Predicting Audio Sample Parameters in a Synthesizer

Esteban Gomez

06/13/2024

Table of Contents

1. Abstract 1
2. Introduction 4
3. Data Wrangling & EDA 4
4. Pre-Processing & Training 6
5. Modeling & Discussion 9
6. Conclusion 12

Abstract

Sound design and music production have become very popular in recent years, driven by technological advancements that have made them accessible to a broad audience. This report focuses on developing a neural network model that predicts the parameters to recreate an audio sample in an extremely popular synthesizer by predicting its parameter values. We collected audio samples and corresponding parameter data from Serum presets, processed these samples into Mel Spectrograms, and trained a neural network to predict the parameters based on these spectrograms. Despite achieving a low Mean Squared Error (MSE) during training, the model's predictions showed notable discrepancies when applied to test data, highlighting areas for further refinement. This study outlines the data collection, preprocessing, and training processes, discusses the model's initial performance, and suggests improvements to enhance accuracy and reliability in audio sample parameter prediction.

Introduction

Sound design and music production have grown tremendously in the past several years, with technological advancements making them accessible to everyday consumers. The sound design market includes platforms and websites that have established themselves as major players, offering an extensive selection of sample packs tailored to diverse music genres. Industry leaders such as Splice, Loopmasters, ADSR Sounds, and Cymatics provide a vast library of samples, loops, and presets. These platforms commonly operate on a subscription model or offer individual purchases, catering to the needs of both amateur and professional music producers. As a music producer, having various options is critical to achieving a unique sound, and many music producers, such as myself, depend on the art of sound design to achieve this.

Sound design is a whole art in and of itself, however. There are many disciplines that a music producer already must conquer. That is why there are so many sample and preset libraries available for music producers; that way, they can focus on the creativity of making music through the arrangement of sounds rather than spending hours attempting to create a sound they have in their mind only to produce a sound of much lower quality. Despite the insurmountable amount of effort needed to master sound design, there are many simple concepts that music producers can quickly learn to slightly tweak the sounds they have at their disposal to fit the sound to their song. That is why preset libraries have become so popular; a producer can take a sound that has already been created and slightly tweak it to meet their needs. The issue, however, becomes that presets are only a fraction of the libraries offered by websites such as Splice, Loopmasters, and more. Samples comprise most libraries; one cannot edit the audio source since they are recorded audio. All one can do with a sample is add effects or warp the sound.

This project aims to train a neural network model that can predict the parameters needed to recreate an audio sample. We will render audio samples from the famous synthesizer, Serum, taken from the presets available to create a dataset to train the model. We will save the parameter values for the presets in combination with the audio samples associated with the preset.

Data Wrangling & EDA

The first step in the process is to gather the data and ensure it is clean. To create data, we recorded audio samples from Serum and saved the parameters that generated the audio. We needed a Digital Audio Workstation (DAW) to access Serum and retrieve the parameters set for each Serum preset. Reaper is the best DAW to implement the process, as it allows for nearly unlimited custom processes. We loaded Serum into Reaper, and to generate the audio samples, we recorded a 2-second audio clip of the preset being played for one beat at 120 beats per minute. The functionality that allows Reaper to be so customizable is the Reaper Scripts feature. Reaper Scripts allows for writing custom code to automate processes as one makes music. We have written the following code in Python and ran the script in Reaper to extract the parameters from all the Serum presets we save.

