1 Author

Student Name: Dnyanesh Walwadkar Advance Solution Notebook

2 Problem formulation

Building Audio Deep Learning Model for forcasting melody name.

Sound Classification is one of the foremost broadly utilized applications in Audio Deep Learning. It includes learning to classify sounds and to anticipate the category of that sound. I am going begin with sound files, convert them into spectrograms, input them into a CNN plus Linear Classifier model, and create forecasts almost the lesson to which the melody has a place.

I started with sound *.wav files, converted them into spectrograms, input them into a CNN plus Linear Classifier model, and produced predictions about the class (Song) to which the sound belongs.

I Trained model for 119 Epoch and Got following results:

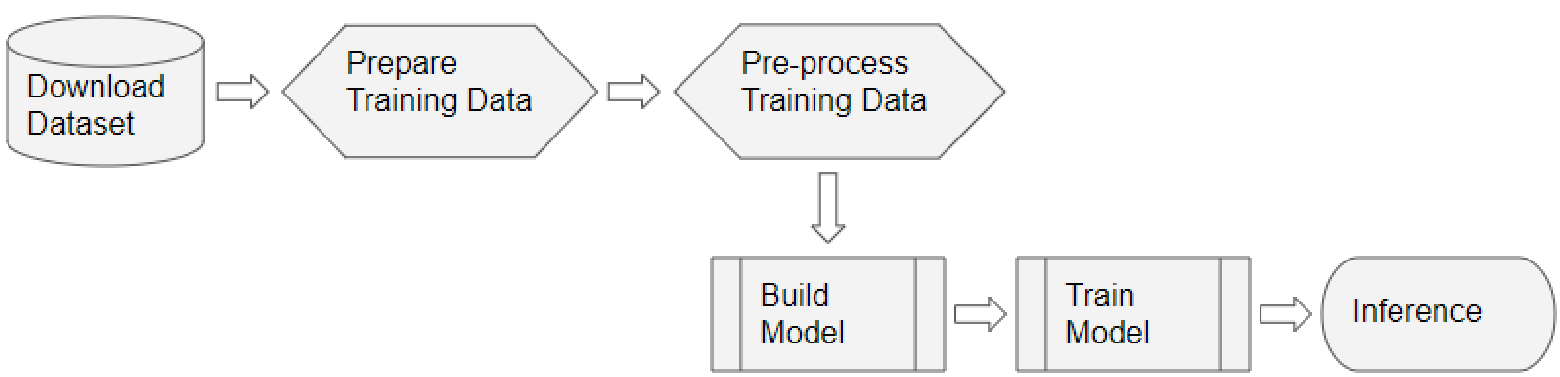
Epoch: 200, Loss: 0.14, Traning Accuracy: (97% - 98%)

Validation Accuracy : (52% - 70%)

(I included range of accuracies beacuse as traning dataset get different number of audio files from different songs, model get affected. If we increse number of audio files per song, this model will get improved.)

We can improve Validation accuracy, for that we have to consider large number of audio files per songClass in our traning dataset.

3 Machine Learning pipeline



Import Dataset from Google Drive

Dataset Contains 8 different types of Songs Audio files. Total 98. We are preprocessing given Dataset to build our required or desired dataset as per follows :

The training data for this problem are classified as:

- The features (X) are the audio file paths.
- The target labels (y) are the class names.

