Capstone Project

Machine Learning Nanodegree

Sam Wachtel

Using Deep Neural Network to Predict Musical Instrument Family

I. Definition

Project Overview

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, 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 with domain knowledge (a musician, for example) would have to intervene at this point to separate the notes out into multiple bars.

This project will explore one potential machine learning method to solve this problem.

Problem Statement

The goal is to attempt the construction of a machine learning model that will be able to predict the instrument family of acoustic instruments of a range recorded notes.

The method for this project will leverage a sound visualization technique, called a spectrogram. Spectrograms, which represent audio information visually, will serve as input to train a Convolutional Neural Network to identify the instrument families of audio sound sound samples.

Metrics

Although Precision and Recall will be explored, categorical accuracy will be a good indication of performance because, although the instrument family classes are not evenly distributed, they are distributed in a way that is fairly consistent with real-world instrument popularity.

- Categorical Accuracy: the number of correct predictions divided by the total number of predictions.
- Precision: the number of times an instrument family (flute, for example) is accurately predicted (true positives) over the total number of times the instrument family is predicted (total predicted positives).
- Recall: the number of times an instrument family is accurately predicted (true positives) over the total number of instances of that instrument family (total actual positives)

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. [1] 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

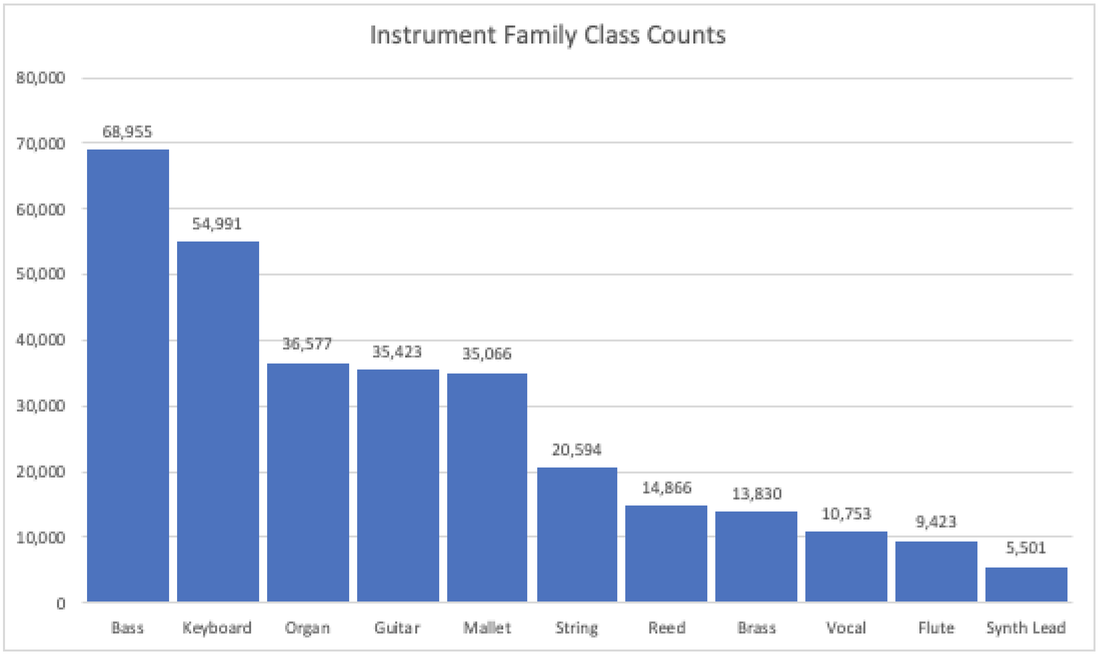
[1] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders." 2017.

    - 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. It is divided into training, validation, and testing sets with 289,205, 12,678, and 4,096 records in each, respectively.

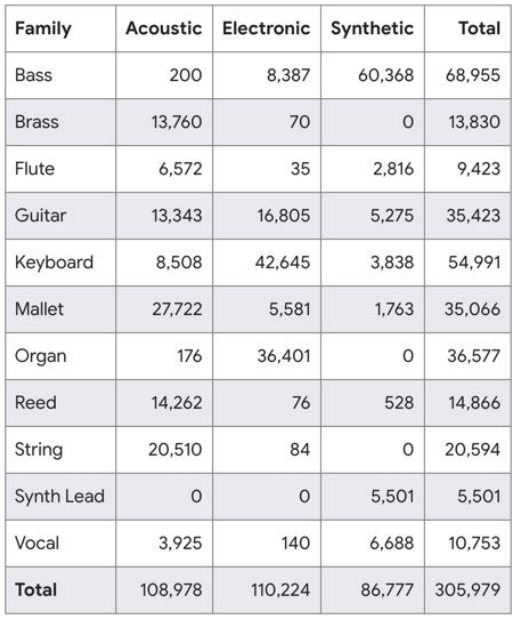
Class Distribution of Dataset

Total Sound Samples in Dataset: 305,979



Breakdown of Instruments by type of sound (acoustic, electronic, etc): [2]

