                             for i in range(0, len(embeddings), hparams.synthesis\_batch\_size)]

        specs = []

        for i, batch in enumerate(batched\_inputs, 1):

            if self.verbose:

                print(f"\n| Generating {i}/{len(batched\_inputs)}")

            # Pad texts so they are all the same length

            text\_lens = [len(text) for text in batch]

            max\_text\_len = max(text\_lens)

            chars = [pad1d(text, max\_text\_len) for text in batch]

            chars = np.stack(chars)

            # Stack speaker embeddings into 2D array for batch processing

            speaker\_embeds = np.stack(batched\_embeds[i-1])

            # Convert to tensor

            chars = torch.tensor(chars).long().to(self.device)

            speaker\_embeddings = torch.tensor(speaker\_embeds).float().to(self.device)

            # Inference

            \_, mels, alignments = self.\_model.generate(chars, speaker\_embeddings)

            mels = mels.detach().cpu().numpy()

            for m in mels:

                # Trim silence from end of each spectrogram

                while np.max(m[:, -1]) < hparams.tts\_stop\_threshold:

                    m = m[:, :-1]

                specs.append(m)

        if self.verbose:

            print("\n\nDone.\n")

        return (specs, alignments) if return\_alignments else specs

    @staticmethod

    def load\_preprocess\_wav(fpath):

        """

        Loads and preprocesses an audio file under the same conditions the audio files were used to

        train the synthesizer.

        """

        wav = librosa.load(str(fpath), hparams.sample\_rate)[0]

        if hparams.rescale:

            wav = wav / np.abs(wav).max() \* hparams.rescaling\_max

        return wav

    @staticmethod

    def make\_spectrogram(fpath\_or\_wav: Union[str, Path, np.ndarray]):

        """

        Creates a mel spectrogram from an audio file in the same manner as the mel spectrograms that

        were fed to the synthesizer when training.

        """

        if isinstance(fpath\_or\_wav, str) or isinstance(fpath\_or\_wav, Path):

            wav = Synthesizer.load\_preprocess\_wav(fpath\_or\_wav)

        else:

            wav = fpath\_or\_wav

        mel\_spectrogram = audio.melspectrogram(wav, hparams).astype(np.float32)

        return mel\_spectrogram

    @staticmethod

    def griffin\_lim(mel):

        """

        Inverts a mel spectrogram using Griffin-Lim. The mel spectrogram is expected to have been built

        with the same parameters present in hparams.py.

        """

        return audio.inv\_mel\_spectrogram(mel, hparams)

def pad1d(x, max\_len, pad\_value=0):

    return np.pad(x, (0, max\_len - len(x)), mode="constant", constant\_values=pad\_value)

**Vocoder:  
  
Hparams.py**

from synthesizer.hparams import hp as \_syn\_hp

# Audio settings------------------------------------------------------------------------

# Match the values of the synthesizer

sample\_rate = \_syn\_hp.sample\_rate

n\_fft = \_syn\_hp.n\_fft

num\_mels = \_syn\_hp.num\_mels

hop\_length = \_syn\_hp.hop\_size

win\_length = \_syn\_hp.win\_size

fmin = \_syn\_hp.fmin

min\_level\_db = \_syn\_hp.min\_level\_db

ref\_level\_db = \_syn\_hp.ref\_level\_db

mel\_max\_abs\_value = \_syn\_hp.max\_abs\_value

preemphasis = \_syn\_hp.preemphasis

apply\_preemphasis = \_syn\_hp.preemphasize

bits = 9                            # bit depth of signal

mu\_law = True                       # Recommended to suppress noise if using raw bits in hp.voc\_mode

                                    # below

# WAVERNN / VOCODER --------------------------------------------------------------------------------

voc\_mode = 'RAW'                    # either 'RAW' (softmax on raw bits) or 'MOL' (sample from

