Final Project Report: UrbanSounds8K

Course: DATS 6203 – Machine Learning II

Professor: Dr. Amir Jafari

Authored By: Tristin Johnson

Date: December 6th, 2021

Table of Contents

Introduction

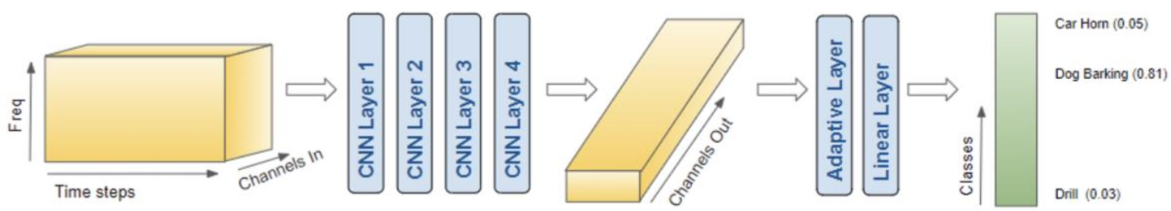
For my final project, I decided to work with a Deep Speech topic. I have always been interested in creating a neural network to train an accurate model using audio files and speech recognition. For this project, I decided to work on a dataset called UrbanSounds8K, which is an audio classification problem. In order to complete this project, there are multiple steps that need to be accomplished: understanding the dataset, what an audio file looks like when initially reading one in, data preprocessing, implementing a neural network, defining the architecture of a neural network, training and validating the model, and finally, testing the performance of the model. With completing all these steps, the goal is to create and implement the best neural network to classify any and all sound files, and that is what I set out to do.

Description of the Dataset

The UrbanSounds9K dataset consists of 8,732 labeled sound excerpts from the 'Urban Sound Taxonomy'. Each audio file is in a .wav format or around 4 seconds long, and classified into 10 different classes (air conditioner, car horn, jackhammer, children playing, dog bark, drilling, engine idling, gun shot, siren, and street music). When downloading the data, the data is pre-sorted into ten folds (labeled fold1, fold2, …, fold10) to support a 10-fold cross-validation, which was very helpful. Furthermore, the data comes with a metadata analysis, 'UrbanSound8K.csv'. This file included information such as the audio file name, the fold number, the class, and more. I was able to use this information to help with mapping the correct audio files with their correct label to the file path in the pre-sorted folds where all the .wav files live.

Deep Learning Network – CNN

The neural network that I used for this project was a Convolutional Neural Network (CNN). CNN's have proven to be very effective and accurate when it comes to image classification. As I am doing audio classification, part of the preprocessing is converting the raw audio files to a Mel Spectrogram, which is quite close to an image itself. More on Mel Spectrograms and the preprocessing will be stated later in this report. Furthermore, a CNN was chosen as it is one of the more powerful neural networks when it comes to handling large amounts of data. The general workflow of the audio files going through a CNN can be seen below:



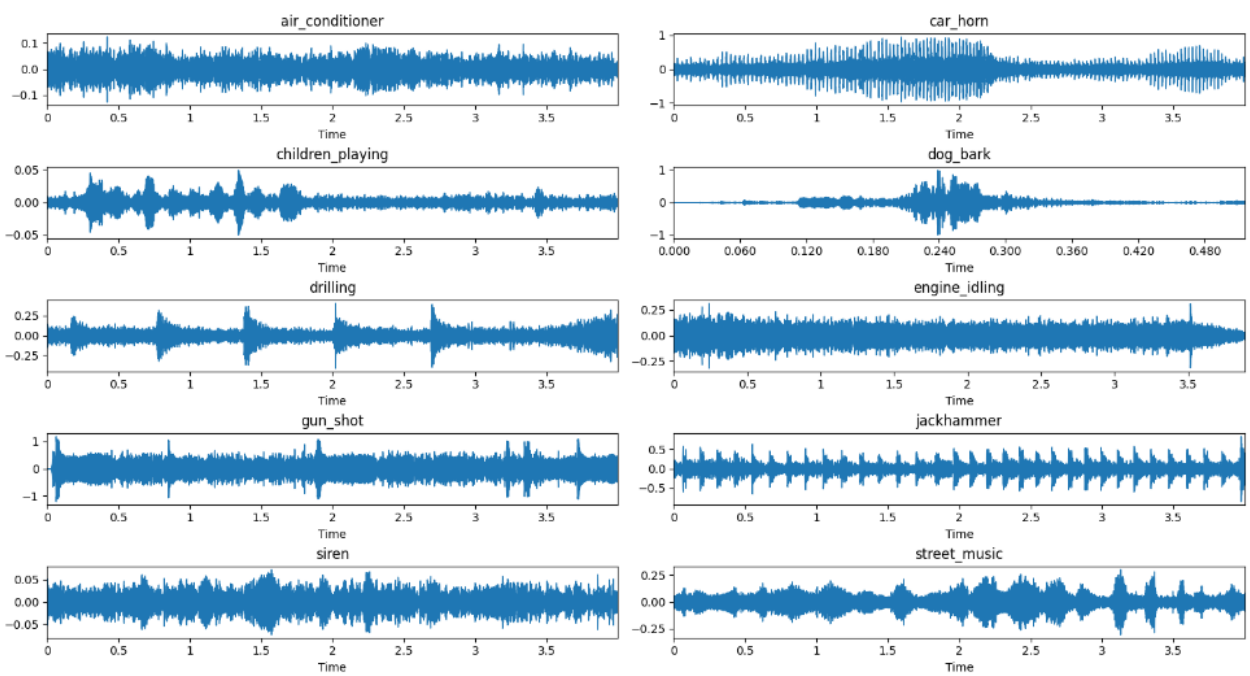
This figure is from Ketan Doshi, Ref. # 1

