Drum Sample Captioning Using Deep Learning

Alec Ames — 6843577
COSC 4P80, Brock University, St. Catharines, ON

***Abstract*—This deep learning project presents an audio classifier designed to identify and timestamp drum samples within an audio file. The classifier, built using the PyTorch platform, classifies audio into five main types of drum samples: kicks, snares, cymbals, and hats. The system utilizes a Convolutional Neural Network (CNN) trained on a dataset of drum samples. The trained model is then used in a script that detects transients in an audio file and classifies each transient window as a drum hit. The model demonstrates impressive accuracy, with the majority of misclassifications occurring between cymbals and hi-hats. Overall, this audio classifier demonstrates significant potential for accurately classifying and time-stamping drum samples within a recording.**

I. Introduction

Audio classification is an important task in many areas of music. In particular, the classification of drum samples can be an incredibly useful tool in music production, where the ability to accurately locate and identify an instrument sound in an audio clip can allow for automated sample extraction from a drum session recording. It can also be extended to enhance accessibility by automatically captioning audio recordings with their respective instrument, and automatic generation of drum sequence tablature. However, this task can be challenging due to the complexity and variability of drum sounds.

Deep learning has proven to be an effective method to classify multiple types of data, such as text and images. It can also be used to identify patterns and extract features in audio data that are not possible to capture with traditional signal processing techniques.

In this project, I implement a deep learning-based audio classifier and captioning tool using the PyTorch platform. The classifier is designed to identify, timestamp, and optionally extract five types of drum samples: kicks, snares, toms, cymbals, and hats within an audio file.

II. Dataset

The dataset used to train the network is a combination of the Drum Kit Sound Samples dataset from Kaggle [3], which consists of 160 labeled samples, and 370 samples from a free collection of drum samples from MusicRadar [2], originally from the Computer Music magazine. Initially, only the Kaggle dataset was used to train the model, but the small size led it to fail to generalize. The second dataset was added as it contained different examples of drum samples from different type of drum kit, which should allow it to generalize effectively.

Prior to being used for training, the audio data is resampled from 44.1kHz to 16kHz. This lowers the resolution of the audio, reducing the required training time. However, there is a trade-off when downsampling. In audio, the highest frequency an audio file can create, known as the Nyquist frequency, is half the sampling rate as shown in Fig. 1.

$$f_N = \frac{\texttt{sample rate}}{2}$$
$$f_N = \frac{16000Hz}{2} = 8000Hz$$

Fig. 1. Nyquist frequency

This means that drum samples with frequencies higher than 8kHz, such as the high frequency harmonics may not be perceptible to the neural network. When analyzing a cymbal under a spectrum analyzer, however, we can see that themain characteristics in cymbal recordings exist within the 4-6kHz frequency range, meaning that it will still be able to detect these features [5]. In Fig. 2, the reduced frequency range when downsampling to 16kHz is shown highlighted in green, meaning the range lost is only the very high end of the spectrum.

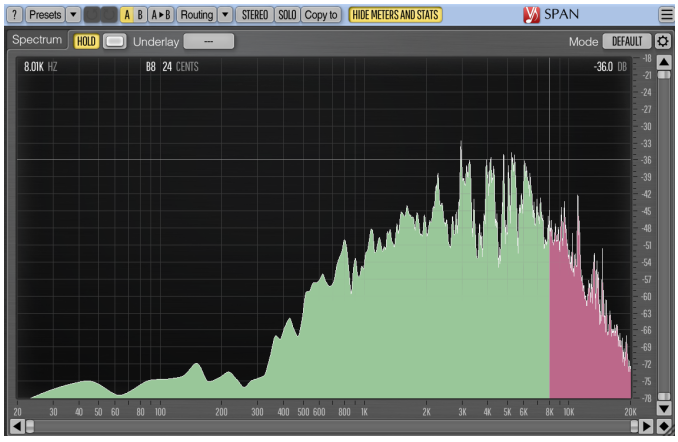


Fig. 2. Effect of downsampling to 16kHz

In the following step in data preparation, stereo audio samples are converted into a monophonic format. This process, known as downmixing, merges the multiple audio channels into one. This is done for two reasons. Firstly, the difference in sound between stereo and mono audio in this case is negligible, and a one-dimensional audio input streamlines and accelerates the training process.