# mixture of logistics)

#voc\_upsample\_factors = (5, 5, 8)    # NB - this needs to correctly factorise hop\_length

voc\_upsample\_factors = (4, 4, 4, 4)  # 4 × 4 × 4 × 4 = 256

voc\_rnn\_dims = 512

voc\_fc\_dims = 512

voc\_compute\_dims = 128

voc\_res\_out\_dims = 128

voc\_res\_blocks = 10

# Training

voc\_batch\_size = 100 #Number of audio clips per training batch for WaveRNN.

voc\_lr = 1e-4

voc\_gen\_at\_checkpoint = 5           # number of samples to generate at each checkpoint

voc\_pad = 2                         # this will pad the input so that the resnet can 'see' wider

                                    # than input length

voc\_seq\_len = hop\_length \* 5        # must be a multiple of hop\_length

# Generating / Synthesizing

voc\_gen\_batched = True              # very fast (realtime+) single utterance batched generation

voc\_target = 8000                   # target number of samples to be generated in each batch entry

voc\_overlap = 400                   # number of samples for crossfading between batches

**Audio.py**

import math

import numpy as np

import librosa

import vocoder.hparams as hp

from scipy.signal import lfilter

import soundfile as sf

def label\_2\_float(x, bits) :

    return 2 \* x / (2\*\*bits - 1.) - 1. # Converts integer values (labels) to a float in [-1, 1] range.

def float\_2\_label(x, bits) :

    assert abs(x).max() <= 1.0

    x = (x + 1.) \* (2\*\*bits - 1) / 2

    return x.clip(0, 2\*\*bits - 1) # Converts float in [-1, 1] back to integer labels.

def load\_wav(path) :

    return librosa.load(str(path), sr=hp.sample\_rate)[0]

def save\_wav(x, path) :

    sf.write(path, x.astype(np.float32), hp.sample\_rate)

def split\_signal(x) :

    unsigned = x + 2\*\*15

    coarse = unsigned // 256

    fine = unsigned % 256

    return coarse, fine

def combine\_signal(coarse, fine) :

    return coarse \* 256 + fine - 2\*\*15

def encode\_16bits(x) : #Converts float waveform in [-1, 1] to 16-bit signed integers for storage or processing.

    return np.clip(x \* 2\*\*15, -2\*\*15, 2\*\*15 - 1).astype(np.int16)

mel\_basis = None

def linear\_to\_mel(spectrogram):

    global mel\_basis

    if mel\_basis is None:

        mel\_basis = build\_mel\_basis()

    return np.dot(mel\_basis, spectrogram)

def build\_mel\_basis():

    return librosa.filters.mel(hp.sample\_rate, hp.n\_fft, n\_mels=hp.num\_mels, fmin=hp.fmin)

def normalize(S): #Normalizes spectrogram to range [0, 1].

    return np.clip((S - hp.min\_level\_db) / -hp.min\_level\_db, 0, 1)

def denormalize(S): #Reverts normalized spectrogram back to dB scale.

    return (np.clip(S, 0, 1) \* -hp.min\_level\_db) + hp.min\_level\_db

def amp\_to\_db(x):

    return 20 \* np.log10(np.maximum(1e-5, x))

def db\_to\_amp(x):

    return np.power(10.0, x \* 0.05)

def spectrogram(y):

    D = stft(y)

    S = amp\_to\_db(np.abs(D)) - hp.ref\_level\_db

    return normalize(S)

def melspectrogram(y):

    D = stft(y)

    S = amp\_to\_db(linear\_to\_mel(np.abs(D)))

    return normalize(S)

def stft(y):

    return librosa.stft(y=y, n\_fft=hp.n\_fft, hop\_length=hp.hop\_length, win\_length=hp.win\_length)

def pre\_emphasis(x): #Boosts high frequencies to improve STFT quality and reduce low-frequency noise.

