[NEMO] - HARNESSING NEURAL NETWORKS FOR ANALOG GUITAR EFFECTS REAL-TIME SIMULATION

Ferdinando Terminiello

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano
Piazza Leonardo Da Vinci 32, 20122 Milano, Italy
[ferdinando.terminiello]@mail.polimi.it

ABSTRACT

The paper investigates the potential of neural networks to replicate analog effects, specifically focusing on circuits for distortion effects.This approach operates as a black-box method, deriving the parameters of the neural model exclusively from input and output data. For this objective, the paper introduces NEMO (Neural Effects for Musical Output), a plugin leveraging neural network capabilities to seamlessly emulate a wide array of distortion effects and potentially for any musical output. NEMO is a user-friendly plugin designed for effortless integration of any trained model across various DAW platforms and for real-time purposes.

Index Terms— Neural Network, Plugin,Audio, Audio Effects

1. INTRODUCTION

As music production shifts towards digital platforms and DAWs become the norm, there's an increasing demand for digital effects that authentically replicate the rich and warm tones of traditional analog equipment.Specifically, in the domain of guitar effects, virtual analog modeling has ventured into diverse and innovative approaches. This article provides a comprehensive explanation of the NEMO plugin, specifically designed to emulate analog effects for electric guitars. The black box methodology eliminates the need to understand the intricate physical and circuitry details of the analog effect. This approach offers significant benefits in terms of faster implementation and broader applicability of the plugin. Users can curate a dataset based on the specific pedal they aim to replicate, train the model accordingly, and seamlessly incorporate it into the plugin, all while leveraging recordings from their authentic analog effects.

Below is a concise overview of the two types of audio effects to consider. The second chapter provides an in-depth discussion of the plugin and its implementation. In the third chapter, we explore the training process for the neural models. Finally, the last chapter presents the plugin's results and performance.

1.1. Guitar Effects

Within the realm of sound processing for the guitar, comprehending the diverse categories of available sound effects and their temporal impact on sound is crucial. This paper directs its attention solely towards the implementation of time no variance effects. This decision arises from the recognition that the architecture of time no variance effects often presents heightened complexity and difficulty in real-time management.

1.1.1. Time-varying audio effects

Effects classified as "time-varying" are those that impact sound over time, introducing variations and temporal modulations. These effects add depth and dynamism to the guitar sound, creating rich and intriguing atmospheres. Examples of "time-varying effects" include delay, flanger, and phaser. Delay creates a reverberation of the original sound that repeats over time, while flanger and phaser introduce modulations in the phase and frequency of the audio signal, creating distinctive effects that vary over time. These sounds are typically achieved by modulating the signal with a Low Frequency Oscillator (LFO). The LFO generates a low-frequency periodic signal that is used to control parameters of the audio signal, such as delay time, filter frequency, or modulation depth. This modulation adds rhythmic variations to the sound, enhancing its depth and complexity over time. The use of an LFO allows for dynamic manipulation of the guitar sound, providing musicians with a versatile tool for creative expression.

1.1.2. No-time-varying audio effects

In contrast to "time-varying-effects," sound effects categorized as "no-time-varying effects" alter sound without introducing significant variations over time. These effects may include changes to tonal characteristics of the sound, compression level, equalization, and distortion. Despite their temporal staticity, these effects are essential for modeling and optimizing the sound of a guitar. Examples of "no-time-varying effects" encompass equalization (EQ), compression, distortion.

2. THE NEMO PLUGIN

The NEMO Plugin aims to accurately emulate the "no-time-varying effects" of analog guitar pedals. This application is developed using JUCE [1], a widely used C++ framework for audio applications. The neural models were trained separately using PyTorch in a Python environment and then integrated into the plugin using a library called RTNeural [2]. As we will see in the results, this integration method enables significantly better performance compared to simple integration with Torch.

