Report

February 29, 2024

1 Sonic Saviors: Autoencoders for Audio Noise Reduction - Project Report

1.1 1. Introduction

Welcome to the thrilling world of audio noise reduction with autoencoders! This project, named "Sonic Saviors," delves into two innovative approaches to cleanse audio signals from unwanted noise:

- **Feature-Level Denoising:** This method extracts intricate features like Mel-Frequency Cepstral Coefficients (MFCC) and Mel Spectrogram from pristine audio. We then introduce noise, train an autoencoder to reconstruct the original features, effectively removing the noise from the signal.
- **Audio-Level Denoising:** In this approach, we directly manipulate the raw audio waveform by adding noise and subsequently extracting MFCC and Mel Spectrogram features. The autoencoder then works its magic to restore the clean audio waveform, employing these features.

1.2 2. Project Structure

- **README.md:** Your trusty guide through this auditory adventure.
- **data/:** Dive into this directory to uncover the audio data used for our experiments.
- **flickr_audio_eda.ipynb (Optional):** Embark on an auditory journey with this Jupyter Notebook containing exploratory data analysis (EDA) for audio data (if applicable).
- **noise_audio/:** This directory houses the arsenal of scripts dedicated to audio-level denoising:
  - **create_hybrid_file.py:** Craft hybrid files melding clean and noisy audio representations.
  - **denoisening_dae_audio.ipynb:** Unravel the mysteries of audio-level denoising with this Jupyter Notebook.
  - **processing_data.py (Optional):** Script to preprocess audio data for audio-level denoising (if needed).
- **noise_features/:** This directory houses scripts related to feature-level denoising:
  - **create_hybrid_file.py:** Generates hybrid files combining clean and noisy feature representations.
  - **denoisening_dae_features.ipynb:** Explore feature-level denoising techniques with this Jupyter Notebook.

  - **processing_data.py (Optional):** Script to preprocess audio data for feature-level denoising (if needed).
- **prediction.ipynb:** Witness the prowess of our trained autoencoder models in action through this Jupyter Notebook.

1.3 3. Autoencoder Architecture

Behold the architecture defined within the `create_autoencoder` function, serving as the backbone for both models.

[ ]:
```
def create_autoencoder(input_shape):
    input_layer = Input(shape=input_shape)  # (148, 109, 1)

    # Encoder
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Flatten()(x)

    # Latent space
    latent_space = Dense(128, activation='relu')(x)

    # Decoder
    x = Dense(37 * 28 * 64, activation='relu')(latent_space)
    x = Reshape((37, 28, 64))(x)
    x = Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    x = Cropping2D(cropping=((0, 0), (0, 1)))(x)
    x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    # Correction: Pad 'SAME' to avoid information loss during upsampling
    x = Cropping2D(cropping=((0, 0), (1, 0)))(x)
    output_layer = Conv2DTranspose(1, (3, 3), activation='sigmoid',␣
↪padding='same')(x)

    autoencoder = Model(inputs=input_layer, outputs=output_layer)
    autoencoder.compile(optimizer='adam', loss='mean_squared_error')

    return autoencoder
```

1.4 4. Preprocessing Parameters

- **Sampling rate (sr):** 22050 Hz
- **Fast Fourier Transform (FFT) window size (n_fft):** 2048
- **Hop length (hop_length):** 512
- **Number of Mel-frequency cepstral coefficients (n_mels):** 128
- **Number of MFCC coefficients (n_mfcc):** 20

- **Fixed length for feature vectors (fixed_length):** 55296

1.5 5. Noise Preprocessing Code (Audio-Level)

