

ADAPTIVE NOISE CANCELLATION SYSTEM USING AI

Monsoon 2024

*Submitted
by*

Aadityan Gupta (Roll No. 2110110002)
and
Bhavya Puri (Roll No. 2110110166)

*Under Supervision
of*

Mr. Aakash Kumar Sinha
Department of Electrical Engineering



**SCHOOL OF
ENGINEERING**

Department of Electrical Engineering,
School of Engineering,
Shiv Nadar Institution of Eminence Deemed to be University,
Delhi-NCR

ABSTRACT

The project titled "Adaptive Noise Cancellation System Using AI" aims to design and implement an advanced noise cancellation system that dynamically reduces unwanted ambient sounds using artificial intelligence. The system will employ a microphone array to capture environmental noise, which will then be processed in real time using sophisticated signal processing algorithms. Leveraging adaptive machine learning techniques, the system will effectively filter out unwanted noise while preserving the integrity of the desired audio. This technology is intended to enhance audio quality in various settings, from personal audio devices to large-scale industrial applications. The adaptive nature of the AI-driven approach ensures that the system continuously learns and improves its noise cancellation capabilities, making it highly efficient across different environments and noise profiles.

Contents

1	Introduction	7
1.1	Overview of Adaptive Noise Cancellation Systems	7
1.2	Adaptive Noise Cancellation Techniques:	8
1.2.1	Active Noise Cancellation:	8
1.2.2	Passive Noise Cancellation:	8
1.2.3	AI-Based Noise Cancellation:	8
1.2.4	Hybrid Noise Cancellation:	8
1.3	Applications of Adaptive Noise Cancellation:	9
1.3.1	Challenges in Adaptive Noise Cancellation:	9
1.3.2	Future Directions in Adaptive Noise Cancellation:	9
2	Literature Survey	10
2.1	Introduction to Noise Cancellation:	10
2.1.1	Traditional Adaptive Noise Cancellation (ANC) Techniques:	10
2.1.2	AI-based Techniques in Noise Cancellation	11
2.1.3	Advantages Over Traditional ANC	11
2.1.4	Current Trends and Future Directions	12
3	Work Done till Midterm	13
3.1	Code	13
3.2	Outputs Obtained	20
3.2.1	Graphs	21
3.2.2	Spectrogram	23
4	Work done Post Midterm	26
4.1	Code	26
4.1.1	Midterm Project Overview	26
4.1.2	End-Semester Objective	26
4.1.3	Datasets Used	26
4.2	Code Snippets and Explanation	27
4.2.1	Training Model	28
4.2.2	AudioDataset Class	35
4.2.3	UNetANC Model Architecture	37
4.2.4	Audio Recording and Denoising Process	40

5	Future Work	45
5.1	Future Work in Adaptive Noise Cancellation System Using AI	45
5.1.1	Enhanced Model Architecture	45
5.1.2	Real-Time Implementation	45
5.1.3	Dataset Expansion	46
5.1.4	Custom Hardware Integration	46
5.1.5	Advanced Use Cases	46
5.1.6	Improved Evaluation Metrics	46
5.2	Conclusion	47
	References	48

List of Figures

1.1	ANC v/s ANC using AI	9
3.1	Code Snippet 1	13
3.2	Code Snippet 2	14
3.3	Code Snippet 3	15
3.4	Code Snippet 4	16
3.5	Code Snippet 5	18
3.6	Code Snippet 6	19
3.7	Audio Waveform Graphs	21
3.8	Audio Waveform Spectrogram	23
4.1	Audio Waveform Graphs- EndTerm	33

List of Tables

2.1	Comparison Between ANC Technologies	11
-----	---	----

Chapter 1

Introduction

Artificial intelligence powered adaptive noise cancellation (ANC) systems are transforming the audio technology landscape by combating noise disturbances. These innovative systems have the capability to eliminate background noises while enabling users to concentrate better on preferred audio inputs, like speech or music. This differs from ANC techniques which adhere to predetermined algorithms; AI enhanced ANC adjusts dynamically to varying environments, in real time. The flexibility of these systems boosts their ability to reduce noise effectively in a range of applications, like consumer electronics and professional communication environments which enhances clarity and user satisfaction in the end.

1.1 Overview of Adaptive Noise Cancellation Systems

Adaptive noise cancellation systems use AI to actively dampen unwanted noises while maintaining sounds users actually want to hear. They have three main parts: several microphones that pick up various ambient sounds and the sound the wearers are interested in hearing, the signal processing unit which processes the sound waves and implements the noise cancelling algorithm, and the output which sends the processed audio to the wearer's in-ears, on-ears and over-ears. The role AI plays here is in the processing unit, where it can learn and adjust in real-time, something that is impossible for traditional systems. This learning ability dramatically increases the systems' ability to effectively block out all types of noises and work in many diverse environments, and why adaptive ANC systems are much better than their traditional, fixed kind.

Key Components:

1. **Microphones:** Capture ambient sounds and user audio.
2. **Signal Processing Unit:** Analyzes incoming audio signals and applies noise reduction algorithms.
3. **Output Mechanism:** Delivers the processed audio to headphones or speakers.

1.2 Adaptive Noise Cancellation Techniques:

Adaptive noise cancellation (ANC) employs various techniques to effectively reduce unwanted background sounds. Here are the main types of adaptive noise cancellation:

1.2.1 Active Noise Cancellation:

Active noise cancellation (ANC) utilizes microphones to detect external sounds and generate anti-noise signals that are 180 degrees out of phase with the incoming noise. This phase inversion effectively cancels out unwanted sounds when they combine with the original audio signal. ANC is particularly effective for low-frequency noises like engine sounds or air conditioning units. However, its performance can vary based on factors such as microphone placement and environmental conditions. ANC technology is commonly found in headphones and earphones designed for use in noisy environments.

1.2.2 Passive Noise Cancellation:

Passive noise cancellation relies on physical barriers to block external sounds rather than generating counteracting sound waves. This method often involves using materials that absorb sound or creating a tight seal around the ears to isolate them from ambient noise. While passive cancellation can be effective for certain frequencies, it typically does not provide the same level of noise reduction as active methods. Passive techniques are commonly used in traditional headphones and earplugs, making them suitable for various applications where complete sound isolation is not critical.

1.2.3 AI-Based Noise Cancellation:

AI-based noise cancellation leverages artificial intelligence algorithms to enhance traditional ANC methods by continuously analyzing and adapting to environmental sounds. These algorithms can distinguish between different types of noises and dynamically adjust their filtering techniques accordingly. AI-based systems are particularly effective in complex acoustic environments where background noises vary significantly over time. By learning from ongoing audio input, these systems can optimize their performance for clearer communication during calls or recordings, making them invaluable in modern communication devices and applications.

1.2.4 Hybrid Noise Cancellation:

Hybrid noise cancellation combines both active and passive techniques to achieve superior sound isolation and clarity. By integrating microphones for active cancellation with physical barriers for passive isolation, hybrid systems can effectively reduce a wider range of frequencies and improve overall audio quality. This approach allows users to benefit from the strengths of both methods, making hybrid ANC suitable for various environments, including public transportation and busy offices. The versatility of hybrid systems ensures that users experience minimal disruption from ambient noise while enjoying their audio content.

1.3 Applications of Adaptive Noise Cancellation:

Adaptive noise cancellation technology finds applications across diverse fields, including consumer electronics, telecommunications, automotive industries, and healthcare. In consumer electronics, ANC is widely used in headphones and earbuds to provide users with an immersive listening experience by minimizing background distractions. In telecommunications, adaptive algorithms enhance voice clarity during calls by filtering out background noises from both ends of the conversation. In automotive settings, ANC helps reduce cabin noise for a quieter driving experience. Additionally, healthcare applications include hearing aids that utilize adaptive noise cancellation to improve speech recognition in noisy environments.

1.3.1 Challenges in Adaptive Noise Cancellation:

Despite its advantages, adaptive noise cancellation faces several challenges that can impact its effectiveness. One significant challenge is accurately capturing ambient sounds while distinguishing them from the desired audio signal. Variability in environmental conditions can also affect performance; for instance, sudden changes in background noise levels may require rapid adjustments by the adaptive algorithms. Furthermore, the complexity of implementing effective algorithms can lead to increased power consumption in battery-operated devices like headphones. Addressing these challenges is essential for improving the reliability and efficiency of adaptive noise cancellation systems across various applications.

1.3.2 Future Directions in Adaptive Noise Cancellation:

The future of adaptive noise cancellation technology looks promising as research continues to advance its capabilities. Innovations in machine learning and artificial intelligence are expected to enhance algorithm efficiency and adaptability further, allowing systems to better understand complex acoustic environments. Additionally, developments in hardware design will likely lead to more effective microphone placements and improved sound isolation materials. As user demand for high-quality audio experiences grows across industries, continued investment in adaptive noise cancellation technologies will play a crucial role in shaping the future landscape of audio processing solutions.

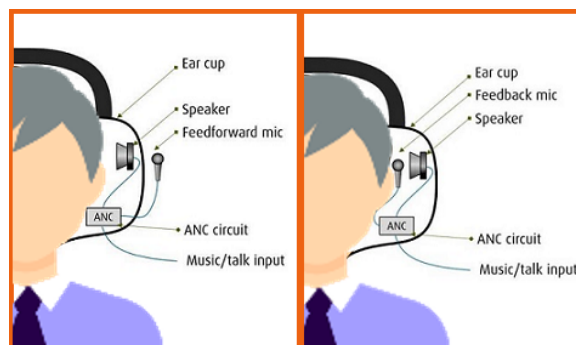


Figure 1.1: ANC v/s ANC using AI

Chapter 2

Literature Survey

The literature survey provides an overview of the developments and research in the field of noise cancellation, with a specific focus on the transition from traditional approaches to AI-based systems. Noise cancellation has been a critical topic in signal processing for decades, evolving from simple analogue techniques to more sophisticated methods using digital signal processing (DSP). However, with the rise of artificial intelligence (AI), modern noise cancellation systems are becoming more intelligent and adaptive, enabling them to operate effectively in complex and dynamic noise environments.

