FXNet: Convolutional Neural Network for Audio Effects Classification

Christopher Relyea
Boston University
crelyea@bu.edu

ABSTRACT

This project explores the creation of a multi-label convolutional neural network capable of identifying the presence of four audio effects, reverb, delay, chorus, and distortion, applied to musical instrument recordings. To train the model, over 4,000 audio samples were generated by applying all possible combinations of these effects to a diverse set of "dry" instrument audio files. These processed samples were converted to Mel spectrograms and used to train a CNN, which was gradually improved through hyperparameter tuning, audio representation experiments, and architecture changes. Results show high F1 scores for distortion and reverb, while chorus and delay proved more challenging due to their primarily temporal nature. The project highlights the effectiveness of spectrogram-based CNNs for audio classification and identifies key limitations in modeling time-domain effects. Potential improvements include adopting temporal models or raw waveform input to better capture subtle timing-based modifications.

1. INTRODUCTION AND BACKGROUND

1.1 Audio Effects

"Audio effects" are several popular discrete signal processing manipulations to modify a source music signal in ways to create interesting sounds. This project will deal with four such effects:

- Reverb – widens the sound, adds a long decay after the sound has finished playing. Sounds like we've put the instrument in a large room (or concert hall, or small studio, etc. depending on parameters)
- Delay – adds an echo to the sound; the sound repeats itself every x milliseconds, getting quieter each time until inaudible
- Chorus – adds some depth to the signal by playing a slightly displaced signal at the same time as the original. Usually described as a swirly, dreamy effect
- Distortion – clips the waveform of audio so it is less smooth/refined. Results in a sound that is aggressive, gritty, maybe more "ugly"

In this project, I aim to design a multi-label classifier that can recognize which of these four audio effects have been applied to an original signal.

2. COLLECTION OF TRAINING DATA

2.1 Finding Samples

To train the model, I needed to source many WAV files of musical instrument recordings that have been filtered through any combination of the four audio effects. I decided that the best way to create a large and accurately-labeled dataset would be to source "dry" (no effects added) instrument samples of many kinds, and then generate my own processed samples by applying effects to those unmodified audio files using a music creation software. After some research, I found that BandLab, a web-based music creation tool, provides a free and comprehensive database of uncopyrighted music samples meant to be manipulated in user-created songs. These samples, referred to as "loops," present a large collection of genres and styles, and can be sorted by instrument type in the online database.

In an effort to collect samples that represent a wide variety of sound types, I decided to source loops from each of four instrument types. I collected 40 samples each of guitar, voice, keyboards, and drums. I added 97 "miscellaneous" samples to this list to add more variety: these loops cover instruments or sounds outside of the four previous categories.

2.2 Rendering Audio Effects

I then had to find a way to render each sample with all of the audio effects I'd like to include in my classifier. Usually, this is easily done using a Digital Audio Workstation (DAW) and any number of audio effects "plugins" which can apply one of these four effects to a signal. These plugins often come included with a DAW, with an operating system, or can be downloaded from the Internet (many of them are free to use).

The aim was to take each of the 257 WAV files in my raw data and render each using every possible combination of the four effects types. The order in which effects are applied matters, and the decided-upon "industry standard" to produce the cleanest musical signals follows the order of distortion, chorus, delay, reverb. In this order, any of the four effects can be toggled on or off, so there are $2^4 = 16$ possible combinations for each input signal. $257 * 16 = 4112$ processed files should be rendered.

It would take quite a long time to do this the traditional way: dragging each sample into the DAW, adding the plugins for a single combination of effects, and rendering the project to another WAV file, following this process 16 times for each of the 257 input files. In an effort to make this process much faster, I discovered that a widely-used, lightweight DAW called Reaper has built-in scripting support through a Python library. I wrote a script that would, for each of the input files, load that file into Reaper and render 16 output files, each representing a new combination of effects. The output filename includes a binary string that acts as the label for that sample (ex. keys_9_0101.wav represents the tenth keyboard sample rendered with only chorus and reverb). Reaper's automation capability was a great help in the data collection process, and script completed its run in a few hours, likely tens of times slower than the rendering would have taken by hand.