The following script was created to automate the rendering process of the presets:

```
import os

vst_name = 'VSTi: Serum(Xfer Records)'
render_path = '..\\data_audio'

# Function to set a preset from a file to Serum
def set_preset(idx):
    track = RPR_GetTrack(0, 0)  # Assuming Serum is on the first track
    fx = 0  # Assuming Serum is the first FX on the track
    preset_index = idx  # Index of the preset you want to load

    RPR_TrackFX_SetPresetByIndex(track, fx , preset_index)

def get_total_presets(track, fx):
    # Initialize a variable to hold the number of presets
    num_presets = 0

    # This tuple will hold the current preset index, the total number of presets,
    # and whether the retrieval was successful (True/False)
    success, track, fx, num_presets = RPR_TrackFX_GetPresetIndex(track, fx, 0)

    # The function returns -1 if there are no presets, so check for that
    if num_presets == -1:
        return 0  # If there are no presets, return 0
    else:
        return num_presets

# Function to render the recorded audio to a file
def render_to_file(render_path, file_name):
    full_path = os.path.join(render_path)
    # Ensure the path is in the format REAPER expects
    formatted_path = full_path.replace('\\', '\\\\')

    # Set the render path
    RPR_GetSetProjectInfo_String(0, 'RENDER_FILE', formatted_path, True)

    # Set the render path
    RPR_GetSetProjectInfo_String(0, 'RENDER_PATTERN', file_name, True)

    # Command ID for rendering the project automatically
    RPR_Main_OnCommand(42230, 0)
```

```
# Main script execution
def main():

    track = RPR_GetTrack(0, 0)  # Get the first track
    fx = 0  # Index of Serum, assuming it is the first FX on the track
    total_presets = get_total_presets(track, fx)

    for idx in range(total_presets):
        # Load the preset into Serum
        set_preset(idx)

        # Get the preset name
        success, preset_name = get_preset_name(track, fx)

        files = os.listdir(render_path)

        # Adjust this line to match the file extension you are using
        file_name_with_extension = f"{preset_name}.mp3"

        files_lower = [file.lower() for file in files]

        if success:
            if file_name_with_extension.lower() not in files_lower:
                # Proceed if we successfully retrieved the preset name
                render_to_file(render_path, preset_name)


if __name__ == "__main__":
    main()
```

Once the data is gathered, we load it into our Jupyter notebook, combine the parameters for each preset and the preset file name, and combine both into a DataFrame. Upon inspection, all the parameter values fall between 0 and 1. That is expected, as we generated the parameter values by normalizing them. There was only one row with missing data, which we removed with no cause for concern since it was just one row of data.

Pre-Processing & Training

In the next step, we preprocessed the data and trained the model. The data preprocessing converts the audio samples into spectrograms, specifically Mel Spectrograms. We convert the audio into spectrograms because deep learning models do not take raw audio as input. Instead, we turn the audio into a picture representing the sample, which becomes similar to an image classification problem. We create the spectrograms using Fourier Transforms, breaking down the audio into all the frequencies it contains and portraying the amplitude of the audio file at specific frequencies with color intensity. That way, three dimensions are used to visualize the audio sample: time, frequency, and amplitude.

