The code snippets represent different parts of a speech enhancement system using a U-Net architecture.

**1. main.py**

* The main.py script acts as the entry point to the program. Depending on the --mode argument (data\_creation, training, or prediction), it calls the appropriate function to:

Create data for training.

Train the U-Net model.

Use the trained model for prediction.

* It defines various arguments related to audio processing, data paths, and model training parameters.

### 2. ****args.py****

* This script handles command-line arguments using the argparse module. It defines all the necessary parameters, including paths to data directories, model configurations, and audio processing parameters.

### 3. ****prepare\_data.py****

* Contains the create\_data() function that prepares the dataset for training. It combines clean speech and noise, creates spectrograms, and saves them for later use. It relies on functions from data\_tools.py to handle audio processing tasks such as converting audio to numpy arrays, creating spectrograms, and blending noise with clean speech.

This function creates the training dataset by blending clean voice and noise audio samples.

It reads audio files from specified directories and converts them to spectrograms.

It randomly blends noise with clean voices at different levels to create noisy training samples.

It saves the spectrograms of noisy voice, clean voice, and noise along with their time-series and audio representations.

### 4. ****data\_tools.py****

* This module contains helper functions for audio processing and spectrogram generation.

A utility script that provides functions to process audio files. These include:

Splitting audio into frames with a specified hop length.

Converting audio files and merging them into a single numpy arrays.

Randomly blending noise with clean voice for training data.

Converting audio to spectrograms and their inverse transformations. (magnitude and phase)

Scaling spectrogram values for model training input/output.

### 5. ****train\_model.py****

This function trains a U-Net model for speech enhancement.

It loads the preprocessed spectrograms (noisy and clean voices) created by prepare\_data.py.

It defines a U-Net model architecture (imported from model\_unet.py).

It preprocesses the data by scaling spectrograms between -1 and 1 for better training.

It splits the data into training and validation sets.

It trains the model with options for starting from scratch or using pre-trained weights.

It implements callbacks to save the best performing model during training.

It plots the training and validation loss curves.

**main.py file:**

from args import parser

import os

from prepare\_data import create\_data

from train\_model import training

from prediction\_denoise import prediction

**Imports**:

* from args import parser: This line imports the parser object from the args.py module. The parser is an ArgumentParser object that defines various command-line arguments for the script.
* import os: This imports the os module, which provides a way of interacting with the operating system, such as handling file paths and directories.
* from prepare\_data import create\_data: This imports the create\_data function from the prepare\_data.py module. This function handles the creation of data for training by blending noise with clean voice samples and generating the corresponding spectrograms.
* from train\_model import training: This imports the training function from the train\_model.py module. This function handles the training process of the U-Net model.
* from prediction\_denoise import prediction: This imports the prediction function from the prediction\_denoise.py module. This function is responsible for making predictions (denoising audio) using the trained U-Net model.

if \_\_name\_\_ == '\_\_main\_\_':

**Main Entry Point**:

* This line is a standard Python convention. The code inside this block will only be executed if the script is run directly (not imported as a module). This is useful for organizing code that should only be executed when the script is intended to run as the main program.

args = parser.parse\_args()

**Parse Command-Line Arguments**:

* args = parser.parse\_args(): This line parses the command-line arguments provided by the user when running the script. The args object will contain all the argument values defined in args.py. For example, args.mode will hold the value of the --mode argument provided by the user.

mode = args.mode

**Extract Mode**:

* mode = args.mode: This line extracts the value of the mode argument from the args object. The mode specifies which operation the user wants to perform: data creation, training, or prediction.

data\_mode = False

training\_mode = False

prediction\_mode = False

**Initialize Mode Flags**:

* These lines initialize three flags (data\_mode, training\_mode, and prediction\_mode) to False. These flags will later be used to determine which mode the script should operate in.

if mode == 'prediction':

prediction\_mode = True

elif mode == 'training':

training\_mode = True

elif mode == 'data\_creation':

data\_mode = True

**Set Mode Flags Based on User Input**:

* This block checks the value of mode and sets the corresponding flag to True. For example, if mode is 'prediction', prediction\_mode will be set to True, and the script will later execute the code related to prediction.

if data\_mode:

**Data Creation Mode**:

* This checks if the data\_mode flag is True, meaning the user wants to create data for training.

noise\_dir = args.noise\_dir

voice\_dir = args.voice\_dir

path\_save\_time\_serie = args.path\_save\_time\_serie

path\_save\_sound = args.path\_save\_sound

path\_save\_spectrogram = args.path\_save\_spectrogram

sample\_rate = args.sample\_rate

min\_duration = args.min\_duration

frame\_length = args.frame\_length

hop\_length\_frame = args.hop\_length\_frame

hop\_length\_frame\_noise = args.hop\_length\_frame\_noise

nb\_samples = args.nb\_samples

n\_fft = args.n\_fft

hop\_length\_fft = args.hop\_length\_fft

**Extract Data Creation Arguments**:

* These lines extract various arguments from the args object that are necessary for data creation:

noise\_dir: Directory containing noise audio files.

voice\_dir: Directory containing clean voice audio files.

path\_save\_time\_serie: Path to save the time series data.

path\_save\_sound: Path to save the generated audio files.

path\_save\_spectrogram: Path to save the spectrograms.

sample\_rate: Sample rate for reading audio files.

min\_duration: Minimum duration of audio files to be considered.

frame\_length: Frame length for segmenting audio.

hop\_length\_frame: Hop length for creating frames from clean voice files.

hop\_length\_frame\_noise: Hop length for creating frames from noise files.

nb\_samples: Number of samples to create.

n\_fft: Number of points for the Fast Fourier Transform (FFT) for spectrogram computation.

hop\_length\_fft: Hop length for FFT.

create\_data(noise\_dir, voice\_dir, path\_save\_time\_serie, path\_save\_sound, path\_save\_spectrogram, sample\_rate,

min\_duration, frame\_length, hop\_length\_frame, hop\_length\_frame\_noise, nb\_samples, n\_fft, hop\_length\_fft)

**Create Data**:

* This line calls the create\_data function with the extracted arguments. This function will blend noise with clean voice data, create spectrograms, and save the data to the specified locations.

elif training\_mode:

**Training Mode**:

* This checks if the training\_mode flag is True, meaning the user wants to train the model.

path\_save\_spectrogram = args.path\_save\_spectrogram

weights\_path = args.weights\_folder

name\_model = args.name\_model

training\_from\_scratch = args.training\_from\_scratch

epochs = args.epochs

batch\_size = args.batch\_size

**Extract Training Arguments**:

* These lines extract various arguments from the args object that are necessary for training:

path\_save\_spectrogram: Path where spectrograms of noisy and clean voices are saved.

weights\_path: Path to find pre-trained weights or save trained models.

name\_model: Name of the model to use or save.

training\_from\_scratch: Boolean indicating whether to train from scratch or use pre-trained weights.

epochs: Number of epochs for training.

batch\_size: Batch size for training.

training(path\_save\_spectrogram, weights\_path, name\_model, training\_from\_scratch, epochs, batch\_size)