song = "Potter" Class = 1

song = "StarWars" Class = 2

song = "Frozen" Class = 3

song="Panther" Class = 4

song = 'Rain' Class = 5

song = "Showman" Class = 6

song = "Mamma" Class = 7

song = "Hakuna" Class = 8

4 Transformation stage

Read audio from a file

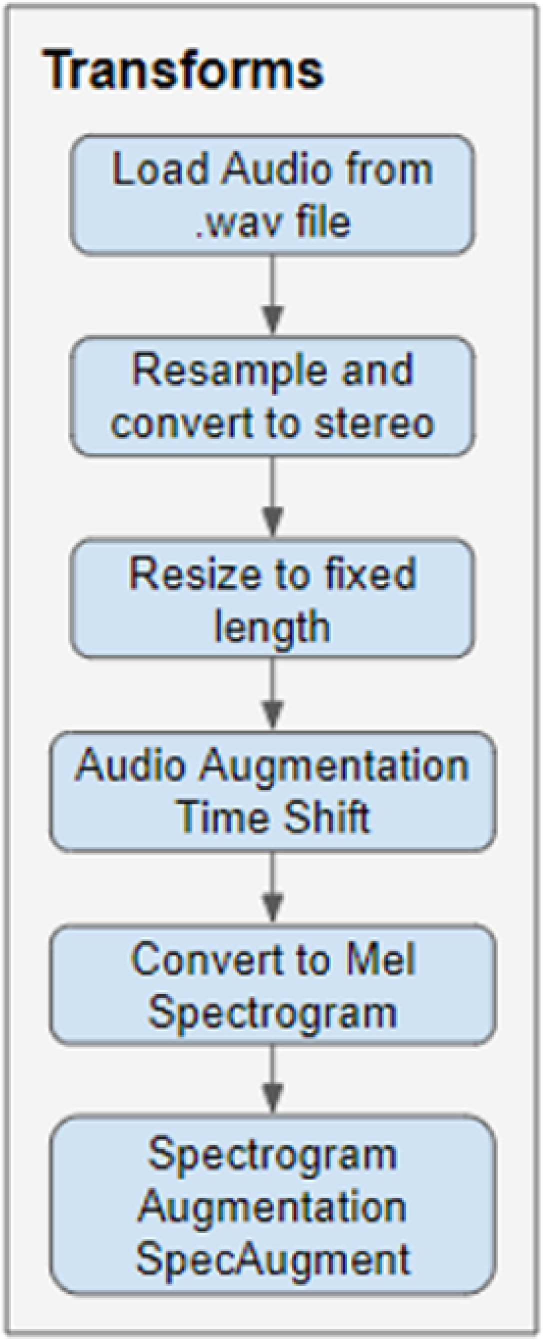
The first thing we need is to read and load the audio file in ".wav" format as per our project specification. Since we are using Pytorch for this example, the implementation below uses torchaudio for the audio processing, but librosa will work just as well. Librosa use we saw in earlier solution.

In this code we are taking care of Mono & Stereo Sounds.

The difference between monophonic (mono) and stereophonic (stereo) sound is the number of channels used to record and playback audio. Mono signals are recorded and played back using a single audio channel, while stereo sounds are recorded and played back using two audio channels. As a listener, the most noticeable difference is that stereo sounds are capable of producing the perception of width, whereas mono sounds are not.

We are using following libraries for this task.

- torch
- torchaudio - Torchaudio is a library for audio and signal processing with PyTorch. It provides I/O, signal and data processing functions, datasets, model implementations and application components
- Ipython.display.Audio




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

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

  def rechannel(aud, new_channel):
```

```
  sig, sr = aud

  if (sig.shape[0] == new_channel):

    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))

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))

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)

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)

  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)

  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
```

Custom Data Loader

Now that we have defined all the pre-processing transform functions we will define a **custom Pytorch Dataset object.** To feed your data to a model with Pytorch, we need two objects:

- A custom Dataset object that uses all the audio transforms to pre-process an audio file and prepares one data item at a time.
- A built-in DataLoader object that uses the Dataset object to fetch individual data items and packages them into a batch of data.

Demo Steps for Converting Audio Signals to Spectrogram

```
# ----------------------------
# Sound Dataset
# ----------------------------

