Ecole Polytechnique Fédérale de Lausanne

Signal Processing Laboratory LTS2

Semester Project Report

End-to-end learning for music audio exploration

Author:
Justine Weber

Supervisors:
Benjamin Ricaud
Helena Peic Tukuljac
Prof. Vandergheynst

June 7, 2019

Contents

Abstract

Deep learning algorithms allow learning complex models and are at the center of more and more research topics. Putting the theory of deep learning into practice requires thinking deeply about design and optimization, in order to get good performance while keeping it feasible. This can be applied to tasks such as music classification, which is the core of the project.

1 Introduction

The goal of this project is to get a better idea on the way artificial neural networks can be trained using raw waveforms as input, for machine learning tasks related to music.
The original purpose of this study was to get an insight on *WaveNet*, one such Deep Learning algorithm for music generation. Through the project, we have decided to take another path and focus on another algorithm intended for music classification, based on the following paper : *End-to-end learning for music audio tagging at scale*. This project has mainly been focused on implementing the solution suggested in the latter paper and trying to train the model.

In this report, I will describe the different approaches that I could try, the associated challenges, and try to highlight what is to take away from this 14 weeks-long research project.

2 Problem description

2.1 Why deep learning ?

2.1.1 Deep Learning

Deep Learning is the name for machine learning algorithms which are based on an artificial neural network architecture. These kinds of algorithms enable to learn complex models, taking big data sets as input for training. They can be applied to various tasks with various data types, such as natural language processing, computer visions, audio recognition. In the musical domain, two main tasks can be highlighted : music generation and music classification. The former belongs to a class of machine learning algorithms called unsupervised learning, whereas the latter is a case of supervised learning. Through this project, I was able to handle both types. In order to learn more about deep learning, I recommend consulting the course material of François Fleuret, *EE-559 Deep Learning* [19].

2.1.2 Audio processing & Neural Networks

There exist various preprocessing methods which can be applied to a raw audio signal in order to extract information from it. These methods enable to change the representation of the signal (typically in the frequency domain or the time domain), and highlight key features which we want to work on. One such preprocessing tool is the spectrogram.
This representation has the advantage of being two-dimensional, which brings the problem down to a matter similar to image processing. This is probably one of the reasons why the spectrogram representation is commonly used as input of deep learning algorithms. Audio preprocessing also enables to reduce drastically the size of the input data, which is useful in order to speed up training the algorithm. However, reducing the size of the data obviously means dropping some information. Recent researches have therefore investigated on the feasibility of treating raw audio waveforms directly as input to machine learning algorithms, without any preprocessing.

We could expect two outcomes from this method : either that the trained model will be more precise, since the input is given sample by sample and not compressed at all, which would therefore give better performance, or that it won't be able to learn the features that were highlighted through the preprocessing step, leading to worse performance. The latter suggests that preprocessing the audio input "helps" the model to learn, whereas the former suggests that the algorithm can learn by itself key features from this input, and even some more features that were "hidden" by preprocessing.

The challenge then lies into designing a neural network architecture which can learn from raw audio waveforms. In particular, in the case of WaveNet (which is described in the following section), the output of the algorithm *"sounds more natural than the best existing Text-to-Speech systems, reducing the gap with human performance by over 50%"* [1]. In the case of music classification, we can expect that a "good" neural network architecture will lead to more precise tagging and better performance in terms of accuracy. As we will explain later, a drawback from this method is performance in terms of speed.

2.2 Related Literature

Through this project I have reviewed several papers tackling the problem presented above. Mainly three papers were used for this project : *WaveNet : A Generative Model For Raw Audio* (2016) [1], *End-to-End Learning For Music Audio* (2014) [3] and *End-to-End Learning For Music Audio Tagging At Scale* (2018) [4].

2.2.1 Wavenet

The starting point of this project was WaveNet. The original goal was to understand better what it is, how it works and how it can be implemented. WaveNet is an artificial neural network architecture, presented by Google in September 2016 [1]. I highly recommend reading the article which summarizes the paper, referenced at [2].

Essentially it presents a neural network architecture for generating speech (TTS) or music. The neural network takes as input raw audio waveforms and can generate new waveforms. Through the paper they explain how they have been able to provide a solution for increasing the receptive field in an efficient manner, i.e. without requiring many layers, or large filters.

2.2.2 End-to-End Learning For Music Audio - *Tagging at Scale*

After some time spent on WaveNet (four weeks), we have decided to follow another direction for the next part of the project, based on another paper : *End-to-End Learning For Music Audio* (2014), on which the scientist Sander Dieleman has also contributed. This paper presents a neural architecture designed for classifying music audios according to predefined tags (using MTT dataset, described in section 3.2.1). It describes how they have tried to adapt it in order to make the algorithm learn from raw audio without prior knowledge. In a nutshell, the conclusion of their experiments is that using spectrogram as input to the network gives better results in terms of accuracy than using raw audio waveform. However they also show that despite worse results in terms of accuracy, the network was able to learn useful features from raw audio, and in particular frequency decomposition. This is encouraging and shows there is potential for end-to-end learning.

