LSTM GEAR

A study project on LSTM models with RTNeural library in C++

Marco Viviani

August 2023

1 Introduction

The goal of this project is to create a working plugin that emulates electric guitar effects using neural models in C++ through blackbox modelling of analog pedals. At the same time this plugin is intended to implement a chain of the above mentioned effects.
This report will try to highlight all the aspects, experiences and considerations addressed in drafting the code and how they were resolved. In this way it is hoped that this project will be useful for future works.
In the end we came up with an effective plugin capable of emulating any distortion effect of a real analog pedal with a low computational cost.
As per the premises the plugin is able to emulate a chain of effects composed of two stages showing satisfactory results.
Finally, this work has been a reason to deepen neural networks tailored to this task, in particular simple RNN models with LSTM (Long-Short Time Memory) architecture. Video's demo and Github links can be found in the "Links" section.

2 Project's overview

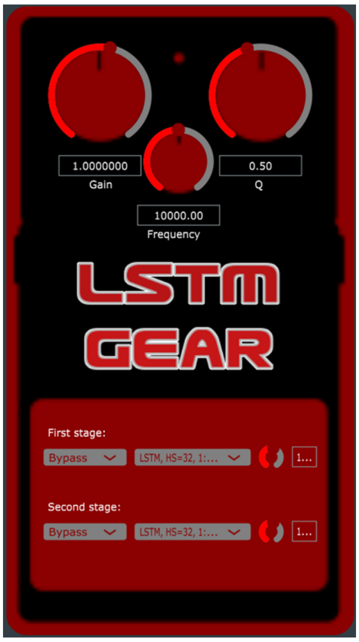
1.0000000
Gain
0.50
Q
10000.00
Frequency

Figure 1: plugin's final GUI

As mentioned before the idea behind this work is to create a plugin capable of emulating analog distortion pedals via neural networks. The software must also be able to combine effects where the order is chosen by the user. Therefore it will be possible to listen the trained models individually or in chain with other effects and make comparisons of the performances. In addition the user will be able to compare the same network trained in a different way, for example by changing the input and output dataset provided in the training phase or by changing the hidden size of the model. Accordingly you can create chains of models trained in different ways.
To do so the software will make use of a simple LSTM based RNN because it has been shown to have excellent performance compared to other networks (*paragraph 9.1*).
The initial idea was to develop a plugin capable of emulating any effect present in the author's pedalboard, but this was not possible due to LSTM, capable of reproducing only distortion effects (pure distorsion, overdrive, fuzz) and tube amplifiers. Thus a simple model like the one under consideration is not able to reproduce effects with large variations over time (tremolo, chorus, flanger and reverb). For this reason the range of available sounds has been reduced to overdrives and distortions.
Because of this it was considered useless to maintain a third stage - previously implemented - as three distortion effects in succession are excessive. It must be said that the third stage has been removed also because it has proved difficult to manage the gain of the various networks. Thus, it has been noticed that when a model is applied to an input signal, the gain decreases significantly. In addition to this,

it has not proved easy to train models with sufficient volume and only a few of them have managed to achieve an acceptable level. Lastly, each model has its own gain and this makes the output of each model always different. For these reasons (and also for time reasons) it was preferred to implement only two stages and concentrate efforts to obtain less but better models and improve them and the quality of the software.

For educational purposes it has been decided to leave the possibility of choosing a reverb (the best result obtained) in order to hear the sound of an effect different from distorsion.

It is believed that having excellent trained networkss, a third stage can easily be re-implemented as in the original demo.

3 Setup

The project was done on Windows OS and is built using CMake. It is necessary to create a folder for the project that contains another **folder with source files, CMakeLists.txt, images files, *.json* models and README.md.** Training has already been done. You need to include in the CMakeLists.txt the RTNeural library - explained later - and save it locally on the PC. Be sure to change the path before running the build, as well for the JUCE path.

The src folder contains the *PluginProcessor.cpp* and *PluginProcessor.h* files. In addition there are also the *PluginEditor.h* and *PluginEditor.cpp* files to make it possible to develop a GUI for our plugin. Only *.cpp* files have been added to the target sources. It was also necessary to add in the private part of the target link libraries JUCE'S DSP module: it allows to define and use DSP processes quickly.