    return lfilter([1, -hp.preemphasis], [1], x)

def de\_emphasis(x):

    return lfilter([1], [1, -hp.preemphasis], x)

def encode\_mu\_law(x, mu) :

    mu = mu - 1

    fx = np.sign(x) \* np.log(1 + mu \* np.abs(x)) / np.log(1 + mu)

    return np.floor((fx + 1) / 2 \* mu + 0.5)

def decode\_mu\_law(y, mu, from\_labels=True) :

    if from\_labels:

        y = label\_2\_float(y, math.log2(mu))

    mu = mu - 1

    x = np.sign(y) / mu \* ((1 + mu) \*\* np.abs(y) - 1)

    return x

**Vocoder\_dataset.py**from torch.utils.data import Dataset

from pathlib import Path

from vocoder import audio

import vocoder.hparams as hp

import numpy as np

import torch

class VocoderDataset(Dataset):

    def \_\_init\_\_(self, metadata\_fpath: Path, mel\_dir: Path, wav\_dir: Path):

        print("Using inputs from:\n\t%s\n\t%s\n\t%s" % (metadata\_fpath, mel\_dir, wav\_dir))

        with metadata\_fpath.open("r") as metadata\_file:

            metadata = [line.split("|") for line in metadata\_file]

        gta\_fnames = [x[1] for x in metadata if int(x[4])]

        gta\_fpaths = [mel\_dir.joinpath(fname) for fname in gta\_fnames]

        wav\_fnames = [x[0] for x in metadata if int(x[4])]

        wav\_fpaths = [wav\_dir.joinpath(fname) for fname in wav\_fnames]

        self.samples\_fpaths = list(zip(gta\_fpaths, wav\_fpaths))

        print("Found %d samples" % len(self.samples\_fpaths))

    def \_\_getitem\_\_(self, index):

        mel\_path, wav\_path = self.samples\_fpaths[index]

        # Load the mel spectrogram and adjust its range to [-1, 1]

        mel = np.load(mel\_path).T.astype(np.float32) / hp.mel\_max\_abs\_value

        # Load the wav

        wav = np.load(wav\_path)

        if hp.apply\_preemphasis:

            wav = audio.pre\_emphasis(wav)

        wav = np.clip(wav, -1, 1)

        # Fix for missing padding   # TODO: settle on whether this is any useful

        r\_pad =  (len(wav) // hp.hop\_length + 1) \* hp.hop\_length - len(wav)

        wav = np.pad(wav, (0, r\_pad), mode='constant')

        assert len(wav) >= mel.shape[1] \* hp.hop\_length

        wav = wav[:mel.shape[1] \* hp.hop\_length]

        assert len(wav) % hp.hop\_length == 0

        # Quantize the wav

        if hp.voc\_mode == 'RAW':

            if hp.mu\_law:

                quant = audio.encode\_mu\_law(wav, mu=2 \*\* hp.bits)

            else:

                quant = audio.float\_2\_label(wav, bits=hp.bits)

        elif hp.voc\_mode == 'MOL':

            quant = audio.float\_2\_label(wav, bits=16)

        return mel.astype(np.float32), quant.astype(np.int64)

    def \_\_len\_\_(self):

        return len(self.samples\_fpaths)

def collate\_vocoder(batch):

    mel\_win = hp.voc\_seq\_len // hp.hop\_length + 2 \* hp.voc\_pad

    max\_offsets = [x[0].shape[-1] -2 - (mel\_win + 2 \* hp.voc\_pad) for x in batch]

    mel\_offsets = [np.random.randint(0, offset) for offset in max\_offsets]

    sig\_offsets = [(offset + hp.voc\_pad) \* hp.hop\_length for offset in mel\_offsets]

    mels = [x[0][:, mel\_offsets[i]:mel\_offsets[i] + mel\_win] for i, x in enumerate(batch)]

    labels = [x[1][sig\_offsets[i]:sig\_offsets[i] + hp.voc\_seq\_len + 1] for i, x in enumerate(batch)]