Looking at the image, the input is going to be a matrix consisting of the number of channels by the time steps (in Mel's) by the frequency (in Decibel's). This will then go through all the layers of the convolution network. We then can add an adaptive average pooling layer, and a linear layer to flatten the output with the correct dimensions and number of classes (batch size by number of classes). The framework I will be using to implement this network will by PyTorch, as I wanted to get familiar in using this framework and get comfortable in working with Tensors. When it comes to defining the layers of a CNN, I used a 4-layer convolution network, that includes a kernel, stride, padding, max and average pooling, activation function, and a full-connected linear layer as output. Below is a screenshot of my CNN that I used for this project.

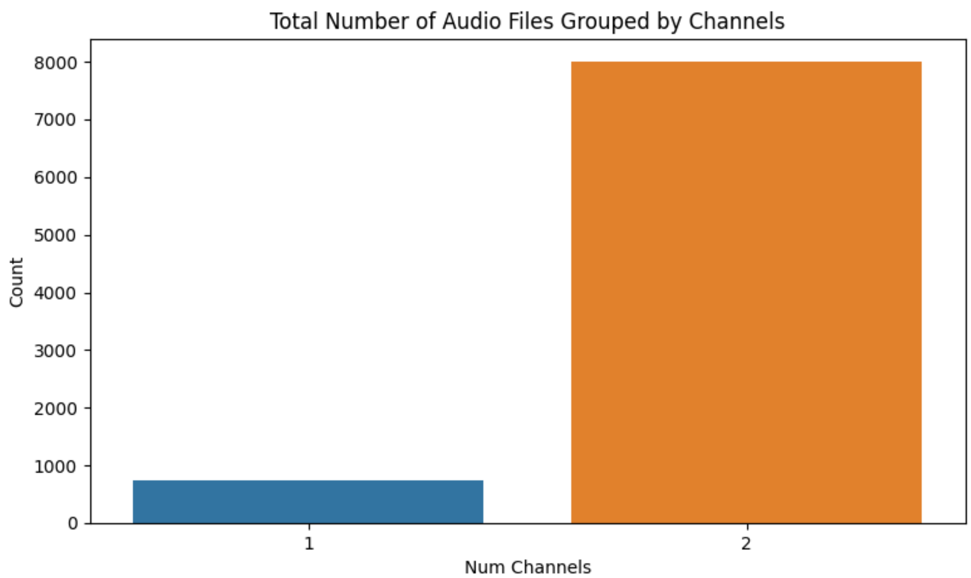
```
AudioClassifier(
  (conv1): Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (batch1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pad1): ZeroPad2d(padding=(2, 2, 2, 2), value=0.0)
  (pool1): MaxPool2d(kernel_size=5, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(8, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (batch2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pad2): ZeroPad2d(padding=(2, 2, 2, 2), value=0.0)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (batch3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (batch4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (act): ReLU()
  (pool2): AdaptiveAvgPool2d(output_size=1)
  (linear1): Linear(in_features=128, out_features=10, bias=True)
)
```