An external sound card to connect the guitar is required with a low buffer size setup - for latency issues - and a sampling frequency of 44.1 kHz. Once all this is done it will be possible to build the project using the command "***cmake -B build***" via terminal, after being positioned in the project folder. Also **remember to change the paths** of the models in *.json* format (explained later) in *PluginProcessor.cpp* and of the *.png* files loaded by the *PluginEditor.cpp.*

4 LSTM model

The plugin uses a simple RNN based on LSTM as model. The project description from here on will be based on this architecture but it should be said that - except for the model loading part - this plugin is easily extendable to other model's types.

Long Short Term Memory (LSTM) was first proposed for the task of amplifier modelling.

LSTM is a RNN variant that incorporates the use of various gates to control information flow through each recurrent layer in order to avoid the exploding gradients of regular RNN.

Wright et al. [1] tested a new LSTM architecture for black-box modelling of nonlinear audio systems and both were compared to the WaveNet architecture and the Gated Recurrent Unit (GRU). Other comparisons have been made in [2]. Results are shown in *paragraph 9.1.* The architecture used in this plugin is the Wright's one.

The model under consideration consists in one lstm layer with one input and 8 hidden states plus one fully connected linear layer with 8 inputs and 1 output.

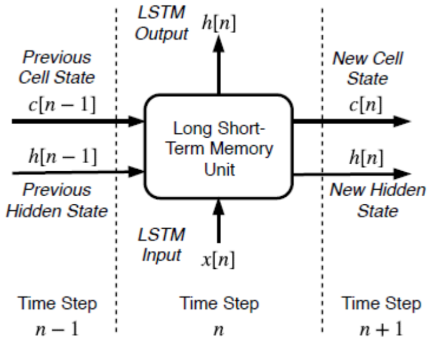

Figure 2: diagram of an LSTM unit, where c represents the cell state, h the hidden state and x[n] is the input signal at time step n. Image taken from [1].

As previously said this simple RNN model can't model more complicated effects like tremolo and flanger and is best suited for simpler distorsion effects or tube amplifiers.

5 RTNeural

Before going into detail, it is necessary to talk about how the models will be loaded from the C++ code and how inference will be performed. To do this we use the RTNeural library. RTNeural is a lightweight neural network inferencing engine written in C++. This library was designed with the intention of being used in real-time systems, specifically real-time audio processing.
RTNeural is capable of taking a neural network that has already been trained (usually with Tensorflow or PyTorch), loading the weights from that network, and running inference directly from a file in .json format. RTNeural has shown excellent performances for small layer sizes as in our case, out-performing PyTorch. Results are shown in *paragraph 9.3.*

6 Training with PyTorch

The script trains in PyTorch the LSTM model previously cited according to the method described in [1].
The script needs clean input data of a guitar that is played using various playing techniques and an output consisting in the input processed through a distortion effect. With only these two data the code will train a new model able to emulate the analog pedal.
For the plugin were trained neural networks for two pedals: an Ibanez TubeScreamer Turbo overdrive and a MOOER Distortion Solo. For completeness of the demo it was also decided to train the MOOER Sky Verb model - a digital reverb - in order to see the low results of the LSTM network for more complicated effects. All the recordings have been done at 44100 Hz.
The training has been done three times for every audio effect by modifying some parameters of the training or the dataset itself in order to study the performance when the parameters change. In particular:

- 1:00 min input, hidden size = 32
- 1:00 min input, hidden size = 96
- 3:30 min input, hidden size = 32

The trainings have been done for 1000 epochs obtaining the results presented in *paragraph 9.2.*

7 Technical description of the code

In this paragraph we will analyze how the models have been loaded, how the inference is run and how the 2-stage effects chain is implemented.