This section will examine the evolution of noise cancellation techniques, starting from passive and active noise control methods, followed by adaptive filter-based systems, and finally focusing on the latest advancements involving AI-based noise cancellation techniques. The survey will provide an in-depth analysis of the strengths and limitations of various approaches, drawing comparisons between traditional and AI-based methods and exploring the potential of AI to enhance adaptive noise cancellation in real-time, multi-source, and non-stationary environments.

2.1 Introduction to Noise Cancellation:

This section will discuss the fundamental concepts of noise cancellation and its significance in signal processing. The development of traditional methods, such as passive and active noise control (ANC), will be introduced, and their evolution into more advanced digital techniques will be highlighted. The limitations of classical approaches, particularly in dealing with complex and dynamic noise environments, motivate the need for intelligent systems.

2.1.1 Traditional Adaptive Noise Cancellation (ANC) Techniques:

This section will cover the most widely used traditional techniques for noise cancellation, focusing on adaptive filters like Least Mean Squares (LMS) and Recursive Least Squares (RLS). These algorithms dynamically adjust filter coefficients to minimize the error between the desired signal and the noise. While these techniques have been successful in many applications, they require accurate modelling and face challenges in non-linear and time-varying noise environments.

Emergence of AI in Noise Cancellation:

Artificial intelligence has introduced new possibilities for noise cancellation by leveraging data-driven techniques. This section will explore the emergence of AI-based methods and how they differ from traditional approaches. AI techniques like machine learning, deep learning, and neural networks can model complex noise environments and dynamically adapt in real time, offering significant improvements over traditional methods.

2.1.2 AI-based Techniques in Noise Cancellation

An overview of various AI-based noise cancellation techniques:

- 1. Deep Learning Approaches:** Use of convolutional and recurrent neural networks (CNNs and RNNs) for speech enhancement and audio processing.
- 2. Auto-encoder:** Application of Denoising Autoencoders (DAEs) for separating noise from the desired signal.
- 3. Reinforcement Learning (RL):** Techniques that optimize noise cancellation dynamically in real-time.
- 4. Generative Adversarial Networks (GANs):** Used to improve the performance of noise cancellation systems by generating clean signals.

2.1.3 Advantages Over Traditional ANC

AI-based noise cancellation offers several benefits:

- 1. Real-Time Adaptability:** Unlike fixed ANC systems, AI algorithms can continuously learn and adjust to new types of noise, improving performance in diverse settings.
- 2. Enhanced Performance:** AI systems can suppress a broader range of frequencies, including those associated with human voices, which traditional ANC might struggle with.

Table 2.1: Comparison Between ANC Technologies

Feature	Traditional ANC	AI-Based Noise Cancellation
Adaptability	Fixed design for specific noises	Continuously learns from data
Processing Speed	May introduce delays	Real-time processing capabilities
Noise Range	Effective for low-frequency noises	Handles wide frequency ranges
Complexity	Simpler algorithms	More complex due to machine learning
Applications	Headphones, active control	Voice assistants, conferencing tools

2.1.4 Current Trends and Future Directions

1. Advancements in Technology

Recent developments in AI have led to more sophisticated noise suppression technologies:

1. Companies like Nvidia are leveraging deep learning for two-end noise suppression systems that work effectively during calls.
2. Startups like Krisp provide versatile solutions compatible with multiple platforms, enhancing user experience across different devices.

2. Future Research Directions

Future research may focus on:

1. Developing hybrid models that combine traditional ANC techniques with advanced machine learning methods.
2. Exploring applications in emerging fields such as augmented reality (AR) and virtual reality (VR), where clear audio communication is critical.

Chapter 3

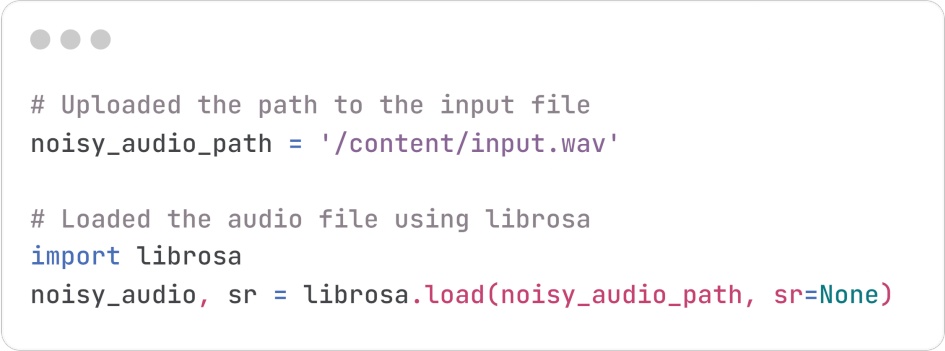
Work Done till Midterm

To understand the project and its working properly, we researched and explored the module 'noisereduce' and we implemented that with some other libraries to get the desired output which is a noise reduced audio.

We used a website MyEditOnline to generate an AI audio where a man is speaking something in the middle of traffic noise. We used the libraries and the input audio to get the noise-reduced audio.

Here is the code with the explanation of everything implemented. We worked on Google Colab and created a ipynb file to run the code.

3.1 Code



```
# Uploaded the path to the input file
noisy_audio_path = '/content/input.wav'

# Loaded the audio file using librosa
import librosa
noisy_audio, sr = librosa.load(noisy_audio_path, sr=None)
```

Figure 3.1: Code Snippet 1

In the first line, we define the path for the file where we used a variable noisyaudiopath which is assigned to the location of the input.wav.

In the next line we import the librosa library, which is commonly used for audio processing tasks in Python. librosa provides functionality to load, manipulate, and analyze audio


data.

`librosa.load()` is used to load the audio file into the program. It reads the file specified by `noisyaudiopath`.

`sr=None` means that the audio will be loaded with its original sampling rate, rather than resampling it to a different rate. If you wanted to resample the audio to a specific rate, you could set `sr` to a desired value (e.g., `sr=16000`).

The function returns two values:

1. `noisyaudio`: A NumPy array containing the audio data (amplitude values).
2. `sr`: The sampling rate of the audio file, i.e., how many samples of audio data are captured per second.

A code snippet box with a light gray border and rounded corners. At the top left, there are three small gray circles. The box contains three lines of Python code in a blue monospace font.

```
import matplotlib.pyplot as plt
import librosa.display
import numpy as np
```

Figure 3.2: Code Snippet 2

```
import matplotlib.pyplot as plt:
```

This line imports the `pyplot` module from the `matplotlib` library, a widely-used Python library for creating static, animated, and interactive visualizations. By importing `pyplot` as `plt`, you can utilize its functions to generate various types of plots and charts.

```
import librosa.display:
```

Here, the `display` module from the `librosa` library is imported. `librosa.display` provides functions to visualize audio data, such as waveforms and spectrograms, which are useful for analyzing the frequency and time characteristics of audio signals.

```
import numpy as np:
```

This line imports the `numpy` library, a fundamental package for scientific computing with Python. `numpy` offers powerful N-dimensional array objects and an array of functions

to perform operations on these arrays, enabling efficient numerical computations.

```
# Plotting the waveform of the noisy audio
plt.figure(figsize=(10, 4))
librosa.display.waveshow(noisy_audio, sr=sr)
plt.title("Noisy Audio Waveform")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()

# Short-Time Fourier Transform (STFT) to visualize the spectrogram
D_noisy = librosa.stft(noisy_audio)
plt.figure(figsize=(10, 6))
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_noisy), ref=np.max),
sr=sr, y_axis='log', x_axis='time')
plt.title('Noisy Audio Spectrogram')
plt.colorbar(format='%+2.0f dB')
plt.show()
```

Figure 3.3: Code Snippet 3

1. Plot the Waveform of the Noisy Audio:

`plt.figure(figsize=(10, 4))`: Sets the figure size for the plot.

`librosa.display.waveshow()`: Plots the waveform of the audio signal (`noisyaudio`), with the x-axis showing time and the y-axis showing amplitude.

`plt.title()`, `plt.xlabel()`, `plt.ylabel()`: Adds the title, x-axis label (Time in seconds), and y-axis label (Amplitude).

`plt.show()`: Displays the plot. 2. Short-Time Fourier Transform (STFT) and Spectrogram:

`librosa.stft(noisyaudio)`: Performs a Short-Time Fourier Transform (STFT) on the audio, converting it to the frequency domain.

`librosa.display.specshow()`: Displays the spectrogram (frequency vs. time) with decibel scaling.

`plt.colorbar()`: Adds a color bar indicating decibel levels.

`plt.show()`: Displays the spectrogram.

```

● ● ●

from transformers import AutoModel, Wav2Vec2Processor
import librosa
from transformers import Wav2Vec2ForCTC, Wav2Vec2Processor
import torch

# Load the noisy audio and resample it to 16 kHz
noisy_audio_path = '/content/input.wav'
noisy_audio, sr = librosa.load(noisy_audio_path, sr=16000) # Resampling to 16 kHz

# Load pre-trained Wav2Vec2 model and processor
model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-large-960h")
processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-large-960h")

# Convert noisy audio to the input format expected by the model
input_values = processor(noisy_audio, return_tensors="pt",
                          sampling_rate=16000).input_values

# Pass the input through the model to get predicted speech
with torch.no_grad():
    logits = model(input_values).logits

# Get the predicted transcription
predicted_ids = torch.argmax(logits, dim=-1)
transcription = processor.batch_decode(predicted_ids)[0]

print("Transcription: ", transcription)

```

Figure 3.4: Code Snippet 4

1. Importing Required Libraries:

transformers: Provides access to pre-trained models such as Wav2Vec2, which is used for automatic speech recognition (ASR).

librosa: Used to load and resample the audio.

torch: PyTorch library, used for handling tensor operations and running the model.

