Capstone Project

Machine Learning Nanodegree

Sam Wachtel

Using Deep Neural Network to Predict Musical Instrument Family

I. Definition

Project Overview

The goal is to attempt the construction of a machine learning model that will be able to predict the instrument family of a single recorded note.

Problem Statement

The practice of music transcription is required any time a live musical performance must be transformed into the written language of Western music. With traditional methods, the transcription of live music into music notation is a slow, painstaking, and manual task.

Every day, new tools are introduced that do a fair job of aiding musicians in this task, however, even the more advanced tools do not distinguish between instrument types. For example, if a musician were to transcribe music of two instruments (flute and oboe, for example), he/she would use two bars (lines) on the score to represent those two instruments. Automatic transcription software today might detect the correct notes (and timing of the notes), but would not distinguish between the instruments and the result would be a score with all of the notes placed on one bar (line). A human would have to intervene at this point to separate the notes out into multiple bars.

Metrics

Accuracy reflects model performance. It is defined simply as the number of correct predictions divided by the total number of predictions.

II. Analysis

Data Exploration

The NSynth Dataset by Google Inc. (see NSynth Paper and Magenta in References) was developed to create a neural network that synthesizes new instrument sounds by training on sound samples from existing instruments. The dataset contains 305,979 audio samples in wav format and contains 1,006 instruments sampled for every pitch within each instrument's range at five different velocities. Along with each sample, the following relevant attributes were captured:.

- Pitch
- Instrument
- Velocity
- Source: acoustic, electronic, synthetic
- Family: bass, brass, flute, etc
- Qualities: bright, dark, multiphonic, etc

The advantages to using this dataset are:

- Creative Commons Attribution 4.0
- Designed specifically for machine learning
    - well labeled
    - standardized length
    - fully labeled
    - available in tfrecord format
    - already broken out into Training, Validation, and Testing sets
- Samples are clean with no ambient interference
- Samples are all monophonic
- Samples are all 16kHz
- Dataset is large

While the NSynth dataset was not developed specifically for the purpose of predicting instrument families, it is general enough to very easily suit this purpose.

TFRecord Files

TFRecord [1] files store serialized data in a streamable TensorFlow format. Conveniently, the NSynth Dataset comes packaged in this format.

An example of the contents of a single record:

[1] TensorFlow TFRecords: https://www.tensorflow.org/tutorials/load_data/tf_records

NOTE: **Bolded** values below (note_str and audio) were required for this project, however other values were used for experimentation.

Name	**Value**
audio	array([[-1.7977802e-05, 5.2617561e-05, 3.7967369e-05, ..., 1.9680847e-05, -1.9699957e-05, 1.9492341e-05]], dtype=float32)
instrument	46
instrument_family	10
instrument_family_str	**vocal**
instrument_source	0
instrument_source_str	acoustic
instrument_str	vocal_acoustic_000
note	12425
note_str	**vocal_acoustic_000-064-075**
pitch	64
qualities	array([[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])
sample_rate	**16000**
velocity	75

Exploratory Visualizations of Data

There are two primary ways of visualizing audio data: waveform and spectrogram. But first, a short description of audio data itself.

Encoded Audio Data

Digital audio data is a time series of mathematically encoded audio samples taken at a fixed rate. To generate the encoded data from audio, an analog-to-digital converter is used.[2] Each of these samples can be thought of as a mathematical snapshot in time of the sound. The rate in which the samples are