    mels = np.stack(mels).astype(np.float32)

    labels = np.stack(labels).astype(np.int64)

    mels = torch.tensor(mels)

    labels = torch.tensor(labels).long()

    x = labels[:, :hp.voc\_seq\_len]

    y = labels[:, 1:]

    bits = 16 if hp.voc\_mode == 'MOL' else hp.bits

    x = audio.label\_2\_float(x.float(), bits)

    if hp.voc\_mode == 'MOL' :

        y = audio.label\_2\_float(y.float(), bits)

    return x, y, mels

**Train.py**

import sys

import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

import time

from pathlib import Path

import numpy as np

import torch

import torch.nn.functional as F # Contains functions like loss functions.

from torch import optim

from torch.utils.data import DataLoader

import vocoder.hparams as hp

from vocoder.display import stream, simple\_table

from vocoder.distribution import discretized\_mix\_logistic\_loss #A custom loss function used when model mode is "MOL".

from vocoder.gen\_wavernn import gen\_testset

from vocoder.models.fatchord\_version import WaveRNN

from vocoder.vocoder\_dataset import VocoderDataset, collate\_vocoder

def train(run\_id: str, syn\_dir: Path, voc\_dir: Path, models\_dir: Path, ground\_truth: bool, save\_every: int,

          backup\_every: int, force\_restart: bool):

    # Check to make sure the hop length is correctly factorised

    assert np.cumprod(hp.voc\_upsample\_factors)[-1] == hp.hop\_length

    # Instantiate the model

    print("Initializing the model...")

    model = WaveRNN(

        rnn\_dims=hp.voc\_rnn\_dims,

        fc\_dims=hp.voc\_fc\_dims,

        bits=hp.bits,

        pad=hp.voc\_pad,

        upsample\_factors=hp.voc\_upsample\_factors,

        feat\_dims=hp.num\_mels,

        compute\_dims=hp.voc\_compute\_dims,

        res\_out\_dims=hp.voc\_res\_out\_dims,

        res\_blocks=hp.voc\_res\_blocks,

        hop\_length=hp.hop\_length,

        sample\_rate=hp.sample\_rate,

        mode=hp.voc\_mode

    )

    if torch.cuda.is\_available():

        model = model.cuda()

    # Initialize the optimizer

    optimizer = optim.Adam(model.parameters())

    for p in optimizer.param\_groups:

        p["lr"] = hp.voc\_lr

    loss\_func = F.cross\_entropy if model.mode == "RAW" else discretized\_mix\_logistic\_loss

    # Load the weights

    model\_dir = models\_dir / run\_id

    model\_dir.mkdir(exist\_ok=True)

    weights\_fpath = model\_dir / "vocoder.pt"

    if force\_restart or not weights\_fpath.exists():

        print("\nStarting the training of WaveRNN from scratch\n")

        model.save(weights\_fpath, optimizer)

    else:

        print("\nLoading weights at %s" % weights\_fpath)

        model.load(weights\_fpath, optimizer)

        print("WaveRNN weights loaded from step %d" % model.step)

    # Initialize the dataset

    metadata\_fpath = syn\_dir.joinpath("train.txt") if ground\_truth else \

        voc\_dir.joinpath("synthesized.txt")

    mel\_dir = syn\_dir.joinpath("mels") if ground\_truth else voc\_dir.joinpath("mels\_gta")

    wav\_dir = syn\_dir.joinpath("audio")

    dataset = VocoderDataset(metadata\_fpath, mel\_dir, wav\_dir)

    test\_loader = DataLoader(dataset, batch\_size=1, shuffle=True)