Data Preprocessing & Data Augmentation

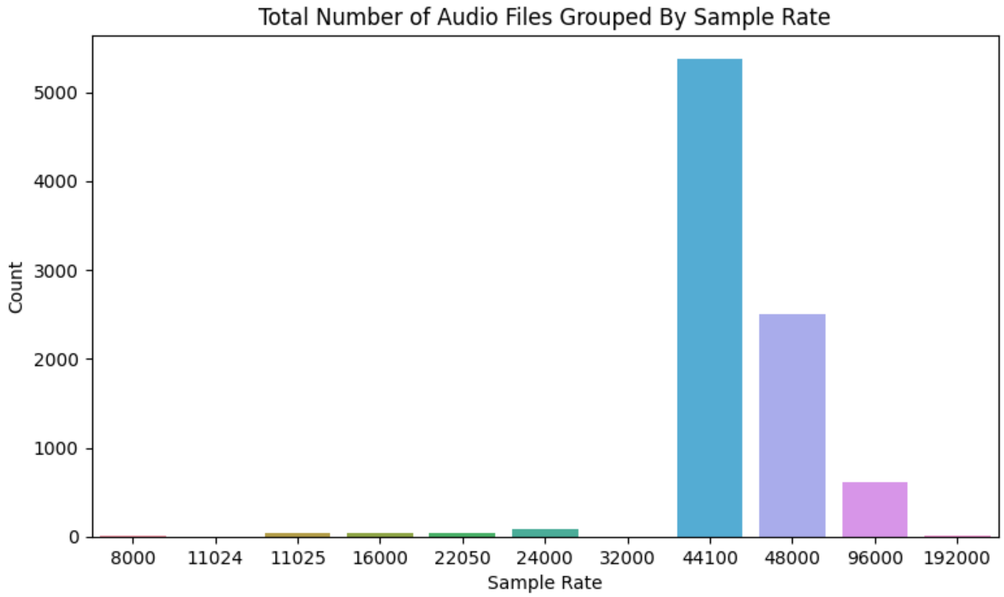
Now that we know what dataset we are going to use along with what neural network we are implementing, we can dive into the data itself. When the audio files are first read in, we can use Librosa to display a wave-plot of the raw data, which can be seen below (TorchAudio was used for the rest of the project to help keep all the data in the form of a Tensor):



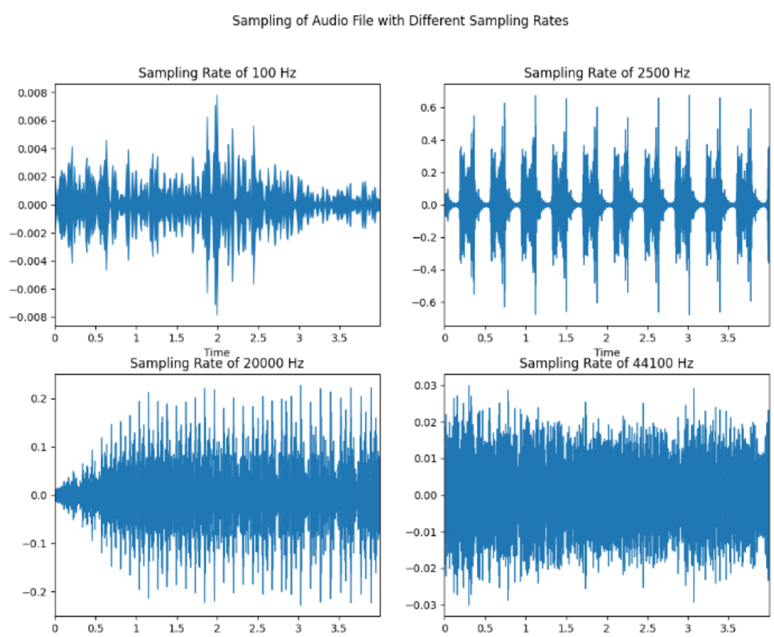
This is what the raw audio files look like for each class, which is a frequency by time plot. The next step is to look at the number of channels in each audio file:



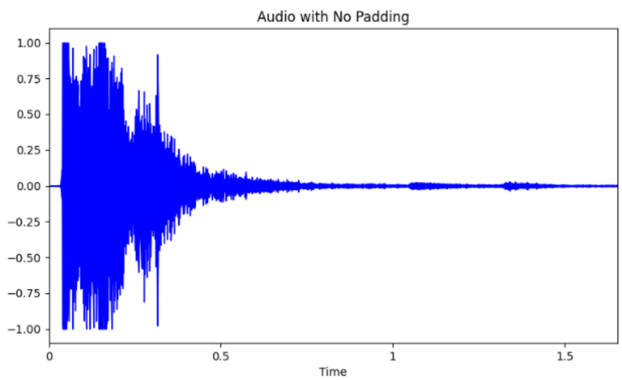
We can see that most of the audio files have two channels, which means they are stereophonic sounds. A stereo audio sends two signals, using two different channels, which comes out to one signal for each speaker. Stereo audios are used to create directionality and perspective to sound. Furthermore, there are some audio files with one channel, which is a monophonic sound. A mono audio means that only a single audio signal is sent to all speakers. Unlike stereo audio, mono will produce the same signal through all its speakers and there won't be any difference between them. Therefore, we must convert all the 1-channel files to a 2-channel files to make each audio input have the same dimensions. This was done simply by gathering all the 1-channel files and duplicating the frequency to 2-channels. Next, we want to standardize the sampling rate of each audio file. Below, you will see how many audio files have what sampling rate:

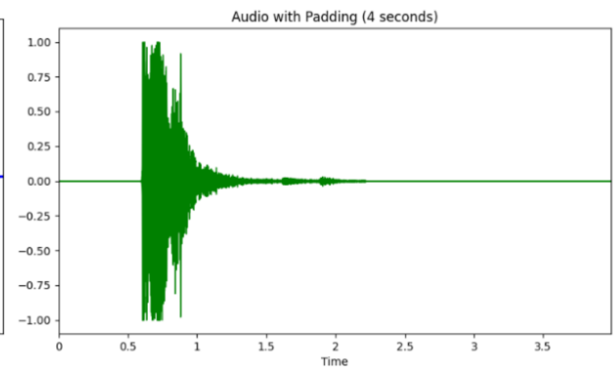


Looking above, we can see that most of the audio files are sampled at 44.1 kHz (44,100 Hz). This means that for every 1 second of audio, the array will have a size of 44,100 values (4 seconds of audio will have a size of 176,400). 44.1 kHz is a standard sampling rate for .wav files, and we will convert all the audio files to a sampling rate of 44.1 kHz. Below is what an audio file looks like with different types of sampling rates of 100 Hz, 2500 Hz, 20 kHz, and 44.1 kHz:

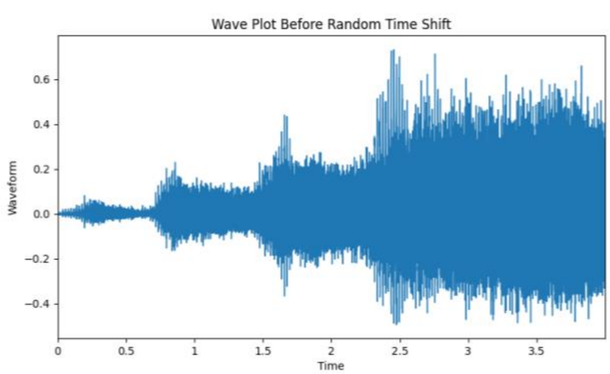


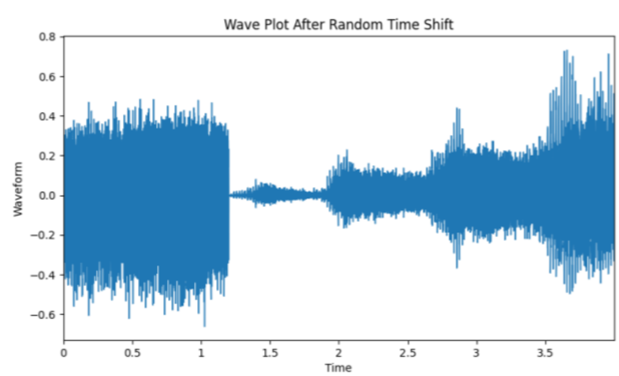
Now that the number of channels and the sampling rate are all the same dimensions, the final step in standardizing the data is to add padding to the audio files to make them the same length of time. Most of these files were of 4 seconds long, but there were a few that were less than 4 seconds. To do this, I randomly padded 0's to the front and/or back of each array. Below is an example of what this looks like:



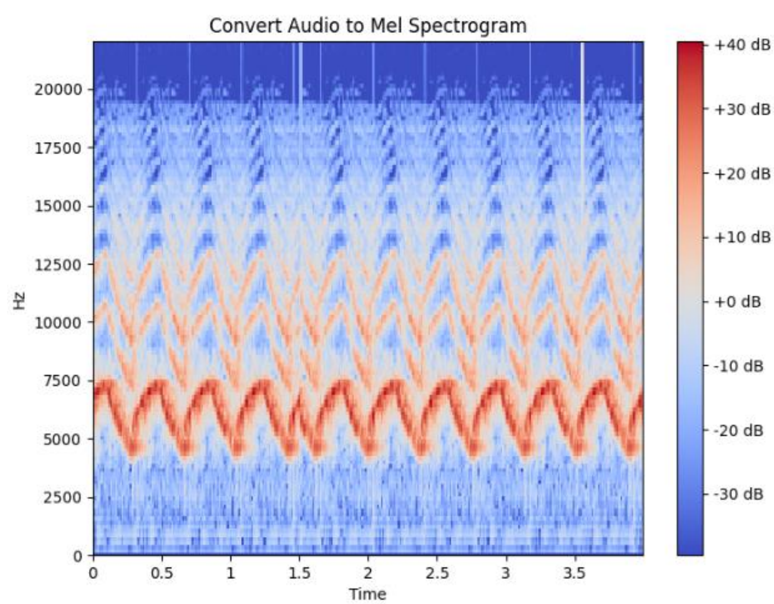


We can see that the original audio file was around 1.75 seconds long. After padding this file, the new length is now 4 seconds. This was the last step in standardizing the audio files. Now, each input has the same number of channels, the same sampling rate, and the same length of time. We can then move on to performing data augmentation on each input to help train the model. The first is to add a random time shift, which can be seen below:



