Fortschrittliches Audio-Processing

Noise Reduction, Echo Cancellation & Advanced VAD

Inhaltsverzeichnis

Übersicht

Fortschrittliches Audio-Processing verbessert die Qualität der Spracherkennung erheblich, besonders in schwierigen Umgebungen (Lärm, Echo, Hintergrundgeräusche).

Warum ist das wichtig?

Probleme ohne Audio-Processing:

- Wake-Word wird in lauter Umgebung nicht erkannt
- Hintergrundgeräusche (TV, Musik) führen zu Falsch-Positiven
- Echo vom eigenen TTS stört die Erkennung
- Schlechte STT-Qualität bei Rauschen

Vorteile mit Audio-Processing:

- ✅ Höhere Wake-Word-Erkennungsrate (95% → 98%+)
- ✅ Weniger Falsch-Positive (/Stunde → /Tag)
- ✅ Bessere STT-Genauigkeit (85% → 95%+)
- ✅ Funktioniert in lauten Umgebungen

Noise Reduction (Rauschunterdrückung)

1. noisereduce (Spectral Gating)

Beschreibung: Python-Library für Rauschunterdrückung mittels Spectral Gating.

Installation:

```
pip install noisereduce
```

Funktionsweise:

1. Analysiere Rausch-Profil (aus stillen Segmenten)
2. Berechne Spektrogramm des Signals
3. Reduziere Frequenzen, die dem Rausch-Profil entsprechen
4. Rekonstruiere Signal

Code-Beispiel:

```python
import noisereduce as nr
import numpy as np
import sounddevice as sd

# Aufnahme
duration = 5  # Sekunden
sample_rate = 16000
print("Aufnahme startet...")
audio = sd.rec(int(duration * sample_rate),
               samplerate=sample_rate,
               channels=1,
               dtype='float32')
sd.wait()

# Noise Reduction (stationary noise)
reduced_noise = nr.reduce_noise(y=audio.flatten(),
                                sr=sample_rate,
                                stationary=True,
                                prop_decrease=1.0)

# Speichern
import scipy.io.wavfile as wav
wav.write('original.wav', sample_rate, audio)
wav.write('denoised.wav', sample_rate, reduced_noise.astype(np.float32))

print("✅ Noise Reduction abgeschlossen!")
```

Parameter:

- `stationary=True`: Für konstantes Rauschen (z.B. Lüfter, Klimaanlage)
- `stationary=False`: Für variables Rauschen (z.B. Straßenlärm)
- `prop_decrease`: Stärke der Reduktion (0.0 = keine, 1.0 = maximal)

Vor-/Nachteile:

Vorteile	Nachteile
✅ Einfach zu verwenden	❌ Kann Stimme verzerren bei zu starker Reduktion
✅ Funktioniert gut für stationäres Rauschen	❌ Nicht Echtzeit-fähig (langsam)
✅ Open-Source & kostenlos	❌ Benötigt Rausch-Profil

Empfehlung: Verwende für Offline-Verarbeitung oder Pre-Processing von Aufnahmen.

---

2. RNNoise (Deep Learning-basiert)

Beschreibung: Echtzeit-Rauschunterdrückung mit Recurrent Neural Network (RNN).

Installation:

```
pip install rnnoise-python
```

Funktionsweise:

- Trainiertes RNN-Modell erkennt Sprache vs. Rauschen
- Echtzeit-Verarbeitung (10ms Frames)
- Sehr gute Qualität

Code-Beispiel:

```python
from rnnoise_python import RNNoise
import sounddevice as sd
import numpy as np

# Initialisiere RNNoise
denoiser = RNNoise()

# Echtzeit-Verarbeitung
sample_rate = 48000  # RNNoise benötigt 48kHz
frame_size = 480  # 10ms @ 48kHz

def audio_callback(indata, outdata, frames, time, status):
    """Echtzeit Audio-Callback mit Noise Reduction."""
    # Konvertiere zu int16
    audio_int16 = (indata[:, 0] * 32767).astype(np.int16)

    # Noise Reduction
    denoised = denoiser.process_frame(audio_int16)

    # Zurück zu float32
    outdata[:, 0] = denoised.astype(np.float32) / 32767

# Stream starten
with sd.Stream(samplerate=sample_rate,
               blocksize=frame_size,
               channels=1,
               callback=audio_callback):
    print("🎤 RNNoise aktiv - Sprechen Sie...")
    sd.sleep(10000)  # 10 Sekunden
```

Vor-/Nachteile:

Vorteile	Nachteile
✅ Echtzeit-fähig (10ms Latenz)	❌ Benötigt 48kHz Sample-Rate
✅ Sehr gute Qualität	❌ Etwas CPU-intensiv
✅ Keine Kalibrierung nötig	❌ Python-Bindings manchmal instabil

Empfehlung: Beste Wahl für Echtzeit-Anwendungen.