7.1 Model declaration

First we have to declare at the top of *PluginProcessor.h* what kind of model we are using:

```
using ModelType = RTNeural::ModelT<float, 1, 1, RTNeural::LSTMLayerT<float, 1, 8>,
        RTNeural::DenseT<float, 8, 1>>;
```

Subsequently - always in *PluginProcessor.h* - all the variables that will contain the model loaded with the values present in the *.json* are instantiated:

```
ModelType distorsion_lstm_32_1_min_input_FIRST[2];
```

These variables report the type of effect, the way in which they were trained and an adjective (first/second) that represents in which of the two stages this model will be used. Thus, initially the same network was used in all the stages for the same effect, but this was the cause of a disturbing white noise. This is due to the fact that for each block coming from the buffer the model was used several times without being reset in the same *processBlock()* function. So it was necessary to declare more variables for the same model. By doing so the white noise is gone.
Each model actually consists of a 2x1 array of *ModelType* for the same reason, where each element of an array represents a stereo channel. Without doing so the same model will be applied for both channels in the 'for' loop without being reset the second time. The *prepareToPlay()* function is essential because it is the means by which we reset the models at every block of samples:

```
distorsion_lstm_32_1_min_input_FIRST[0].reset();
```

7.2 LoadModel() function

The *loadModel()* function takes the trained *.json* model in the form of a binary stream as input and loads it into the second input, i.e. the type of network in which we want to load the weights. This is done in the first two lines of the function via the *nlohmann* function.

```
void StmaeprojectpluginAudioProcessor::loadModel(std::ifstream& jsonStream,ModelType& model)
{
    nlohmann::json modelJson;
    jsonStream >> modelJson;
    auto& lstm = model.get<0>();
    std::string prefix = ”lstm.”;
    auto hidden_count = modelJson[prefix + ”.weight_ih_l0”].size() / 4;
    RTNeural::torch_helpers::loadLSTM<float>(modelJson, prefix, lstm);
    auto& dense = model.get<1>();
    RTNeural::torch_helpers::loadDense<float>(modelJson, ”dense.”, dense);
}
```

Since the network has been trained with PyTorch it is loaded one layer at a time via *model.get* and suitable Torch helper functions provided by RTNeural. This is done at the beginning of *PluginProcessor.cpp* directly from the path of your PC:

```
auto modelFilePath1 = ”C:/.../distorsion_lstm_32_1_min_input.json”;
assert(std::filesystem::exists(modelFilePath1));
std::ifstream jsonStream1(modelFilePath1, std::ifstream::binary);
loadModel(jsonStream1, distorsion_lstm_32_1_min_input_FIRST[0]);
```

7.3 Audio Processor Value Tree State and input gains

A key aspect of the project was the implementation of input gains. Without them the loaded model can hardly be heard. Therefore it was necessary to do a preprocessing phase before each stage in which the gain was brought to audible volume levels.
To do so we must introduce the concepts of *AudioProcessorValueTreeState*: most of the plugin's functionalities - except stage gains and master gain - have been implemented with this class.
AudioProcessorValueTreeState is a class for sharing parameters between *Editor* and *Processor* and contains a *ValueTree* that is used to manage an *AudioProcessor*'s entire state.
We add as member of the *AudioProcessor* an instance of *AudioProcessorValueTreeState* class *apvts* and a function *createParameters()* that create and associate all parameters returning a ParameterLayout object:

```
juce::AudioProcessorValueTreeState apvts;
juce::AudioProcessorValueTreeState::ParameterLayout createParameters();
```

In *createParameters()* we start by creating a vector *parameters* of unique pointers to *RangeAudioParameter*, an abstract class for different types for *AudioParameter*:

```
std::vector<std::unique_ptr<juce::RangedAudioParameter>> parameters;
```

Then we fill the vector by instantiating different *AudioParameterFloat*, including the parameters for the input gain of the two stages. For example:

```
parameters.push_back(std::make_unique<AudioParameterFloat>(”first_input_gain_db”,
”Gain␣[dB]”, −12.0f, 12.0f, 0.0f));
```

Finally we return two iterators pointing to the first and one to the last element of the vector.
The *AudioProcessorValueTreeState* object has to be instantiated in the initialization list of the *AudioProcessor* constructor with the *createParameters()* function:

```
apvts(*this, nullptr, ”Parameters”, createParameters())
```

To handle input gains we declare two variables for each stage:

```
std::atomic<float>* firstInGainDbParam = nullptr;
dsp::Gain<float> firstInputGain;
```

