Producing song embeddings for music recommendation using convolutional autoencoders

Caleb Clothier
Luke Benson

Deep Learning: Spring 2021

Abstract

Music recommendation is typically performed using collaborative filtering on user listening preference data. However, such techniques fail to recommend new or obscure songs due to a lack of relevant user listening data. In this paper, we seek to address this problem by training a convolutional autoencoder to learn lower dimensional representations of song snippets. These embeddings could be used to train other models, for example a genre-prediction model. We find that the autoencoder accurately reproduces the input audio data, albeit with some smoothing and a general inability to capture very fine musical features. Reconstruction quality on the test set is considerably worse, as expected given the relatively small size of the training data.

I. MOTIVATION & BACKGROUND

Music recommendation is at the forefront of modern-day music streaming and listening platforms. Spotify, for example, designs daily and weekly playlists for its users based on past streaming data, makes song recommendations for additions to user-curated playlists, and even develops entire "radio stations" based on a single song. Determining which data to use when making these recommendations, however, is often a difficult task.

Naive song recommendation systems primarily utilize collaborative filtering algorithms: a listener's predicted "rating" of a song they haven't heard is based only on the entire history of listener-song ratings. Through some kind of linear matrix factorization, collaborative filtering separates this listener-song matrix into a listener embedding matrix and a song embedding matrix. These two matrices leave us with a low-dimensional feature representation of each listener and each song. The predicted rating for song i by listener j is then defined by the inner product of the embedding for song i and the embedding of listener j. The problem these algorithms run into, however, is that they can never recommend new songs – if no one has listened to the song, there is no data on it. This is what is known as the cold start problem.

We are interested in designing a neural network which takes a step toward solving the cold start problem. Specifically, we seek to train a convolutional autoencoder to develop latent representations of songs based solely on raw audio data. These latent representations are essentially lower-dimensional embeddings of raw audio clips, and so obtaining them allows us to compare songs to each other without relying on any listener rating data. In future iterations of this project, the similarities between the latent representations of

songs could then be used to make recommendations to users based on their listening history.

II. PROBLEM FORMULATION

The audio representations used in this project are known as mel-spectrograms, arrays which contain decibel values for each frequency level over time. To retrieve this data, we interfaced with Spotify's API to download 15,000 30-second song previews and convert them into mel-spectrograms, with a frequency resolution of 128 bins and a temporal resolution of 1292 frames per 30 second preview.

The ultimate goal of this project is to develop useful embeddings – i.e. learned low-dimensional vector representations – of songs' mel-spectrograms. One of the most popular deep learning model architectures for obtaining low-dimensional embeddings in an unsupervised or semi-supervised manner is an autoencoder. This kind of model (1) inputs high-dimensional data, (2) works its way down toward a low-dimensional "informational bottleneck" layer, and then (3) outputs a high-dimensional structure which is the same size as the input. The network layers that take the input to the bottleneck layer are often referred to collectively as the "encoder," while the layers transforming the bottleneck layer into the output layer comprise the "decoder."

The autoencoder is trained to reconstruct the input: the more closely our output resembles the input, the lower the associated cost. This kind of training scheme compels the lower-dimensional bottleneck layer to retain as much information from the input as possible. Consequently, the resulting bottleneck layer values can often constitute a useful low-dimensional embedding space for our inputs.

In our project, we use a variation on a vanilla autoencoder called a 1-D convolutional autoencoder. This architecture is essentially a CNN for autoencoders: it convolves over a 2-D input to arrive at a 1-D bottleneck layer, and then reconstructs the 2-D input from the hidden bottleneck layer. In our case, we do not convolve over both dimensions of our input, but rather only over the axis of time, for the purpose of picking up on musical-temporal patterns.

Let's say we have the following elements in our convolutional autoencoder:

$$\begin{aligned} X &: \text{input audio data} \\ e(X) &: \text{encoder function} \\ Z = e(X) &: \text{embedding layer} \\ d(Z) &: \text{decoder function} \\ X' = d(Z) &: \text{output audio data} \end{aligned}$$

It follows that the optimal song embeddings minimizes the following expression:

$$\mathcal{L}(X) = ||X - d(e(X))||^2$$

III. METHOD & JUSTIFICATION

Next, we will delve into the specifics of our method. Our full autoencoder can be broken down into an encoder and decoder which perfectly mirror each other across the bottleneck layer. In the encoder, the input is first passed through a series of three 1-D convolutional layers and three 1-D max-pooling layers. Before arriving at our embedding space, we then pass the result of the final convolutional and max-pooling layer through three dense layers with ReLU activation.