[2] https://en.wikipedia.org/wiki/Analog-to-digital_converter

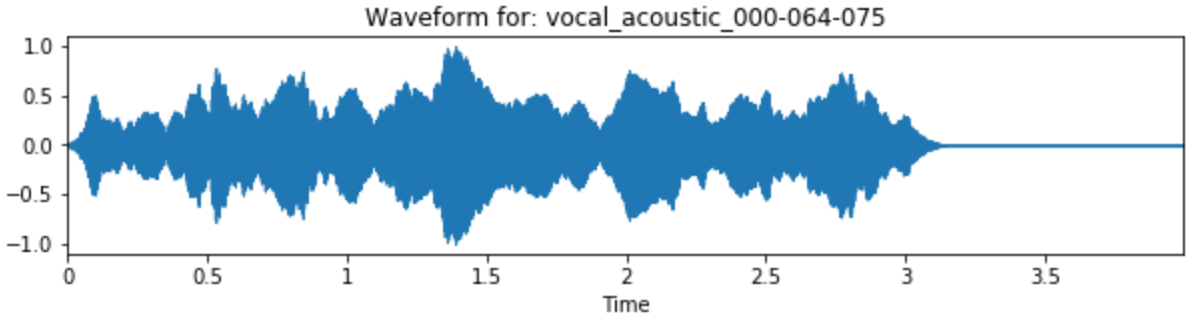
taken is called the sample rate, also known and Hz and "samples per second" [3]. The most common method of representing analog sound digitally is called pulse-code modulation (PCM).[4] [5]

The audio files in our dataset have the following attributes:

Number of total samples per clip	64,000
Sample Rate	16000 Hz (or 16000 samples per second)
Sample length	64000 / 16000 = 4 seconds
Audio Data Sample	[-1.7977802e-05 5.2617561e-05 3.7967369e-05 ... 1.9680847e-05 -1.9699957e-05 1.9492341e-05]

Waveform

Audio is usually represented as a waveform. The following is a four second audio clip of human voices singing a single note. The x axis is time and the y-axis is the amplitude of the sound sample at that point in time.



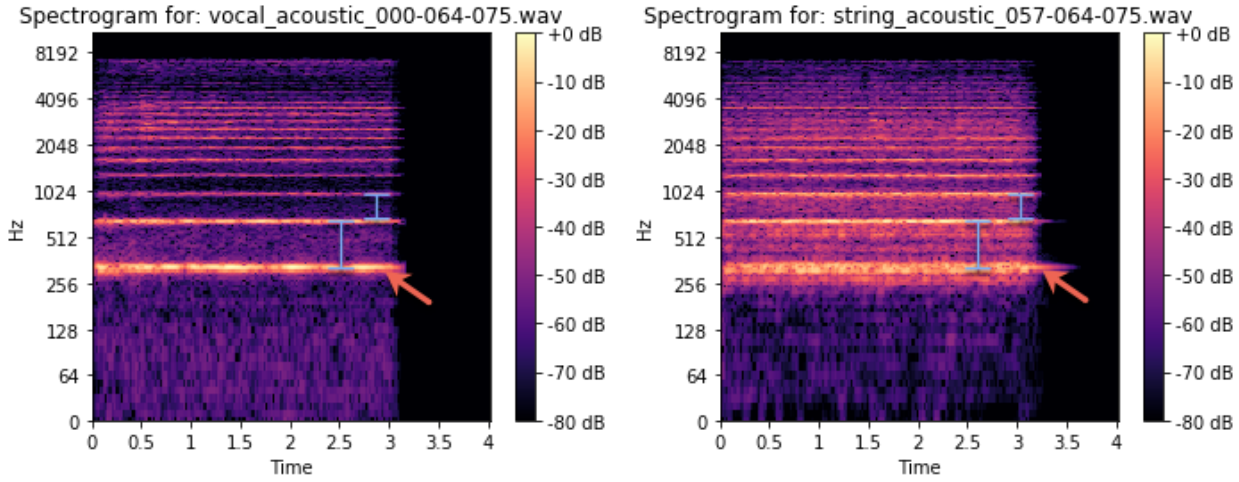
This type of basic waveform is often used by audio engineers to understand the timing and volume (loudness) of the sound. This visualization is limited, however, in that it does not express other information like frequency, frequency intensity, or note overtones. Notice that there is no way of telling that the above note is an E (E4, to be precise, which is the 4th E from the lowest E on a piano) with a frequency of 329.63Hz except by the name of the instrument that contains the midi pitch (64).[6,7] See TFRecord Files section above for more details on note information that comes packaged with the dataset.

