Eleftheria Bargiota - AI/ML Research Engineer

Orfium Assessment

February 2025

1 Cover Song Similarity - Comprehensive Description

The main goal of this project is to design a similarity metric between cover related songs, given a dataset that consists of cover song cliques. My main suggestion in the problem was to train a model on labeled pairs (cover related and non-cover related) using a contrastive loss function to minimize the distance between similar pairs and maximize the distance between dissimilar pairs. The method proposed utilizes Dimensionality Reduction, a powerful method that aims to translate high dimensional data, as the ones in the given Da-Tacos dataset, to a low dimensional representation such that similar input objects are mapped to nearby points, and dissimilar inputs are mapped in more distant points.

Low Dimensional Mapping: The problem is to find a function that maps high dimensional input patterns to lower dimensional outputs, given neighborhood relationships between samples in input space. More precisely, given a set of input vectors $I = X_1, ..., X_n$ find a parametric function G_w such that it has the following properties:

1. Simple distance measures in the output space (such as euclidean distance) should approximate the neighborhood relationships in the input space.

2. The mapping should not be constrained to implementing simple distance measures in the input space and should be able to learn invariances to complex transformations.

3. It should be faithful even for samples whose neighborhood relationships are unknown.

Constrastive Loss Function A contrastive loss function is employed to learn the parameters W of a parameterized function G_w, in such a way that "neighbor" songs (cover songs) are pulled together and "non-neighbor" songs(non-cover) are pushed apart. This loss function runs over pairs of songs. Let X_1, X_2 be a pair of vectors that depict the two input songs, and Y is a binary label assigned to this pair (Y=0 for cover songs, Y=1 for non-cover songs). The final similarity metric is defined as the Euclidean distance D_w between the ouputs of a learned mapping function G_w which is parameterized by W. This distance is calculated as follows:

$$D_w(X_1, X_2) = ||G_w(X_1) - G_w(X_2)||_2$$

The contrastive loss function in a general form is :

$$L(W, Y, X_1, X_2) = (1 - Y) * \frac{1}{2} * D_w^2 + Y * \frac{1}{2} max(0, m - D_w)$$

where m is a margin. Dissimilar pairs contribute to the loss function only if their distance is within the radius m.

1.1 Structure of the implementation

The Da-tacos dataset consists of several subfolders. For this implementation, I used the subfolder "da-tacos-coveranalysis-subset-single-files". This subset consists of 5000 cliques (subfolders), each one consisting of two cover-related songs. I chose this part of the dataset because its structure is convenient for easily aligning labels to cover-related songs and randomly generating pairs of non-cover songs.

1.1.1 Dataset

Building the class DATACOSDataset(Dataset): The class reads pairs of songs stored in .h5 files and prepares them for use in the model. The songs in each pair are associated with a label indicating whether they are a "cover song" pair (label=0) or a "non-cover song" pair (label=1).
Functions:

1. *init(self, data-dir, transform):*

   Input: The dataset expects a directory (data-dir) with subfolders containing pairs of .h5 files.

   Output: For each access to an index in the dataset, it returns a pair of songs (represented as tensors) and a label indicating whether they are cover-related (0) or non-cover (1).

   Transforms: An optional transform argument can be applied to the song data before returning it. The transforms will be analyzed later.

   Details:

   - **Cover-related pairs:** For each subfolder, it retrieves the two .h5 files and pairs them together, labeling them as cover-related (label=0).
   - **Non-cover pairs:** It then creates non-cover pairs. This is done by randomly selecting two different songs from any subfolder, having prior knowledge of the number of cover-related pairs (5000) that were created. These pairs are labeled 1 to indicate they are non-related.

Result: The pairs list stores tuples of the form (song1-path, song2-path, label), representing the two songs in the pair and their label.

2. *load-h5(self, file-path):*

   This method loads a specific .h5 file and extracts the relevant data. It assumes the .h5 file contains musical features, and it retrieves a specific feature (in this case, chroma-cens).

   Feature Extraction: The class reads all the features in each .h5 file such as chroma-cens-data, crema-data, hpcp-data, key-extractor-data, madmom-features-data, mfcc-htk-data, and tags-data. For each feature, the function converts it to a PyTorch tensor. It also ensures that the tensors are in the correct shape by adding a dimension if necessary, to reshape 1D tensors into 2D.

   Output: A tensor representing the feature data from the .h5 file.

Training/Validation Set Knowing our dataset from the DATACOSDataset class now consists of 5000 0-labled pairs (covers) and 5000 1-labeled pairs (non-cover) we create a training dataset consisting of the first 4000 0-labeled pairs and the first 4000 1-labaled pairs and a validation dataset consisting of the last 1000 0-labeled songs and the last 1000 1-labeled songs. Then we create our DataLoader objects trainloader and validloader that will be used in training.

1.1.2 Model Architecture

The architecture used for this task is called Siamese network and consists of two identical Convolution Neural Networks (sharing weights) that take two inputs, encode them and then the output features are compared. This comparison can be done in number of ways, but in this project we chose constrastive loss. The idea is that the two branches of the network should learn to encode the inputs in a way that allows them to compute the similarity between them.

