URBAN SOUND CLASSIFIER

Using Deep Learning to classify sounds of the city

Spencer Goble

03.14.2022

Capstone 3 White Paper

Data Science Career Track

Springboard

[Project Repository]

INTRODUCTION

Considering my background in sound engineering, working with audio in the realm of machine learning is of great interest to me. I am familiar with the characteristics of sound and curious to dive more into how the 'personality' of audio can be extracted and analyzed by employing a Neural Network.

In my hunt for a dataset I came across Urban Sound 8k. Some interested gentlemen from NYU had collected over 8,000 urban sounds and wrangled them into a clean, labeled database. Being that this project was to be more about exploring audio feature extraction and deep learning, starting with a clean dataset was fine with me.

DATA

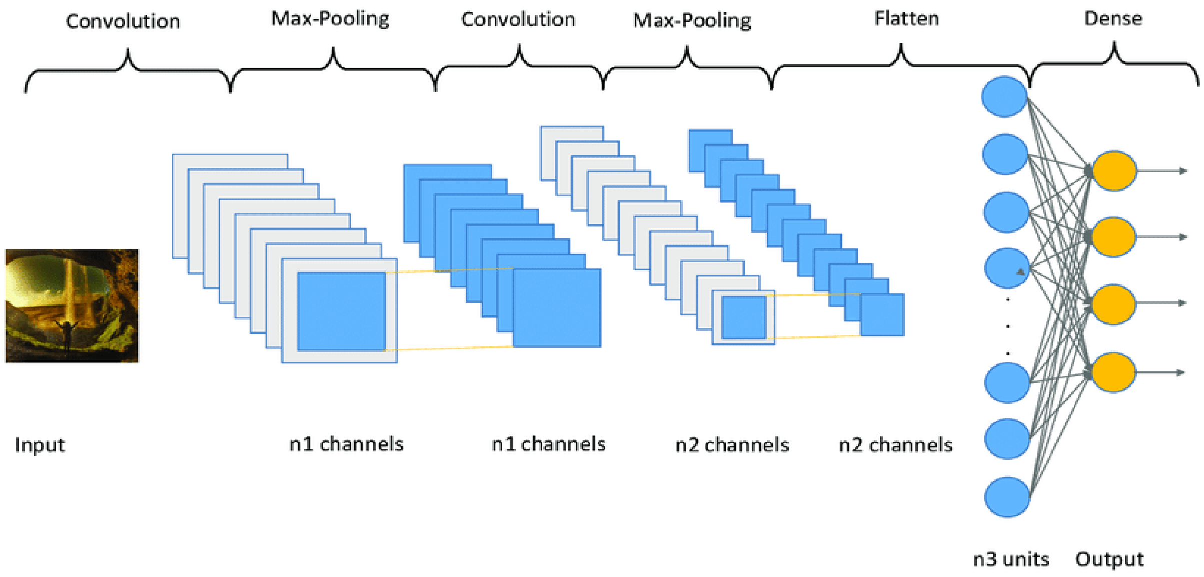
The dataset consisted of 8,732 sound excerpts, each one being no longer than 4 seconds in duration. Each sound belonged to one of ten 'classes' or type of sound. The classes were as follows:

1. Air conditioner
2. Car horn
3. Children playing
4. Dog bark
5. Drilling
6. Engine idling
7. Gun shot
8. Jackhammer
9. Siren
10. Street music

Each sound was in the .wav format, which is a lossless digital file format. Lossless means the formatting doesn't compress the sound which in turn degrades the quality. Mp3 is an example of a 'lossy' format, in which the audio quality suffers. The sample rate, bit depth, and number of channels (mono vs. stereo) varied from sound to sound because the sounds were originally collected in different contexts. The dataset came with the audio files accompanied by a .csv file that contained information about the sounds, including the class name and id.

APPROACH

My primary approach was using various methods to extract tone and amplitude based characteristics from the sounds using mathematical functions. Many of these methods used for audio feature extraction create an image of the sound. Having a graphical representation of the sound then allows us to essentially employ an image classification algorithm. **Convolutional Neural Networks** are the most suited tool for image classification in the realm of deep learning.