**Train Model**:

* This line calls the training function with the extracted arguments. This function will train the U-Net model using the provided spectrogram data, save the best-performing models, and potentially use pre-trained weights if specified.

elif prediction\_mode:

**Prediction Mode**:

* This checks if the prediction\_mode flag is True, meaning the user wants to make predictions (denoise audio).

weights\_path = args.weights\_folder

name\_model = args.name\_model

audio\_dir\_prediction = args.audio\_dir\_prediction

dir\_save\_prediction = args.dir\_save\_prediction

audio\_input\_prediction = args.audio\_input\_prediction

audio\_output\_prediction = args.audio\_output\_prediction

sample\_rate = args.sample\_rate

min\_duration = args.min\_duration

frame\_length = args.frame\_length

hop\_length\_frame = args.hop\_length\_frame

n\_fft = args.n\_fft

hop\_length\_fft = args.hop\_length\_fft

**Extract Prediction Arguments**:

* These lines extract various arguments from the args object that are necessary for prediction:

weights\_path: Path to find pre-trained weights.

name\_model: Name of the model to use for prediction.

audio\_dir\_prediction: Directory containing the noisy audio files to denoise.

dir\_save\_prediction: Directory to save the denoised audio files.

audio\_input\_prediction: List of noisy audio files to denoise.

audio\_output\_prediction: Name of the denoised audio file to save.

sample\_rate: Sample rate for reading audio files.

min\_duration: Minimum duration of audio files to be considered.

frame\_length: Frame length for segmenting audio during prediction.

hop\_length\_frame: Hop length for creating frames from audio during prediction.

n\_fft: Number of points for FFT for spectrogram computation.

hop\_length\_fft: Hop length for FFT.

prediction(weights\_path, name\_model, audio\_dir\_prediction, dir\_save\_prediction, audio\_input\_prediction,

audio\_output\_prediction, sample\_rate, min\_duration, frame\_length, hop\_length\_frame, n\_fft, hop\_length\_fft)

**Make Predictions**:

* This line calls the prediction function with the extracted arguments. This function will load the pre-trained U-Net model, denoise the specified audio files, and save the results to the specified location.

**args.py file:**

import argparse

**Import Argument Parsing Module**:

* **import argparse:** This line imports the **argparse module**, which is a standard library in Python used for creating command-line interfaces. It allows you to define what arguments your program requires, assign them to variables, and provides automatic help messages.

parser = argparse.ArgumentParser(description='Speech Enhancement using U-Net')

**Initialize Argument Parser**:

* parser = argparse.ArgumentParser(description='Speech Enhancement using U-Net'): This line creates an ArgumentParser object named parser. The description parameter provides a brief description of the program, which will be shown when the user requests help using the -h or --help option.

parser.add\_argument('--mode', type=str, default='data\_creation',

help='Mode of operation: data\_creation, training or prediction')

**Mode Argument**:

* parser.add\_argument('--mode', type=str, default='data\_creation', help='Mode of operation: data\_creation, training or prediction'):

This line adds an argument named --mode to the parser.

type=str specifies that the value provided for this argument should be a string.

default='data\_creation' sets the default value to 'data\_creation' if the user does not provide a value.

help='Mode of operation: data\_creation, training or prediction' provides a help message describing what this argument does.

parser.add\_argument('--noise\_dir', type=str, default='./data/Noise', help='Directory of noise files')

parser.add\_argument('--voice\_dir', type=str, default='./data/Voice', help='Directory of clean voice files')

parser.add\_argument('--path\_save\_time\_serie', type=str, default='./data/time\_serie', help='Path to save time series data')

parser.add\_argument('--path\_save\_sound', type=str, default='./data/sound', help='Path to save generated sound files')

parser.add\_argument('--path\_save\_spectrogram', type=str, default='./data/spectrogram', help='Path to save spectrograms')

**Data Paths and Directories**:

* These lines add arguments to specify the paths for various data-related directories:

--noise\_dir: Directory containing noise audio files.

--voice\_dir: Directory containing clean voice audio files.

--path\_save\_time\_serie: Directory to save the generated time series data (blended noisy and clean voices).

--path\_save\_sound: Directory to save the generated noisy audio files.

--path\_save\_spectrogram: Directory to save the spectrograms of noisy and clean audio.

parser.add\_argument('--sample\_rate', type=int, default=16000, help='Sampling rate for audio files')

parser.add\_argument('--min\_duration', type=float, default=1.0, help='Minimum duration of audio files in seconds')

**Audio Processing Parameters**:

* These lines add arguments to specify parameters related to audio processing:

--sample\_rate: Sampling rate for reading audio files (default is 16,000 Hz).

--min\_duration: Minimum duration (in seconds) for an audio file to be considered valid for processing.

parser.add\_argument('--frame\_length', type=int, default=8064, help='Frame length for audio segmentation')

parser.add\_argument('--hop\_length\_frame', type=int, default=8064, help='Hop length for creating frames from clean voice')

parser.add\_argument('--hop\_length\_frame\_noise', type=int, default=500, help='Hop length for creating frames from noise')

**Frame Segmentation Parameters**:

* These lines add arguments related to how the audio is segmented into frames:

--frame\_length: The length of each frame in samples (default is 8064 samples).

--hop\_length\_frame: The hop length (number of samples to move forward) when creating frames from clean voice data.

--hop\_length\_frame\_noise: The hop length when creating frames from noise data.

parser.add\_argument('--nb\_samples', type=int, default=5000, help='Number of samples to create for training')

**Number of Samples**:

* This line adds an argument to specify the number of samples to create for training:

--nb\_samples: The number of noisy-clean audio pairs to generate (default is 5000).

parser.add\_argument('--n\_fft', type=int, default=255, help='Number of FFT points for spectrogram computation')

parser.add\_argument('--hop\_length\_fft', type=int, default=63, help='Hop length for FFT in spectrogram computation')

**FFT and Spectrogram Parameters**:

* These lines add arguments related to the Fast Fourier Transform (FFT) used in creating spectrograms:

--n\_fft: The number of FFT points to use in the spectrogram computation (default is 255).

--hop\_length\_fft: The hop length for the FFT, determining the overlap between consecutive frames in the spectrogram.

parser.add\_argument('--weights\_folder', type=str, default='./weights', help='Directory to save or load model weights')

parser.add\_argument('--name\_model', type=str, default='model\_unet', help='Name of the model to save or load')

parser.add\_argument('--training\_from\_scratch', type=bool, default=True, help='Whether to train the model from scratch or use pre-trained weights')

parser.add\_argument('--epochs', type=int, default=50, help='Number of epochs for training')

parser.add\_argument('--batch\_size', type=int, default=16, help='Batch size for training')

**Training Parameters**:

* These lines add arguments related to the training process:

--weights\_folder: Directory to save or load model weights.

--name\_model: Name of the model file to save or load (default is 'model\_unet').

--training\_from\_scratch: Boolean flag to indicate whether to train the model from scratch (True) or to use pre-trained weights (False).

--epochs: Number of training epochs (default is 50).