After some research, I came across a similar and more recent paper : *End-to-End Learning For Music Audio Tagging At Scale* (2018). The purpose of the research paper is more or less the same as the previous one and as the problem described previously, which I won't go through once again. Moreover they rely on results presented in the previous paper for comparison and implementations. The method used and the comparison are, however, not the same and will be described in more details in the following paragraph (2.3). One advantage of this paper is that code for the neural network architecture was provided. Therefore we have decided to go with this paper and start implementing the proposed algorithm.

2.3 End-to-end learning for music audio tagging at scale

The goal of this paper is to investigate on the need of *domain-knowledge* for designing deep-learning algorithms and in particular, depending on the size of the dataset. In their experiments, they have compared two different neural network

architectures, and three datasets of different sizes. Both neural network architectures have the same skeleton, that they call "back-end" and a different "front-end", depending on the input : spectrogram versus raw audio waveform.

Essentially the conclusion of the paper says that the larger the dataset, the less prior domain knowledge is needed : *"we have shown that, given enough data, assumption-free models processing waveforms outperform those that rely on musical domain knowledge"*.

We have decided to implement the solution they present, for the waveform part. Note that this project is a different purpose from theirs. We won't compare different datasets or different representations for the input, but simply rely on their code and neural network architecture.

2.3.1 Architecture description

As said above, the architecture of the neural network lies in two parts : a "front-end" and a "back-end". Figures 1 and 2 (taken from the original paper) illustrates it.

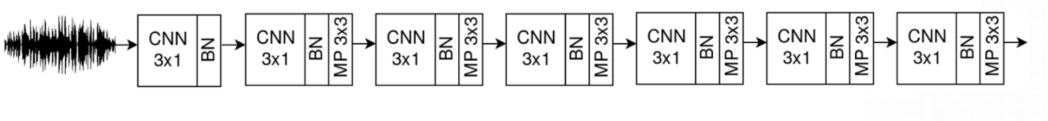

Figure 1: "Front-end" architecture.

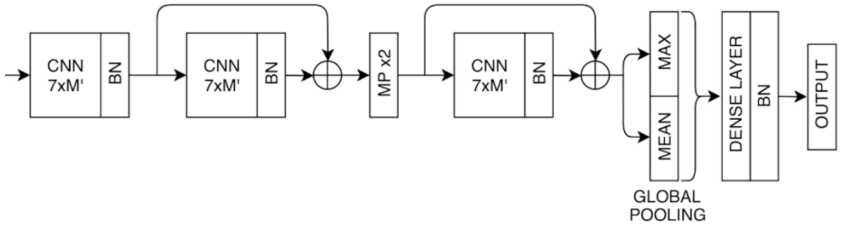

Figure 2: "Back-end" architecture.

We have used the "basic configuration" described in the paper. The "front-end" part consists of seven consecutive layers of 1-dimensional convolutionnal neural network, batch norm and max pool. The output of this first part is then fed to the back-end part, consisting of three convolutionnal neural network layers, two pooling layers and one dense layer. As shown by the arrows on figure 2, residual connections are added between layers and enable to *stabilize learning while slightly improving performance* according to the paper. Details on the shape of each layer for example

can be found in the original paper [4], or in the code and are not described here as there hasn't been any experiments on those in this project.

2.3.2 Loss, optimization function and convergence

Their model is trained using the Adam optimizer of SGD (stochastic gradient descent). The optimizer has the role of minimizing some loss function in order to make the network learn. The loss function used is cross-entropy. Finally, the metric used in order to analyse the performance of the algorithm is AUC (*Area Under the Curve*). The AUC value is a number between 0 and 1 which gives a measure of "separability". The closer the AUC value is to 1, the better the algorithm is at determining whether a song has a particular tag, and more precisely, the less it gives "false positive" or "false negative". An AUC score of 0.5 means the model has no class separation capacity. A good explanation of AUC is given in the following article : [15].

3 Implementation and Training

The goal of this report is not to explain in detail the theory behind the neural network architecture used, as it is already the purpose of the paper [4], but rather to give an overview on the way it could be implemented and describe some experiments. In this section I will explain how the dataset has been handled, how tensorflow has been used and finally how experiments could be done.In order to get a better understanding on the situations described in the present section, I encourage the reader to consult the project code, available online on github [10].

3.1 Environment and resource description

The code we started from had been produced using `TensorFlow API`. We have therefore decided to use this library for the whole project. Two other options could have been considered : `Keras`, which is a library built on top of TensorFlow, or `Pytorch`, another open source library for deep learning.