[2]Chart from: https://magenta.tensorflow.org/datasets/nsynth#files



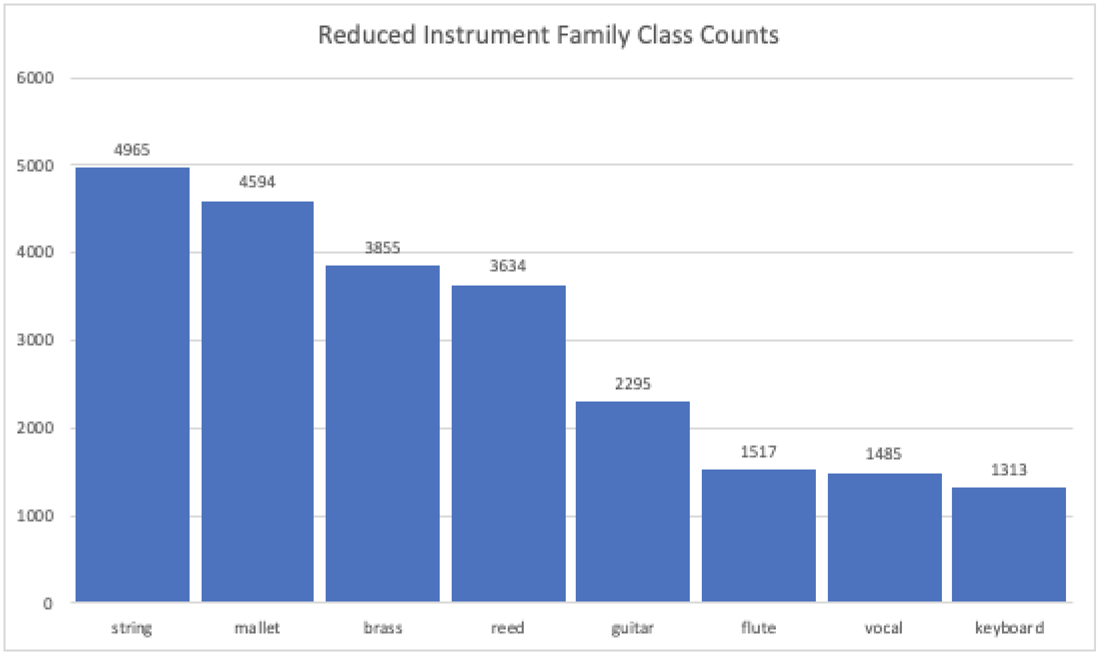
Family	Acoustic	Electronic	Synthetic	Total
Bass	200	8,387	60,368	68,955
Brass	13,760	70	0	13,830
Flute	6,572	35	2,816	9,423
Guitar	13,343	16,805	5,275	35,423
Keyboard	8,508	42,645	3,838	54,991
Mallet	27,722	5,581	1,763	35,066
Organ	176	36,401	0	36,577
Reed	14,262	76	528	14,866
String	20,510	84	0	20,594
Synth Lead	0	0	5,501	5,501
Vocal	3,925	140	6,688	10,753
Total	108,978	110,224	86,777	305,979

Subset of Data

Due to the size of the NSynth set, and also due to it including two categories of instruments that might not naturally be considered traditional instruments, we are using a subset of the entire data set. This subset includes only acoustic instruments within the range of C4 to C5 (midi notes 60 - 72).

The reason for limiting the set to a one-octave (plus one) range is to both simplify the learning task and to further reduce the processing time. The reduced set also excludes the bass and organ families as they do not have enough instruments in the training, validation, and testing datasets to properly be included.

The subset after limiting it to the above specifications represents a total size of 23,658 sound samples, which is greatly reduced from 305,979.



TFRecord Files

TFRecord [3] 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:

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

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

`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.[4] Each of these samples can be thought of as a mathematical snapshot in time of the sound. The rate in which the samples are taken is called the sample rate, also known and Hz and "samples per second" [5]. The most common method of representing analog sound digitally is called pulse-code modulation (PCM).[6] [7]

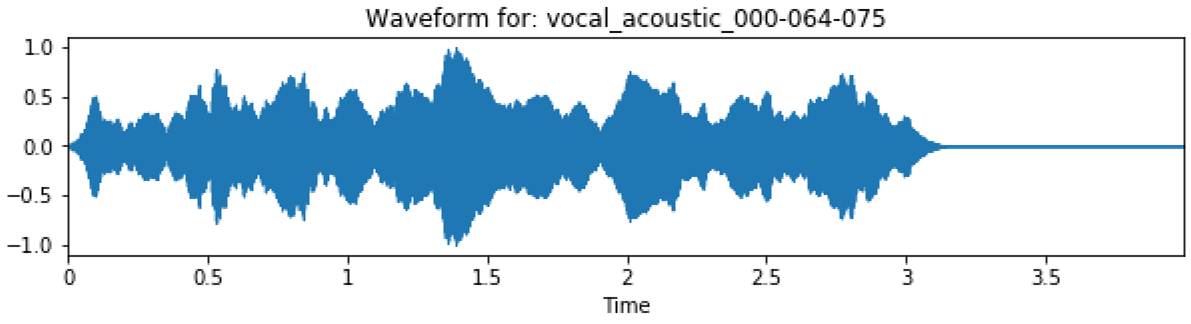
The audio files in our dataset have the following attributes:

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]

