Defence against VCS Adversirial Attacks

Pawan Vijayanagar
Arizona State University
Tempe, USA
psvijaya@asu.edu

Atharva Anand Barwe
Arizona State University
Tempe, USA
abarwe@asu.edu

***Abstract*— This document is a proposal for CSE598 Machine Learning Security and Fairness taught by Dr. Chaowei Xiao. The idea behind this proposal is to develop ways in which we can defend against adversarial attacks on Voice Controlled Systems.**

I. Problem definition

The idea behind the paper SMACK: Semantically Meaningful Adversarial Audio Attack is to improve upon already existing speech recognition attacks to remove perturbations that can be heard and be identified easily in the attacking audio track. To address this problem the authors developed and introduced a new attack as the name of the paper suggests Semantically Meaningful Adversarial Audio Attack (SMACK) such that the resulting audio of the added acoustics is not heard by the human ears. The efficacy of smack was tested on 5 transcription systems and 2 speech recognition assistant devices.

The authors of the paper explore the manipulation of prosody i.e the pattern of rhythm and sound to base their attack which is a semantic attribute to generate adversarial attacks. The idea behind this is that it makes the adversarial attack stealthy. As prosody varies from person to person, it is not possible to have a fixed system to work them all. Hence, the authors proposed an adapted generative model to build finely tuned adversarial attacks. SMACK utilizes the confidence score in speech recognition to iteratively update the estimation on the speaker verification threshold, significantly reducing the number of queries. In other referenced papers, we see that authors merge pre-existing audio samples of tailored noise at a non-perceivable frequency with the intended audio sample to create an adversarial attack. This adversarial attack on the other hand is easily detected by the human ear when closely observed.

Voice Controlled Systems rely and utilize a simple analog microphone to input user data, along with audio these MEMS microphones are also sensitive to photons - there exist attacks on the microphone such as laser-based audio injection attacks where an attacker injects desired audio signals to a target VCS by aiming an amplitude-modulated light at the microphone's aperture. Attacks using a laser-based approach were observed to be extremely effective at even 110m away.

As speech recognition and voice controllable systems are getting used by the everyday population and more importantly is being utilized in security sensitive areas where voice input is the only source of information exchange, there must exist preventative methods that fight against such adversarial attacks.

II. Challenges and Main Contribution

A limitation of this study is that since it involves speech and text, there is a restriction to how much we can perturb the samples to create adversarial samples. This is done to preserve semantics and is likely present in varying degrees in studies involving semantics. This means that if the target transcriptions have a large phenomic difference with the original speech, it is particularly difficult to craft adversarial samples.

Another drawback of this approach is that it isn't well equipped to work in real time. In our demonstrations, we will show adversarial samples that have been generated in advance for the adversarial attack and sent to the target's API or played over a speaker near the target device. However, in an ideal scenario, the attacking system must be able to generate adversarial perturbations in real time and play them alongside background sounds. This requires the attacker to know the timestamp of the audio and produce the perturbation on the spot. A simpler alternative is word-based prosody perturbation. Additionally, it is difficult to alter things like accent or speech rate just by adding noise; this is what differentiates our method from traditional LP approaches.

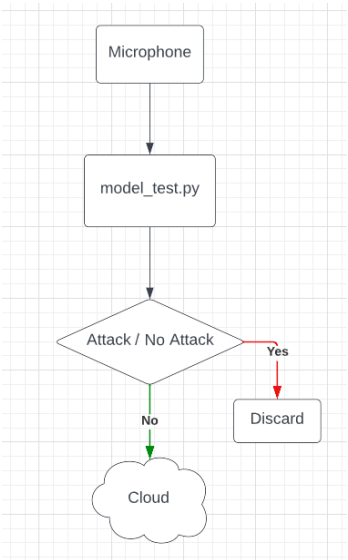

Figure 1. Block diagram of our proposed method implemented in VCS systems

We elected to improve upon an audio identification system using the concepts we learned in class. A basic audio recognition system can identify the source of a sound i.e., given an audio

input, it can identify where it came from (car, bird, construction equipment, etc.). This is already a very practical application by itself. What we aimed to do is add another 'layer of security' to this application by making it resistant to perturbed audio attacks. In addition to the audio recognition functionality, the proposed model would also be capable of detecting sounds laced with a perturbation. Overall, what we hoped to achieve is a combination of a fully functional system and a perturbed attack resistance. In the best-case scenario, the crafted malicious input would successfully be identified before it can be misinterpreted by the system.

One of the challenges we faced was getting the model to train quickly enough. As an example, our model trained for 1.5 hrs when run on a system with the following specifications - Mac OS Ventura, i9, 16GB RAM (All processes running on CPU as AMD not supported by pytorch)

Another challenge, due to the limited timeframe we had was that we weren't able to conduct an extensive hyperparameter search to find the optimal hyperparameters for our model. If it weren't prohibitively expensive, even testing a handful of models would have been quite productive toward the result.

