Lab 11: Speech Command Classification With Torchaudio

Objective

The objective of this notebook is to implement a speech command classification model using PyTorch and Torchaudio. The task involves processing audio datasets, building neural networks, and evaluating the model's performance.

Data Dictionary

- Download dataset using from torchaudio.datasets import SPEECHCOMMANDS

Since this notebook works on audio data, the data dictionary includes:

- Audio Clips: Short audio files containing speech commands.
- File Name: Unique identifier for each audio file.
- Command Label: The word spoken in the audio clip (e.g., "yes," "no," "up").
- Sampling Rate: Number of samples per second in the audio data.
- Waveform: The raw audio signal in time-domain format.
- Mel-Spectrogram: Time-frequency representation of audio data.
- MFCCs: Mel-Frequency Cepstral Coefficients for feature extraction.

Task

1. Environment Setup

- Install the required libraries including librosa, pytorch, and other audio processing dependencies, ensuring all version compatibilities are maintained Verify GPU availability and configure CUDA settings for optimal training performance if applicable

2. Data Preprocessing

- Import and analyze the structure of the audio dataset, checking for consistency in sample rates and durations Transform the raw audio files into spectral features by applying Mel-Spectrogram and MFCC calculations, with appropriate parameters for the specific use case

3. Exploratory Data Analysis (EDA)

- Generate visual representations of audio waveforms across different classes to understand signal characteristics Conduct auditory analysis of sample files to identify potential patterns and anomalies in the dataset

4. Model Building

- Design a neural network architecture suitable for audio classification, considering both CNN and RNN components Configure the training framework with appropriate optimizer settings and a loss function aligned with the task objectives

5. Training

- Execute the training pipeline with proper batch sizes and learning rate scheduling Implement logging and checkpointing to track model progression and save intermediate states

6. Evaluation

- Assess model performance on a held-out test set using industry-standard metrics Generate and analyze confusion matrices to identify patterns in classification errors

```
%matplotlib inline
```

```
# Uncomment the line corresponding to your "runtime type" to run in Google Colab

# CPU:
# !pip install pydub torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f https://download.pytorch.org/whl/torch_st

# GPU:
!pip install pydub torch==1.7.0+cu101 torchvision==0.8.1+cu101 torchaudio==0.7.0 -f https://download.pytorch.org/whl/torch_
```

```
Looking in links: https://download.pytorch.org/whl/torch_stable.html
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl.metadata (1.4 kB)
ERROR: Could not find a version that satisfies the requirement torch==1.7.0+cu101 (from versions: 1.11.0, 1.11.0+cpu, 1
ERROR: No matching distribution found for torch==1.7.0+cu101
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchaudio
import sys

import matplotlib.pyplot as plt
import IPython.display as ipd

from tqdm import tqdm
```

Let's check if a CUDA GPU is available and select our device. Running the network on a GPU will greatly decrease the training/testing runtime.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