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


class SoundDS(Dataset):
```

```
  def __init__(self, df, song_path):
    self.df = df
    self.data_path = str(song_path)
    self.duration = 4000
    self.sr =  11000
    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
    song_Audio = self.data_path + str(self.df.loc[idx,'file_id'])
    # Get the Song ID
    class_id = int(self.df.loc[idx, 'song'])
    print(AudioUtil.open(song_Audio))

    song = AudioUtil.open(song_Audio)

    reaud = AudioUtil.resample(song, self.sr)
    rechan = AudioUtil.rechannel(reaud, self.channel)

    dur_song = AudioUtil.pad_trunc(rechan, self.duration)
    shift_song = AudioUtil.time_shift(dur_song, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_song, 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
```

Prepared Batches of Data with the Data Loader for traning & Validation

All of the functions we need to input our data to the model have now been defined.

- We use our custom Dataset to load the Features and Labels from our Pandas dataframe and split that data randomly in an 90:10 ratio into training and validation sets.
- We then use them to create our training and validation Data Loaders.

```
sample_path1 = '/content/drive/MyDrive/Data/MLEndHW/sample/MLEndHW_Sample/'

```

```
#Run this cell after running Dataset Cell
#----------------------------------------------------------------------

```

```
from torch.utils.data import random_split
from torch.utils.data import DataLoader

#MLENDHW_df = PandasDataset(MLENDHW_df)
myds = SoundDS(MLENDHW_df, sample_path1)

# Random split of 90:10 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])
#train_ds = PandasDataset(train_ds)
#val_ds = PandasDataset(val_ds)

# numTD = round(num_items * 0.6)
# numD = num_items - numTD
# traD, vaD = random_split(myds, [numTD, numD])

# Create training and validation data loaders
train_dl = DataLoader(train_ds, batch_size=4)
val_dl = DataLoader(val_ds, batch_size=4)
```

5 Modelling

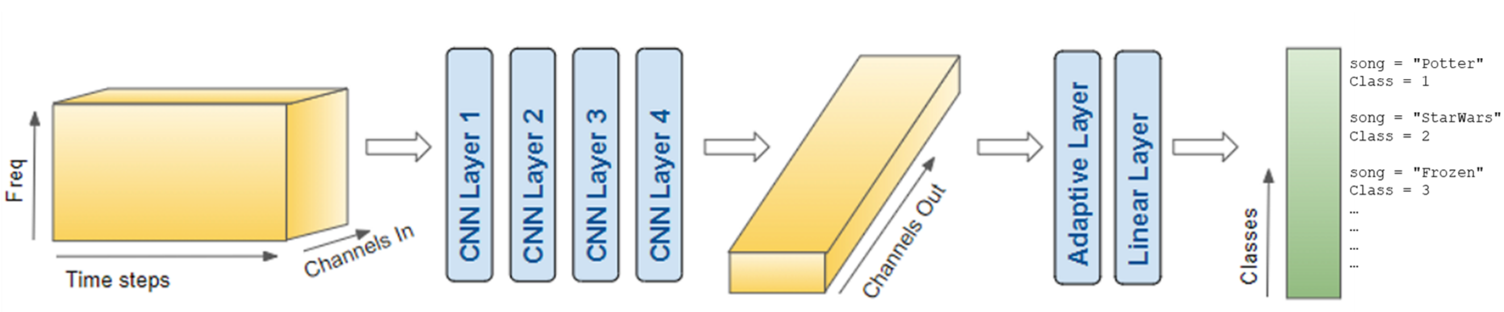
Model

Since our data now consists of Spectrogram images, we build a CNN classification architecture to process them. It has four convolutional blocks which generate the feature maps. That data is then reshaped into the format we need so it can be input into the linear classifier layer, which finally outputs the predictions for the 8 song classes.

Spectrogram ▶

A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time. When applied to an audio signal, spectrograms are sometimes called sonographs, voiceprints, or voicegrams.

Spectrograms are used extensively in the fields of music speech processing,seismology, and others. Spectrograms of audio can be used to identify spoken words phonetically, and to analyse the various audio patterns



```
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=10)