---

[3] https://en.wikipedia.org/wiki/Sampling_(signal_processing)#Sampling_rate
[4] https://en.wikipedia.org/wiki/Pulse-code_modulation
[5] https://www.presonus.com/learn/technical-articles/sample-rate-and-bit-depth
[6] https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies
[7] The instrument names are formatted: <instrument_str>-<pitch>-<velocity>. See https://magenta.tensorflow.org/datasets/nsynth for more information about the dataset.

Spectrogram, Visualization of Sound Frequencies and Envelope

The same audio sample above can be represented by a spectogram. This type of visual representation of sound includes information about the spectrum of *frequencies* over time as well as the *envelope* of the sound. The X axis is time in seconds, the Y axis is the frequency in hertz and the colors represent the strength of the sound in decibels (dB), or volume.



In the examples above (Spectrogram for: vocal_acoustic_000-064-075 and string_acoustic_057-064-075) there are two instruments playing the same E (E4) note. The frequency of the note is 329.63Hz. Frequency is the *rate of vibration* of a sound, which means that the sound wave is causing compression and decompression of air at a rate of about 329.63 cycles per second.[8]

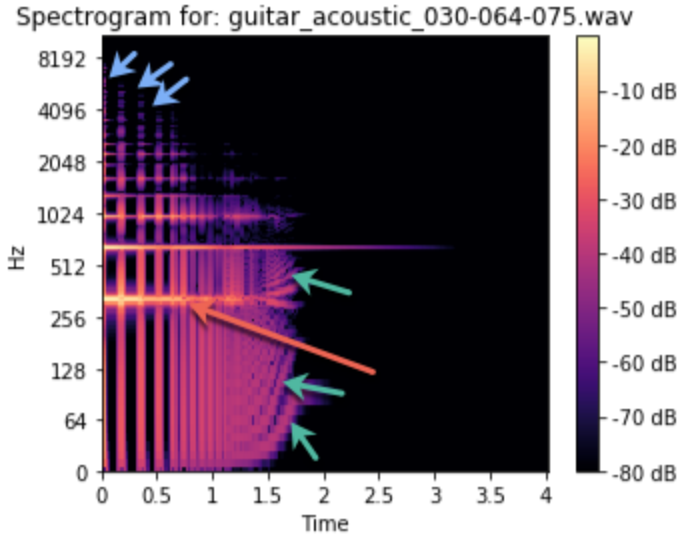
The first spectrogram is of the vocal note. The second is of a string note. Notice that the bottom-most line (with the red arrows) lies just around that 329Hz mark. That line represents the *fundamental tone* (the main tone that is the E being played or sung) and all of the lines above that are called *overtones*, *harmonics*, or *semitones*.[9] It is the quality of these tones (as well as the *envelope*, discussed below) that create the *timbre* (or distinguishing qualities) of an instrument. They are part of what makes a trumpet sound like a trumpet.

To be clear, an E4 note played on two different instruments will generally have the same set of overtones (see the blue markers in the above diagram). What is different is the quality (strength, vibrato, etc) of the overtones that is different. Notice the way they appear differently on the spectrograms. On the string spectrogram, the lines are brighter and it looks as though the overtones have a lot more sound in between them.

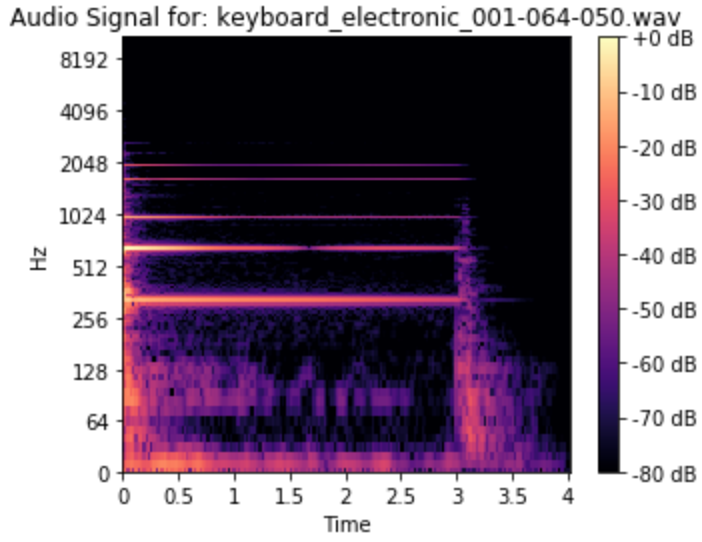
The other aspect feature of sound that differentiates one instrument's timbre from another is *sound envelope*. The envelope describes the attack (start), sustain (middle), decay (decline), and release (end) of that sound. [10]