In this case - unlike other parameters implemented with *apvts* - there is no communication between *Editor* and *Processor* because there are no edited elements to control input gains. The values are loaded at the beginning of the constructor from the standard values (0dB) returned by the *push.back* of the *createParameters()* function. For example:

```
inGainDbParam = apvts.getRawParameterValue(”first_input_gain_db”);
```

In the *prepareToPlay()* function we setup *firstInGainDbParam* and *firstInputGain* as well as the variables of the second stage. This is done by initializing the filter *void prepare(const ProcessSpec) noexcept.* It takes as input a *ProcessSpec* structure, where *sampleRate* and *samplesPerBlock* need to be specified:

```
firstInputGain.prepare(spec);
firstInputGain.setRampDurationSeconds(0.05);
```

Then - in the ”if” branch before each stage - a linear gain of +35.0f is applied and converted to dB:

```
firstInputGain.setGainDecibels(firstInGainDbParam->load() + 35.0f);
firstInputGain.process(context);
```

7.4 Filters and noise

Filters have been used to reduce noise of various kinds. First, the *dcBlocker* filter was implemented to suppress strong noise due to the flow of audio and direct current frequencies.
Secondly, two filters were used to suppress the white noise caused by input gains for the two stages:

```
juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter<float>, juce::dsp::IIR::
Coefficients<float>> dcBlocker;
juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter<float>, juce::dsp::IIR::
Coefficients<float>> firstNoiseBlocker;
juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter<float>, juce::dsp::IIR::
Coefficients<float>> secondNoiseBlocker;
```

In *prepareToPlay()* function we need to reset our filters and initialize the filter *void prepare(const ProcessSpec) noexcept* as in the previous paragraph:

```
*dcBlocker.state = *juce::dsp::IIR::Coefficients<float>::makeHighPass(sampleRate, 35.0f);
...
dcBlocker.prepare(spec);
dcBlocker.reset();
```

Filters *firstNoiseBlocker* and *secondNoiseBlocker* are then used in the *processBlock()* function in the ”if” branches before each stage to suppress noise before gain is applied to the network:

```
firstNoiseBlocker.process(context);
```

Filter *dcBlocker* is used at the end of the processBlock():

```
dcBlocker.process(context);
```

A fourth filter controllable by the user will be described in *paragraph 8.3*.

7.5 Model selection

Another key aspect that allows the plugin to work are the 4 easy-to-implement functions that allow the selection of the type of effect and the type of training:

```
int selectFirstEffectFunction();
int selectSecondEffectFunction();
int selectFirstModelTypeFunction();
int selectSecondModelTypeFunction();
```

For example *selectFirstEffectFunction()* returns an int number. Thanks to these outputs in the *processBlock()* the code enters a branch of the ”if-else” according to the model selected in the boxes. Since this time boxes have a graphical counterpart in the editor (*paragraph 8.4*), it was necessary to establish a link between them and the functions with the *apvts* class. In the *PluginEditor.h* we declare four variables *ComboBox*:

```
juce::ComboBox selectFirstModelBox;
juce::ComboBox selectSecondModelBox;
juce::ComboBox selectFirstModelTypeBox;
juce::ComboBox selectSecondModelTypeBox;
```

and an attachment variable for each box. For example:

```
std::unique_ptr<juce::AudioProcessorValueTreeState::ComboBoxAttachment>
selectFirstModelBoxAttachment;
```

Then we need to instantiate our attachments and link the *ValueTreeState* and the sliders in *PluginEditor.cpp*:

```
selectFirstModelBoxAttachment = std::make_unique<juce::AudioProcessorValueTreeState::
ComboBoxAttachment>(audioProcessor.apvts, ”FIRST_MODEL_EFFECT”, selectFirstModelBox);
```

Each time an option is selected in one of the four boxes the *selectFirstEffectFunction()* function outputs an *int* number and compares it to another *int* in each ”if-else” branch. Thus you enter the desired block of code.