2.1. Real Time Implementation with JUCE

To ensure good real-time performance of the plugin, in addition to using the RTNeural library, we decided to utilize the OSC protocol [3] to communicate parameter changes of knobs or other values. The OSC protocol, short for Open Sound Control, is a network protocol that facilitates communication between musical devices and

software. It's designed to be lightweight, fast, and flexible, enabling the real-time transfer of multimedia data over Ethernet or the Internet. OSC is widely used in the realm of music and audio to send and receive commands, parameters, and audio signals between different devices and applications. Its flexibility and ability to handle multimedia data make it an ideal choice for real-time control and communication in musical and audio contexts.This choice also allows us to make further future developments if, for example, we wanted to control the plugin with external hardware.

Figure 1: A screenshot of NEMO's GUI

2.2. Integration of neural models in C++

To integrate in C++ the previously trained models in PyTorch, we leverage the RTNeural library. To do this, we need to convert our model into an appropriate JSON file. This operation is performed by the following Python script

Listing 1: Save model in JSON file

```
def save_model(self, file_name, direc=''):
    model_info = {
        'model_data': {
            'model': 'SimpleModel',
            'input_size': self.lstm.input_size,
            'output_size': self.dense.out_features,
            'num_layers': self.lstm.num_layers,
            'hidden_size': self.lstm.hidden_size,
            'bias_fl': True
        }
    }

    model_state = self.state_dict()
    for key in model_state:
        model_state[key] = model_state[key].tolist()

    model_info['state_dict'] = model_state

    miscfuncs.json_save(model_info, file_name, direc)
```

This code allows saving all the relevant information of the model, such as the model type, the number of layers, hidden size, input and output size, and the values of weights and biases. After the following JSON file is read, it is processed in C++ by the following code snippets.

Listing 2: Load JSON file in C++

```
void RT_LSTM::load_json(const char* filename)
{
    // Read in the JSON file
    std::ifstream i2(filename);
    nlohmann::json weights_json;
    i2 >> weights_json;

    // Get the input size of the JSON file
    int input_size_json = weights_json["/model_data\
■■■■/input_size"_json_pointer];
    input_size = input_size_json;

    // Load the appropriate model
    if (input_size == 1) {
        set_weights(&model, filename);
    }
    else if (input_size == 2) {
        set_weights(&model_cond1, filename);
    }
    else if (input_size == 3) {
        set_weights(&model_cond2, filename);
    }
}
```

2.3. GUI

The NEMO GUI (Fig. 1) is very simple and user-friendly. Here's an updated list of features:

- *3 knobs dedicated to equalization*: these knobs enable users to adjust the frequency response of the audio signal. Typically, they control bass, midrange, and treble frequencies, allowing users to boost or cut specific frequency ranges to tailor the sound to their preferences.
- *1 knob for delay*: this knob controls the amount of delay effect applied to the audio signal.
- *1 knob for adjusting reverb*: this knob adjusts the amount of reverberation or the spaciousness of the audio signal.
- *2 knobs (gain and level)*: they control the conditioning parameters of the model associated with the analog pedal.

- *A dropdown menu*: this menu allows users to select and switch between different neural models that have been loaded into the plugin.

3. NEURAL NETWORKS AND CONDITIONING TRAINING

In this chapter, the paper aims to explain in detail how the neural networks were trained and the datasets used. The entire training process was developed in PyTorch.

3.1. Neural Networks Models

For our problem, we mainly considered three different types of architectures [4]:

- *Convolutional Neural Networks (CNNs)*: CNNs are commonly used for image recognition tasks but can also be applied to sequential data. They consist of convolutional layers that apply filters to input data, capturing spatial patterns and hierarchies of features.
- *Long Short-Term Memory (LSTM)*: LSTM networks are a type of recurrent neural network (RNN) designed to capture long-term dependencies in sequential data. They have a memory cell that can maintain information over time and gates to control the flow of information, allowing them to learn and remember patterns in time series data.
- *Gated Recurrent Unit (GRU)*: GRU networks are another type of recurrent neural network similar to LSTM but with a simpler architecture. They also have mechanisms to capture long-term dependencies in sequential data but use fewer parameters compared to LSTM, making them computationally more efficient.