Libraries:

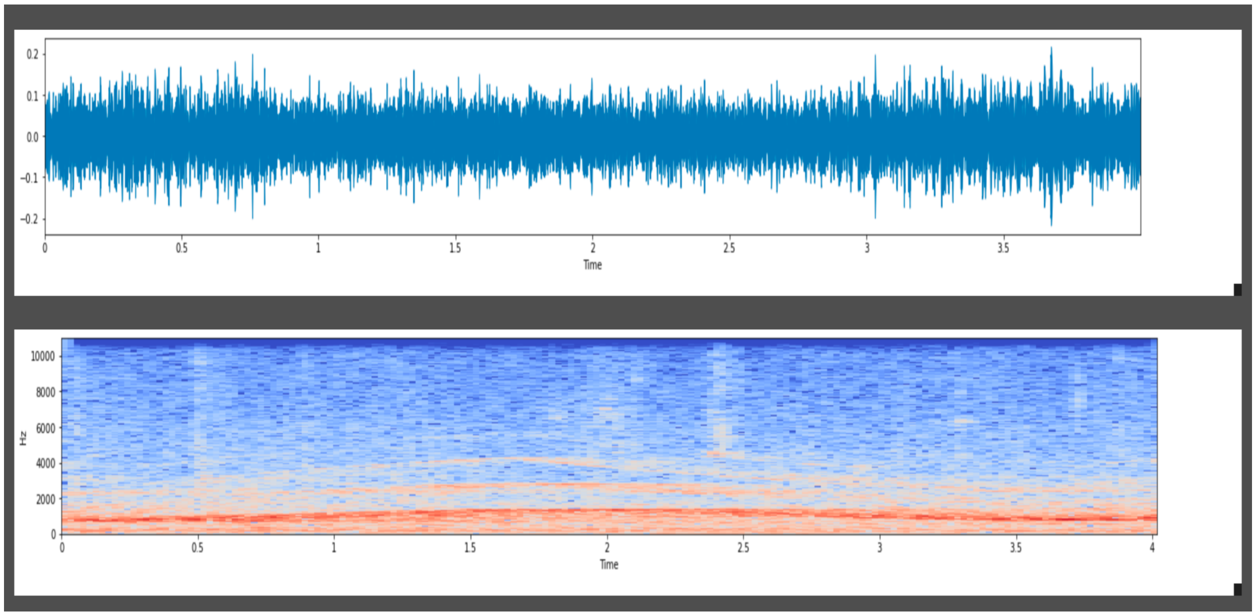
After deciding on my project and locating the dataset, I knew I needed a way to manipulate large arrays, create formatted data frames, visually analyze data, and invoke a plethora of machine learning algorithms. To construct my tool kit I leveraged the following libraries:

1. **Numpy** - Supports large multidimensional arrays and matrices. Allows for quick computation of statistical metrics. Formats datasets for modeling.
2. **Pandas** - Creates well formatted data-frames that allow for data organization and manipulation.

3. **SciKit Learn** - An extremely robust collection of machine learning algorithms and tools.
4. **Librosa** - A tool-kit specifically designed for working with audio. The gold standard for sound based feature extraction.
5. **Keras -** An API that provides a Python interface for implementing Neural Networks. Built on top of Tensorflow.
6. **Matplotlib** - A very robust toolkit for creating readable plots and graphs.

EXPLORATORY DATA ANALYSIS

Being that the data arrived in a cleaned-up form, there wasn't much wrangling to be done. The first task at hand was two load and preview some audio samples. I employed the **Librosa** library to load one sound of each class - ten sounds in total. I generated an audio waveform and a spectrogram for each sound. An *audio waveform* is a graph of a line undulating through peaks and valleys, where the spikes and dips represent amplitude. The X-axis is time and Y-axis is amplitude or what we could refer to as 'loudness'. A *spectrogram* displays frequency content. The X-axis represents time and the Y-axis represents frequency in Hertz. The color represents the presence of sound at a given frequency. The lighter the color is, the greater the amplitude of sound is at that frequency. The image below is the waveform(top) and spectrogram(bottom) of a ***siren*** sound. In the bottom image you can see the lightly colored line slowly move up at around 1 second and then move back down after 2 seconds. This curve represents how a siren slowly moves up and then down in pitch.