--batch\_size: Batch size for training (default is 16).

parser.add\_argument('--audio\_dir\_prediction', type=str, default='./data/prediction/input/', help='Directory of noisy audio files for prediction')

parser.add\_argument('--dir\_save\_prediction', type=str, default='./data/prediction/output/', help='Directory to save denoised audio files')

parser.add\_argument('--audio\_input\_prediction', type=str, default='./data/prediction/input/input.wav', help='Noisy audio file for prediction')

parser.add\_argument('--audio\_output\_prediction', type=str, default='./data/prediction/output/output.wav', help='File to save the denoised output')

**Prediction Parameters**:

* These lines add arguments related to the prediction (denoising) process:

--audio\_dir\_prediction: Directory containing noisy audio files for prediction.

--dir\_save\_prediction: Directory to save the denoised audio files.

--audio\_input\_prediction: Specific noisy audio file to denoise (default is input.wav).

--audio\_output\_prediction: File to save the denoised audio output (default is output.wav).

**prepare\_data.py file:**

import os

import librosa

import numpy as np

import soundfile as sf

from tqdm import tqdm

**Import Necessary Libraries**:

* import os: This imports the os module, which provides a way to interact with the operating system, such as navigating the file system.
* import librosa: This imports librosa, a powerful Python library for audio and music analysis, including loading audio files, computing spectrograms, and more.
* import numpy as np: This imports numpy, a fundamental package for scientific computing with Python, especially useful for handling arrays and matrices.
* import soundfile as sf: This imports the soundfile library, which is used for reading and writing sound files (e.g., WAV files).
* import tqdm: This imports tqdm, a library that provides a fast, extensible progress bar for loops and other iterable tasks.

def save\_sound(path, sample, sample\_rate):

sf.write(path, sample, sample\_rate)

**Define save\_sound Function**:

* def save\_sound(path, sample, sample\_rate): This line defines a function named save\_sound that saves an audio sample to a specified path.
* sf.write(path, sample, sample\_rate): This line uses the soundfile library's write function to save the audio data (sample) to the given path with the specified sample\_rate.

def add\_noise(sample, noise, factor):

return sample + factor \* noise

**Define add\_noise Function**:

* def add\_noise(sample, noise, factor): This line defines a function named add\_noise that adds noise to an audio sample.
* return sample + factor \* noise: This line adds the noise (scaled by factor) to the original sample and returns the resulting noisy sample.

def prepare\_data(args):

if not os.path.exists(args.path\_save\_time\_serie):

os.makedirs(args.path\_save\_time\_serie)

**Define prepare\_data Function and Create Directory**:

* def prepare\_data(args): This line defines the main function prepare\_data, which prepares the dataset by combining voice and noise samples.
* if not os.path.exists(args.path\_save\_time\_serie): This checks if the directory specified by args.path\_save\_time\_serie exists.
* os.makedirs(args.path\_save\_time\_serie): If the directory does not exist, this line creates it.

voice\_files = librosa.util.find\_files(args.voice\_dir, ext=['wav'])

noise\_files = librosa.util.find\_files(args.noise\_dir, ext=['wav'])

**Find Voice and Noise Files**:

* voice\_files = librosa.util.find\_files(args.voice\_dir, ext=['wav']): This line finds all .wav files in the directory specified by args.voice\_dir using librosa and stores them in voice\_files.
* noise\_files = librosa.util.find\_files(args.noise\_dir, ext=['wav']): This line finds all .wav files in the directory specified by args.noise\_dir and stores them in noise\_files.

for i, voice\_file in enumerate(tqdm(voice\_files, desc="Processing voice files")):

voice, \_ = librosa.load(voice\_file, sr=args.sample\_rate)

**Process Each Voice File**:

* for i, voice\_file in enumerate(tqdm(voice\_files, desc="Processing voice files")): This line starts a loop to process each voice file. The tqdm wrapper displays a progress bar for the loop with the description "Processing voice files".
* voice, \_ = librosa.load(voice\_file, sr=args.sample\_rate): This line loads the voice audio file using librosa at the sample rate specified by args.sample\_rate. The audio data is stored in voice, and the sample rate is discarded (\_).

if len(voice) < args.min\_duration \* args.sample\_rate:

continue

**Skip Short Audio Files**:

* if len(voice) < args.min\_duration \* args.sample\_rate: This checks if the length of the loaded voice file is shorter than the minimum duration specified by args.min\_duration (converted to samples).
* continue: If the voice file is too short, the loop skips to the next file without processing further.

for j, noise\_file in enumerate(noise\_files):

noise, \_ = librosa.load(noise\_file, sr=args.sample\_rate)

**Process Each Noise File**:

* for j, noise\_file in enumerate(noise\_files): This line starts another loop to process each noise file for the current voice file.
* noise, \_ = librosa.load(noise\_file, sr=args.sample\_rate): This line loads the noise audio file using librosa at the sample rate specified by args.sample\_rate. The audio data is stored in noise.

if len(noise) < args.min\_duration \* args.sample\_rate:

continue

**Skip Short Noise Files**:

* if len(noise) < args.min\_duration \* args.sample\_rate: This checks if the noise file is shorter than the minimum duration.
* continue: If the noise file is too short, the loop skips to the next noise file without processing further.

noise\_segment = noise[:len(voice)]

noisy\_voice = add\_noise(voice, noise\_segment, factor=np.random.uniform(0.2, 0.8))

**Add Noise to Voice**:

* noise\_segment = noise[:len(voice)]: This line trims the noise file to match the length of the voice file. Only the first part of the noise file is used, which corresponds to the length of the voice file.
* noisy\_voice = add\_noise(voice, noise\_segment, factor=np.random.uniform(0.2, 0.8)): This line calls the add\_noise function to add the trimmed noise to the voice. The factor parameter, which scales the noise level, is chosen randomly between 0.2 and 0.8.

save\_sound(os.path.join(args.path\_save\_time\_serie, f'noisy\_{i}\_{j}.wav'), noisy\_voice, args.sample\_rate)

save\_sound(os.path.join(args.path\_save\_time\_serie, f'clean\_{i}\_{j}.wav'), voice, args.sample\_rate)

**Save Noisy and Clean Samples**:

* save\_sound(os.path.join(args.path\_save\_time\_serie, f'noisy\_{i}\_{j}.wav'), noisy\_voice, args.sample\_rate): This line saves the noisy voice sample to a file named noisy\_{i}\_{j}.wav in the directory specified by args.path\_save\_time\_serie.
* save\_sound(os.path.join(args.path\_save\_time\_serie, f'clean\_{i}\_{j}.wav'), voice, args.sample\_rate): This line saves the corresponding clean voice sample to a file named clean\_{i}\_{j}.wav in the same directory.

if len(os.listdir(args.path\_save\_time\_serie)) >= args.nb\_samples \* 2:

break

if len(os.listdir(args.path\_save\_time\_serie)) >= args.nb\_samples \* 2:

break

**Check Sample Count and Break Loop**:

* if len(os.listdir(args.path\_save\_time\_serie)) >= args.nb\_samples \* 2: This checks if the number of generated files in the args.path\_save\_time\_serie directory has reached or exceeded the required number of samples (args.nb\_samples for clean and noisy files, hence multiplied by 2).
* break: If the condition is met, the inner loop (for noise files) breaks. The outer loop (for voice files) also breaks if the condition is met again, stopping further processing.