These architectures were selected for their efficacy in capturing temporal dependencies and patterns in sequential data, which is crucial for modeling audio signals in our context, particularly for no-time-varying effects.

Furthermore, experiments were conducted for "time-varying" effects primarily based on the work of [5], whose architecture is depicted in Figure 2. The main challenge lies in making this architecture usable for real-time purposes.

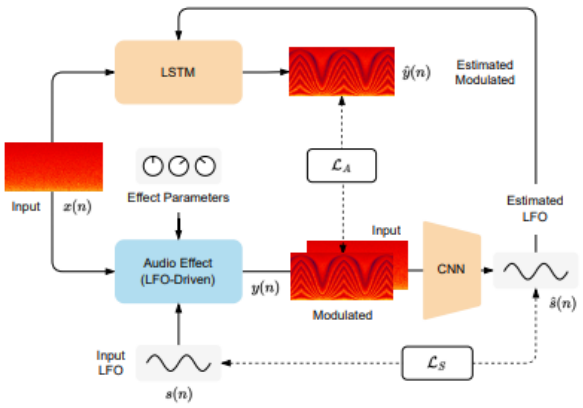

Figure 2: Neural Network Architecture for "Time-Varying" Effects

The input dimension of these networks is equal to 1 if we consider replicating the effect for fixed knob positions. Otherwise, the number of inputs increases as the number of knobs to replicate grows.The output dimension is always 1 and corresponds to the processed audio sample.

In our model architectures, we utilize two key loss functions: ESR (Error Suppression Ratio) and DC (DC offset).

- *E*SR measures the model's ability to suppress errors between the output and target signals. It computes the ratio of the squared difference between the mean output and target signals to the mean energy of the target signal.
- *D*C quantifies the reduction in the dynamic range of the audio signal achieved by the model. It calculates the squared difference between the mean output and target signals across all dimensions, then normalizes this loss by the mean energy of the target signal.

The two loss values are combined using a weighted sum. Specifically, the weight assigned to ESR is 0.75, whereas the weight for DC is 0.25. The validation loss, used for model evaluation, is computed similarly to the training loss.

3.2. Dataset

The dataset was entirely "handcrafted", originating from a recording of a Fender Telecaster guitar using a Behringer UMC202HD [6] audio interface, thus obtaining a clean signal.Subsequently, the clean signal was processed using the well-known distortion pedal, the Electro-Harmonix Big Muff Pi (Fig. 3) which features three knobs: one for volume, one for tone, and one for sustain. For convenience, we'll refer to the dataset used for conditioning as MuffCond and the other simply as Muff.

The Muff dataset contains only the clean signal (input) and the processed signal (target) with all pedal knobs set to 0.5.

The MuffCond dataset, on the other hand, comprises all signals obtained from pedal combinations set at four values: 0, 0.35, 0.7, and 1.

3.3. Experiments

To assess the different models and determine the optimal hyperparameters, a series of experiments were conducted. The experiments involved systematically evaluating each selected neural model individually and then exploring variations in the hyperparameters.During the experiments, the following hyperparameters were varied: batch size, hidden size, number of layers, and learning rate.The training comprised 300 epochs and was executed using the following CPU and GPU specifications:

- CPU: Intel Xeon E5-1607 v2, 4 Cores @ 3 GHz, 31 GiB RAM
- GPU: 1x GTX 1080 (2560 CUDAs @ 1607MHz, 8 GiB VRAM)

Additionally, it's noteworthy that each training session lasts approximately 1 hours for the Muff Dataset. For the MuffCond Dataset, each training session lasts approximately 3 hours. The training process for the MuffCond Dataset was performed using a public script available in this `https://github.com/GuitarML/Automated-GuitarAmpModelling/blob/main/ExampleOfDifferentModelTraining.ipynb`.