7.6 ProcessBlock()

The *processBlock()* is where the real signal processing takes place. In the first stage, as well as in the second, a pre-processing phase is done only if a model is running. For example:

```
if (selectFirstEffectFunction() == 1 || selectFirstEffectFunction() == 2
    || selectFirstEffectFunction() == 3) {
            firstNoiseBlocker.process(context);
            firstInputGain.setGainDecibels(firstInGainDbParam->load() + 35.0f);
            firstInputGain.process(context);
    }
```

This is done because the incoming clean sound still has some white noise. The filter suppresses the white noise and then gain is applied. If you don't do this the white noise will be amplified with the gain. In this way instead we can hear the effect of the model without disturbing noises.
How the model applies to the incoming signal is described below; the code represents the first part of the first stage:

```
for (int channel = 0; channel < buffer.getNumChannels(); ++channel) {
        auto* x = buffer.getWritePointer(channel);

        if (selectFirstEffectFunction() == 0) {
            for (int n = 0; n < buffer.getNumSamples(); ++n)  {
                x[n] = buffer.getSample(channel, n) * firstGain;
            }

        else if (selectFirstEffectFunction() == 1 &&
        selectFirstModelTypeFunction() == 0) {
            for (int n = 0; n < buffer.getNumSamples(); ++n)   {
                float input[] = { x[n] };
                x[n] = distorsion_lstm_32_1_min_input_FIRST[channel].forward(input)
                        * firstGain;}
            }
```

In the first stage we want to apply an effect or bypass it. We enter a for loop because the actions we will describe are performed for both stereo channels. Each sample of the incoming block from the buffer is stored in a variable x pointing to each channel. At this point, thanks to *selectFirstEffectFunction()* and *selectFirstModelTypeFunction()*, the signal is sorted: depending on what number the functions return you enter a different block of code. Assuming that *selectFirstEffectFunction() == 0* as in the example, x is rewritten for each sample contained in the block and multiplied by a *firstGain* variable to modify the gain. This case represents the bypass present in each stage.
On the contrary if *selectFirstEffectFunction() == 1* and *selectFirstModelTypeFunction() == 0* as in the example a model is applied. Each sample present in the block is saved in the input variable. The input variable is then processed through the network using RTNeural's *forward()* function for each sample in a very signal processing style way. Also in this case there is a multiplication by *firstGain.* The ”if-else” sequence continues for each model implemented. The second stage is designed consecutively according to the same principle.
The last for cycle controls the output level:

```
    for (int channel = 0; channel < buffer.getNumChannels(); ++channel) {
        auto* x = buffer.getWritePointer(channel);
        for (int n = 0; n < buffer.getNumSamples(); ++n)
        {
            x[n] = buffer.getSample(channel, n) * rawVolume;
        }
    }
```

In conclusion the block enters the first stage and then the second one consecutively. If both stages are active the block is processed twice within the same *processBlock()* creating a chain of effects, if not it is processed once or bypassed for both stages. In this way the performance of the code depends only on the quality of the trained model. With better models and a gain control it is possible to implement a third stage.
It is possible to use any other type of neural network by properly declaring the model used at the top of the header file.

8 GUI

This section explains all the remaining aspects of the code regarding the GUI and the missing knobs.

8.1 Master gain

As in a real analog pedal a master gain has been created to control the final output of the plugin, in addition to the input gains already mentioned and the stage gains explained later.
In *PluginEditor.h* the master gain was declared:

```
juce::Slider sliderGain;
```

In the *PluginEditor*'s contructor all the specifications of the master gain knob have been set:

```
sliderGain.setSliderStyle(juce::Slider::SliderStyle::Rotary);
sliderGain.setTextBoxStyle(juce::Slider::TextBoxBelow, true, 100, 25);
sliderGain.setRange(-12.0, 12.0);
sliderGain.setValue(1.0);
sliderGain.addListener(this);
addAndMakeVisible(sliderGain);
```

while the position in the editor has been specified in the *resized()* function.
To make the editor communicate with the processor, the *sliderValueChanged()* function has been implemented for all sliders:

```
void Stmaeproject_pluginAudioProcessorEditor::sliderValueChanged(juce::Slider* slider)
{
    if (slider == &sliderGain)
    {
        audioProcessor.rawVolume = pow(10, sliderGain.getValue() / 20);
    }

    if (slider == &firstStageGain)
    {
        audioProcessor.firstGain = pow(10, firstStageGain.getValue() / 20);
    }

    if (slider == &secondStageGain)
    {
        audioProcessor.secondGain = pow(10, secondStageGain.getValue() / 20);
    }
}
```

This function modifies the value of the variable *rawVolume* that is located in the last for cycle of the *processBlock()* (*paragraph 7.6*). It converts the selected dB value to linear in order to make possible the product for each sample.