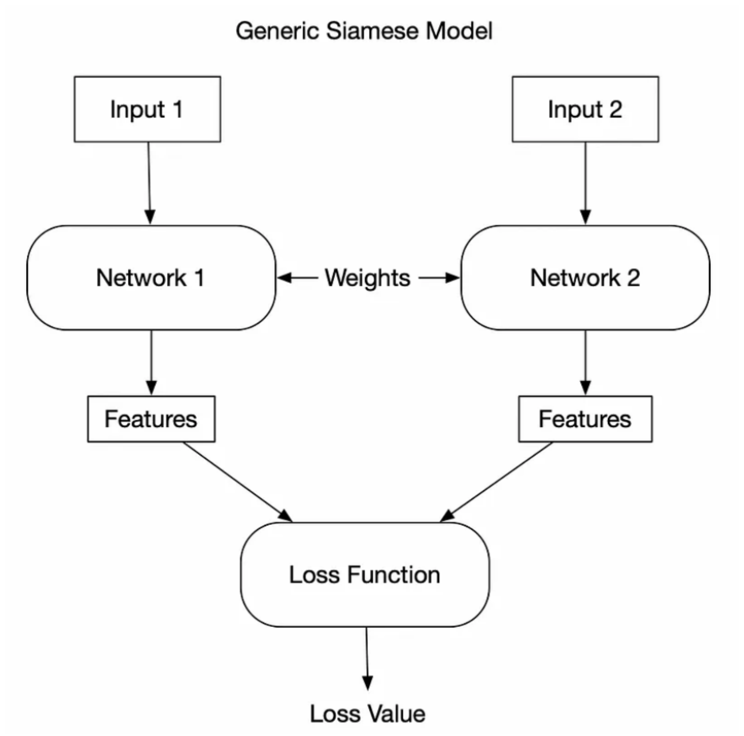

Figure 1: Siamese Network.

In this project the network consists of two copies of the function G_W (the learned mapping function) which share the same set of parameters W, and a cost module. The input to the entire system is the pair of songs (X_1, X_2) and a label Y. The songs are passed through the functions, giving two outputs $G(X_1)$ and $G(X_2)$. The cost module then generates the distance $D_w(G_w(X_1), G_w(X_2))$, which represents the similarity metric. The loss function combines D_w with label Y to produce the loss, depending on the label Y. The parameter W is updated using stochastic gradient. The gradients can be computed by back-propagation through the loss, the cost, and the two instances of G_w. The total gradient is the sum of the contributions from the two instances.

1.1.3 Code implementation

The code was implemented in Python using resources from Google Colab Pro+. Google Colab offers 83.5GB RAM, 40 GB GPU VRAM and 235 GB of disk space. The Google colab notebook is attached in the github link provided.

GithubLink

1.2 Discussion

The main issue that arises in this project is the complexity and multi-dimensionality of the dataset. Building a class for such a complex dataset can be challenging and extremely resource-consuming. Every `.h5` file contains various features in the form of arrays, strings, and single values that must be combined into one tensor, which eventually becomes extremely large.

In the example below, we can see the outcome of a random pair in the dataset, where each song tensor has very high shape values:

Pair 9501:
Song 1 Data: `torch.Size([16599, 33178])`
Song 2 Data: `torch.Size([19630, 39239])`
Label: 1.0

The shapes we are encountering `([16599, 33178])` and `([19630, 39239])` suggest that each song data tensor is very large (tens of thousands of rows and columns), meaning that each sample occupies a large amount of memory.

Printing the RAM usage for only one pair, `([20890, 41760]`, `[12453, 24887])` gives the following outcome:

Memory usage of song1-data: 9.848953329026699 GB
Memory usage of song2-data: 2.1034150011837482 GB

There are several suggestions for handling this large memory usage such as downsampling the data, reduce precision (from float64 to float32), using efficient data formats or other batching and streming techniques of the data that could prevent insufficient memory.

1.2.1 Transforms

Preprocessing the data seems to be necessary in this project as we are dealing with extremely large data tensors in the ouput of the Datacos class. This is why we created a class that can utilize an independent transform parameter to minimize the complexity of the data before being returned in the class.

```
def __init__(self, data_dir, transform=None)
      self.transform = transform

def __getitem__(self, idx):
    if self.transform:
        song1_data = self.transform(song1_data)
        song2_data = self.transform(song2_data)
```

The transform parameter is used to apply a transformation to the song data (i.e., the .h5-loaded features like chroma-cens). This is an optional argument and allows you to preprocess or augment the data before it is returned from the dataset. If transform is not None, the data from both songs will go through transform(song1-data) and transform(song2-data) before being returned. This could involve things like scaling the feature values or applying a neural network-based preprocessing step.

Potential Transforms

1. Fourier Transform *(FFT-based Downsampling)*: The Fast Fourier Transform (FFT) is a widely used method for transforming a signal or tensor from the spatial (or time) domain to the frequency domain. Once the data is transformed into frequency space, you can selectively retain or discard certain frequencies based on their importance in the given task.

2. PCA *(Principal Component Analysis)* is a popular technique that finds the most important frequency components (directions of maximum variance) in a dataset and projects the data into a lower-dimensional space along those components. PCA can help reduce the number of frequency components by projecting the tensor onto a lower-dimensional subspace. This project could benefit importantly from the use of PCA in the preprocessing step, reducing the number of features and removing noisy or redundant features, potentially improving model performance.