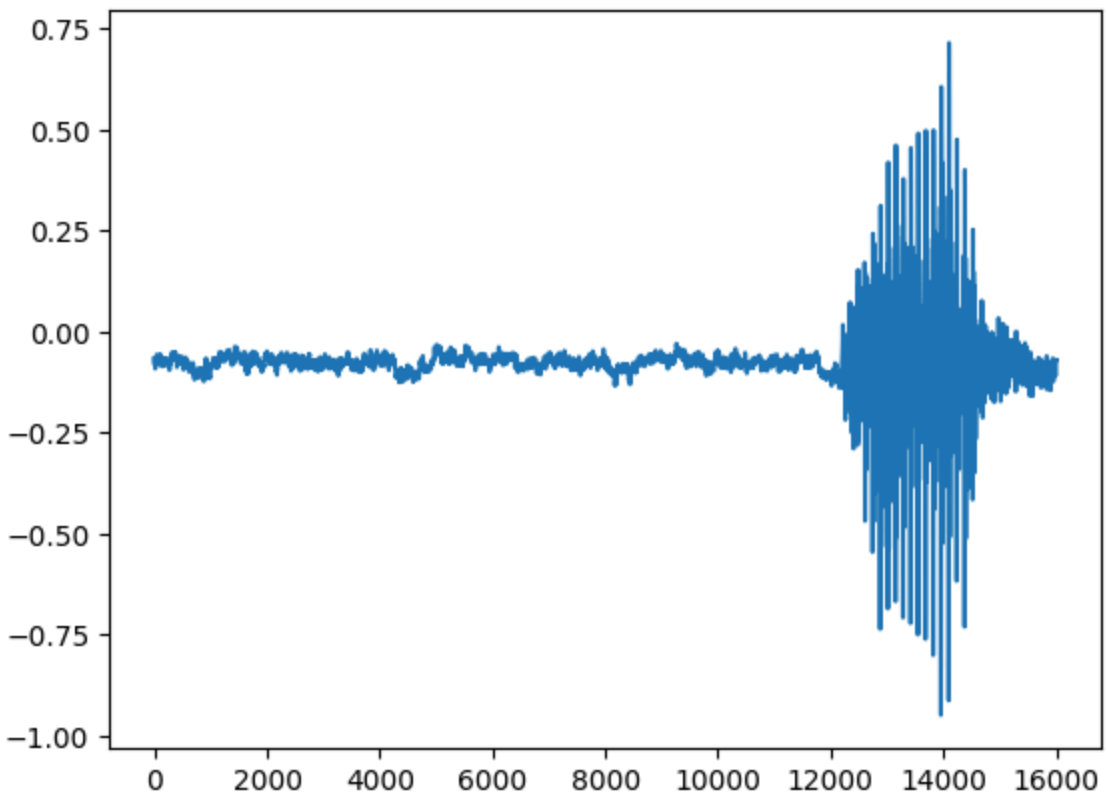
```
from torchaudio.datasets import SPEECHCOMMANDS
import os


class SubsetSC(SPEECHCOMMANDS):
    def __init__(self, subset: str = None):
        super().__init__("./", download=True)

        def load_list(filename):
            filepath = os.path.join(self._path, filename)
            with open(filepath) as fileobj:
                return [os.path.normpath(os.path.join(self._path, line.strip())) for line in fileobj]

        if subset == "validation":
            self._walker = load_list("validation_list.txt")
        elif subset == "testing":
            self._walker = load_list("testing_list.txt")
        elif subset == "training":
            excludes = load_list("validation_list.txt") + load_list("testing_list.txt")
            excludes = set(excludes)
            self._walker = [w for w in self._walker if w not in excludes]


# Create training and testing split of the data. We do not use validation in this tutorial.
train_set = SubsetSC("training")
test_set = SubsetSC("testing")

waveform, sample_rate, label, speaker_id, utterance_number = train_set[0]
```

```
100%|██████████| 2.26G/2.26G [00:15<00:00, 155MB/s]
```

A data point in the SPEECHCOMMANDS dataset is a tuple made of a waveform (the audio signal), the sample rate, the utterance (label), the ID of the speaker, the number of the utterance.

```
print("Shape of waveform: {}".format(waveform.size()))
print("Sample rate of waveform: {}".format(sample_rate))

plt.plot(waveform.t().numpy());
```

```
Shape of waveform: torch.Size([1, 16000])
Sample rate of waveform: 16000
```



Let’s find the list of labels available in the dataset.

```
labels = sorted(list(set(datapoint[2] for datapoint in train_set)))
labels
```

```
['backward',
 'bed',
 'bird',
 'cat',
 'dog',
 'down',
 'eight',
 'five',
 'follow',
 'forward',
 'four',
 'go',
 'happy',
 'house',
 'learn',
 'left',
 'marvin',
 'nine',
 'no',
 'off',
 'on',
 'one',
 'right',
 'seven',
 'sheila',
 'six',
 'stop',
 'three',
 'tree',
 'two',
 'up',
 'visual',
 'wow',
 'yes',
 'zero']
```

The 35 audio labels are commands that are said by users. The first few files are people saying “marvin”.

```
waveform_first, *_ = train_set[0]
ipd.Audio(waveform_first.numpy(), rate=sample_rate)

waveform_second, *_ = train_set[1]
ipd.Audio(waveform_second.numpy(), rate=sample_rate)
```

0:01 / 0:01

The last file is someone saying “visual”.

```
waveform_last, *_ = train_set[-1]
ipd.Audio(waveform_last.numpy(), rate=sample_rate)
```

0:00 / 0:01

Formatting the Data

This is a good place to apply transformations to the data. For the waveform, we downsample the audio for faster processing without losing too much of the classification power.