    # Begin the training

    simple\_table([('Batch size', hp.voc\_batch\_size),

                  ('LR', hp.voc\_lr),

                  ('Sequence Len', hp.voc\_seq\_len)])

    for epoch in range(1, 350):

        data\_loader = DataLoader(dataset, hp.voc\_batch\_size, shuffle=True, num\_workers=2, collate\_fn=collate\_vocoder)

        start = time.time()

        running\_loss = 0.

        for i, (x, y, m) in enumerate(data\_loader, 1): # Loops over batches of (x = input audio, y = target audio, m = mel spectrogram)

            if torch.cuda.is\_available():

                x, m, y = x.cuda(), m.cuda(), y.cuda()

            # Forward pass

            y\_hat = model(x, m) # Formats y\_hat and y to match the shape required by the loss function.

            if model.mode == 'RAW':

                y\_hat = y\_hat.transpose(1, 2).unsqueeze(-1)

            elif model.mode == 'MOL':

                y = y.float()

            y = y.unsqueeze(-1)

            # Backward pass

            loss = loss\_func(y\_hat, y)

            optimizer.zero\_grad()

            loss.backward()

            optimizer.step()

            running\_loss += loss.item()

            speed = i / (time.time() - start)

            avg\_loss = running\_loss / i

            # Saves model checkpoint regularly and does backups.

            step = model.get\_step()

            k = step // 1000

            if backup\_every != 0 and step % backup\_every == 0 :

                model.checkpoint(model\_dir, optimizer)

            if save\_every != 0 and step % save\_every == 0 :

                model.save(weights\_fpath, optimizer)

            msg = f"| Epoch: {epoch} ({i}/{len(data\_loader)}) | " \

                f"Loss: {avg\_loss:.4f} | {speed:.1f} " \

                f"steps/s | Step: {k}k | "

            stream(msg)

        gen\_testset(model, test\_loader, hp.voc\_gen\_at\_checkpoint, hp.voc\_gen\_batched,

                    hp.voc\_target, hp.voc\_overlap, model\_dir)

        print("")

**Inference.py**from vocoder.models.fatchord\_version import WaveRNN

from vocoder import hparams as hp

import torch

\_model = None   # type: WaveRNN

def load\_model(weights\_fpath, verbose=True):

    global \_model, \_device

    if verbose:

        print("Building Wave-RNN")

    \_model = WaveRNN(

        rnn\_dims=hp.voc\_rnn\_dims,

        fc\_dims=hp.voc\_fc\_dims,

        bits=hp.bits,

        pad=hp.voc\_pad,

        upsample\_factors=hp.voc\_upsample\_factors,

        feat\_dims=hp.num\_mels,

        compute\_dims=hp.voc\_compute\_dims,

        res\_out\_dims=hp.voc\_res\_out\_dims,

        res\_blocks=hp.voc\_res\_blocks,

        hop\_length=hp.hop\_length,

        sample\_rate=hp.sample\_rate,

        mode=hp.voc\_mode

    )

    if torch.cuda.is\_available():

        \_model = \_model.cuda()

        \_device = torch.device('cuda')

    else:

        \_device = torch.device('cpu')

    if verbose:

        print("Loading model weights at %s" % weights\_fpath)

    checkpoint = torch.load(weights\_fpath, \_device)

    \_model.load\_state\_dict(checkpoint['model\_state'])

    \_model.eval()

def is\_loaded():

    return \_model is not None

def infer\_waveform(mel, normalize=True,  batched=True, target=8000, overlap=800,

                   progress\_callback=None):

    """

    Infers the waveform of a mel spectrogram output by the synthesizer (the format must match

    that of the synthesizer!)

    :param normalize:

    :param batched:

    :param target:

    :param overlap:

    :return:

    """

    if \_model is None:

        raise Exception("Please load Wave-RNN in memory before using it")

    if normalize:

        mel = mel / hp.mel\_max\_abs\_value

    mel = torch.from\_numpy(mel[None, ...])

    wav = \_model.generate(mel, batched, target, overlap, hp.mu\_law, progress\_callback)

    return wav