2. Loading and Resampling the Audio:

librosa.load(): Loads the audio file from the specified path (/content/input.wav) and resamples it to 16 kHz (the required input sample rate for Wav2Vec2).

noisyaudio: Contains the audio data as a NumPy array.

sr: The sampling rate, which is set to 16,000 Hz.

3. Loading the Pre-trained Wav2Vec2 Model and Processor:

Wav2Vec2ForCTC: Loads a pre-trained Wav2Vec2 model from Facebook for Connection-

ist Temporal Classification (CTC) based speech-to-text tasks.

Wav2Vec2Processor: Loads the corresponding processor for pre-processing the audio input (tokenizing and encoding) and post-processing the output (decoding).

4. Pre-processing the Audio for Model Input:

processor(): Converts the audio (noisyaudio) to the input format expected by the model.

return_tensors="pt": Ensures the output is a PyTorch tensor, ready for processing.

input_values: The processed audio data, formatted as a tensor for input to the model.

5. Passing the Audio Through the Model:

torch.no_grad(): Disables gradient computation since we are only performing inference (not training).

model(input_values): Feeds the processed audio into the Wav2Vec2 model to get the logits (raw predictions from the model).

6. Decoding the Predictions:

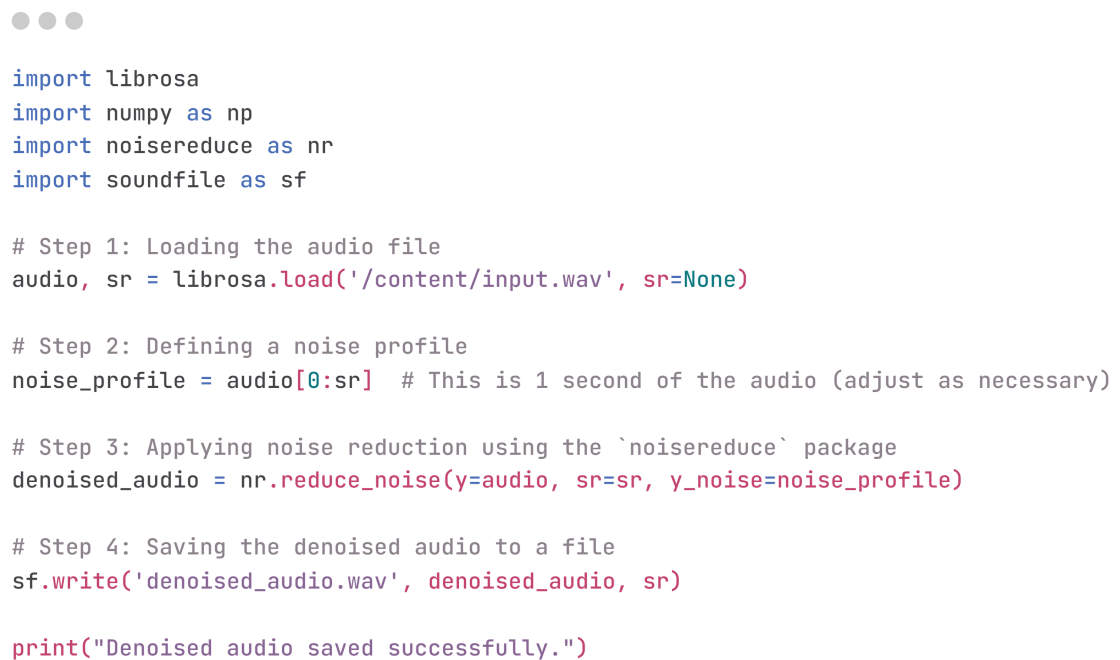
torch.argmax(): Extracts the most likely prediction for each time frame from the model's logits by taking the index with the highest probability.

processor.batch_decode(): Decodes the predicted token IDs back into text (transcription).

transcription: The final transcribed text from the noisy audio.

7. Displaying the Transcription:

Outputs the transcribed text, i.e., the speech converted to text from the audio input.



```

import librosa
import numpy as np
import noisereduce as nr
import soundfile as sf

# Step 1: Loading the audio file
audio, sr = librosa.load('/content/input.wav', sr=None)

# Step 2: Defining a noise profile
noise_profile = audio[0:sr] # This is 1 second of the audio (adjust as necessary)

# Step 3: Applying noise reduction using the `noisereduce` package
denoised_audio = nr.reduce_noise(y=audio, sr=sr, y_noise=noise_profile)

# Step 4: Saving the denoised audio to a file
sf.write('denoised_audio.wav', denoised_audio, sr)

print("Denoised audio saved successfully.")

```

Figure 3.5: Code Snippet 5

1. Defining a noise profile:

1. A noise profile is a segment of the audio that contains only noise. In this case, it's assumed that noise is prominent at the beginning of the file (the first 1 second).
2. `audio[0:sr]` selects the first second of audio data, where `sr` represents the number of samples in one second (since `sr` is the sample rate).

2. Noise reduction:

1. The `noisereduce.reduce_noise()` function reduces the noise from the audio signal.
2. `y=audio`: The original audio signal.
3. `sr=sr`: The sample rate of the audio.
4. `y_noise=noiseprofile`: The noise profile defined earlier, which helps the algorithm identify and remove noise from the full audio signal.

3. Saving the denoised audio:

1. This line saves the processed (denoised) audio to a new file called `denoisedaudio.wav` using the `soundfile.write()` function.
2. `denoisedaudio`: The noise-reduced audio.
3. `sr`: The sample rate, which is retained from the original audio file.

```

● ● ●

# Plotting the denoised waveform
plt.figure(figsize=(10, 4))
librosa.display.waveshow(denoised_audio, sr=sr)
plt.title("Denoised Audio Waveform")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()

# Visualizing the spectrogram of the denoised audio
D_denoised = librosa.stft(denoised_audio)
plt.figure(figsize=(10, 6))
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_denoised), ref=np.max),
sr=sr, y_axis='log', x_axis='time')
plt.title('Denoised Audio Spectrogram')
plt.colorbar(format='%+2.0f dB')
plt.show()

```

Figure 3.6: Code Snippet 6

1. Plot the denoised waveform:

1. `plt.figure(figsize=(10, 4))`: Creates a figure with a width of 10 units and a height of 4 units.
2. `librosa.display.waveshow(denoisedaudio, sr=sr)`: This function plots the waveform of the denoised audio. It visualizes the changes in amplitude over time. The `sr` (sample rate) is passed to ensure the time axis is correctly scaled.
3. `plt.title("Denoised Audio Waveform")`: Sets the title of the plot as "Denoised Audio Waveform".
4. `plt.xlabel("Time (s)")`: Labels the x-axis as "Time (s)", representing time in seconds.
5. `plt.ylabel("Amplitude")`: Labels the y-axis as "Amplitude", which represents the loudness or intensity of the audio signal.
6. `plt.show()`: Displays the waveform plot.

2. Visualize the spectrogram of the denoised audio:

1. `Ddenoised = librosa.stft(denoisedaudio)`: This performs a Short-Time Fourier Transform (STFT) on the denoised audio signal. STFT transforms the audio from the time domain into the frequency domain, producing a complex matrix `Ddenoised` where each element represents the magnitude and phase of a specific frequency component at a particular time.
2. `plt.figure(figsize=(10, 6))`: Creates a figure with a width of 10 units and a height of 6 units.

3. `librosa.display.specshow(librosa.amplitudetodb(np.abs(Ddenoised)), ref=np.max, sr=sr, yaxis='log', xaxis='time')`:

1. `np.abs(Ddenoised)`: Takes the absolute value of the STFT output, which represents the magnitude of the frequency components.

2. `librosa.amplitudetodb()`: Converts the magnitude to decibels (dB) for better visualization.

3. `librosa.display.specshow()`: Displays the spectrogram, showing how different frequencies change over time.

4. `yaxis='log'`: Uses a logarithmic scale for the frequency axis, which is common in audio analysis because it reflects how humans perceive sound (low frequencies are perceived more broadly than high frequencies).

5. `xaxis='time'`: Represents time along the x-axis.

6. `plt.title('Denoised Audio Spectrogram')`: Sets the title of the plot as "Denoised Audio Spectrogram".

7. `plt.colorbar(format='+2.0f dB')`: Adds a color bar to the side of the spectrogram to show the dB scale, with values in decibels (dB) indicating the intensity of frequencies.

8. `plt.show()`: Displays the spectrogram plot.

3.2 Outputs Obtained

We obtained 2 graphs and 2 spectrogram on Noised Input and Denoised Output each. We also were successful in achieving the Denoised Output Audio too.

