Music visualization with end-to-end learning

Bruno Godefroy, godefroy@kth.se

May 31, 2017

Source code: `https://github.com/BGodefroyFR/Deep-Audio-Visualization`
Live demo: `https://bgodefroyfr.github.io/Deep-Audio-Visualization/web-app`

1 Abstract

Music visualization is nowadays very commonly used in media player softwares, such as *Winamp* or *Itunes*. These systems rely on the extraction of features from the raw audio, such as spectrograms, pitch or timbre. Then, preprogrammed models, called "presets", define how animations behave, depending on these features [1].
The features commonly used are usually hand-crafted and depend on some historical background. In recent years, there has been increasing interest in using features learning and deep architectures instead, thus reducing the required engineering effort and the need for prior knowledge.
Since autoencoders provide the best possible lower dimensional representation of some input data, this could obviously present a great interest for visualization as well. In this project, we propose to experiment some music visualization based on features extracted with a stacked autoencoder neural network.

2 Background and related work

Music visualization probably started with abstract film-making techniques, in the beginning of the 20^{th} century. Many artists, like Oskar Fischinger (1900-1967), started developing abstract musical animation which "*was composed of a series of geometric shapes and synchronized music*" (CVM and Moritz, 2006-13). Fischinger saw his experiments as a new art which he named *Raumlichtmusik* (space light music). In the late 1940s, Fischinger also invented the *Lumigraph*, a mechanical instrument which was performed to create imagery in public performances.

More recently, Ciuha et al. [2] have focused on visualizing harmonic relationships between tones and colours. Based on a mathematical model, their system uses color hue and saturation to help the user perceive tones, keys, dissonance and consonance.

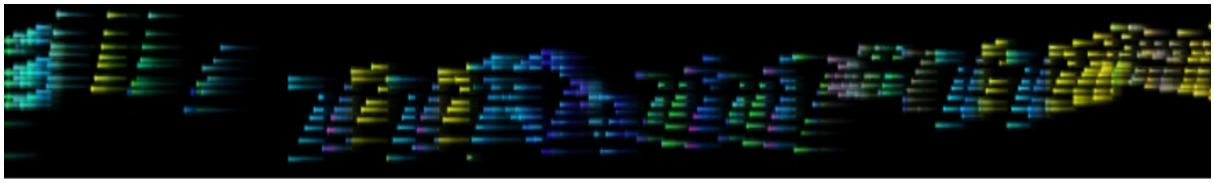

Figure 1: Ciuha et al. [2]. Excerpt from Debussy's Clair de Lune

In the same spirit, Dixon et al. [3] have implemented a real-time animation of a worm, based on music tempo and loundness. The system is described as an intuitive view of a number of high-level aspects of expressive music performance, which makes it a very useful tool for musical analysis.

Regarding *end-to-end learning* for feature extraction, S. Sigtia and S. Dixon [4] have demonstrated, throughout their work, the efficiency of deep learning for sound analysis and compared various network architectures. More generally, the literature provides many successful applications to feature learning,

such as music classification and music auto-tagging [6, 7, 8]. Their authors don't always overcome previous techniques, but demonstrate a great potential for future researches and applications.

3 Method

The system is composed of two main phases: feature extraction and real-time visualization. The first one consists in training an autoencoder neural network and then inferring it to automatically extract features from audio data. Then, some animation is generated from these features, in a Web browser, in real time.