We don't need to apply other transformations here. It is common for some datasets though to have to reduce the number of channels (say from stereo to mono) by either taking the mean along the channel dimension, or simply keeping only one of the channels. Since SpeechCommands uses a single channel for audio, this is not needed here.

```
new_sample_rate = 8000
transform = torchaudio.transforms.Resample(orig_freq=sample_rate, new_freq=new_sample_rate)
transformed = transform(waveform)

ipd.Audio(transformed.numpy(), rate=new_sample_rate)
```

0:00 / 0:01

We are encoding each word using its index in the list of labels.

```
def label_to_index(word):
    # Return the position of the word in labels
    return torch.tensor(labels.index(word))


def index_to_label(index):
    # Return the word corresponding to the index in labels
    # This is the inverse of label_to_index
    return labels[index]


word_start = "yes"
index = label_to_index(word_start)
word_recovered = index_to_label(index)

print(word_start, "-->", index, "-->", word_recovered)
```

```
yes --> tensor(33) --> yes
```

```
def pad_sequence(batch):
    # Make all tensor in a batch the same length by padding with zeros
    batch = [item.t() for item in batch]
    batch = torch.nn.utils.rnn.pad_sequence(batch, batch_first=True, padding_value=0.)
    return batch.permute(0, 2, 1)


def collate_fn(batch):

    # A data tuple has the form:
    # waveform, sample_rate, label, speaker_id, utterance_number

    tensors, targets = [], []

    # Gather in lists, and encode labels as indices
    for waveform, _, label, *_ in batch:
        tensors += [waveform]
        targets += [label_to_index(label)]

    # Group the list of tensors into a batched tensor
    tensors = pad_sequence(tensors)
    targets = torch.stack(targets)

    return tensors, targets


batch_size = 256

if device == "cuda":
```

```
    num_workers = 1
    pin_memory = True
else:
    num_workers = 0
    pin_memory = False

train_loader = torch.utils.data.DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    collate_fn=collate_fn,
    num_workers=num_workers,
    pin_memory=pin_memory,
)
test_loader = torch.utils.data.DataLoader(
    test_set,
    batch_size=batch_size,
    shuffle=False,
    drop_last=False,
    collate_fn=collate_fn,
    num_workers=num_workers,
    pin_memory=pin_memory,
)
```

```
class M5(nn.Module):
    def __init__(self, n_input=1, n_output=35, stride=16, n_channel=32):
        super().__init__()
        self.conv1 = nn.Conv1d(n_input, n_channel, kernel_size=80, stride=stride)
        self.bn1 = nn.BatchNorm1d(n_channel)
        self.pool1 = nn.MaxPool1d(4)
        self.conv2 = nn.Conv1d(n_channel, n_channel, kernel_size=3)
        self.bn2 = nn.BatchNorm1d(n_channel)
        self.pool2 = nn.MaxPool1d(4)
        self.conv3 = nn.Conv1d(n_channel, 2 * n_channel, kernel_size=3)
        self.bn3 = nn.BatchNorm1d(2 * n_channel)
        self.pool3 = nn.MaxPool1d(4)
        self.conv4 = nn.Conv1d(2 * n_channel, 2 * n_channel, kernel_size=3)
        self.bn4 = nn.BatchNorm1d(2 * n_channel)
        self.pool4 = nn.MaxPool1d(4)
        self.fc1 = nn.Linear(2 * n_channel, n_output)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = self.pool2(x)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = self.pool3(x)
        x = self.conv4(x)
        x = F.relu(self.bn4(x))
        x = self.pool4(x)
        x = F.avg_pool1d(x, x.shape[-1])
        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        return F.log_softmax(x, dim=2)


model = M5(n_input=transformed.shape[0], n_output=len(labels))
model.to(device)
print(model)


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


n = count_parameters(model)
print("Number of parameters: %s" % n)
```

```
M5(
  (conv1): Conv1d(1, 32, kernel_size=(80,), stride=(16,))
  (bn1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv1d(32, 32, kernel_size=(3,), stride=(1,))
  (bn2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv1d(32, 64, kernel_size=(3,), stride=(1,))
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
```

```
    (conv4): Conv1d(64, 64, kernel_size=(3,), stride=(1,))
    (bn4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (pool4): MaxPool1d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=64, out_features=35, bias=True)
  )
  Number of parameters: 26915
```

```
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.0001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)  # reduce the learning after 20 epochs by a facto
```

