Note Recognition using Machine Learning

CS 254 UVM Fall 2020

Cameron Hudson, Phillip Nguyen, and Nick Corwin

1. Introduction

The overall goal of the project is to dive into audio analysis in music with machine learning. Previously, Cameron Hudson has done work with music lyrics and lyrical analysis to result in a visual representation comparing unique words per song over an entire album. Due to the strict simplicity limitations given by Prof. Safwan, focused primarily on designing a learning model for note recognition. We achieved this by getting a large dataset with unique notes across different instruments with different audio properties, such as pitch and velocity. This posed some challenge since the exact frequency for the same notes will rarely be perfectly matched. Investigating audio data allowed us to gain some insight into voice and song recognition methods that Siri and Alexa employ. The best results we can get is for note recognition to work with only one note. Multiple notes will not be given or trained. The basic approach we will be using is neural networks specifically with Keras under TensorFlow. The dataset describes features in a Tensor format, where each feature resides. We can extract features from the tensor. We will use the “audio” and “pitch” features to generate our dataset. Keras will examine the audio of the track and do image classification to match audio files to pitch values, then directly map those to conventional note values. With our CNN made, we got an accuracy of 85.7% and were able to show how we can experimentally demonstrate our original hypothesis that we can map audio clips to notes values using a supervised classification learning model.

2. Problem Definition and Algorithm

2.1 Task Definition

We used the Magenta NSynth dataset which is a large-scale and high-quality collection of labeled audio samples containing over 305,979 unique notes from 1,006 different instruments each with unique pitch, timbre, and envelope. The samples are

16kHz mono audio snippets, descriptions of the sound in the MIDI standard (see below for the table of features). Each note is held for three seconds with one second of decay, for a total of four seconds in length. The dataset is available for public use with proper citation. The following features that are available in the dataset are below, but we only used “pitch” and “audio” as stated above.

The inputs for the problem are from the “audio” feature extraction. This will give a tensor with shape (64,000) of type *float32*. This contains a numpy array that contains waveform amplitude values within the range [-1, 1]. This audio is four seconds long, containing 16,000 samples per second, giving us 64,000 samples per second. We then had to visualize this audio sample through some type of medium. Initially a MFCC was used, but after later testing a Log (scale) Mel Spectrogram proved to be the best visualization for our image mapping. Log Mel Spectrogram has been proved to better represent the audio frequencies perceived by the human ear as humans don’t perceive frequencies on a linear scale. Humans can distinguish 500 to 1000 Hz easily, but not 10,000 to 10,5000 Hz, which is the same jump in Hz. Log Mel Spectrogram accounts for this. Now the shape of the Log Mel Spectrogram will be input size for our CNN and all X variables across Training, Test, and Validation set. Each note in each data set has a Log Mel Spectrogram image.

The output is pitch values that were pulled from the “pitch” feature from the Tensor which is an integer in range [0, 127]. The CNN built predicts pitch from the Log Mel Spectrogram and outputs a pitch value within this range using one-hot encoding.

Feature	Type	Description
note	`int64`	A unique integer identifier for the note.
note_str	`bytes`	A unique string identifier for the note in the format `<instrument_str>-<pitch>-<velocity>` .
instrument	`int64`	A unique, sequential identifier for the instrument the note was synthesized from.
instrument_str	`bytes`	A unique string identifier for the instrument this note was synthesized from in the format `<instrument_family_str>-<instrument_production_str>-<instrument_name>` .
pitch	`int64`	The 0-based MIDI pitch in the range [0, 127].
velocity	`int64`	The 0-based MIDI velocity in the range [0, 127].
sample_rate	`int64`	The samples per second for the `audio` feature.
audio*	`[float]`	A list of audio samples represented as floating point values in the range [-1,1].
qualities	`[int64]`	A binary vector representing which sonic qualities are present in this note.
qualities_str	`[bytes]`	A list IDs of which qualities are present in this note selected from the sonic qualities list.
instrument_family	`int64`	The index of the instrument family this instrument is a member of.
instrument_family_str	`bytes`	The ID of the instrument family this instrument is a member of.
instrument_source	`int64`	The index of the sonic source for this instrument.
instrument_source_str	`bytes`	The ID of the sonic source for this instrument.

2.2 Algorithm Definition

There was some preprocessing required on our inputs. For our inputs we used librosa.feature.melspectrogram() to convert our extracted ‘audio’ feature from the tfrecord into a mel spectrogram, which are then scaled relative to the maximum amplitude of the sample.