For the effects themselves, each effect type made use of one of two plugins representing that effect. For example, when rendering a signal to use reverb, the script would randomly

choose one of two downloaded reverb plugins to load into the DAW. Additionally, to ensure some variability and to discourage the model from memorizing specific sounds or patterns, the parameters of each plugin were randomized on each use. When adding a delay plugin, for example, it would be quite unhelpful for the model to use the same delay speed in milliseconds for each use. By randomizing the parameters, some generated samples have fast delays, while some have slower ones. This not only combats overfitting to the training set, but also considers how these effects are used in the real world. Every recording artist or music producer tweaks effects parameters to fit the musical genre, style, or their personal preferences.

After the data collection was complete, I was left with 4112 labeled WAV files.

3. AUDIO REPRESENTATION AND PARAMETERS

3.1 Mel Spectrograms

While a WAV file is the most complete representation of an audio signal, they are quite large at higher sampling rate, and thus will cause a slow training process. In many code examples which use libraries like Keras or Pytorch, AI designers have opted to use a Mel spectrogram to represent an audio file.

Rather than encoding all of the sampling values of a raw audio signal, a Mel spectrogram sorts the signal into existing frequencies and investigates the change in these frequencies over time. Mel spectrograms are commonly used in audio machine learning applications, especially those involving classification of an audio excerpt into a category or label. Spectrograms can be displayed and interpreted as images, so they are a natural fit for CNNs.

Every audio effect I am investigating makes some changes to the frequency content or time content of an audio signal. Figures 1-5 show spectrograms for a raw signal, then the same signal rendered with each of the four effects.

The level of detail in a Mel spectrogram can be adjusted, resulting in a larger vector representation as input to the CNN. As detailed in my experimental process, I adjusted the N_MELS parameter, which results in a more detailed frequency content representation when increased, similar to the resolution of an image.

3.2 Duration and Trimming

The lengths of audio clips in my training data range from about three seconds to fourteen seconds, so it is important to decide on an audio clip length that will produce the best training results. My experiment results show an exploration of different audio lengths to evaluate the impact on larger input vectors on the success of the model.

Some clip lengths tested in my project are longer than the shortest clips in my training data. To accommodate for this, I used padding with empty values. For example, if the clip length I decided to use was 4 seconds, clips shorter than this would be filled in with 0s after the end of their audio signal, to achieve a uniform clip length for all samples.

A final parameter included in this category is a toggleable option to detect the onset of an audio signal. Every signal in

