CSUDH
DEPARTMENT OF COMPUTER SCIENCE

Spring 2024

Senior Design Project Report

Computer Science Department

California State University, Dominguez Hills

Ryan Junge
Student ID: 204824698
CSC-492
Dr. Ali Ashkan Jalooli

Table of Contents

Chapter 1:

Introduction

This project will address automotive diagnostics utilizing audio recognition technology. I will develop a method to diagnose automobiles using audio recordings of engine sounds. By employing deep learning technologies, this software could make strides in pursuit of helping individuals to accurately diagnose and address vehicle malfunctions.

The motivation behind this project is to advance the field of automotive repair through the application of artificial neural networks towards a sound classification framework. Using this theoretical framework, anyone who owns a car would be able to perform diagnostic maintenance in the comfort of their own home. The application of this technology could revolutionize the standard model of automotive preventative maintenance.

While this project will not implement an interface for the end-user, it should be viewed as a step in the right direction. The resources required to implement a project of such magnitude are unavailable to me at present. In the future, state of the art technologies will have a more clear path forward, but for the time being, it is often difficult to portray the abilities of artificial intelligence due to lack of education and experience among the general public.

Problem Statement

A lack of transparency, consumer education, and skilled labor have all contributed to a less than ideal automotive repair market in the United States in recent years. Consumers are spending more, waiting longer, and receiving worse service from their auto mechanics (Torque360, 2024). Many local auto repair shops have gone out of business during COVID, and the labor market shortage has increased salaries for technicians, driving prices up. These factors all combine to produce a potential market bubble within the automotive maintenance industry.

Relevance

The current conditions of the market present an opportunity to obtain a customer base for an end-user based software platform integrating artificial neural networks with an interface designed for the everyday automobile owner. A seamless system could allow users to pass their own engine sounds through the pre-trained neural network by recording from a smartphone and uploading the audio to a server/cloud based platform. This paper will not seek to establish this end-user experience, however I seek to move the goalposts forward so that this technology can improve the lives of billions of car owners.

Related Works

Audio classification is generally a broad topic in regards to data science and machine learning. This project draws upon the works of previous publications and best-practices within the field of machine learning, (Ekman, 2022) signal processing, (Doshi, 2021) and data preprocessing. (Roberts, 2024)

Large-Scale Acoustic Automobile Fault Detection: Diagnosing Engines Through Sound

Dennis Fedorishin
dcfedori@buffalo.edu
University at Buffalo
Buffalo, New York, USA

Livio Forte III
ACV Auctions, Inc.
Buffalo, New York, USA

Justas Birgiolas
ACV Auctions, Inc.
Ronin Institute
Buffalo, New York, USA

Deen Dayal Mohan
University at Buffalo
Buffalo, New York, USA

Philip Schneider
ACV Auctions, Inc.
Buffalo, New York, USA

Srirangaraj Setlur
University at Buffalo
Buffalo, New York, USA

Venu Govindaraju
University at Buffalo
Buffalo, New York, USA

This publication's end-goal is very similar to my own. The system described in this work is designed with an auction vendor in mind. In addition, their system is built upon a set of included cars, so if the car for auction is not in the list, the software will be useless. My own work aims to direct the product design in the direction towards mass availability, rather than proprietary licensing to vendors.

Limitation of Existing Works

Several limitations exist that hold back the development of projects such as these. First, there is a surprisingly limited availability of collected recordings portraying automobile engine sound. This creates issues when it comes to gathering data for training and testing purposes. Within my own search for audio data, I experienced several highly priced paywalls locking away vast amounts of valuable data.

Perhaps noticing this vacuum of data, a scholarly group of researchers from Spain and Ecuador managed to create a database for automotive engine fault classification. (Vergara, 2023) EngineFaultDB uses a range of parameters to simulate various engine states and also emulates the internal combustion of the C14NE spark ignition engine. With this ability to analyze across a wide variety of situations, the possibility of an end-user based product is even more viable.

The limitations, however, are still significant. With only one single engine spec used for this sampling, a neural network built upon this training data would be severely limited in the scope of everyday life.

Instead, I seek to generalize my model further than the previously mentioned works. To do this, I have scraped the USC Optical Sound Effects Library (Smith, 2023) to gather a collection of audio samples with which to train my network. This library consists of tens of thousands of first generation copies of original nitrate optical sound effects created for Hollywood studios. As such, many of the recordings are of high quality, and in close proximity to the sound source with little to no background noise.

Chapter 2:

Proposed Method

I will be training an artificial neural network to identify automobiles with various mechanical issues. Using deep learning techniques, I will train the AI to recognize patterns within the sounds of combustion engine vehicles. Through developing a framework for diagnostics, the model will be able to discern and acutely analyze the audio signals' variances and identify engine malfunctions.

Before I can begin analyzing audio signals, I must first normalize the data and create spectrograms and labels for our training data (Seo, 2019).

There exist several options, but Mel Spectrograms (like the one shown in fig. 1) seem like a good starting place as they are "often the most suitable way to input audio data into deep learning models." (Doshi, 2021)

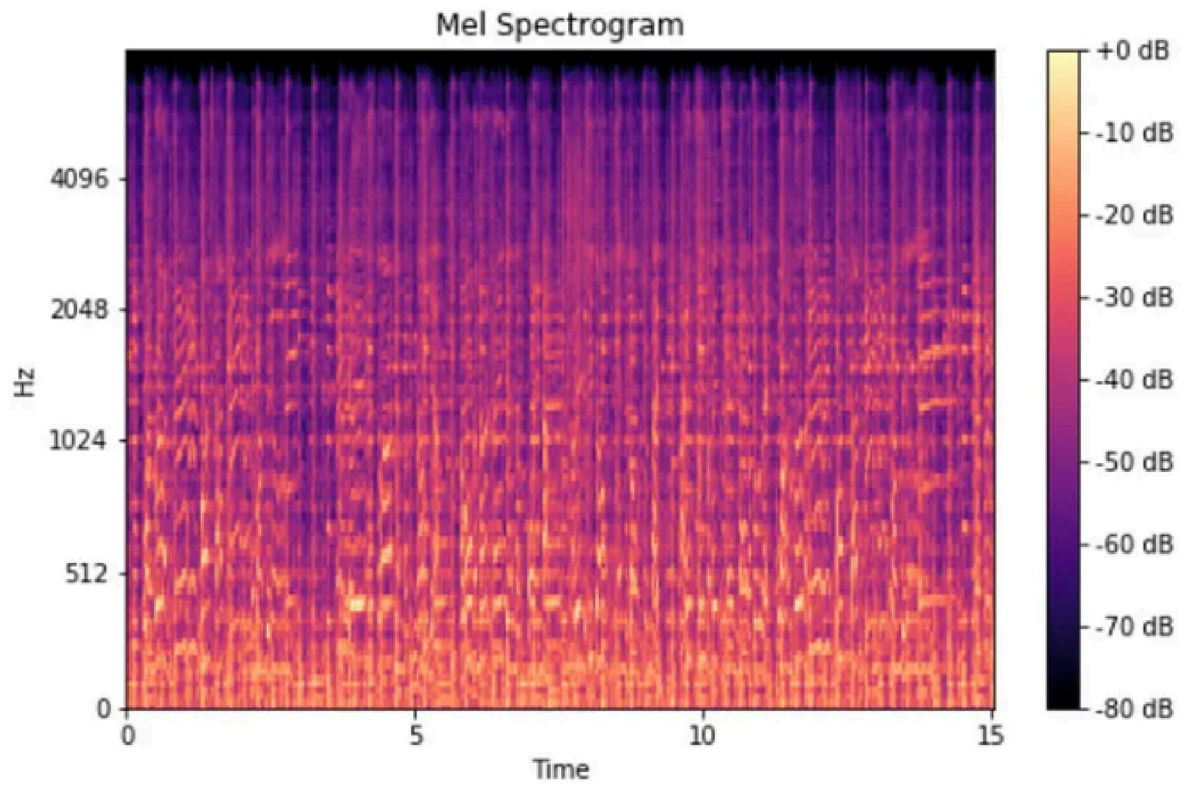


Fig. 1 - (Roberts, 2024)

Since the metadata for my sound files will be saved as a .CSV (comma separated value) I can use Pandas to read it.

After importing the metadata,

I will be using PyTorch to do the following:

- Load the audio data using data from Pandas
- Preprocess audio data - convert or normalize audio
- Build the model - implementing input, conv, relu, activation, hidden, pool, fc, and output layers

- Train the model - feeding data and labels through the model
- Evaluate accuracy - Analyze the accuracy after a set number of epochs
- Modify the model and repeat until desired accuracy is achieved.

Gathering and organizing training data will be a crucial aspect of this project. I will need to gather a large enough sampling of audio recordings to provide a sufficient dataset with which to train the model. Special consideration must be given to ensure that variances in recording equipment or capture software are not able to create anomolous conditions from which the learning model gets confused or misled. For example, if a portion of the healthy dataset is recorded with faulty equipment, it could result in unexpected situations where unhealthy cars are labeled as healthy erroneously.

Labeling data will also be an arduous task, as I will need to classify audio recordings into their respective classes. Making sure that each error is recorded under the correct number of varying conditions such as air temperature and pressure, vehicle make/model, and recording hardware will require trial and error down the road.

Comparison, Explanation of Superiority, and Justification

The preferred method of vehicular diagnostics will always be to use licensed equipment and technicians. This project, however, aims to help those without direct access to affordable automotive maintenance.

Automotive diagnostics usually require highly specialized equipment. OBD2 Code readers are expensive, and manufacturer specific software keeps individuals from accessing all the data within their engine's computer. These elements effectively gate-keep the consumer from self-diagnosing issues within their own engine.

Without access to data from the engine's computer, we are limited in the methodology we can consider to solve this problem. Traditional algorithms will be unsuitable for our use because of the lack of specificity in our data. And since we lack standardization in our data, we cannot communicate in any meaningful way, which exact sounds we are listening for. This lack of access to quality hardware restricts us to audio file classification.

With audio as our only real source of data that we can use for diagnosis, using traditional algorithms would be problematic due to a lack of standardized data.AI seems to be a good fit for this task. If we properly train the neural network to learn the various sounds of ailing vehicles, along with the differences from healthy engines, perhaps they can be used to help diagnose any vehicle that needs maintenance.

Neural networks have proven quite useful when it comes to learning based on loosely defined data, rather than typical software which usually requires rigid data types and comparators in order to classify data. Specifically, with the use of computer vision, neural networks can pick up on acoustic details and intricacies that would fall deaf upon human ears. These factors show AI to be our best and only available option.

Project Design Elements

Use Cases:

Personal Preventative Maintenance - the average person who doesn't know much about cars could one day be able to use a later version of this software to decide whether or not to seek out a mechanic. Normally, people have to rely on their check-engine light, which is generally not very reliable. This alarm light also doesn't give any diagnostic information. Using AI based software, the user could one day definitively know whether their car either does, or doesn't need maintenance.

Under-Served Communities - Many communities and countries throughout the world face a shortage of auto mechanics. Using this software, a person without access to an auto mechanic could diagnose and repair their own vehicle. Once a diagnosis is generated, the user can access thousands of how-to videos online on how to replace various mechanical parts.

Real-Time "On the Fly" Maintenance - A user who is driving their car hears a funny sound. Their engine seems to be making noises that the owner does not recognize. Normally, the driver doesn't have many options other than keep driving. This can be dangerous and expensive if continued operation results in more damage to the vehicle. Instead, the driver can simply pull over and record the sound of their engine. In seconds, AI could inform the owner if the sound is cause for alarm. In fact, if it is a system-critical malfunction, it could alert the driver that they should immediately stop their engine and call a tow truck. In contrast, if it is simply a squeaky belt or a non-serious issue, it would give the driver peace of mind to continue on with their day.

Chapter 3:

Implementation of the Proposed Solution

External Dependencies

- Python 3.12.3 (a High Level Dynamic Programming Language)
- Pandas (a Python Data Analysis Library)
- PyTorch (a Python Machine Learning Library)
- Torchaudio 2.3.0 (a PyTorch Audio/Signal Processing Library)

Data

My data is sourced from the USC Optical Sound Effects Library (Smith, 2023). I've provided a visual representation of the data in fig. 2 to give credit to the author of this database. In order to implement this data into a format that our network can understand, I manually typed out the metadata for these sound files. I took special care to only sample recordings that are clear and absent of excessive variance and noise.