Each 1-D convolutional layer hypothetically captures some sort of sonic patterns across our temporal dimension. All convolutional filters have a kernel size of 2. With this kernel size, the first two convolutional layers capture patterns within roughly a 0.4-second and 0.8-second window, respectively. The third layer, then, searches for sonic patterns within roughly a 1.6-second time frame. Given the wider range of time that each filter in the third convolutional layer considers, this third convolutional layer contains 512 filters, while the first two contain only 256.

In between each convolutional layer is a max-pooling layer with a kernel size of 2. These max-pooling layers serve the function of downsampling our convolutional outputs and, thus, adding some invariance with respect to time. We believed that the small kernel sizes of our convolutional and max-pooling layers in our network ($k = 2$) would lead it to recognize more of the audio clip's sonic profile – e.g. chords, chord progressions, rhythms – rather than capture general song attributes.

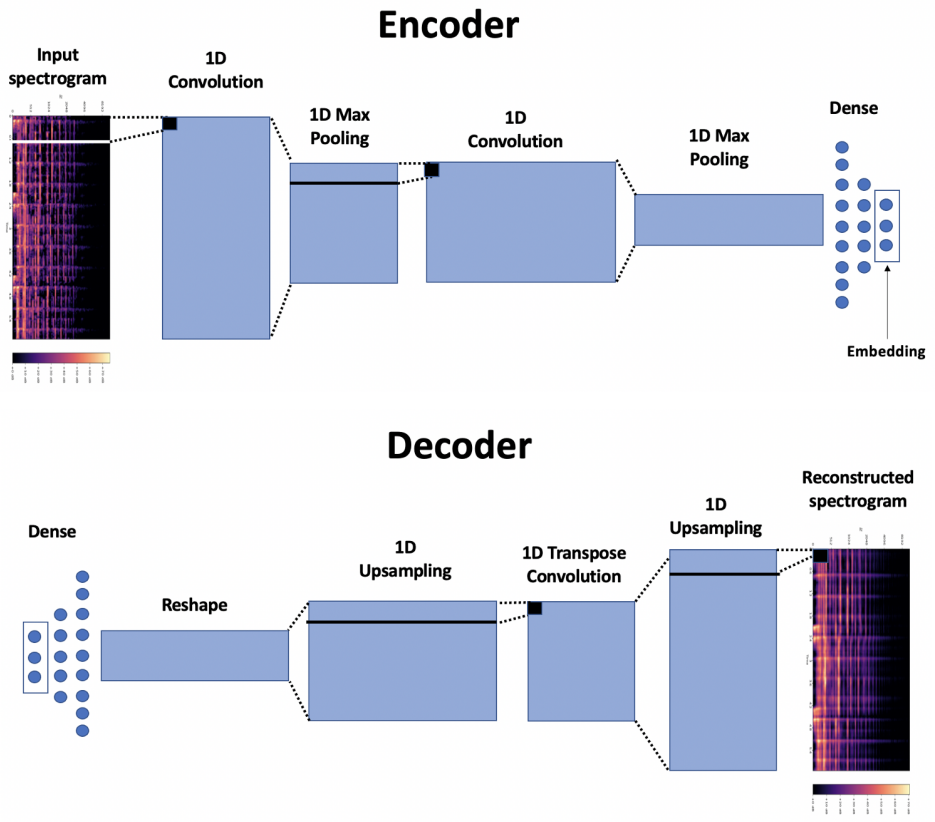


Figure 1: *Illustration of the general architecture of the convolutional autoencoder. Note that an additional convolutional (deconvolutional) layer and max-pooling (upsampling) layer were included in the final encoder and decoder architectures, not depicted here.*

While learning general features about each audio snippet could certainly be useful for song recommendation systems, it may pose too difficult of a task for the project at hand. Our goal, then, is not to try to understand the whole song, but to be able to pick up on its sonic attributes within small periods of time.