**data\_tools.py file:**

import os

import numpy as np

import librosa

import torch

from torch.utils.data import Dataset, DataLoader

**Import Necessary Libraries**:

* import os: This imports the os module, which allows interaction with the operating system, such as file and directory management.
* import numpy as np: This imports numpy, a library for numerical operations, especially useful for handling arrays.
* import librosa: This imports librosa, a library for audio and music analysis.
* import torch: This imports torch, the core library of PyTorch, used for building and training deep learning models.
* from torch.utils.data import Dataset, DataLoader: This imports the Dataset and DataLoader classes from torch.utils.data, which are essential for handling data in PyTorch. Dataset is an abstract class representing a dataset, while DataLoader provides an iterable over a dataset.

class AudioDataset(Dataset):

def \_\_init\_\_(self, clean\_dir, noisy\_dir, transform=None):

self.clean\_dir = clean\_dir

self.noisy\_dir = noisy\_dir

self.clean\_files = librosa.util.find\_files(clean\_dir, ext=['wav'])

self.noisy\_files = librosa.util.find\_files(noisy\_dir, ext=['wav'])

self.transform = transform

**Define AudioDataset Class (Initialization)**:

* class AudioDataset(Dataset): This line defines a new class AudioDataset that inherits from PyTorch's Dataset class. It represents a dataset of paired clean and noisy audio files.
* def \_\_init\_\_(self, clean\_dir, noisy\_dir, transform=None): This line defines the constructor (\_\_init\_\_) method, which initializes the dataset object.
* self.clean\_dir = clean\_dir: This stores the directory path containing clean audio files in the clean\_dir attribute.
* self.noisy\_dir = noisy\_dir: This stores the directory path containing noisy audio files in the noisy\_dir attribute.
* self.clean\_files = librosa.util.find\_files(clean\_dir, ext=['wav']): This line finds all .wav files in the clean\_dir using librosa and stores their paths in self.clean\_files.
* self.noisy\_files = librosa.util.find\_files(noisy\_dir, ext=['wav']): Similarly, this finds all .wav files in the noisy\_dir and stores their paths in self.noisy\_files.
* self.transform = transform: This stores any optional data transformations to be applied to the audio samples in the transform attribute.

def \_\_len\_\_(self):

return len(self.clean\_files)

**Define \_\_len\_\_ Method**:

* def \_\_len\_\_(self): This method returns the total number of samples in the dataset. It is a standard method that must be implemented for PyTorch datasets.
* return len(self.clean\_files): This returns the number of clean audio files, which is assumed to be the same as the number of noisy files (i.e., the dataset is paired).

def \_\_getitem\_\_(self, idx):

clean\_file = self.clean\_files[idx]

noisy\_file = self.noisy\_files[idx]

clean, \_ = librosa.load(clean\_file, sr=None)

noisy, \_ = librosa.load(noisy\_file, sr=None)

**Define \_\_getitem\_\_ Method (Loading Files)**:

* def \_\_getitem\_\_(self, idx): This method retrieves the clean and noisy audio samples at a given index idx. This method must also be implemented for PyTorch datasets.
* clean\_file = self.clean\_files[idx]: This retrieves the path of the clean audio file at index idx from self.clean\_files.
* noisy\_file = self.noisy\_files[idx]: Similarly, this retrieves the path of the noisy audio file at index idx from self.noisy\_files.
* clean, \_ = librosa.load(clean\_file, sr=None): This loads the clean audio file using librosa. The sr=None parameter ensures that the original sampling rate of the file is used. The audio data is stored in clean, and the sampling rate is ignored (\_).
* noisy, \_ = librosa.load(noisy\_file, sr=None): Similarly, this loads the noisy audio file using librosa.

if self.transform:

clean = self.transform(clean)

noisy = self.transform(noisy)

**Apply Transformations (if any)**:

* if self.transform: This checks if any transformations were provided during the initialization of the dataset.
* clean = self.transform(clean): If a transformation is provided, it is applied to the clean audio data.
* noisy = self.transform(noisy): Similarly, the transformation is applied to the noisy audio data.

clean = torch.from\_numpy(clean).float()

noisy = torch.from\_numpy(noisy).float()

**Convert to PyTorch Tensors**:

* clean = torch.from\_numpy(clean).float(): This converts the clean audio data, which is in the form of a NumPy array, into a PyTorch tensor of type float.
* noisy = torch.from\_numpy(noisy).float(): Similarly, this converts the noisy audio data into a PyTorch tensor of type float.

return noisy, clean

**Return Noisy and Clean Pairs**:

* return noisy, clean: This returns a tuple containing the noisy and clean audio tensors. In PyTorch, this tuple is used as the input and target (label) for the model.

def create\_dataloader(clean\_dir, noisy\_dir, batch\_size=32, shuffle=True, num\_workers=0, transform=None):

dataset = AudioDataset(clean\_dir, noisy\_dir, transform=transform)

return DataLoader(dataset, batch\_size=batch\_size, shuffle=shuffle, num\_workers=num\_workers)

**Define create\_dataloader Function**:

* def create\_dataloader(clean\_dir, noisy\_dir, batch\_size=32, shuffle=True, num\_workers=0, transform=None): This line defines a function create\_dataloader that creates and returns a DataLoader object for the audio dataset.
* dataset = AudioDataset(clean\_dir, noisy\_dir, transform=transform): This creates an instance of the AudioDataset class using the provided directories for clean and noisy files, and any optional transformations.
* return DataLoader(dataset, batch\_size=batch\_size, shuffle=shuffle, num\_workers=num\_workers): This creates and returns a DataLoader for the dataset. The DataLoader provides an efficient way to iterate over the dataset in batches (batch\_size) and optionally shuffles the data (shuffle). num\_workers specifies the number of subprocesses to use for data loading.

**train\_model.py file:**

import torch

import torch.nn as nn

import torch.optim as optim

from tqdm import tqdm

import os

**Import Necessary Libraries**:

* import torch: This imports the core PyTorch library, which is used for building and training neural networks.
* import torch.nn as nn: This imports the nn module from PyTorch, which contains classes for building neural network layers, loss functions, etc.
* import torch.optim as optim: This imports the optim module from PyTorch, which contains classes for optimization algorithms like SGD, Adam, etc.
* from tqdm import tqdm: This imports the tqdm library, which is used to create progress bars, helping visualize the progress of training loops.
* import os: This imports the os module, which allows interaction with the operating system, such as file and directory management.

def train\_model(model, train\_loader, criterion, optimizer, num\_epochs=25, device='cuda', save\_dir='./model'):

model = model.to(device)

best\_loss = float('inf')

**Define train\_model Function (Initialization)**:

* def train\_model(model, train\_loader, criterion, optimizer, num\_epochs=25, device='cuda', save\_dir='./model'): This line defines a function train\_model that trains a neural network model. It takes several parameters:

model: The neural network model to be trained.

train\_loader: The DataLoader object that provides the training data.

criterion: The loss function used to calculate the error.

optimizer: The optimization algorithm used to update the model's parameters.

num\_epochs: The number of training epochs (default is 25).

device: The device on which the model will be trained (default is 'cuda' for GPU).

save\_dir: The directory where the best model (with the lowest loss) will be saved.