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

    # ----------------------------
    # Forward pass computations
```

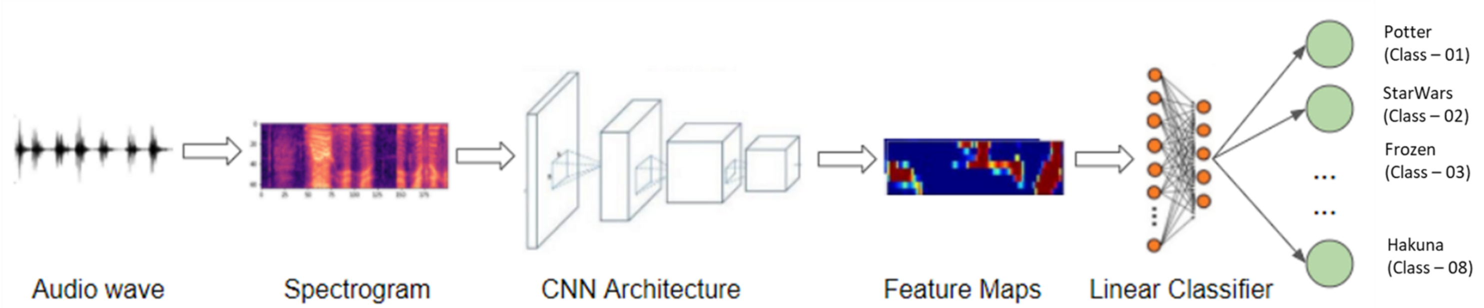
```
    # ----------------------------
    def forward(self, x):
        # Run the convolutional blocks
        x = self.conv(x)

        # 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
```

```
device(type='cpu')
```

6 Methodology

We characterize the capacities for the optimizer, loss, and scheduler to powerfully change our learning rate as preparing advances, which ordinarily permits preparing to focalize in less epochs.

We prepare the demonstrate for a several epochs, handling a clump of information in each emphasis. We keep track of a simple accuracy metric which measures the percentage of correct forcast.



7 Dataset

```


sample_path = '/content/drive/MyDrive/Data/MLEndHW/sample/MLEndHW_Sample/*.wav'
files = glob.glob(sample_path)
print(files)
```

```
['/content/drive/MyDrive/Data/MLEndHW/sample/MLEndHW_Sample/S3_whistle_1_Panther.wav', '/content/dri
```

```
for _ in range(5):
  n = np.random.randint(98)
  display(ipd.Audio(files[n]))
```

0:00 / 0:21

0:00 / 0:15

0:00 / 0:15

0:00 / 0:15

0:00 / 0:14

Preprocessing

Replace Song Name Colunm as Distinct Numric Class Value. We will use This Classes as our Target value during Model Building.

This training data with audio file paths cannot be input directly into the model. We have to load the audio data from the file and process it so that it is in a format that the model expects.

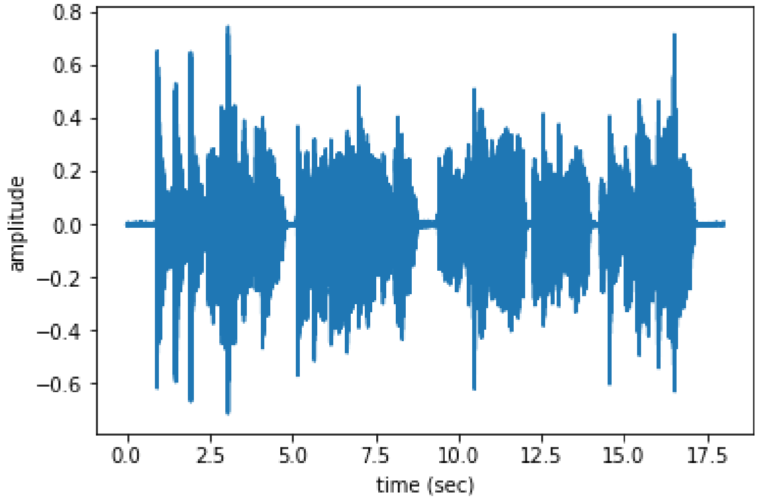
```
MLENDHW_table = []
flag=0

for file in files:
  file_name = file.split('/')[-1]
  flag=0
  try:
    song = file.split('/')[-1].split('_')[3].split('.')[0]
    if(song=="Potter"):
      song = 1
    elif(song=="StarWars"):
      song=2
    elif(song=="Frozen"):
      song=3
    elif(song=="Panther"):
      song=4
    elif(song=='Rain'):
      song=5
    elif(song=="Showman"):
      song=6
    elif(song=="Mamma"):
```