Training and Testing the Network

Now let's define a training function that will feed our training data into the model and perform the backward pass and optimization steps. For training, the loss we will use is the negative log-likelihood. The network will then be tested after each epoch to see how the accuracy varies during the training.

```
def train(model, epoch, log_interval):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):

        data = data.to(device)
        target = target.to(device)

        # apply transform and model on whole batch directly on device
        data = transform(data)
        output = model(data)

        # negative log-likelihood for a tensor of size (batch x 1 x n_output)
        loss = F.nll_loss(output.squeeze(), target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # print training stats
        if batch_idx % log_interval == 0:
            print(f"Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)} ({100. * batch_idx / len(trai

        # update progress bar
        pbar.update(pbar_update)
        # record loss
        losses.append(loss.item())
```

Now that we have a training function, we need to make one for testing the networks accuracy. We will set the model to `eval()` mode and then run inference on the test dataset. Calling `eval()` sets the training variable in all modules in the network to false. Certain layers like batch normalization and dropout layers behave differently during training so this step is crucial for getting correct results.

```
def number_of_correct(pred, target):
    # count number of correct predictions
    return pred.squeeze().eq(target).sum().item()


def get_likely_index(tensor):
    # find most likely label index for each element in the batch
    return tensor.argmax(dim=-1)


def test(model, epoch):
    model.eval()
    correct = 0
    for data, target in test_loader:

        data = data.to(device)
        target = target.to(device)

        # apply transform and model on whole batch directly on device
        data = transform(data)
        output = model(data)

        pred = get_likely_index(output)
        correct += number_of_correct(pred, target)

        # update progress bar
        pbar.update(pbar_update)

    print(f"\nTest Epoch: {epoch}\tAccuracy: {correct}/{len(test_loader.dataset)} ({100. * correct / len(test_loader.datase
```