[4] https://en.wikipedia.org/wiki/Analog-to-digital_converter
[5] https://en.wikipedia.org/wiki/Sampling_(signal_processing)#Sampling_rate
[6] https://en.wikipedia.org/wiki/Pulse-code_modulation
[7] https://www.presonus.com/learn/technical-articles/sample-rate-and-bit-depth

Waveform

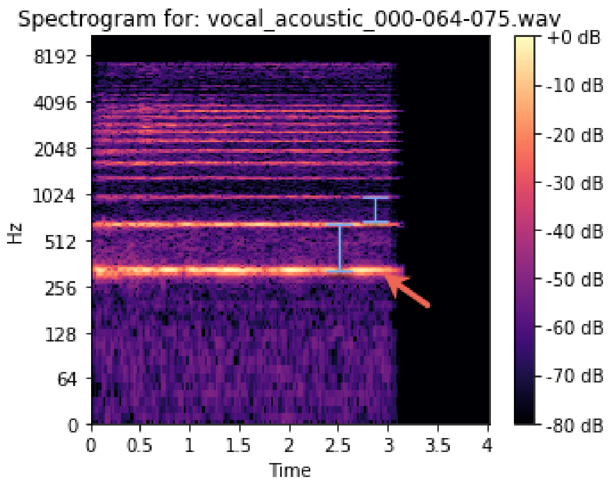
Audio is usually represented as a waveform. The following is a four second audio clip of human voices singing a single note. The x axis is time and the y-axis is the amplitude of the sound sample at that point in time.

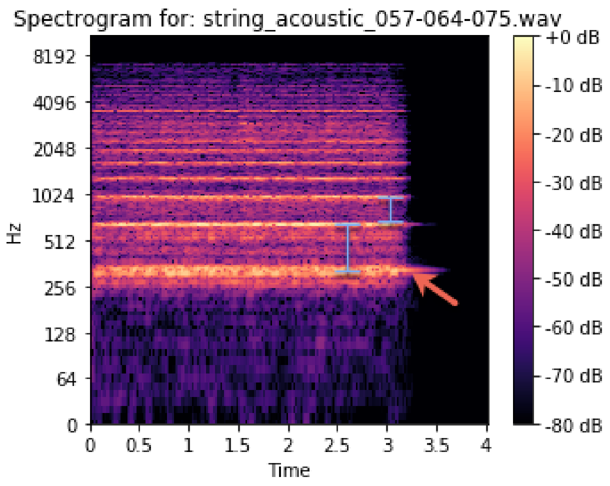


This type of basic waveform is often used by audio engineers to understand the timing and volume (loudness) of the sound. This visualization is limited, however, in that it does not express other information like frequency, frequency intensity, or note overtones. Notice that there is no way of telling 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).[8,9] See TFRecord Files section above for more details on note information that comes packaged with 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.





[8] https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies

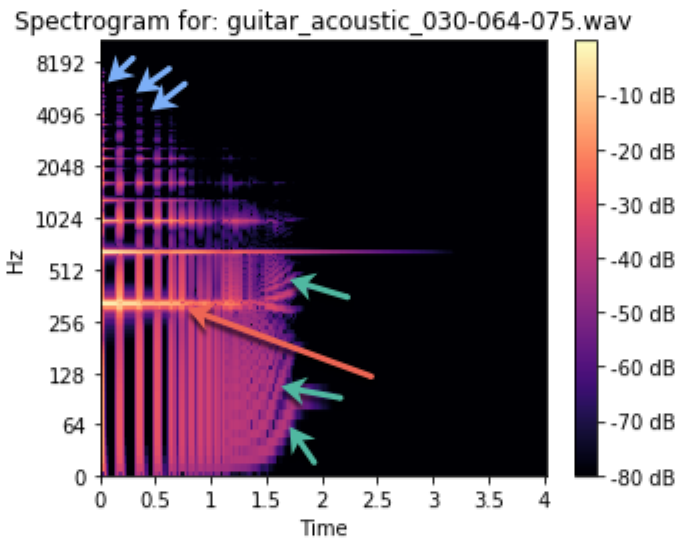
[9] The instrument names are formatted: <instrument_str>-<pitch>-<velocity>. See https://magenta.tensorflow.org/datasets/nsynth for more information about the dataset.

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.[10]

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*.[11] 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.

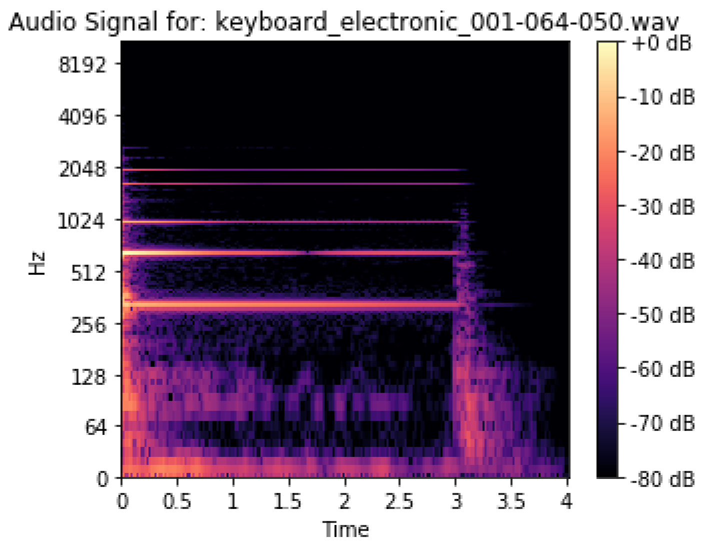
The other aspect feature of sound that differentiates one instrument's timbre from another is *sound envelope*. The envelope describes the attack (start), sustain (middle), decay (decline), and release (end) of that sound. [12]



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

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

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.[13] 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.