[ ]:
```
class AudioProcessor:
    """
    Class to process audio files and compute Mel spectrogram and MFCC features.
    """

    def __init__(self, sr=22050, n_fft=2048, hop_length=512, n_mels=128,␣
↪n_mfcc=20, fixed_length=55296):
        self.sr = sr
        self.n_fft = n_fft
        self.hop_length = hop_length
        self.n_mels = n_mels
        self.n_mfcc = n_mfcc
        self.fixed_length = fixed_length
        self.scal = StandardScaler()

    def compute_mel_mfcc(self, audio_path: str or np.array) -> Tuple[np.
↪ndarray, np.ndarray, np.ndarray]:
        """
        Compute Mel spectrogram and MFCC features for a given audio file.

        Parameters:
            audio_path (str): Path to the audio file or raw audio data as a␣
↪NumPy array.

        Returns:
            Tuple[np.ndarray, np.ndarray]: Tuple containing Mel spectrogram and␣
↪MFCC features.
        """

        if isinstance(audio_path, str):
            y, _ = librosa.load(audio_path, sr=self.sr)
            y = librosa.util.fix_length(y, size=self.fixed_length)
        else:
            y = audio_path

        mel = librosa.feature.melspectrogram(y=y, sr=self.sr, n_fft=self.n_fft,␣
↪hop_length=self.hop_length,
                                             n_mels=self.n_mels)
        mfcc = librosa.feature.mfcc(y=y, sr=self.sr, n_fft=self.n_fft,␣
↪hop_length=self.hop_length, n_mfcc=self.n_mfcc)
        mfcc = self.scal.fit_transform(mfcc)
        return mel, mfcc, y
```

```
    @staticmethod
    def add_noise(audio: np.ndarray, mean=0.1, std=0.07) -> np.ndarray:
        """
        Add Gaussian noise to audio data.

        Parameters:
            audio (np.ndarray): Input audio data.
            mean (float): Mean of the Gaussian noise. Default is 0.1.
            std (float): Standard deviation of the Gaussian noise. Default is 0.
 ↪07.

        Returns:
            np.ndarray: Noisy audio data.
        """
        audio_noisy = np.random.normal(mean, std, audio.shape)
        return audio_noisy


class RepresentationSaver:
    """
    Class to save hybrid representations to files.
    """

    def __init__(self):
        pass

    @staticmethod
    def save_hybrid_representations(audio_paths: list, clean_save_dir: str,␣
 ↪noisy_save_dir: str,
                                    processor: AudioProcessor):
        """
        Save hybrid representations (clean and noisy) to files.

        Parameters:
            audio_paths (list): List of paths to audio files.
            clean_save_dir (str): Directory to save clean representations.
            noisy_save_dir (str): Directory to save noisy representations.
            processor (AudioProcessor): Instance of AudioProcessor class.
        """
        for audio_path in tqdm(audio_paths, desc='Processing audio files'):
            mel_clean, mfcc_clean, y = processor.compute_mel_mfcc(audio_path)
            audio_noisy = processor.add_noise(y)
            mel_noisy, mfcc_noisy, _ = processor.compute_mel_mfcc(audio_noisy)

            filename = os.path.splitext(os.path.basename(audio_path))[0]
```

```
            np.save(os.path.join(clean_save_dir,␣
 ↪f'{filename}_hybrid_representation_clean.npy'),
                    np.concatenate((mel_clean, mfcc_clean), axis=0))

            np.save(os.path.join(noisy_save_dir,␣
 ↪f'{filename}_hybrid_representation_noisy.npy'),
                    np.concatenate((mel_noisy, mfcc_noisy), axis=0))


def main():
    # Paths
    audio_dir = 'path/to/audio/files'
    train_clean_representations_dir = 'path/to/save/clean/representations'
    train_noisy_representations_dir = 'path/to/save/noisy/representations'
    test_audio_dir = 'path/to/save/test/audio'

    # Create directories if they don't exist
    os.makedirs(train_clean_representations_dir, exist_ok=True)
    os.makedirs(train_noisy_representations_dir, exist_ok=True)
    os.makedirs(test_audio_dir, exist_ok=True)

    processor = AudioProcessor()
    saver = RepresentationSaver()

    # Split audio files into train and test sets
    audio_paths = [os.path.join(audio_dir, file) for file in os.
 ↪listdir(audio_dir) if file.endswith('.wav')]
    np.random.shuffle(audio_paths)  # Shuffle the list of audio paths
    num_train = int(len(audio_paths) * 0.9)  # 90% for training, 10% for testing
    train_paths = audio_paths[:num_train]
    test_paths = audio_paths[num_train:]

    # Process and save hybrid representations for training set
    saver.save_hybrid_representations(train_paths,␣
 ↪train_clean_representations_dir, train_noisy_representations_dir,
                                      processor)
    print('Hybrid representations for training set saved successfully!')

    # Copy test audio files to test directory
    for path in test_paths:
        filename = os.path.basename(path)
        dest_path = os.path.join(test_audio_dir, filename)
        shutil.copyfile(path, dest_path)
    print('Test audio files copied successfully!')

    # Save scaler
    joblib.dump(processor.scal, 'path/to/save/scaler.save')
```

```
    print('Scaler saved successfully!')


if __name__ == "__main__":
    main()
```

1.6 6. Noise Preprocessing Code (Feature-Level)