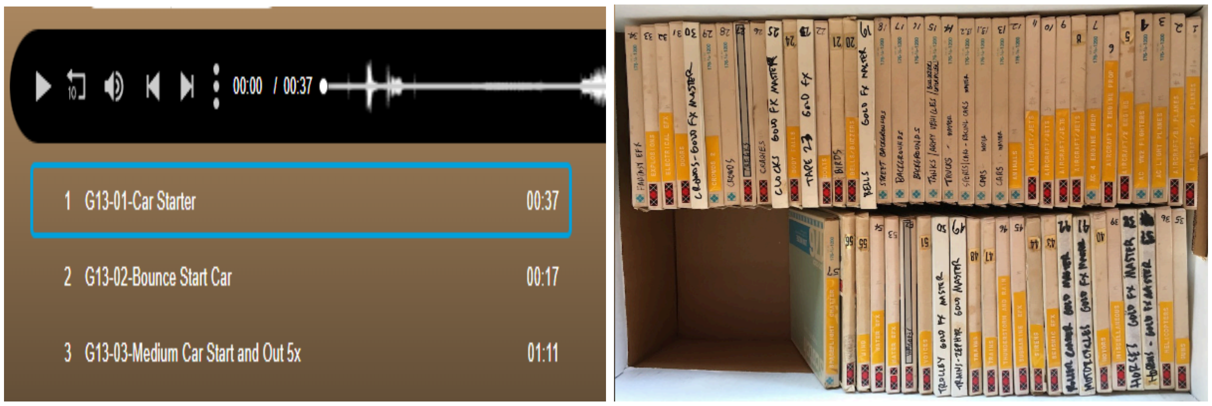
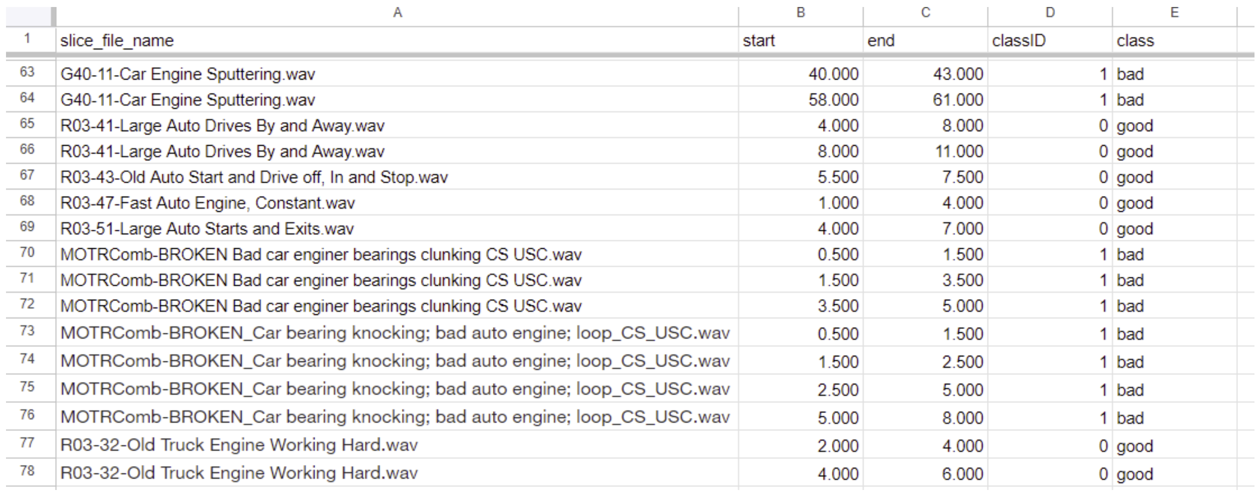

00:00 / 00:37
1 G13-01-Car Starter 00:37
2 G13-02-Bounce Start Car 00:17
3 G13-03-Medium Car Start and Out 5x 01:11

Fig. 2 - Left: built-in audio player and amplifier Right: Image of the original audio tapes converted to digital (Smith, 2023)

In addition, as evident in fig. 3, I have labeled the data into its respective classes resembling good or bad engine sounds. Since we have 2 classes of data, we can implement a binary framework with our artificial neural network.



	A	B	C	D	E
1	slice_file_name	start	end	classID	class
63	G40-11-Car Engine Sputtering.wav	40.000	43.000	1	bad
64	G40-11-Car Engine Sputtering.wav	58.000	61.000	1	bad
65	R03-41-Large Auto Drives By and Away.wav	4.000	8.000	0	good
66	R03-41-Large Auto Drives By and Away.wav	8.000	11.000	0	good
67	R03-43-Old Auto Start and Drive off, In and Stop.wav	5.500	7.500	0	good
68	R03-47-Fast Auto Engine, Constant.wav	1.000	4.000	0	good
69	R03-51-Large Auto Starts and Exits.wav	4.000	7.000	0	good
70	MOTRComb-BROKEN Bad car enginer bearings clunking CS USC.wav	0.500	1.500	1	bad
71	MOTRComb-BROKEN Bad car enginer bearings clunking CS USC.wav	1.500	3.500	1	bad
72	MOTRComb-BROKEN Bad car enginer bearings clunking CS USC.wav	3.500	5.000	1	bad
73	MOTRComb-BROKEN_Car bearing knocking; bad auto engine; loop_CS_USC.wav	0.500	1.500	1	bad
74	MOTRComb-BROKEN_Car bearing knocking; bad auto engine; loop_CS_USC.wav	1.500	2.500	1	bad
75	MOTRComb-BROKEN_Car bearing knocking; bad auto engine; loop_CS_USC.wav	2.500	5.000	1	bad
76	MOTRComb-BROKEN_Car bearing knocking; bad auto engine; loop_CS_USC.wav	5.000	8.000	1	bad
77	R03-32-Old Truck Engine Working Hard.wav	2.000	4.000	0	good
78	R03-32-Old Truck Engine Working Hard.wav	4.000	6.000	0	good

Fig. 3 - screenshot from MetaData.csv (Image by Ryan Junge)

“For training and testing purposes of our model, we should have our data broken down into three distinct dataset splits.” (Baheti. 2021)

<u>The Training Set</u>

The set of data used to train the model to learn hidden features/patterns within the data.

<u>The Validation Set</u>

The set of data, separate from the training set, that is used to validate our model’s performance during training. The main idea is to prevent the model from overfitting.

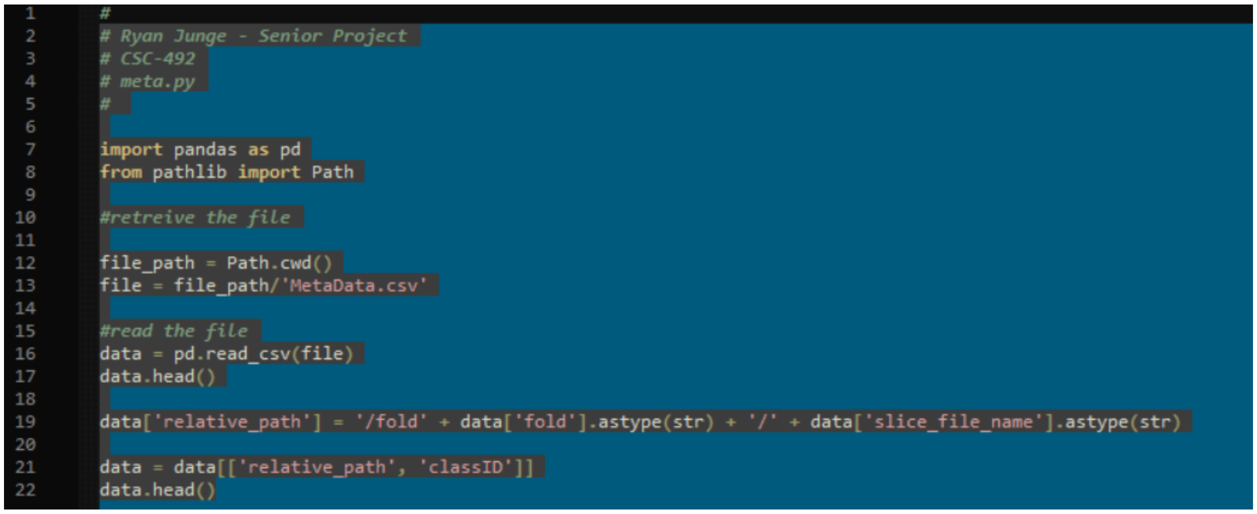
<u>The Test Set</u>

A separate set of data used to test the model after completing training. According to Baheti, it provides an unbiased final model of performance metrics in terms of accuracy and precision.

In addition to the training files, I have also composed a testing set from the same USC library.

Preprocessing

Before we can begin preprocessing our data, we need to read in our metadata so that we can define our data types and begin to organize the structure of our network. To do this, I have a simple python program which uses pandas to label our data. (see Fig. 4)



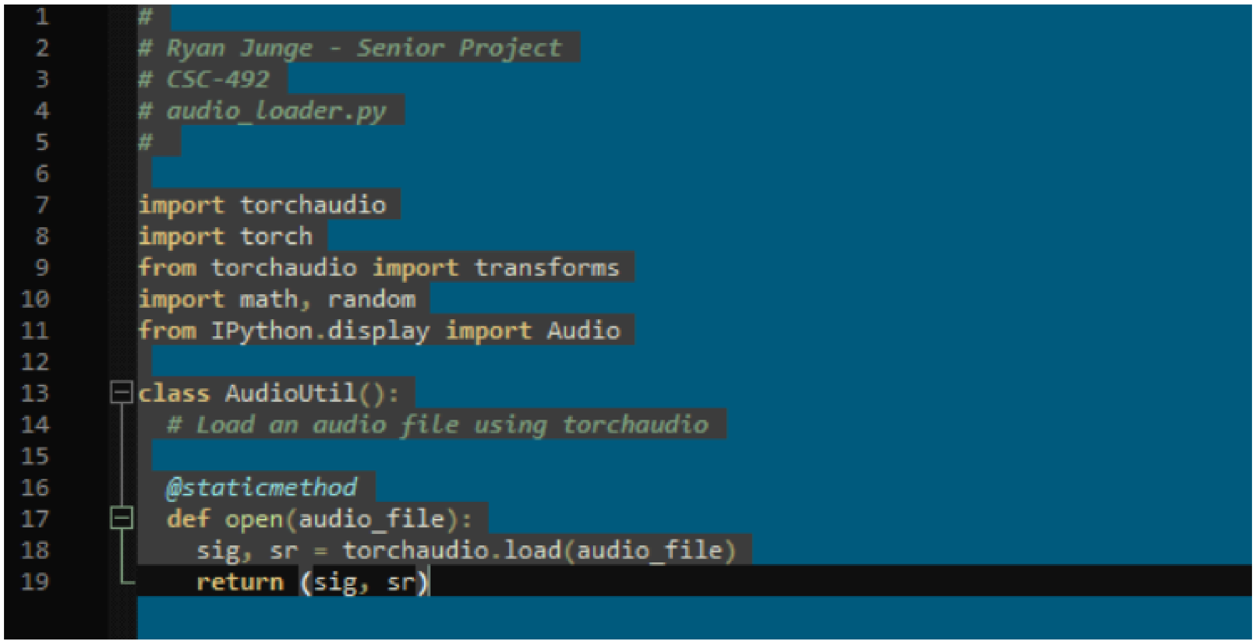
```python
#
# Ryan Junge - Senior Project
# CSC-492
# meta.py
#

import pandas as pd
from pathlib import Path

#retreive the file

file_path = Path.cwd()
file = file_path/'MetaData.csv'

#read the file
data = pd.read_csv(file)
data.head()

data['relative_path'] = '/fold' + data['fold'].astype(str) + '/' + data['slice_file_name'].astype(str)

data = data[['relative_path', 'classID']]
data.head()
```

Fig. 4 - meta.py - Appendix A

Now I have the metadata loaded, which includes all file paths of the data so that I can load them into memory sequentially. Using this method saves a lot of memory by allocating a single space in memory where data is loaded to be passed to the network. Instead of loading all data into memory, the network will iterate through our pandas dataframe to pass one loaded file at a time.

First, I need to establish a way to read and load our audio files which are all in .wav format.