The final audio processing step is to standardize the duration of the training data. This is achieved by either truncating or padding the audio to a length of one second. The majority of drum samples are shorter than 1 second, and standardizing the input size allows us to efficiently configure the network with a fixed layer size.

III. Network Architecture

The Convolutional Neural Network (CNN) model defined in the following section is a sequence of layers designed to process and classify one-dimensional input data, a monophonic audio file. The model consists of two main parts: convolutional layers, and a fully connected layer. The following is the sequence of layers:

`Conv1d` → `BatchNorm1d` → `ReLu` → `MaxPool1d` → `Conv1d` → `BachNorm1d` → `ReLU` → `MaxPool1d`

The Convolutional Layers apply filters to the audio input to extract features. The first layer takes the one-dimensional waveform as input and uses 32 filters, and the second uses 64. Each filter scans the audio data and transforms it to highlight specific features. Batch Normalization layers normalize the output of the convolutional layers, which helps speed up learning and improves the stability of the network [4]. ReLU (Rectified Linear Unit) is the activation function used to introduce non-linearity to the network. The Max Pooling layers reduce the dimensionality of the data by taking the maximum value over each sliding window of size 4, simplifying the input for the next layers. The Fully Connected Layer maps the high-level features extracted by the convolutional layers to the final output classes (kick, snare, tom, hat, cymbal).

IV. Training Parameters

The training parameters used to train the model are listed in Table I.

TABLE I
Training Parameters

Parameter	Value
Optimizer	Adam
Loss Function	Cross Entropy Loss
Epochs	250
Learning Rate	0.001
Test Split	20/80 (0.2)

- **Optimizer:** The optimizer used in this model is Adam (Adaptive Moment Estimation). This is a commonly used optimizer as it is computationally efficient and converges rapidly.
- **Loss function:** The loss function used in this model is the Cross Entropy Loss function. This loss function was chosen due to its properties and effectiveness in handling multi-class classification problems, such as different drum hit sounds.
- **Epochs:** The number of epochs is set to 250. This parameter determines the number of complete passes through the entire training dataset. The value was chosen after several rounds of testing. Higher values led to overfitting, where the model performed well on the training data but poorly on unseen data. Therefore, 250 was found to be a suitable choice that allowed the model to learn effectively without overfitting to the training data.
- **Learning rate:** The learning rate is set to 0.001. This parameter controls the rate or speed at which the model learns. Several values were tested for this parameter, however, this learning rate paired with training the model over 250 epochs yielded the most accurate classification model.
- **Test split:** The test size is set to 0.2, meaning that 20% of the data is used for testing the model, while the remaining 80% is used for training. This split was determined after testing various proportions. A 20% test size was found to provide a sufficient amount of unseen data to testing the model's performance while leaving enough data for effective training.

During the training process, using the PyTorch platform proved to be beneficial as it features hardware acceleration. For comparison, training the model with the CPU took upwards of 5 minutes, whereas only 11seconds on the GPU.

```
Using device: cuda
100% 250/250 [00:11<00:00, 21.58epoch/s]
...
```

V. Transient Detection

The transient detection algorithm uses the onset detection utility from the `librosa` audio processing Python library [1]. This tool detects abrupt changes in amplitude given a threshold, and returns the timestamp of the transient. The detected transients are then filtered to ensure a minimum time gap of 100ms between consecutive transients. This is done to avoid multiple detections of the same drum hit due to reverberations or reflections. The resulting list of transient starting timestamps is used as the basis for the classification process.

VI. Classification Script

The classification script uses the trained model to classify the detected transients. For each transient, a window of audio data, from the start of the transient to the start of the next transient, is extracted and padded or truncated to fit the model's input size of one second.

The script then loads the trained model, `model_v0.12.pt`, and defines the labels for the drum classes. The audio file is loaded and the transients are detected using the algorithm defined above. The classification results, including the timestamp, predicted label, and confidence score, are stored in a list. The waveform for the input audio is plotted, along with markers for each transient and the label at their respective timestamp.