3.2.1 Graphs

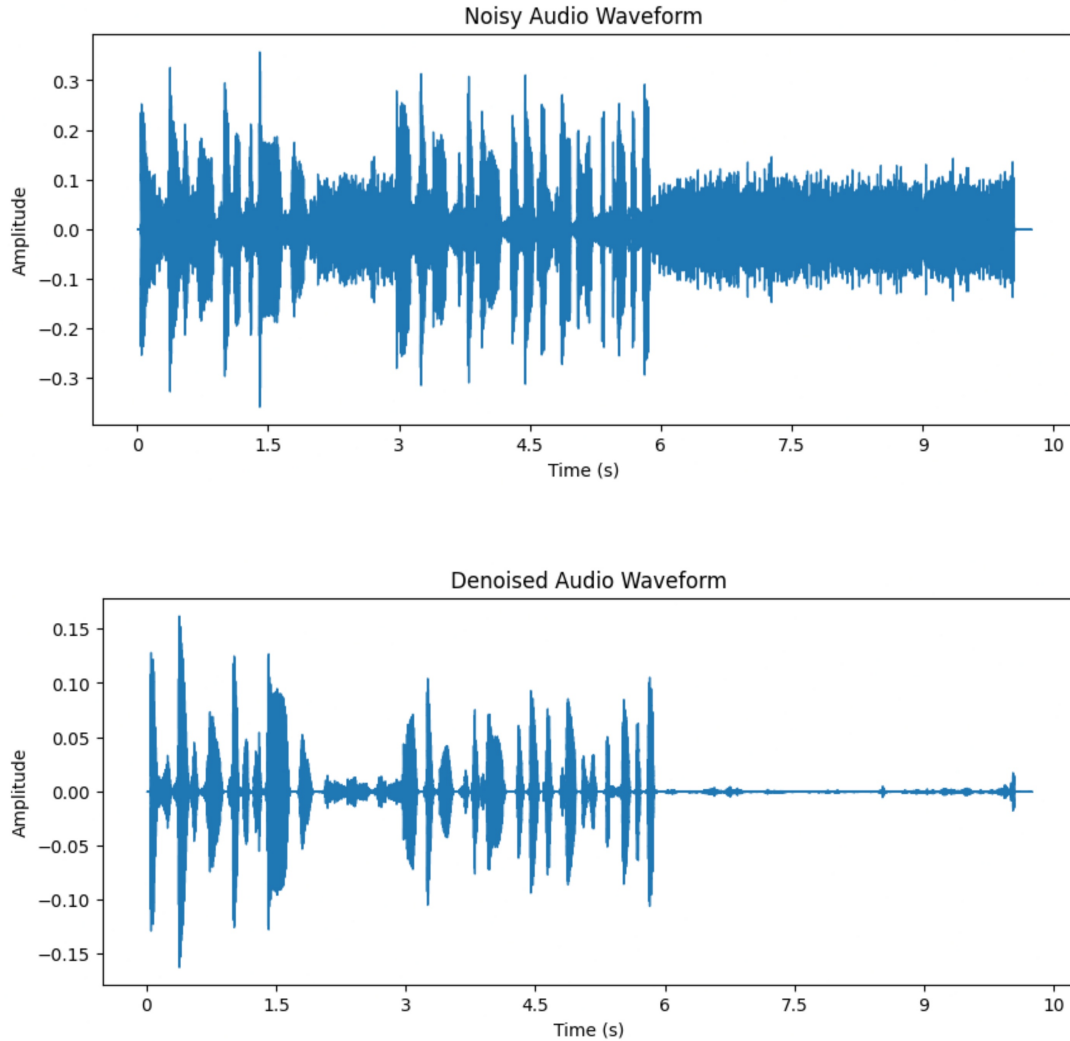


Figure 3.7: Audio Waveform Graphs

The above graphs show the waveforms of the input noisy audio signal and its corresponding denoised version over a period of 10 seconds, with amplitude on the y-axis and time on the x-axis.

Noisy Audio Waveform

- 1. Characteristics:** The waveform shows a lot of irregular, high-amplitude spikes throughout, especially in the first 6 seconds. These spikes represent significant noise, distorting the underlying audio signal.
- 2. Amplitude:** The signal ranges between -0.3 to +0.3 amplitude, with noticeable high-amplitude regions between 0 to 6 seconds. After the 6-second mark, the amplitude

becomes more regular but still exhibits some noise.

3. Interpretation: The signal is affected by noise, which could be due to environmental factors or unwanted interference. The noise is especially prevalent in the first half.

Denoised Audio Waveform

1. Characteristics: The denoised waveform has much smaller amplitude peaks, particularly after 3 seconds. The noise is significantly reduced, with the signal becoming very stable and flat after 6 seconds.

2. Amplitude: The amplitude has decreased, ranging between -0.15 and +0.15. There is much less variation in the waveform, indicating a successful removal of most noise.

3. Interpretation: The denoising process has effectively reduced the unwanted noise, especially in the latter half of the waveform. The signal appears cleaner, particularly after 6 seconds, where it becomes almost flat.

Analysis

There is a clear improvement in the clarity of the signal after the denoising process. The top graph shows a noisy signal with significant amplitude variations, while the bottom graph demonstrates that the noise has been reduced substantially, especially after the 6-second mark.

Despite the noise reduction, the key parts of the original signal (main audio content) in the first 3 seconds remain, though at a lower amplitude. This indicates that the denoising algorithm was able to retain the important parts of the audio while filtering out the noise.

3.2.2 Spectrogram

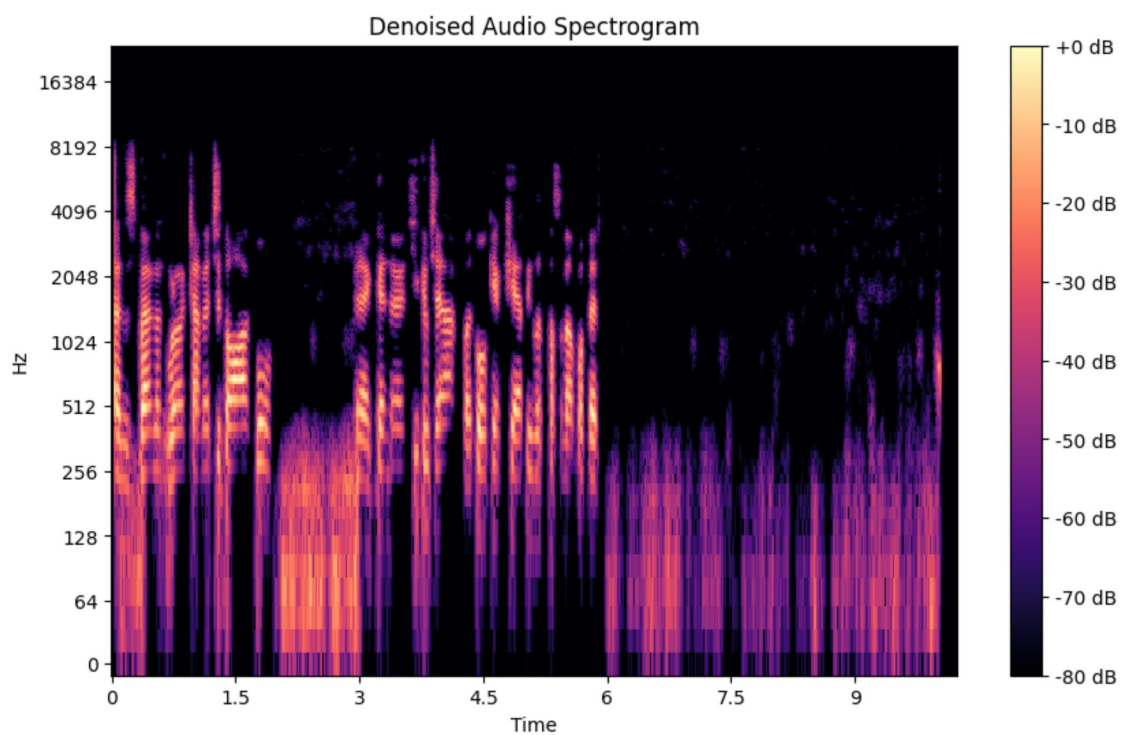
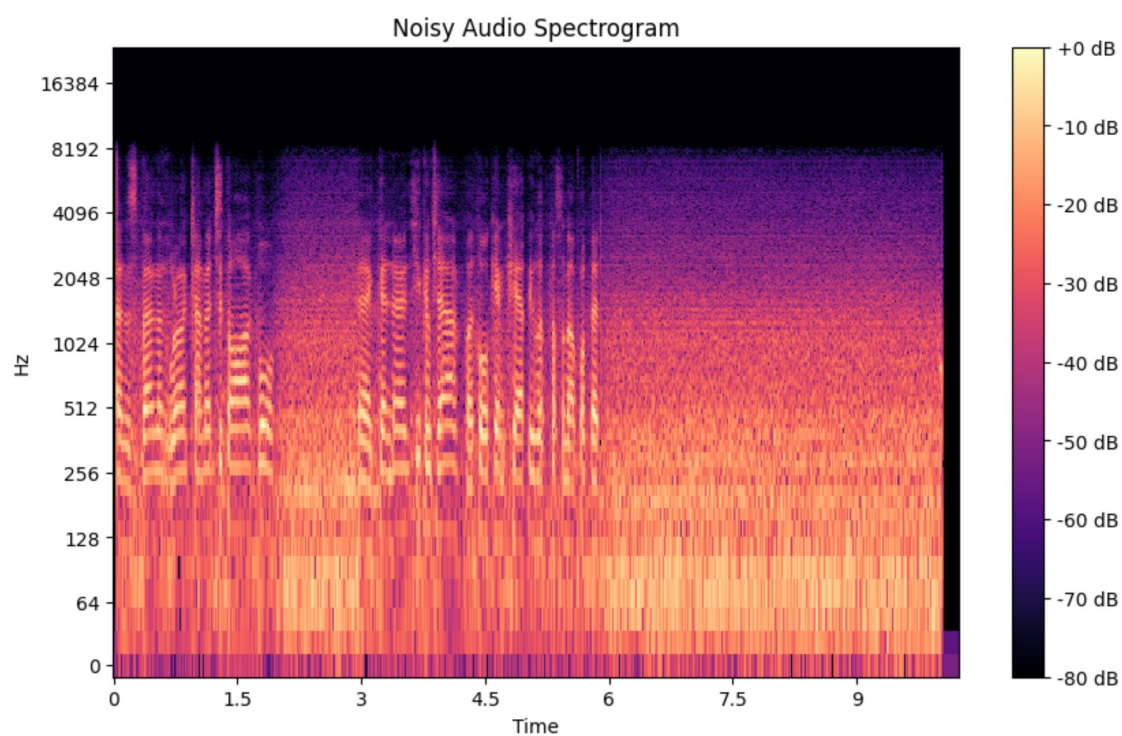


Figure 3.8: Audio Waveform Spectrogram

The spectrograms depict the frequency content of an audio signal over time. The vertical axis (y-axis) represents frequency in Hertz (Hz), the horizontal axis (x-axis) represents time in seconds, and the color intensity represents the magnitude of the signal in decibels (dB), with brighter colors indicating higher magnitude.