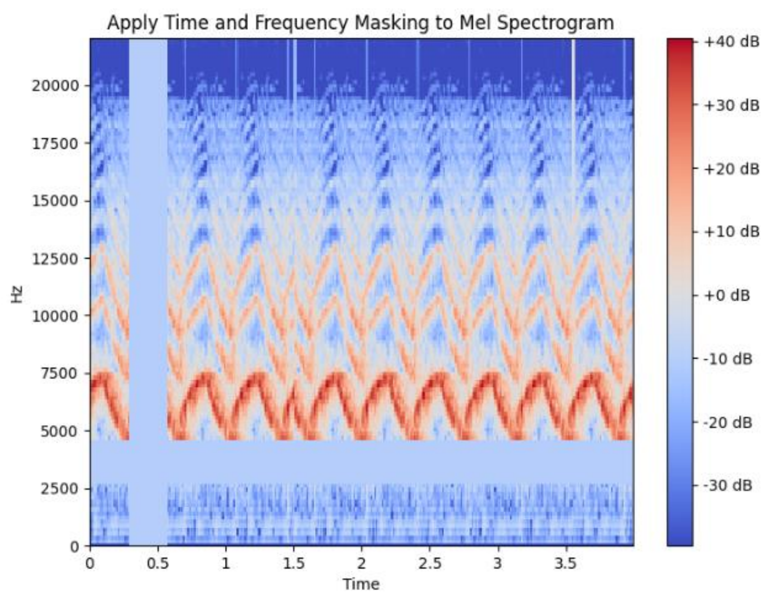


When performing a random time shift, I set the maximum shift percentage to 40%, meaning the limit of time the function will shift is 40%. Looking above, you can see that around 25% of the audio file was shifted from the back to the front. As of now, we have standardized the data to the same dimensions, and performed one iteration of data augmentation. However, the current dimension of this data is number of channels by frequency (2 x 176,400), which is very large and will take a lot of time and computation power. To make these dimensions smaller, the preferred step is to convert the vectors into a Mel Spectrogram. Why Mel Spectrograms are the preferred step as input into a neural network is because they chop up the duration of a sound signal into smaller segments, which significantly reduces the size of each input. An example of a Mel Spectrogram can be seen below:



This spectrogram using a Mel Scale instead of Frequency, to help generalize the frequency into smaller batches (the number of Mel's can be set, and I used the default 64 Mel's). Furthermore, a Mel Spectrogram uses a Decibel scale instead of Amplitude, as this was proven to provide more useful information to a deep learning model. Converting the audio files to a Mel

Spectrogram now gives us an input size of number of channels by number of Mel's by decibels (2 x 64 x 344), which is much better of an input into a CNN. However, there is one last augmentation we can perform on the Mel Spectrogram, which is Time and Frequency Masking. Frequency masking is when we can randomly mask out a range of consecutive frequencies by adding horizontal bars. Time masking is very similar to frequency masking, except we randomly mask out a range of consecutive time by adding vertical bars. Below is an example of what this looks like:



You can see both the vertical and horizontal bars added in the above Mel Spectrogram. The masked sections are replaced with the mean value of each Mel Spectrogram. The purpose of this is to prevent overfitting and help the model generalize more.

<u>Experimental Setup, Training & Validation</u>

Now that we have standardized the data, performed data augmentation to help generalize the model, and converted the data into Mel Spectrograms for input, we can get into training the model. I split the data set into 70% training, 15% validation, and 15% testing. The total number of epochs I used was 20, with a batch size of 16, and a learning rate of 0.001. Furthermore, I used a 'ReduceLROnPlateau' scheduler to adjust the learning rate when training, the AdamW algorithm as the optimizer, and the 'CrossEntropyLoss' as the loss function, since this was a classification problem. All these model parameters were most consistent with performance and accuracy of the model, as I did play around with these parameters a lot. In terms of evaluating the performance of the model, the standard is to use accuracy as there is very little class imbalance, and it is the metric that is used on all models using this dataset. Now that the CNN has been defined, we have standardized the inputs, and have the model parameters set, we can train and validate the model:

	Training	**Validation**
Accuracy	91.95%	90.992%
Avg Time per Epoch	1 minute 58 seconds	24 seconds
Total Run Time	39 minutes 33 seconds	7 minutes 58 seconds

Looking at the table above, I was able to achieve relatively high accuracy scores for both training and validation. The validation accuracy score was used to decide whether to save and update the model weights or not. Regarding the total time to run the model, I did run both training and validation on a NVIDIA Tesla T4 GPU using the Google Cloud Platform, which tremendously helped with training time.