The final bottleneck layer is achieved by passing the results of third max-pooling layer through three dense layers of size 2048. The decoder then perfectly mirrors the encoder. We first pass the results of the bottleneck layer through three dense layers, and then a series of 1-D upsampling and 1-D convolutional transpose layers to arrive at a matrix which is the size of our original spectrogram.

The model was trained using mean squared error loss, an Adam optimizer, and a learning rate of 0.0002. Mean squared error is commonly used as a loss function for autoencoders, as it nicely describes the average distance between an input and its reconstruction. Adam (Adaptive Momentum Estimation) is widely seen as the best-performing adaptive optimizer for most cases, and is also known to be high-performing on sparse data sets. Given that our mel-spectrograms contain many 0-value cells, we suspected that Adam would be particularly effective for our model.

Before training, we also modified our mel-spectrograms. After loading the 15,000 mel-spectrograms using the Spotify API – an incredibly time-consuming process – each 30-second preview was divided into 4 segments lasting around 6 seconds, with dimension 256 (temporal) by 128 (frequency). The reasons for dividing the data in this way were threefold. First, as previously stated, we believed that the small convolutional kernel sizes and max-pooling sizes in our network would lead it to recognize more of the audio clip's sonic profile. Thus, sending an entire 30-second preview through our autoencoder may not have resulted in a very interpretable or useful embed-

ding. Secondly, obtaining embeddings for different parts of songs would allow us to compare embeddings with known existing relationships: how similar or dissimilar two embeddings from different parts of the same song are, for example, might give us some insight into the song structure. Finally, dividing the previews also gave us more training data by artificially expanding our data set size by a factor of four.

Before training, we split our data into 90% training data and 10% test data. We believed that with the amount of data we now had (60,000 total spectrograms after splitting), we could afford to utilize more data in training than the standard 80%-20% split.

IV. QUANTITATIVE & QUALITATIVE EVALUATION

The most obvious way in which we can evaluate our model performance is by calculating the loss on our training set over the course of many epochs. Evaluating our training loss over time will tell us if our model is actually learning anything about the structure of our spectrograms. As shown before, the loss is calculated by taking the mean squared error between the input spectrograms and our reconstructions:

$$\mathcal{L}(X) = ||X - d(e(X))||^2$$

Upon completion of training, we can then qualitatively assess the performance of our model by displaying our input and our reconstructions side-by-side. Given that the bottleneck layer in our model compresses the input by a factor of 128 (we go from a 128 by 256 matrix to a 256-length vector), we presume that the size of our latent space is small enough to prevent the autoencoder from learning something close to the identity function. Therefore, when analyzing the reconstructions of our spectrograms, we hope to find images which are very similar to our inputs.

After training, we can then evaluate our model performance on the test set. While neural networks generally avoid drastically overfitting, the test set performance will give us some insight into how well our autoencoder performs on data it has never seen before.

Another way in which we can evaluate our results is by visualizing our learned song embeddings. At the completion of training our model, we have 256-length vectors for each of our song snippets. We can then use PCA to find the top two principle components of these embeddings, and plot the embeddings in this two-dimensional space. While using the top two principle components from PCA only compresses our song data even further, visualizing our embeddings in this way could be useful. We may, for example, find that song snippets from the same song lie in similar regions of this space, or that snippets of songs which are similar in some way – either through qualitative or quantitative assessment – lie near each other in this space.

When determining if songs near each other in this space are "similar," we can quantitatively assess the similarity between two songs by utilizing a version of the collaborative filtering method discussed earlier in this paper. Using the implicit package, we implement the alternating least squares algorithm for collaborative filtering on a matrix of user-song ratings. The result of this collaborative filtering algorithm is two matrices: a "user factors" matrix (with dimensions number of unique users by latent space size), and a "item factors" matrix (with dimensions latent space size by number of unique songs). We can then compute the inner product between the latent space song representations to find songs which are most similar to each other in terms of user preferences.

Finally, we could utilize our length-2 PCA compressions of each embedding quantitatively to find the cosine similarities between each pair of embeddings. Cosine similarity computes how close two vectors are by computing the L2-normalized dot product: for two vectors, x and y, the cosine similarity is computed as

$$\frac{xy^T}{||x|| * ||y||}$$

Like our PCA visualization, we may expect that more similar songs would have high-magnitude positive cosine similarity coefficients.