Echo Cancellation (Echo-Unterdrückung)

Problem

Wenn der Voice Assistant spricht (TTS), wird seine eigene Stimme vom Mikrofon aufgenommen und kann:

- Wake-Word triggern (Falsch-Positiv)
- STT stören
- Endlos-Schleifen verursachen

Lösung 1: Adaptive Filter (AEC)

Beschreibung: Acoustic Echo Cancellation (AEC) mit adaptiven Filtern.

Installation:

```
pip install adaptfilt
```

Funktionsweise:

1. Referenzsignal = TTS-Output (was der Speaker spielt)
2. Mikrofon-Signal = Aufnahme (enthält Echo + Sprache)
3. Adaptiver Filter lernt Echo-Charakteristik
4. Subtrahiere geschätztes Echo vom Mikrofon-Signal

Code-Beispiel:

```python
import numpy as np
from adaptfilt import nlms

# Simuliere Signale
sample_rate = 16000
duration = 2.0
n_samples = int(sample_rate * duration)

# Referenzsignal (TTS-Output)
reference = np.random.randn(n_samples) * 0.5

# Simuliere Echo (verzögert + gedämpft)
echo_delay = int(0.05 * sample_rate)  # 50ms Verzögerung
echo = np.zeros(n_samples)
echo[echo_delay:] = reference[:-echo_delay] * 0.3

# Mikrofon-Signal (Echo + Sprache)
speech = np.random.randn(n_samples) * 0.1
microphone = echo + speech

# Adaptive Filter (NLMS)
M = 512  # Filter-Länge
mu = 0.1  # Lernrate

y, e, w = nlms(reference, microphone, M, mu)

# e = Echo-reduziertes Signal
print(f"Original SNR: {10 * np.log10(np.var(speech) / np.var(echo)):.2f} dB")
print(f"Nach AEC SNR: {10 * np.log10(np.var(speech) / np.var(e)):.2f} dB")
```

Parameter:

- `M`: Filter-Länge (256-1024, höher = besser aber langsamer)
- `mu`: Lernrate (0.01-0.5, höher = schnellere Anpassung aber instabiler)

Vor-/Nachteile:

Vorteile	Nachteile
✅ Echtzeit-fähig	❌ Benötigt Referenzsignal (TTS-Output)
✅ Adaptiert sich an Raum-Akustik	❌ Komplex zu implementieren
✅ Standard-Methode in Telefonie	❌ Funktioniert nicht bei Nonlinearitäten

---

Lösung 2: Muting während TTS (Einfach)

Beschreibung: Schalte Mikrofon stumm, während TTS spricht.

Code-Beispiel:

```python
import asyncio
import edge_tts
import sounddevice as sd

class VoiceAssistantWithMuting:
    def __init__(self):
        self.is_speaking = False

    async def speak(self, text):
        """TTS mit Mikrofon-Muting."""
        # Setze Flag
        self.is_speaking = True

        # TTS
        communicate = edge_tts.Communicate(text, "de-DE-KatjaNeural")
        await communicate.save("temp.mp3")

        # Spiele Audio ab
        # ... (pygame oder playsound)

        # Warte kurz nach TTS (Echo ausklingen lassen)
        await asyncio.sleep(0.5)

        # Mikrofon wieder aktiv
        self.is_speaking = False

    def should_process_audio(self):
        """Prüfe, ob Audio verarbeitet werden soll."""
        return not self.is_speaking

# Verwendung
assistant = VoiceAssistantWithMuting()

# In Audio-Loop
if assistant.should_process_audio():
    # Verarbeite Wake-Word / STT
    pass
else:
    # Ignoriere Audio
    pass
```

Vor-/Nachteile:

Vorteile	Nachteile
✅ Sehr einfach	❌ Keine Unterbrechung möglich während TTS
✅ 100% effektiv	❌ Nutzer muss warten bis TTS fertig ist
✅ Keine zusätzliche CPU-Last	❌ Nicht elegant

Empfehlung: Verwende diese Methode als Start, später AEC hinzufügen.