Testing the Model & Results

Now that we have our best model saved, we can use this model and apply the weights to the testing set, and those metrics can be seen below:

	Testing
Accuracy	91.145%
Total Run Time	21 seconds

After testing the model on the test set, I was able to achieve a high accuracy score of 91.145% and only took 21 seconds to compute. Furthermore, I was able to write and save the model predictions to an excel file in which a comparison of the true labels vs. the predictions can be made. On the other hand, I was able to do some further analysis regarding the performance of different data augmentations that I applied to the data. As mentioned earlier in the report, I added 3 different iterations of data augmentation: a random time shift, converting to a Mel Spectrogram, and adding random time and frequency masks. I was able to analyze how each augmentation increased the accuracy of the model. Below is a table highlighting this analysis:

	Accuracy Increase (Low)	**Accuracy Increase (High)**
Random Time Shift	3.16%	4.33%
Mel Spectrogram	5.14%	8.04%
Time & Frequency Masking	3.2%	5.47%

Looking at the table above, you can see that we were able to increase the accuracy by a decent margin after each iteration of augmentation. On the low end, I was able to increase accuracy by around 11.5% and on the high end, increase accuracy by up to 17.84%. This proves that adding these features to the data most definitely increased the performance of the model.

Future Work

With regards to future work, I would like to apply this same dataset to a few different transformers and pre-trained models. There are two audio-specific transformers, by Hugging Face, that I would like to implement in the future: Wav2Vec2 and XLSR-Wav2Vec2. Both of these transformers are considered state-of-the-art frameworks, and I am sure that they would be able to achieve very high levels of accuracy on this dataset and is something I plan on achieving in the future. Furthermore, I do think there is room for improvement on the CNN architecture. I applied several different architectures, with each one gaining slightly more accuracy than the previous, but I do believe there is a sufficient architecture that can be used on this dataset to exceed performance.

Conclusion

Overall, I was very happy with the results of the model. All the training, validation, and testing scores were able to achieve over 90% accuracy, which is quite accurate. Furthermore, I learned a lot in doing this project and working with audio files. I was able to read in a raw audio file and convert that to a wave-plot, standardize the audio files (converting number of channels, changing sampling rate, padding the time), convert the wave-plots to a Mel Spectrogram, and perform various iteration of data augmentation (random time shifts, frequency masking, time masking). I also was able to implement Convolutional Neural Network, which showed promising results. Another key factor was using PyTorch as the framework. In the past, I have mainly used Tensorflow as my neural network framework and using PyTorch for this project taught me a lot about the framework itself, and how to utilize PyTorch's functions (DataSet and DataLoader PyTorch packages) to best fit the network's needs. All in all, I had a lot of fun in doing this project, and I am looking forward to applying my knowledge of Deep Speech to other real-world datasets and other key topics regarding speech.

Implementation of Code

I did this project individually and implemented the project in its entirety by myself. In completing this project, I did use code from other sources to help create the neural network. For a majority of the data preprocessing, I used reference number 1 in the References section, which can be seen in the training and testing scripts. For the metadata analysis, I used reference numbers 4 and 5 to help with the analysis of the dataset which can be seen in the metadata analysis script. All in all, there were 969 lines of code that make up this project, and I used about 360 lines of code from the references stated above. Which means, I wrote 609 lines of code myself, or wrote 62.84% of the code to complete this project.

References

1. Doshi, Ketan. "Audio Deep Learning Made Simple: Sound Classification." *TowardsDataScience,* 18, Mar. 2020, https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

   This reference by Ketan Doshi helped with the preprocessing of audio files. I used a lot of his functions for reference and applied them to my preprocessing pipeline, with a few changes in the code to suite my model.

2. Doshi, Ketan. "Audio Deep Learning Made Simple (Part 2): Why Mel Spectrograms Perform Better." *TowardsDataScience,* 19 Feb. 2020, https://towardsdatascience.com/audio-deep-learning-made-simple-part-2-why-mel-spectrograms-perform-better-aad889a93505

   This reference by Ketan Doshi helped me understand Mel Spectrograms, and why they are best suited as input into a neural network with it comes to Deep Speech Topics. No code from here was used.

3. Gorgolewski, Chris. "UrbanSound8K." *Kaggle*, 4 Feb. 2020, https://www.kaggle.com/chrisfilo/urbansound8k.

   This reference refers to where I downloaded the data from. It was much simpler and faster to download the dataset from Kaggle using their API, compared to downloading the data from the official website. Using this dataset was an exact replica of the dataset from the original home of the data.

4. Kim, Ricky. "Urban Sound Classification." *GitHub,* 14 Aug. 2018, https://github.com/tthustla/urban_sound_classification

   This reference by Ricky Kim helped me understand the Urban Sound dataset and helped me perform certain analysis on the data. I used some of his functions to help analyze the dataset, with a few changes.

5. Lukyamuzi, Shaban. "Urban Sound Dataset." *Jovian,* 5 June 2021, https://jovian.ai/charmzshab/urban-sound-dataset

   This reference by Shaban Lukyamuzi helped me perform more data analysis on the dataset, specifically in working with the Python package Librosa. I used some of the functions and methods in this paper to help with the analysis of the dataset, with a few changes.