Noisy Audio Spectrogram

1. Frequency Range: The noisy signal exhibits strong frequency content throughout the entire frequency spectrum, from 0 Hz up to 16,384 Hz. There is significant energy (shown in brighter colors) spread across low, mid, and high frequencies.

2. Noise Characteristics:

1. Low-frequency regions (up to 256 Hz) and mid-range frequencies (256 Hz to 2,048 Hz) show consistent energy levels throughout the entire duration, indicating that noise is present in both the lower and mid-frequency bands.

2. High-frequency content is more prominent up to the 4,000 Hz range, which is more associated with noise.

3. After around 6 seconds, the intensity of the frequencies starts to fade, though some residual noise persists.

3. Noise Prevalence: There is a noticeable amount of noise scattered across the entire frequency range, indicated by the large presence of yellow and orange colors, particularly in the early part of the audio (up to 6 seconds).

Denoised Audio Spectrogram

1. Frequency Range: The denoised version shows a much cleaner spectrogram. Energy in the lower frequencies (below 256 Hz) and mid-frequencies (up to 2,048 Hz) has been retained, but the high-frequency noise has been significantly reduced.

2. Noise Characteristics:

1. The frequency content in the denoised audio is more focused on specific frequency bands, especially below 4,000 Hz. The areas of high energy in the noisy version (shown in yellow) are replaced by much darker regions (blues and purples), indicating that the high-frequency noise has been effectively removed.

2. There is much less color intensity (low dB values) in the high-frequency range, particularly after 6 seconds, where the signal becomes almost entirely dark, indicating very little noise.

3. Signal Retention: The essential parts of the audio (in lower and mid frequencies) have been preserved in the denoised spectrogram, especially up to 2,048 Hz, where some of the original signal remains. Beyond 6 seconds, the spectrogram is much cleaner with minimal noise.

Analysis

The denoised spectrogram shows a clear reduction in high-frequency noise compared to the noisy spectrogram. Frequencies above 4,000 Hz, which were dominated by noise in the original signal, are almost entirely removed in the denoised version.

The denoising process has maintained the core content of the signal, particularly in the lower and mid-frequency bands, where meaningful audio information typically resides

(e.g., voice or music). The signal becomes much clearer after 6 seconds, as most of the noise has been successfully suppressed.

Chapter 4

Work done Post Midterm

4.1 Code

4.1.1 Midterm Project Overview

In the midterm project, we explored **Active Noise Cancellation (ANC)** using artificial intelligence by leveraging pre-existing models. These models were already trained on large datasets to filter out noise and enhance audio quality. Our implementation primarily focused on understanding the behavior and output of these models, evaluating their effectiveness, and examining how AI could be applied to audio denoising.

4.1.2 End-Semester Objective

For the end-semester project, we aim to go beyond using pre-trained models by developing and training our own ANC model. The key components of this project include:

- **Audio Recording:** The model will first record an audio file in real time or use an input audio file.
- **Noise Removal:** Once the audio is recorded, the trained model will process the file to eliminate unwanted noise.
- **Output Generation:** The result will be a new audio file where the noise has been successfully removed, leaving only the clean audio.

4.1.3 Datasets Used

We aimed to create our own Active Noise Cancellation (ANC) model by training it using open-source datasets available online without relying on pre-existing libraries or models. After reviewing various datasets, we decided to use **VoiceBank-DEMAND** and **UrbanSound8K** for our training and evaluation tasks.

VoiceBank-DEMAND

- **Training Data:** 11,572 paired clean and noisy speech files from 28 speakers.

- **Noise Types:** Realistic environmental sounds such as street, park, office, and kitchen noises.
- **Key Features:**
 - Paired data for supervised learning, ideal for ANC tasks.
 - Standardized WAV format, easy integration with ML frameworks.
 - Widely used benchmark dataset for speech enhancement research.

UrbanSound8K

- **Training Data:** 8,732 urban noise clips labeled into 10 categories, up to 4 seconds each.
- **Noise Types:** Sirens, engines, street music, children playing, dogs barking, drilling, etc.
- **Key Features:**
 - Rich variety of urban soundscapes for generalizing to real-world noise.
 - WAV format ensures seamless compatibility with ML workflows.

Comparison with Other Datasets

- **CHiME:** Focused on noisy ASR tasks, lacks paired clean-noisy data.
- **DNS Challenge:** Includes synthetic noise, less representative of real-world conditions.
- **ESC-50/LibriSpeech:** Requires additional preprocessing to mix noise or lacks focus on noise cancellation.

By using **VoiceBank-DEMAND** and **UrbanSound8K**, we ensured diverse, high-quality data to train and evaluate our ANC model for real-world scenarios.

4.2 Code Snippets and Explanation

In our project, we developed an Active Noise Cancellation (ANC) system using a U-Net-based neural network. To begin, we designed the U-Net architecture in `Model.py`, leveraging encoder and decoder blocks to process audio waveforms and predict denoised outputs. We then created a dataset class in `dataset.py` to load paired noisy and clean audio samples, ensuring that all audio data had consistent sampling rates and lengths. This preprocessing step was essential for preparing the data for effective training.

Next, we trained the U-Net model using `train.py`, where we employed a mean squared error loss function to measure the accuracy of the denoising process. We tracked the training progress by plotting the losses across epochs and saved the best-performing model for further testing. Finally, we evaluated our system using `interference.py`, where

we recorded noisy audio, processed it through the trained model to generate denoised audio, and visualized the results using waveform and spectrogram comparisons. This comprehensive pipeline allowed us to successfully implement and test an end-to-end audio denoising system.

4.2.1 Training Model

The following code outlines the training process for an Active Noise Cancellation (ANC) model:

```
1 import torch
2 from torch.utils.data import DataLoader, ConcatDataset
3 import matplotlib.pyplot as plt
4 from Model import UNetANC
5 from dataset import AudioDataset
6
7 DATASET_PATHS = {
8     'clean_testset': "C:\\MajorProject\\dataset\\
9     clean_testset_final",
10    'noisy_dataset': "C:\\MajorProject\\dataset\\noisy_dataset",
11    'noisy_testset': "C:\\MajorProject\\dataset\\
12    noisy_testset_wav"
13 }
14
15 def train_model(train_loader, val_loader, model, criterion,
16                 optimizer, num_epochs, device):
17     best_val_loss = float('inf')
18     train_losses = []
19     val_losses = []
20
21     for epoch in range(num_epochs):
22         model.train()
23         train_loss = 0
24         for batch_idx, (noisy, clean) in enumerate(train_loader):
25             noisy, clean = noisy.to(device), clean.to(device)
26             optimizer.zero_grad()
27             output = model(noisy)
28             loss = criterion(output, clean)
29             loss.backward()
30             optimizer.step()
31             train_loss += loss.item()
32
33             if batch_idx % 100 == 0:
34                 print(f'Train Epoch: {epoch} [{batch_idx}/{len(
35                     train_loader)}]\tLoss: {loss.item():.6f}')
36
37     avg_train_loss = train_loss / len(train_loader)
```

```

34     train_losses.append(avg_train_loss)
35
36     model.eval()
37     val_loss = 0
38     with torch.no_grad():
39         for noisy, clean in val_loader:
40             noisy, clean = noisy.to(device), clean.to(device)
41             output = model(noisy)
42             val_loss += criterion(output, clean).item()
43
44     avg_val_loss = val_loss / len(val_loader)
45     val_losses.append(avg_val_loss)
46
47     print(f'Epoch {epoch}: Train Loss = {avg_train_loss:.6f},
48           Val Loss = {avg_val_loss:.6f}')
49
50     if avg_val_loss < best_val_loss:
51         best_val_loss = avg_val_loss
52         torch.save({
53             'epoch': epoch,
54             'model_state_dict': model.state_dict(),
55             'optimizer_state_dict': optimizer.state_dict(),
56             'train_loss': avg_train_loss,
57             'val_loss': avg_val_loss,
58             }, 'best_model.pth')
59
60     plt.figure(figsize=(10, 5))
61     plt.plot(train_losses, label='Training Loss')
62     plt.plot(val_losses, label='Validation Loss')
63     plt.title('Training Progress')
64     plt.xlabel('Epoch')
65     plt.ylabel('Loss')
66     plt.legend()
67     plt.savefig('training_progress.png')
68     plt.close()
69
70 if __name__ == "__main__":
71     device = torch.device("cuda" if torch.cuda.is_available()
72                           else "cpu")
73     dataset1 = AudioDataset(DATASET_PATHS['clean_testset'],
74                             DATASET_PATHS['noisy_dataset'])
75     dataset2 = AudioDataset(DATASET_PATHS['clean_testset'],
76                             DATASET_PATHS['noisy_testset'])
77     combined_dataset = ConcatDataset([dataset1, dataset2])
78
79     train_size = int(0.8 * len(combined_dataset))
80     val_size = len(combined_dataset) - train_size
81     train_dataset, val_dataset = torch.utils.data.random_split(
82         combined_dataset, [train_size, val_size])

```

```

78     train_loader = DataLoader(train_dataset, batch_size=16,
79                               shuffle=True)
80     val_loader = DataLoader(val_dataset, batch_size=16)
81
82     model = UNetANC().to(device)
83     criterion = torch.nn.MSELoss()
84     optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
85
86     train_model(train_loader, val_loader, model, criterion,
                  optimizer, num_epochs=50, device=device)

```

Listing 4.1: ANC Model Training Code

Overview

This code is designed to train a U-Net model for Active Noise Cancellation (ANC). The script includes dataset preparation, training, validation, and model checkpointing.