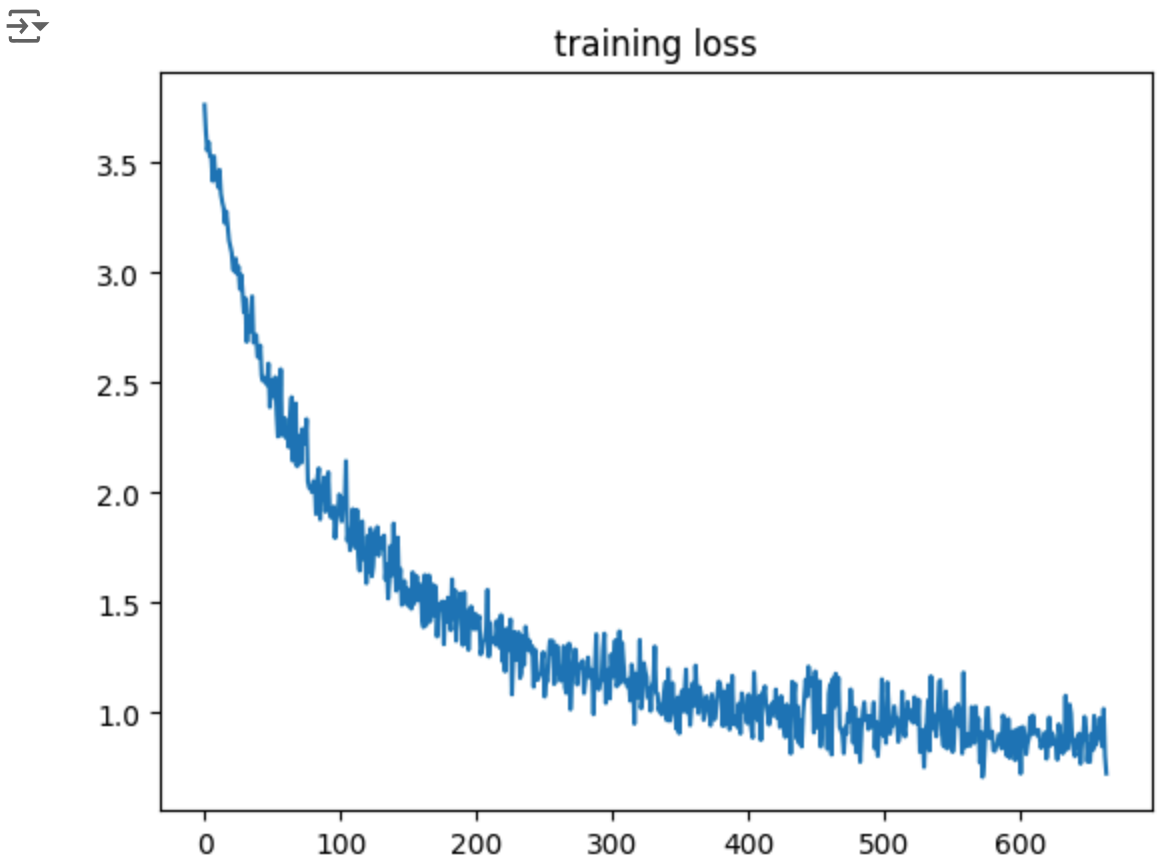
Finally, we can train and test the network. We will train the network for ten epochs then reduce the learn rate and train for ten more epochs. The network will be tested after each epoch to see how the accuracy varies during the training.

```
log_interval = 20
n_epoch = 2

pbar_update = 1 / (len(train_loader) + len(test_loader))
losses = []

# The transform needs to live on the same device as the model and the data.
transform = transform.to(device)
with tqdm(total=n_epoch) as pbar:
    for epoch in range(1, n_epoch + 1):
        train(model, epoch, log_interval)
        test(model, epoch)
        scheduler.step()
```

```
  0%|          | 0.0026666666666666666/2 [00:02<28:38, 860.26s/it]Train Epoch: 1 [0/84843 (0%)] Loss: 3.762943
  3%|▎         | 0.05599999999999999/2 [00:20<10:51, 335.19s/it]Train Epoch: 1 [5120/84843 (6%)]       Loss: 3.090956
  5%|▌         | 0.10933333333333328/2 [00:38<10:04, 319.76s/it]Train Epoch: 1 [10240/84843 (12%)]     Loss: 2.616976
  8%|▊         | 0.16266666666666676/2 [00:57<10:47, 352.27s/it]Train Epoch: 1 [15360/84843 (18%)]     Loss: 2.248950
 11%|█         | 0.21600000000000033/2 [01:14<09:37, 323.87s/it]Train Epoch: 1 [20480/84843 (24%)]     Loss: 2.052880
 13%|█▎        | 0.2693333333333337/2 [01:33<10:57, 379.75s/it] Train Epoch: 1 [25600/84843 (30%)]     Loss: 1.978313
 16%|█▌        | 0.3226666666666667/2 [01:50<09:08, 326.74s/it] Train Epoch: 1 [30720/84843 (36%)]     Loss: 1.804310
 19%|█▉        | 0.3759999999999997/2 [02:09<10:11, 376.55s/it]Train Epoch: 1 [35840/84843 (42%)]     Loss: 1.657334
 21%|██        | 0.42933333333333273/2 [02:27<08:43, 333.01s/it]Train Epoch: 1 [40960/84843 (48%)]     Loss: 1.427591
 24%|██▍       | 0.48266666666666574/2 [02:44<08:02, 318.22s/it]Train Epoch: 1 [46080/84843 (54%)]     Loss: 1.431276
 27%|██▋       | 0.5359999999999995/2 [03:03<08:26, 346.29s/it]Train Epoch: 1 [51200/84843 (60%)]     Loss: 1.382255
 29%|██▉       | 0.5893333333333336/2 [03:21<07:40, 326.10s/it]Train Epoch: 1 [56320/84843 (66%)]     Loss: 1.333533
 32%|███▏      | 0.6426666666666677/2 [03:39<08:28, 374.88s/it]Train Epoch: 1 [61440/84843 (72%)]     Loss: 1.274874
 35%|███▍      | 0.6960000000000018/2 [03:57<07:02, 324.05s/it]Train Epoch: 1 [66560/84843 (78%)]     Loss: 1.207128
 37%|███▋      | 0.749333333333336/2 [04:15<08:04, 387.75s/it] Train Epoch: 1 [71680/84843 (84%)]     Loss: 1.149792
 40%|███▉      | 0.8026666666666701/2 [04:33<06:33, 328.65s/it]Train Epoch: 1 [76800/84843 (90%)]     Loss: 1.163362
 43%|████▎     | 0.8560000000000042/2 [04:51<06:09, 322.76s/it]Train Epoch: 1 [81920/84843 (96%)]     Loss: 1.331745
 50%|█████     | 1.0000000000000062/2 [05:39<05:02, 302.57s/it]