The classification script has an optional argument `-e` or –export_path which, if defined, allows the user to define a folder for the program to extract and label the classified samples from the audio source.

VII. Results and Discussion

The performance of the trained model was evaluated using the unseen testing data, and it achieved an impressive accuracy of 100%. This high level of accuracy was consistent when the model was applied to single drum sample audio files (Figs. 3-7), where it correctly classified each sample with 100% accuracy.

```
Using device: cuda
```

```
100%|| 250/250 [00:11<00:00, 21.58epoch/s]
Training samples #: 422
Testing samples #: 106
Test Accuracy: 100.0%
```

However, when the model was used to classify sequences of drum recordings, it demonstrated a tendency to misclassify hi-hats as cymbals. This was observed in the `test.wav` file (Fig. 8), where every alternate sample should have been a hi-hat, not a cymbal. Similarly, in the `test_2.wav` file (Fig. 9), the first three cymbals were incorrectly classified as hi-hats. Despite these occasional errors, the model performed extremely well in most cases. It was able to correctly identify kicks, snares, and toms with high confidence. The misclassification of hi-hats and cymbals could be attributed to the acoustic similarities between these two types of drum samples, as cymbals are essentially just a longer variation of a hi-hat sample.

VIII. Conclusion

In this project, I successfully implemented an audio classifier using deep learning and the PyTorch platform to timestamp and label drum sounds within an audio file. The model achieved 100% accuracy on the unseen testing dataset and consistently classified single drum sample audio files with 100% accuracy.

One limitation of the model is that it often failed to distinguish hi-hats and cymbals. This is likely due to the two sounds consisting of the same acoustic properties, with the major difference being in duration. This could also be attributed to the relatively small dataset, with only 530 samples. In future iterations, increasing the size of the dataset and including more examples of hi hats and cymbals would be effective ways to improve the performance of the model.

Despite this, the model performed exceptionally well in most cases, identifying the rest of the drum samples with high confidence. This model could be refined and used in many applications, such as music production, accessibility, and automatic generation of drum sequence tablature.

References

[1] Librosa: Audio and Music Signal Analysis in Python. [Online]. Available: https://librosa.org/doc/latest/generated/librosa.onset.onset$_detect.html$

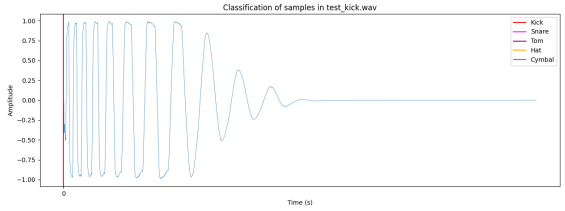
[2] 1000 free drum samples. [Online]. Available: https://www.musicradar.com/news/drums/1000-free-drum-samples

[3] Drum Kit Sound Samples. [Online]. Available: https://www.kaggle.com/datasets/anubhavchhabra/drum-kit-sound-samples

[4] PyTorch: BatchNorm1d. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html

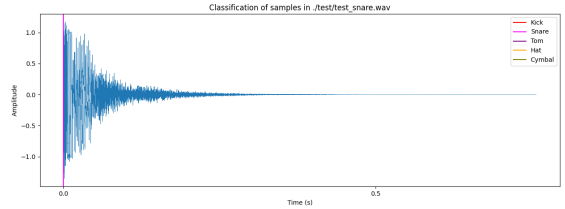
[5] Percussion Frequencies, Part 2: Cymbals. [Online]. Available: https://www.musical-u.com/learn/percussion-frequencies-part-2-cymbals/

APPENDIX



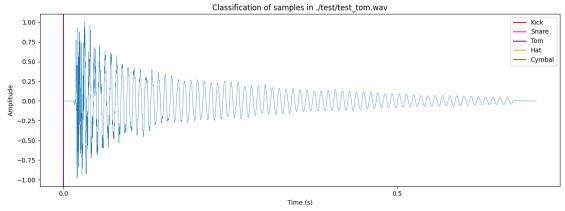

```
Using device: cuda
Loaded test_kick.wav
Resampling to 16000Hz
Detected 1 transients
0.000: kick (100.00%)
```