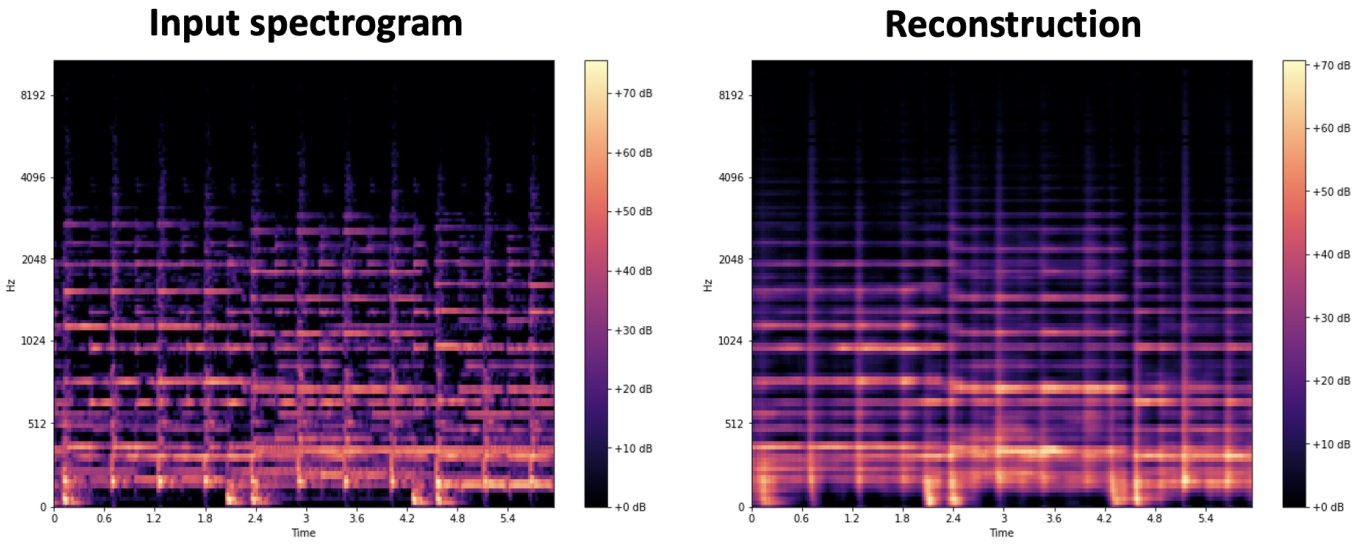


Figure 2: *An example input spectrogram and its reconstruction, as generated by the trained convolutional autoencoder. Though some smaller scale features appear to be missing in the reconstruction, rhythmic and chordal structures are well-preserved.*

V. RESULTS & ANALYSIS

Due to limitations in RAM and computing power, even when using Google Colab's High-RAM GPU clusters, we were unable to incorporate all 15,000 spectrograms into the training of our model. Instead, we used 10,000 30-second song previews, yielding 36,000 5-second audio snippets for training, and 4,000 for testing.

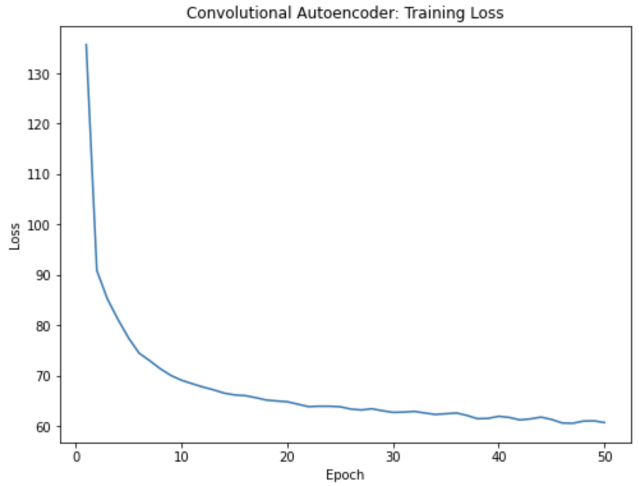


Figure 3: *Plot of the training loss for our final model, trained for 50 epochs on 36,000 5-second snippets from 9,000 different songs.*

The training loss for our final model is shown in Figure 3. For an embedding size of 256, the minimum achieved training MSE loss was around 60; for comparison, a randomly initialized CAE achieves a training MSE loss equal to around 500. Performance on the testing set was worse, as expected, with a final MSE loss of around 100. Increasing the embedding size by a factor of two was found to decrease the minimum achieved training loss to around 45, but at the expense of the test set loss, suggesting a smaller embedding size is more conducive to generalization in this case.

