Regression for high-level timbral features - Report

# Introduction

I am interested in Generative neural networks architecture models. Specifically, I am interested in Neural Audio Synthesis Architectures that expose continuous and high-level Synthesis control parameters. In other words, to truly be Synthesizers, these kind of Generative models should expose controls which, representing some synthesis parameters, are both perceptually meaningful to humans (expressing latent spaces representations which are as disentangled as possible) and numerical, as opposed to nominal (continuous parametric values rather than class names).

At the light of this, I decided to perform a supervised regression task, rather than classification (my model can be, after training, used as Encoder in an Autoencoder Synthesizer).

# Dataset

I used the *Freesound one-shot percussive sounds* dataset (Ramires, 2020), created as part of the PhD thesis project (Ramires, 2023) by the same author, which is composed by 10254 one shot electronic percussive sounds (like kick, snare, ecc.) sampled at 16 kHZ, annotated with 7 continuous-valued synthesis parameters (*depth, brightness, roughness, warmth, sharpness,* and *boominess*) as well as other spectral features, which have been ignored since not relevant for the scope of this Project. The annotations, a set of high-level timbral descriptors which is derived from the most common adjectives used to describe sounds in Freesound, were automatically computed with the AudioCommons extractor (Font, 2019). To some extent, my Project can thus be seen as a Neural re-implementation of the AudioCommons extractor only for the 7 synthesis parameters descriptors.

In the notebook ‘Regression for high-level timbral features - Data pre-processing.ipynb’, I implemented some preprocessing functionalities by creating a .csv file and pandas.DataFrame with input examples audio samples and corresponding annotations/ground truth -in order to parse the whole dataset only once, and to test the validity of the data I also created some dictionaries data structures, which I then serialized in pickle files for later access, in order to have constant low complexity-time access data structures mappings between data groupings (e.g. file name <-> file path, ecc.)-.

# Algorithms/methods

The Software framework used for creating the neural network model(s), visible in the notebooks named ‘Regression for high-level timbral features - Keras NN 7 filters model.ipynb’ and similar, is Keras, a higher level API built on top of Tensorflow. When I first tried to feed my pandas.DataFrame or .csv file dataset into the model, I encountered many problems of tensor shaping (it was difficult to give my tensor the right number of input dimensions); to fix this issue I used a tensorflow.data.Dataset object with a generator, directly out of the mir data library ‘freesound\_one\_shot\_percussive\_sounds’ dataset loader. Inside the generator, I used my pandas.DataFrame only for computing the minimum and maximum values over all training examples for each annotation, information needed in order to normalize the annotations values between 0. and 1.. Since the audio samples’ durations were shorter than 1 second but often not equal to each other, I have 0 padded the entire dataset, so that all audio files matched 1600 samples (1 second at 16 kHZ), as this fixed-size also determined the number of nodes in the input layer of my Neural model. The train/test split is 80/20 %, and train/validation split has also the same ratio. The activation functions are ReLU because of its non-negative linearity (e.g. sigmoid is non-linear and not suited for linear regressions), and the output activation function is linear.

Since Convolutional layers seemed to work fine from the beginning, I decided to experiment with heavier and lighter versions of the same CNN-based model rather than with other types of Architectures. I implemented 4 models; they all follow the traditional CNN pipeline (Convolutional layers alternated by MaxPooling layers, flat layer, dense layers) but they vary in terms of number of layers -especially Convolutional layers-, layers size and hence number of parameters (2,122,371, 348,063, 283,623 and 130,697 parameters). All models implement 1D convolution layers, which all have the same filter size (7 samples - note that this is not at all related to the number of predicted features, even though the same number) and stride (1, except for the smallest model, which has a stride of 3 audio samples), but different number of filters. The down-sampling ratio of the Pooling layers is always 2, which means that downstream Convolutions are performed on audio waveforms with half the sampling-rate than the upstream layers. The smallest and ‘7 filters’ models present 7 filters (‘ideally’, 1 for each predicted feature) at the output of each Convolutional layer. As we will see, this characteristic provides the best performance.