3.1 Features extraction

For this task, contrary to a majority of music visualization systems, like [2, 3], we don't use hand-crafted features. Instead, following the method from S. Dieleman and B. Schrauwen [5], those are directly inferred from the data, thanks to a stacked autoencoder.
To do so, I have first created a training and a test dataset. Those contain spectrograms data, computed using STFT (Short Time Fourier Transform), a Fourier transform computation over a short range of time. This data are stored in LMDB files (Lightning Memory-Mapped Database), in order to efficiently feed the neural network, during training.
The dimension of the input spectrograms depends on two parameters: audio sampling rate and window size. The audio sampling rate represents the number of frames per second (usually 44.1 kHz or 48 kHz) and the window size, the time period over which spectrograms are computed. The choice of these parameters is critical and I spent a lot of time to fine-tune them. Indeed, both parameters should have reasonably low values in order to make learning computationally tractable. Finally I have reduced the sampling rate to 4.096 kHz (which affects a lot the audio quality) and the window size has been set to 1 second.
For the audio data, I used random songs downloaded from *YouTube*, using *Youtube-dl* [11]. This was very useful to guarantee a wide range of music genres in training data. Then, some Python scripts handle spectrograms extraction; using libraries, this is pretty straight-forward. This way, I made a training set containing about 70,000 spectrograms ($\sim$ 350 songs) and a test set with about 12,000 spectrograms ($\sim$ 60 songs).

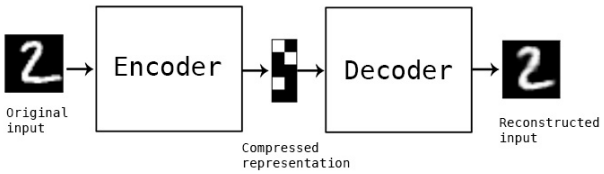

Figure 2: The autoencoder framework

Deep architecture training is extremely demanding in computational power and memory, which makes it often intractable with CPUs. Therefore, I have used a g2.2xlarge instance from *AWS*, which provided me powerful GPUs for an efficient training (a few minutes). For the project, I have used a 10 layers (4,096 * 2,048 * 1,024 * 512 * 64 * 10 * 64 * 512 * 1,024 * 2,048 * 4,096, 22 million parameters to learn!) autoencoder, with fully-connected layers and sigmoid activation function, created with the library *Caffe*.

Once the model trained, the features extraction can be done easily. Given a song we want to visualize, we extract its spectrograms and run a forward pass in the neural network for each of them, collecting the middle layer output. That is, for each input spectrogram (of size $window\_size * sample\_rate = 4,096$), the system generates a lower dimensional representation of the data (compression with loss) of size 10. Finally, for a given song, we have a set of 10-dimensional features in JSON format, which could be used later for visualization.

3.2 Real-time visualization

To make the project more interactive, I have chosen to implement visualization in Web browsers, with *Three.js*, a JavaScript library which uses *WebGL*. Since it is very powerful and well-documented, implementing the animation in browsers didn't really increase the project difficulty.

The more challenging feature of the Web client was probably the synchronization of music and animation. Indeed, to produce its expected effect, sound and image need to be very precisely synchronized. In practice, it is almost impossible to control the audio stream ; this is handled natively by browsers. Therefore, the animation advancement needs to rely on the audio stream; at every tick, we check the audio advancement and compute the animation based on it. This system enables to reliably manage FPS variations or interruptions in audio stream and animation.
Thanks to the documentation and examples provided by *Three.js*, the implementation of the animations themselves was pretty straight-forward. The main focus was about building a general framework so that new animations could be added easily to the project, like *presets*. That is, each animations are encapsulated into a class ("pseudo-class" from JavaScript) and follow a common template, consisting of two methods which could be called by the Web application itself:

- *init()*: sets the scene, the camera and creates animation environment.
- *update (timeDelta, parameters)*: updates the animation, given time elapsed since last update and the animation parameters for the current frame (features extracted from the audio data).

In addition, a JSON file is created for each animation, defining how audio features should be mapped to the input parameters of the animation.