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. [14]

[13] 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.

[14] 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.

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. [15,16] 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 tools (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 used a set of techniques partially inspired by a work by Adrian Yijie Xu.[17] Xu used a CNN to detect various urban sounds found in the Urban Sound Dataset.[18]

As stated in the Data Exploration section of the document, we are limiting the dataset to only acoustic instruments playing notes in the range of C4 to C5 (midi notes 60 - 72). The limited range should simplify the model's learning process while the reduced dataset size will speed up processing.

The approach used in this experiment is to convert the reduced NSynth Dataset to Mel-scaled Spectrograms and use the training and validation sets to train a Convolutional Neural Network to recognize instrument families. [19]

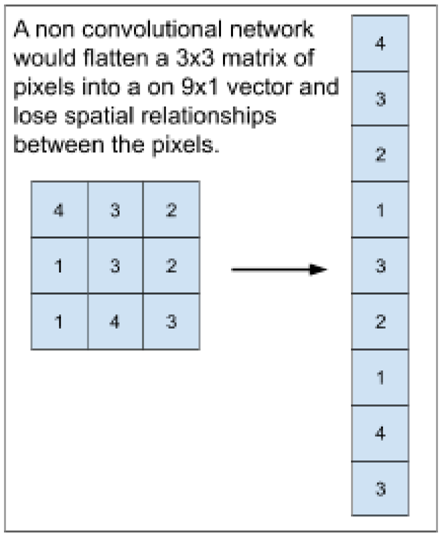
Convolutional Neural Networks (CNN)

CNNs are a class of Deep Neural Networks often used for classifying objects in image data. CNNs, in large part, are to thank for image search and classification capabilities in products such as Google Image Search[20] and Google Photos[21].

[15] https://en.wikipedia.org/wiki/Mel_scale
[16] https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0
[17] Urban Sound Classification using Convolutional Neural Networks with Keras: Theory and Implementation, February, 2019
[18] https://urbansounddataset.weebly.com/urbansound8k.html
[19] 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.
[20] https://images.google.com/
[21] https://www.google.com/photos/about/

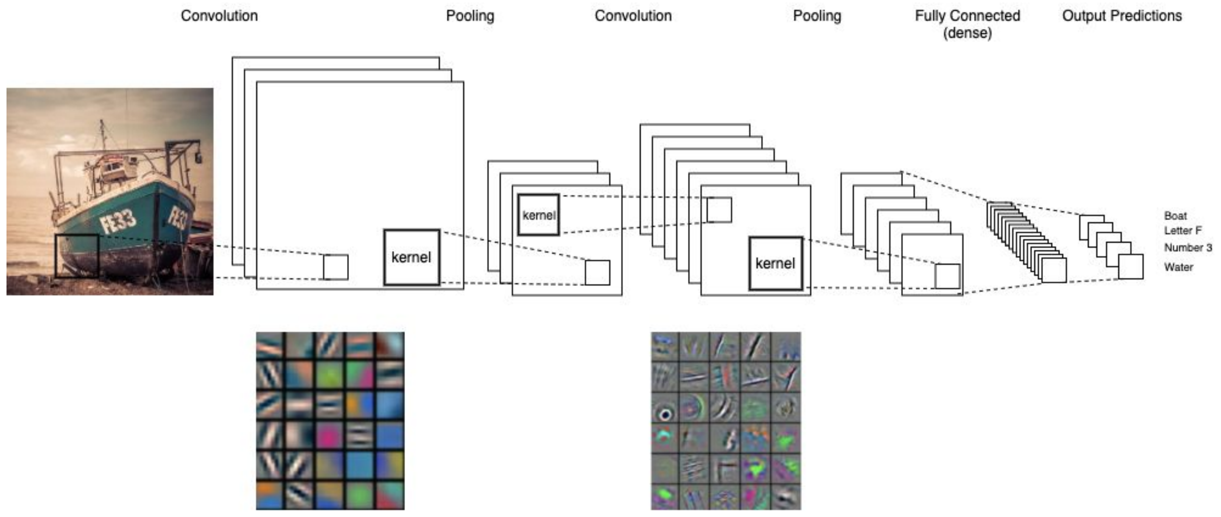
One of the aspects of this project that make it so interesting is that while a human that has a lot of experience in referencing spectrograms might be able to detect features in those images that reflect a certain type of instrument, most people would not easily be able to do it consistently across all instrument types. Furthermore, a person that has no experience interpreting spectrogram images would not be able to identify anything at all. Therefore, a CNN that can learn to identify instrument families by spectrogram analysis is next to magic because it can do something a human can not easily do.