We also did postprocessing on our outputs. This is only done for human readability and is not essential to our model. For our outputs we converted the midi pitch value into frequency which was then converted to a note. This was all done using a pipeline of functions: midi_pitch_to_node(), midi_pitch_to_frequency(), frequency_to_note().

```python
def midi_pitch_to_frequency(d):
    return (2**((d-69)/12.0))*440.0

from math import log2, pow

A4 = 440
C0 = A4*pow(2, -4.75)
name = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]

def frequency_to_note(freq):
    h = round(12*log2(freq/C0))
    octave = h // 12
    n = h % 12
    return name[n] + str(octave)

def midi_pitch_to_note(d):
    return frequency_to_note(midi_pitch_to_frequency(d))

f = midi_pitch_to_frequency(100)
print("Frequency: ", f)
print("Note: ", frequency_to_note(f))
```

```
Frequency:  2637.02045530296
Note:  E7
```

A CNN model was made with Keras and TensorFlow with the respected input and output described above. Describing the CNN, the input size was the shape of the log mel spectrogram of size (128, 126), then it went to a norm_layer with preprocessing normalization, then two Conv2D layers, then MaxPooling2D, a dropout of 0.25, was flattened, then through a dense layer of size (None, 128), another dropout of 0.5, before output to layer of size 128 (one-hot encoding of our range of pitch values). Many models were tested but we decided to use a CNN. Further explanation of CNN and other models in Section 3.1: Methodology.

3. Experimental Evaluation

3.1 Methodology

We set out to experimentally demonstrate our original hypothesis that we can map audio clips to notes values using a supervised classification learning model. We are using validation and test data to evaluate our model as well as a loss graph between training loss and validation loss to see if the model is overfitting, which it did frequently. We evaluated these tests with general training, validation, and test

accuracy. The NSynth training and test data was imported with TensorFlow tf.data.TFRecordDataset, and is a realistic dataset as it contains hundreds of different instruments with notes differing by pitch and velocity to give over 100,000 unique notes. The dataset is interesting as it used deep neural networks to generate the audio samples used, and we used convolutional neural networks to essentially work backwards to identify the note. The NSynth dataset was able to make acoustically convincing audio samples that cannot reasonably be generated manually by a human synthesizer, giving us truly a very unique dataset.

We will now talk on the models. The first model we made was a basic SVM with input being an array of MFCCs extracted from the audio samples and output being pitch as described before. With initial tests the SVM gave an accuracy of 51%. We needed the model to drastically improve, using a reasonably sized training set, therefore we turned to CNNs. With initial testing 20,000 training samples, 1,000 test samples, and 4,000 validation samples, and early stopping, generated an accuracy of 58%. Without early stopping implemented the model would overfit most of the time. Some changes were made such as a number of samples for testing and validation, with a max accuracy of 62% generated. The group was limited to only using 20,000 training samples as that was already using 11Gb of RAM. This was an issue for us as if we wanted to make the model more accurate, we would have to use more training data, but this wasn't an option for us.

We saw a drastic improvement in the model when we changed our input features to a Log Mel Spectrogram, represented by a 2D matrix containing decibel values where the axes represented time and frequency. The accuracy jumped to around 80%. With some small changes to the model, such as changing the kernel size, stride size, and type of padding, as well as batch size, the model accuracy was 86%. Below is the framework for the model used. In the github repository, you will see two CNNs made, both gave similar results, but this CNN outperformed the other by a couple of percentage values.

Layer (type)	Output Shape	Param #
normalization_26 (Normalizat	(None, 128, 126, 1)	3
conv2d_46 (Conv2D)	(None, 62, 61, 32)	832
conv2d_47 (Conv2D)	(None, 62, 61, 64)	51264
max_pooling2d_22 (MaxPooling	(None, 31, 30, 64)	0
dropout_47 (Dropout)	(None, 31, 30, 64)	0
flatten_26 (Flatten)	(None, 59520)	0
dense_51 (Dense)	(None, 128)	7618688
dropout_48 (Dropout)	(None, 128)	0
dense_52 (Dense)	(None, 128)	16512