Figure 3: A picture of the Big Muff Pi

4. TECHINICAL EVALUATION

This section aims to offer a technical evaluation of two primary aspects of our problem. The first aspect entails the selection of the neural network model along with its hyperparameters, while the second aspect concerns the plugin's performance in terms of signal processing speed. To assess these aspects, we address the following questions:

4.1. Q1: Which architecture best reproduces "no-time-varying" audio effects?

Around 40 training sessions were executed for each architecture, with variations in different hyperparameters. The search process for optimal hyperparameters also requires less time due to the Bayesian approach employed, further optimizing efficiency. The findings revealed that LSTM and GRU architectures performed best, while CNN exhibited comparatively lower results.In particular, GRU appears to be the architecture that achieves the highest value of validation loss, as reported in Table 1.

	Lr	Bs	Hs	Ly	Val. Loss
CNN	0.001	32	100	1	0.7
LSTM	0.001	16	100	1	0.046
GRU	0.001	32	100	1	0.03

Table 1: Table showing the best validation loss value for each architecture along with the respective hyperparameters

Moreover, due to its simpler architecture compared to LSTM, the GRU model trains in a shorter amount of time.

4.2. Q2: How do different hyperparameters affect the model's performance?

To analyze the effect of hyperparameters, graphs referring only to the GRU are shown, as the same considerations can be made for the LSTM. The hidden size hyperparameter stands out as the most influential factor affecting model performance. This parameter determines the number of units in the hidden layer of the neural network architecture, influencing the network's capacity to capture and represent complex patterns in the data.The observed trend suggests that increasing the hidden size generally leads to improved performance. Larger hidden sizes provide the model with more capacity to learn and represent intricate relationships within the data, resulting in better predictive accuracy and generalization capabilities. This trend is evident from the graph in Figure 4 illustrating the relationship between validation loss and different hidden size values. As the hidden size increases, the validation loss tends to decrease, indicating enhanced model performance.

However, it is important to note that good performance can still be achieved with hidden size values equal to 20. This suggests that while larger hidden sizes may offer performance benefits, models with moderate hidden sizes can still effectively learn and represent the underlying patterns in the data.

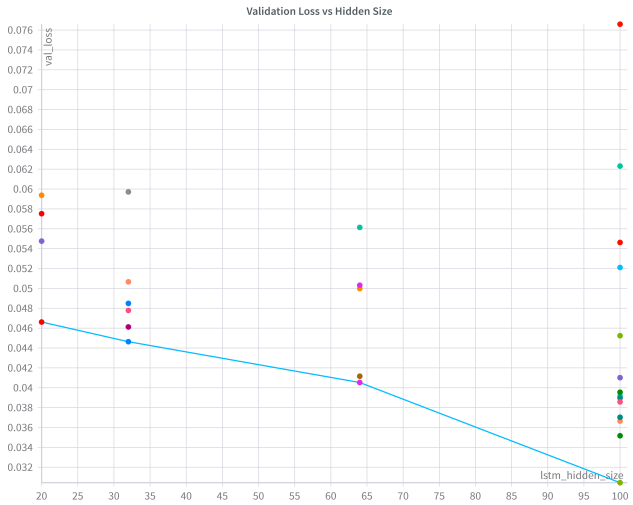

Figure 4: Graph illustrating the obtained values from various validation runs across different hidden sizes.

Another critical hyperparameter that significantly influences model performance is the batch size . The batch size determines the number of training samples used in each iteration during the training process.

A batch size that is too high can lead to challenges such as memory constraints and slower convergence, potentially deteriorating performance. On the other hand, a batch size that is too small may lead to noisy gradients and inefficient training.

Through experimentation, it has been observed that a batch size ranging between 16 and 32 tends to strike a balance between computational efficiency and training effectiveness. This range allows

for efficient use of computational resources while ensuring stable and effective optimization during training.