6. Mandal, Manav. "Introduction to Convolution Neural Networks." *Analytics Vidhya,* 1 May 2021, https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

   This reference by Manav Mandal was to help me understand the concept of a Convolutional Neural Network and the processes of implementing a CNN. No code was used from this reference.

7. Salamon, Justin. "UrbanSound8K." *PapersWithCode.* https://paperswithcode.com/dataset/urbansound8k-1

   This reference by Justin Salamon refers to multiple different reports and papers on past people who have used this dataset to create a neural network. I used these as a reference to a baseline accuracy score, along with the potential accuracy scores. No code was used from this reference.

8. "UrbanSound8K." *Urban Sound Datasets*, https://urbansounddataset.weebly.com/urbansound8k.html.

   This reference refers to the original download of the data, along with all the information about the data itself. You can download the data from here, but I used Kaggle as it was more efficient to download the data into the cloud.

Code Appendix

Here, is all the code that I used from the stated references above, with a short description of what each function does, and the reference number I got the code from. All the code from reference #1 is in both the training and testing script, and all the code from references #4 and #5 are in the metadata analysis script.

1. Convert all audio files with 1 channel to 2 channels (reference #1):

```
# convert all audio files with 1 audio channel to 2 channels (majority
have 2 channels)
def convert_channels(audio, num_channel):
    waveform, sampling_rate = audio

    if waveform.shape[0] == num_channel:
        return audio

    if num_channel == 1:
        new_waveform = waveform[:1, :]
    else:
        new_waveform = torch.cat([waveform, waveform])

    return new_waveform, sampling_rate
```

2. Standardize the sampling rate of all audio files to 44.1 kHz (reference #1):

```
# standardize the sampling rate of each audio file
def standardize_audio(audio, new_sample_rate):
    new_waveform, sampling_rate = audio

    if sampling_rate == new_sample_rate:
        return audio

    # get number of channels
    num_channel = new_waveform.shape[0]

    # standardize (resample) the first channel
    waveform_1 = torchaudio.transforms.Resample(sampling_rate,
new_sample_rate)(new_waveform[:1, :])

    # if number of channels > 1, resample second channel
    if num_channel > 1:
        waveform_2 = torchaudio.transforms.Resample(sampling_rate,
new_sample_rate)(new_waveform[1:, :])
```

```
        # merge both channels
        new_waveform = torch.cat([waveform_1, waveform_2])

    return new_waveform, new_sample_rate
```

3. Add padding to audio files to make them all 4 seconds long (reference #1):

```
# pad the waveform of all audio files to a fixed length in ms
(milliseconds)
def pad_audio_files(audio, max_ms):
    waveform, sampling_rate = audio
    rows, wave_len = waveform.shape
    max_len = sampling_rate//1000 * max_ms

    # pad the waveform to the max length
    if wave_len > max_len:
        waveform = waveform[:, :max_len]

    # add padding to beginning and end of the waveform
    elif wave_len < max_len:
        padding_front_len = random.randint(0, max_len - wave_len)
        padding_end_len = max_len - wave_len - padding_front_len

        # pad the waveforms with 0
        padding_front = torch.zeros(rows, padding_front_len)
        padding_end = torch.zeros(rows, padding_end_len)

        # concat all padded Tensors
        waveform = torch.cat((padding_front, waveform, padding_end), 1)

    return waveform, sampling_rate
```

4. Apply a random time shift to the audio (reference #1):

```
# apply a random time shift to shift the audio left or right by a random
amount
def random_time_shift(audio, shift_limit):
    waveform, sample_rate = audio
    _, wave_len = waveform.shape
    shift_amount = int(random.random() * shift_limit * wave_len)

    return waveform.roll(shift_amount), sample_rate
```

5. Convert audio files to a Mel Spectrogram (reference #1):

```
# get a Mel Spectrogram from audio files
def mel_spectrogram(audio, num_mel=64, num_fft=1024, hop_len=None):
    waveform, sampling_rate = audio
    top_decibel = 80  # min negative cut-off in decibels (default is 80)
```

```
    # fit audio to a mel spectrogram
    spectrogram = torchaudio.transforms.MelSpectrogram(sampling_rate,
                                                        n_fft=num_fft,
                                                        hop_length=hop_len,

n_mels=num_mel)(waveform)

    # convert spectrogram to decibels
    spectrogram =
torchaudio.transforms.AmplitudeToDB(top_db=top_decibel)(spectrogram)

    return spectrogram
```