* model = model.to(device): This moves the model to the specified device (e.g., GPU or CPU).
* best\_loss = float('inf'): This initializes best\_loss to infinity. This variable will track the lowest loss encountered during training.

if not os.path.exists(save\_dir):

os.makedirs(save\_dir)

**Create Directory for Saving Model**:

* if not os.path.exists(save\_dir): This checks if the directory specified by save\_dir exists.
* os.makedirs(save\_dir): If the directory does not exist, this creates it. This ensures that the model can be saved in the specified location.

for epoch in range(num\_epochs):

model.train()

running\_loss = 0.0

for inputs, targets in tqdm(train\_loader):

inputs, targets = inputs.to(device), targets.to(device)

**Training Loop (Begin Epoch)**:

* for epoch in range(num\_epochs): This loop iterates over each epoch. An epoch is one complete pass through the entire training dataset.
* model.train(): This sets the model to training mode. This is important because some layers (e.g., dropout, batch normalization) behave differently during training and evaluation.
* running\_loss = 0.0: This initializes running\_loss to zero. This variable will accumulate the loss over the epoch.
* for inputs, targets in tqdm(train\_loader): This loops over the batches of data provided by train\_loader. The tqdm function wraps the loop to display a progress bar.
* inputs, targets = inputs.to(device), targets.to(device): This moves the inputs and targets (labels) to the specified device (e.g., GPU).

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, targets)

**Forward Pass and Loss Calculation**:

* optimizer.zero\_grad(): This clears the gradients of all optimized model parameters. This is essential because, by default, gradients in PyTorch accumulate after each backward pass.
* outputs = model(inputs): This performs the forward pass, passing the inputs through the model to obtain the outputs (predictions).
* loss = criterion(outputs, targets): This calculates the loss by comparing the model's outputs with the true targets using the specified criterion (loss function).

loss.backward()

optimizer.step()

**Backward Pass and Optimization Step**:

* loss.backward(): This performs the backward pass, computing the gradient of the loss with respect to the model parameters.
* optimizer.step(): This updates the model parameters using the calculated gradients and the optimization algorithm defined by optimizer.

running\_loss += loss.item()

**Accumulate Loss**:

* running\_loss += loss.item(): This adds the current batch's loss to running\_loss. The .item() method extracts the scalar value of the loss from the PyTorch tensor.

epoch\_loss = running\_loss / len(train\_loader)

print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {epoch\_loss:.4f}')

**Calculate and Print Epoch Loss**:

* epoch\_loss = running\_loss / len(train\_loader): This calculates the average loss for the epoch by dividing the total running loss by the number of batches in train\_loader.
* print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {epoch\_loss:.4f}'): This prints the current epoch number and the average loss for that epoch to the console, formatted to four decimal places.

if epoch\_loss < best\_loss:

best\_loss = epoch\_loss

torch.save(model.state\_dict(), os.path.join(save\_dir, 'best\_model.pth'))

print('Model saved.')

**Save Best Model**:

* if epoch\_loss < best\_loss: This checks if the current epoch's loss is lower than the best loss encountered so far.
* best\_loss = epoch\_loss: If the current loss is the lowest, this updates best\_loss to the current epoch's loss.
* torch.save(model.state\_dict(), os.path.join(save\_dir, 'best\_model.pth')): This saves the model's state (its parameters) to a file named best\_model.pth in the save\_dir directory. The state\_dict() method returns a dictionary containing the model's parameters.
* print('Model saved.'): This prints a message indicating that the model has been saved.

print('Training complete.')

**End of Training**:

* print('Training complete.'): After all epochs are completed, this line prints a message indicating that the training process is complete.

**model\_unet.py file:**

import torch

import torch.nn as nn

import torch.nn.functional as F

**Import Necessary Libraries**:

* import torch: This imports the core PyTorch library, which is used for building and training neural networks.
* import torch.nn as nn: This imports the nn module from PyTorch, which provides classes for building neural network layers, including convolutional layers, pooling layers, etc.
* import torch.nn.functional as F: This imports the F module, which contains functions for operations that don't have learnable parameters (like ReLU, max pooling, etc.), and provides an interface for applying various functions to tensors.

class DoubleConv(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super(DoubleConv, self).\_\_init\_\_()

self.conv = nn.Sequential(

nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1),

nn.ReLU(inplace=True))

**Define DoubleConv Class**:

* class DoubleConv(nn.Module): This defines a class DoubleConv that inherits from nn.Module. It represents a building block for the U-Net model, consisting of two convolutional layers followed by ReLU activation functions.
* def \_\_init\_\_(self, in\_channels, out\_channels): This is the constructor for the DoubleConv class. It takes two arguments: in\_channels (number of input channels) and out\_channels (number of output channels).
* super(DoubleConv, self).\_\_init\_\_(): This calls the constructor of the parent class nn.Module, initializing the base module.
* self.conv = nn.Sequential(...): This defines a sequential container self.conv that contains the layers of the DoubleConv block:
  + nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1): This is the first 2D convolutional layer with a 3x3 kernel and padding of 1, which maintains the spatial dimensions.
  + nn.ReLU(inplace=True): This applies the ReLU activation function to the output of the first convolution. inplace=True means that the operation is done in-place, saving memory.
  + nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1): This is the second 2D convolutional layer, with the same kernel size and padding.
  + nn.ReLU(inplace=True): This applies the ReLU activation function to the output of the second convolution.

def forward(self, x):

return self.conv(x)

**Forward Method of DoubleConv**:

* def forward(self, x): This defines the forward pass for the DoubleConv block. The x parameter represents the input tensor.
* return self.conv(x): This applies the sequential conv layers defined in the constructor to the input tensor x and returns the result. This effectively performs two convolutional operations followed by ReLU activations.

class UNet(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super(UNet, self).\_\_init\_\_()

**Define UNet Class**:

* class UNet(nn.Module): This defines a class UNet that inherits from nn.Module. It represents the U-Net architecture for image segmentation.
* def \_\_init\_\_(self, in\_channels, out\_channels): This is the constructor for the UNet class. It takes two arguments: in\_channels (number of input channels) and out\_channels (number of output channels, typically corresponding to the number of classes).
* super(UNet, self).\_\_init\_\_(): This calls the constructor of the parent class nn.Module, initializing the base module.

self.down1 = DoubleConv(in\_channels, 64)

self.pool1 = nn.MaxPool2d(2)

self.down2 = DoubleConv(64, 128)

self.pool2 = nn.MaxPool2d(2)

self.down3 = DoubleConv(128, 256)

self.pool3 = nn.MaxPool2d(2)

self.down4 = DoubleConv(256, 512)

self.pool4 = nn.MaxPool2d(2)

self.bridge = DoubleConv(512, 1024)