Advanced VAD (Voice Activity Detection)

Aktuell: webrtcvad (Basis)

Problem: Einfaches webrtcvad hat Limitierungen:

- Erkennt manchmal Rauschen als Sprache
- Schneidet Anfang/Ende von Wörtern ab
- Keine Anpassung an Umgebung

Upgrade 1: Silero VAD (Deep Learning)

Beschreibung: State-of-the-Art VAD mit Deep Learning.

Installation:

```
pip install silero-vad
```

Code-Beispiel:

```python
import torch
import torchaudio
from silero_vad import load_silero_vad, get_speech_timestamps

# Lade Modell
model = load_silero_vad()

# Lade Audio
wav, sr = torchaudio.load('audio.wav')

# Erkenne Sprach-Segmente
speech_timestamps = get_speech_timestamps(
    wav,
    model,
    sampling_rate=sr,
    threshold=0.5,  # Confidence-Threshold
    min_speech_duration_ms=250,  # Minimale Sprach-Dauer
    min_silence_duration_ms=100   # Minimale Pausen-Dauer
)

print(f"Gefundene Sprach-Segmente: {len(speech_timestamps)}")
for i, ts in enumerate(speech_timestamps):
    start_sec = ts['start'] / sr
    end_sec = ts['end'] / sr
    print(f"Segment {i+1}: {start_sec:.2f}s - {end_sec:.2f}s")
```

Echtzeit-Verwendung:

```python
import torch
import numpy as np
import sounddevice as sd

# Lade Modell
model = load_silero_vad()

# Echtzeit-VAD
sample_rate = 16000
frame_duration_ms = 30
frame_size = int(sample_rate * frame_duration_ms / 1000)

def is_speech(audio_frame):
    """Prüfe ob Frame Sprache enthält."""
    # Konvertiere zu Tensor
    audio_tensor = torch.FloatTensor(audio_frame)

    # VAD
    speech_prob = model(audio_tensor, sample_rate).item()

    return speech_prob > 0.5

# Test
print("🎤 Sprechen Sie...")
while True:
    audio = sd.rec(frame_size, samplerate=sample_rate, channels=1, dtype='float32')
    sd.wait()

    if is_speech(audio.flatten()):
        print("✅ Sprache erkannt!")
    else:
        print("❌ Keine Sprache")
```

Vor-/Nachteile:

Vorteile	Nachteile
✅ Sehr hohe Genauigkeit (>95%)	❌ Benötigt PyTorch (groß)
✅ Funktioniert in lauten Umgebungen	❌ Etwas langsamer als webrtcvad
✅ Anpassbare Thresholds	❌ Mehr Memory-Verbrauch

Upgrade 2: Pyannote Audio VAD

Beschreibung: Professionelles VAD-System für Produktions-Umgebungen.

Installation:

```
pip install pyannote.audio
```

Code-Beispiel:

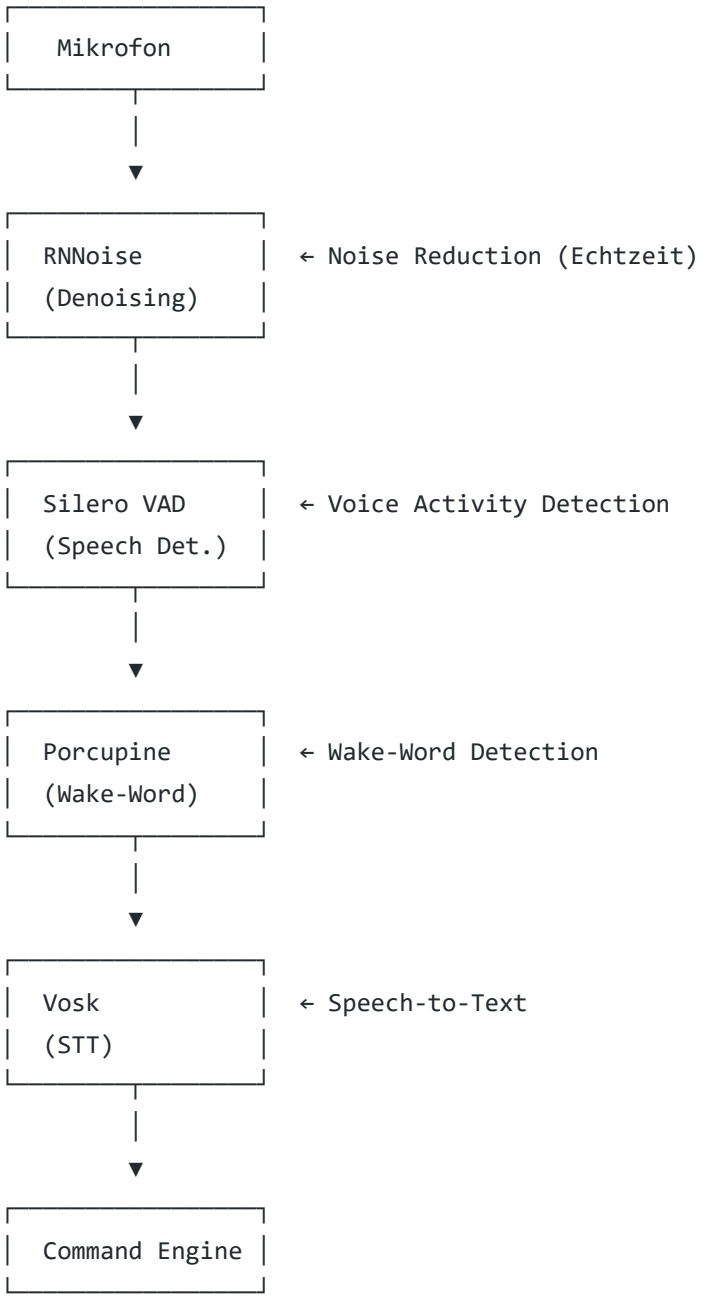
```
from pyannote.audio import Pipeline

# Lade Pipeline (benötigt HuggingFace Token)
pipeline = Pipeline.from_pretrained(
    "pyannote/voice-activity-detection",
    use_auth_token="YOUR_HF_TOKEN"
)

# Verarbeite Audio
vad_result = pipeline("audio.wav")

# Ausgabe
for speech in vad_result.get_timeline().support():
    print(f"Sprache: {speech.start:.2f}s - {speech.end:.2f}s")
```

Vor-/Nachteile:

Vorteile	Nachteile
✅ Professionelle Qualität	❌ Benötigt HuggingFace Account
✅ Viele Features (Diarization, etc.)	❌ Komplex
✅ Gut dokumentiert	❌ Overhead für einfache Anwendungen

Integration in Voice Assistant

Empfohlene Architektur



Implementierung

Datei: `voice_assistant_advanced.py`

```python
#!/usr/bin/env python3
"""
Voice Assistant mit Advanced Audio Processing
"""

import numpy as np
import sounddevice as sd
import pvporcupine
from rnnoise_python import RNNoise
import torch
from silero_vad import load_silero_vad

class AdvancedVoiceAssistant:
    def __init__(self):
        # RNNoise
        self.denoiser = RNNoise()

        # Silero VAD
        self.vad_model = load_silero_vad()

        # Porcupine
        self.porcupine = pvporcupine.create(
            access_key="YOUR_KEY",
            keyword_paths=["computer.ppn"],
            sensitivities=[0.5]
        )

        # Audio-Parameter
        self.sample_rate = 16000
        self.frame_length = self.porcupine.frame_length

    def process_audio_frame(self, audio_frame):
        """Verarbeite Audio-Frame mit allen Schritten."""
        # 1. Noise Reduction
        denoised = self.denoise(audio_frame)

        # 2. VAD
        if not self.is_speech(denoised):
            return None

        # 3. Wake-Word Detection
        keyword_index = self.porcupine.process(denoised)

        return keyword_index
```

```
    def denoise(self, audio_frame):
        """Noise Reduction mit RNNoise."""
        # Konvertiere zu int16
        audio_int16 = (audio_frame * 32767).astype(np.int16)

        # Denoise
        denoised = self.denoiser.process_frame(audio_int16)

        # Zurück zu float32
        return denoised.astype(np.float32) / 32767

    def is_speech(self, audio_frame):
        """VAD mit Silero."""
        audio_tensor = torch.FloatTensor(audio_frame)
        speech_prob = self.vad_model(audio_tensor, self.sample_rate).item()
        return speech_prob > 0.5

    def run(self):
        """Hauptschleife."""
        print("🎤 Voice Assistant läuft (mit Advanced Audio Processing)...")

        with sd.InputStream(samplerate=self.sample_rate,
                            channels=1,
                            blocksize=self.frame_length,
                            dtype='float32') as stream:
            while True:
                audio_frame, _ = stream.read(self.frame_length)
                result = self.process_audio_frame(audio_frame.flatten())

                if result is not None and result >= 0:
                    print("✅ Wake-Word erkannt!")
                    # ... STT & Command Execution

# Main
if __name__ == "__main__":
    assistant = AdvancedVoiceAssistant()
    assistant.run()
```