---

[8] https://simple.wikipedia.org/wiki/Hertz
[9] https://www.teachmeaudio.com/recording/sound-reproduction/fundamental-harmonic-frequencies/
[10] https://en.wikipedia.org/wiki/Envelope_(music)



Compare the vocal and keyboard sounds to the one above of an acoustic guitar playing the same E4 note. The red arrow is pointing to the same E4 at 329.63Hz, however, the sound 'looks' entirely different. There are a number of interesting aspects to the spectrogram that gives us an idea of how the guitar sounds. The first comes from the vertical lines (blue arrows). They suggest a kind of on/off quality to the guitar sound.[11] The second comes from the first overtone lasting longer than the fundamental tone. The third is that the frequencies warp (green arrows) as the note begins to decay. The fourth is that the note doesn't not last as long as the vocal and keyboard notes. So, quite a lot can be determined about a sound from its spectrogram representation.



[11] Note: The vertical lines did not look like a traditional guitar sound so I listened to it. In fact, it sounds as if the guitar sound was sampled through a filter of some sort, so while this is an acoustic guitar sound, it is not a clean one. This is a good example of how a spectrogram can be used to understand a sound visually.

And this last example is of an electronic keyboard sound. The sound appears to be less complex as there are fewer strong overtones. Note that the same overtones do exist as the vocal and string sounds, but they are much weaker and can not all be seen on the spectrogram. In this keyboard sound sample, the attack and release of the sound appear chaotic. A chaotic (non-tonal) start to a sound often reflects a percussive element to the instrument, like a piano hammer hitting a string where the tone of the instrument is not yet cleanly established. The end of a piano note can also cause a non-tonal sound when the damper touches the string to stop it from vibrating.

It is these special properties of the piano sound envelope that give the piano such a distinctive percussive sound. [12]

Mel Spectrogram

This model is currently using a variation on the traditional spectrogram called a Mel Spectrogram. It is based on the Mel Scale, which is a non-linear transformation of the Hz frequency scale. [13,14] The Mel Scale is based more on how humans perceive the distances between tones.

Algorithms and Techniques

The technique used to identify the family of instrument in an audio sample is based on the same approach ordinarily used to identify and classify objects in a photograph. It may seem counterintuitive at first, but given the efficiency of representing sound visually, audio analysis via imagery makes a lot of sense. The real time identification of sound attributes like primary frequency (Hz), overtone (also Hz), envelope, and volume (dB) contained within a digital audio format is easily done with existing instruments (equipment used to aid in the tuning of pianos, for example), however, the training of a machine learning model is more complicated for this type of approach, although there are approaches, like Long-Short Term Memory neural networks (LSTMs). I followed a set of techniques inspired by a work by Adrian Yijie Xu.[15] Xu used a CNN to detect various urban sounds found in the urban sound dataset.[16]

In order to simplify the problem, we are only using one note (pitch) across all of the instruments to make sure that the changes in pitch on the spectrograms will not interfere with the detection of instrument families.

---

[12] In the early days of sound synthesis the biggest challenge of replicating the piano sound was the envelope. To this day, some well loved vintage keyboard sounds can trace their lineage back to the early attempts at synthesizing a realistic piano sound. It is still a challenge today and most electronic musicians will use sampled (recorded) instruments rather than synthesized tones when they want a realistic sound.