Finally, a limitation was the dataset size; the 8K dataset contained less than 9,000 labeled sound excerpts and had a hand in how much the model could learn. In the ideal case, we would have liked a dataset of at least 100,000 samples. However, as the first limitation states, the training time was already an issue. Increasing the dataset size, even if available, would have hampered the training time.

As we needed a perturbed dataset, we created a program on python to convolve the 8k dataset along with 10 random perturbed noise file samples generated with the help of the paper "Adversarial Attacks Against Automatic Speech Recognition Systems via Psychoacoustic Hiding". The final trained dataset consisted of 8k non-attack audio files and 6k attack audio files with a total classification list of 11. These are the numeric representations of each label shown in Table 1 below:

Actual sound	Corresponding numeric label
air_conditioner	0
car_horn	1
children_playing	2
dog_bark	3
drilling	4
engine_idling	5
gun_shot	6
jackhammer	7
siren	8
street_music	9
generated perturbed attack audio	10

Table 1. Data classification based on audio type

As mentioned above, we synthesized 37% of our final dataset from the original audio files. Each audio sample has a corresponding audio sample that has been perturbed and is essentially an 'attack' sample. For example, for each existing sound of traffic, there is another similar sound that only has the perturbation added to it.

III. Most Related Prior Work and Its Shortcomings

The work we have based our addition on starts off by converting sound files into spectrograms. It then proceeds to feed the spectrograms into a Convolutional Neural Network followed by a few Fully Connected layers. The final outputs are the probabilities of the sound being from each of the potential sources.

The above work simply implements a classification model that classifies audio samples using a DNN classification tool but does not classify perturbated audio. The shortcomings of this work would create a dataset of attack audio and implement the above classification algorithm to distinguish between an attack and other non-attack classes.

IV. Methodology

This above described work uses the Urban Sound 8K dataset[5] that consists of a corpus of ordinary sounds recorded from day-to-day city life. As stated above, the sounds are taken from 10 classes such as drilling, dogs barking, and sirens and a separate class is created by convolving these 10 classes with a couple perturbed audio samples. Each sound sample is labeled with the class to which it belongs. The class labels are entirely numeric : from 0 to 9. Figure 2 a) is one of the audio samples and Figure 2 b) is the perturbed version of the sound created from audio convolution of non-attack audio and attack noise found in paper [x] by Lea Schönherr et.al.

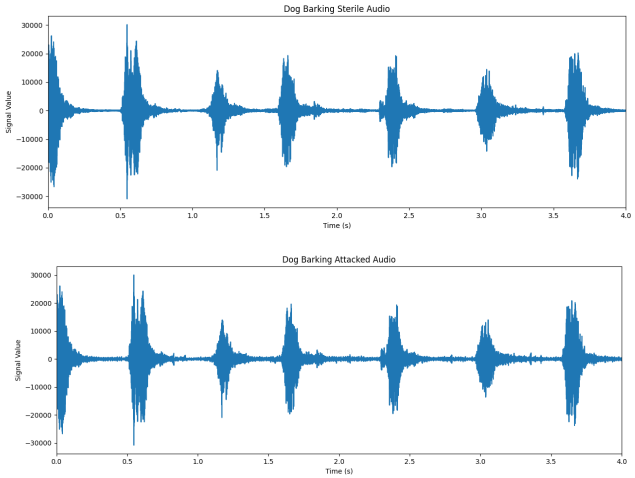

Figure 2. top a) sterile/non-attacked dog barking audio L-CH waveform and bottom b) attacked dog barking audio L-CH waveform

Just like a standard Deep Learning project, we follow a simple workflow:

Figure 3 is a sample of the dataset:

	A	B	C	D	E	F	G	H
1	slice_file_na	fsID	start	end	salience	fold	classID	class
2	100032-3-0-(	100032	0	0.317551	1	5	3	dog_bark
3	100263-2-0-	100263	58.5	62.5	1	5	2	children_playing
4	100263-2-0-	100263	60.5	64.5	1	5	2	children_playing
5	100263-2-0-	100263	63	67	1	5	2	children_playing
6	100263-2-0-	100263	68.5	72.5	1	5	2	children_playing
7	100263-2-0-	100263	71.5	75.5	1	5	2	children_playing
8	100263-2-0-	100263	80.5	84.5	1	5	2	children_playing
9	100263-2-0-	100263	1.5	5.5	1	5	2	children_playing
10	100263-2-0-	100263	18	22	1	5	2	children_playing
11	100648-1-0-(	100648	4.823402	5.471927	2	10	1	car_horn
12	100648-1-1-(	100648	8.998279	10.052132	2	10	1	car_horn

Figure 3. Sample dataset

Note that one of the columns contains only the name of the audio file. This isn't something we can feed into a Machine Learning model. To resolve this, we need to have the audio data processed in a way that the model can understand. Here is the creative part, the preprocessing will be performed on the spot during runtime. If we had images instead, we would likely use a similar approach since they both use a relatively large amount of memory. Thus, it makes sense to keep the audio file names as they are in the dataset. The imported data into the ML algorithm was created using pandas to incorporate file location of the 8k perturbed audio as well as non-perturbed audio files, each one had their class labels labeled accordingly in the pandas CSV. Please note that the perturbed audio generation and file location allocation (dataset and label generation) in pandas was automated using our developed python script.

