POLITECNICO
MILANO 1863

Selected Topics in Music and Acoustic Engineering

DIST-NN

Neural Network based distortion plugin

Di Palma Riccardo (Person code: 10602207, ID: 988804)
Orsatti Alessandro (Person code: 10680665, ID: 994757)

Selected Topics in Music and Acoustic Engineering
Homework Assignment

POLITECNICO
MILANO 1863

Contents

1 Introduction

We present to you our project for the course of Selected Topics in Music and Acoustic Engineering. **DIST-NN** is a *Neural-Network-based plugin* implemented in the *JUCE* framework with the help of other softwares such as *CMake*. The main goal of the project was to implement a parametric model of a distortion pedal using inputs sampled at different values of the distortion parameter. We decided to try different architectures for the neural network in order to test the results of the plugin applied to an audio against the ground truth obtained using the distortion pedal with the same settings.

2 Background

As we know distortion is one of the most difficult analog effects to model due to its non-linear asymmetric nature. There are different methods for modeling analog distortion. The **waveshaping** approach consists in applying a (typically polynomial) transformation to the input signal. Another possible way is to perform **virtual analog modeling** of every single component in the circuit through the wave digital filter technique, or at least the parts of the circuit which affects the sound. The third option, the one we chose, is to utilize a **neural network** to "learn" an implicit representation of the input-output relationship of the pedal.

2.1 The distortion

Distortion is characterized by a complex behaviour in the frequency realm, *overdrive pedals* and in particular *tube guitar amplifiers* are known for emphasizing the second harmonic and to provide a rich increase in all harmonics in a non-linear fashion. Their response to the guitar and interaction with the volume are really difficult to mimic using mathematical functions. In this case a neural network could come handy. The ability of a neural network, especially the one we used, to learn implicit function is fundamental to recreate all the details and the response at different volumes that a distortion pedal can provide.

2.2 The neural networks

The neural network architecture is really simple, it consists of just 2 inputs and a variable number of hidden layers (8, 16, 24 or 32) for the **LSTM** layer and the

corresponding number of inputs and one output for the **dense** layer. *LSTM (Long short-term memory)*[1], a specific kind of recursive neural networks, are used to classify sequential data because of their ability to learn long-term dependencies between time steps of data. The neural network is trained in python using *pytorch* and then exported as a *.json* file through the *torchscript api*, to be loaded into our *C++* application.

3 Implementation

The following chapter will introduce to you all the main design choices and implementation methods we used to design our plugin.

3.1 Requirements

The main requirements for the plugin in order to run correctly are:

- **RTNeural**: a lightweight neural network inferencing engine written in C++ and designed for real-time audio processing[2]. The main idea behind the integration of *RTNeural* in the system is:

  1. exporting the weights from a trained network
  2. creating a model
  3. running inference

  In our case, we designed an *LSTM* model with 16 hidden layers (for the standard configuration) and a Dense layer at the end.

- **JUCE**: the most widely used framework for audio application and plug-in development[3]. We used it for the plug in implementation.

- **CMake**: an open-source, cross-platform family of tools designed to build, test and package software[4]. We used it for building and testing our system, and for its ability to import neural network libraries into *JUCE*.

3.2 Train

For what concerns the training procedure, we decided to train the model in python. We started from the **non-parametric training**, creating a *dataset* with

a *.wav* clean file as input, and the corresponding distorted *.wav* file as output. Once the *dataset* is created, we passed at the model implementation with the proper structure, and the training procedure can start. We decided to train with 10000 iterations, with the recording of the loss function at each iteration. Every time the loss function is the lowest, a *.json* file containing all the weights is created.

Once encountered some coherent results on the static model, we decided to pass to a parametric approach: the main idea is to create a model that has a two dimensional input (input file + parameter) with a one dimensional output (output file). This type of training is called **"Conditional Training"**. The input file is structured as an horizontal sequence of 6 original files on the first dimension, and a sequence of parameters on the second dimension: 0, 0.2, 0.4, 0.6, 0.8 and 1. The output file is instead a sequence of 6 processed files with increasing distortion rate. For a better explanation, Figure 1 shows the files' structures. After changing the number of inputs in the model, the training can start and evolve as the static one.

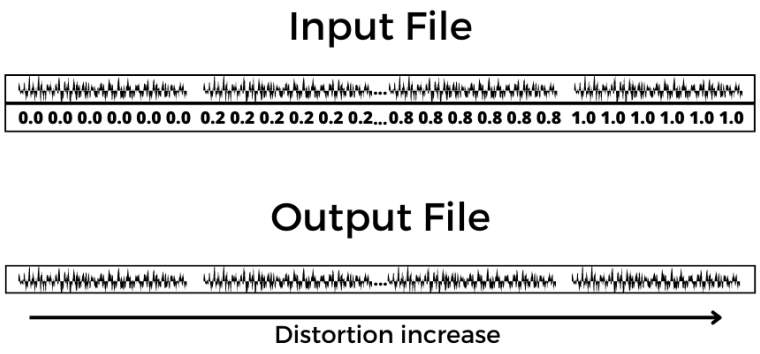

Figure 1: Parametric data structures

3.3 Design choices

For what concerns the design choices we decided to go with a separated header file for the **setLookAndFeel** class, which controls everything about our graphical user interface. The **on/off button** and the **parameter slider** are created in the Editor and passed to the Processor through the *audioProcessor* reference in order to be triggered every time the listener for the button or for the slider is called. The button globally acts as an *enabler/disabler* of the process block and the slider is passed, bounded in a 1 dimensional array with the input read from the buffer, to be the *input* of the forward function of our neural network. The **neural network architecture** itself is defined through the standard *RTNeural* procedure for *.json* files generated with *PyTorch* (an alternative would have been to use *TensorFlow*). The weights are passed with a path to the *.json* file, specified before, and loaded into our model.