Regarding the environment, the code has been written and partially tested on my own machine : MacBook Pro 2013, Intel Core i7 2,3 GHz CPU, 4 cores, 16GB RAM. All training and the other testing parts have been done on the laboratory's servers, by connecting through ssh. Through this server, I had access to 4 GPU GeForce GTX 1080 Ti and a lot of RAM.

The code is organised in several python files and experiments are presented in a jupyter notebook.

3.2 Dataset

3.2.1 MagnaTagATune

The dataset used for training our network is the same as the one used in the paper [4] (and [3]) : the MagnaTagATune dataset. It contains 25′863 songs and tables containing tags associated with each song. The songs are mp3 files of 30 seconds.

3.2.2 Data analysis

There are 188 different tags, related to musical genre, instrument, rythm, etc such as "voice", "guitar", "string", "techno", ... The most represented one is *"guitar"*, with 4′852 associated songs. Each song can be associated with more than one label. The maximum number of tags associated to a same song is 27.

The following figures illustrate some interesting insights about the data :

singer	duet	plucking	hard rock	world	bongos	harpsichord	female singing	...	rap	metal	hip hop	quick	water	baroque	women	fiddle	english	mp3_path
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	f/american_bach_soloists-j_s__bach_solo_cantat...
0	0	0	0	0	0	0	0	...	0	0	0	0	0	1	0	0	0	f/american_bach_soloists-j_s__bach_solo_cantat...
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	f/american_bach_soloists-j_s__bach_solo_cantat...
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	f/american_bach_soloists-j_s__bach_solo_cantat...
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	f/american_bach_soloists-j_s__bach_solo_cantat...
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	c/lvx_nova-lvx_nova-01-contimune-30-59.mp3

Figure 3: Label Table Snapshot.

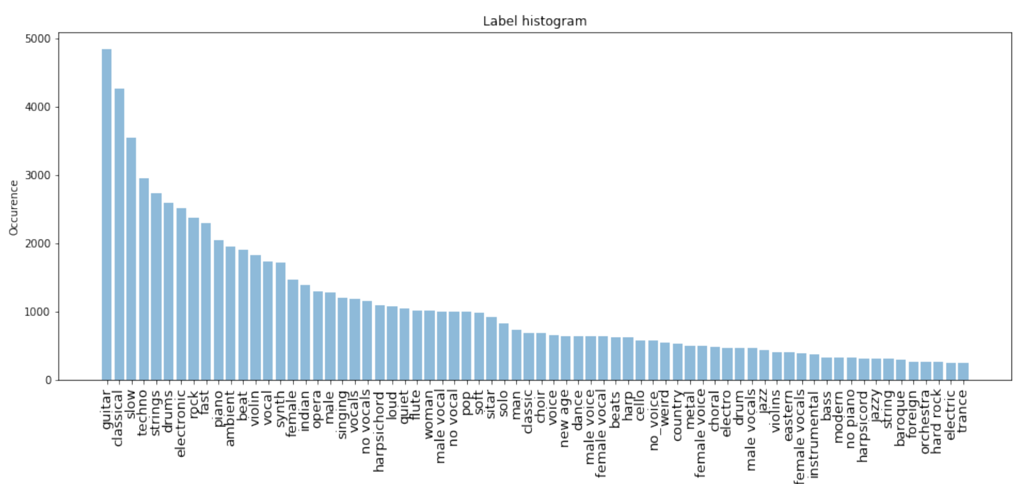

Figure 4: Label histogram.

- Figure 3 is a snapshot of the way the label table in constructed. The first columns are hidden and contain the index of the table and an identification number for each song (which we don't use). As shown in the figure, the last column contains the name of the song. MP3 files are named the same way and we use this column to link audio files to labels.

- Figure 4 presents the distribution for the 10 most represented tags.

- Figure 5 is a cooccurrence matrix showing how likely it is that a given song has some label if you already know it has another one. The darker the red color is, the less likely it is that a song has both labels. Tending towards blue color means that the two labels have high chances of overlapping. This analysis will be useful when selecting labels on which to train the algorithm.