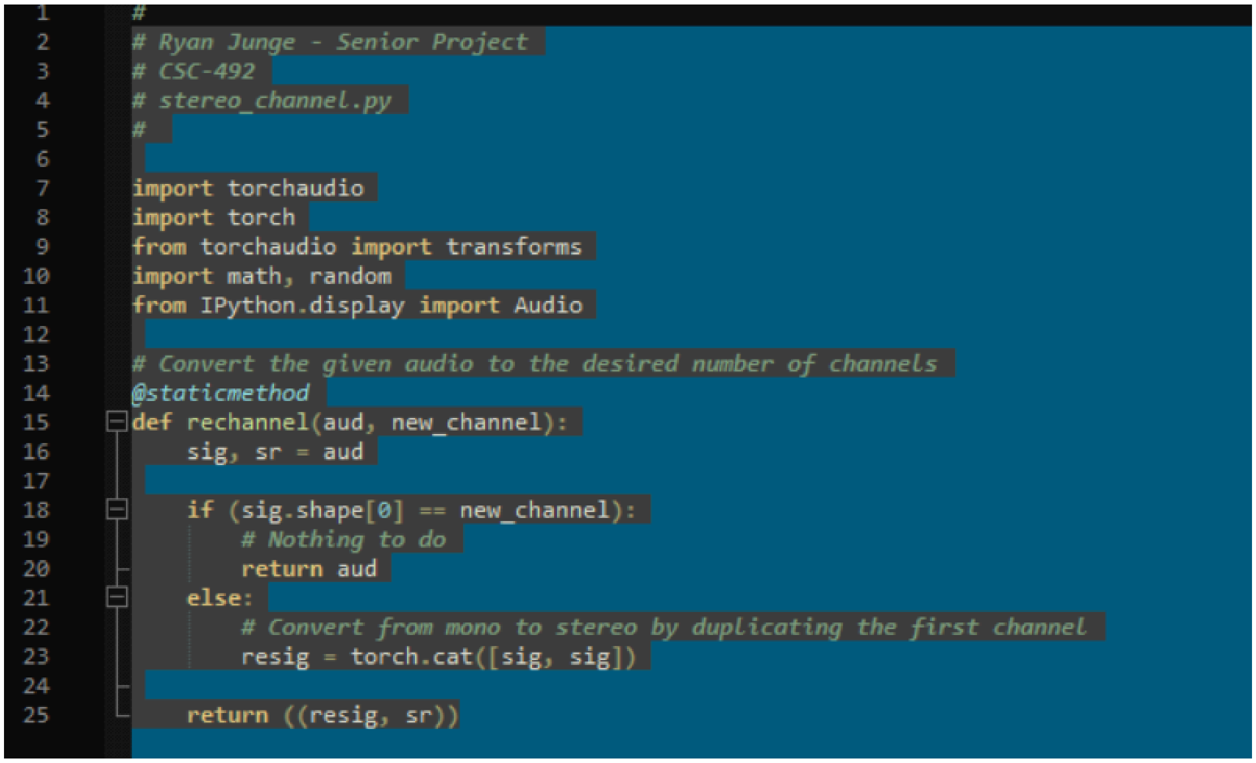
Torchaudio has a class called load() to load an audio file and return the signal and sampling rate. This method is shown being used in Fig. 5



```
#
# Ryan Junge - Senior Project
# CSC-492
# audio_loader.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

class AudioUtil():
  # Load an audio file using torchaudio

  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)
```

Fig. 5 - audio_loader.py - Appendix B

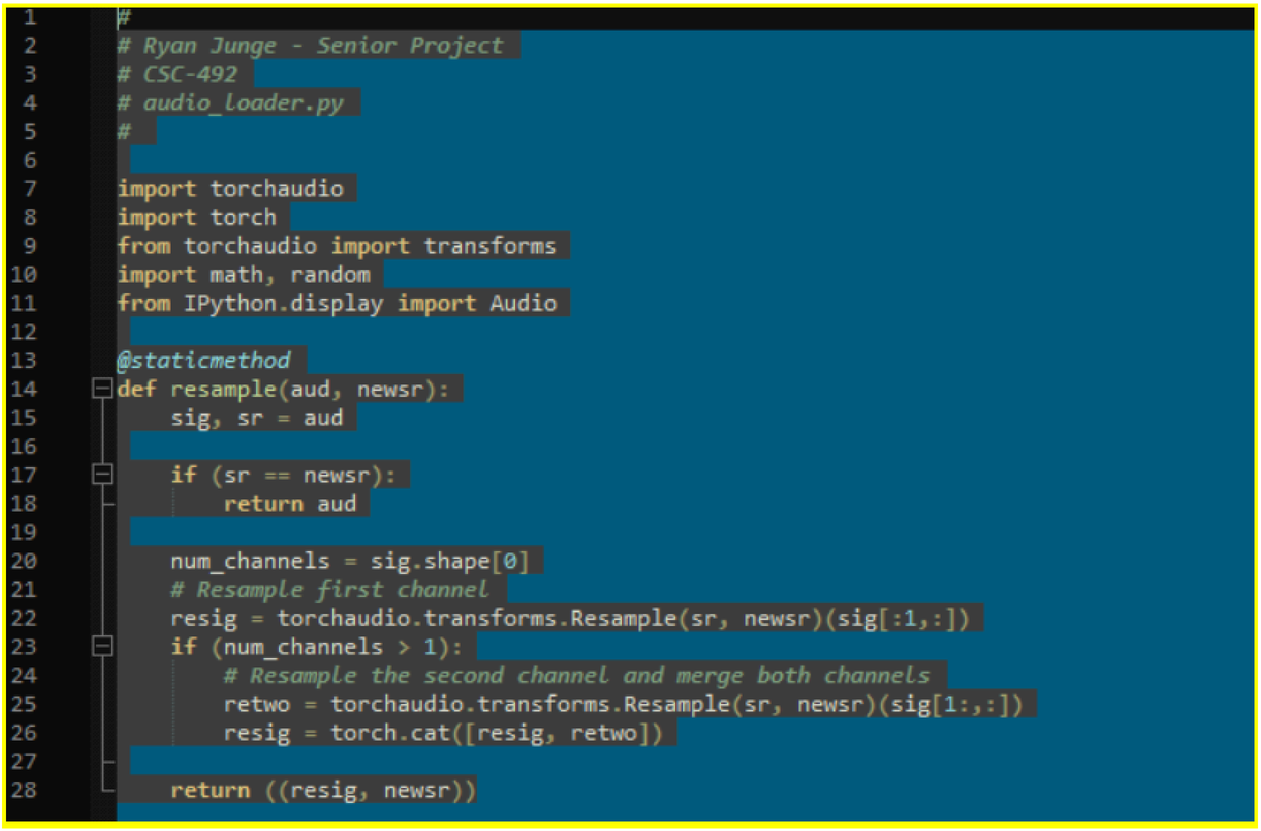
Next, I verify that all the sound files are stereo, with 2 sound channels. If any sound files are mono, I will transform them into stereo by copying the solo channel to the second. (See Fig. 6)



```python
#
# Ryan Junge - Senior Project
# CSC-492
# stereo_channel.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

# Convert the given audio to the desired number of channels
@staticmethod
def rechannel(aud, new_channel):
    sig, sr = aud

    if (sig.shape[0] == new_channel):
        # Nothing to do
        return aud
    else:
        # Convert from mono to stereo by duplicating the first channel
        resig = torch.cat([sig, sig])

    return ((resig, sr))
```

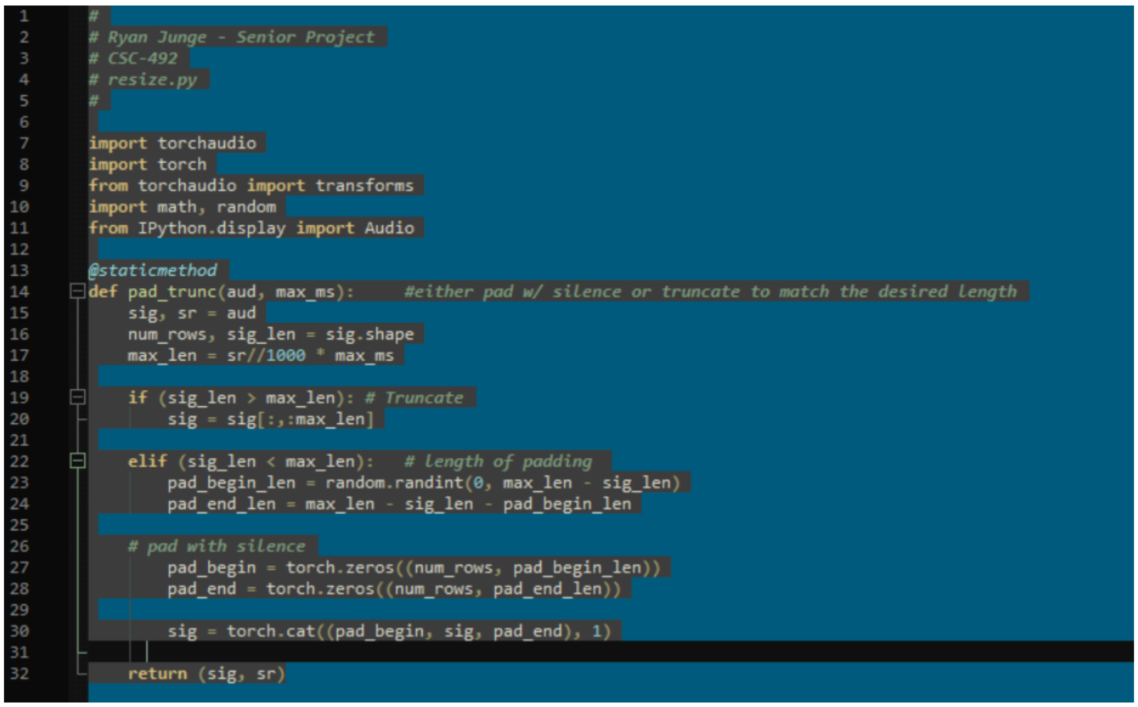
Fig. 6 - stereo_channel.py - Appendix C

I need to standardize the sampling rate, and resize the sound files so that they are all the same length. This can be accomplished by padding it with silence, or truncating depending if the file's sampling rate is higher or lower than the desired rate. (see Fig. 7 & 8)



```python
#
# Ryan Junge - Senior Project
# CSC-492
# audio_loader.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def resample(aud, newsr):
    sig, sr = aud

    if (sr == newsr):
        return aud

    num_channels = sig.shape[0]
    # Resample first channel
    resig = torchaudio.transforms.Resample(sr, newsr)(sig[:1,:])
    if (num_channels > 1):
        # Resample the second channel and merge both channels
        retwo = torchaudio.transforms.Resample(sr, newsr)(sig[1:,:])
        resig = torch.cat([resig, retwo])

    return ((resig, newsr))
```

Fig. 7 - sampling_rate.py - Appendix D

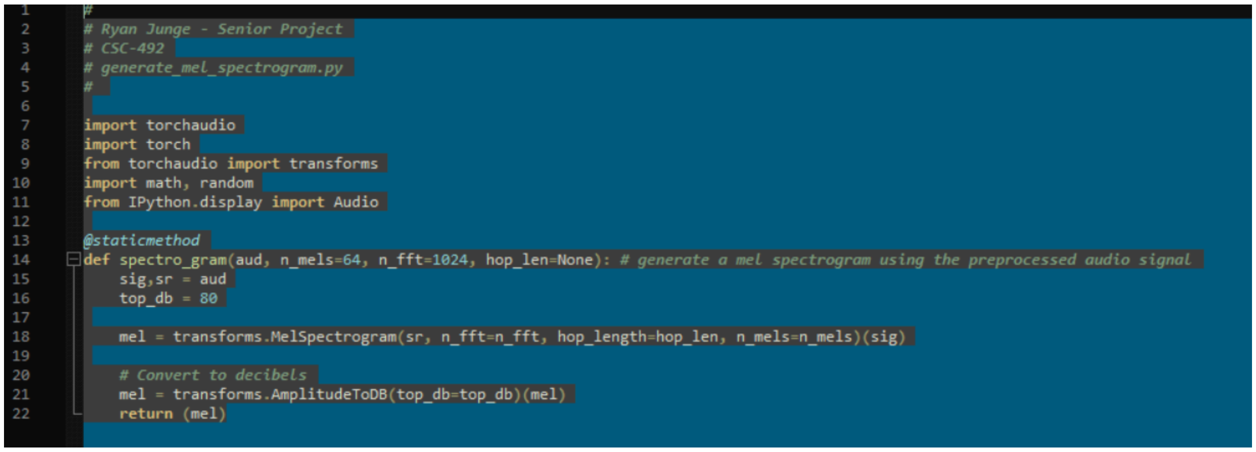


```python
#
# Ryan Junge - Senior Project
# CSC-492
# resize.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def pad_trunc(aud, max_ms):     #either pad w/ silence or truncate to match the desired length
    sig, sr = aud
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms

    if (sig_len > max_len): # Truncate
        sig = sig[:,:max_len]

    elif (sig_len < max_len):    # length of padding
        pad_begin_len = random.randint(0, max_len - sig_len)
        pad_end_len = max_len - sig_len - pad_begin_len

    # pad with silence
        pad_begin = torch.zeros((num_rows, pad_begin_len))
        pad_end = torch.zeros((num_rows, pad_end_len))

        sig = torch.cat((pad_begin, sig, pad_end), 1)

    return (sig, sr)
```

Fig. 8 - resize.py - Appendix E

Now, I am ready to convert the preprocessed audio into a Mel Spectrogram.

Note in figure 9, I make sure to convert the amplitude to decibels.