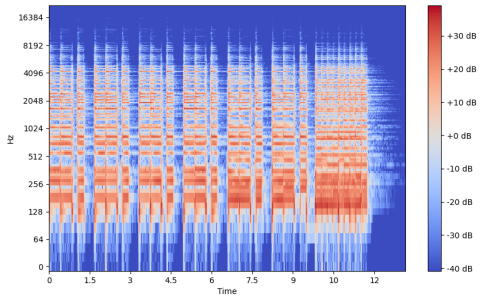

Figure 1: Mel Spectrogram: Dry Signal

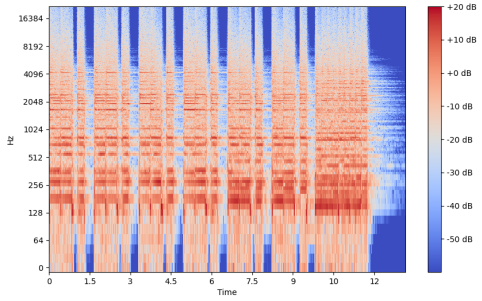

Figure 2: Mel Spectrogram: Signal with Distortion

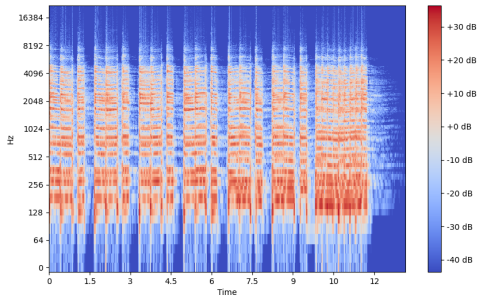

Figure 3: Mel Spectrogram: Signal with Chorus

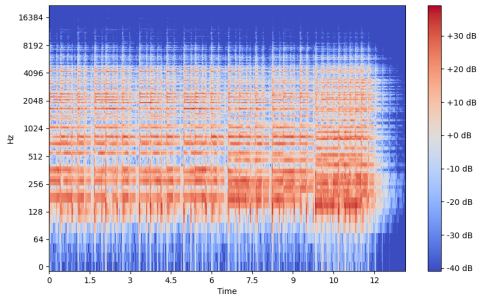

Figure 4: Mel Spectrogram: Signal with Delay

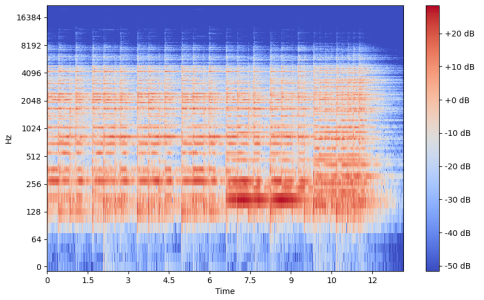

Figure 5: Mel Spectrogram: Signal with Reverb

the data set, for the purpose of ease of use in a music creation context, starts with some amount of silence. It would not be useful to train a model using silent signals which do not react at all to an audio effect. To accommodate this, I used Librosa's (my audio library of choice, as explained in Section 5) "trim" method, which automatically trims the silence off of a signal so that the start of the file will be the start of relevant audio. Again, if a signal is trimmed to shorter than the desired clip length for training, padding will occur.

3.3 Number of Channels

Most music you will hear on a regular basis contains either one or two channels of audio. "Mono" audio uses a single channel, while "Stereo" audio delivers two signals simultaneously in the left and right channels.

While distortion and delay do not behave differently in a stereo vs. a mono environment, chorus and reverb result in a "widening" of sound that spreads a signal further out between left and right channels.

Through experimentation, I will investigate the effect both mono and stereo input audio have on the model's success.

4. TOOLS AND SOURCES

4.1 Libraries

For the creation of the model and for training, I decided to use the PyTorch library [4]. I had originally planned on using Keras, but opted for PyTorch due to the lower-level control allowed by the library, since I wanted the option to edit more detailed elements of the training loop if necessary for my unique use case. I had also originally meant to run training on my local machine using a Nvidia GPU, and discovered that Keras had discontinued support for CUDA connection, making PyTorch a better option at that time (before opting for cloud-based computing).

PyTorch comes with its own audio processing library, but I opted for Librosa [2], another Python-based set of audio tools, for a number of reasons: Librosa provides more control over a larger set of audio processes, including pitch and time shifting (relevant for data augmentation), the aforementioned "trim" method, and a more detailed system for converting to Mel spectrograms. I also discovered quickly that the documentation for Librosa is easier to work with and much more extensive than Torch Audio, the PyTorch-based alternative.

4.2 Existing Code

I gathered inspiration and guidance from several code sources. To get a handle on the Torch code structure and an application to audio, I used an example provided by the library: a speech command classifier ([1]). While this example uses raw WAV audio instead of spectrograms, it is a good starting point to learn the basics of a machine learning library I was not initially familiar with in a context parallel to my project's mission.

To design the CNN itself, I made use of notes from class lectures as well as a GitHub repository defining a sound classification model in Keras ([3]). The resources from DS340 allowed me to quickly prototype a basic neural network based on image classification examples discussed in lecture, especially where network structure and hyperparameters are concerned, and the GitHub example was a great help in adapting the basic CNN structure for audio purposes.