[13] https://en.wikipedia.org/wiki/Mel_scale

[14] https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0

[15] Urban Sound Classification using Convolutional Neural Networks with Keras: Theory and Implementation, Febuary, 2019

[16] https://urbansounddataset.weebly.com/urbansound8k.html

The approach used in this experiment is to convert the entire NSynth Dataset to Spectrograms and use the training and validation sets to train a Convolutional Neural Network to recognize instrument families. [17]

Benchmark

Other studies on this problem are based on entirely different approaches. It is, therefore, reasonable to use a naive predictor (random) benchmark.

It is worth mentioning that a study that focuses on this problem (using a different dataset and approach) did achieve a 93.5% accuracy. [18]

A naive predictor would result in an accuracy of approximately 9%, given the prevalence of each instrument in the test data.

Instrument	Sample Count	Chances of choosing an instrument correctly at random
Bass	8	13.11%
Brass	7	11.48%
Flute	2	3.28%
Guitar	8	13.11%
Keyboard	9	14.75%
Mallet	1	1.64%
Organ	11	18.03%
Reed	4	6.56%
String	6	9.84%
Vocal	5	8.20%
	Average	**10.00%**

[17] Note that the data set does not break down the instruments by individual instrument. There are a few reasons for this. One is that instruments in the same family share qualities, like timber and envelope. Another reason is that these instruments have overlapping range. The violin and viola are a good example. The violin has a standard range of G3 to A7 while the viola has a standard range of C3 to E6. That means the viola and violin have overlapping range from G3 to E6, which is quite a large overlap. Their tones (timbres) are different in that the viola has a 'richer' tone, but the differences a subtle enough to make the discrimination of the two instruments a greater challenge.

[18] Five experiments were conducted in this study with a range of 64.2% to 93.5% accuracy: https://arxiv.org/pdf/1705.04971.pdf

III. Methodology

Data Preprocessing and Implementation

Due to the nature of the NSynth data, there were no traditional data preprocessing requirements like one-hot encoding and data cleaning.

However, since we are only using one note (across all instruments) in the dataset for this work, only samples of the note E4 (midi number 64) were extracted from the dataset.

The required TFRecord data had to be used to generate and save the spectrograms into an appropriate file structure, however, these are more implementation details for the next section.

Implementation

Implementation steps:

1. Data Preparation
    a. Convert the TFRecord dataset into a file-based dataset with spectrograms
        i. Read TFRecords
        ii. Filter for only E4-note audio samples
        iii. Generate spectrograms and store in file structure
        iv. Use file structure as datasource
    b. Load the three file-based datasets into a dictionary
    c. Convert the dictionaries into dataframes
    d. Create ImageDataGenerators for each dataframe
2. Run the model
3. Get training results
4. Get testing results

The implementation code was arranged within the following file structure:

- project folder
    - `instrument_identification.ipynb`: The main implementation steps for this project. Each of the steps above are run from this file.
    - `visualizations.ipynb`: Extra visualizations required for this report.
    - `models.py`: The various versions of the CNN labeled, create_model_v1, create_model_v2, etc.
    - `generate_spectrograms.py`: A standalone script to generate spectrograms.
    - `data/`: See 'File Structure as Datasource' below
    - `saved_models/`: The saved weights from the models in models.py, such as weights.best.v1_1.hdf5, weights.best.v2_1.hdf5, etc
    - `utils/`
        - `audio_utils.py`: utility methods for working with and processing audio
        - `general_utils.py`: convenience methods

Data Preparation

The NSynch data comes in two formats: TFRecord files, which are Tensorflow Example protocol buffers [19], and in zipped json/wav files. Both contain the non-audio features as well as the wave audio files. This experiment uses the TFRecord files because they are native to TensorFlow and streaming the data for processing is very efficient.

Three elements are used for the project:

- instrument_family_str: eg. bass, guitar, organ, etc
- note_str: the full descriptive name of the file in the format:
    - `<instrument_str>-<pitch>-<velocity>` where instrument_str is
    - `<instrument_family_str>-<instrument_production_str>-<instrument_name>`
- audio: which is the list of audio samples (floating point values) in the range of -1 to 1

Read TFRecords

The first step is to read the TFRecord files to allow the iterating and processing of the data. Since the model is not using the raw audio data (it is using spectrogram representations of the audio data), the TFRecord files will only be used to unpack the data and process the audio into images.