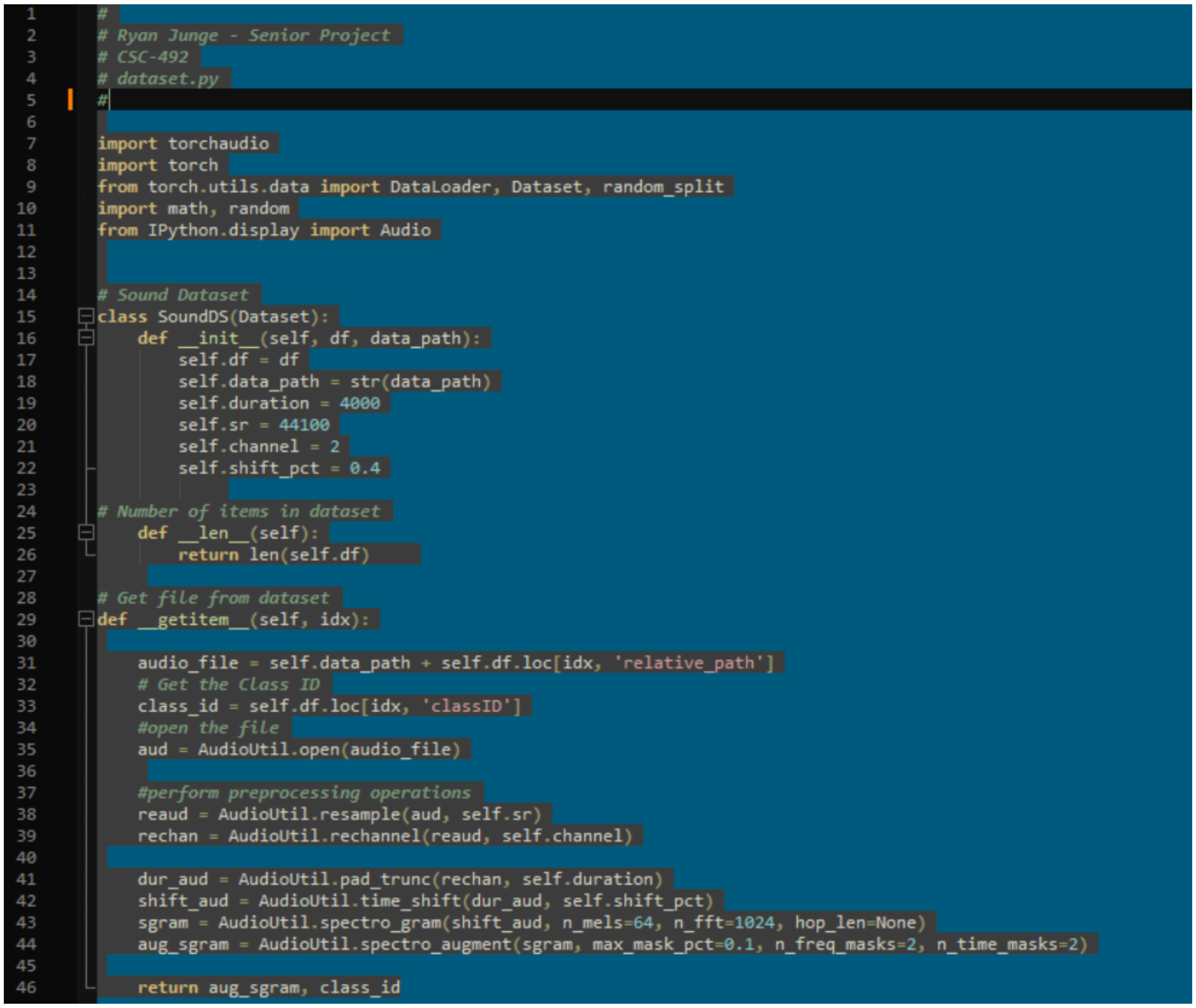
```python
#
# Ryan Junge - Senior Project
# CSC-492
# generate_mel_spectrogram.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None): # generate a mel spectrogram using the preprocessed audio signal
    sig,sr = aud
    top_db = 80

    mel = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

    # Convert to decibels
    mel = transforms.AmplitudeToDB(top_db=top_db)(mel)
    return (mel)
```

Fig. 9 - generate_mel_spectrogram.py - Appendix F

Loading Data

Since I have finished all preprocessing required for the data, I can now define a custom dataset object using PyTorch.

In addition, I will use the built-in DataLoader object class to import my custom dataset.
(See fig. 10)



```
#
# Ryan Junge - Senior Project
# CSC-492
# dataset.py
#

import torchaudio
import torch
from torch.utils.data import DataLoader, Dataset, random_split
import math, random
from IPython.display import Audio


# Sound Dataset
class SoundDS(Dataset):
    def __init__(self, df, data_path):
        self.df = df
        self.data_path = str(data_path)
        self.duration = 4000
        self.sr = 44100
        self.channel = 2
        self.shift_pct = 0.4

# Number of items in dataset
    def __len__(self):
        return len(self.df)

# Get file from dataset
def __getitem__(self, idx):

    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']
    #open the file
    aud = AudioUtil.open(audio_file)

    #perform preprocessing operations
    reaud = AudioUtil.resample(aud, self.sr)
    rechan = AudioUtil.rechannel(reaud, self.channel)

    dur_aud = AudioUtil.pad_trunc(rechan, self.duration)
    shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_aud, n_mels=64, n_fft=1024, hop_len=None)
    aug_sgram = AudioUtil.spectro_augment(sgram, max_mask_pct=0.1, n_freq_masks=2, n_time_masks=2)

    return aug_sgram, class_id
```

Fig. 10 - dataset.py - Appendix G

I use my custom dataset to load the metadata and labels from my Pandas Dataframe shown earlier. Using this method, I can split the dataset randomly at a 9:1 ratio between training and validation sets. This will allow 90% of data to be utilized for training, and 10% to be utilized for validation.

Using the data loader, each object will have two tensors, one labeled X containing data and Mel Spectrograms, and the other labeled Y which contains the target labels, which contain the classID representing a healthy vs unhealthy engine.

Finally, the data is ready to be input into the network.

Creating the Model

Now that I have performed the required preprocessing steps, building the network will be very similar to traditional image classification problems. The network architecture is laid out in Fig. 11

Network Architecture

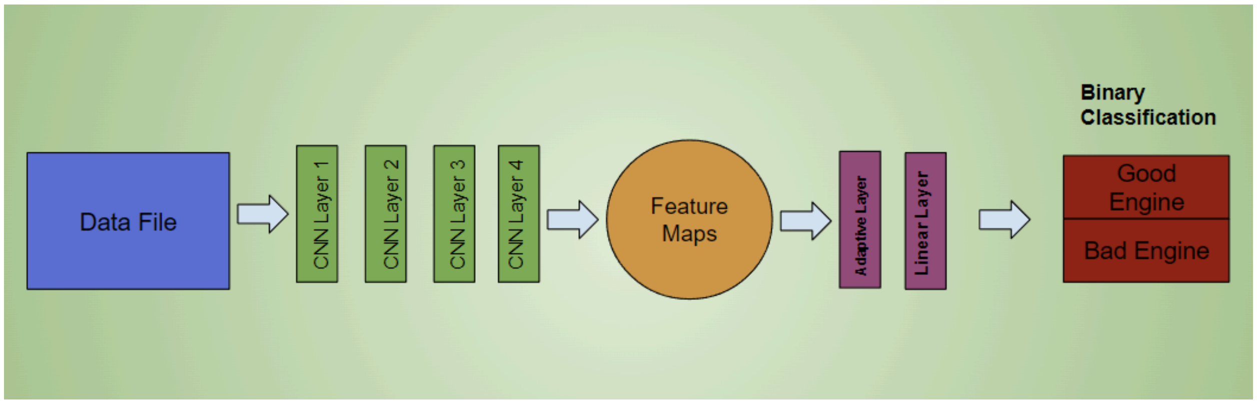
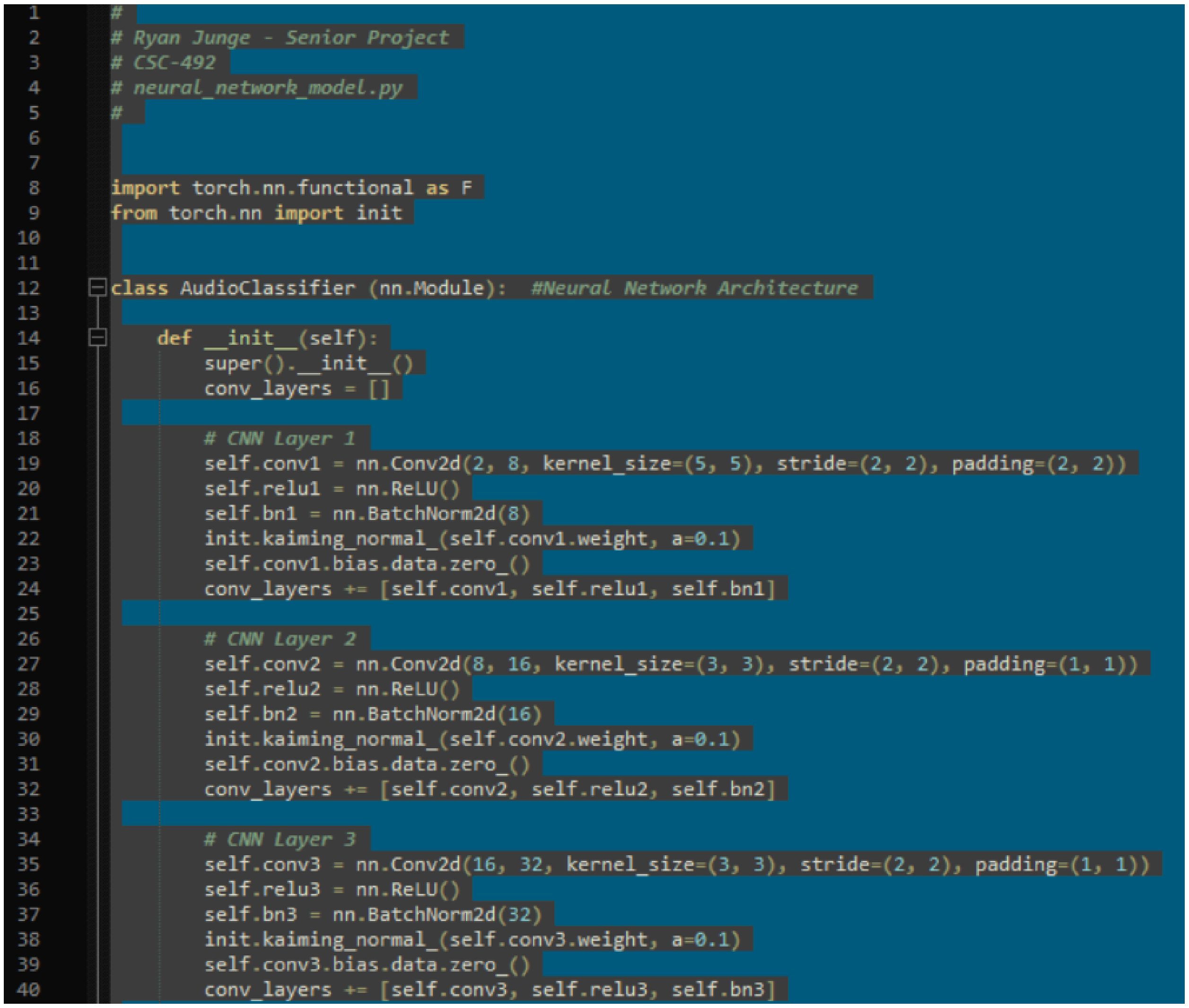


Fig. 11 - Network Architecture (Image by Ryan Junge)

To summarize, a batch of data images will be input to the network. Each CNN layer steps up the image depth. After passing through the layers, the output is a feature map. The adaptive layer will then flatten the feature maps to a shape of (16,64) and pass on to the linear layer. Finally, the linear layer will output a binary value, corresponding to a good or bad result.