5. RESULTS

5.1 Metrics Overview

As discussed previously, the following metrics for input audio signals and their respective possible values were considered:

- Audio duration: 2-7 seconds
- Number of channels: 1, 2
- Number of Mel bins: 128, 160, 200, 256

In evaluating the success of the model, I determined that the F1 metric, both for the system as a whole and for each classified label (effect), would be more useful than accuracy. If accuracy were used, it would have been quite low throughout the design and redesign process for the network. This is because accuracy is measured by the times the network is entirely successful in classifying an audio sample. For example, if a sample contains only chorus and reverb, but the classifier guesses it contains chorus, reverb, and distortion, the calculations for an accuracy metric would count this classification as a complete failure. This would not be useful for determining how to design the system going forward. It will be more useful to consider the F1 scores for each effect. If in that case one effect is not as well recognized as the others by this classifier, I can more easily target that label by making informed adjustments to the data format or network design.

5.2 Initial Network

Using the existing code sources, I defined a four-layer convolutional neural network (Figure 6). Weights were scaled from 32 to 128 across the four layers. No dropout or regularization techniques were used in the initial design. Exact network design can be viewed in the Colab notebook.

5.3 Testing Audio Parameters

I decided to systematically test every audio parameter, beginning with duration. I used a basic early stopping system to monitor validation loss, and tested the model using durations of two to seven seconds, using default values for the rest of the parameters (128 Mel bins, batch size 32, mono audio,

Table 1: F1 Scores by Duration and Effect

Duration (seconds)	Overall F1	Distortion F1	Chorus F1	Delay F1	Reverb F1
2	0.7975	0.9728	0.5246	0.7516	0.9410
3	0.7974	0.9474	0.6214	0.6587	0.9623
4	0.7798	0.9577	0.4726	0.7157	0.9731
5	0.7874	0.9592	0.5125	0.7476	0.9302
6	0.8018	0.6532	0.6237	0.6842	0.9462
7	0.7222	0.9378	0.3889	0.6093	0.9528

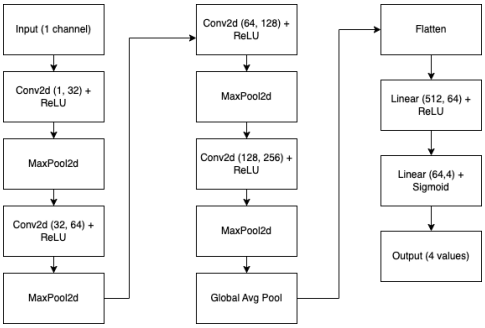

Figure 6: Initial Network Design

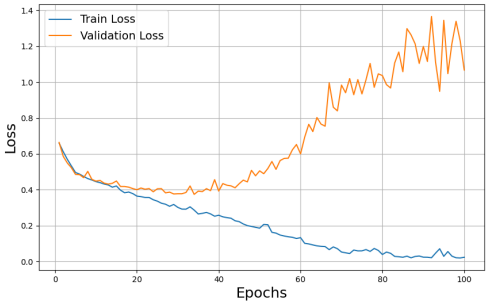

Figure 7: Validation and Training Loss, Five-Layer Network

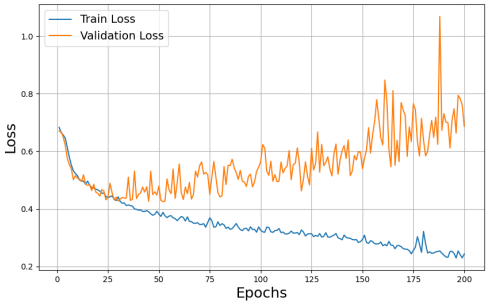

Figure 8: Validation and Training Loss, Five-Layer Network with Dropout

22.05 kHz sampling rate). The results are reported in Table 1, including the overall F1 score and the per-label F1. The learning rate was set at 0.001, and a batch size of 32 was used.