CNNs start with the input of an image. Unlike with a non convolutional neural net, the data is not flattened into a single dimension vector as illustrated on the right. This is an important distinction between CNNs and other neural network model types. CNNs use image spatial information (what pixels are in proximity to other pixels) to learn features.


A non convolutional network would flatten a 3x3 matrix of pixels into a on 9x1 vector and lose spatial relationships between the pixels.
4
3
2
1
3
2
1
4
3

CNNs are organized by layers. The first layer takes the image as input. The next layer processes the output of the first layer. And so on, each layer in the chain processes the image information it receives from the layer before it. Early layers identify general features in the image (like vertical and horizontal lines). The next layers identify more complex features (like circles, grids, shapes). Layers toward the end of the model identify complex features (like water, a boat, a number, etc). This is done in stages, or layers, within the neural network.

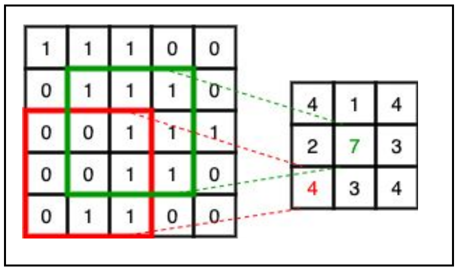
The illustration reflects the general idea of how the image data is processed and changes shape with each layer in the chain.



Convolutional layers detect regional patterns in an image. They work by sliding a window (aka, kernel) over the entire image from top-left to the bottom-right by going right in strides (size steps, as in 2 pixels) until it reaches the right side of the image and then moving down a stride it starts again from the left and repeats until it reaches the bottom-right. The kernel (window) size and stride are part of the definition of each convolution layer.

At each position a calculation happens that takes the learned weights into account. In an untrained model, all weights are randomly set. During model training, each filter's weights are refined with each training cycle.

Images lose height and width at each convolution layer. If the kernel size is 3 (3x3), it covers a 9-pixel grid. That means a 25 pixel grid would be reduced down to a 9 pixel grid after a single convolution. Below is an example of a 25 pixel image being convolved into a 9 pixel image. Two positions (red and green) are illustrated.



As an image loses height and width, it gains depth as more filters defined for each convolutional layer. These layers commonly have tens or even hundreds of filters, making them quite deep. Each filter represents a kind of object, like a mouth, nose, or eyes.

The output layer is called a Dense layer. It is used to take the now small, but deep, image data as input and convert it into a set of feature probabilities. The Dense layer is configured with the number of dimensions in the output space. For example, if the model were classifying hand-written single digits, the number of dimensions would likely be 10 (zero through nine). The output would be an array of probability assignments, one for each digit. The probabilities for all of the 10 dimensions add to 1 and the dimension with the highest probability is the 'prediction'.

In the case of this work, the final Dense layer is configured for eight (8) dimensions. See section III for details on the model itself.

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. [22]

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

A naive predictor would result in an accuracy of approximately 12.5%, given the prevalence of each instrument in the test data.

Instrument	Sample Count	Chances of choosing an instrument correctly at random
Brass	71	20.88%
Flute	6	1.76%
Guitar	81	23.82%
Keyboard	21	6.18%
Mallet	32	9.41%
Reed	68	20.00%
String	51	15.00%
Vocal	10	2.94%
	Average	**12.50%**

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 using a subset of notes across just the acoustic instruments in the dataset, we only extracted sound samples of notes from C4 to C5 (midi notes 60 - 72).

The training, validation, and testing TFRecord data files were used to extract the sound data and generate the spectrograms. The images were stored in a file structure suitable for using as a dataset. Implementation details below.

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 acoustic note audio samples in the MIDI range of 60 - 72
        iii. Generate mel-scaled spectrograms and store in file structure

            iv. Use file structure as datasource
    b. Load the three (train, validation, test) file-based datasets into a dictionary
    c. Convert the dictionaries into dataframes
    d. Create ImageDataGenerators for each dataframe
2. Repeat until satisfactory results are attained
    a. Run the model
    b. Get training results
    c. 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 [23], 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

[23] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto

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. [24,25]

The Tensorflow filter method was used to extract only the required samples from the dataset using the below regex filter. [26]

```
regex_filter = '.*acoustic.*-0(6[0-9]|7[0-2])-.*'
```

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
            - …
            - [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. [27]

[24] See https://www.tensorflow.org/guide/datasets for more information on working with Tensorflow datasets.
[25] See https://docs.python.org/3.4/library/multiprocessing.html?highlight=apply_async for more on how to run batches in parallel threads.
[26] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#filter
[27] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_files.html

Write Spectrograms

Spectrograms are generated from the wave audio data using the LibROSA library. [28]

The Model

A Sequential model was used for all of the tests. There were fourteen models tested that gave distinctly different results. The method used for experimenting with models was to take the previous model tested and tweak it until a difference in result was observed.These n-between models did not get saved as there was no real benefit to keeping them.

A four-CPU AWS Deep Learning AMI architecture with Tensorflow compiled for GPU was used for all training.