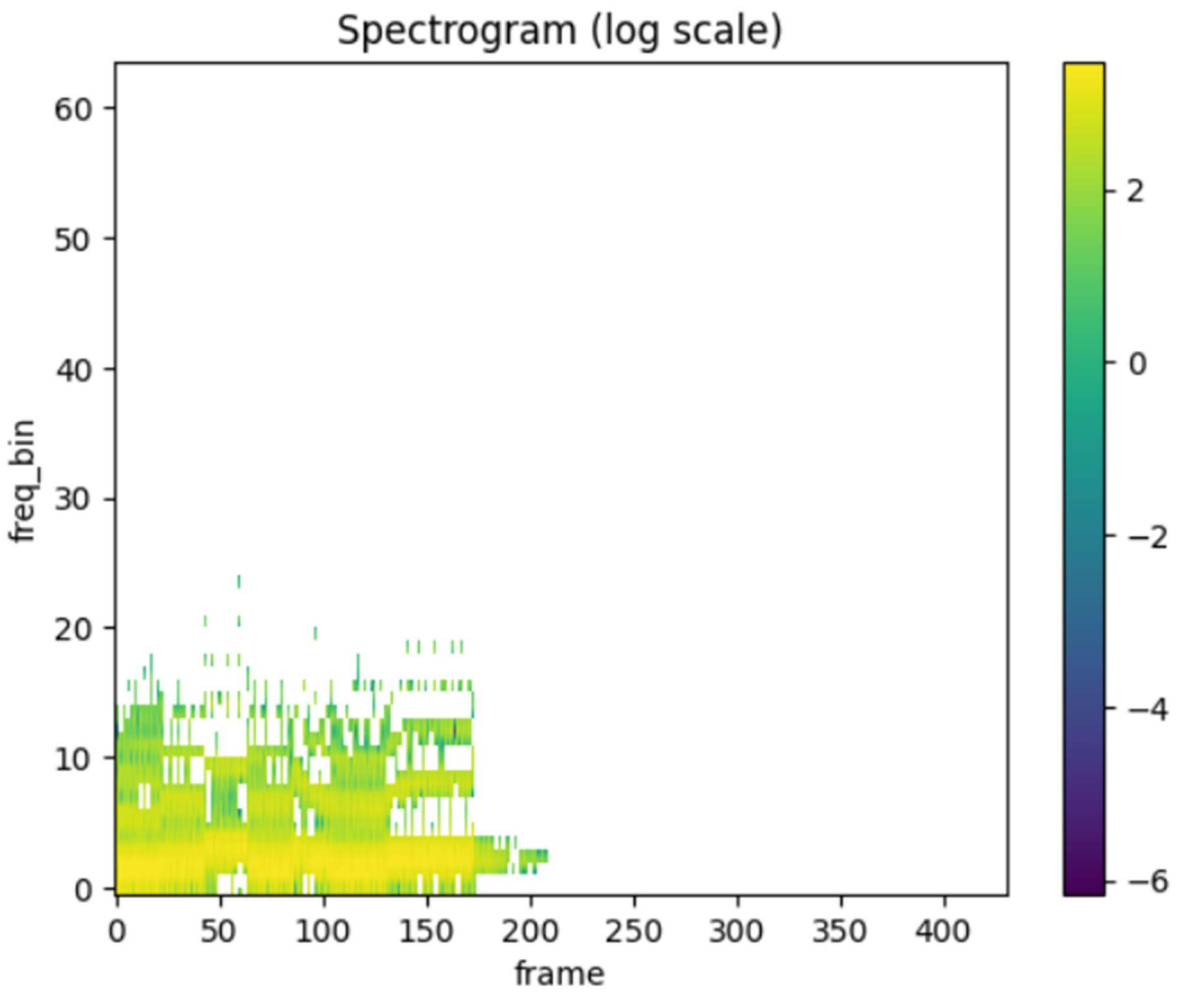

Figure 1. Mel Spectrogram of an audio sample.

Here, we visualize the spectrograms. Notice how the frequencies are predominantly in the bottom, indicating that this audio sample is bass-heavy. The frame displays the length of the audio. Recall that all the samples are two seconds long. Even though the audio length is two seconds, the samples may not occupy that entire time because some sounds may be more like a string with sustained notes and others like a pluck, such as the second figure, which only occupies the frames between 0 and 50.