**Downsampling Path (Contracting Path)**:

* self.down1 = DoubleConv(in\_channels, 64): This defines the first downsampling block, consisting of two convolutional layers with 64 output channels. The input channels are specified by in\_channels.
* self.pool1 = nn.MaxPool2d(2): This applies 2x2 max pooling after the first downsampling block, reducing the spatial dimensions by half.
* self.down2 = DoubleConv(64, 128): This defines the second downsampling block, with 128 output channels.
* self.pool2 = nn.MaxPool2d(2): This applies 2x2 max pooling after the second downsampling block.
* self.down3 = DoubleConv(128, 256): This defines the third downsampling block, with 256 output channels.
* self.pool3 = nn.MaxPool2d(2): This applies 2x2 max pooling after the third downsampling block.
* self.down4 = DoubleConv(256, 512): This defines the fourth downsampling block, with 512 output channels.
* self.pool4 = nn.MaxPool2d(2): This applies 2x2 max pooling after the fourth downsampling block.
* self.bridge = DoubleConv(512, 1024): This defines the "bridge" (bottom layer of the U-Net), with 1024 output channels. It connects the contracting path with the expanding path.

self.up1 = nn.ConvTranspose2d(1024, 512, kernel\_size=2, stride=2)

self.conv1 = DoubleConv(1024, 512)

self.up2 = nn.ConvTranspose2d(512, 256, kernel\_size=2, stride=2)

self.conv2 = DoubleConv(512, 256)

self.up3 = nn.ConvTranspose2d(256, 128, kernel\_size=2, stride=2)

self.conv3 = DoubleConv(256, 128)

self.up4 = nn.ConvTranspose2d(128, 64, kernel\_size=2, stride=2)

self.conv4 = DoubleConv(128, 64)

self.out\_conv = nn.Conv2d(64, out\_channels, kernel\_size=1)

**Upsampling Path (Expanding Path)**:

* self.up1 = nn.ConvTranspose2d(1024, 512, kernel\_size=2, stride=2): This defines the first upsampling layer using a transposed convolution (also called deconvolution). It upsamples the feature map from 1024 channels to 512 channels, doubling the spatial dimensions.
* self.conv1 = DoubleConv(1024, 512): This defines the convolutional block after the first upsampling layer. The input to this block is concatenated from the output of the first upsampling layer (512 channels) and the corresponding feature map from the contracting path (512 channels), making a total of 1024 input channels.
* self.up2 = nn.ConvTranspose2d(512, 256, kernel\_size=2, stride=2): This defines the second upsampling layer, upsizing from 512 channels to 256 channels.
* self.conv2 = DoubleConv(512, 256): This defines the convolutional block after the second upsampling layer. The input is concatenated from the output of the second upsampling layer and the corresponding feature map from the contracting path.
* self.up3 = nn.ConvTranspose2d(256, 128, kernel\_size=2, stride=2): This defines the third upsampling layer, upsizing from 256 channels to 128 channels.
* self.conv3 = DoubleConv(256, 128): This defines the convolutional block after the third upsampling layer, with 256 input channels.
* self.up4 = nn.ConvTranspose2d(128, 64, kernel\_size=2, stride=2): This defines the fourth upsampling layer, upsizing from 128 channels to 64 channels.
* self.conv4 = DoubleConv(128, 64): This defines the convolutional block after the fourth upsampling layer, with 128 input channels.
* self.out\_conv = nn.Conv2d(64, out\_channels, kernel\_size=1): This defines the final output layer, which is a 1x1 convolution that reduces the number of channels from 64 to out\_channels (the number of output classes).

def forward(self, x):

down1 = self.down1(x)

pool1 = self.pool1(down1)

down2 = self.down2(pool1)

pool2 = self.pool2(down2)

down3 = self.down3(pool2)

pool3 = self.pool3(down3)

down4 = self.down4(pool3)

pool4 = self.pool4(down4)

bridge = self.bridge(pool4)

**Forward Method (Downsampling Path)**:

* def forward(self, x): This defines the forward pass for the UNet model. The x parameter represents the input tensor.
* down1 = self.down1(x): This applies the first downsampling block to the input x.
* pool1 = self.pool1(down1): This applies max pooling to the output of the first downsampling block.
* down2 = self.down2(pool1): This applies the second downsampling block to the output of the first pooling layer.
* pool2 = self.pool2(down2): This applies max pooling to the output of the second downsampling block.
* down3 = self.down3(pool2): This applies the third downsampling block to the output of the second pooling layer.
* pool3 = self.pool3(down3): This applies max pooling to the output of the third downsampling block.
* down4 = self.down4(pool3): This applies the fourth downsampling block to the output of the third pooling layer.
* pool4 = self.pool4(down4): This applies max pooling to the output of the fourth downsampling block.
* bridge = self.bridge(pool4): This applies the bridge (bottom layer of U-Net) to the output of the fourth pooling layer.

up1 = self.up1(bridge)

up1 = torch.cat([up1, down4], dim=1)

up1 = self.conv1(up1)

up2 = self.up2(up1)

up2 = torch.cat([up2, down3], dim=1)

up2 = self.conv2(up2)

up3 = self.up3(up2)

up3 = torch.cat([up3, down2], dim=1)

up3 = self.conv3(up3)

up4 = self.up4(up3)

up4 = torch.cat([up4, down1], dim=1)

up4 = self.conv4(up4)

out = self.out\_conv(up4)

**Forward Method (Upsampling Path)**:

* up1 = self.up1(bridge): This applies the first upsampling layer to the output of the bridge.
* up1 = torch.cat([up1, down4], dim=1): This concatenates the upsampled feature map with the corresponding feature map from the downsampling path (down4) along the channel dimension (dim=1).
* up1 = self.conv1(up1): This applies the first convolutional block in the upsampling path to the concatenated feature map.
* up2 = self.up2(up1): This applies the second upsampling layer to the output of the first convolutional block in the upsampling path.
* up2 = torch.cat([up2, down3], dim=1): This concatenates the upsampled feature map with the corresponding feature map from the downsampling path (down3).
* up2 = self.conv2(up2): This applies the second convolutional block in the upsampling path.
* up3 = self.up3(up2): This applies the third upsampling layer.
* up3 = torch.cat([up3, down2], dim=1): This concatenates the upsampled feature map with the corresponding feature map from the downsampling path (down2).
* up3 = self.conv3(up3): This applies the third convolutional block in the upsampling path.
* up4 = self.up4(up3): This applies the fourth upsampling layer.
* up4 = torch.cat([up4, down1], dim=1): This concatenates the upsampled feature map with the corresponding feature map from the downsampling path (down1).
* up4 = self.conv4(up4): This applies the fourth convolutional block in the upsampling path.
* out = self.out\_conv(up4): This applies the final output convolution to reduce the number of channels to out\_channels, producing the final segmentation map.

return out

**Return Output**:

* return out: This returns the final output tensor, which represents the predicted segmentation map.