From these results, I determined that the most effective audio duration to use is three seconds (Table 1). While two and six seconds did have higher overall F1 scores, the spread of F1 scores between the individual effects are less consistent than three seconds. I decided to prioritize a well-balanced model.

Through testing of each Mel bin value (three attempts for each size, reporting the average), I determined that better results are produced when using 256 bins, which is twice as detailed as the default value for that parameter (128, as recommended by Librosa). There appears to be a positive relationship between level of detail in the Mel spectrograms and the success of the model (Table 2).

5.4 Network Redesigns

In beginning to test every audio parameter, I noticed that the validation loss for the network was never reaching values below 0.4. Since training loss was just slightly lower than 0.4 at this point, the model appeared to be underfitting. To introduce some complexity, I added a fifth layer to the network. I turned off the early stopping parameter, and observed a training loss that could reach much lower values, while validation loss would eventually diverge at a point where early stopping would kick in (Figure 7).

In terms of per-label metrics up to this point, it is clear that the model is easily learning how to recognize reverb and distortion, with F1 values well above 0.90. Chorus and delay are proving to be a bit more difficult. At this point, I switched the input data to stereo audio. To accommodate the increasingly complicated data (two input channels instead of one) I added dropout - 0.2 for the input layer, and 0.5 before the final activation. The results present a model that presents marginally better performance for chorus and delay (Figure 8).

To begin experimenting with hyperparameter adjustment, I trained my model using batch sizes of 16, 32, and 64. The F1 results revealed that batches of size 32 yield both the most correct and most consistent results across all effects: Overall F1 was the highest with 32-sized batches, and every effect was best recognized at this level as well.

I decided that the next course of action should be to continue with this dropout model which did not seem to overfit as much, and experiment with hyperparameter adjustment. In my initial network design, I determined that, when coupled with the most ideal audio parameters (256 Mel bins, 3 seconds, stereo audio), a batch size of 32 yielded the best F1 results. So, I continued to experiment with different early stopping setups on the 5-layer model with 0.5 dropout. With a patience of 15, I tried out models that would respond to

Table 2: Mel Spectrogram Detail Experiments

Number of Mel Bins	Overall F1	Distortion F1	Chorus F1	Delay F1	Reverb F1
128	0.784	0.527	0.510	0.684	0.9463
160	0.782	0.927	0.526	0.701	0.968
200	0.780	0.945	0.582	0.640	0.954
256	0.791	0.953	0.559	0.690	0.967

Table 3: Early Stopping Delta Experiments

Delta Value	Epochs Trained	Distortion F1	Chorus F1	Delay F1	Reverb F1
0.1	22	0.931	0.612	0.641	0.891
0.01	45	0.937	0.582	0.684	0.935
0.001	41	0.955	0.489	0.655	0.937

validation loss deltas of 0.1, 0.01, and 0.001. Per-label performance is reported in Table 3.

I will concede that there is a possible error in my early stopping configuration for which I did not have the time to fix, as after these experiments I decided to turn off the setting received conflicting results when training for concrete epoch numbers higher than the points where early stopping was activated. 70 epochs seems to yield the best performance, with Overall F1 = 0.799, Distortion = 0.923, Chorus = 0.624, Delay = 0.724, Reverb = 0.928.

5.5 Augmentation

In an attempt to help the model learn to better recognize delay and chorus, I decided to increase some data augmentation in the training set. This is a similar process to augmenting an image data set by shifting, blurring, or cropping pieces from the training data. In the audio space, two common subtle transformations that can help a model more easily learn and generalize outside of a training set are time and pitch shifting. I decided to try some basic experiments with time shifting samples in the training set, making some samples slightly slower or faster at random.

For the first experiment, 30% of the training samples were slightly time-shifted, and the 5-layer model trained for 70 epochs. The model did not show significant improvement in reverb, distortion or delay, and actually reported a significant decrease in chorus performance, with an F1 score of 0.302. In a follow-up attempt, the rate of augmentation was reduced to 20%, and the model trained for 150 epochs. The success of this model was in line with earlier experiments, with F1 scores for distortion, chorus, delay, and reverb at 0.936, 0.629, 0.668, and 0.958 respectively. It seems that this type of augmentation did not provide a useful improvement to the model.