This code for this network is sampled in Fig. 12.



```python
#
# Ryan Junge - Senior Project
# CSC-492
# neural_network_model.py
#


import torch.nn.functional as F
from torch.nn import init


class AudioClassifier (nn.Module):  #Neural Network Architecture

    def __init__(self):
        super().__init__()
        conv_layers = []

        # CNN Layer 1
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # CNN Layer 2
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]

        # CNN Layer 3
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]
```

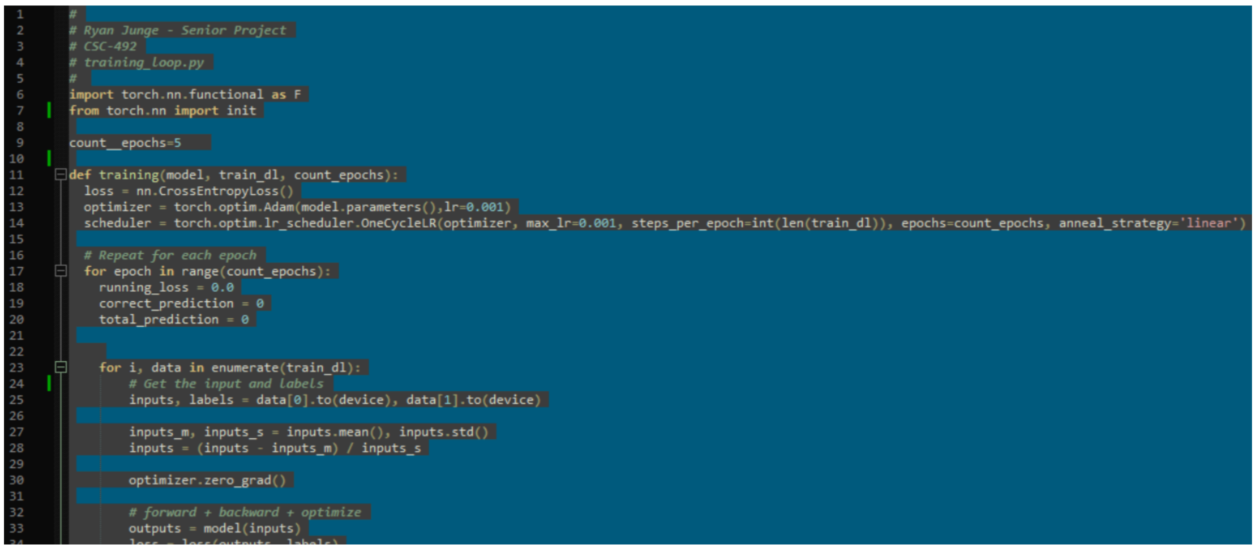
Fig. 12 - neural_network_model.py - Appendix H

Evaluation, Results, and Discussions

Training

Now that I have implemented the network structure and defined all necessary functions, I can begin training the model. To do this, I execute the training loop over several epochs, making sure to keep track of the accuracy metric to measure the ratio of correct and incorrect guesses.

A snippet of this implementation is shown in Fig. 13.



```python
#
# Ryan Junge - Senior Project
# CSC-492
# training_loop.py
#
import torch.nn.functional as F
from torch.nn import init

count__epochs=5

def training(model, train_dl, count_epochs):
  loss = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001, steps_per_epoch=int(len(train_dl)), epochs=count_epochs, anneal_strategy='linear')

  # Repeat for each epoch
  for epoch in range(count_epochs):
    running_loss = 0.0
    correct_prediction = 0
    total_prediction = 0


    for i, data in enumerate(train_dl):
        # Get the input and labels
        inputs, labels = data[0].to(device), data[1].to(device)

        inputs_m, inputs_s = inputs.mean(), inputs.std()
        inputs = (inputs - inputs_m) / inputs_s

        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
```

Fig. 13 - training_loop.py - Appendix I

Testing

I've attached the results to the most optimized version of my model. After spending considerable time adjusting every parameter I could try, our results led to a decent conclusion.

As you can see in the pyplot below(Fig. 14), the loss function is not perfect, there is no smooth curve to show consistent learning. Instead, there are small hills and valleys which show that each iteration/epoch may or may not improve the current model. Over time, more adjustments to network structure may be required. In addition, there may be more optimized activation and loss functions which may be developed.

In addition to fig. 14, I have included Fig. 15 as a snippet of the final data received from the program. This is simply a csv showing the error slowly decreasing over 50 epochs. Overall, the main functionality is implemented and although refactoring will be necessary, I have reached a good place to close this chapter of my research. In the future, I will seek to spend time improving my model and expanding its scope to reach my final vision of where this product belongs. Ultimately, that place is installed on as many smartphones as possible.

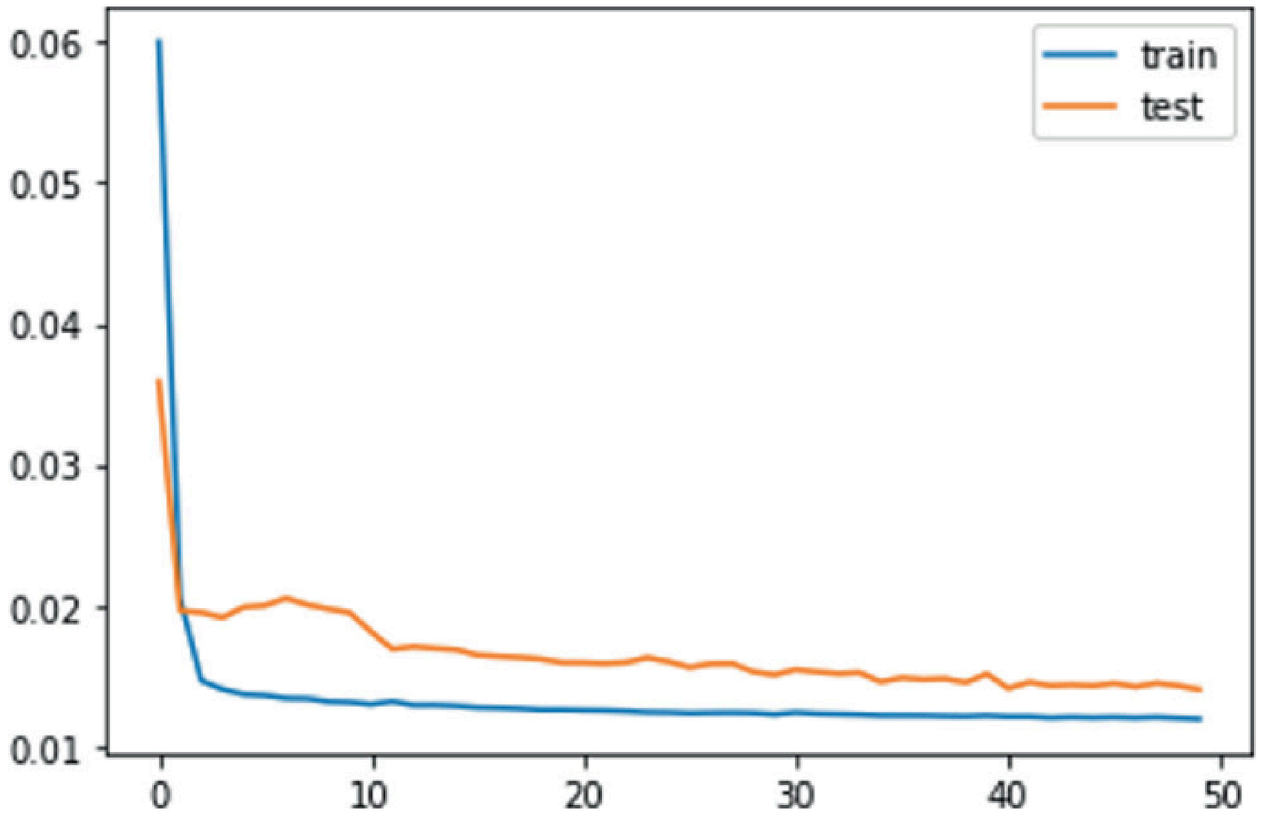
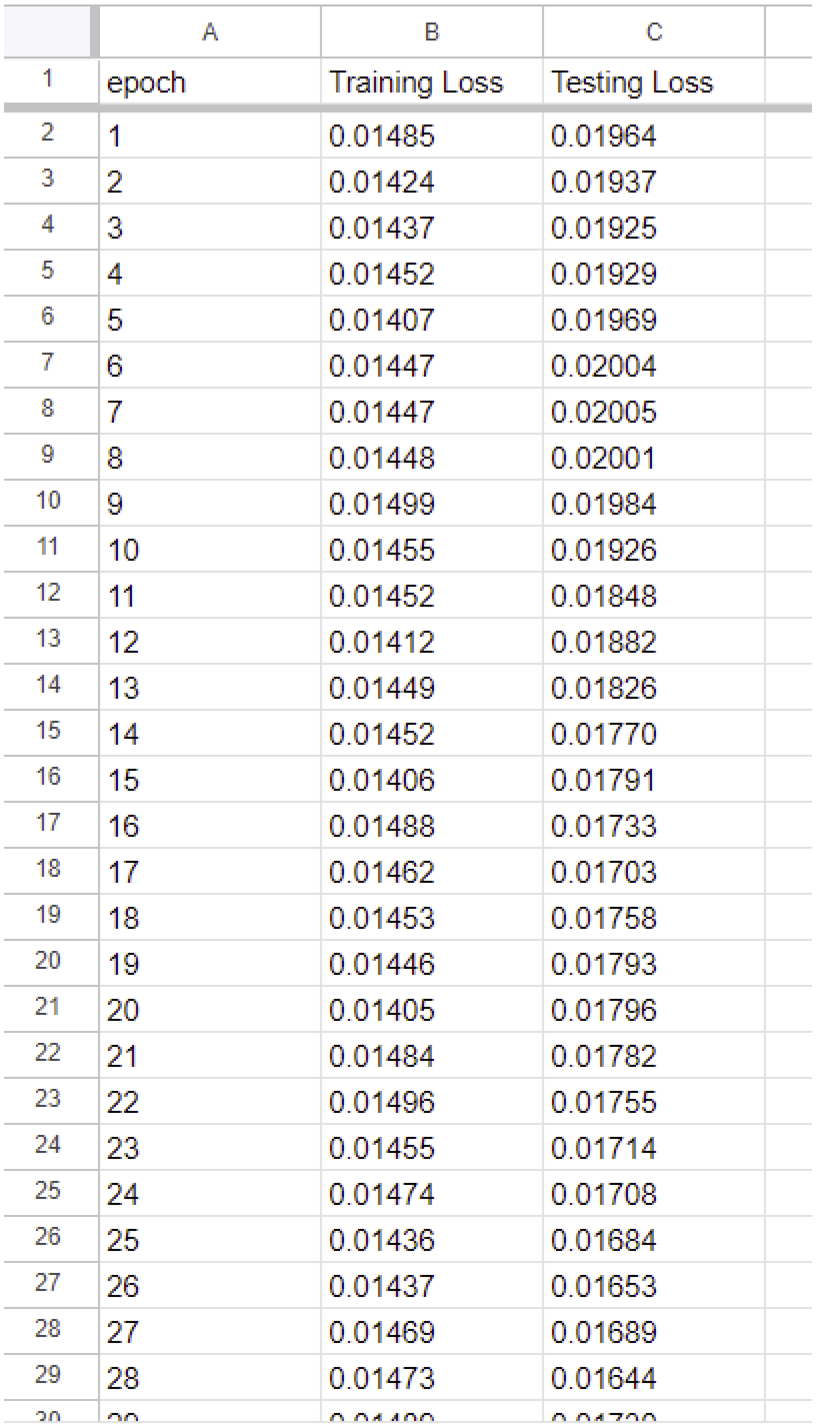


Fig. 14 - Final Output Data Plotted (image by Ryan Junge)



	A	B	C
1	epoch	Training Loss	Testing Loss
2	1	0.01485	0.01964
3	2	0.01424	0.01937
4	3	0.01437	0.01925
5	4	0.01452	0.01929
6	5	0.01407	0.01969
7	6	0.01447	0.02004
8	7	0.01447	0.02005
9	8	0.01448	0.02001
10	9	0.01499	0.01984
11	10	0.01455	0.01926
12	11	0.01452	0.01848
13	12	0.01412	0.01882
14	13	0.01449	0.01826
15	14	0.01452	0.01770
16	15	0.01406	0.01791
17	16	0.01488	0.01733
18	17	0.01462	0.01703
19	18	0.01453	0.01758
20	19	0.01446	0.01793
21	20	0.01405	0.01796
22	21	0.01484	0.01782
23	22	0.01496	0.01755
24	23	0.01455	0.01714
25	24	0.01474	0.01708
26	25	0.01436	0.01684
27	26	0.01437	0.01653
28	27	0.01469	0.01689
29	28	0.01473	0.01644

Fig. 15 - final_output.csv

Chapter 4:

Conclusion and Future Work

Key Findings

Although I succeeded in proving the concept, there are many areas that need refinement before production of a releasable package is feasible.

Firstly, a much larger team is required to optimize the model for commercial release. There are simply too many factors and parameters to take into consideration for this to be a single-person endeavor. Great care and coordination must go into fine-tuning the parameters and architecture of the network in order to achieve accurate and meaningful results.

Second, I lack access to a large database of organized audio recordings. Many such databases exist, but several are locked behind a licensing paywall aimed at selling sound data to audio engineers for media editing.

Future Direction

As mentioned in Chapter 1, I have a vision encapsulating a wide range of possibilities for this technology in the future. Rather than aiming towards a corporate licensing model, I think this software could be aimed at the entire population. Establishing an interface with which the end-user can utilize their smartphone to record their own car's engine sounds would prove revolutionary.

To reach that goal, further progress is required regarding data collection within the domain of automotive combustion engine classification. Perhaps access to manufacturing logs and specifications from a mainstream vehicle production company. For example, if given access to Toyota's vault of system information and data, a model could be trained specifically for use with Toyota vehicles.

In Conclusion

This research project has laid groundwork for a promising future in automotive diagnostics through the utilization of audio classification strategies in the domain of deep learning. Through the use of artificial neural networks, this paper demonstrates the potential for accurately diagnosing vehicle malfunctions and faults based on the sounds emitted by combustion engines.