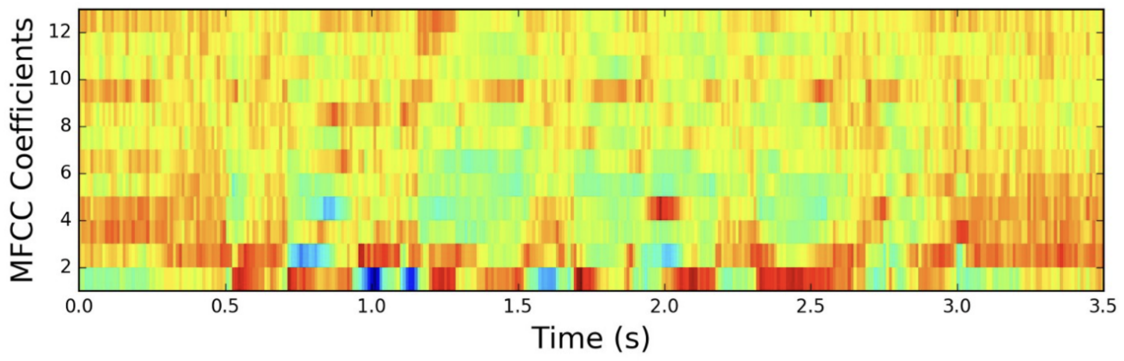
I then extracted the sample rate, bit depth, and channel count for each sound. When loading a sound, librosa automatically adjusts the sample rate to 22,050 frames per second and the channel count to 1 (mono). This makes subsequent processing easier because the sounds get normalized as they are loaded. Bit depth represents the quality of the audio, or how much information is contained in each sample frame. The standard CD quality bit depth is 16. When I observed the bit depth for each sample I noticed that some of them had a bit depth of 4, which is very low. I listened to those sounds (there were only eight) and sure enough they were very poor quality, almost completely unidentifiable as a particular sound. Being that eight sounds isn't a significant portion of the sample space, I dropped them from the dataset.

FEATURE ENGINEERING

After gaining a sense of the audio I was working with, it was time to decompose the sounds and generate some features; *numerical audio*. The Librosa library is very developed and loaded with a vast array of functions for pulling out the characteristics of sound. After doing a lot of research I settled on three main methods:

Mel-Frequency Cepstrum:

The Mel-Frequency Cepstrum (yes, that's *spectrum* with the first half reversed) of a sound is the collection of **Mel-Frequency Cepstrum Coefficients(MFCC's)**. This method makes use of the **Mel Scale,** which is a frequency measurement based on how humans perceive pitch. The idea is that we hear certain pitches more distinctly than others even if they are played at the same gain level. This perceptual scale makes a sort of bell curve, where very low and very high frequencies are harder to distinguish than ones in the midrange. So two frequencies in the midrange, like 900 and 1,000 Hertz will sound farther apart (or more distinct) from each other than will 15,000 and 15,100 Hertz. A logarithmic formula is used to convert Hertz into Mels. Once the Mel-Frequency Cepstrum is created it gets decomposed into coefficients (MFCC's). These numbers represent the subtle variations in amplitude that characterize the Mel-Frequency representation of the sound. What we get as a result is a highly detailed graphical representation of the audio clip. Below is a visualization of MFCC's:



Chroma Features:

The Chroma Features, otherwise known as **Chromagrams**, relate more closely to the 12 pitches of the Western Music scale. This function decomposes the sound and produces a numerical representation of which note each frame of the sound correlates to. Chromagram extraction depends on a more primal method called the **Fourier Transform(FT)**. This mathematical operation transforms a time based function into a temporal-frequency based function, which can then be used to decompose sound.

Zero Crossing Rate:

The Zero Crossing Rate is the measure or count of times that the amplitude of a signal crosses through the zero value in a given time interval. It is very useful in identifying percussive sounds because it is sensitive to transients, and the ADSR characteristics of sound; that is, quick sudden strikes vs. slow rising sound or sound that slowly tails off vs. sound that ends abruptly. This method is very popular for speech recognition and sound identification in general.