The Sequential model below is the one that produced the best results (see section IV for results). It is the 13th model in the set of fourteen saved models and has seven convolutional layers separated out by dropout layers and ReLU activation layers. See *models.py* for all of the tested models.

```
model = Sequential()
model.add(Conv2D(32, (8, 8), padding='same', input_shape=(21,20,3)))
model.add(Activation('relu'))
model.add(Conv2D(64, (8, 8)))
model.add(Activation('relu'))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(8, activation='softmax'))

model.compile(optimizers.RMSprop(lr=0.0005, decay=1e-5),
   loss="categorical_crossentropy", metrics=["categorical_accuracy"])
```

The first convolutional layer takes a `(21,20,3)` shape. (The spectrogram widths are slightly bigger than their height.) The Keras ImageDataGenerator was configured with a matching target size of

[28] https://librosa.github.io/librosa/

`(21,20)`. Other shapes tried were `32,32` and `64,64` however, both were slower and yielded worse results.

The first two convolutional layers are set to a kernel size of `(8,8)`. This produced better results than `(3,3)`.

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

- lr = 0.0005
- decay = 1e-5

See the Refinement section below for a short discussion on how different learning and decay rates and different loss and optimizer methods affected model performance.

Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_63 (Conv2D)           (None, 21, 20, 32)        6176      
_________________________________________________________________
activation_72 (Activation)   (None, 21, 20, 32)        0         
_________________________________________________________________
conv2d_64 (Conv2D)           (None, 14, 13, 64)        131136    
_________________________________________________________________
activation_73 (Activation)   (None, 14, 13, 64)        0         
_________________________________________________________________
dropout_36 (Dropout)         (None, 14, 13, 64)        0         
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 14, 13, 64)        36928     
_________________________________________________________________
activation_74 (Activation)   (None, 14, 13, 64)        0         
_________________________________________________________________
conv2d_66 (Conv2D)           (None, 12, 11, 64)        36928     
_________________________________________________________________
activation_75 (Activation)   (None, 12, 11, 64)        0         
_________________________________________________________________
conv2d_67 (Conv2D)           (None, 10, 9, 64)         36928     
_________________________________________________________________
activation_76 (Activation)   (None, 10, 9, 64)         0         
_________________________________________________________________
dropout_37 (Dropout)         (None, 10, 9, 64)         0         
_________________________________________________________________
conv2d_68 (Conv2D)           (None, 10, 9, 128)        73856     
_________________________________________________________________
activation_77 (Activation)   (None, 10, 9, 128)        0         
_________________________________________________________________
```