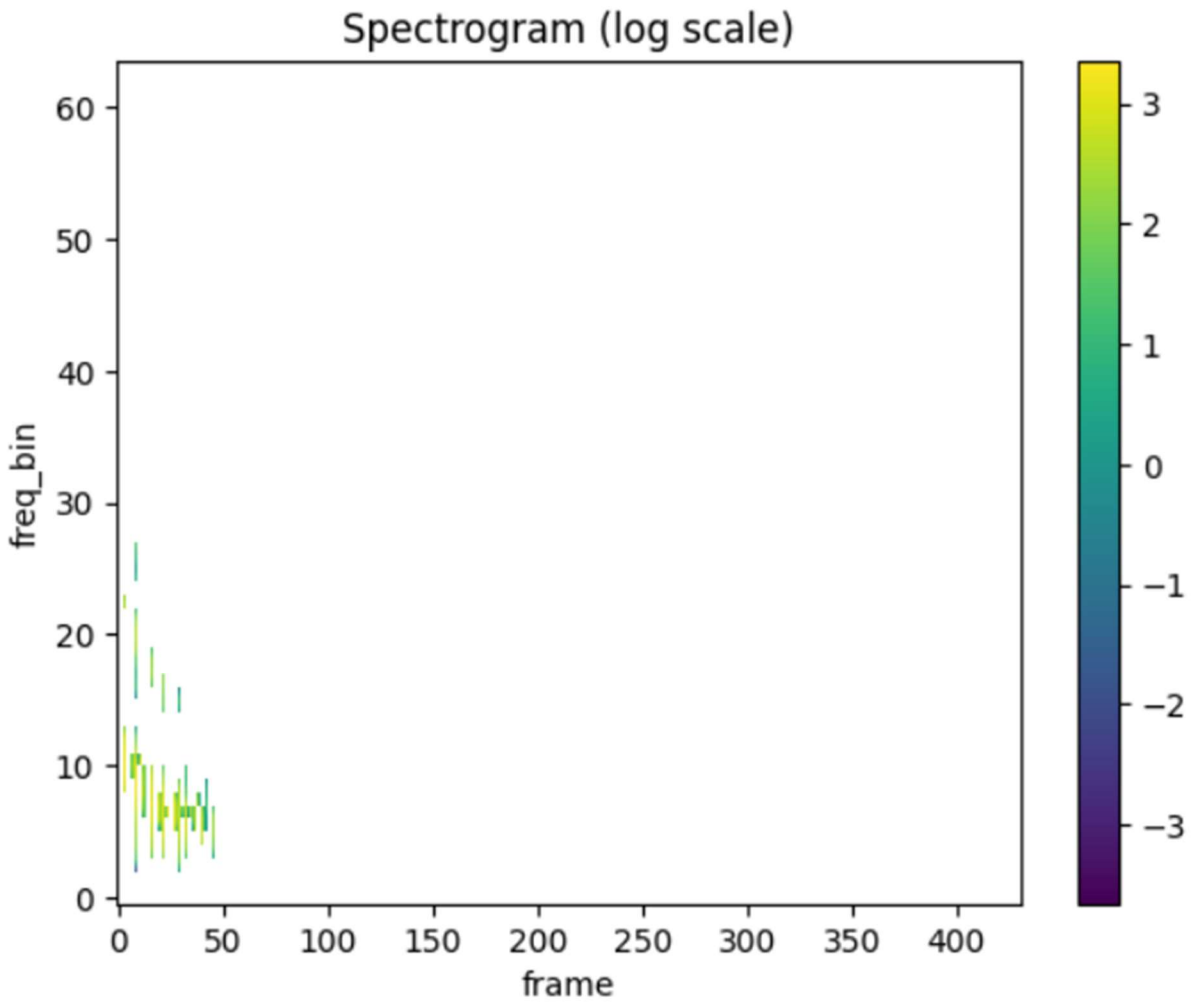

Figure 2. Mel Spectrogram of an audio sample with a shorter length.

Upon processing the entire data and beginning the training loop, an issue arose where a batch in the data contained missing values. We had to skip the batch and continue without the batch in each epoch of training. The training loop ran for 175 epochs, and despite skipping one batch in each epoch, the Mean-Squared Error (MSE) resulted in 0.03, which is a fantastic score.