8.2 Stage gains

The stage gains arise from the need to have greater control over the volume of each single stage and go to correct the output. As previously mentioned, it is not possible to know in advance the exact output volume from a model. Stage gains are implemented exactly like master gain with the *sliderValueChanged()* function, except that *firstGain* and *secondGain* variables are located in each "if-else" branch of the two stages. In this way it is possible to change the model from the boxes without the knob being reset to the standard value.

8.3 LPF and quality factor

The plugin also two knobs for tone control, a low pass filter knob and a quality factor control knob. The sliders are declared in *PluginEditor.h* as follows:

```
juce::Slider frequencySlider;
juce::Slider qualitySlider;
```

In the *Editor* constructor we specify the properties of the Sliders and Labels and in the *resized()* function we specify their location. In the *Editor* constructor we actually allocate memory for these two pointers declared in *PluginEditor.h*:

```
std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment>qualitySliderAttachment;
std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment>frequencySliderAttachment;
```

and we link the GUI components to *AudioProcessorValueTreeState* parameters. For example:

```
frequencySliderAttachment = std::make_unique<juce::AudioProcessorValueTreeState::
SliderAttachment>(audioProcessor.apvts, ”FREQ”, frequencySlider);
```

Also in this case, unique pointers to the class *RangeAudioParameter* are created and returned in the *createParameters()* function:

```
parameters.push_back(std::make_unique<juce::AudioParameterFloat>
                    (”FREQ”, ”CutOff_Frequency”, 50.0f, 20000.0f, 10000.0f));
parameters.push_back(std::make_unique<juce::AudioParameterFloat>
                    (”Q”, ”Q_Factor”, 0.1f, 1.0f, 0.5f));
```

Then we need the class *dsp::IIR:Filter* of the DSP module to control the cut-off frequency and the quality factor using methods of the class *dsp::IIR:Coefficients.* To allow multichannel processing we are going to use a *ProcessorDuplicator* to “wrap” our filter:

```
juce::dsp::ProcessorDuplicator <juce::dsp::IIR::Filter<float>,
juce::dsp::IIR::Coefficients<float>> lowPassFilter;
float lastSampleRate;
```

In the constructor initialization list we add the instantiation for IIR filter calling the *makeLowPass()* method using default values:

```
lowPassFilter(dsp::IIR::Coefficients<float>::makeLowPass(44100, 20000.0f, 0.1))
```

In the *prepareToPlay()* method of the *AudioProcessor* we ”prepare” the IIR filter. This is done by exploiting the *ProcessSpec* structure to store some basic specifications for the filter. By using the *reset()* function we simply reset the processing pipeline (the same has been done for all the other filters).

```
juce::dsp::ProcessSpec spec{ sampleRate, static_cast<juce::uint32> (samplesPerBlock),2};
lastSampleRate = sampleRate;
spec.maximumBlockSize = samplesPerBlock;
spec.sampleRate = sampleRate;
spec.numChannels = getTotalNumOutputChannels();
lowPassFilter.prepare(spec);
lowPassFilter.reset();
```

Now we are ready to actually apply the filtering algorithm. In the *processBlock()* method we update the filter coefficients and we apply the filtering to the audio buffer with the *dsp::AudioBlock* class template. Then we update the frequency and quality factors taken from the *AudioProcessorValueTreeState.* The process method will read from the input, process it and replace the content in the input buffer (then streamed to the output).

```
juce::dsp::AudioBlock <float> block (buffer);
float freq = *apvts.getRawParameterValue(”FREQ”);
float quality = *apvts.getRawParameterValue(”Q”);
*lowPassFilter.state = *juce::dsp::IIR::Coefficients<float>::makeLowPass(lastSampleRate,
freq, quality);
lowPassFilter.process(juce::dsp::ProcessContextReplacing<float> (block));
```