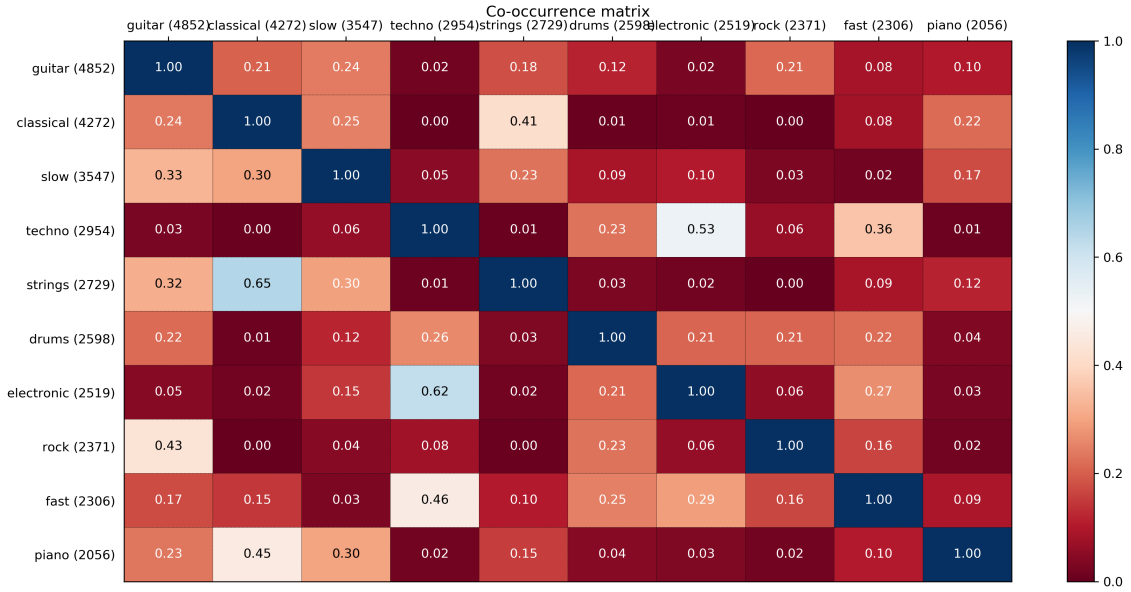

Figure 5: Cooccurrence matrix.

3.2.3 Loading data

- Labels
  The labels are given as a `csv` file. Some cleaning processing has been needed in order to properly extract the labels in numerical format. For that purpose, I have used the `pandas` library. Basically what has been done is : removing quotes from the data, separating columns which had a special separator ('\t') and reordering the labels for statistical analysis (paragraph above 3.2.2).

- Songs
  The songs are given in `mp3` format. The first step has been to load audios. Several libraries are available for that purpose, such as `ffmpeg` or `wave` (python module). We have decided to use `librosa` as it is broadly used and supports `mp3` format. Moreover, librosa was used for the WaveNet project and I could build on that example. Note that while loading songs I have sparsly encountered `EOF Error` (*"End of file"*). These are probably due to anomalies in a few files and I have decided to ignore those (using `try - except`).

3.3 Neural Network Model & TensorFlow

As said in paragraph 2.2.2, code for the neural network architecture only was already provided. The available code corresponds exactly to the design described in paragraph 2.3.1. The bulk of the work was therefore to assimilate the code and be able to adapt it to the MTT dataset in order to train the algorithm, according to the description of the paper regarding loss function, optimization function ,...

The main challenges related to this work were : understanding what the model expects technically and then finding a way to build the data into the expected shape in order for it to be trainable.

3.3.1 Input format

Once audios had been loaded (i.e. we have it available in python lists or `numpy` arrays), the next step was to give it as input to the network. This is where it gets challenging. Despite having the code and the paper at our disposal, there was no evidence of the format expected for the input of the network. In the code, they suggest using a *placeholder* x, without indication on the shape it should have. In the paper they suggest cutting audios into several pieces in order to use less memory, but it is not more accurate than that. Groping to try guessing the shape of the input, I first deduced that the expected input should have three dimensions, as it is the shape expected by tensorflow function `tf.layers.conv1d` [12]. The three dimensions should correspond to the following : `(batch, length, channels)`. Corresponding dimensions have yet to be found. The channel dimension is kind of a third dimension that doesn't appear at first sight. In the case of colored images for example, the number of channels is 3, corresponding to red, blue and green values. In the case of audio signals we don't have such analogy. Looking at WaveNet again and on the web, I figured the number of channels for audio signals is simply 1.

The other two dimensions are then quite intuitive. We have decided to first try to cut each song into 9 pieces (as the length of each audio file is $465'984$, a multiple of 9), leading to chunks of approximately 3 seconds. Typical shape for the input

of the network at each epoch is : $(180, 51776, 1)$, which corresponds to a group of 20 songs divided into 9 pieces. The dataset is converted into several tensors of this shape.

3.3.2 Feeding input to the network

Then comes the question of feeding it to the network. The first approach I have tried is to feed the input when building the network :

```
# Create tensor of expected shape from audio data
audios = load_audio(dataset)

# Build the model using provided function
net = build_model(x=audios, is_training=True,
                                        config=my_config)
# Some other stuff for initializing tf model
[...]
# Launch training
sess.run()
```

* some parts of the code have been simplified for comprehension. Cf. github repository for full code [10]

The problem with this method is that we have to feed all the data at once into the algorithm. This is way too much to handle for the GPU Memory and we therefore have to feed data to the network little by little. We have empirically found that batches of 20 songs is an acceptable number (not too low and which manageable for the GPU). In order to update the input at each iteration, we use the parameter `feed_dict` of tensorflow function :

```
# Create placeholder (tensor) of expected shape
x = tf.placeholder(tf.float32, shape=(180, 51776, 1))

# Build the model using provided function
net = build_model(x, is_training=True,
                                        config=my_config)
# Some other stuff for initializing tf model
[...]

# Launch training
for each batch :
    # Create tensor of expected shape from audio data
```

```
audios_batch = load_audio_batch(dataset)
# Feed to the network
sess.run(feed_dict={x: audios_batch})
```

* some parts of the code have been simplified for comprehension. Cf. github repository for full code [10]

As we will talk about more in detail in the following section, this last solution can be further improved in order to avoid alternating between I/O and training operations at each iteration (cf. 3.4.2).