# Evaluation metrics

Since we perform a regression task on continuous values normalized between 0. and 1., the loss is calculated using the mean absolute error, which is the sum of all the absolute residuals divided by the total number of examples in the dataset. I did not use the mean square error because, even though it is usually used to make bigger losses more evident, in my case any loss calculated with squared residuals would look smaller than it actually is because no annotation value is bigger than 1., and squaring the residuals would have the opposite effect, that is, it would make them smaller than they actually are. So, loss and evaluation metrics are practically the same in my regression model.

Since there are 7 nodes in the output layer of any model (one for each corresponding annotated spectral parameter), the total loss is the sum of all of the 7 single losses, and each loss has the same weight (1/7). It is important to note that, as opposed to classification, residuals will always be present between the output of a regression model and the ground truth, since it is not ideal or possible for a model to exactly predict a continuous value, which usually has a precision of many decimal points.

# Results

All models were trained over 20 epochs. ‘Conv1D’ represents a 1D Convolutional layer followed by a MaxPooling layer. ‘Loss’ means mean absolute error. For reproducibility, all models were trained and tested with exactly the same dataset shuffling and splits.

| **Model name** | **Num. of layers** | **Architecture** | **Num. of parameters** | **Train time** | **Training loss after last epoch** | **Test loss** |
| --- | --- | --- | --- | --- | --- | --- |
| Big-size | 9 | Conv1D x 5  Flat  Dense x 3 | 2.122.371 | ~ 23 min | 0.048 | 0.047 |
| 7 filters | 9 | Conv1D x 5  Flat  Dense x 3 | 348.063 | ~ 23 min | 0.048 | 0.046 |
| Mid-size | 5 | Conv1D x 2  Flat  Dense x 2 | 283.623 | ~ 21 min | 0.155 | 0.158 |
| Small-size | 3 | Conv1D x 1  Flat  Dense | 130.697 | ~16 min | 0.080 | 0.081 |

# Discussion

We can see that the biggest model has practically the same performance as the ‘7 filters’ model, and the smallest model outperforms the mid-sized model. Comparing the big model and the small model, the former reached almost half of the loss of the latter, at the cost of having ~16 times more parameters and 3 times the number of layers. **For the smallest and ‘7 filters’ models, I used, in the Convolutional layers (only 1 layer of this type in the smallest model), a number of filters equal to the number of spectral parameters to be predicted (7), and these are the models that perform best relative to their number of parameters.** In fact, the smallest model works better than the mid-size one, despite having smaller size, and also the ‘7 filters’ model works practically as good as the biggest model (which has ~6 times its numbers of parameters). This can be probably due to the fact that, by having as many kernels as the number of features to extract, each kernel tends to learn to recognize one specific feature only. In other words, we get probably closer to a 1-to-1 mapping between each Convolutional filter and each label to be predicted. Hence, basing hyperparameters choice on domain knowledge may increase performance and significantly reduce the network size. By normalizing the annotations’ values I solved a problem during early development, where the loss went from big values to ‘nan’, probably partially because of exploding gradients, caused by non-scaled output values; when scaled, the loss decreased significantly from the very first training epoch. Finally, feature selection is not needed since we are dealing with the raw audio waveform in an end-to-end fashion, rather than with extracted audio features. I could have padded the dataset audio files with extremely soft noise rather than 0s, and this would probably improve the performance of the model with non-trimmed percussive sounds of 1 second duration (implying some recorded ‘silence’, which is actually very soft noise, before or after the one-shot sound), but it did not seem to represent a major factor in my case. Also, I could use a different Pooling strategy, average rather than max, as it would produce a down-sampled audio-waveform which better represents the original one.

**I realized that MIR is very important even for those focused on Synthesis/Generation. Also, properly pre-processing and formatting the data for feeding it into a network, is sometimes harder and more time-consuming than designing the network itself.**

# References

Font, F. (2019). *Audio Commons Audio Extractor*. GiHub. Retrieved March 21, 2023, from https://github.com/AudioCommons/ac-audio-extractor

Ramires, A. (2020, February 12). *Freesound One-Shot Percussive Sounds*. Zenodo. Retrieved March 21, 2023, from https://zenodo.org/record/4687854#.ZBmPrOzMInU

Ramires, A. (2023, August 2). *Automatic characterization and generation of music loops and instrument samples for electronic music production*. CORA TDX Tesis Doctorals en Xarxa. Retrieved March 21, 2023, from http://hdl.handle.net/10803/687697