**prediction\_denoise.py file:**

import os

import torch

import torchaudio

import argparse

from model\_unet import UNet

**Import Necessary Libraries**:

* import os: This imports Python's built-in os module, which provides a way to interact with the operating system, including file and directory operations.
* import torch: This imports the core PyTorch library, used for tensor operations and building neural networks.
* import torchaudio: This imports the torchaudio library, which provides tools for working with audio data, including loading and processing audio files.
* import argparse: This imports the argparse module, used for parsing command-line arguments.
* from model\_unet import UNet: This imports the UNet class from the model\_unet.py file, which defines the U-Net architecture used for denoising.

def load\_model(model\_path, device):

model = UNet(in\_channels=1, out\_channels=1)

model.load\_state\_dict(torch.load(model\_path, map\_location=device))

model.to(device)

model.eval()

return model

**Define load\_model Function**:

* def load\_model(model\_path, device): This function loads a pre-trained U-Net model from the specified path and prepares it for inference.
* model = UNet(in\_channels=1, out\_channels=1): This creates an instance of the UNet model with one input channel (mono audio) and one output channel (denoised audio).
* model.load\_state\_dict(torch.load(model\_path, map\_location=device)): This loads the model's state dictionary (trained weights) from the file located at model\_path. The map\_location=device argument ensures that the model is loaded onto the specified device (CPU or GPU).
* model.to(device): This moves the model to the specified device (e.g., GPU for faster inference).
* model.eval(): This sets the model to evaluation mode, which disables certain layers like dropout and batch normalization that behave differently during training.
* return model: This returns the loaded model, ready for use in denoising predictions.

def denoise\_audio(model, noisy\_audio, device):

noisy\_audio = noisy\_audio.to(device)

with torch.no\_grad():

denoised\_audio = model(noisy\_audio.unsqueeze(0).unsqueeze(0))

denoised\_audio = denoised\_audio.squeeze(0).squeeze(0).cpu()

return denoised\_audio

**Define denoise\_audio Function**:

* def denoise\_audio(model, noisy\_audio, device): This function takes a pre-trained U-Net model, a noisy audio tensor, and a device, and returns the denoised audio tensor.
* noisy\_audio = noisy\_audio.to(device): This moves the noisy audio tensor to the specified device (e.g., GPU).
* with torch.no\_grad(): This context manager disables gradient calculation, which reduces memory usage and speeds up computation during inference.
* denoised\_audio = model(noisy\_audio.unsqueeze(0).unsqueeze(0)): This passes the noisy audio through the U-Net model to obtain the denoised audio. The unsqueeze(0) is used twice to add batch and channel dimensions to the tensor, as the model expects a 4D input (batch, channel, height, width). For audio, height and width correspond to the time and frequency dimensions (e.g., spectrogram).
* denoised\_audio = denoised\_audio.squeeze(0).squeeze(0).cpu(): This removes the added batch and channel dimensions from the denoised audio tensor and moves the tensor back to the CPU.
* return denoised\_audio: This returns the denoised audio tensor.

def save\_audio(file\_path, audio\_tensor, sample\_rate):

audio\_tensor = audio\_tensor.cpu().detach().numpy()

torchaudio.save(file\_path, audio\_tensor.unsqueeze(0), sample\_rate)

**Define save\_audio Function**:

* def save\_audio(file\_path, audio\_tensor, sample\_rate): This function saves a given audio tensor to a specified file path with the provided sample rate.
* audio\_tensor = audio\_tensor.cpu().detach().numpy(): This moves the audio tensor to the CPU, detaches it from the computation graph (if it's still attached), and converts it to a NumPy array for saving.
* torchaudio.save(file\_path, audio\_tensor.unsqueeze(0), sample\_rate): This uses torchaudio.save to save the audio tensor as a file at the specified file\_path. The unsqueeze(0) adds a batch dimension, as torchaudio.save expects the input tensor to be 2D (channels, samples).

def main():

parser = argparse.ArgumentParser(description="Denoise audio using a pre-trained U-Net model.")

parser.add\_argument("--model", type=str, required=True, help="Path to the pre-trained U-Net model.")

parser.add\_argument("--input", type=str, required=True, help="Path to the input noisy audio file.")

parser.add\_argument("--output", type=str, required=True, help="Path to save the denoised audio file.")

parser.add\_argument("--device", type=str, default="cpu", help="Device to run the model on (e.g., 'cpu' or 'cuda').")

args = parser.parse\_args()

**Define main Function and Parse Arguments**:

* def main(): This defines the main function, which is the entry point of the script.
* parser = argparse.ArgumentParser(...): This creates an argument parser to handle command-line arguments.
* parser.add\_argument("--model", type=str, required=True, help="Path to the pre-trained U-Net model."): This adds a --model argument to the parser, which specifies the path to the pre-trained U-Net model.
* parser.add\_argument("--input", type=str, required=True, help="Path to the input noisy audio file."): This adds a --input argument to specify the path to the input noisy audio file.
* parser.add\_argument("--output", type=str, required=True, help="Path to save the denoised audio file."): This adds a --output argument to specify where to save the denoised audio file.
* parser.add\_argument("--device", type=str, default="cpu", help="Device to run the model on (e.g., 'cpu' or 'cuda')."): This adds a --device argument to specify the device on which to run the model, defaulting to "cpu" if not provided.
* args = parser.parse\_args(): This parses the command-line arguments and stores them in the args variable.

device = torch.device(args.device)

model = load\_model(args.model, device)

**Load Model and Set Device**:

* device = torch.device(args.device): This sets the device for computation (CPU or GPU) based on the argument provided by the user.
* model = load\_model(args.model, device): This calls the load\_model function to load the pre-trained U-Net model onto the specified device.

waveform, sample\_rate = torchaudio.load(args.input)

waveform = waveform.mean(dim=0)

**Load and Preprocess Input Audio**:

* waveform, sample\_rate = torchaudio.load(args.input): This loads the input noisy audio file using torchaudio.load. It returns the audio waveform tensor and the sample rate.
* waveform = waveform.mean(dim=0): This averages the waveform across channels (if it's stereo) to convert it to mono.

denoised\_waveform = denoise\_audio(model, waveform, device)

**Denoise the Audio**:

* denoised\_waveform = denoise\_audio(model, waveform, device): This calls the denoise\_audio function to denoise the waveform using the pre-trained U-Net model.

save\_audio(args.output, denoised\_waveform, sample\_rate)

**Save the Denoised Audio**:

* save\_audio(args.output, denoised\_waveform, sample\_rate): This calls the save\_audio function to save the denoised waveform to the specified output file path.

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

main()

**Script Entry Point**:

* if \_\_name\_\_ == "\_\_main\_\_":: This checks if the script is being run directly (not imported as a module).
* main(): If the script is being run directly, this calls the main function to execute the denoising process.

**data\_display.py:**

import matplotlib.pyplot as plt

import torchaudio

import numpy as np

**Import Necessary Libraries**:

* import matplotlib.pyplot as plt: This imports the pyplot module from matplotlib, which is a popular library for creating static, animated, and interactive visualizations in Python. It is commonly used for plotting graphs.
* import torchaudio: This imports the torchaudio library, which provides audio processing utilities, including tools to load, transform, and save audio data.
* import numpy as np: This imports the numpy library, a fundamental package for scientific computing in Python, which supports large, multi-dimensional arrays and matrices, along with a collection of mathematical functions.

def plot\_waveform(waveform, sample\_rate, title="Waveform", xlim=None, ylim=None):

num\_channels, num\_frames = waveform.shape

time\_axis = np.arange(0, num\_frames) / sample\_rate

**Define plot\_waveform Function**:

* def plot\_waveform(waveform, sample\_rate, title="Waveform", xlim=None, ylim=None): This function plots the waveform of an audio signal. It takes the following parameters:

waveform: The audio signal as a 2D tensor (channels x frames).

sample\_rate: The sample rate of the audio signal.

title: The title of the plot (default is "Waveform").

xlim: The x-axis limits for the plot (optional).

ylim: The y-axis limits for the plot (optional).