[ ]:
```
class AudioProcessor:
    """
    Class to process audio files and compute Mel spectrogram and MFCC features.
    """

    def __init__(self, sr=22050, n_fft=2048, hop_length=512, n_mels=128,␣
↪n_mfcc=20, fixed_length=55296):
        self.sr = sr
        self.n_fft = n_fft
        self.hop_length = hop_length
        self.n_mels = n_mels
        self.n_mfcc = n_mfcc
        self.fixed_length = fixed_length
        self.scal = StandardScaler()

    def compute_mel_mfcc(self, audio_path: str) -> Tuple[np.ndarray, np.
↪ndarray]:
        """
        Compute Mel spectrogram and MFCC features for a given audio file.

        Parameters:
            audio_path (str): Path to the audio file.

        Returns:
            Tuple[np.ndarray, np.ndarray]: Tuple containing Mel spectrogram and␣
↪MFCC features.
        """
        y, _ = librosa.load(audio_path, sr=self.sr)
        y = librosa.util.fix_length(y, size=self.fixed_length)
        mel = librosa.feature.melspectrogram(y=y, sr=self.sr, n_fft=self.n_fft,␣
↪hop_length=self.hop_length,
                                             n_mels=self.n_mels)
        mfcc = librosa.feature.mfcc(y=y, sr=self.sr, n_fft=self.n_fft,␣
↪hop_length=self.hop_length, n_mfcc=self.n_mfcc)
        mfcc = self.scal.fit_transform(mfcc)
        return mel, mfcc

    @staticmethod
```

```
    def add_noise(mel: np.ndarray, mfcc: np.ndarray, mean=0, std=0.05) ->␣
↪Tuple[np.ndarray, np.ndarray]:
        """
        Add Gaussian noise to Mel spectrogram and MFCC features.

        Parameters:
            mel (np.ndarray): Mel spectrogram.
            mfcc (np.ndarray): MFCC features.
            mean (float): Mean of the Gaussian noise. Default is 0.
            std (float): Standard deviation of the Gaussian noise. Default is 0.
↪05.

        Returns:
            Tuple[np.ndarray, np.ndarray]: Tuple containing noisy Mel␣
↪spectrogram and MFCC features.
        """
        mel_noisy = mel + np.random.normal(mean, std, mel.shape)
        mfcc_noisy = mfcc + np.random.normal(mean, std, mfcc.shape)
        return mel_noisy, mfcc_noisy


class RepresentationSaver:
    """
    Class to save hybrid representations to files.
    """

    def __init__(self):
        pass

    @staticmethod
    def save_hybrid_representations(audio_paths: list, clean_save_dir: str,␣
↪noisy_save_dir: str,
                                    processor: AudioProcessor):
        """
        Save hybrid representations (clean and noisy) to files.

        Parameters:
            audio_paths (list): List of paths to audio files.
            clean_save_dir (str): Directory to save clean representations.
            noisy_save_dir (str): Directory to save noisy representations.
            processor (AudioProcessor): Instance of AudioProcessor class.
        """
        for audio_path in tqdm(audio_paths, desc='Processing audio files'):
            mel_clean, mfcc_clean = processor.compute_mel_mfcc(audio_path)
            mel_noisy, mfcc_noisy = processor.add_noise(mel_clean, mfcc_clean)

            filename = os.path.splitext(os.path.basename(audio_path))[0]
```

```
            np.save(os.path.join(clean_save_dir,␣
 ↪f'{filename}_hybrid_representation_clean.npy'),
                    np.concatenate((mel_clean, mfcc_clean), axis=0))

            np.save(os.path.join(noisy_save_dir,␣
 ↪f'{filename}_hybrid_representation_noisy.npy'),
                    np.concatenate((mel_noisy, mfcc_noisy), axis=0))


def main():
    # Paths
    audio_dir = '/path/to/audio/files'
    train_clean_representations_dir = '/path/to/save/clean/representations'
    train_noisy_representations_dir = '/path/to/save/noisy/representations'
    test_audio_dir = '/path/to/save/test/audio'

    # Create directories if they don't exist
    os.makedirs(train_clean_representations_dir, exist_ok=True)
    os.makedirs(train_noisy_representations_dir, exist_ok=True)
    os.makedirs(test_audio_dir, exist_ok=True)

    processor = AudioProcessor()
    saver = RepresentationSaver()

    # Split audio files into train and test sets
    audio_paths = [os.path.join(audio_dir, file) for file in os.
 ↪listdir(audio_dir) if file.endswith('.wav')]
    np.random.shuffle(audio_paths)  # Shuffle the list of audio paths
    num_train = int(len(audio_paths) * 0.9)  # 90% for training, 10% for testing
    train_paths = audio_paths[:num_train]
    test_paths = audio_paths[num_train:]

    # Process and save hybrid representations for training set
    saver.save_hybrid_representations(train_paths,␣
 ↪train_clean_representations_dir, train_noisy_representations_dir,
                                      processor)
    print('Hybrid representations for training set saved successfully!')

    # Copy test audio files to test directory
    for path in test_paths:
        filename = os.path.basename(path)
        dest_path = os.path.join(test_audio_dir, filename)
        shutil.copyfile(path, dest_path)
    print('Test audio files copied successfully!')

    # Save scaler
```

```
    joblib.dump(processor.scal, '/path/to/save
```

1.7 7. Conclusions

This report has provided a comprehensive overview of the "Sonic Saviors" project, exploring the application of autoencoders for audio noise reduction. We've delved into the two approaches, showcased the project structure, and presented the core components like autoencoder architecture and preprocessing parameters. The provided noise preprocessing code snippets demonstrate the data preparation steps for both audio-level and feature-level denoising approaches.

1.8 8. Future Work

This project lays the foundation for further exploration. We can investigate:

- Hyperparameter tuning to optimize the performance of the autoencoders.
- Different autoencoder architectures to potentially achieve better noise reduction.
- Evaluation metrics beyond subjective listening tests to quantitatively assess the effectiveness of the denoised audio.

By continuing this research, we can further refine the capabilities of autoencoders in the realm of audio noise reduction, ultimately enhancing the listening experience for everyone.