8.4 Boxes

To select the type of effect (distortion, overdrive, reverb, bypass) two boxes have been created for each stage. In the same way, two boxes have been created to select the type of training. The implementation has already been discussed in *paragraph 7.5* in the description of the function *selectFirstEffectFunction()* and the like. All box properties have been set in *PluginEditor.cpp.*

8.5 Background and style settings

Two images (pedal's silhouette and logo) have been loaded and used for the plugin thanks to the help of two variables image and *imageComponent*:

```
image1 = juce::ImageFileFormat::loadFrom(juce::File::File(”C:/.../logo.png”));
    if (image1.isValid()){
        imageComponent1.setImage(image1);
    addAndMakeVisible(&imageComponent1);
    }
```

Shades of red have been inserted into the *paint()* function. The editing of knobs and boxes has been implemented thanks to the methods of the *getLookAndFeel()* class on top of the constructor:

```
getLookAndFeel().setColour(juce::Slider::thumbColourId , juce::Colours::darkred);
getLookAndFeel().setColour(juce::Slider::rotarySliderFillColourId , juce::Colours::red);
getLookAndFeel().setColour(juce::Slider::rotarySliderOutlineColourId ,juce::Colours::grey);
getLookAndFeel().setColour(juce::ComboBox::backgroundColourId , juce::Colours::grey);
getLookAndFeel().setColour(juce::ComboBox::arrowColourId , juce::Colours::darkred);
getLookAndFeel().setColour(juce::ComboBox::outlineColourId , juce::Colours::grey);
getLookAndFeel().setColour(juce::ComboBox::textColourId , juce::Colours::darkred);
```

The position of the edited elements in the plugin has been established in the *resized()* function of the *PluginEditor.cpp*.

9 Technical evaluation

9.1 LSTM model

In this work it was decided to use the LSTM architecture after having viewed the results of the state-of-the-art literature.
In [2] a comparison was made between LSTM and the other most suitable blackbox models to emulate real analog pedals or tube amplifiers. One network from each class of networks has been trained using the same data and loss function. The LSTM with 32 recurrent units outperforms all other models in terms of objective quality measures.

Table 1: Black-box architecture comparison from [2] - **LSTM:** Sequence-to-sequence LSTM with 32 recurrent units; **WaveNet:** number of channels = 12, dilation depth = 10, kernel size = 3; **Convolution LSTM:** number of channels = 35, stride = 4, kernel size = 3; **Shallow TCN:** number of channels = 32, number of blocks = 4, stack size = 10, dilation growth = 10.
RTF = Processing Time RT constraint . RTF lower than 1 is required for real-time operation; **MSE** = Mean-Squared Error; **ESR** = Error-to-Signal Ratio; **MAE** = Mean Absolute Error; **STFT** = Short-Term Fourier Transform.

Architecture	RTF	MSE	ESR	MAE	STFT
RNN (LSTM-32) [33]	0.51	0.0040	0.0244	0.0378	0.5952
CNN (WaveNet) [25]	0.35	0.0703	0.4337	0.1359	0.6542
Hybrid (Conv-LSTM) [35]	3.25	0.0069	0.0423	0.0530	0.6937
CNN (Shallow TCN) [28]	0.14	0.3190	2.1371	0.4510	1.2348

In [1] the single layer RNN LSTM model presented requires much less processing power and time to run in comparison to a WaveNet-like convolutional neural network and can be run in real time. The accuracy of the RNN model was also comparable to or better than the WaveNet, depending on the device being modelled. LSTM network is recommended also over the GRU - the second method proposed in [1] - as tests indicate that LSTM generally achieved higher accuracy.