```
      song=7
    elif(song=="Hakuna"):
      song=8
    else:
      flag=1
    if flag==0:
      MLENDHW_table.append([file_name, song])
  except:
    pass



MLENDHW_table
```

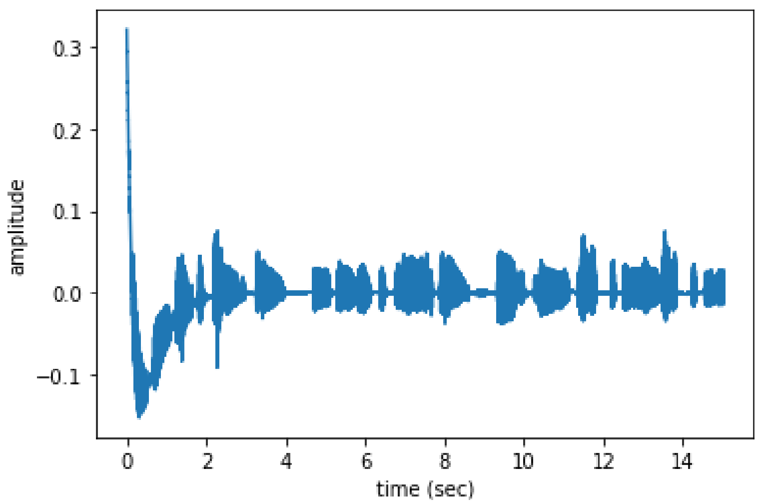
```
MLENDHW_df = pd.DataFrame(MLENDHW_table,columns=['file_id','song'])
MLENDHW_df
MLENDHW_df.groupby('song').count()
```

	file_id
song	
1	34
2	31
3	32
4	34
5	34
6	33
7	29
8	34

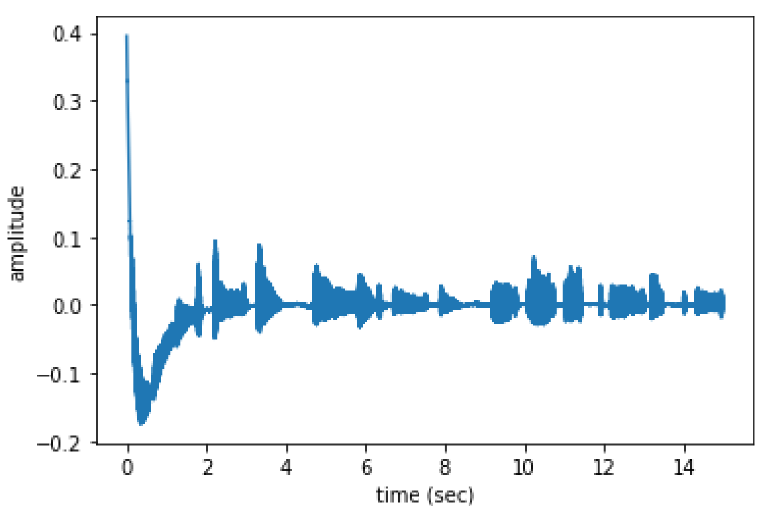
```
n=0
fs = None # Sampling frequency. If None, fs would be 22050
for i in range(5):