3.3.3 Where to put the labels ?

Once we can feed the audio to the network, the next question is what should we do with the labels ? They should not be given as input together with the audio, first of all, because it doesn't make sense given the architecture of the neural network and second because the shape wouldn't fit.

The idea is that the labels are used at the end of each iteration to compare the output of the neural network with the truth. The comparison is done through calculating a loss value, and serves for the optimization function, which is Adam here (cf. 2.3.2). We want the output of the neural network to have the shape : `(number of songs, number of labels to predict)` (i.e. for each fed song it should make a prediction on each label we focus on). In neural networks, the shape of the output is determined by the number of neurons in the last layer. So we want the number of neurons at the last layer to be equal to the number of labels which we train our algorithm on (and which we want to predict). The predictions could have several different numerical representations but the most logical one is a probability : for each song we give a probability that it is tagged with some label. The sigmoid activation function gives values between 0 and 1 and is therefore well adapted to this kind of problem where we want a probability as output.

To summarize, for each song given as input we want the neural network to give a probability that it is tagged with a given label : we need the last layer of the neural network to have the size of the number of labels we try to predict, and use sigmoid as activation function. Once we have this probability, we can compare it to the truth (0 or 1), i.e. compute the loss value, and make the algorithm converge through the optimization function.

As illustrated in the provided code here, a parameter "`config`" is given for building the model (as argument to function `build_model`). It is actually a dictionary which expects a parameter called "`numOutputNeurons`". A question that arose is : isn't that parameter actually the number of output neurons that we want at the

end and therefore the number of labels we try to predict ? However there is nothing referring to this in the paper, and they even suggest using 500 for this parameter *"a dense layer with 500 units connects the pooled features to a sigmoidal output"* (section 4 of [4]), whereas the total number of labels (for the MTT dataset at least) is 188 (cf. 3.2.2).

Therefore in order to make the algorithm learn, we first add a dense layer to the output, with the following characteristics :

- input : the output of the neural network model
- number of neurons : the number of labels we try to predict (learn)
- activation function : sigmoid

3.3.4 Making the algorithm learn

As explained in the paragraph right above (3.3.3), once we have the output of the network, we make it pass through an added layer. Then we compute the loss function, which is the cross-entropy (cf. 2.3.2). Finally, we update the weights of the neural network by minimizing the loss through the optimization function : Adam (cf. 2.3.2).

The final step before ending this part is therefore to compute the loss value. There are a lot of different tensorflow functions for computing cross-entropy and choosing one can be tough. Some use softmax, other sigmoid, some are "with logits", ... The following post from stackoverflow explains very well which function should be used in which case : [14]. Essentially, cross entropy functions using sigmoid allows to deal with non-exclusive labels and so apply to cases of multinomial classification, while softmax deals with exclusive classes. In our case each song can be tagged with several labels to we have to pick one using sigmoid. Now three options are available : `tf.nn.weighted_cross_entropy_with_logits()`, `tf.losses.sigmoid_cross_entropy()`, `tf.nn.sigmoid_cross_entropy_with_logits()`. The first two allow additional parameters that are not needed here. We therefore use the last of the three.

The following example of code illustrates how we give the labels to the algorithm and make it learn. Note that we also compute the AUC score in the end, which is not used in the learning process but as a metric to see how the algorithm learns through the iterations.

```
x = tf.placeholder(tf.float32, shape=(180, 51776, 1))
y = tf.placeholder(tf.float32, shape=(180, nb_labels))
```

```
net = build_model(x, is_training=True, config=my_config)

predictions = tf.layers.dense(net, nb_labels,
                                    activation=tf.sigmoid)

loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=y,
                                     logits=predictions)

reduced_loss = tf.reduce_mean(loss)

train_op = tf.train.AdamOptimizer(learning_rate=0.001)
                                   .minimize(reduced_loss)

auc = tf.metrics.auc(labels = y, predictions=predictions)
```

* some parts of the code have been simplified for comprehension. Cf. github repository for full code [10]

3.3.5 Saving the model state

Once the model has been trained, we need a way to recover its state, in order to make new predictions. This is done using TensorFlow Saver (`tf.train.Saver()`). The following code snippets show a small example on how to use it for saving and restoring respectively.

```
saver = tf.train.Saver(var_list=tf.global_variables())
saver.save(sess, checkpoint_path)
```