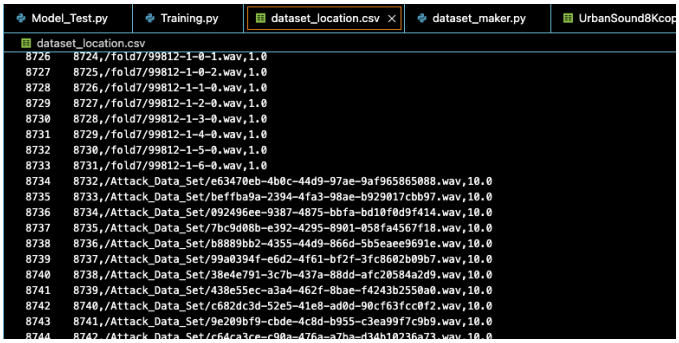

Figure 4. showing generated audio number, audio location, and audio classification number (0-10)

Another little memory optimization step is that we only load the audio for a single batch at a time; since the model is trained on one batch at a time, this ensures that only a batch's worth of memory is occupied at a time.

There is a series of transformations applied to the audio since not all audio files are in the same form. Some of these steps are conversion to stereo (single audio channel to double), resizing to a fixed length, and conversion to a Mel spectrogram. The defined transforms aren't processing the audio beforehand. They perform their respective operations when we feed the data to the model.

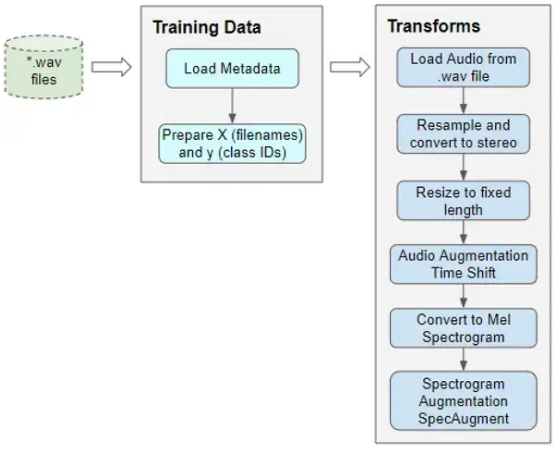

Figure 5. Flowchart of training algorithm

Figure 6. 5921/6245 perturbed audio files generated in 18 minutes and 29 seconds.

V. Experimental Setup and Results

As we coded the algorithm to accept perturbed as well as unperturbed audio, we ran the machine learning algorithm to 25 epochs that roughly took an hour to learn with the equipment we had in hand and saved the model to run our tests, available in our zipped file. "Dataset_maker.py" is the python script that convolves attack audio and 8k dataset with a location and label table "data_set.csv" to load the attack and non-attack audio files in the "Model_Train.py" file. There were issues with multiple files in folder 08 of 8k dataset and hence, those were skipped in dataset maker and limited our attack dataset to 6245 perturbed audio files. The python script "Model_Train.py" is built to train, plot loss and accuracy vs epoch on the generated dataset of 8k and 6k perturbed (attack) audio files and saves the trained model to a first_model.pth file. The first_model.pth model file is tested on test data using "Triaining.py" for accuracy and confusion matrix generation. We found that our accuracy reached a stagnation point of roughly 81% on train data and an average accuracy of 81% on test data. The following are the graphs we got for the 25 epochs:

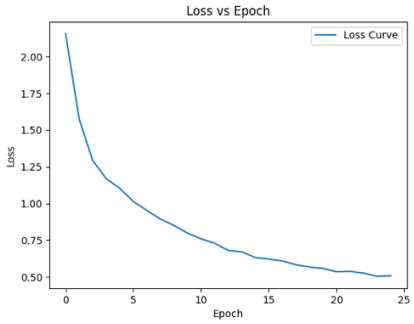

Figure 7. Loss vs Epoch graph of training

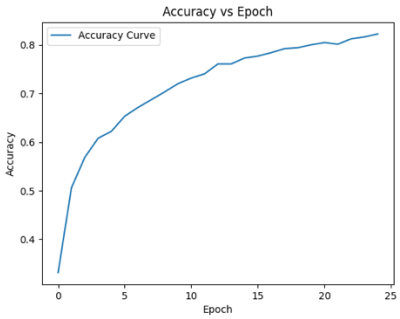

Figure 8. Accuracy vs Epoch graph of training

As it would be an interesting find, out of curiosity we implemented a confusion matrix to perform further data analysis to find the best and work performing classification our ML algorithm produced. The following is the result of our confusion matrix:

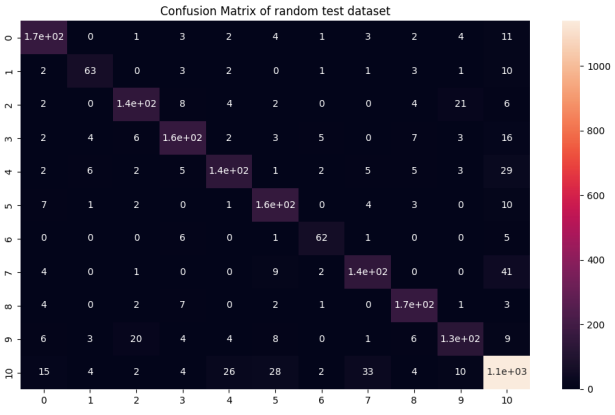

Figure 9. Confusion matrix of test dataset. Test dataset is randomly generated and contains roughly 37% perturbed audio (10) and 63% non-perturbed audio (0-9) and is not identical to train data.

```
Class 10 has occurred 1268 times in test dataset
Class 10 has been correctly predicted in test 1280 times
```

Figure 10. Attack class has 99.06% accuracy

Once the code was run, the model produced outputs that are capable of distinguishing perturbed attacks from non-perturbed attacks very easily. The graphs of accuracy vs epoch and loss vs epoch show that the model is being trained correctly and that the model produces accurate classification. Using the graphs, we could also see that the loss graph just barely settles to a value of 0.51 which suggests that the model needs to be trained with more epochs (just enough for proof of concept). As the training took over a couple hours to complete, it was not worth the results to doing close to 50 epochs (25 was enough for 99% attack classification).

The model essentially was capable of distinguishing attacks with an accuracy of 99.06%, which is perfect and a total accuracy of classification of both attacks and non-attacks of close to 82%. The reason behind a 99.06% accuracy for classification could be because the attack dataset is close to 40% of the test data and train data and are not classified into further subcategories i.e train-attack, children playing-attack, car-horn-attack. Essentially, the system only cares if an attack has been detected regardless of audio classification type, i.e., the system is binary for attack classification. This would be ideal as more classification only means more processing time and if the system can classify and predict VNC attacks just based on audio samples then it could certainly be said that the system works perfectly. It is important to note that the model did not misclassify other non-attack classifications into the classification of attack type which would help prevent false alarms.

Although we used 8k audio data for our training as it was readily available and as this is just a proof of concept, our results would accurately match real world tests if we conducted the same test with a larger human voice dataset as opposed to random environmental sounds. Although, an argument could be made that the system should support any audio-based attacks ranging from music to random audio files to a cat purring - perturbed audio could be generated for any audio file and hence it would be ideal to build a dataset with random audio tracks that can possibly include most if not all possible audio inputs that a VCS might hear on the daily. And thus, we conclude that as VCS are getting more and more popular it would be in the best interest for VCS manufacturers such as Amazon, Google, Microsoft and Meta to include such an attack detection model into its edge processing algorithm.

Link to the GitHub: https://github.com/Dr-Drone/CSE598---ML-Security-and-Awareness

REFERENCES

[1] Vassil Panayotov. LibriSpeech. https://paperswithcode.com/dataset/librispeech

[2] NDSS Symposium. (2019, April 2). NDSS 2019 - Adversarial Attacks Against ASR Systems via Psychoacoustic Hiding. [Video]. YouTube. https://www.youtube.com/watch?v=l_AkXxZt10I

[3] Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems | USENIX. (n.d.). https://www.usenix.org/conference/usenixsecurity20/presentation/sugawara

[4] Yitao He, Junyu Bian, Xinyu Tong, Zihui Qian, Wei Zhu, Xiaohua Tian, and Xinbing Wang. 2019. Canceling Inaudible Voice Commands Against Voice Control Systems. In The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19). Association for Computing Machinery, New York, NY, USA, Article 28, 1–15. https://doi.org/10.1145/3300061.3345429

[5] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. 2014. A Dataset and Taxonomy for Urban Sound Research. In Proceedings of the 22nd ACM international conference on Multimedia (MM '14). Association for Computing Machinery, New York, NY, USA, 1041–1044. https://doi.org/10.1145/2647868.2655045

[6]

[7] Philip Jackson and Sanaul Haq. Surrey Audio-Visual Expressed Emotion (SAVEE) Database. 2 April 2015. University of Surrey. http://kahlan.eps.surrey.ac.uk/savee/

[8] Ardila. Common Voice. https://paperswithcode.com/dataset/common-voice

[9] Yitao He, Junyu Bian, Xinyu Tong, Zihui Qian, Wei Zhu, Xiaohua Tian, and Xinbing Wang. 2019. Canceling Inaudible Voice Commands Against Voice Control Systems. In The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19). Association for Computing Machinery, New York, NY, USA, Article 28, 1–15. https://doi.org/10.1145/3300061.3345429

[10] Takeshi Sugawara and Benjamin Cyr and Sara Rampazzi and Daniel Genkin and Kevin Fu,

https://www.usenix.org/conference/usenixsecurity20/presentation/sugawara

[11] "Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems", Takeshi Sugawara, The University of Electro-Communications; Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu, University of Michigan, https://www.usenix.org/conference/usenixsecurity20/presentation/sugawara

[12] "NDSS 2019 - Adversarial Attacks Against ASR Systems via Psychoacoustic Hiding", https://youtu.be/l_AkXxZt10I