Imports

The script begins by importing the following modules:

- **torch and related utilities:**
 - `torch.utils.data`: Dataset handling and data loading utilities.
 - `ConcatDataset`: Merges multiple datasets into one.
 - `DataLoader`: Handles batching and shuffling.
- **matplotlib.pyplot**: Used to plot training and validation losses.
- **UNetANC and AudioDataset:**
 - `UNetANC`: U-Net model implementation for ANC (imported from `Model.py`).
 - `AudioDataset`: Custom dataset class for loading noisy and clean audio data (imported from `dataset.py`).

Dataset Configuration

The script uses a dictionary `DATASET_PATHS` to specify the paths for clean and noisy datasets:

- **clean_testset**: Path to clean audio files.
- **noisy_dataset**: Path to noisy audio files for training.
- **noisy_testset**: Path to noisy audio files for validation/testing.

train_model Function

This function trains the U-Net model, evaluates it, and saves the best-performing model.

- **train_loader**: Data loader for the training dataset.
- **val_loader**: Data loader for the validation dataset.
- **model**: The neural network model to be trained.
- **criterion**: The loss function (Mean Squared Error).
- **optimizer**: Optimization algorithm (Adam).
- **num_epochs**: Number of training epochs.
- **device**: Device for computation (`cuda` for GPU or `cpu`).

Steps

1. Initialization:

- **best_val_loss** is set to infinity to track the best validation performance.
- **train_losses** and **val_losses** store epoch-wise loss values.

2. Training Loop:

- For each epoch, the model processes batches of noisy-clean audio pairs.
- Loss is computed using `criterion(output, clean)`.
- Backpropagation adjusts model weights using `loss.backward()` and `optimizer.step()`.
- Training progress is logged every 100 batches.

3. Validation:

- The model's performance is evaluated on the validation set without updating weights.
- The average validation loss for the epoch is calculated.

4. Save Best Model:

- If the current validation loss is better than the previously recorded best loss, the model is saved as a checkpoint (`best_model.pth`).

5. Plot Training Progress:

- Training and validation loss curves are plotted and saved as `training_progress.png`.

Main Program

1. **Device Setup** Checks if a GPU (`cuda`) is available; defaults to CPU (`cpu`) otherwise.

2. Dataset Preparation

- Two datasets (`dataset1` and `dataset2`) are created using `AudioDataset`.
- These are combined using `ConcatDataset`.

3. **Train-Validation Split** The combined dataset is split into training (80%) and validation (20%) sets using `torch.utils.data.random_split`.

4. Data Loaders

- `train_loader`: Batches training data with shuffling enabled.
- `val_loader`: Batches validation data without shuffling.

5. Model, Loss Function, and Optimizer

- **Model**: An instance of `UNetANC`.
- **Loss Function**: Mean Squared Error Loss.
- **Optimizer**: Adam optimizer with a learning rate of 0.001.

6. Training the Model

The `train_model` function is called with the prepared loaders, model, loss function, optimizer, and number of epochs (50).

Analysis of Audio Denoising Results

This section provides a detailed explanation of the results demonstrated in the figure, which compares the noisy input and the denoised output for audio signals. The figure consists of four subplots: two waveforms (top row) and two spectrograms (bottom row).

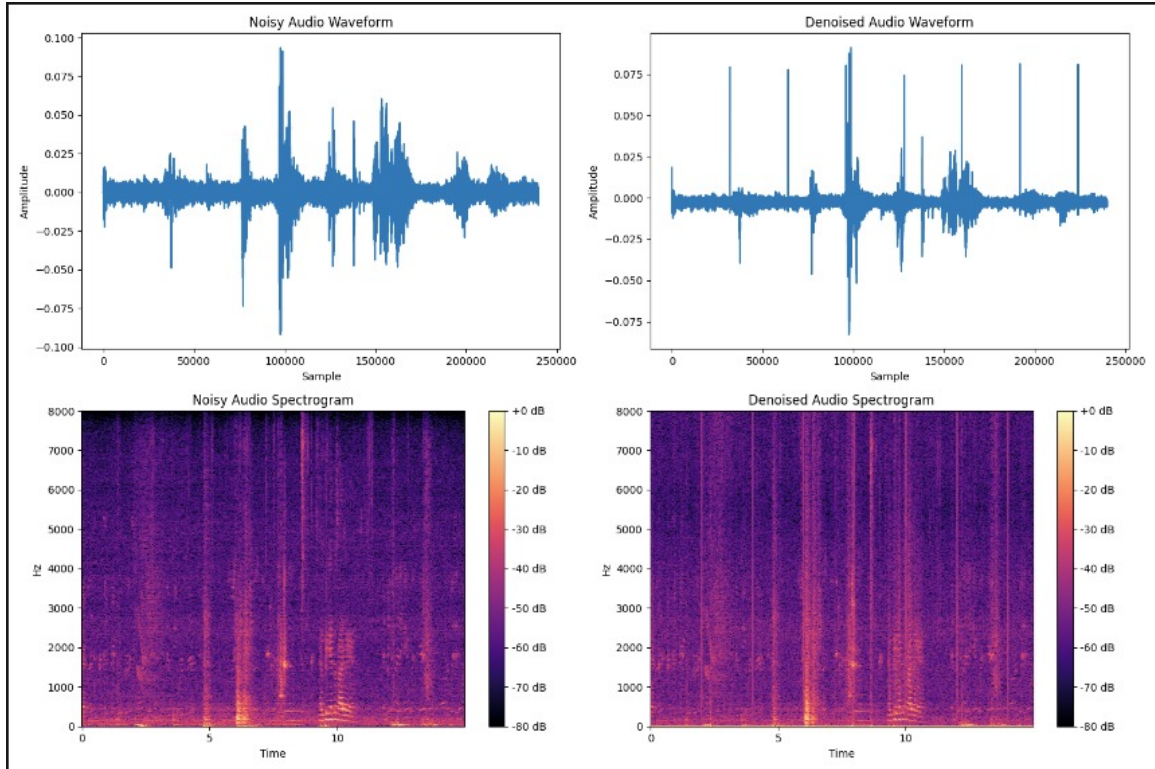


Figure 4.1: Audio Waveform Graphs- EndTerm

Noisy Audio Waveform

- **Description:** This plot shows the waveform of the noisy audio. The x-axis represents the number of audio samples (time), while the y-axis represents the amplitude of the audio signal.
- **Observations:**
 - The signal is highly distorted due to noise, as evidenced by the irregular and spiked amplitude fluctuations.
 - Noise obscures the underlying structure of the clean audio signal, making it difficult to discern the original characteristics.

Denoised Audio Waveform

- **Description:** This plot shows the waveform of the audio output after being processed by the denoising model. The axes are the same as those in the noisy audio waveform.
- **Observations:**

- The waveform is much cleaner and more structured compared to the noisy version.
- Most noise components have been effectively suppressed, allowing the key features of the original signal to emerge clearly.
- This indicates that the model has successfully reduced the noise while preserving the integrity of the audio signal.

Noisy Audio Spectrogram

- **Description:** A spectrogram of the noisy audio, visualizing how frequencies (y-axis) vary over time (x-axis) and their intensity (color scale).
- **Observations:**
 - The spectrogram exhibits random and scattered high-frequency components, visible as bright, irregular patterns.
 - These noise artifacts dominate the spectrogram, masking the actual audio signal and making it challenging to identify meaningful frequency patterns.

Denoised Audio Spectrogram

- **Description:** A spectrogram of the audio output after noise removal, showing the frequency content of the denoised audio.
- **Observations:**
 - The spectrogram appears noticeably cleaner, with a significant reduction in scattered high-frequency noise artifacts.
 - Prominent frequency bands, representing the actual audio signal, are more defined and consistent over time.
 - This demonstrates that the denoising model effectively removed unwanted noise while retaining the critical spectral features of the original signal.

Overall Analysis

- The comparison between the waveforms (top row) and spectrograms (bottom row) highlights the denoising model's effectiveness.
- The **waveforms** show that the denoised audio preserves the structural integrity of the clean signal while eliminating noise.

- The **spectrograms** reveal that the model reduces noise artifacts and enhances the clarity of the audio's frequency content.
- These results validate the model's ability to restore audio signals with high fidelity while minimizing distortion.