```
conv2d_69 (Conv2D)           (None, 8, 7, 128)        147584    
_________________________________________________________________
activation_78 (Activation)   (None, 8, 7, 128)        0         
_________________________________________________________________
dropout_38 (Dropout)         (None, 8, 7, 128)        0         
_________________________________________________________________
flatten_9 (Flatten)          (None, 7168)             0         
_________________________________________________________________
dense_18 (Dense)             (None, 512)              3670528   
_________________________________________________________________
activation_79 (Activation)   (None, 512)              0         
_________________________________________________________________
dropout_39 (Dropout)         (None, 512)              0         
_________________________________________________________________
dense_19 (Dense)             (None, 8)                4104      
=================================================================
Total params: 4,144,168
Trainable params: 4,144,168
Non-trainable params: 0
_________________________________________________________________
```

Methodology for Obtaining Training and Testing Results

Loss and Categorical 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 implemented. This method runs `predict_generator` on the model (the model is a parameter) and compares the predicted results with the data labels.

Refinement

Model variations across fifteen (15) iterations of the model included changes to the number of layers, the initial input size and shape, kernel sizes of convolutional layers, various dropout configurations, optimization and loss methods, and more.

The most successful model as described above was two percent better than the best before it. Due to changes in overall methodology along the way (like limiting the data rather than using the whole set) refinement of the model itself is hard to judge from version nine, back.

RMSprop seemed to work better for this learning task than Adam. Introducing Adam as the optimizer always resulted in ten or more percent degradation in performance. More experimentation should be done here.

Learning rate (lr) also made a big difference in the outcome. We found a 0.0005 learning rate to be optimal. 0.0007 and 0.005 were both appeared to cause the model to choose between one and four classes of the 8.

The decay rate was tested at 1e-4 and 1e-6. Although both are associated with models with worse results, other aspects of the model were unfortunately different, so it can not be stated that those rates are wrong for the model. Again, more experimentation with decay is required.

The loss function is `categorical_crossentropy` and the metric is `categorical_accuracy`. The combination of these yielded the best results. categorical_hinge and accuracy were both used in previous iterations.

Different layer configurations seem to improve performance, however an (8, 8) kernel size for the first two convolutional layers appears essential to maintain the current performance performance.

And as stated above, we originally 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 range and instrument type (acoustic). This both simplified the problem and reduced 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

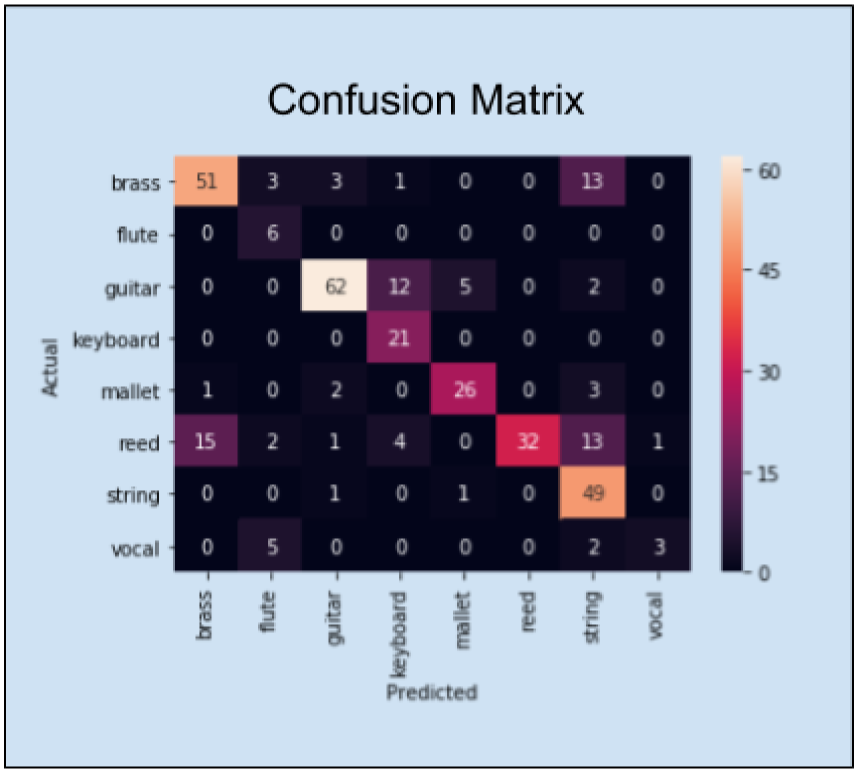
Model Evaluation and Validation