6. Apply time and frequency masking to the Mel Spectrogram (reference #1):

```
# data augmentation on audio files
# 1. frequency mask --> randomly mask out a range of consecutive
frequencies (horizontal bars)
# 2. time mask --> randomly block out ranges of time from spectrogram
(vertical bars)
def data_augmentation(spectrogram, max_mask_pct=0.1, num_freq_masks=1,
num_time_masks=1):
    # get channels, number of mels, and number of steps from spectrogram
    channels, num_mels, num_steps = spectrogram.shape

    # get the mask value from spectrogram (the mean)
    mask_value = spectrogram.mean()

    # spectrogram augmentation
    augmented_spectrogram = spectrogram

    # apply number of frequency masks to audio file
    freq_mask_params = max_mask_pct * num_mels
    for _ in range(num_freq_masks):
        augmented_spectrogram =
torchaudio.transforms.FrequencyMasking(freq_mask_params)(augmented_spectro
gram,

mask_value)

    # apply number of time masks to audio file
    time_mask_params = max_mask_pct * num_steps
    for _ in range(num_time_masks):
        augmented_spectrogram =
torchaudio.transforms.TimeMasking(time_mask_params)(augmented_spectrogram,
mask_value)

    return augmented_spectrogram
```

7. Define custom DataSet for the data (reference #1):

```
# define custom variables for UrbanSounds DataSet
class UrbanSoundsDS(Dataset):
```

```
    def __init__(self, data, data_path):
        self.data = data
        self.data_path = data_path
        self.duration = audio_duration
        self.sampling_rate = sample_rate
        self.channel = num_channels
        self.shift_pct = 0.4

    # total number of items in dataset
    def __len__(self):
        return len(self.data)

    # get the i'th item in dataset
    def __getitem__(self, index):
        # get path of audio file
        audio_file = self.data_path + self.data.loc[index, 'file_path']

        # get class id from audio file
        class_id = self.data.loc[index, 'classID']

        # load the audio file
        audio = load_file(audio_file)

        # standardize all audio files
        resample_audio = standardize_audio(audio, self.sampling_rate)

        # make all audio files have same number of channels
        rechannel = convert_channels(resample_audio, self.channel)

        # add padding
        pad_audio = pad_audio_files(rechannel, self.duration)

        # randomize time shift
        shift_audio = random_time_shift(pad_audio, self.shift_pct)

        # get mel spectrogram
        spectrogram = mel_spectrogram(shift_audio)

        # augment the spectrogram
        augment_spectrogram = data_augmentation(spectrogram,
num_freq_masks=2, num_time_masks=2)

        return augment_spectrogram, class_id
```

8. Get the number of channels and sample rate for each audio file (reference #4):

```
# get the number of channels and sample rate for each audio file
def get_more_info(file_name):
    path, _ = get_path(file_name)
    wave_file = open(path, 'rb')
    format = wave_file.read(36)

    num_channels = format[22:24]
    num_channels = struct.unpack('H', num_channels)[0]
```

```
    sample_rate = format[24:28]
    sample_rate = struct.unpack('I', sample_rate)[0]

    return num_channels, sample_rate
```

9. Randomly plot one row of each class as a wave-plot (reference #4):

```
# plot each row as a waveplot using librosa
fig, axs = plt.subplots(5, 2, figsize=(15, 8), constrained_layout=True)
axs = np.reshape(axs, -1)

for (idx, row), ax in zip(unique_audio.iterrows(), axs):
    ax.set_title(row.values[-1])
    data, sr = librosa.load(f'Data/urbansound8k/fold{row.values[-5]}/' +
row.values[0])
    _ = librosa.display.waveplot(data, ax=ax)

plt.show()
```

10. Plot audio file with different sampling rates (reference #5):

```
# plot an audio file with different sample rates
fig, axs = plt.subplots(2, 2, figsize=(12, 9))
fig.suptitle('Sampling of Audio File with Different Sampling Rates')
axs = np.reshape(axs, -1)
diff_sr = [100, 2500, 20000, 44100]
for ax, sr in zip(axs, diff_sr):
    data, sample_rate = librosa.load(audio, sr=sr)
    librosa.display.waveplot(data, sr=sample_rate, ax=ax)
    ax.set_title(f'Sampling Rate of {sr} Hz')

plt.show()
```

11. Plot how many audio files have a certain sample rate (reference #5):

```
# plot how many audio files have a certain sample rate
sample_rate_totals = metadata['sampling_rate'].value_counts()

plt.figure(figsize=(9, 5))
sns.barplot(x=sample_rate_totals.index, y=sample_rate_totals.values)
plt.title('Total Number of Audio Files Grouped By Sample Rate')
plt.xlabel('Sample Rate')
plt.ylabel('Count')
plt.show()
```

12. Plot how many audio files have a certain number of channels (reference #5):

```
# plot how many audio files have a certain number of channels
num_channels_total = metadata['num_channels'].value_counts()

plt.figure(figsize=(9, 5))
sns.barplot(x=num_channels_total.index, y=num_channels_total.values)
plt.title('Total Number of Audio Files Grouped by Channels')
plt.xlabel('Num Channels')
plt.ylabel('Count')
plt.show()
```