Next examining the example input spectrogram and CAE-generated reconstruction shown in Figure 2, it is evident that the autoencoder is able to capture most high-level musical features, including rhythmic and chordal structures. In general, the reconstructions of the training spectrograms were remarkably high fidelity. Songs with quicker tempos and more complex melodies proved to be the most difficult for the CAE to accurately reconstruct. Possible ways to remedy this include adding more convolutional layers with smaller kernel sizes.

Given the small number of songs that we had available, testing for meaningful relationships between the song embeddings proved to be a difficult task. First, we evaluated the cosine similarity between different snippets of the same song, to check whether the autoencoder was capable

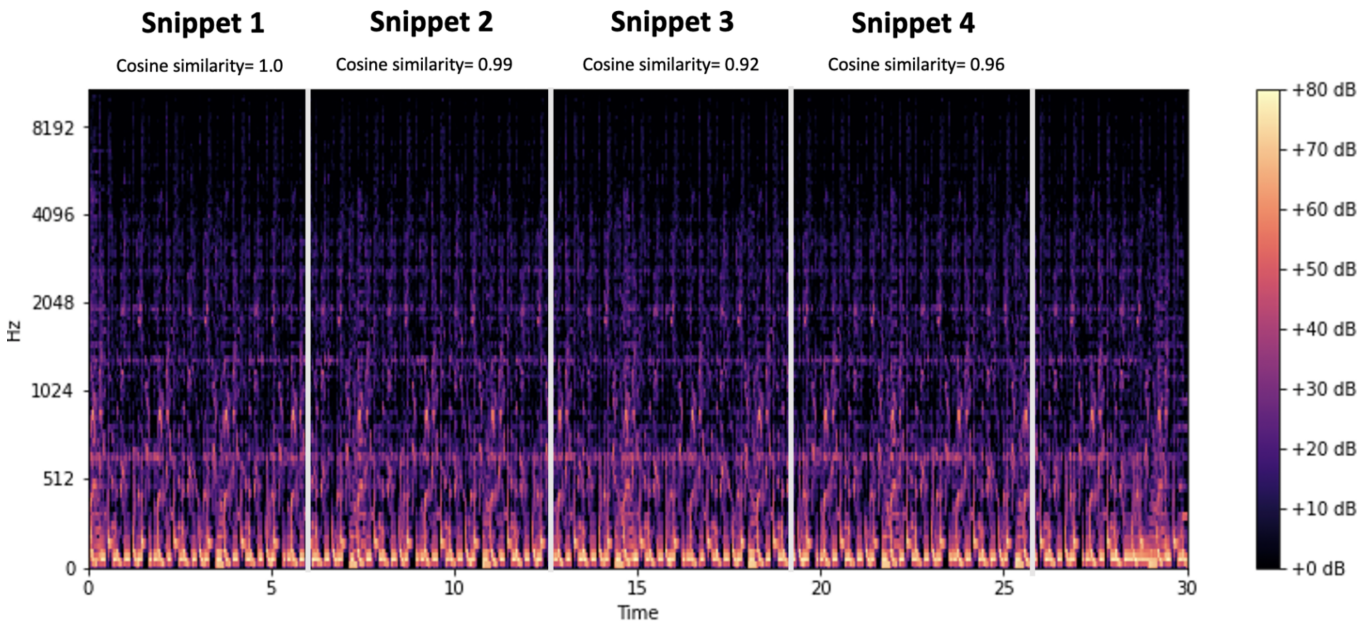


Figure 4: *Example of a spectrogram that has high cosine similarity between its constituent snippets. The cosine similarity of each snippet is calculated with respect to the first snippet.*

of "recognizing" that the snippets were related. Though for some songs the snippets varied significantly in musical content, and thus achieved low scores on mutual cosine similarity, songs with a sufficient degree of repetitiveness typically scored high on mutual cosine similarity, as can be seen in Figure 4.