* num\_channels, num\_frames = waveform.shape: This unpacks the shape of the waveform tensor into num\_channels (number of audio channels) and num\_frames (number of time frames).
* time\_axis = np.arange(0, num\_frames) / sample\_rate: This creates a time axis for the waveform. np.arange(0, num\_frames) generates a range of values from 0 to num\_frames - 1, and dividing by sample\_rate converts these values into seconds.

figure, axes = plt.subplots(num\_channels, 1, figsize=(15, 4\*num\_channels))

if num\_channels == 1:

axes = [axes]

**Create Subplots**:

* figure, axes = plt.subplots(num\_channels, 1, figsize=(15, 4\*num\_channels)): This creates a figure and subplots for each channel in the waveform. The figsize argument sets the size of the figure, with a width of 15 inches and a height proportional to the number of channels (4\*num\_channels).
* if num\_channels == 1: axes = [axes]: If there is only one channel (i.e., mono audio), axes is a single Axes object instead of a list. This line converts it into a list for consistent handling in the subsequent code.

for i, channel in enumerate(waveform):

axes[i].plot(time\_axis, channel, linewidth=1)

axes[i].grid(True)

axes[i].set\_title(f'{title} (Channel {i+1})')

axes[i].set\_xlabel('Time [s]')

axes[i].set\_ylabel('Amplitude')

if xlim:

axes[i].set\_xlim(xlim)

if ylim:

axes[i].set\_ylim(ylim)

**Plot Each Channel**:

* for i, channel in enumerate(waveform): This loops through each channel in the waveform.

i: The index of the current channel.

channel: The waveform data for the current channel.

* axes[i].plot(time\_axis, channel, linewidth=1): This plots the waveform of the current channel against the time axis. The linewidth=1 argument sets the line width of the plot.
* axes[i].grid(True): This enables the grid on the plot for better readability.
* axes[i].set\_title(f'{title} (Channel {i+1})'): This sets the title of the subplot, indicating the waveform title and channel number.
* axes[i].set\_xlabel('Time [s]'): This sets the label for the x-axis, indicating that the axis represents time in seconds.
* axes[i].set\_ylabel('Amplitude'): This sets the label for the y-axis, indicating that the axis represents the amplitude of the audio signal.
* if xlim: axes[i].set\_xlim(xlim): If xlim is provided, this sets the x-axis limits to the specified range.
* if ylim: axes[i].set\_ylim(ylim): If ylim is provided, this sets the y-axis limits to the specified range.

plt.show(block=False)

**Display the Plot**:

* plt.show(block=False): This displays the plot. The block=False argument allows the script to continue running after the plot is shown (non-blocking), which is useful in interactive environments like Jupyter notebooks.

def plot\_spectrogram(waveform, sample\_rate, title="Spectrogram", xlim=None, ylim=None, n\_fft=400, hop\_length=None, win\_length=None):

spectrogram = torchaudio.transforms.Spectrogram(n\_fft=n\_fft, hop\_length=hop\_length, win\_length=win\_length)(waveform)

spectrogram = torchaudio.transforms.AmplitudeToDB()(spectrogram)

**Define plot\_spectrogram Function**:

* def plot\_spectrogram(waveform, sample\_rate, title="Spectrogram", xlim=None, ylim=None, n\_fft=400, hop\_length=None, win\_length=None): This function plots the spectrogram of an audio signal. It takes the following parameters:

waveform: The audio signal as a 2D tensor (channels x frames).

sample\_rate: The sample rate of the audio signal.

title: The title of the plot (default is "Spectrogram").

xlim: The x-axis limits for the plot (optional).

ylim: The y-axis limits for the plot (optional).

n\_fft: The number of FFT (Fast Fourier Transform) points (default is 400).

hop\_length: The number of audio frames between STFT (Short-Time Fourier Transform) columns (optional).

win\_length: The size of the window used in STFT (optional).

* spectrogram = torchaudio.transforms.Spectrogram(n\_fft=n\_fft, hop\_length=hop\_length, win\_length=win\_length)(waveform): This applies the STFT to the waveform to compute the spectrogram. n\_fft, hop\_length, and win\_length are passed to configure the STFT operation.
* spectrogram = torchaudio.transforms.AmplitudeToDB()(spectrogram): This converts the amplitude of the spectrogram to a decibel scale for better visualization.

figure, axes = plt.subplots(1, 1, figsize=(15, 4))

axes.imshow(spectrogram.log2()[0,:,:].numpy(), cmap='viridis', aspect='auto', origin='lower',

extent=[0, spectrogram.size(-1) \* hop\_length / sample\_rate, 0, sample\_rate / 2])

**Create Spectrogram Plot**:

* figure, axes = plt.subplots(1, 1, figsize=(15, 4)): This creates a figure and a single subplot to display the spectrogram. The figsize argument sets the size of the figure.
* axes.imshow(spectrogram.log2()[0,:,:].numpy(), cmap='viridis', aspect='auto', origin='lower', extent=[0, spectrogram.size(-1) \* hop\_length / sample\_rate, 0, sample\_rate / 2]): This visualizes the spectrogram using imshow, which displays data as an image. The arguments are as follows:

spectrogram.log2()[0,:,:].numpy(): Converts the logarithm (base 2) of the spectrogram for the first channel to a NumPy array.

cmap='viridis': Specifies the colormap to use ('viridis' is a popular perceptually uniform colormap).

aspect='auto': Adjusts the aspect ratio of the plot automatically.

origin='lower': Sets the origin of the plot to the lower-left corner.

extent=[0, spectrogram.size(-1) \* hop\_length / sample\_rate, 0, sample\_rate / 2]: Defines the bounds of the axes, setting the x-axis from 0 to the duration of the audio and the y-axis from 0 to the Nyquist frequency (half the sample rate).

axes.set\_title(title)

axes.set\_xlabel('Time [s]')

axes.set\_ylabel('Frequency [Hz]')

if xlim:

axes.set\_xlim(xlim)

if ylim:

axes.set\_ylim(ylim)

**Configure Spectrogram Plot**:

* axes.set\_title(title): Sets the title of the spectrogram plot.
* axes.set\_xlabel('Time [s]'): Sets the label for the x-axis, indicating time in seconds.
* axes.set\_ylabel('Frequency [Hz]'): Sets the label for the y-axis, indicating frequency in Hertz.
* if xlim: axes.set\_xlim(xlim): If xlim is provided, sets the x-axis limits to the specified range.
* if ylim: axes.set\_ylim(ylim): If ylim is provided, sets the y-axis limits to the specified range.

plt.show(block=False)

**Display the Spectrogram Plot**:

* plt.show(block=False): Displays the spectrogram plot. The block=False argument allows the script to continue running after the plot is shown.