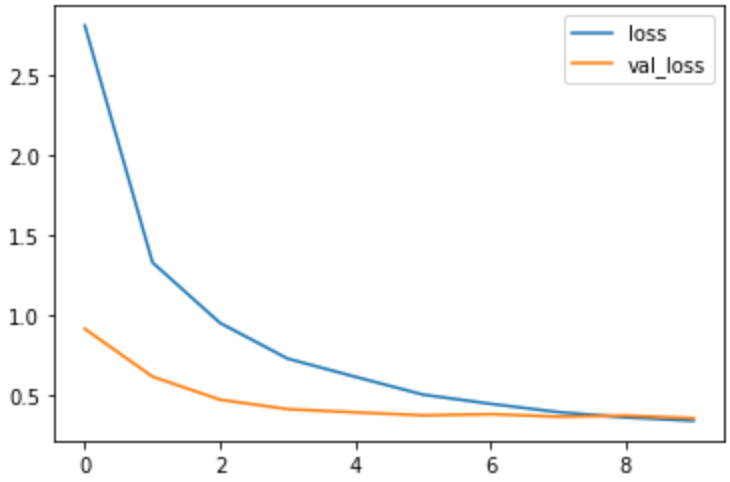
Total params: 7,687,299
Trainable params: 7,687,296
Non-trainable params: 3

Further testing was completed with the SGD optimizer used instead of Adam. Adam performed well with the model, but SGD was tested for any potential improvements in convergence. There was a slight decrease in accuracy from Adam to SGD when tested using a batch size of 128. However, when tested on the previous model with the last dropout layer removed, the SGD optimizer performed only slightly worse than the Adam optimizer, with a difference in accuracy of only 0.02%. The batch size was 128 and 32 for Adam and SGD, respectively. Further testing was carried out by modifying the batch sizes for SGD on the model without the last dropout layer. The SGD performed consistently at a similar accuracy for each batch size, ranging between ~81-83% for batch sizes of 16, 64, and 128. When the same procedure was carried out on the model with the last dropout layer included, the results were promising. Further testing would be needed for more conclusive evidence, but modifications to the learning rate, batch size, and number of epochs to the model depicted above yielded results at and above 86%.

3.2 Results

Below is a graph of the loss and validation loss over each Epoch. The graph below shows some overfitting at the start, but the model fixed itself and stopped overfitting as the number of epochs increased and by the last epoch the loss values were almost the same. Therefore the model was not overfitting or underfitting, it was performing just as intended. If we were able to add more training samples, the

accuracy would be higher. But with limited training samples the model generated a minimum loss of 0.7996 with an accuracy of ~86% (85.72%) on testing data. Looking at the weighted average for the precision and recall we have a value of 0.86 and an F1-score of 0.85. All three statistics show that the model is performing well and within expectations.



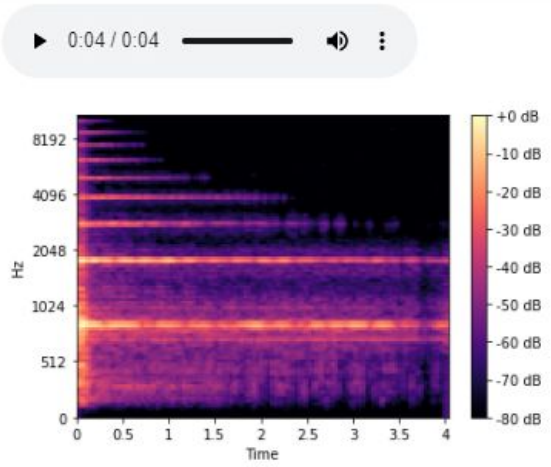

```
Test set accuracy: 86%
125/125 - 29s - loss: 0.7996 - accuracy: 0.8572

Test accuracy for CNN: 85.72%
```

	precision	recall	F1-score	
accuracy			0.86	4000
macro avg	0.72	0.71	0.71	4000
weighted avg	0.86	0.86	0.85	4000

Finally, we are able to save our model and use it to identify notes from new data. Our file, demo.ipynb, contains several examples of how this is done. Here is an example of our model correctly identifying a note from a real (non synthetic) piano which has a considerable amount of noise due to being recorded in a reverberant space:

```
In [6]: # from https://freesound.org/people/TEDAgame/sounds/448553/
        g5_piano_with_reverb, _ = librosa.load('demofiles/g5_piano_reverb.wav', sr=16000, mono=True, duration=4)
        demo_input2 = format_audio(g5_piano_with_reverb)
        ipd.Audio('demofiles/g5_piano_reverb.wav')
```

Out[6]:




```
In [7]: # from https://freesound.org/people/Tesabob2001/sounds/203495/
        output2 = np.argmax(model.predict(demo_input2))
        print("Note: ", midi_pitch_to_note(output2))
```

```
Note:  G5
```

For more demonstrations of our model's performance on new inputs please refer to demo.ipynb at the Github repository linked in the code and dataset section of the paper.