* some parts of the code have been simplified for comprehension. Cf. github repository for full code [10]

```
saver = tf.train.Saver(var_list=tf.global_variables())
ckpt = tf.train.get_checkpoint_state(checkpoint_path)
saver.restore(sess, ckpt.model_checkpoint_path)
```

* some parts of the code have been simplified for comprehension. Cf. github repository for full code [10]

In our case, as the network connection was often unstable (I was connected to the labs servers through ssh) and training was very long, it has been important to make checkpoints storing more flexible than the simple example presented above. First, each time training in launched, a new checkpoint folder is created with a

custom name (timestamp). Second, there is a possibility to restart training from a previously stored checkpoint : for example it is possible to stop training at 32/50 epochs and continue for epochs 33 until 50, later, after restoring previous state.

I have relied heavily on the functions from WaveNet code, in order to build these functionalities [8].

3.4 Experiments

3.4.1 Selecting data

To start, we have decided to select two or three labels only, among the 188 ones. The idea was to choose well represented labels which overlap as little as possible. For selecting the labels, I have therefore relied on the cooccurrence matrix (cf. 5). I have tested two configurations :

1. guitar vs. techno
2. piano vs. rock

In order to try to facilitate and speed up learning for the algorithm, for each of these experiments I have selected only audios which are at least tagged with one of the two labels. This has considerably reduced the number of songs available to train : for configuration 1 we go down to $7'727$ and for configuration 2 to $4'387$ available audios (all subfiles combined), over $25'863$. Note that we feed audios by batches of 20 songs to the network and therefore make the total number of audios a multiple of 20 (we discard the last audios).

Moreover, since data is separated into 2 sets : training and testing, we have less available data for training. Splitting data between training and testing is made easier by the division of audio files into subfiles. Typically I use subfiles [0123456789abc] for training, and [def] for testing. Note that we need to be careful on the distribution of labels among subfiles.

3.4.2 Timing

After some first experiments, I have found out that training the algorithm would be very long. Typically for configuration 2, with $3'360$ songs for training, it takes approximately 30 minutes per epoch. So training it with 50 epochs for example takes 25 hours.

(as a reminder, we distinguish epoch and iterations : one iteration corresponds to running the algorithm with one batch of song, whereas one epoch corresponds to running the algorithm with the whole training set. one epoch = several iterations)

At each iteration, as described in section 3.3.4, the algorithm loads the data batch (typically 20 songs, each divided into 9 chunks) and feeds it to the network. Doing some time measurements, I have realized that it took as much time for loading as for training. In order to make running faster, we have therefore tried to understand what could speed up loading data. First we have tried converting all `mp3` files into `wav` and load `wav` files with `librosa` instead of `mp3` files. This has indeed slightly fastened the loading step but not significantly enough to make a big difference. Typically loading 100 songs in mp3 format takes 37.91 seconds, whereas loading them in wav format takes 42.58 seconds. Note that loading files on my machine was much faster than on the lts2 servers : it takes less than 6 seconds for 100 `mp3` files. This is probably due to that fact that the dataset is stored *closer* to where the algorithm runs on my computer than on the servers (on the server I put all the data in a particular folder, different from the one where the code lies).

We also thought about using other libraries than `librosa` such as `ffmpeg` of `wave` but din't get time to make proper measurements with those. However `librosa` is very broadly used (and was used for WaveNet for example) and the problem should not come from the library.

Finally, what came out from discussions at the end is that alternating between loading and training was not optimal at all for using the resources of the computer. Loading corresponds to I/O operations which are done by the CPU whereas training runs on the GPU (cf. Figure 6). As the way code was written makes operations alternate very frequently between loading tasks and training tasks, the GPU is probably never fully used.

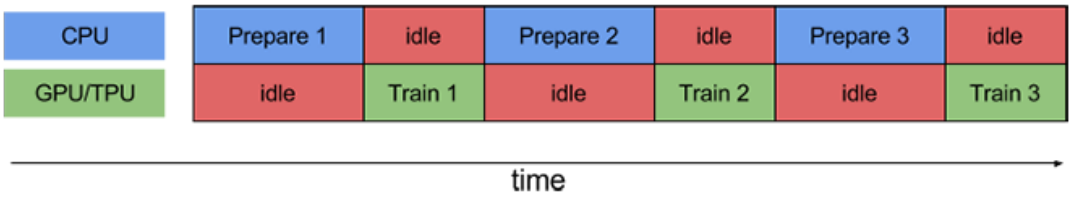

Figure 6: CPU / GPU use. Source image : TensorFlow, Data input pipeline performance [13]

The next step is therefore to load all data at once (if there is enough available memory on the computer) and then split it onto several groups and feed them one by one to the network. That way we should optimize the use of GPU resources.
One should also look at the `Dataset API` from Tensorflow, which provides solutions for handling big datasets in an optimal way.