6. ANALYSIS

Based on the performance of my model, it is clear that the distortion and reverb effects are quite easily recognized without much need for network redesigns or optimization of audio format or hyperparameters. However, there is more work to be done to improve the model's recognition of chorus and delay.

This is likely due to the fact that distortion and reverb are more frequency-based than chorus and delay, which are largely time-based. As mentioned earlier, distortion and reverb alter the frequency content of the signal, distortion by "clipping" the waveform of the sound and reverb by adding frequencies to the signal to account for those which resonate more heavily in a large physical space. By contrast, chorus and delay manipulate the signal in the time domain by adding delayed copies of the original signal. In fact, chorus and delay are essentially the same effect: delay adds an "echo" of the signal (or several of them) usually around 50-500ms after the original, while chorus is just a much faster delay, usually 1-30ms, for the purpose of "thickening" the sound rather than adding a clear echo. If we consider chorus and delay to be two different intensity levels of the same effect rather than two distinct effects, it makes sense that my model would have roughly the same level of difficulty learning to accurately place both labels.

Additionally, the audio representation I decided to use for my model, a Mel spectrogram, is primarily concerned with the frequency content of a signal. While these spectrograms do consider how frequencies change over time, the way they cluster frequencies into time "chunks" does not prioritize short-term timing details characteristic of chorus and delay. To improve the model's ability to recognize these effects, it may be worthwhile to change from using Mel spectrograms as input to raw WAV files. This would, however, require a much larger input vector and more detailed network, since even at a downsampled 22.05 kHz a one-second audio clip used as input would be represented as a vector of 22050 floats.

7. CONCLUSIONS AND FUTURE WORK

This paper presents FXNet, a convolutional neural network that is trained to classify common audio effects—reverb, delay, chorus, and distortion—applied to musical instrument recordings. Using synthesized audio samples, the model was trained on Mel spectrogram representations and exhaustively tested with audio parameter changes, network architecture, and training procedures.

Results indicate that FXNet performs very well at detecting reverb and distortion, with F1 values well above 0.90, while chorus and delay classification remains problematic. It is likely that the reason these effects are difficult to recognize is that they have a time-domain characteristic which is not adequately represented by frequency-focused Mel spectro-

grams.

Several experiments were conducted to adjust input audio length, resolution of spectrogram, and network depth, and to integrate data augmentation. Albeit changes in spectrogram hyperparameters and depth of the network yielded modest gains, data augmentation by pitch and time shifts did not enhance performance considerably for more difficult labels.

These findings point to two main directions of future enhancement. Firstly, as input the unprocessed waveform data would better capture the nuances of chorus and delay but at a gargantuan increase in input dimensionality and CNN complexity. Secondly, employing hybrid or temporal models, such as CNN-RNN models or transformers, could provide more context for short-term timing variations.

Overall, FXNet illustrates the power of convolutional models for effect classification and highlights the limitation of spectral representations for time-based audio features. Future work will involve combining other data representations and architectures to close the performance gap between effect classes.

REFERENCES

[1] Bamblebam. Audio classification and regression using pytorch. Accessed: 2025-04-28. [Online]. Available: https://bamblebam.medium.com/audio-classification-and-regression-using-pytorch-48db77b3a5ec

[2] B. McFee *et al.* Librosa documentation. Accessed: 2025-04-28. [Online]. Available: https://librosa.org/doc/latest/index.html

[3] O. Medhat. Sound classification with mel-spectrogram. GitHub repository, Accessed: 2025-04-28. [Online]. Available: https://github.com/OmarMedhat22/Sound-Classification-Mel-Spectrogram/blob/master/mel%20spectrogram.ipynb

[4] P. Team. Pytorch documentation. Accessed: 2025-04-28. [Online]. Available: https://pytorch.org/docs/stable/index.html