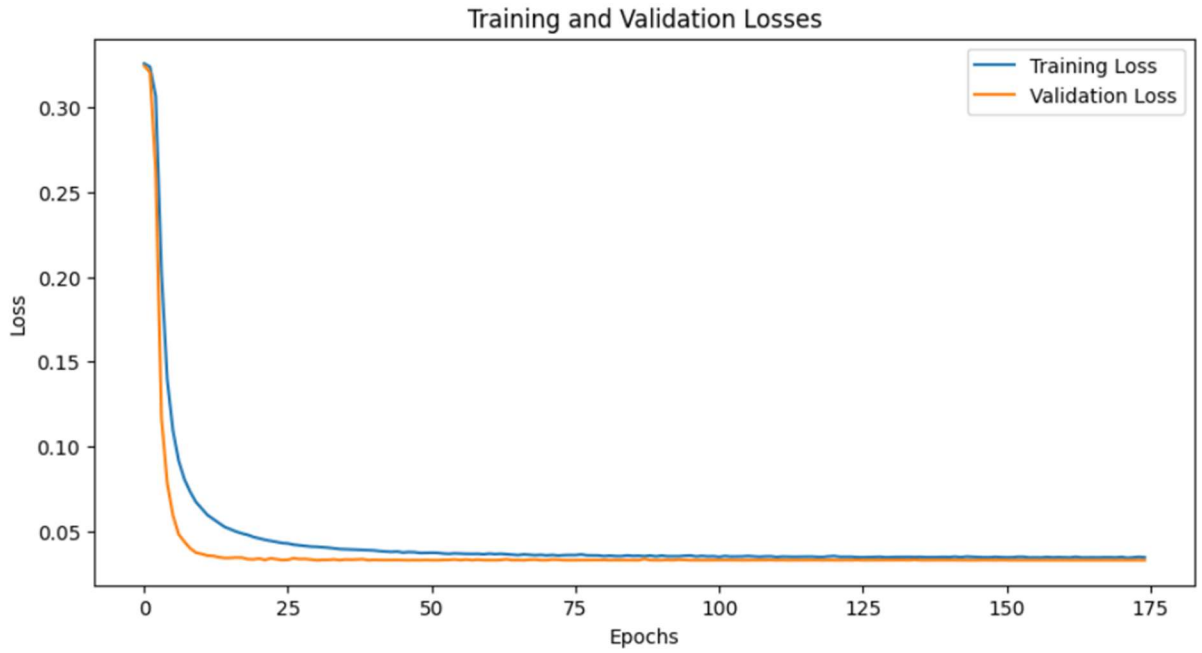

Figure 3. Training and Validation Losses (MSE)

As Figure 3 shows, an MSE of 0.03 can be reached with fewer epochs. Interestingly, the validation loss decreased much faster than the training loss.

Modeling & Discussion

Following the preprocessing and training, we proceed to identify the model's usefulness rather than simply using the MSE as a success metric. To test the model's ability, we select a preset at random, render the audio, convert it into a Mel spectrogram, and then allow the model to predict how to recreate the sound using Serum's parameters.

Upon arriving at a prediction and setting the parameter values predicted, the results indicated that the model's predictions were not as accurate as expected.

Figure 4. Model's predicted parameters loaded into Serum.

Figure 5. Preset selected for model's test.

Upon immediate inspection without delving into the parameter values themselves, which are set mainly by the knobs spread across the user interface of the synthesizer, it is worth noting that the oscillator shapes are different, which will already lead to big differences in sound. The oscillators used are the default oscillators for serum, which are saw waves.

Figure 6. Oscillators (saw waves) from the model's prediction.

Figure 7. Oscillators from the preset selected to test the model.

It makes sense that oscillators cannot, by default, be extracted as part of the parameters since they are customizable. The name of an oscillator does not determine its shape. The model is working with what it can to recreate the sound, and therefore, an opportunity for improvement will be finding a way of analyzing the oscillators used to create sounds.

Figure 8. Envelope from the model's prediction.

Figure 9. Envelope from the preset selected to test the model.

The envelope of the sound is something that the model should be able to pick out and has been reasonably accurate on. The purpose of the envelope is to shape the sound. The more complex task is figuring out the underlying wave shapes and the combination of effects needed to achieve a sample of audio. However, the shape is a relatively simple task to achieve. The ability of the model to recognize the sample's approximate shape means it is progressing towards identifying sounds' characteristics and how to re-create them. The small data set of about a thousand samples limited the model's ability; however, we can create a more robust and accurate model with a larger dataset.

Conclusion

This project developed a neural network model to predict parameters for recreating audio samples in Serum. Despite achieving a low Mean Squared Error (MSE) during training, significant discrepancies arose in test data predictions, highlighting areas for improvement. Factors such as limited data and inaccessibility to potentially important features contributed to these inaccuracies. Future efforts will focus on expanding the dataset and developing methods for extracting more valuable data.