[13] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, & Dorothea Kolossa (2019). Adversarial Attacks Against Automatic Speech Recognition Systems via Psychoacoustic Hiding. In Network and Distributed System Security Symposium (NDSS).

```
# ----------------------------
# Creating an attack Dataset
# ----------------------------
import pandas as pd
from pathlib import Path
from pydub import AudioSegment
import uuid
import numpy

from tqdm import tqdm

#pydub used to convolve sound data


#Instructions
''''
You need to have a filder in the subdirectry of UrbanSound8K/audio/ named Attack_Data_Set for
this script to work
You also need to have a the attack noise stored in a folder called /attack_noise/ in root

The script used metadata file to read original file location, it then uses that to add noise
to the data
# Attcack noise taken from: https://github.com/rub-
ksv/adversarialattacks/tree/316b1f026cc55ee5c712240be656c68da940d743/docs/audio

'''

download_path = Path.cwd()/'UrbanSound8Kcopy.csv'

# Read metadata file
metadata_file = '/Users/pawanvijayanagar/Documents/ASU Masters/CSE598 - Machine Learning
Security and Fairness/Midterm_Check/UrbanSound8K/metadata/UrbanSound8K.csv'
df = pd.read_csv(metadata_file)
df.head()

# Construct file path by concatenating fold and file name
df['relative_path'] = '/fold' + df['fold'].astype(str) + '/' +
df['slice_file_name'].astype(str)

# Take relevant columns
df = df[['relative_path', 'classID']]
df.head()


print(len(df))
print(df.loc[1,'relative_path'])
length = len(df)

for i in tqdm(range(6245)):
    sound_1 = AudioSegment.from_file("UrbanSound8K/audio" + df.loc[i,'relative_path'])
    sound_2 = AudioSegment.from_file("attack_noise/attack_noise.mp3")
    combined = sound_1.overlay(sound_2)
    filename = str(uuid.uuid4())
    combined.export("./UrbanSound8K/audio/Attack_Data_Set/"+filename+".wav", format='wav')
    df.loc[i + length,'relative_path'] = "/Attack_Data_Set/" +filename+ ".wav"
    df.loc[i + length,'classID'] = 10
df.to_csv("dataset_location.csv")
#17463
```

```
# ----------------------------
# Prepare training data from Metadata file
# ----------------------------
import pandas as pd
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sn


y_pred = []
y_true = []



download_path = Path.cwd()/'UrbanSound8K'

# Read metadata file
metadata_file = '/Users/pawanvijayanagar/Documents/ASU Masters/CSE598 - Machine Learning
Security and Fairness/Midterm_Check/UrbanSound8K/metadata/UrbanSound8K.csv'
df = pd.read_csv(metadata_file)
df.head()

# Construct file path by concatenating fold and file name
df['relative_path'] = '/fold' + df['fold'].astype(str) + '/' +
df['slice_file_name'].astype(str)

# Take relevant columns
df = df[['relative_path', 'classID']]
df.head()
df = pd.read_csv('dataset_location.csv', usecols= ['relative_path', 'classID'])
df['classID'] = df['classID'].astype(int)

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

class AudioUtil():
  # ----------------------------
  # Load an audio file. Return the signal as a tensor and the sample rate
  # ----------------------------
  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)

  # ----------------------------
  # 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

    if (new_channel == 1):
      # Convert from stereo to mono by selecting only the first channel
```

```
      resig = sig[:1, :]
    else:
      # Convert from mono to stereo by duplicating the first channel
      resig = torch.cat([sig, sig])

    return ((resig, sr))


  # ----------------------------
  # Since Resample applies to a single channel, we resample one channel at a time
  # ----------------------------
  @staticmethod
  def resample(aud, newsr):
    sig, sr = aud

    if (sr == newsr):
      # Nothing to do
      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))


  # ----------------------------
  # Pad (or truncate) the signal to a fixed length 'max_ms' in milliseconds
  # ----------------------------
  @staticmethod
  def pad_trunc(aud, max_ms):
    sig, sr = aud
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms

    if (sig_len > max_len):
      # Truncate the signal to the given length
      sig = sig[:,:max_len]

    elif (sig_len < max_len):
      # Length of padding to add at the beginning and end of the signal
      pad_begin_len = random.randint(0, max_len - sig_len)
      pad_end_len = max_len - sig_len - pad_begin_len

      # Pad with 0s
      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)

  # ----------------------------
  # Shifts the signal to the left or right by some percent. Values at the end
  # are 'wrapped around' to the start of the transformed signal.
  # ----------------------------
  @staticmethod
  def time_shift(aud, shift_limit):
    sig,sr = aud
    _, sig_len = sig.shape
    shift_amt = int(random.random() * shift_limit * sig_len)
```

```
    return (sig.roll(shift_amt), sr)


  # ----------------------------
  # Generate a Spectrogram
  # ----------------------------
  @staticmethod
  def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None):
    sig,sr = aud
    top_db = 80

    # spec has shape [channel, n_mels, time], where channel is mono, stereo etc
    spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

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


  # ----------------------------
  # Augment the Spectrogram by masking out some sections of it in both the frequency
  # dimension (ie. horizontal bars) and the time dimension (vertical bars) to prevent
  # overfitting and to help the model generalise better. The masked sections are
  # replaced with the mean value.
  # ----------------------------
  @staticmethod
  def spectro_augment(spec, max_mask_pct=0.1, n_freq_masks=1, n_time_masks=1):
    _, n_mels, n_steps = spec.shape
    mask_value = spec.mean()
    aug_spec = spec

    freq_mask_param = max_mask_pct * n_mels
    for _ in range(n_freq_masks):
      aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, mask_value)

    time_mask_param = max_mask_pct * n_steps
    for _ in range(n_time_masks):
      aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_value)

    return aug_spec



from torch.utils.data import DataLoader, Dataset, random_split
import torchaudio

# ----------------------------
# 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 i'th item in dataset
  # ----------------------------
  def __getitem__(self, idx):
    # Absolute file path of the audio file - concatenate the audio directory with
    # the relative path
    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']

    aud = AudioUtil.open(audio_file)
    # Some sounds have a higher sample rate, or fewer channels compared to the
    # majority. So make all sounds have the same number of channels and same
    # sample rate. Unless the sample rate is the same, the pad_trunc will still
    # result in arrays of different lengths, even though the sound duration is
    # the same.
    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


from torch.utils.data import random_split

myds = SoundDS(df, data_path = '/Users/pawanvijayanagar/Documents/ASU Masters/CSE598 -
Machine Learning Security and Fairness/Midterm_Check/UrbanSound8K/audio')

# Random split of 80:20 between training and validation
num_items = len(myds)
num_train = round(num_items * 0.8)
num_val = num_items - num_train
train_ds, val_ds = random_split(myds, [num_train, num_val])

# Create training and validation data loaders
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)
val_dl = torch.utils.data.DataLoader(val_ds, batch_size=16, shuffle=False)

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


# ----------------------------
# Audio Classification Model
# ----------------------------
class AudioClassifier (nn.Module):
    # ----------------------------
    # Build the model architecture
    # ----------------------------
    def __init__(self):
        super().__init__()
        conv_layers = []

        # First Convolution Block with Relu and Batch Norm. Use Kaiming Initialization
        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]
```

```
        # Second Convolution Block
        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]

        # Second Convolution Block
        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]

        # Second Convolution Block
        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
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=11)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # ----------------------------
    # Forward pass computations
    # ----------------------------
    def forward(self, x):
        # Run the convolutional blocks
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear 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

PATH = './Saved_Model/first_model.pth'

myModel = AudioClassifier()
myModel.load_state_dict(torch.load(PATH))
myModel.eval()
# ----------------------------
# Training Loop
# ----------------------------
def training(model, train_dl, num_epochs):
```

```
    # Loss Function, Optimizer and Scheduler
    criterion = 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=num_epochs,
                                                    anneal_strategy='linear')

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

      # Repeat for each batch in the training set
      for i, data in enumerate(train_dl):
          # Get the input features and target labels, and put them on the GPU
          inputs, labels = data[0].to(device), data[1].to(device)

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

          # Zero the parameter gradients
          optimizer.zero_grad()

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

          # Keep stats for Loss and Accuracy
          running_loss += loss.item()

          # Get the predicted class with the highest score
          _, prediction = torch.max(outputs,1)
          # Count of predictions that matched the target label
          correct_prediction += (prediction == labels).sum().item()
          total_prediction += prediction.shape[0]

          #if i % 10 == 0:    # print every 10 mini-batches
          #    print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 10))

      # Print stats at the end of the epoch
      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')

num_epochs = 1   # Just for demo, adjust this higher.
# training(myModel, train_dl, num_epochs)

# ----------------------------
# Inference
# ----------------------------
def inference (model, val_dl):
  correct_prediction = 0
  total_prediction = 0

  # Disable gradient updates
  with torch.no_grad():
```

```
    for data in val_dl:
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)
      # Normalize the inputs
      inputs_m, inputs_s = inputs.mean(), inputs.std()
      inputs = (inputs - inputs_m) / inputs_s
      # Get predictions
      outputs = model(inputs)

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]
    print(inputs[1], labels[1])

  acc = correct_prediction/total_prediction
  print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')



# Run inference on trained model with the validation set
# inference(myModel, val_dl)

''''
def confusion_mat1 (model, val_dl):
    # Disable gradient updates
    with torch.no_grad():
        for data in val_dl:
            # Get the input features and target labels, and put them on the GPU
            inputs, labels = data[0].to(device), data[1].to(device)
            # Normalize the inputs
            inputs_m, inputs_s = inputs.mean(), inputs.std()
            inputs = (inputs - inputs_m) / inputs_s
            # Get predictions
            outputs = model(inputs)

            # Get the predicted class with the highest score
            _, prediction = torch.max(outputs,1)
            y_pred.extend(prediction) # Save Prediction
            y_true.extend(labels) # Save Truth