Listing 1: Parameters file for the particle system animation

```
{
  "parameters": [
    { "name": "velocityRandomness", "min": 0.0, "max": 3.0, "step": 0.5, "FPS": 2.0 },
    { "name": "positionRandomness", "min": 0.0, "max": 3.0, "step": 0.5, "FPS": 3.0 },
    { "name": "size", "min": 1.0, "max": 20.0, "step": 5.0, "FPS": 5.0 },
    { "name": "sizeRandomness", "min": 0.0, "max": 25.0, "step": 5.0, "FPS": 0.1 },
    { "name": "colorRandomness", "min": 0.0, "max": 1.0, "step": 0.3, "FPS": 0.1 },
    { "name": "velocityRandomness", "min": 0.0, "max": 3.0, "step": 0.8, "FPS": 3.0 },
    { "name": "lifetime", "min": 0.1, "max": 4.0, "step": 0.2, "FPS": 3.0 },
    { "name": "turbulence", "min": 0.0, "max": 1.0, "step": 0.1, "FPS": 1.0 },
    { "name": "spawnRate", "min": 10.0, "max": 1000.0, "step": 100, "FPS": 3.0 },
    { "name": "timeScale", "min": 0.0, "max": 1.0, "step": 0.3, "FPS": 1.0 }
  ]
}
```

For each animation parameter, its mapping with the underlying features is described with:

- *name*: identifier of the parameter.
- *min*, *max*, *step*: how the feature should be mapped to fit the parameter. *min* and *max* are the bounds for the parameter and *step* represents its resolution. Considering the above parameters file, the feature *velocityRandomness*, for instance, should be mapped into the set of values {*0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0*}.
- *FPS*: the maximum update rate of the parameter. This is useful for parameters, such as *colorRandomness*, which should not be updated too frequently for a good visual effect (at most once every 10 seconds here).

4 Results and discussion

At the time of writing, the application is more a proof-of-concept prototype than a finished product. Only basic features are implemented but it clearly demonstrates that the method works. Indeed, the animation reacts well to rhythms and harmonic patterns, sometimes in very surprising ways. Since music visualization is very intuitive and subjective, evaluating the quality of the system without collecting

people feedback seems almost impossible though.
For simplicity, the developed Web application is only front-end. That is, features are pre-computed and the user could only play with provided sample songs. Also, there are only a few animation and those could be improved a lot.
Considering features extraction, many hyper-parameters have not been fine-tuned (window size, sampling rate, network architecture...) due to time constraints, but this would probably improve results as well.

References

[1] Robyn Taylor, Pierre Boulanger, and Daniel Torres. *Real-time Music Visualizations using Responsive Imagery.*

[2] P. Ciuha, B. Klemenc and F. Solina *Visualization of Concurrent Tones in Music with Colours.* Univ. of Ljubljana, Slovenia.

[3] S. Dixon, W. Goebl, and G. Widmer. *The performance worm: Real time visualisation based on langner's representation.* In M. Nordahl, editor, Proceedings of the 2002 International Computer Music Conference, pages 361–364, San Francisco, CA, 2002. International Computer Music Association.

[4] S. Sigtia and S. Dixon. *Improved music feature learning with deep neural networks.* In Proceedings of the 38th International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2014.

[5] Sander Dieleman and Benjamin Schrauwen. *End-to-end learning for music audio.* Conference Paper in Proceedings - ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing, May 2014.

[6] J. Nam, J. Herrera and K. Lee. *A Deep Bag-of-Features Model for Music Auto-Tagging.* Eprint arXiv:1508.04999. 2015.

[7] J. Schlüter. *Unsupervised Audio Feature Extraction for Music Similarity Estimation.* Technische Universität München, Fakultät für Informatik.

[8] E.J. Humphrey, J.P. Bello, Y. LeCun. *Feature Learning and Deep Architectures: New Directions for Music Informatics.* Journal of Intelligent Information Systems 41 (3), 461-481.

[9] Felix Turner. *Loop Waveform Visualizer.* https://airtightinteractive.com/demos/js/reactive.

[10] Annaliese Micallef Grima. *Visual music: development of an art.* 2015.

[11] *Youtube-dl.* https://rg3.github.io/youtube-dl.