Fig. 3. Classification of samples in test_kick.wav (single sample)



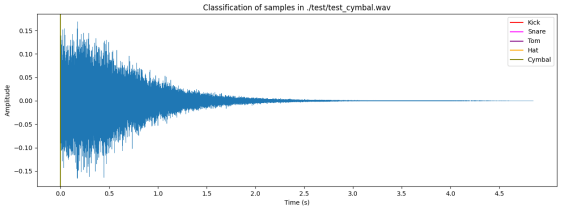

```
Using device: cuda
Loaded ./test/test_snare.wav
Resampling to 16000Hz
Detected 1 transients
0.000: snare (100.00%)
```

Fig. 4. Classification of samples in test_snare.wav (single sample)



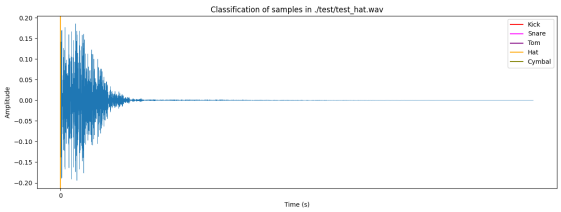

```
Using device: cuda
Loaded ./test/test_tom.wav
Resampling to 16000Hz
Detected 1 transients
0.000: tom (100.00%)
```

Fig. 5. Classification of samples in test_tom.wav (single sample)



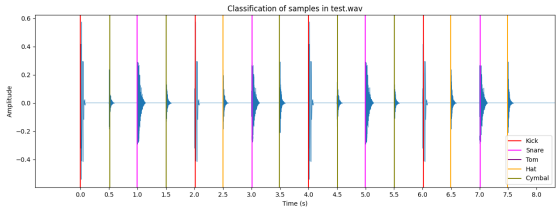

```
Using device: cuda
Loaded ./test/test_cymbal.wav
Resampling to 16000Hz
Detected 1 transients
0.000: cymbal (100.00%)
```

Fig. 6. Classification of samples in test_cymbal.wav (single sample)



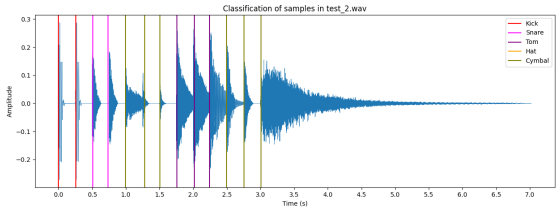

```
Using device: cuda
Loaded ./test/test_hat.wav
Resampling to 16000Hz
Detected 1 transients
0.000: hat (100.00%)
```

Fig. 7. Classification of samples in test_hat.wav (single sample)




```
Using device: cuda
Loaded test.wav
Resampling to 16000Hz
Detected 16 transients
0.000: kick (100.00%)
0.512: cymbal (99.98%)
0.992: snare (100.00%)
1.504: cymbal (99.96%)
2.016: kick (100.00%)
2.496: hat (94.20%)
3.008: snare (100.00%)
3.488: cymbal (95.73%)
4.000: kick (100.00%)
4.512: cymbal (99.98%)
4.992: snare (100.00%)
5.504: cymbal (99.96%)
6.016: kick (100.00%)
6.496: hat (94.20%)
7.008: snare (100.00%)
7.488: hat (100.00%)
```

Fig. 8. Classification of samples in test.wav (drum sequence)




```
Using device: cuda
Loaded test_2.wav
Resampling to 16000Hz
Detected 13 transients
0.000: kick (99.13%)
0.256: kick (99.64%)
0.512: snare (57.97%)
0.736: snare (75.33%)
0.992: cymbal (99.94%)
1.280: cymbal (100.00%)
1.504: cymbal (100.00%)
1.760: tom (100.00%)
2.016: tom (100.00%)
2.240: tom (95.66%)
2.496: cymbal (100.00%)
2.752: cymbal (100.00%)
3.008: cymbal (100.00%)
```

Fig. 9. Classification of samples in test_2.wav (drum sequence)