PRE-PROCESSING

The main preprocessing task was to reshape the feature array. My features were all consolidated into one column in the dataframe during the Feature Engineering step. I took that column and made it into the datatype 'list' and then made it back into an array. I then created another dimension by using '.reshape'. At that point the shape of my feature variable was (40, 174, 1); 40 rows, 174 columns, and 1 channel for each audio clip in the dataset. This reshaping was necessary because of how CNN's use a kernel to scan each image. I also label-encoded the class or target feature, which is what I was ultimately trying to predict. The target feature was also reshaped to fit into the final output layer of the CNN model where the prediction occurs.

MODELING

Model Architecture:

First I began by instantiating a **Sequential** model in Keras, which is a model type that allows for linear, layer by layer construction. The first layer I added was a **Conv2D** layer, which is the application of a filter to an input (in our case an image of sound), which then creates a map that summarizes the presence of certain features.

The first parameter in the Conv2D layer is ***filters*** which allows us to set the number of kernels/filters to be applied to our input(image). It is convention to choose a power of 2 as the filter count. This is because it is computationally costly to pin down the exact optimal batch size, so stepping in powers of 2 is a logical way to exponentially reduce the search space and still arrive close to the optimum value. I set the first Conv2D layer to a filter count of 16.

The next parameter is ***kernel_size***. This value is a 2-dimensional figure that determines the height and width of the filter. It is common to use odd numbers because this allows there to be a 'center' pixel in the layer. If we imagine a 3x3 grid there is one central square, whereas a 2x2 grid there is not. Odd kernel sizes are used for implementation simplicity. Typically if an image has dimensions less than 128x128 a kernel size of 3x3 or 1x1 is used. I set the *kernal_size* parameter to 3x3 in the first layer.

The next parameter to address is ***activation***. Activation functions make back-propagation possible. Back-propagation is an extremely important process in neural networks; it allows the model to update the weights and biases of the neurons on the basis of the error at the output. I chose to use the ***'ReLU'*** function which stands for ***Rectified linear unit.*** The ReLU function is a very popular choice and proves advantageous because it does not activate all the neurons at the same time making the network sparse and in turn making computation efficient.

The second layer is called a **MaxPooling2D** layer. *Pooling layers* downsample the feature maps created by colvolving. They create small changes in the location of the feature in the input detected by the convolutional layer. Practically speaking, the point of pooling layers is to enable the model to adapt to subtle variations in the features of the input data. The convention is to use a 2x2 pooling window which reduces the size of each feature map by a factor of 2.

The next layer added is called a **Dropout** layer. The purpose of this layer is to prevent all neurons in a layer from synchronously optimizing their weights. In turn, this prevents the neurons from converging to the same goal. In essence this process is implemented to prevent the model from *overfitting*. The parameter of this layer takes a float between 0 and 1 as a percentage of the input units to drop out.

This group of layers (Conv2D, MaxPooling2D, Dropout) is repeated again with the same parameters except the filter count increases from 16 to 32. Then another Conv2D layer is added where the filter count is doubled again from 32 to 64. After the 3rd Conv2D layer I added a **Flatten** layer which flattens the multi-dimensional input tensors into a single dimension. This is needed in order to implement a Dense layer.

A **Dense** layer is a simple layer of neurons in which each neuron receives input from all the neurons in the previous layer. I implemented one dense layer after flattening the input tensors. The primary parameter for a dense layer is *'units'* which sets the dimensionality of the output space. In this case I set *units* to 64, matching the number of filters from the previous Conv2D layer. The *'activation'* parameter was set to 'ReLU' matching the previous activation settings.

Our final **output layer** is also a **Dense** layer; the final layer is responsible for performing the actual classification task. The *'units'* parameter is set to the shape of our target variable array. In this case it's 10 because there are 10 classes to which any given sound can belong. The *'activation'* parameter was set to ***softmax*** which assigns decimal probabilities to each class. So for every sound there is a likelihood metric for each class that it could potentially belong to. Softmax is a popular choice for multi-class classification problems such as this one.

Compiling & Fitting:

After building our model, the next step is to *'compile'* it. This is where we define the optimizer, loss function, and metrics. **Optimizers** are methods used to alter the attributes of the model such as weights and learning rates in order to reduce the losses. I chose to use the ***Adam*** optimization method which uses a process called stochastic gradient descent. It's a very popular algorithm due to its efficiency and accuracy.