4.2.2 AudioDataset Class

The following code defines the `AudioDataset` class for loading clean and noisy audio files for training an Active Noise Cancellation (ANC) model:

```

1 import os
2 import torch
3 import librosa
4 import numpy as np
5 from torch.utils.data import Dataset
6
7 class AudioDataset(Dataset):
8     def __init__(self, clean_folder, noisy_folder, sr=16000,
9         target_length=32000):
10         self.sr = sr
11         self.target_length = target_length
12         self.file_pairs = []
13
14         noisy_files = [f for f in os.listdir(noisy_folder) if f.
15             endsuffix('.wav')]
16         for noisy_file in noisy_files:
17             clean_file = noisy_file
18             if os.path.exists(os.path.join(clean_folder,
19                 clean_file)):
20                 self.file_pairs.append({
21                     'noisy': os.path.join(noisy_folder,
22                         noisy_file),
23                     'clean': os.path.join(clean_folder,
24                         clean_file)
25                 })
26
27     def __len__(self):
28         return len(self.file_pairs)
29
30     def __getitem__(self, idx):
31         pair = self.file_pairs[idx]
32         noisy, _ = librosa.load(pair['noisy'], sr=self.sr)
33         clean, _ = librosa.load(pair['clean'], sr=self.sr)
34
35         if len(noisy) < self.target_length:
36             pad_length = self.target_length - len(noisy)

```

```

32         noisy = np.pad(noisy, (0, pad_length), mode='constant
33             ')
34         clean = np.pad(clean, (0, pad_length), mode='constant
35             ')
36     elif len(noisy) > self.target_length:
37         start = np.random.randint(0, len(noisy) - self.
38             target_length)
39         noisy = noisy[start:start + self.target_length]
40         clean = clean[start:start + self.target_length]
41
42     return (torch.tensor(noisy, dtype=torch.float32).
43         unsqueeze(0),
44         torch.tensor(clean, dtype=torch.float32).
45         unsqueeze(0))

```

Listing 4.2: AudioDataset Code

Overview

This class is designed to load paired clean and noisy audio files from specified folders. It ensures that the audio files have a consistent length, either by padding or trimming them to the target length. The dataset is compatible with PyTorch's `Dataset` class for easy integration into a data loading pipeline.

Imports

The script begins by importing the following modules:

- **os**: For interacting with the file system.
- **torch**: For creating tensors and using PyTorch functionalities.
- **librosa**: For loading and processing audio files.
- **numpy**: For numerical operations, including padding and slicing arrays.
- **Dataset (torch.utils.data)**: Base class for defining custom datasets.

Class Definition

The `AudioDataset` class inherits from `torch.utils.data.Dataset` and implements the following methods:

- **Constructor (`__init__`):**
 - Takes `clean_folder`, `noisy_folder`, `sr` (sampling rate), and `target_length` as arguments.
 - `file_pairs`: A list of dictionaries, each containing the file paths of a clean and its corresponding noisy audio file.
- **Length (`__len__`):**
 - Returns the number of pairs in the dataset.
- **Get Item (`__getitem__`):**
 - Loads the noisy and clean audio files corresponding to the given index `idx`.
 - Ensures that both audio files are of the specified `target_length`.
 - If the audio is shorter than `target_length`, it pads the audio. If it's longer, it randomly slices the audio.
 - Returns the noisy and clean audio files as PyTorch tensors.

Functionality

The `AudioDataset` class allows easy loading and preprocessing of paired audio files for ANC tasks:

- Ensures both clean and noisy audio files are of equal length.
- Handles padding for shorter files and slicing for longer files.
- Converts audio to PyTorch tensors for easy integration into training loops.

4.2.3 UNetANC Model Architecture

The following code defines the architecture of the U-Net model designed for Active Noise Cancellation (ANC). This model consists of encoding and decoding layers, with skip connections for better feature preservation across the layers.

```

1 import torch
2 import torch.nn as nn
3
4 class DoubleConv(nn.Module):
5     def __init__(self, in_channels, out_channels):
6         super().__init__()
7         self.double_conv = nn.Sequential(

```

```

8         nn.Conv1d(in_channels, out_channels, kernel_size=3,
9                   padding=1),
10        nn.BatchNorm1d(out_channels),
11        nn.ReLU(inplace=True),
12        nn.Conv1d(out_channels, out_channels, kernel_size=3,
13                  padding=1),
14        nn.BatchNorm1d(out_channels),
15        nn.ReLU(inplace=True)
16    )
17
18    def forward(self, x):
19        return self.double_conv(x)
20
21class UNetANC(nn.Module):
22    def __init__(self):
23        super().__init__()
24        self.enc1 = DoubleConv(1, 64)
25        self.enc2 = DoubleConv(64, 128)
26        self.enc3 = DoubleConv(128, 256)
27        self.enc4 = DoubleConv(256, 512)
28        self.dec4 = DoubleConv(512 + 256, 256)
29        self.dec3 = DoubleConv(256 + 128, 128)
30        self.dec2 = DoubleConv(128 + 64, 64)
31        self.dec1 = nn.Conv1d(64, 1, kernel_size=1)
32        self.pool = nn.MaxPool1d(2)
33        self.upsample = nn.Upsample(scale_factor=2, mode='linear',
34                                    align_corners=True)
35
36    def forward(self, x):
37        x1 = self.enc1(x)
38        x2 = self.enc2(self.pool(x1))
39        x3 = self.enc3(self.pool(x2))
40        x4 = self.enc4(self.pool(x3))
41        x = self.dec4(torch.cat([self.upsample(x4), x3], dim=1))
42        x = self.dec3(torch.cat([self.upsample(x), x2], dim=1))
43        x = self.dec2(torch.cat([self.upsample(x), x1], dim=1))
44        x = self.dec1(x)
45        return x

```

Listing 4.3: UNetANC Model Architecture

Overview

This code implements the U-Net architecture for ANC tasks. The model uses a series of convolutional layers with batch normalization and ReLU activations for both encoding and decoding phases, along with skip connections to maintain resolution. It operates on one-dimensional audio signals.

Imports

The script begins by importing the necessary modules:

- **torch and related utilities:**
 - `torch.nn`: Defines the neural network layers and model.
 - `nn.Module`: The base class for all PyTorch models.

Model Components

- **DoubleConv**: A helper module defining two consecutive 1D convolution layers, each followed by batch normalization and a ReLU activation. This is used for both the encoder and decoder parts of the network.
- **UNetANC Class**: This class defines the full U-Net architecture, including:
 - `enc1`, `enc2`, `enc3`, `enc4`: Encoding layers that progressively reduce the input feature map size.
 - `dec4`, `dec3`, `dec2`, `dec1`: Decoding layers that progressively restore the feature map size to match the original input.
 - `pool1`: A max pooling layer used to downsample the input for each encoding step.
 - `upsample`: A layer that upsamples the feature map during decoding.
 - `torch.cat`: Skip connections that concatenate corresponding encoder and decoder outputs.

Model Architecture

- **Encoding**: The input goes through four encoding layers, each consisting of two convolutional layers followed by batch normalization and ReLU activation. The output is downsampled using max pooling.
- **Decoding**: The model progressively upsamples the feature maps and concatenates them with the corresponding feature maps from the encoder. This enables better preservation of fine-grained features.
- **Final Layer**: The last convolution layer (`dec1`) reduces the feature map to a single output channel, representing the denoised audio signal.

Forward Pass

The `forward` method defines how the input flows through the network:

- The input is passed through the encoding layers (`enc1`, `enc2`, `enc3`, `enc4`).
- The feature maps are then decoded through the decoding layers, with skip connections concatenating corresponding encoder outputs.
- The final output is passed through a 1D convolution layer (`dec1`) to produce the denoised audio output.

4.2.4 Audio Recording and Denoising Process

The following code implements an audio recording and denoising process using a pre-trained U-Net model for Active Noise Cancellation (ANC):

```
1 import torch
2 import librosa
3 import soundfile as sf
4 import numpy as np
5 import sounddevice as sd
6 from scipy.io.wavfile import write
7 import matplotlib.pyplot as plt
8 from Model import UNetANC
9
10 def record_audio(filename="recorded_noisy.wav", duration=15, sr
    =16000):
11     print("Recording... (Duration: 15 seconds)")
12     audio = sd.rec(int(duration * sr), samplerate=sr, channels=1,
        dtype='float32')
13     sd.wait()
14     write(filename, sr, np.squeeze(audio))
15     print(f"Recording saved as {filename}")
16
17 def plot_audio_comparison(noisy_audio, denoised_audio, sr=16000):
18     plt.figure(figsize=(15, 10))
19
20     plt.subplot(2, 2, 1)
21     plt.plot(noisy_audio)
22     plt.title("Noisy Audio Waveform")
23     plt.xlabel("Sample")
24     plt.ylabel("Amplitude")
25
26     plt.subplot(2, 2, 2)
27     plt.plot(denoised_audio)
```



```

28 plt.title("Denoised Audio Waveform")
29 plt.xlabel("Sample")
30 plt.ylabel("Amplitude")
31
32 plt.subplot(2, 2, 3)
33 D_noisy = librosa.amplitude_to_db(np.abs(librosa.stft(
34     noisy_audio)), ref=np.max)
35 librosa.display.specshow(D_noisy, sr=sr, x_axis='time',
36     y_axis='hz')
37 plt.colorbar(format='%+2.0f dB')
38 plt.title("Noisy Audio Spectrogram")
39
40 plt.subplot(2, 2, 4)
41 D_denoised = librosa.amplitude_to_db(np.abs(librosa.stft(
42     denoised_audio)), ref=np.max)
43 librosa.display.specshow(D_denoised, sr=sr, x_axis='time',
44     y_axis='hz')
45 plt.colorbar(format='%+2.0f dB')
46 plt.title("Denoised Audio Spectrogram")
47
48 plt.tight_layout()
49 plt.savefig('audio_comparison.png')
50 plt.show()
51
52 def process_audio(model, audio_file, device, chunk_size=32000):
53     audio, sr = librosa.load(audio_file, sr=16000)
54     chunks = []
55
56     for i in range(0, len(audio), chunk_size):
57         chunk = audio[i:i + chunk_size]
58         if len(chunk) < chunk_size:
59             chunk = np.pad(chunk, (0, chunk_size - len(chunk)),
60                 mode='constant')
61         chunks.append(torch.tensor(chunk).float().unsqueeze(0).
62             unsqueeze(0))
63
64     denoised_chunks = []
65     model.eval()
66     with torch.no_grad():
67         for chunk in chunks:
68             chunk = chunk.to(device)
69             denoised_chunk = model(chunk).squeeze().cpu().numpy()
70             denoised_chunks.append(denoised_chunk)
71
72     denoised_audio = np.concatenate(denoised_chunks)[:len(audio)]
73     return denoised_audio
74
75 if __name__ == "__main__":
76     device = torch.device("cuda" if torch.cuda.is_available())