Test Epoch: 1   Accuracy: 6989/11005 (64%)

 50%|█████     | 1.0026666666666728/2 [05:40<05:12, 312.84s/it]Train Epoch: 2 [0/84843 (0%)]    Loss: 1.090464
 53%|█████▎    | 1.0560000000000047/2 [05:59<05:54, 375.10s/it]Train Epoch: 2 [5120/84843 (6%)]     Loss: 1.007195
 55%|█████▍    | 1.1093333333333366/2 [06:16<04:50, 325.70s/it]Train Epoch: 2 [10240/84843 (12%)]     Loss: 0.943283
 58%|█████▊    | 1.1626666666666685/2 [06:35<05:23, 386.56s/it]Train Epoch: 2 [15360/84843 (18%)]     Loss: 0.933667
 61%|██████    | 1.2160000000000004/2 [06:53<04:19, 331.41s/it]Train Epoch: 2 [20480/84843 (24%)]     Loss: 1.120875
 63%|██████▎   | 1.2693333333333323/2 [07:10<03:51, 317.43s/it]Train Epoch: 2 [25600/84843 (30%)]     Loss: 1.141625
 66%|██████▌   | 1.3226666666666642/2 [07:29<03:56, 349.65s/it]Train Epoch: 2 [30720/84843 (36%)]     Loss: 1.144674
 69%|██████▉   | 1.3759999999999961/2 [07:47<03:21, 322.16s/it]Train Epoch: 2 [35840/84843 (42%)]     Loss: 0.904699
 71%|███████   | 1.429333333333328/2 [08:05<03:42, 389.23s/it] Train Epoch: 2 [40960/84843 (48%)]     Loss: 1.047934
 74%|███████▍  | 1.48266666666666/2 [08:23<02:49, 327.05s/it]  Train Epoch: 2 [46080/84843 (54%)]     Loss: 0.939772
 77%|███████▋  | 1.5359999999999918/2 [08:41<02:49, 364.25s/it]Train Epoch: 2 [51200/84843 (60%)]     Loss: 1.058042
 79%|███████▉  | 1.5893333333333237/2 [08:59<02:17, 334.40s/it]Train Epoch: 2 [56320/84843 (66%)]     Loss: 1.024458
 82%|████████▏ | 1.6426666666666556/2 [09:17<01:54, 320.57s/it]Train Epoch: 2 [61440/84843 (72%)]     Loss: 0.708770
```

```
 85%|████████▌  | 1.6959999999999875/2 [09:36<01:47, 355.11s/it]Train Epoch: 2 [66560/84843 (78%)]	Loss: 0.794997
 87%|████████▋  | 1.7493333333333194/2 [09:53<01:20, 321.20s/it]Train Epoch: 2 [71680/84843 (84%)]	Loss: 0.899113
 90%|█████████ | 1.8026666666666513/2 [10:12<01:17, 391.76s/it]Train Epoch: 2 [76800/84843 (90%)]	Loss: 0.846269
 93%|█████████▎ | 1.8559999999999832/2 [10:30<00:46, 326.16s/it]Train Epoch: 2 [81920/84843 (96%)]	Loss: 0.905309
100%|██████████| 1.9999999999999793/2 [11:17<00:00, 338.57s/it]
Test Epoch: 2   Accuracy: 7767/11005 (71%)
```

```
# Let's plot the training loss versus the number of iteration.
plt.plot(losses);
plt.title("training loss");
```




```
def predict(tensor):
    # Use the model to predict the label of the waveform
    tensor = tensor.to(device)
    tensor = transform(tensor)
    tensor = model(tensor.unsqueeze(0))
    tensor = get_likely_index(tensor)
    tensor = index_to_label(tensor.squeeze())
    return tensor


waveform, sample_rate, utterance, *_ = train_set[-1]
ipd.Audio(waveform.numpy(), rate=sample_rate)

print(f"Expected: {utterance}. Predicted: {predict(waveform)}.")
```

```
Expected: zero. Predicted: zero.
```