            # Count of predictions that matched the target label
        return y_pred, y_true

classes = ('1', '2', '3', '4', '5',
        '6', '7', '8', '9', '10', '11')

y_pred, y_true = confusion_mat1(myModel, val_dl)

cf_matrix = confusion_matrix(y_true, y_pred)
df_cm = pd.DataFrame(cf_matrix/np.sum(cf_matrix) *10, index = [i for i in classes],
                     columns = [i for i in classes])
plt.figure(figsize = (12,7))
sn.heatmap(df_cm, annot=True)
plt.show()


# iterate over test data
for inputs, labels in val_dl:
        output = myModel(inputs) # Feed Network

        output = (torch.max(torch.exp(output), 1)[1]).data.cpu().numpy()
        y_pred.extend(output) # Save Prediction

        labels = labels.data.cpu().numpy()
```

```
        y_true.extend(labels) # Save Truth

classes = ('1', '2', '3', '4', '5',
        '6', '7', '8', '9', '10', '11')

cf_matrix = confusion_matrix(y_true, y_pred)
df_cm = pd.DataFrame(cf_matrix/np.sum(cf_matrix) *10, index = [i for i in classes],
                     columns = [i for i in classes])
plt.figure(figsize = (12,7))
sn.heatmap(df_cm, annot=True)
plt.show()

'''

def confusion_mat1 (model, val_dl):
    # Disable gradient updates
    with torch.no_grad():
        for data in val_dl:
            # Get the input features and target labels, and put them on the GPU
            inputs, labels = data[0].to(device), data[1].to(device)
            # Normalize the inputs
            inputs_m, inputs_s = inputs.mean(), inputs.std()
            inputs = (inputs - inputs_m) / inputs_s
            # Get predictions
            outputs = model(inputs)

            # Get the predicted class with the highest score
            _, prediction = torch.max(outputs,1)
            y_pred.extend(prediction) # Save Prediction
            y_true.extend(labels) # Save Truth
            # Count of predictions that matched the target label
        return y_pred, y_true

classes = ('0', '1', '2', '3', '4',
        '5', '6', '7', '8', '9', '10')

y_pred, y_true = confusion_mat1(myModel, val_dl)

def countX(lst, x):
    return lst.count(x)

print('Class {} has occurred {} times in test dataset'.format(10, countX(y_true, 10)))
print('Class {} has been correctly predicted in test {} times'.format(10, countX(y_pred,
10)))

cf_matrix = confusion_matrix(y_true, y_pred)
print(cf_matrix)

df_cm = pd.DataFrame(cf_matrix, index = [i for i in classes], columns = [i for i in classes])
plt.figure(figsize = (12,7))
plt.title('Confusion Matrix of random test dataset')
sn.heatmap(df_cm, annot=True)
plt.show()
```

```
# ----------------------------
# Prepare training data from Metadata file
# ----------------------------
import pandas as pd
from pathlib import Path
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np



download_path = Path.cwd()/'UrbanSound8K'

# Read metadata file
metadata_file = '/Users/pawanvijayanagar/Documents/ASU Masters/CSE598 - Machine Learning
Security and Fairness/Midterm_Check/UrbanSound8K/metadata/UrbanSound8K.csv'
df = pd.read_csv(metadata_file)
df.head()

# Construct file path by concatenating fold and file name
df['relative_path'] = '/fold' + df['fold'].astype(str) + '/' +
df['slice_file_name'].astype(str)

# Take relevant columns
df = df[['relative_path', 'classID']]
df.head()
df = pd.read_csv('dataset_location.csv', usecols= ['relative_path', 'classID'])
df['classID'] = df['classID'].astype(int)

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

class AudioUtil():
  # ----------------------------
  # Load an audio file. Return the signal as a tensor and the sample rate
  # ----------------------------
  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)

  # ----------------------------
  # 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

    if (new_channel == 1):
      # Convert from stereo to mono by selecting only the first channel
      resig = sig[:1, :]
    else:
      # Convert from mono to stereo by duplicating the first channel
      resig = torch.cat([sig, sig])
```

```
    return ((resig, sr))


  # ----------------------------
  # Since Resample applies to a single channel, we resample one channel at a time
  # ----------------------------
  @staticmethod
  def resample(aud, newsr):
    sig, sr = aud

    if (sr == newsr):
      # Nothing to do
      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))


  # ----------------------------
  # Pad (or truncate) the signal to a fixed length 'max_ms' in milliseconds
  # ----------------------------
  @staticmethod
  def pad_trunc(aud, max_ms):
    sig, sr = aud
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms

    if (sig_len > max_len):
      # Truncate the signal to the given length
      sig = sig[:,:max_len]

    elif (sig_len < max_len):
      # Length of padding to add at the beginning and end of the signal
      pad_begin_len = random.randint(0, max_len - sig_len)
      pad_end_len = max_len - sig_len - pad_begin_len

      # Pad with 0s
      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)

  # ----------------------------
  # Shifts the signal to the left or right by some percent. Values at the end
  # are 'wrapped around' to the start of the transformed signal.
  # ----------------------------
  @staticmethod
  def time_shift(aud, shift_limit):
    sig,sr = aud
    _, sig_len = sig.shape
    shift_amt = int(random.random() * shift_limit * sig_len)
    return (sig.roll(shift_amt), sr)


  # ----------------------------
  # Generate a Spectrogram
```

```
  # ----------------------------
  @staticmethod
  def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None):
    sig,sr = aud
    top_db = 80

    # spec has shape [channel, n_mels, time], where channel is mono, stereo etc
    spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

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