9.2 Training

The trainings have been done for 1000 epochs obtaining the following results. The best results were obtained for overdrive training. In general an improvement is observed with increasing the hidden size from 32 to 96 as described in [1]. Furthermore, the results confirm that the 3-minute input is generally

Table 2: training results: total loss and validation loss for each training

Effect:	Distorsion	Overdrive	Reverb
Losses:	Tot - Val	Tot - Val	Tot-Val
1:00 min, hs = 32	0.1641 - 0.1658	0.1074 - 0.0970	0.674 - 0.667
1:00 min, hs = 96	0.1415 - 0.1453	0.0333 - 0.0455	0.643 - 0.669
3:30 min, hs = 32	0.1216 - 0.1277	0.0324 - 0.02511	0.645 - 0.621

better than the 1-minute one because it presents more techniques and is trained to a greater variety of sounds. Therefore results were found in line with expectations: the worst results for each effect are given by training with hidden size equal to 32 and one minute input, the best with hidden size equal to 96 and 3 minute long input. Distortion didn't give as good results as overdrive. This may be due to an inaccurate alignment between the input and output files. As already mentioned, the LSTM network is not suitable for effects such as reverb. Throughout the training the reverb losses have always been close to those values.

9.3 Code and RTNeural

The main reason why RTNeural is used in this plugin is that it is intentionally optimised to perform better for smaller layer sizes like in our case. In [3] the results of the performance benchmarks show that RTNeural out-performs PyTorch for the smaller layer sizes, although Py-Torch shows less change in performance as the layer size grows, implying that it will be better able to "scale up" for larger layer sizes. RTNeural backends perform best for different layer types and sizes.

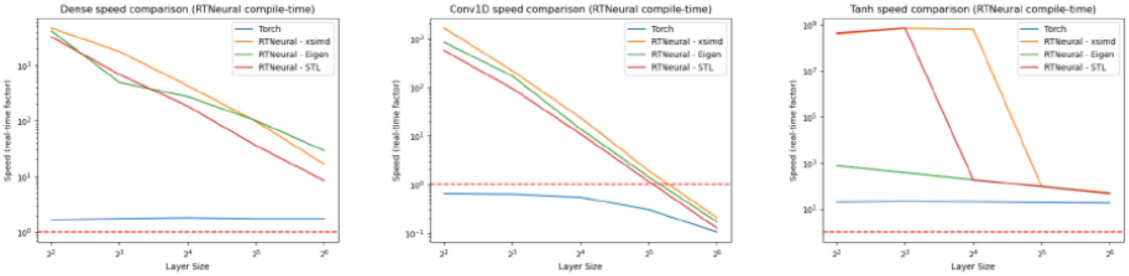

Figure 3: Speed comparison between RTNeural and Torch for dense, Conv1D, and tanh layers, using the RTNeural compile-time API. Image taken from [3]

10 Possible improvements

The plugin still has room for improvement. With better models it will be possible to add a third or even more stages for additional effects. It will also be possible to add more complicated effects in the same way as previously seen once new neural network architectures are selected. Another possible improvement will be to manage the gains of the models differently in the various stages. A possible solution is to have each stage processed by a single *processBlock()* using the *audioProcessorGraph* class provided by JUCE. As far as the GUI is concerned, one could think of adding more knobs to implement other features. A more appealing GUI's design is necessary.
In future improvements it would be beneficial to completely eliminate the slight delay between contact with the strings and the actual output.

11 Conclusions

The LSTM GEAR plugin proved to be fully functional and achieved the goals set by the author. The plugin can easily load a model in *.json* format with less computational effort thanks to RTNeural. Each model can be chosen through a user friendly and intuitive GUI. In the same way it is possible to choose the type of training carried out for that model. The two-stage effect chain works with any combination of models. With further improvements LSTM GEAR can achieve better results and a wider range of effects and neural network architecture.

12 Links

- https://github.com/Marco9723/STMAE-project
- https://www.youtube.com/watch?v=GoWgHEdeOak

References

[1] A. Wright, E.-P. Damskägg, and V. Välimäki, “Real-time black-box modelling with recurrent neural networks,” 09 2019.

[2] T. Vanhatalo, P. Legrand, M. Desainte-Catherine, P. Hanna, A. Brusco, G. Pille, and Y. Bayle, “A review of neural network-based emulation of guitar amplifiers,” *Applied Sciences*, vol. 12, no. 12, 2022.

[3] J. Chowdhury, “Rtneural: Fast neural inferencing for real-time systems,” 2021.

[1] [2] [3]