By following a standardized process of data collection, preprocessing, and neural models, we have provided a proof of concept for the classification of audio recordings for predictive maintenance. This effect could be particularly useful in targeting under-served communities that lack access or means to traditional auto repair services.

The employed methodology, incorporating Mel spectrograms and computer vision via convolutional neural networks, offers a highly flexible framework for identifying engine issues.

Further research into hyperparameter tuning and sound libraries consisting of further expanded engine specifications and specifically labeled mechanical faults is necessary. In theory, if enough data were available, a further specialized framework could be established for each make and model vehicle to enhance accuracy and precision.

In summary, this paper breaks ground on a transformative approach to automotive diagnostics, leveraging state of the art machine learning algorithms in order to democratize access to maintenance services. Through continued exploration and collaboration, the vision of a world where every car owner can diagnose and address mechanical issues with confidence is within reach.

References

Baheti, P. (2021, September 13). *Train test validation split: How to & best practices [2023]*. V7. https://www.v7labs.com/blog/train-validation-test-set

Doshi, K. (2021, May 21). *Audio deep learning made simple: Sound classification, step-by-step*. Medium. https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

Ekman, M. (2022). *Learning deep learning: Theory and practice of neural networks, computer vision, natural language processing, and Transformers using tensorflow*. Addison-Wesley.

Fedorishin, D., Birgiolas, J., Mohan, D. D., Forte, L., Schneider, P. J., Setlur, S., & Govindaraju, V. (2022). Large-Scale acoustic automobile fault detection: diagnosing engines through sound. *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. https://doi.org/10.1145/3534678.3539066

Roberts, L. (2024, January 17). *Understanding the mel spectrogram*. Medium. https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53

Seo, H., Park, J., & Park, Y. (2019, October). Acoustic scene classification using various pre-processed features and convolutional neural networks. In *Proceedings of the Detection and Classification of Acoustic Scenes and Events Workshop (DCASE), New York, NY, USA* (pp. 25-26).

Smith, C. (2023, May 18). *USC Optical Sound Effects Library*. Internet archive: Digital Library of Free & Borrowable Books, movies, Music & Wayback Machine. https://archive.org/details/usc-sound-effect-archive?tab=collection&query=car

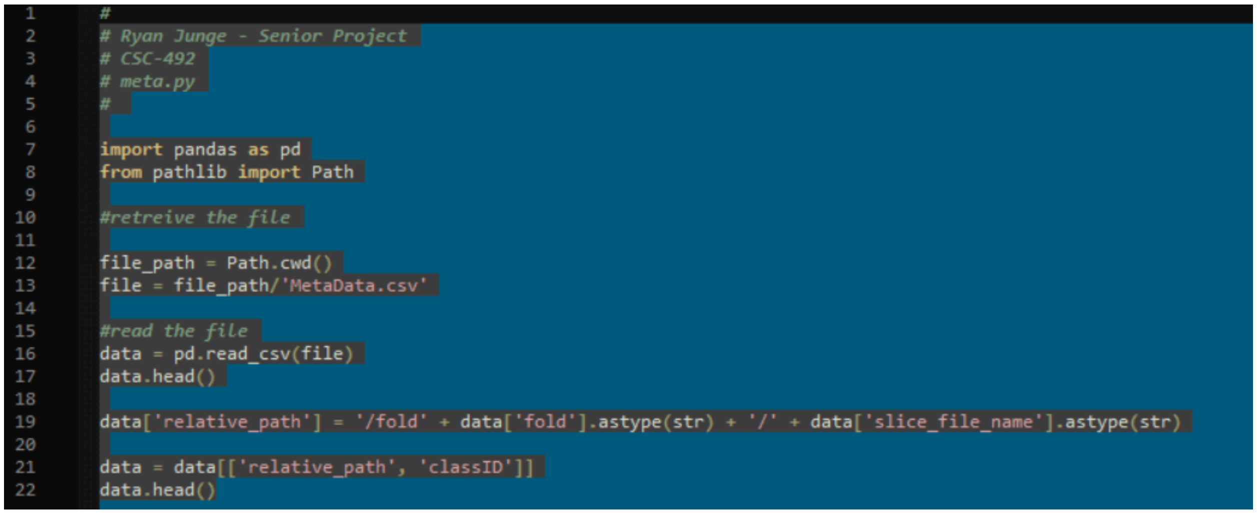
Torque360, Co-written by multiple experts within the Torque360 editorial team. (2024, March 8). *5 problems in the auto repair industry you can help solve*. Torque360. https://blog.torque360.co/problems-in-the-auto-repair-industry-you-can-help-solve/

Vergara, Mary & Ramos, Leo & Rivera, Nestor & Rivas, Francklin. (2023). EngineFaultDB: A Novel Dataset for Automotive Engine Fault Classification and Baseline Results. IEEE Access. PP. 1-1. 10.1109/ACCESS.2023.3331316.

Appendices

Appendix A

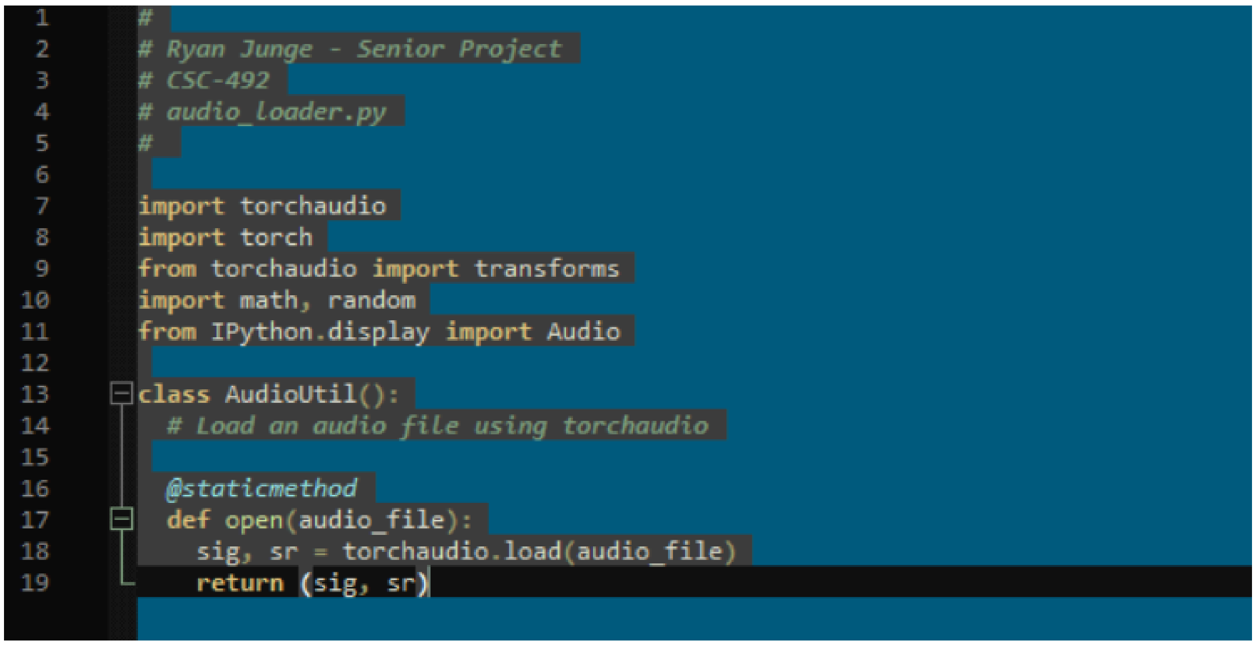
meta.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# meta.py
#

import pandas as pd
from pathlib import Path

#retreive the file

file_path = Path.cwd()
file = file_path/'MetaData.csv'

#read the file
data = pd.read_csv(file)
data.head()

data['relative_path'] = '/fold' + data['fold'].astype(str) + '/' + data['slice_file_name'].astype(str)

data = data[['relative_path', 'classID']]
data.head()
```

Appendix B

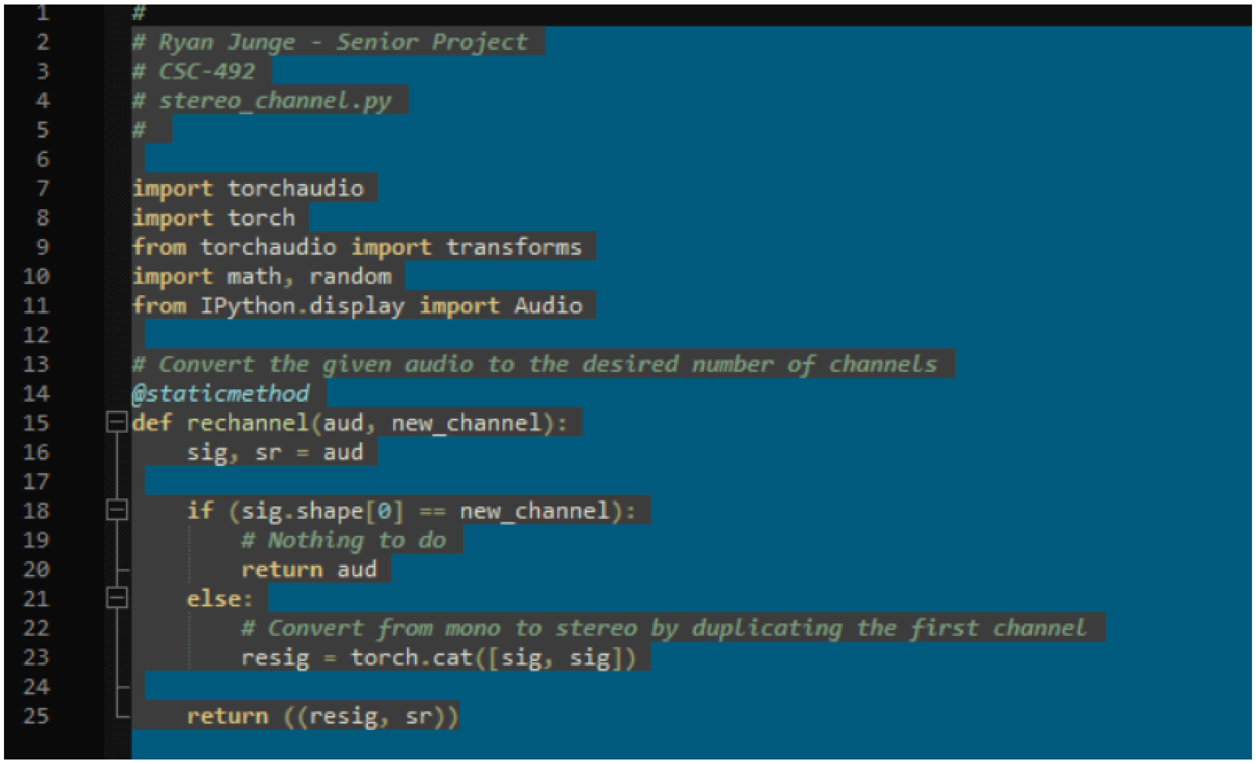
audio_loader.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# audio_loader.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

class AudioUtil():
  # Load an audio file using torchaudio

  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)
```