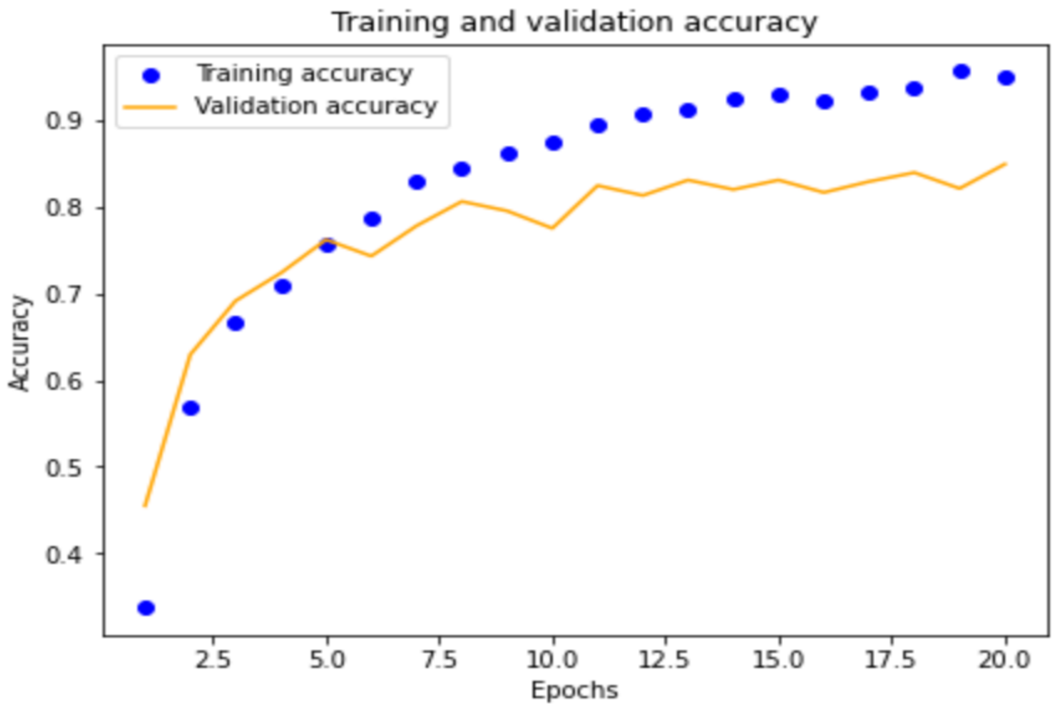
The ***loss function*** computes the quantity that a model should seek to minimize during training. The loss is calculated and the network is updated after each iteration until no further loss reduction (accuracy improvement) is achieved. ***Categorical_crossentropy*** is used when there are more than two label classes, as in my case.

The **evaluation metrics** dictate how we determine the efficacy of the model. In multi-class classification tasks accuracy is a reliable way of measuring performance. ***Accuracy*** is generally defined as: *number of correct predictions / total number of predictions.* In other words, out of all the predictions that were made, how many were correct?

Fitting the model is similar to classic machine learning methods. We call '*.fit*' on our model and enter our independent and dependent variable training arrays as arguments. There is also a '*validation*' parameter where we enter our test data as arguments. The '*epochs*' argument takes a positive number and determines how many times the learning algorithm will work through the entire training data set. I set the *epochs* parameter to 20.

Results:

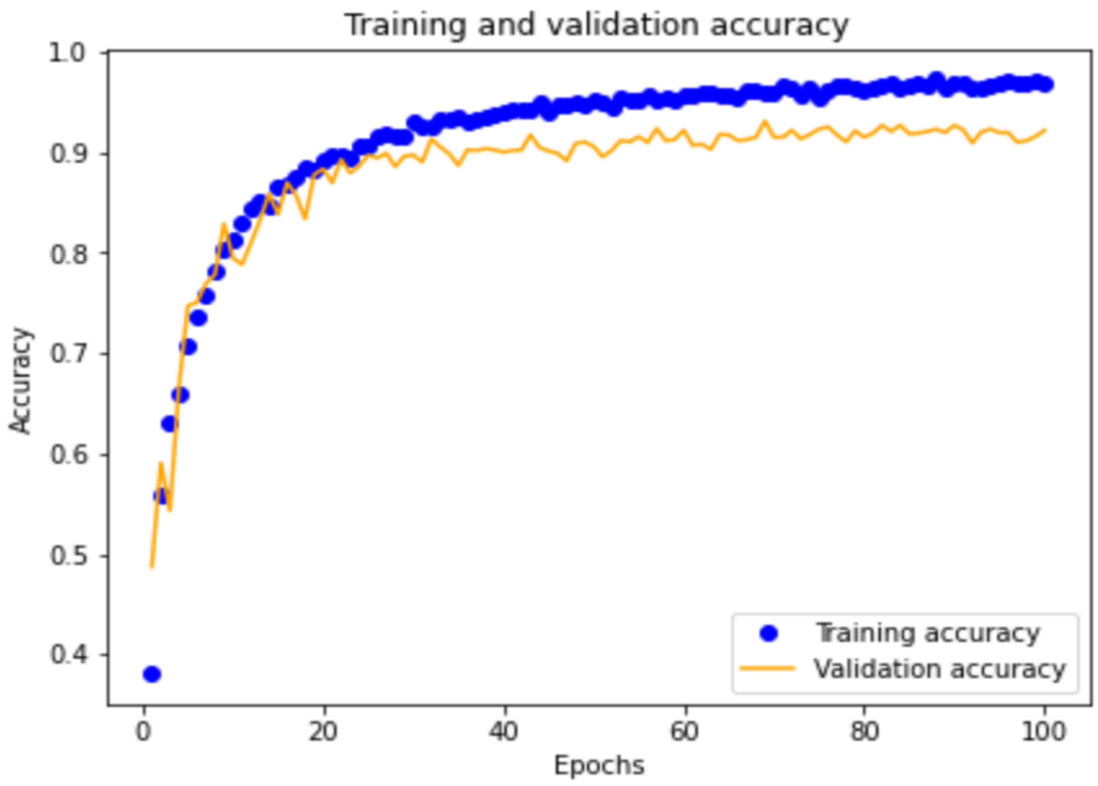
The results of the first model showed a 99% accuracy on the training data and 85.5% accuracy on the testing data. 85% accuracy looks pretty good on its own, however the ~15% disparity between the training and testing score reveals that our model suffers from overfitting. ***Overfitting*** is when a model learns the training data so well that it has a hard time performing on unseen (testing) data; it 'knows' one set of data too well and cannot generalize to new data. The image below illustrates the training and testing/validation accuracy scores. The chart shows that at around 11 epochs the training accuracy continues to improve while the validation accuracy levels off and only improves slightly throughout the remainder.



Model Redesign:

In an attempt to reduce overfitting I redesigned the model. I kept the first three Conv2D layers and added a fourth doubling the filter count (128). However instead of flattening the tensors and adding a dense layer prior to the output layer, I used a ***GlobalAveragePooling2D*** layer. These types of layers are designed to replace fully connected layers (such as Dense layers). Instead of adding the fully connected layer on top of the feature maps, we take the average of each feature map and feed the resulting vector to the output layer. I also increased the number of epochs to 50, 75 and then 100.

The results improved significantly. The training score was again 99%, however the test score was **93.5%**. This suggests that while the model still overfitted a bit it was able to generalize a lot better on unseen data. I noticed the accuracy score kept improving past 50 epochs but leveled off around 75. The chart below displays the accuracy scores for the redesigned network. We can see that the validation score levels off a bit sooner than the training score but they are far more synchronous than in the previous model.



CONCLUSION

This project was a fascinating experience. After months of charting the unfamiliar territory of machine learning, it was nice to feel at home working with audio. The methods for extracting features from sound are rooted in familiar concepts dealing with musical scales, tonal profiles and gain structure. I was able to expand on my previous knowledge and learn about Mel-Frequency Cepstrum Coefficients and how to leverage them to create a traceable image from a sound. Going into the project I didn't expect that audio classification would quickly morph into image classification. Convolutional Neural Networks are clearly powerful and expansive. I learned about CNN model architecture and the purpose of each fundamental component. Deep learning, they say, is a black box, as it's difficult (or impossible) to trace the exact mechanism at play between the *hidden layers*. But it's still a process with a degree of wieldiness. A method of harnessing computational power that I will continue to explore.