Performance-Optimierung

CPU-Last reduzieren

Problem: Alle Audio-Processing-Schritte zusammen können CPU-intensiv sein.

Lösungen:

1. **Selective Processing:**

```
# Nur Noise Reduction wenn Rauschen hoch
if noise_level > threshold:
   audio = denoise(audio)
```

2. **Frame-Skipping:**

```
# VAD nur jedes 3. Frame
frame_counter += 1
if frame_counter % 3 == 0:
   is_speech = vad.check(audio)
```

3. **GPU-Acceleration (wenn verfügbar):**

```
# Silero VAD auf GPU
model = load_silero_vad()
if torch.cuda.is_available():
   model = model.cuda()
```

Code-Beispiele

Vollständiges Beispiel: Noise Reduction + VAD

```python
#!/usr/bin/env python3
"""
Beispiel: Noise Reduction + Advanced VAD
"""

import numpy as np
import sounddevice as sd
import noisereduce as nr
import torch
from silero_vad import load_silero_vad

# Parameter
SAMPLE_RATE = 16000
DURATION = 5  # Sekunden

# Lade VAD
vad_model = load_silero_vad()

# Aufnahme
print("🎤 Aufnahme startet (5 Sekunden)...")
audio = sd.rec(int(DURATION * SAMPLE_RATE),
               samplerate=SAMPLE_RATE,
               channels=1,
               dtype='float32')
sd.wait()
print("✅ Aufnahme beendet")

# Noise Reduction
print("🔧 Noise Reduction...")
audio_denoised = nr.reduce_noise(y=audio.flatten(),
                                 sr=SAMPLE_RATE,
                                 stationary=True)

# VAD
print("🔍 Voice Activity Detection...")
audio_tensor = torch.FloatTensor(audio_denoised)
speech_prob = vad_model(audio_tensor, SAMPLE_RATE).item()

print(f"Speech Probability: {speech_prob:.2%}")
```

```
if speech_prob > 0.5:
    print("✅ Sprache erkannt!")
else:
    print("❌ Keine Sprache erkannt")

# Speichern
import scipy.io.wavfile as wav
wav.write('original.wav', SAMPLE_RATE, audio)
wav.write('denoised.wav', SAMPLE_RATE, audio_denoised.astype(np.float32))
print("💾 Audio gespeichert")
```

Zusammenfassung

Empfohlene Konfiguration

Für Raspberry Pi / Low-Power:

- Noise Reduction: `noisereduce` (Offline) oder keine
- Echo Cancellation: Muting während TTS
- VAD: `webrtcvad` (aktuell)

Für Desktop / High-Performance:

- Noise Reduction: `RNNoise` (Echtzeit)
- Echo Cancellation: Adaptive Filter (AEC)
- VAD: `Silero VAD`

Für Produktion:

- Noise Reduction: `RNNoise`
- Echo Cancellation: Muting + AEC (Hybrid)
- VAD: `Pyannote Audio`

Nächste Schritte:

1. Teste `noisereduce` mit deinen Aufnahmen

2. Implementiere Muting während TTS
3. Evaluiere Silero VAD vs. webrtcvad
4. Messe Performance-Impact
5. Entscheide basierend auf Ergebnissen

Dokumentende
