Generation of Stereo Audio from a Mono Source Using a Nearest-Neighbor Algorithm

Ethan James
Virginia Tech
ethanjamesauto@vt.edu

1 Milestone Report

This project aims to create a trained model that transforms a mono (single-channel) audio track into a stereo (two-channel, left/right) audio track. This model implements the KNN (K-Nearest Neighbor) described in [1] and uses a relatively complex audio encoder and decoder to convert a stereo audio track into data that may be more easily learned by a model.

During the past month, I have completed the implementation of this encoder/decoder. Additionally, I have implemented a KNN and trained the model on a small portion of a music dataset. This was done to ensure all aspects of the model work properly - in other words, ensuring that the model doesn't output a garbled mess of audio. My progress is consistent with the tasks I outlined in my project proposal.

The source code for the implementation is available at `https://github.com/ethanjamesauto/mono-to-stereo`.

1.1 Parametric Stereo Encoder/Decoder

The model's objective is to take a mono audio track $s[n]$ - a time series of digital audio samples at a sample rate $F_s = 44100$ Hz - and generate two audio tracks $l[n]$ and $r[n]$, which also contain digital audio samples at the same sample rate. The model is trained on a dataset of stereo audio tracks, where the features are mono audio tracks created from those stereo audio tracks using the following transformation:

$$s = \frac{l + r}{2}$$

and the labels are the stereo tracks themselves. This makes finding a dataset simple - find a database of stereo audio tracks, convert them to mono, and train the model on the mono and stereo tracks.

Training a model to directly accomplish this task would be extremely difficult - the model would have to learn to generate two audio tracks from one. Instead, the stereo audio tracks are converted into a **parametric representation**, where a stereo track is represented as a mono track and additional data that encodes the stereo information. This way, the model's features are mono audio tracks and the model's label is the parametric stereo information. The stereo information along with the original mono source are then decoded into a stereo audio track. This stereo information is in the form of a spectrogram - a 2D array containing frequency information over time, generated using many short FFTs (Fast Fourier Transforms) over the audio track.

I implemented this encoder/decoder in Python using NumPy and some SciPy (for Short-Time FFT, filtering, etc). The encoder is implemented in an identical fashion to [1], and the decoder is implemented using [2]. This was done because [1] describes a KNN that uses the decoder described in [2], but [1] only describes the encoder and references [2] for the decoder. The encoder outputs parametric stereo parameters for a stereo track, which is used during training time to generate the labels for the model. The decoder is used at all times to convert the parametric stereo information (along with the mono source) into a stereo audio track.

Preprint. Under review.

1.2 K-Nearest Neighbor Algorithm

The KNN regressor outlined in [1] is a 1-nearest neighbor that takes in some frames of the **spectrogram** of the mono audio source and outputs the estimated corresponding parametric stereo data for the last frame. Note that the spectrogram is generated with window size $N = 4096$, overlapping windows with 75% overlap, and a Hann window applied to each frame. I implemented this KNN using scikit-learn's KNeighborsRegressor, and trained the model on a small dataset of stereo audio tracks from the FMA dataset [3].

1.2.1 Training

The model is trained by repeating the following procedure:

1. Randomly select a stereo audio track from the dataset
2. Pick a random $N = 20$ consecutive frames from the spectrogram of the audio track downmixed to mono
3. Place the frames in the KNN structure as a feature, and place the parametric stereo data from the last frame as the label

The resource [1] notes that this process was repeated 1 million times, but it makes no indication about the size of the tracks dataset or how many samples may be used by a single song. To test the model and ensure it's operability, I trained the model with 900 iterations over a dataset of 180 songs.

1.2.2 Prediction

The model is used to convert a mono track to stereo by first taking the spectrogram of the mono track with the aforementioned parameters ($N = 4096$, etc). For every frame in the spectrogram, 20 frames are taken and inputted into the KNN. The output of the KNN is the parametric stereo data for the last frame. This data is then decoded into a coherent stereo track.

1.2.3 Validation

I have not implemented a loss function or method for validating the model. As mono-to-stereo conversion is a subjective task, I did listen to the outputs of a few mono audio tracks that the model had not seen during training, and the outputs were coherent and sounded like stereo audio. Of course, it's important to note that a KNN does not require a loss function to train, as it simply memorizes the training data.

1.3 Next Steps

In the following weeks, I'll be working on the following tasks:

- Implementing a loss function to measure training vs. validation accuracy, and accuracy as the model is trained
- Training the model on a larger dataset of stereo audio tracks
- Speeding up some of the Python code. I've heavily optimized the encoder/decoder to only use vectorized operations instead of Python for-loops, but there are still some slow loops in the KNN implementation.

References

[1] MONO-TO-STEREO THROUGH PARAMETRIC STEREO GENERATION

`https://arxiv.org/pdf/2306.14647`

[2] Parametric Coding of Stereo Audio

`https://asp-eurasipjournals.springeropen.com/articles/10.1155/ASP.2005.1305`

[3] FMA: A Dataset For Music Analysis

`https://github.com/mdeff/fma`