This is done using the Tensorflow toolset, including tensorflow.Graph() and tensorflow.Session() methods. The data files are opened and iterated in asynchronous batches. [20,21]

The Tensorflow filter method was used to extract only the E4 notes from the dataset. [22]

File Structure as Data Source

The three TFRecord files (train, validate, test) were iterated on and each record was converted to its respective spectrogram in a file format loadable by `sklearn.datasets.load_files`.

The file structure makes up the three data sources to be read by `load_files`:

- data_directory
    - **nsynth-train-spectrograms**
        - string
            - string_acoustic_034-058-075.jpg
            - ...

[19] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto
[20] See https://www.tensorflow.org/guide/datasets for more information on working with Tensorflow datasets.
[21] See https://docs.python.org/3.4/library/multiprocessing.html?highlight=apply_async for more on how to run batches in parallel threads.
[22] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#filter

        - [multiple spectrogram files]
    - ...
    - [multiple instrument families]
- **nsynth-valid-spectrograms**
- **nsynth-test-spectrograms**

The **bolded** directories starting with 'nsynth-' are the root containers of each data set. The blue instrument families are the 'labels' for the dataset. The red file names are the spectrograms for each instrument sample. [23]

Write Spectrograms

Spectrograms are generated from the wave audio data using the LibROSA library. [24]

The Model

A Sequential model was used for all of the tests. There were ten models tested. A four-CPU AWS Deep Learning AMI architecture with Tensorflow compiled for GPU.

The final model tested is made up of nine 2D convolutional layers, uses RMSprop as an optimizer with the following parameters:

- lr=0.005
- decay=1e-6

The loss function is categorical_crossentropy and the metric is 'accuracy'.

NOTE: See `models.py` for the other models.

Model Summary

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_74 (Conv2D)           (None, 256, 256, 32)      896
_________________________________________________________________
activation_83 (Activation)   (None, 256, 256, 32)      0
_________________________________________________________________
conv2d_75 (Conv2D)           (None, 254, 254, 128)     36992
_________________________________________________________________
activation_84 (Activation)   (None, 254, 254, 128)     0
_________________________________________________________________
conv2d_76 (Conv2D)           (None, 252, 252, 128)     147584
_________________________________________________________________
activation_85 (Activation)   (None, 252, 252, 128)     0
```

[23] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_files.html
[24] https://librosa.github.io/librosa/