3.4 Functioning

In the process block, the input buffer is processed as follows: if the effect button is enabled, we define two writer pointers, one for the left and one for the right channel. For each sample in both writer pointers, an input variable is defined as

```
float inputL[] = {channelDataL[n], effect};
float inputR[] = {channelDataR[n], effect};
```

where the two *channelData* are the *writerPointers*, n is the sample and the effect variable is the parameter. After that, we put the two variables as input for the model, which process it and assign the new values to *writerPointers*. The full *processBlock* script is:

```
float* channelDataL = buffer.getWritePointer(0);
float* channelDataR = buffer.getWritePointer(1);
for (int n = 0; n < buffer.getNumSamples(); ++n)
    {
        float inputL[] = {channelDataL[n], effect};
        float inputR[] = {channelDataR[n], effect};
        channelDataL[n] = model.forward(inputL);
        channelDataR[n] = model.forward(inputR);
    }
```

4 GUI

For the graphical user interface,as shown in figure 2, we opted for a pedal look with a big knob on top which controls the distortion rate, followed by a button that works as a bypass for the effect.

Figure 2: Plug-in GUI

5 Results

5.1 MSE

We decided to test **DIST-NN** by applying it to a sample in all its different layouts (using 8, 16, 24 and 32 hidden layers), and to use the same distortion effect used to train the model, the same settings and the same clean sample to create the track we compared it to.

The mean squared error is the way we decided to go to test the final results for the plugin. The numbers are:

- 8 hidden layers: **0.0077933217**
- 16 hidden layers: **0.008014743**
- 24 hidden layers: **0.008172143**
- 32 hidden layers: **0.008451953**

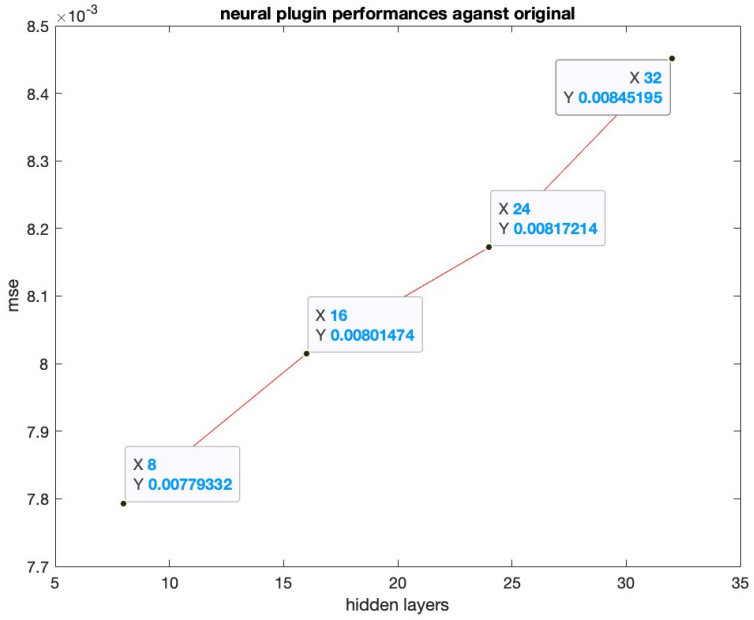

Figure 3: Plugin performances

As we can see in Figure 3 the numbers progressively gets worst with an increase in the number of hidden layers, which is an unexpected behaviour.

5.2 CPU

We also monitored the *CPU* usage from all the different configurations and those are the results:

- 8 hidden layers: **6-8%**
- 16 hidden layers: **8-10%**
- 24 hidden layers: **13-15%**
- 32 hidden layers: **15-18%**

The plugin responds clearly better in the configuration with the lowest number of hidden layers: 8.

6 Conclusions

In conclusion we are happy with the results obtained with our implementation of a real-time parametric neural plugin. The result is completely usable inside a *DAW* without pushing the CPU too far. A first step towards an improvement could be to model a true analog pedal and test the results with an *MSE* metric. Another interesting thing could be to push the limits of a neural network and try to model a time based stereo effect such as a stereo chorus, working on each channel separately.

References

[1] LSTM,
URL: `https://it.mathworks.com/discovery/lstm.html`

[2] RTNeural,
URL: `https://github.com/jatinchowdhury18/RTNeural`

[3] JUCE,
URL: `https://juce.com/`

[4] CMake,
URL: `https://cmake.org/`