Figure 5 provides a visual representation of how varying batch sizes impact the model's performance, demonstrating the diminishing returns associated with excessively large batch sizes and the potential instability associated with very small batch sizes.

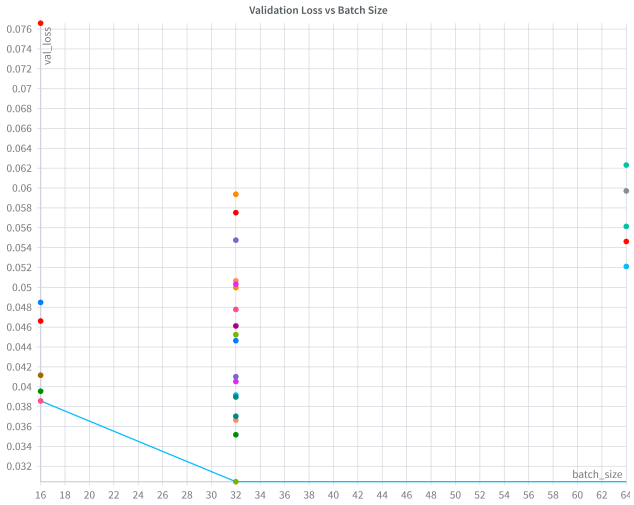

Figure 5: Graph illustrating the obtained values from various validation runs across different batch sizes

The learning rate plays a crucial role in the optimization process, governing the size of the steps taken during parameter updates. However, its direct impact on the model's performance is not as pronounced. Instead, the learning rate influences the convergence speed and stability of the training process. By adjusting the learning rate, practitioners can control how quickly the model learns from the training data and navigate the optimization landscape more effectively.Similarly, the number of layers in the neural network architecture does not seem to significantly enhance or impede performance. Therefore, it has been determined that setting the number of layers to 1 can ensure optimal real-time performance. This decision simplifies the architecture while still allowing for effective information processing and decision-making, especially in scenarios where real-time responsiveness is critical. By reducing architectural complexity, the model can streamline computations and maintain efficient processing without sacrificing performance quality.

4.3. Q3: Are there performance advantages in using RTNeural instead of Torch?

To assess the plugin's performance, the signal processing speed of the models has been defined as follows:

$$Speed = \frac{Number\ of\ samples}{Processing\ time} \tag{1}$$

The performance of the selected architectures, including LSTM, CNN, and GRU, was thoroughly evaluated and compared against both the Torch and RTNeural libraries. The analysis primarily focused on discerning trends in processing speed relative to the hidden size parameter. All tests were conducted on a device with the following specifications:

- Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
- RAM: 8.00 GB (7.80 GB usable)

All three metrics were graphically represented in Figures 6, 7 and 8. Notably, Figure 6, which focuses on LSTM, showcases a remarkable performance boost with RTNeural, particularly noticeable for hidden size values ranging from 10 to 40. The most substantial improvement was observed for a hidden size of 10, where performance surged by 600 times. This observation underscores the efficacy of leveraging RTNeural, especially in scenarios where processing speed is crucial. The significant enhancements achieved across various hidden size configurations highlight RTNeural's effectiveness in optimizing model performance. A similar trend is observed for the GRU architecture, as shown in Figure 7. Unlike LSTM, there is a slight decline in performance with RTNeural models, although the improvement with the RTNeural library remains notable. Even with the CNN architecture, the RTNeural library maintains its performance superiority, retaining its advantage even with higher hidden size parameter values. This consistency underscores the robustness of RTNeural across different neural network architectures and hidden size configurations.