The fifteenth (15) model scored the best with a categorical accuracy of 73.5%

Loss: 0.9298041936229257
Categorical Accuracy: 0.7352941

Total test records: 340
Number of correct: 250

As the confusion matrix illustrates, the model did better with some instrument families than it did with others.



Precision and recall scores for the instrument families:

Family	**Precision**	**Recall**
Brass	77%	80%
Flute	38%	100%
Guitar	90%	77%
Keyboard	55%	100%
Mallet	81%	81%
Reed	100%	47%
String	60%	96%
Vocal	75%	30%

It is interesting to note that for Flute and Keyboards (and almost Strings) recall is 100% and precision is low. For Reeds, it’s the inverse, with Precision at 100% and Recall being low. And for Mallets, Precision and Recall scores are absolutely equal.

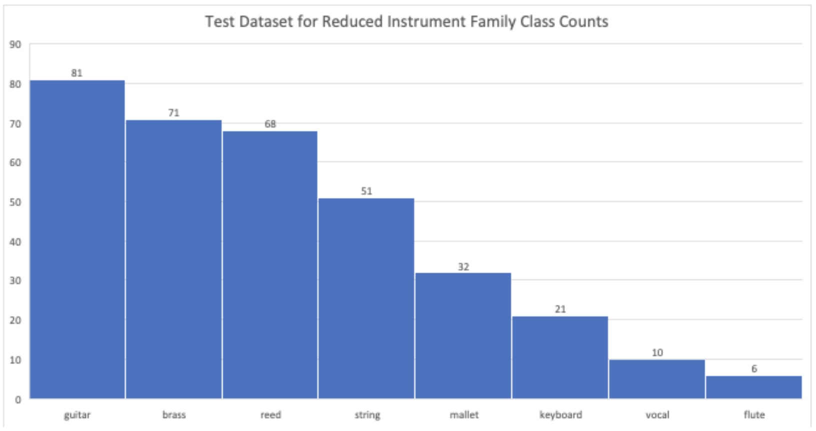
Justification

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 12% accuracy. Our model has far surpassed a 12% accuracy with a 74% accuracy.

V. Conclusion

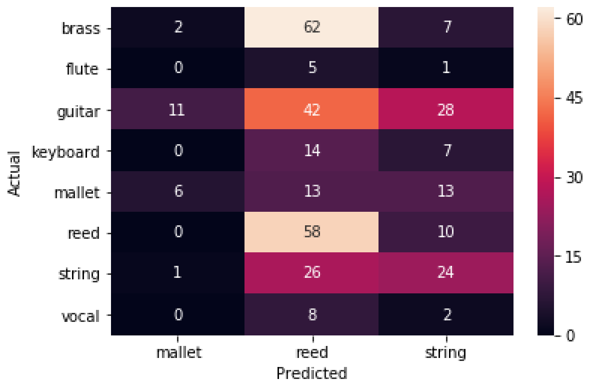
Free-Form Visualizations

Distribution of the Test Dataset



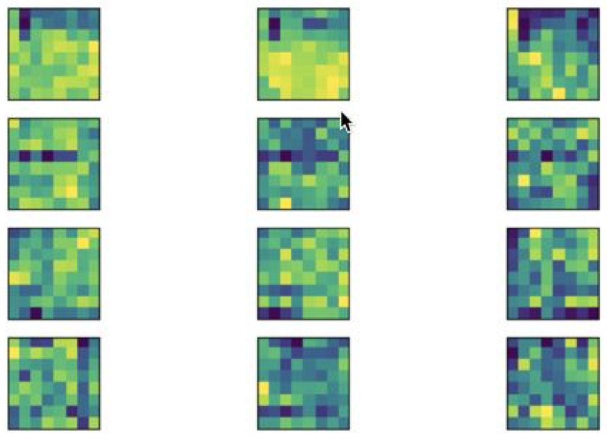
Confusion Matrix of the Untrained Model

Running the testing data through the untrained model yielded interesting results. The predictions are clearly not random and there is something about the reed and string instrument families that caused the untrained model to focus on them. They are not the most prevalent families and they don't sound alike.

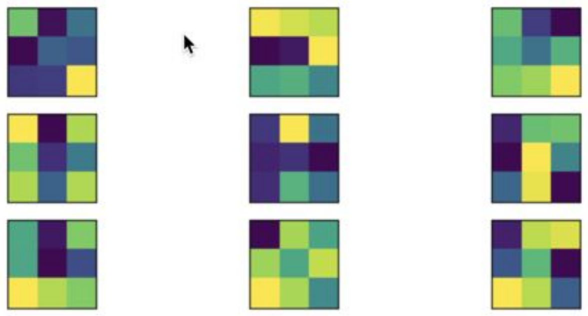


Convolution Filter Visualizations

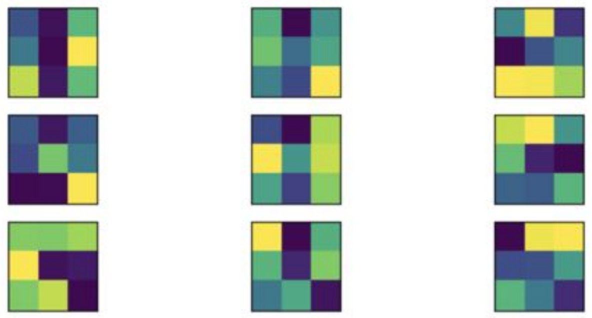
The first 12 8x8 kernel filters of the first Conv2D layer.



The first 9 3x3 kernel filters of the third Conv3D layer.

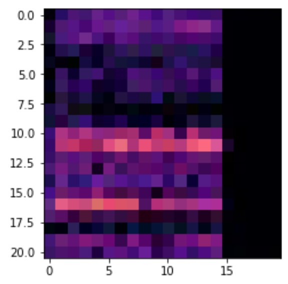


The first 9 3x3 kernel filters of the seventh Conv3D layer.

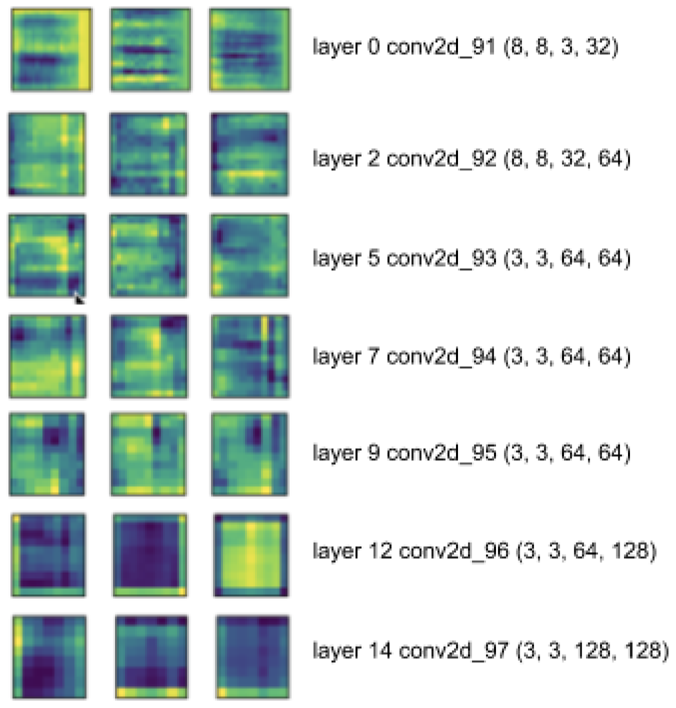


Convolution Feature Maps

Using vocal_acoustic_000-064-075.jpg spectrogram as an example.



The first three feature maps on each of the convolutional layers.


layer 0 conv2d_91 (8, 8, 3, 32)
layer 2 conv2d_92 (8, 8, 32, 64)
layer 5 conv2d_93 (3, 3, 64, 64)
layer 7 conv2d_94 (3, 3, 64, 64)
layer 9 conv2d_95 (3, 3, 64, 64)
layer 12 conv2d_96 (3, 3, 64, 128)
layer 14 conv2d_97 (3, 3, 128, 128)

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 it is possible to predict the instrument family using this machine learning methodology, 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, a subset of acoustic instrument sounds was chosen within a tonal range of just above one octave (13 tones).

With regard to the precision and recall results, there were some interesting patterns that would be interesting to explore further. Why were some instrument families easy for the model to identify (recall) while others were not and why did those with high recall scores have such poor precision scores? And inversely, why did the model find it easy to avoid falsely identifying some instrument families?

Improvement

Although this project is as far as it can go in the context of this capstone, a number of improvements will be made.

- While we reached an accuracy score of 74%, it is fairly clear that with continued tweaking, better performance can be achieved.
- Expanding on the tonal range of the selected instruments incrementally to see if certain ranges or instrument types (like synthetic) will throw off the model.
- 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 interesting 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.