```
_________________________________________________________________
dropout_48 (Dropout)         (None, 252, 252, 128)     0
_________________________________________________________________
conv2d_77 (Conv2D)           (None, 250, 250, 128)     147584
_________________________________________________________________
activation_86 (Activation)   (None, 250, 250, 128)     0
_________________________________________________________________
max_pooling2d_35 (MaxPooling (None, 125, 125, 128)     0
_________________________________________________________________
dropout_49 (Dropout)         (None, 125, 125, 128)     0
_________________________________________________________________
conv2d_78 (Conv2D)           (None, 125, 125, 64)      73792
_________________________________________________________________
activation_87 (Activation)   (None, 125, 125, 64)      0
_________________________________________________________________
conv2d_79 (Conv2D)           (None, 125, 125, 64)      36928
_________________________________________________________________
activation_88 (Activation)   (None, 125, 125, 64)      0
_________________________________________________________________
conv2d_80 (Conv2D)           (None, 123, 123, 64)      36928
_________________________________________________________________
activation_89 (Activation)   (None, 123, 123, 64)      0
_________________________________________________________________
max_pooling2d_36 (MaxPooling (None, 61, 61, 64)        0
_________________________________________________________________
dropout_50 (Dropout)         (None, 61, 61, 64)        0
_________________________________________________________________
conv2d_81 (Conv2D)           (None, 61, 61, 128)       73856
_________________________________________________________________
activation_90 (Activation)   (None, 61, 61, 128)       0
_________________________________________________________________
conv2d_82 (Conv2D)           (None, 59, 59, 128)       147584
_________________________________________________________________
activation_91 (Activation)   (None, 59, 59, 128)       0
_________________________________________________________________
max_pooling2d_37 (MaxPooling (None, 29, 29, 128)       0
_________________________________________________________________
dropout_51 (Dropout)         (None, 29, 29, 128)       0
_________________________________________________________________
flatten_9 (Flatten)          (None, 107648)            0
_________________________________________________________________
dense_18 (Dense)             (None, 512)               55116288
_________________________________________________________________
activation_92 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_52 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_19 (Dense)             (None, 10)                5130
=================================================================
Total params: 55,823,562
Trainable params: 55,823,562
Non-trainable params: 0
_________________________________________________________________
```

Methodology for Obtaining Training and Testing Results

Loss and Accuracy were obtained using the model's evaluate_generator method. For the training and validation results, the model was provided with the validation data generator and for the testing results, the model was provided with the test data generator.

For prediction, a `run_prediction()` method (in the utils library) was developed. This method runs `predict_generator` on the model (the model is a parameter) and compares the predicted results with the data labels.

Refinement

Out of eight iterations of the model, most variations centered around optimizers and optimizer parameters. The most successful was RMSprop. We tested Adam as well, however, it never yielded stable results. At times it resulted in accuracy scores that far surpassed RMSprop (using the same model structure) with accuracy scores of over .7, but the performance of the model against test data was poor.

Different layer configurations did seem to improve performance, however straying from the (3, 3) kernel size always reduced performance.

Originally, we used the full dataset, but because of the time and expense of testing each model (approximately one hour per epoch) it became obvious that a different approach was in order. The most logical was to limit the dataset by pitch, thus simplifying the problem that the model had to solve and also reducing the resources required to do the work. Some of the models tested were not tested on the reduced dataset due to limited time.

IV. Results

Model Evaluation and Validation

All results are based on the last iteration of the model.

Results on Validation Data:

- Accuracy: 0.151
- Loss: 2.185

Untrained Model Results on Testing Data:

- Accuracy: 0.066
- Loss: 5.310
- 3 correct out of 61

Trained Model Results:

- Accuracy: 0.36
- Loss: 1.87
- 12 correct out of 61

It is unclear why the results for the validation set yields a much lower accuracy than the testing set.

Justification

The metric we are using to evaluate the results is a naive predictor approach. The model is simply the average accuracy that would be achieved if the instrument family classes were guessed at random given the test dataset classification prevelances. (See the Benchmark section above for the distribution.) Using a naive predictor, the expected result would be around 10% accuracy.

Although our CNN model did not achieve a particularly high accuracy score, it did perform better than both a naive predictor and an untrained version of the model.

V. Conclusion

Reflection

The intention of the project was to explore whether musical instrument families could be predicted by a CNN machine learning model. While it appears that a model that analyzes spectrograms for this purpose can boast a performance that surpass a naive predictor, it is still not clear just how accurate a model can be. More work needs to be done in this area.

There were some challenges to the project that must be noted. The first is that the full NSynth dataset was too large to easily test multiple models. Each epoch took over an hour and the four-CPU AWS Deep Learning AMI architecture was too expensive to use liberally. The work-around for this was to use a subset of the data, as described in the Data Preprocessing and Implementation section of this document. In short, rather than using all notes available for each instrument, only one note for each instrument was used. The note chosen was E4 because it is in the middle of the sonic spectrum where most instruments will clearly reflect their characteristics.

Improvement

Although this project is as far as it can go in the context of this capstone, many improvements will be made. This is not an exhaustive list of possibilities, but it is a start.

Synthesized v. Non-Synthesized: Two types of sound samples were provided in the NSynth dataset: synthesized and non-synthesized. Of the non-synthesized sounds, there are acoustic and electronic instruments. It is almost certain that not giving the model a way to distinguish between these three types of sounds caused some ‘confusion’ for the model as the difference between a synthesized flute E4 and an acoustic flute E4 is dramatic.

Transfer Learning: In this project, the models were trained from scratch. There are other CNN pre-trained models that could be leveraged here that may yield better results.

Resources: As stated above, a more substantial platform to train these models would be helpful to reduce the time spent on each model run.