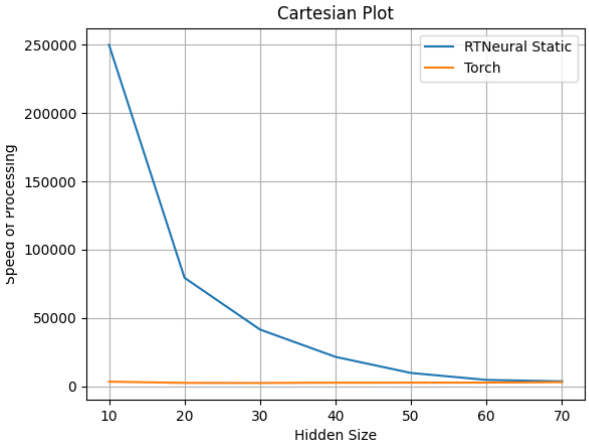

Figure 6: Speed Performance for LSTM

5. CONCLUSION

In conclusion, the present study has demonstrated the promising effectiveness of neural networks in replicating analog effects for the guitar through the NEMO (Neural Effects for Musical Output) plugin. Through the implementation of various neural architectures, including Convolutional Neural Networks (CNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU), we have identified that the LSTM and GRU architectures have proven to be particularly effective in reproducing "no-time-varying" audio effects.The results indicate that using the RTNeural framework for integrating neural models into the plugin has led to significant performance improvements, especially in terms of signal processing speed.In addition, data analysis has highlighted the importance of certain hyperparameters in model training, such as hidden size and batch size.

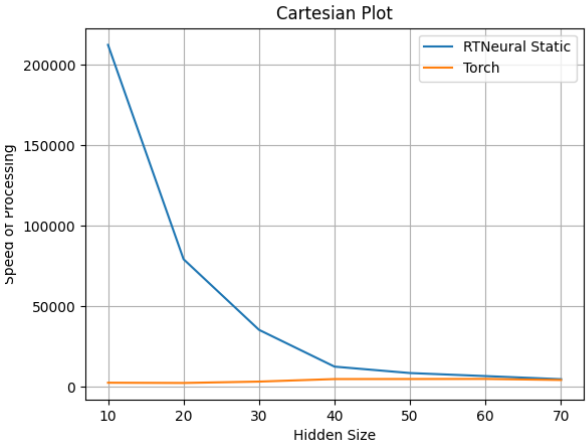

Figure 7: Speed Performance for GRU

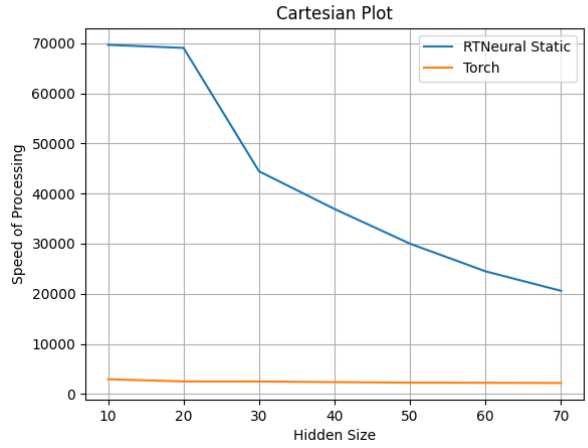

Figure 8: Speed Performance for CNN

5.1. Future Improvements

These could be some ideas to implement in the future:

- Integrate models for ”time-varying-effects” as well.
- Create a database with all the pre-trained models.
- Integrate various effects with the ability to create new sounds.

6. REFERENCES

[1] Juce Documentation, https://juce.com/learn/documentation/.

[2] J. Chowdhury, “Rtneural: Fast neural inferencing for real-time systems,” *arXiv preprint arXiv:2106.03037*, 2021.

[3] OpenSoundControl.org, ”What is OSC?” https://opensoundcontrol.stanford.edu.

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

[5] C. Mitcheltree, C. J. Steinmetz, M. Comunità, and J. D. Reiss, “Modulation extraction for lfo-driven audio effects,” 2023.

[6] Behringer UMC202HD https://www.behringer.com/product.html?modelCode=P0BJZ.