3.4.3 Results

Due to lack of time, I haven't been able to do as much experiments as I would have wanted. The best training I could do corresponds to configuration 2 of previous paragraph (3.4.1) : using 3'360 audios for training, each having at least one of the piano or rock labels and train on those two labels for 50 epochs. As said before the data is fed to the network by batch of 20 songs, each cut into 9 chunks, leading to inputs of shape (180, 51776, 1). One epoch therefore corresponds to 167 iterations. Total time for training was 25 hours, each epoch taking approximately 30 minutes. The AUC score reached on the training set in the end is 0.9457, which shows the algorithm actually learns something. Left to check whether it can predict labels for unknown audios. All the functions for training and training are in the repository, together with some plotting tools for the AUC score and loss values, ready to be used in order to get more results.

4 Discussion

Taking a step back, there is a lot of things I would do differently if I went back in time.

A lot of things have been tried but due to time constrains and the difficulty to train the network, conclusions are missing and a lot of questions are left open. For example, is it better to randomize the order of inputs within a batch than not to ? Should the order in which the batches are fed to the network be different at each epoch ?

4.1 Critical assessment on the model

First I would like to talk about some choices that were made in the algorithm and the paper, that could maybe have been done differently. For example, randomization. In order to randomize the order in which the files are fed to the network, I have relied on a function from WaveNet. This function implements randomization with replacement. In consequence, some songs can be taken several times through a same epoch and some songs are not taken at all. In our case, we do not draw audios from the whole dataset every time, but from subsets one by one. Hence once we have been through some subfile we take another one and cannot draw again from the first subfile. For that reason I think drawing without replacement (i.e. simply shuffling randomly) should be more appropriate in our case and would be something to try.

Another question is the loss function to use. In the paper [4], they say they use cross-entropy for the MTT dataset, but MSE (Mean squared error) for the 1.2M dataset. Why is that ? One thing to try would be to use this loss function.

Anyway this shouldn't make a big difference on the results, but could be interesting to understand why they chose using two different loss functions.

4.2 Timeline & Methodology

What would I tell myself fourteen weeks ago ? Through this project I have been able to try a lot of things in the code and sometimes lost a lot of time on tasks that could have been done faster.

4.2.1 TensorFlow

First of all : TensorFlow. Choosing this library has been the most logical choice as the code we relied on had been implemented using it. However taking a step back now, I think it could have been a good idea to use another library (`keras` or `pytorch`). The problem with TensorFlow is that is it quite low level and very hard to debug. `Keras` which is built on top of TensorFlow is closer to human language and easier to understand for someone who has no practical experience with deep learning. To compensate for this, I have often relied on code from WaveNet, which is actually quite complex. I have sometimes tried to take pieces of code from it and adapt it to my problem but was not the most efficient solution as it took me a lot of time to understand some details that were absolutely not needed in my case.

Anyhow something that I would recommend to anyone starting a practical deep learning project using TensorFlow and who has no experience with it, would be to first spend one or two weeks reading about it, practicing and learning how it works. Throughout the project I have often needed to try lots of different approaches in order to make the code compile, or spend a lot of time understanding parts of the code (in particular for WaveNet), due to a lack of knowledge about TensorFlow functionning. This would also have made me discover useful APIs such as the `Dataset API` of TensorFlow sooner.

4.2.2 Local vs. Remote testing

At the beginning of the project I have started coding on my computer. After a while, I have asked access to the laboratory's servers and started doing some experiments remotely. This is probably one of the errors I have done.

In particular, the reason why I have implemented data feeding in a naive way (i.e. alternating between loading and training at each iteration) (cf. paragraph 3.4.2), is because my computer could not handle loading all the dataset at once. However the laboratory's server has much more available memory and manages loading all data at once. Knowing this from the beginning would have made me build the code another way and make the algorithm faster without having to rearrange all.

Also I have started implementing checkpoint storing and recovering quite late in the timeline of the project and I should have done in earlier in order to avoid losing data because of network issues (ssh connection disrupted).

4.3 What is next ?

As said earlier, this project is not finished and there are still a lot of things to do and test. In particular I think the following tasks are essential :

1. **Post-processing** : for the moment predictions are made on each chunk of song. In order to evaluate the performance of the algorithm more precisely, merging predictions of all chunks related to the same song would be important. One suggestion for selecting labels in the end could be by "majority vote" for example.

2. **Training & Testing** : more generally lots of training should be done with much more audios as input, using more than two labels. I was able to start some training on the algorithm but not to test it properly and testing would also be important to see how we can improve it.

Regarding the future of the project, as I have been able to discuss with people from the laboratory, I think it could be interesting as a second project to try parallelizing the algorithm (in particular loading and training), in order to speed it up and use more resources from the 4 available GPUs.