Appendix C

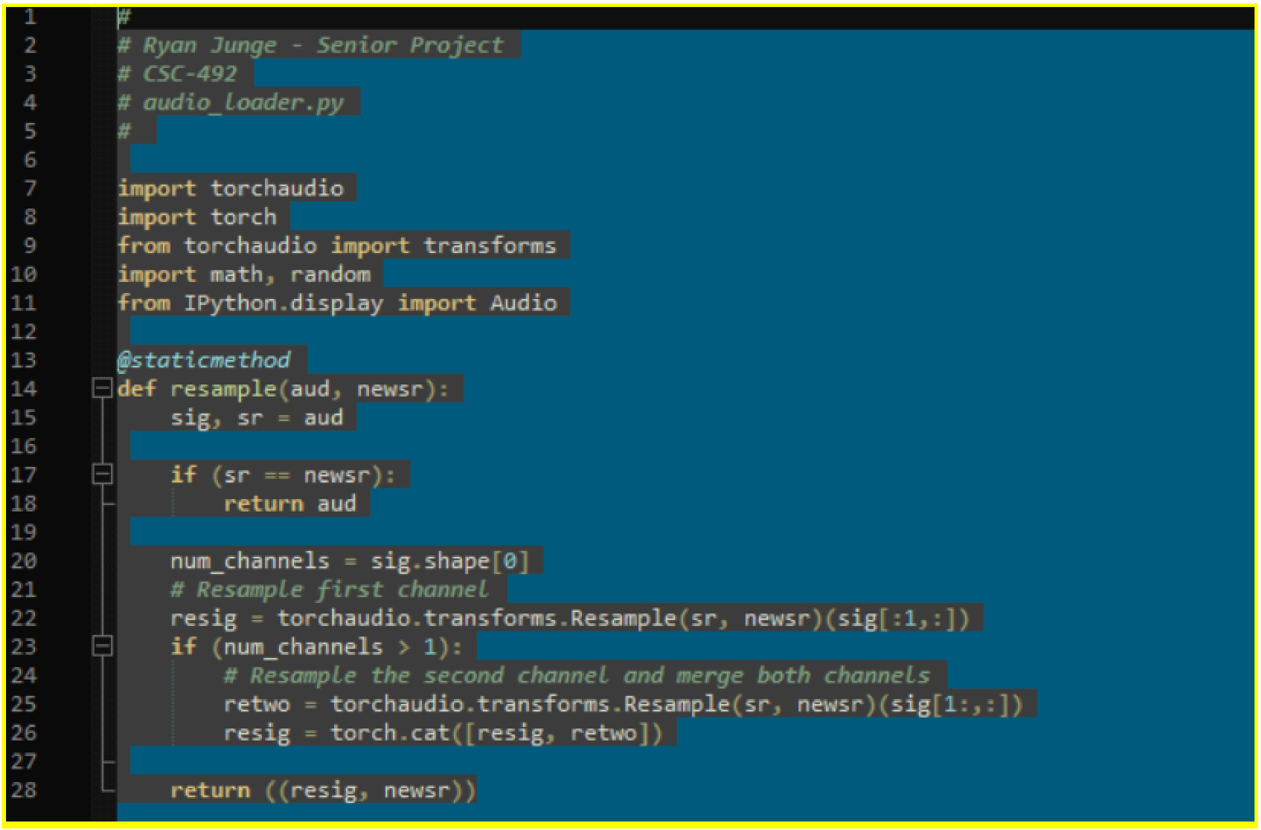
stereo_channel.py



```
#
# Ryan Junge - Senior Project
# CSC-492
# stereo_channel.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

# Convert the given audio to the desired number of channels
@staticmethod
def rechannel(aud, new_channel):
    sig, sr = aud

    if (sig.shape[0] == new_channel):
        # Nothing to do
        return aud
    else:
        # Convert from mono to stereo by duplicating the first channel
        resig = torch.cat([sig, sig])

    return ((resig, sr))
```

Appendix D

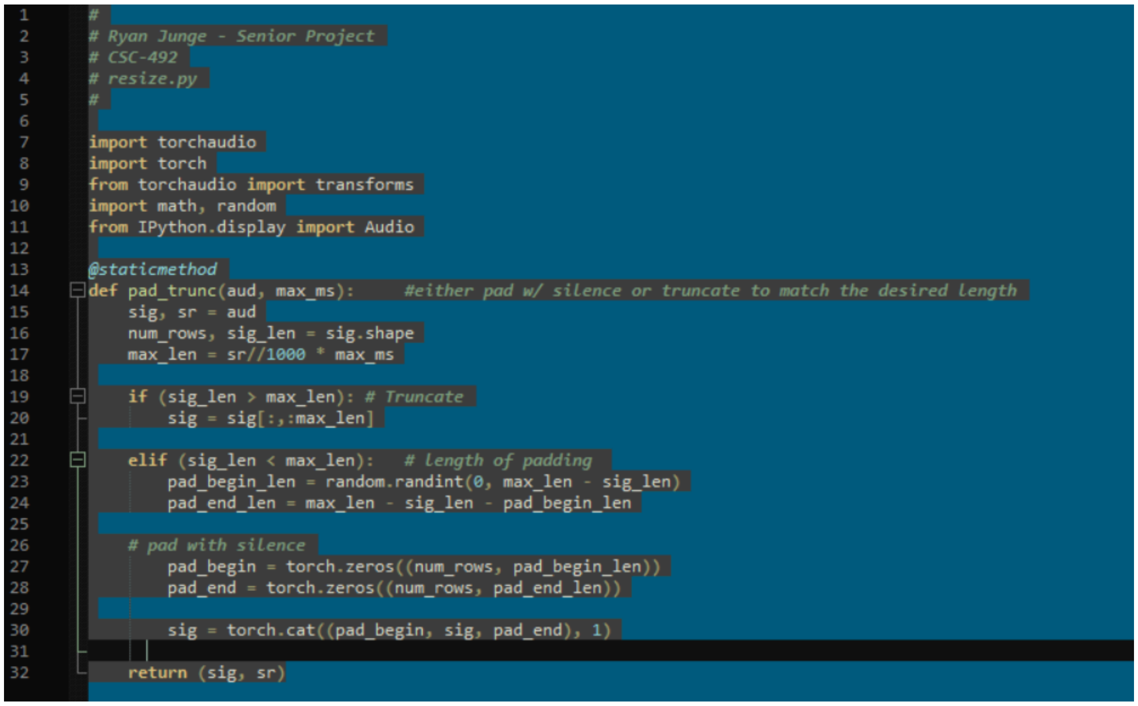
sampling_rate.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# audio_loader.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def resample(aud, newsr):
    sig, sr = aud

    if (sr == newsr):
        return aud

    num_channels = sig.shape[0]
    # Resample first channel
    resig = torchaudio.transforms.Resample(sr, newsr)(sig[:1,:])
    if (num_channels > 1):
        # Resample the second channel and merge both channels
        retwo = torchaudio.transforms.Resample(sr, newsr)(sig[1:,:])
        resig = torch.cat([resig, retwo])

    return ((resig, newsr))
```

Appendix E

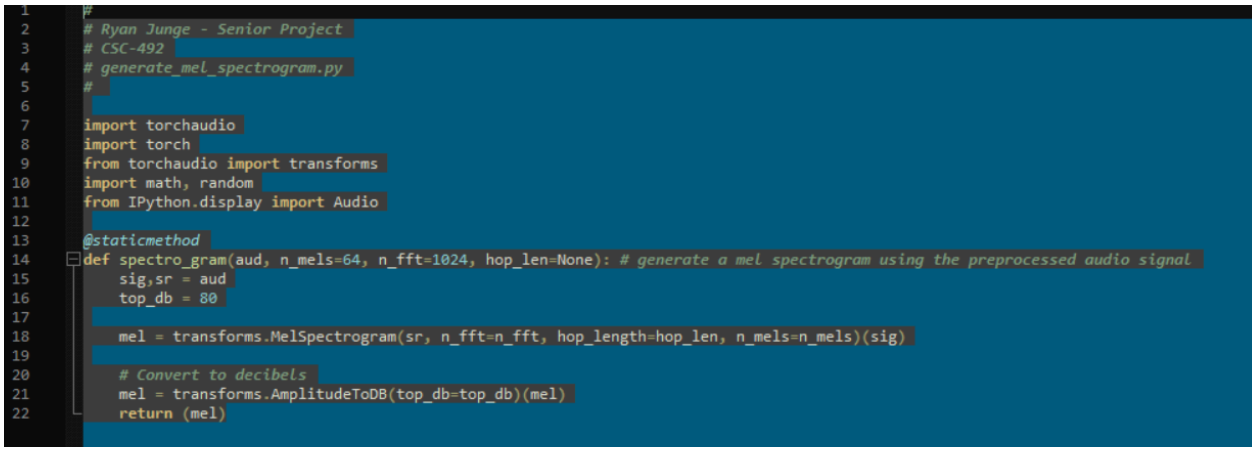
resize.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# resize.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def pad_trunc(aud, max_ms):      #either pad w/ silence or truncate to match the desired length
    sig, sr = aud
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms

    if (sig_len > max_len): # Truncate
        sig = sig[:,:max_len]

    elif (sig_len < max_len):    # length of padding
        pad_begin_len = random.randint(0, max_len - sig_len)
        pad_end_len = max_len - sig_len - pad_begin_len

    # pad with silence
        pad_begin = torch.zeros((num_rows, pad_begin_len))
        pad_end = torch.zeros((num_rows, pad_end_len))

        sig = torch.cat((pad_begin, sig, pad_end), 1)

    return (sig, sr)
```

Appendix F

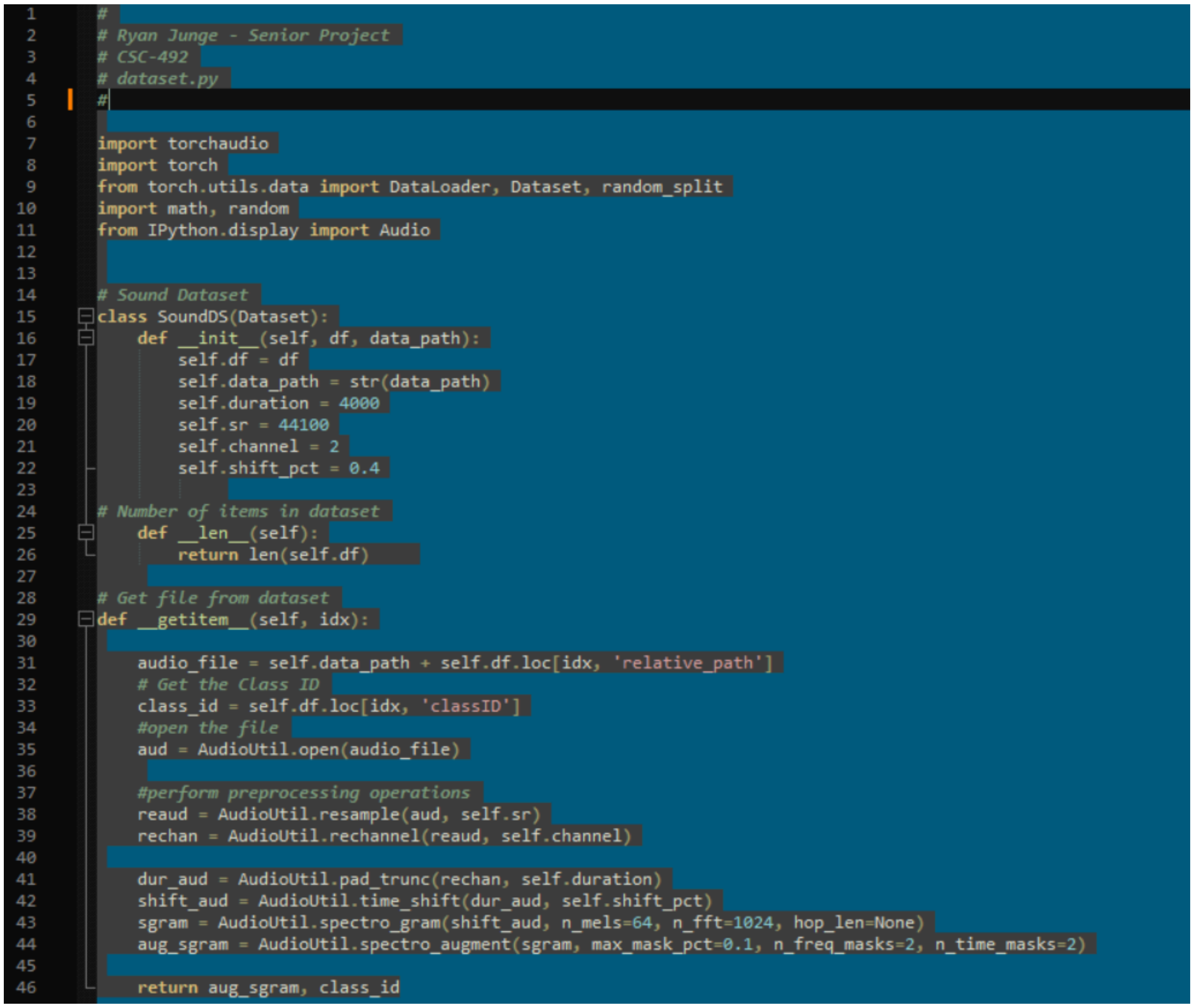
generate_mel_spectrogram.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# generate_mel_spectrogram.py
#

import torchaudio
import torch
from torchaudio import transforms
import math, random
from IPython.display import Audio