3.3 Discussion

It appears that our hypothesis is well supported. We meet and in some cases exceed the performance of other methods, albeit with higher computing requirements (see related work section). The model has the capacity to perform better than what it already produced but with current limitations on the number of training samples, it has converged to a steady accuracy value. To talk more on this, currently we are not able to load more than 20,000 training samples without running out of memory and drastically increasing the execution time of the program. We plan on pursuing performance optimizations to allow for faster development and perhaps to explore ways to cut down our effective dataset and/or cut down feature space without compromising on model predictability. Being able to train on all 289,205 training samples would be ideal and would provide the best results.

4. Related Work

In the paper "Musical Pitch Detection Using Machine Learning Algorithms" by Ghiurutan Bianca used Support Vector Machine, Stochastic Gradient Descent Classifier and K-Nearest Neighbors algorithms for the classification of sounds into musical notes. The authors mentioned that future attempts at this task could leverage Random Forest Classifiers or different types of neural networks. We opted to explore the use of neural networks, specifically convolutional neural networks.

Their dataset was generated by splicing four Bach chorales with four instruments (violin, clarinet, saxophone, and bassoon) into time frames with MIDI pitch values for part. We are approaching this problem with a collection of short, generated samples (Nsynth) for potentially less noise and error in our training data while simultaneously having access to a substantially greater pool of data (and much wider diversity of instruments) than the original authors.

The authors stated that due to the performance challenges posed by 128-way classification, they opted instead to classify only pitch values that range over what their 4 chosen instruments can actually play, which ended up being only 40 classes. They used the FFT (Fast-Fourier Transform) to extract a feature set with 1016 dimensions over 26228 training samples. For this project we decided to tackle the challenge of 128-way classification (across thousands of instruments) because we wanted to do more precise identification of notes beyond just pitch class and in a more general setting.

Our best model only performs slightly better than their best model in terms of accuracy (86% compared to 83.2%) at face value. However, the larger complexity of the problem we set out to solve should also be taken into account. As previously mentioned in the discussion section, our model still shows signs for potential growth and improvement.

5. Code and Dataset

Dataset is from https://magenta.tensorflow.org/datasets/nsynth#motivation. Installed on all our machines are the tfrecord format for the train, validation, and test dataset. These files are roughly 69GB, 3GB, and 1GB respectively. TensorFlow version 2.3.1 was used with jupyter notebooks. The notebook being used can be found on Cameron's github: https://github.com/camhud/Note-Recognition.git

6. Conclusion

The CNN model made meet and in some cases exceeded the performance of other methods, and it did the full 128-way classification. The accuracy for our current 128-way classification model is around ~86%, which is an incredible high accuracy for such a large classification problem. Our model wasn't overfitting or underfitting, it was performing as it should, this can be seen in the Loss vs. Validation Loss graph. All of this was only possible by our dedication to fine tuning the hyperparameters given in the model, such as number of filters, kernel size, stride size, activation functions, padding, pooling size, number of dropouts, the optimizer used, epoch value, early stopping or not, and finally batch size. Our team did an excellent job playing around with these hyperparameters to generate the best model with our current limitations involving the training set. Possibly ways to improve the model would be to implement ensemble approaches such as CNN stacking, which was attempted in testing but ultimately failed. The team had access to an NVIDIA GPU which was used for time-efficient parameter testing. The process involved with associating one's GPU with the cuDNN program (NVIDIA's deep learning toolkit) is somewhat lengthy, but the hardware was able to successfully handle some of the larger operations on the model in a way that allowed for faster and more frequent testing with optimizers, learning rates, and batch sizes. But even with this added computational power, there were still some limitations; specifically, varying batch sizes with the SGD optimizer on the model proved to be very time expensive. The results were strange and further testing was needed, but given the cost in time it wasn't considered feasible to replicate the results of each tested batch size. Next time we should do our preprocessing in advance and cache the results to permanent storage. Hopefully this would decrease our memory usage and cut down on our notebook runtime.

Bibliography

Gartzman, Dalya. “Getting to Know the Mel Spectrogram.” *Medium*, Towards Data Science, 9 May 2020, https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0.

Ghiuruțan, Bianca. “Musical Pitch Detection Using Machine Learning Algorithms.” *Academia.edu*, 2014, www.academia.edu/7560072/Musical_Pitch_Detection_Using_Machine_Learning_Algorithms.

Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders." 2017.

McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. “librosa: Audio and music signal analysis in python.” In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

https://magenta.tensorflow.org/datasets/nsynth#motivation

https://github.com/librosa/librosa