Our most fruitful analyses, however, came from performing PCA on the song embeddings to find the top two principal components, and then plotting these in 2-D space. Though most songs in the dataset were relatively obscure, we were able to select two pairs of songs by well-known artists (Mariah Carey + Celine Dion, and Daft Punk + LCD Soundsystem) and locate them in the 2-D principal component space, as shown in figures (a) and (b) in the Appendix. Given the large sonic disparity between the "crooning" of Mariah Carey and Celine Dion, on one hand, and the throbbing electronic sounds of Daft Punk and LCD Soundsystem, on the other, it makes sense that we significant separation between two pairs in the latent space. Furthermore, each pair is fairly well localized in the latent space, suggesting that the embeddings learned by the autoencoder are capable of capturing genre-specific patterns.

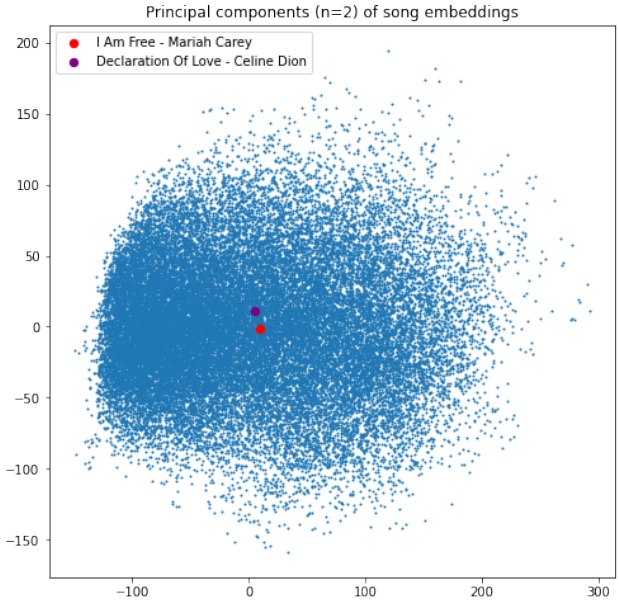
VI. CONCLUSION

In sum, we have trained a convolutional autoencoder to learn song embeddings from mel-spectrogram data. Analysis of the resulting embeddings suggests that these embeddings capture some higher level musical features. In particular, comparing the first two principal components of these embeddings yields apparently meaningful musical differentiation across genre and style, although only for sufficiently simple or repetitive songs. Future work might involve training a genre classification neural network using the song embeddings, rather than raw audio data, as input, thus indirectly evaluating the descriptive power of the learned embeddings.

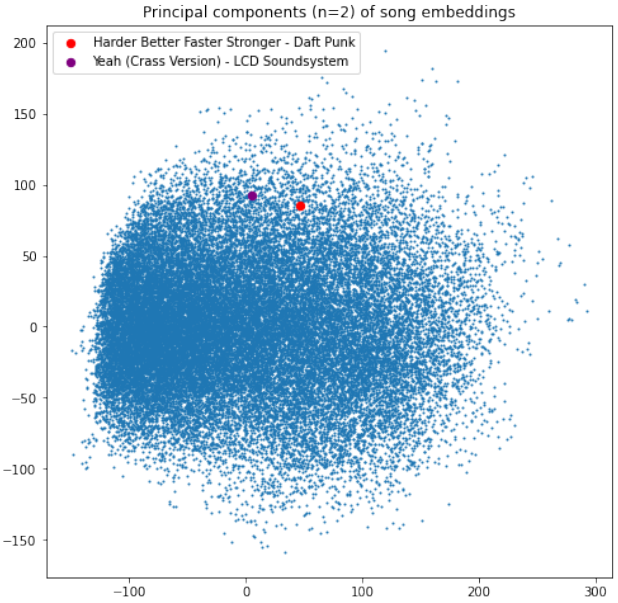
GITHUB

https://github.com/calebclothier/Musical-Embedding-CAE

APPENDIX



(a) *The blue points plot the first two principal components of the embedding for every song in the training data, as determined using PCA. The red and purple points denote two songs determined to be similar using collaborative filtering.*



(b) *Comparing the embedding locations of two songs by electronic artists in this figure, we see that they are spatially close and localized in a different area then the songs in figure (a) – this is expected, given the dramatic difference in genre.*