4.4 Takeaways

Finally, through this project I have learned how to face different challenges, use TensorFlow, handle a big dataset and organise work.

There are a lot of things I haven't done right the first time and that could have saved me a lot of time, but it's by making mistakes that we learn the most. I regret not having more time for testing the algorithm now that we are so close but this could be done if one wants to pursue the project.

I would like to thank in particular my two supervisors : Benjamin Ricaud and HelenaPeic Tukuljac, who have guided me through useful advice, in particular on how to organise and report my work, and who have always been receptive to my questions and give me feed-back.

5 Conclusion

In the end, this 14 weeks-long project has mainly been focused on manipulating TensorFlow, and finding a way to implement training in order to obtain results. The

next step is to get more results, analyse them and try improving the model in order to get better performance.

Using raw audio waveform as input to neural networks definitely has potential, but needs a lot of ressources as I have shown through this report. Is the gain in accuracy worth the cost of resources ? This is what should be investigated furthermore.

Thank you for reading.

References

[1] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu (September 2016) *WaveNet : A Generative Model For Raw Audio*, Google DeepMind, London, UK. Available at : `https://arxiv.org/pdf/1609.03499.pdf`.

[2] Aäron van den Oord, Sander Dieleman, (September 2016) *WaveNet : A Generative Model For Raw Audio (Blog article)*, Google DeepMind, London, UK. Available at : `https://deepmind.com/blog/wavenet-generative-model-raw-audio/`.

[3] Sander Dieleman, Benjamin Schrauwen (2014 IEEE ICASSP) *End-to-end Learning For Music Audio*, Ghent University, Electronics and information systems department. Available at : `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6854950`.

[4] Jordi Pons, Oriol Nieto, Matthew Prockup, Erik Schmidt, Andreas Ehmann, Xavier Serra (ISMIR 2018 - 19th International Society for Music Information Retrieval Conference, Paris, France) *End-to-end Learning For Music Audio Tagging At Scale*, Music Technology Group, Universitat Pompeu Fabra, Barcelona and Pandora Media Inc., Oakland, CA. Available at : `http://ismir2018.ircam.fr/doc/pdfs/191_Paper.pdf`.

[5] Edith Law, Kris West, Michael Mandel, Mert Bay and J. Stephen Downie (2009). *Evaluation of algorithms using games: the case of music annotation.* (ISMIR 2009 - 10th International Society for Music Information Retrieval Conference). MagnaTagATune dataset, Available at : `http://mirg.city.ac.uk/codeapps/the-magnatagatune-dataset`

[6] TagATune website, *TagATune, a game for music and sound annotation.* Available at : `http://tagatune.org/`

[7] Github repository of some data analysis on MTT dataset. Available at : `https://github.com/keunwoochoi/magnatagatune-list`

[8] Github repository of the WaveNet implementation, *A TensorFlow implementation of DeepMind's WaveNet paper* (last commit April 2018). Available at : https://github.com/ibab/tensorflow-wavenet

[9] Github repository of the paper [4] *Tensorflow implementation of the models used in "End-to-end learning for music audio tagging at scale"* (last commit May 2019). Available at : https://github.com/jordipons/music-audio-tagging-at-scale-models

[10] Github repository of the project available at : https://github.com/JustineWeb/pds

[11] Librosa librairy, github documentation. Available at : https://librosa.github.io/librosa/

[12] Tensorflow function documentation : 1D convolution layer. Available at : https://www.tensorflow.org/api_docs/python/tf/layers/conv1d

[13] Tensorflow, *Data Input Pipeline Performance* Available at : https://www.tensorflow.org/guide/performance/datasets#input_pipeline_structure

[14] Stackoverflow post, *How to choose cross-entropy loss in tensorflow?* (October 2017). Available at : https://stackoverflow.com/questions/47034888/how-to-choose-cross-entropy-loss-in-tensorflow

[15] Toward data science article, *Understanding AUC - ROC Curve* (June 2018). Available at : https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

[16] Towards data science article, *How to use Dataset in TensorFlow* (last updated May 2019). Available at : https://towardsdatascience.com/how-to-use-dataset-in-tensorflow-c758ef9e4428.

[17] Tensorflow tutorial, *Custom training: walkthrough*. Available at : https://www.tensorflow.org/tutorials/eager/custom_training_walkthrough

[18] Medium article, *How to Install Tensorflow-GPU version with Jupyter (Windows 10) in 8 easy steps.* (September 2017). Available at : https://medium.com/@viveksingh.heritage/how-to-install-tensorflow-gpu-version-with-jupyter-windows-10-in-8-easy-steps-8797547028a4

[19] Course documentation (slides), *EE-559 – EPFL – DEEP LEARNING (SPRING 2019)*, taught by François Fleuret. Available at https://fleuret.org/ee559/.