    x, fs = librosa.load(files[i],sr=fs)
    t = np.arange(len(x))/fs
    plt.plot(t,x)
    plt.xlabel('time (sec)')
    plt.ylabel('amplitude')
    plt.show()
    display(ipd.Audio(files[n]))
```



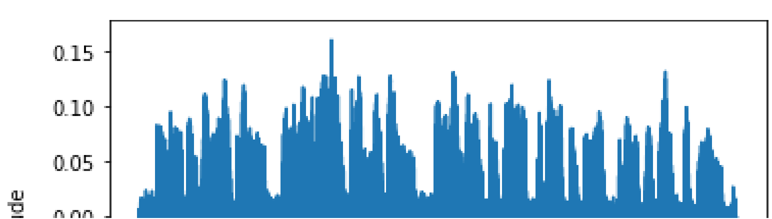
0:00 / 0:17

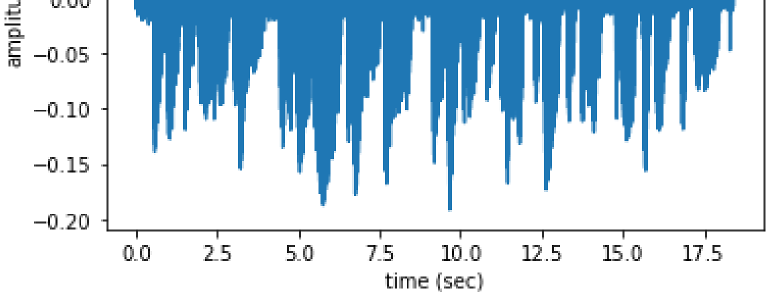


0:00 / 0:17

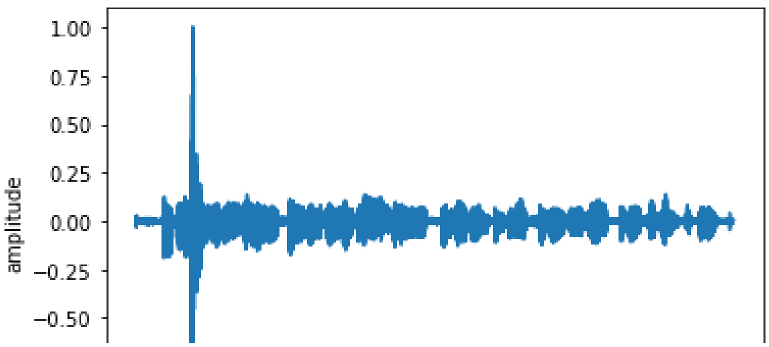


0:00 / 0:17





0:00 / 0:17



8 Results

Traning & Validation

0:00 / 0:17

```
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
    first = 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], data[1]
        # 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)
        labels = tuple_of_tensors_to_tensor(labels)

        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=200  # Just for demo, adjust this higher.
training(myModel, train_dl, num_epochs)
```

Validation Part

We run inference loop taking care to disable the gradient updates. The forward pass is performed with the model to get predictions, but we don't need to backpropagate or run the optimizer.

```
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], data[1]

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

      # Get predictions
```

```
        # Get predictions
        outputs = model(inputs)
        print("Outputs-------------------")
        print(outputs)

        # 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}')

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

9 Conclusions

As Training accuracy is higher than Validation accuracy, there are many opportunities for improvement.

This is a end-to-end case of song classification which is one of the foremost foundational issues in sound Deep learning. Not as it were is this utilized in a wide run of applications, but numerous of the concepts and procedures that we secured here will be pertinent to more complicated sound issues.

I Trained model for 200 Epoch and Got following results:

Epoch: 119, Loss: 0.14, Traning Accuracy: (97% - 98%)

Validation Accuracy : (52% - 70%)

(I included range of accuracies beacuse as traning dataset got different number of audio files from different songs, model get affected. If we increse number of audio files per song, this model will get improved.)

Scope for improvement :

- Dataset Size can be increased to cover more features.
- Deep Learning Network can be Improved. Adding more CNN layes can be one option.
- Focus on selecting desirable time durations from audio files can improve feature selection.
- Model building on single type of audio can be one option. (Selecting only Whistling or Humming audios.