  # ----------------------------
  # Augment the Spectrogram by masking out some sections of it in both the frequency
  # dimension (ie. horizontal bars) and the time dimension (vertical bars) to prevent
  # overfitting and to help the model generalise better. The masked sections are
  # replaced with the mean value.
  # ----------------------------
  @staticmethod
  def spectro_augment(spec, max_mask_pct=0.1, n_freq_masks=1, n_time_masks=1):
    _, n_mels, n_steps = spec.shape
    mask_value = spec.mean()
    aug_spec = spec

    freq_mask_param = max_mask_pct * n_mels
    for _ in range(n_freq_masks):
      aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, mask_value)

    time_mask_param = max_mask_pct * n_steps
    for _ in range(n_time_masks):
      aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_value)

    return aug_spec



from torch.utils.data import DataLoader, Dataset, random_split
import torchaudio

# ----------------------------
# 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 i'th item in dataset
  # ----------------------------
  def __getitem__(self, idx):
    # Absolute file path of the audio file - concatenate the audio directory with
    # the relative path
```

```
    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']

    aud = AudioUtil.open(audio_file)
    # Some sounds have a higher sample rate, or fewer channels compared to the
    # majority. So make all sounds have the same number of channels and same
    # sample rate. Unless the sample rate is the same, the pad_trunc will still
    # result in arrays of different lengths, even though the sound duration is
    # the same.
    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


from torch.utils.data import random_split

myds = SoundDS(df, data_path = '/Users/pawanvijayanagar/Documents/ASU Masters/CSE598 -
Machine Learning Security and Fairness/Midterm_Check/UrbanSound8K/audio')

# Random split of 80:20 between training and validation
num_items = len(myds)
num_train = round(num_items * 0.8)
num_val = num_items - num_train
train_ds, val_ds = random_split(myds, [num_train, num_val])

# Create training and validation data loaders
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)
val_dl = torch.utils.data.DataLoader(val_ds, batch_size=16, shuffle=False)

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


# ----------------------------
# Audio Classification Model
# ----------------------------
class AudioClassifier (nn.Module):
    # ----------------------------
    # Build the model architecture
    # ----------------------------
    def __init__(self):
        super().__init__()
        conv_layers = []

        # First Convolution Block with Relu and Batch Norm. Use Kaiming Initialization
        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]

        # Second Convolution Block
        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]

        # Second Convolution Block
        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]

        # Second Convolution Block
        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
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=11)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # ----------------------------
    # Forward pass computations
    # ----------------------------
    def forward(self, x):
        # Run the convolutional blocks
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear 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


# ----------------------------
# Training Loop
# ----------------------------
def training(model, train_dl, num_epochs):
  # Loss Function, Optimizer and Scheduler
  criterion = 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=num_epochs,
                                                  anneal_strategy='linear')

  # Repeat for each epoch
```

```
loss_graph = []
epoch_graph = []
accuracy_graph = []


for epoch in range(num_epochs):
  running_loss = 0.0
  correct_prediction = 0
  total_prediction = 0
  epoch_graph.append(epoch)

  # Repeat for each batch in the training set
  for i, data in enumerate(train_dl):
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)

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

      # Zero the parameter gradients
      optimizer.zero_grad()

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

      # Keep stats for Loss and Accuracy
      running_loss += loss.item()

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

      #if i % 10 == 0:    # print every 10 mini-batches
      #    print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 10))

  # Print stats at the end of the epoch
  num_batches = len(train_dl)
  avg_loss = running_loss / num_batches
  loss_graph.append(avg_loss)
  acc = correct_prediction/total_prediction
  accuracy_graph.append(acc)
  print(f'Epoch: {epoch}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')

plt.plot(epoch_graph, loss_graph, label = "Loss Curve")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs Epoch')
plt.legend()
plt.show()

plt.plot(epoch_graph, accuracy_graph, label = "Accuracy Curve")
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Epoch')
plt.legend()
plt.show()

print('Finished Training')
```

```
num_epochs = 25   # Just for demo, adjust this higher.
training(myModel, train_dl, num_epochs)

# ----------------------------
# Inference
# ----------------------------
def inference (model, val_dl):
  correct_prediction = 0
  total_prediction = 0

  # Disable gradient updates
  with torch.no_grad():
    for data in val_dl:
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)

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

      # Get predictions
      outputs = model(inputs)

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

  acc = correct_prediction/total_prediction
  print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')

# Run inference on trained model with the validation set
inference(myModel, val_dl)

PATH = './Saved_Model/first_model.pth'
torch.save(myModel.state_dict(), PATH)
```