@staticmethod
def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None): # generate a mel spectrogram using the preprocessed audio signal
    sig,sr = aud
    top_db = 80

    mel = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

    # Convert to decibels
    mel = transforms.AmplitudeToDB(top_db=top_db)(mel)
    return (mel)
```

Appendix G

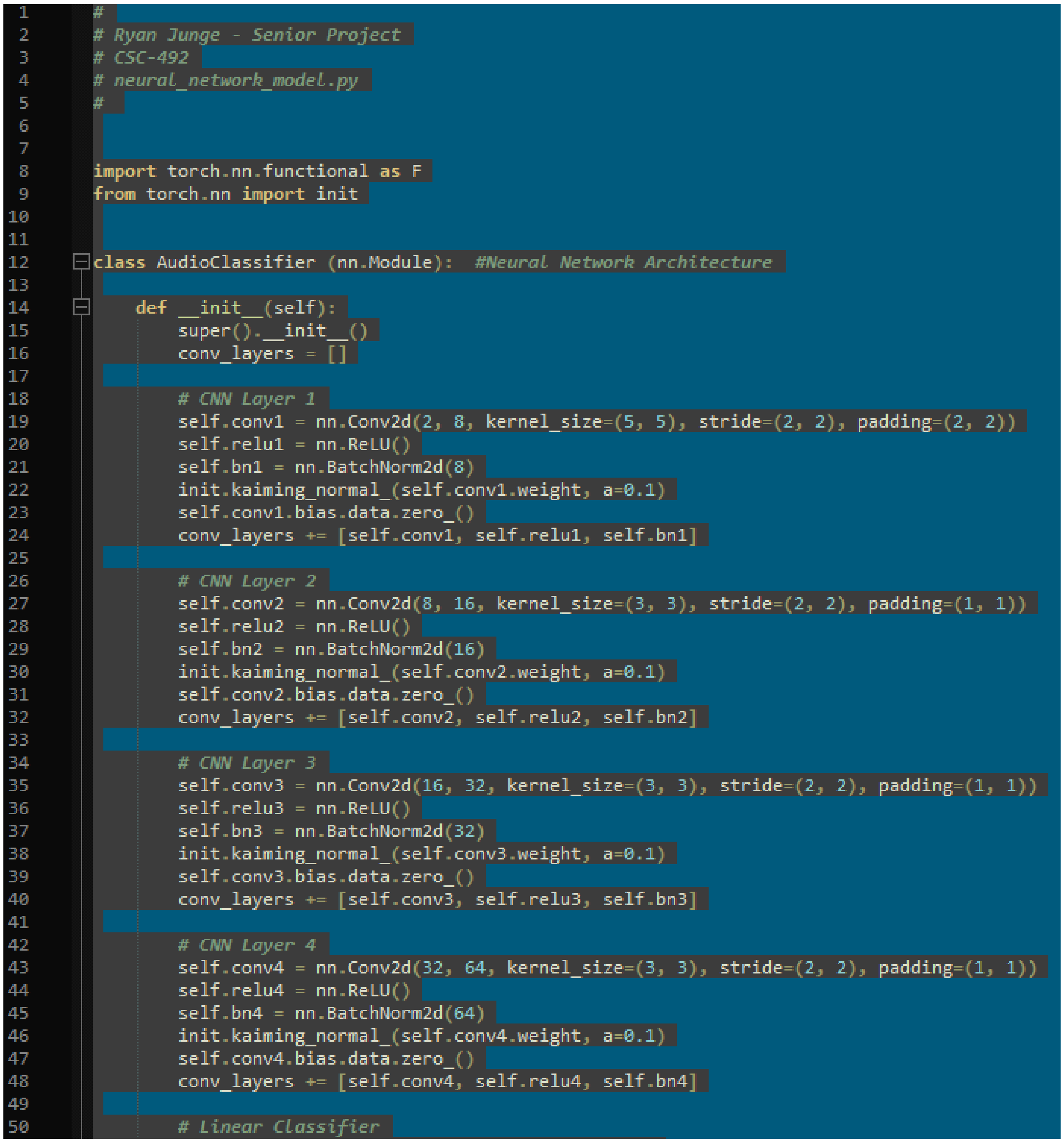
dataset.py



```
#
# Ryan Junge - Senior Project
# CSC-492
# dataset.py
#

import torchaudio
import torch
from torch.utils.data import DataLoader, Dataset, random_split
import math, random
from IPython.display import Audio


# Sound Dataset
class SoundDS(Dataset):
    def __init__(self, df, data_path):
        self.df = df
        self.data_path = str(data_path)
        self.duration = 4000
        self.sr = 44100
        self.channel = 2
        self.shift_pct = 0.4

# Number of items in dataset
    def __len__(self):
        return len(self.df)

# Get file from dataset
def __getitem__(self, idx):

    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']
    #open the file
    aud = AudioUtil.open(audio_file)

    #perform preprocessing operations
    reaud = AudioUtil.resample(aud, self.sr)
    rechan = AudioUtil.rechannel(reaud, self.channel)

    dur_aud = AudioUtil.pad_trunc(rechan, self.duration)
    shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_aud, n_mels=64, n_fft=1024, hop_len=None)
    aug_sgram = AudioUtil.spectro_augment(sgram, max_mask_pct=0.1, n_freq_masks=2, n_time_masks=2)

    return aug_sgram, class_id
```

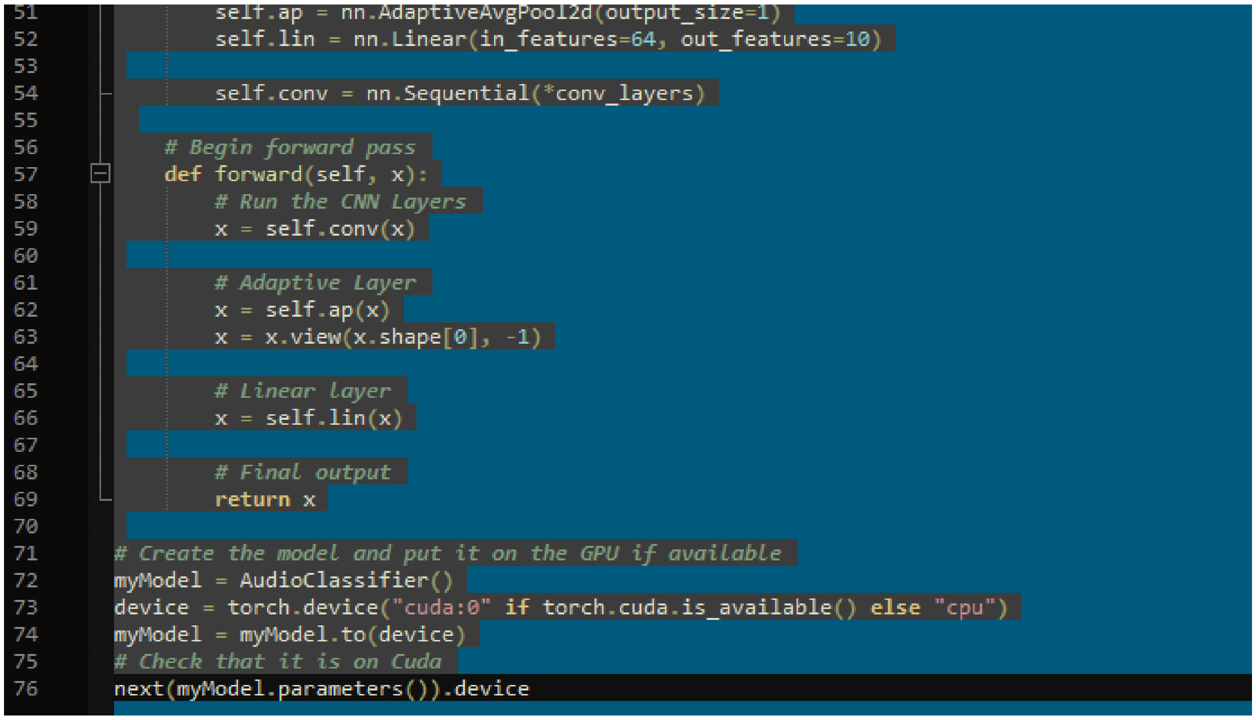
Appendix H

Neural_network_modelt.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# neural_network_model.py
#


import torch.nn.functional as F
from torch.nn import init


class AudioClassifier (nn.Module):  #Neural Network Architecture

    def __init__(self):
        super().__init__()
        conv_layers = []

        # CNN Layer 1
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # CNN Layer 2
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]

        # CNN Layer 3
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]

        # CNN Layer 4
        self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(64)
        init.kaiming_normal_(self.conv4.weight, a=0.1)
        self.conv4.bias.data.zero_()
        conv_layers += [self.conv4, self.relu4, self.bn4]

        # Linear Classifier
```

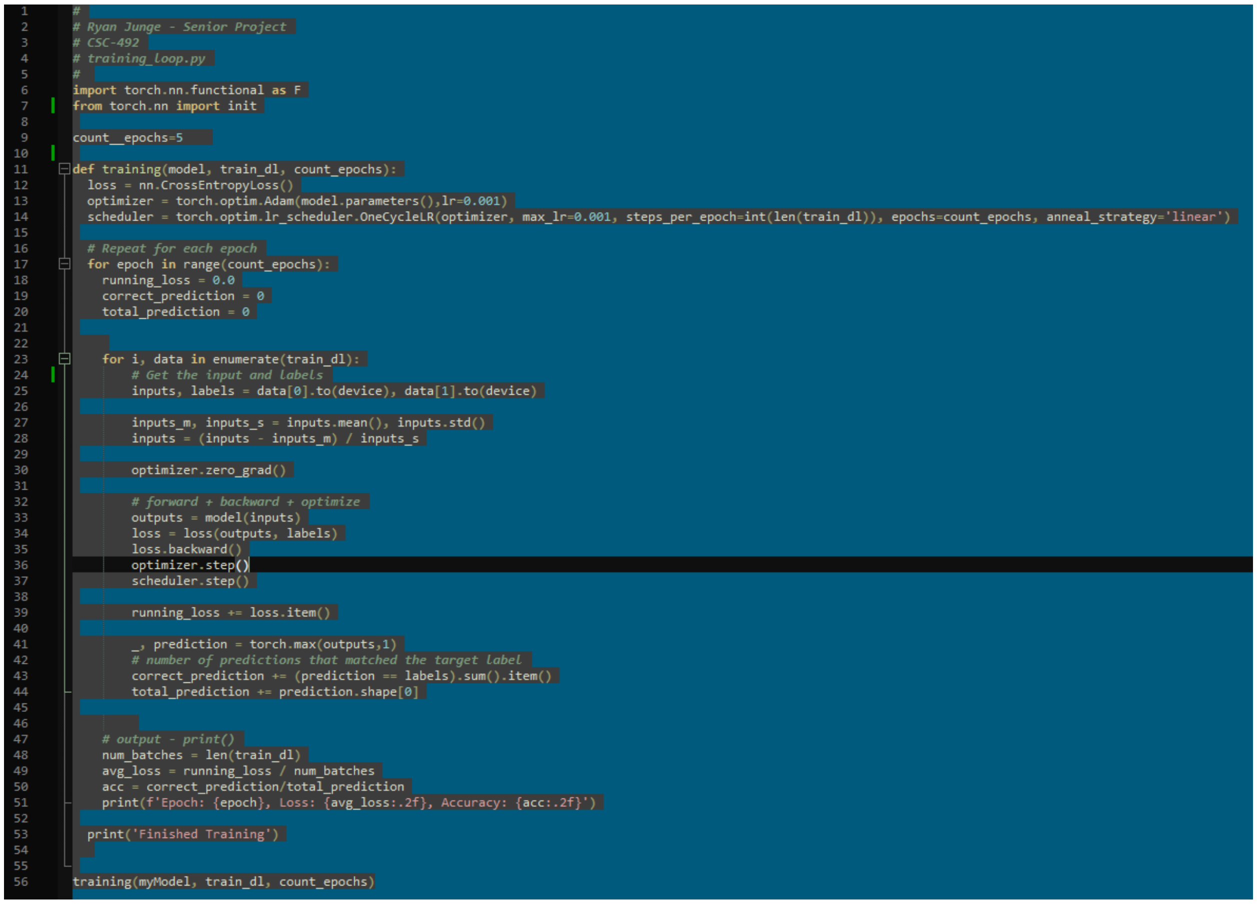
Appendix H - Continued



```python
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=10)

        self.conv = nn.Sequential(*conv_layers)

    # Begin forward pass
    def forward(self, x):
        # Run the CNN Layers
        x = self.conv(x)

        # Adaptive Layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear layer
        x = self.lin(x)

        # Final output
        return x

# Create the model and put it on the GPU if available
myModel = AudioClassifier()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
myModel = myModel.to(device)
# Check that it is on Cuda
next(myModel.parameters()).device
```

Appendix I

training_loop.py



```python
#
# Ryan Junge - Senior Project
# CSC-492
# training_loop.py
#
import torch.nn.functional as F
from torch.nn import init

count__epochs=5

def training(model, train_dl, count_epochs):
  loss = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001, steps_per_epoch=int(len(train_dl)), epochs=count_epochs, anneal_strategy='linear')

  # Repeat for each epoch
  for epoch in range(count_epochs):
    running_loss = 0.0
    correct_prediction = 0
    total_prediction = 0


    for i, data in enumerate(train_dl):
        # Get the input and labels
        inputs, labels = data[0].to(device), data[1].to(device)

        inputs_m, inputs_s = inputs.mean(), inputs.std()
        inputs = (inputs - inputs_m) / inputs_s

        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = loss(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        running_loss += loss.item()

        _, prediction = torch.max(outputs,1)
        # number of predictions that matched the target label
        correct_prediction += (prediction == labels).sum().item()
        total_prediction += prediction.shape[0]


    # output - print()
    num_batches = len(train_dl)
    avg_loss = running_loss / num_batches
    acc = correct_prediction/total_prediction
    print(f'Epoch: {epoch}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')

  print('Finished Training')


training(myModel, train_dl, count_epochs)
```