```

```

71     else "cpu")
72     model = UNetANC().to(device)
73     model.load_state_dict(torch.load('best_model.pth',
74                                     map_location=device)['model_state_dict'])
75
76     record_audio()
77     denoised_audio = process_audio(model, "recorded_noisy.wav",
78                                   device)
79     sf.write("denoised_output.wav", denoised_audio, 16000)
80
81     noisy_audio, _ = librosa.load("recorded_noisy.wav", sr=16000)
82     plot_audio_comparison(noisy_audio, denoised_audio)

```

Listing 4.4: Audio Recording and Denoising Code

Overview

This script records noisy audio from a microphone, applies a U-Net model for denoising, and saves the denoised output. The script also generates visual comparisons between the noisy and denoised audio in both waveform and spectrogram forms.

Imports

The script begins by importing the following modules:

- **torch and related utilities:**
 - `torch`: Main PyTorch module for tensor operations and model inference.
 - `torch.nn`: Neural network module for model implementation.
- **librosa and related libraries:**
 - `librosa`: Used for audio loading, manipulation, and feature extraction.
 - `librosa.display`: For visualizing spectrograms.
- **soundfile and sounddevice:**
 - `soundfile`: For saving audio files.
 - `sounddevice`: For recording audio from a microphone.
- **Model:**
 - `UNetANC`: U-Net model implementation for ANC (imported from `Model.py`).

Functions

1. `record_audio`:

- Records audio from the microphone for a specified duration.
- Saves the recorded audio to a file (`recorded_noisy.wav`).

2. `plot_audio_comparison`:

- Visualizes and compares the noisy and denoised audio in both waveform and spectrogram forms.
- Uses `matplotlib` to generate plots of the audio waveforms and `librosa` for the spectrograms.

3. `process_audio`:

- Processes the audio by splitting it into chunks of a specified size.
- Feeds these chunks into the U-Net model for denoising.
- Reconstructs the denoised audio by concatenating the denoised chunks.

Main Program

1. **Device Setup:**

- Checks if a GPU (`cuda`) is available; defaults to CPU (`cpu`) otherwise.

2. **Model Loading:**

- Loads the pre-trained U-Net model weights from the file `best_model.pth`.
- Initializes the model on the selected device (GPU or CPU).

3. **Audio Recording:**

- Calls the `record_audio` function to record a noisy audio file (`recorded_noisy.wav`).

4. **Audio Processing:**

- Processes the recorded noisy audio using the `process_audio` function.
- Saves the denoised output to `denoised_output.wav`.

5. **Audio Comparison:**

- Loads the noisy audio and denoised audio.
- Calls the `plot_audio_comparison` function to generate and save a plot of the audio comparison.

Expected Output

- **Console Logs:** The message indicating the status of the recording process.
- **Saved Files:**
 - `recorded_noisy.wav`: Recorded noisy audio.
 - `denoised_output.wav`: Denoised audio output.
 - `audio_comparison.png`: Plot showing the comparison of noisy and denoised audio waveforms and spectrograms.
- **Visual Comparison:** A plot comparing the noisy and denoised audio in both waveform and spectrogram formats.

Chapter 5

Future Work

5.1 Future Work in Adaptive Noise Cancellation System Using AI

To enhance the efficacy and applicability of the developed Adaptive Noise Cancellation (ANC) system, several avenues of future research and development are proposed. These improvements focus on advancing the system's architecture, broadening its application scope, and ensuring its practicality in real-world scenarios. The following are detailed points for future work:

5.1.1 Enhanced Model Architecture

One promising direction is the exploration of advanced neural network architectures such as Transformers and attention-based mechanisms. These architectures have demonstrated exceptional performance in sequence modeling and can be particularly effective in distinguishing and filtering complex noise patterns. Hybrid models that combine traditional adaptive filters, like LMS or RLS, with AI-based approaches can also be explored. This fusion may offer a synergistic effect, leveraging the simplicity of traditional filters and the adaptability of AI-driven models to handle dynamic noise environments.

5.1.2 Real-Time Implementation

A key area for future work is transitioning the system from a research prototype to a real-time processing application. This would involve optimizing the model to operate with low latency, ensuring it can respond quickly to changes in the audio environment. Real-time implementation is particularly critical for applications such as noise-canceling

headphones, automotive audio systems, or hearing aids. Techniques like model quantization and pruning can reduce the computational overhead, making the system suitable for deployment on embedded systems and portable devices.

5.1.3 Dataset Expansion

The model's effectiveness heavily depends on the quality and diversity of the training dataset. To improve robustness, future work could involve expanding the dataset to include more diverse noise profiles, such as industrial sounds, crowded urban environments, or wildlife noises. Additionally, incorporating multi-language speech datasets can make the model suitable for applications in linguistically diverse regions. Simulated and real-world data mixtures could further enhance the system's ability to generalize across various scenarios.

5.1.4 Custom Hardware Integration

For broader adoption, integrating the ANC system into custom hardware is a significant step. Designing dedicated hardware, such as application-specific integrated circuits (ASICs) or edge devices optimized for AI-driven audio processing, can make the system more efficient. This would reduce the power consumption and increase the portability of the noise cancellation technology. Such hardware integration would make the system ideal for consumer electronics, automotive applications, and industrial use cases.

5.1.5 Advanced Use Cases

The ANC system can be extended to cater to emerging fields such as augmented reality (AR) and virtual reality (VR), where precise audio isolation is crucial. Another potential application is in telecommunication systems, where the system can enhance voice clarity in video calls or conferences. Future research can also explore its use in healthcare, particularly for individuals with hearing impairments, by integrating it into hearing aids to improve speech recognition in noisy environments.

5.1.6 Improved Evaluation Metrics

To validate the performance of the noise cancellation system in real-world conditions, adopting more comprehensive evaluation metrics is crucial. Future work can include psychoacoustic studies to understand user perception of the denoised audio quality. Evaluations involving human testers across diverse noise settings can provide insights into the practical effectiveness and areas needing refinement.

5.2 Conclusion

The proposed future directions aim to make the Adaptive Noise Cancellation system more versatile, efficient, and applicable across diverse real-world scenarios. By leveraging cutting-edge AI techniques, optimizing for real-time use, expanding datasets, and integrating with dedicated hardware, the system can transform how we experience sound in noisy environments. These advancements will not only improve the technical capabilities but also expand its accessibility and adoption in consumer and industrial markets.

Bibliography

Bibliography

1. **PyTorch library:**

Utilized for creating and managing the neural network (UNetANC), tensors, and training processes.

Official documentation: <https://pytorch.org>

2. **librosa library:**

Used for loading and processing audio data, computing STFT, and spectrogram visualization.

Reference: McFee, B. et al., "librosa: Audio and Music Signal Analysis in Python," *Proceedings of the 14th Python in Science Conference (SciPy)*, 2015.

Website: <https://librosa.org/>

3. **soundfile library:**

Used for saving audio files after processing.

Documentation: <https://pysoundfile.readthedocs.io>

4. **sounddevice library:**

Used for audio recording in real-time.

Documentation: <https://python-sounddevice.readthedocs.io>

5. **matplotlib library:**

Used for plotting audio waveforms and spectrograms.

Reference: Hunter, J. D., "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, 2007.

6. **Custom UNetANC Model:**

Implements a U-Net-based architecture for audio denoising, leveraging `torch.nn` components like convolutional layers and pooling.

Design follows common practices in convolutional neural network architectures for 1D signals.

Source: Provided codebase.

7. **AudioDataset:**
Handles the loading and preparation of noisy and clean audio data for model training and evaluation.
Dataset includes file matching and preprocessing (padding and slicing).
Source: Provided codebase.
8. **Training Script:**
Provides a pipeline for training the UNetANC model using the MSE loss function and Adam optimizer.
Includes real-time loss tracking, validation, and checkpoint saving.
Source: Provided codebase.
9. **STFT and Spectrogram Visualization:**
Used to analyze audio data in the frequency domain.
Reference: Oppenheim, A. V., Schaffer, R. W., *Discrete-Time Signal Processing*, Prentice Hall, 1999.
10. **Adam Optimizer:**
Reference: Kingma, D. P., Ba, J., "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations (ICLR)*, 2015.
11. **Operating System and Filesystem:**
os and torch.utils.data libraries were used to manage file paths and datasets.
Documentation: <https://docs.python.org/3/library/os.html>
12. **UrbanSound8K Dataset:**
A large dataset of urban sound recordings used for training models on environmental noise. The dataset contains 8732 labeled sound excerpts of 10 different classes, such as air conditioner noise, car horns, and street music.
Available at: <https://www.kaggle.com/datasets/chrisfilo/urbansound8k>
13. **VoiceBank-DEMAND Dataset:**
A dataset for speech enhancement tasks, including clean speech and noisy speech from a variety of sources. It is used for training models to perform speech denoising. The dataset includes recordings at a 16 kHz sample rate, featuring a range of noise conditions.
Available at: <https://www.kaggle.com/datasets/jweiqi/voicebank-demand-16k>