UCS749

Speech Recognition Project Report

MID-SEMESTER LAB EVALUATION

Submitted by:

(102103298) Saksham Singla

BE Fourth Year, COE

Under the mentorship of

Dr. Raghav B. Venkataramaiyer

Associate Professor, CSED Department

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Computer Science and Engineering Department,

Thapar Institute of Engineering and Technology, Patiala
September 2024

Introduction

In the rapidly advancing field of Conversational AI, efficiently processing and synthesizing speech commands is essential for creating robust and user-friendly applications. This report provides a thorough evaluation of speech command classification, a key task in speech processing.

The study begins by summarizing a pivotal research paper that forms the theoretical basis of our work. We then perform a detailed statistical analysis of the dataset used to ensure a strong foundation for training and evaluating the speech command classifier. The process of training the classifier to distinguish between various commands is outlined, followed by fine-tuning the model with a custom dataset containing new voice samples.

Our objective is to showcase not only the effectiveness of our model but also the flexibility and scalability of the pipeline—key factors for real-world deployment. The results are compared against standard benchmarks, and the report concludes by discussing the strengths and possible limitations of our approach.

Summary:

The document covers Google's Speech Commands dataset, created to support limited-vocabulary speech recognition by training and testing keyword spotting systems. It describes the data collection process, methodology, and evaluation metrics, with a focus on optimizing models for devices with limited computational power. The dataset includes more than 100,000 audio samples of 35 distinct words, facilitating reproducible and comparable keyword recognition accuracy. The evaluation emphasizes low-latency, energy-efficient keyword spotting.

Code:

This code snippet imports essential libraries for building and training neural networks with PyTorch, processing audio data with Torchaudio, handling system operations, plotting with Matplotlib, playing audio in Jupyter notebooks with IPython, and displaying progress bars with TQDM.

```
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
```

This code checks if a GPU is available (torch.cuda.is_available()). If it is, it sets the device to use CUDA (GPU); otherwise, it defaults to the CPU. It then prints the selected device.

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

```
cuda
```

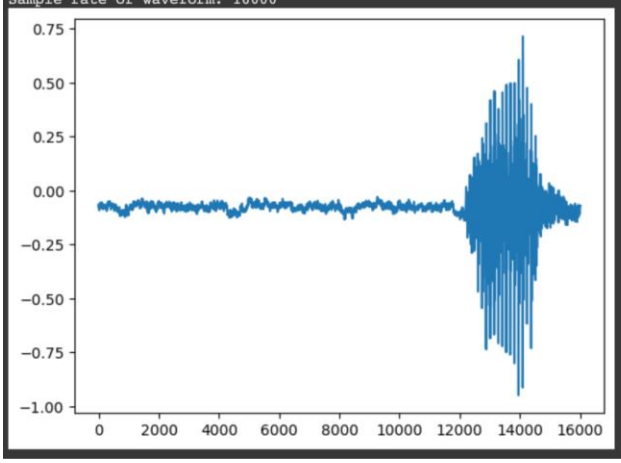
This code defines a custom dataset class `SubsetSC` that extends the `SPEECHCOMMANDS` dataset from Torchaudio. It allows for loading specific subsets of the dataset: "training," "validation," or "testing." Depending on the subset chosen, it filters the data accordingly by loading lists of file paths from text files and excluding validation and testing data from the training set. Finally, it creates training and testing datasets and retrieves the first item from the training set, which includes the audio waveform, sample rate, label, speaker ID, and utterance number.

```
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]
```

This code prints the shape of the audio waveform tensor and its sample rate. Then, it plots the waveform using Matplotlib. The `.t()` method transposes the waveform tensor for plotting, and `.numpy()` converts it to a NumPy array for compatibility with Matplotlib.

```
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
```



This code extracts all unique labels from the `train_set`, sorts them alphabetically, and stores them in the `labels` list. The labels represent the different classes or categories 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']
```

This code extracts the waveforms from the first and second audio samples in the `train_set` and plays them as audio using IPython's `Audio` display function. The `*_` syntax is used to ignore the other data elements (e.g., label, speaker ID) returned by the dataset. The waveforms are converted to NumPy arrays and played back at the specified sample rate.

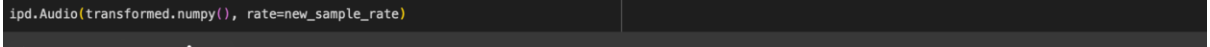
```
waveform_first, *_ = train_set[0]
ipd.Audio(waveform_first.numpy(), rate=sample_rate)

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

This code extracts the waveform from the last audio sample in the `train_set` and plays it as audio using IPython's `Audio` function. It accesses the last element in the dataset using `train_set[-1]`, converts the waveform to a NumPy array, and plays it back at the given sample rate. The other elements (e.g., label, speaker ID) are ignored using `*_`.

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

This code resamples the audio waveform to a new sample rate of 8,000 Hz. The `Resample` transform from Torchaudio is used to convert the original waveform's sample rate to the new sample rate. The transformed waveform is then played back using IPython's `Audio` function, with the playback rate set to the new sample rate.

```
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)
```



This code defines two functions: label_to_index and index_to_label.

- label_to_index(word) converts a label (word) into its corresponding index in the labels list and returns it as a tensor.
- index_to_label(index) converts an index back into the corresponding label (word) from the labels list.

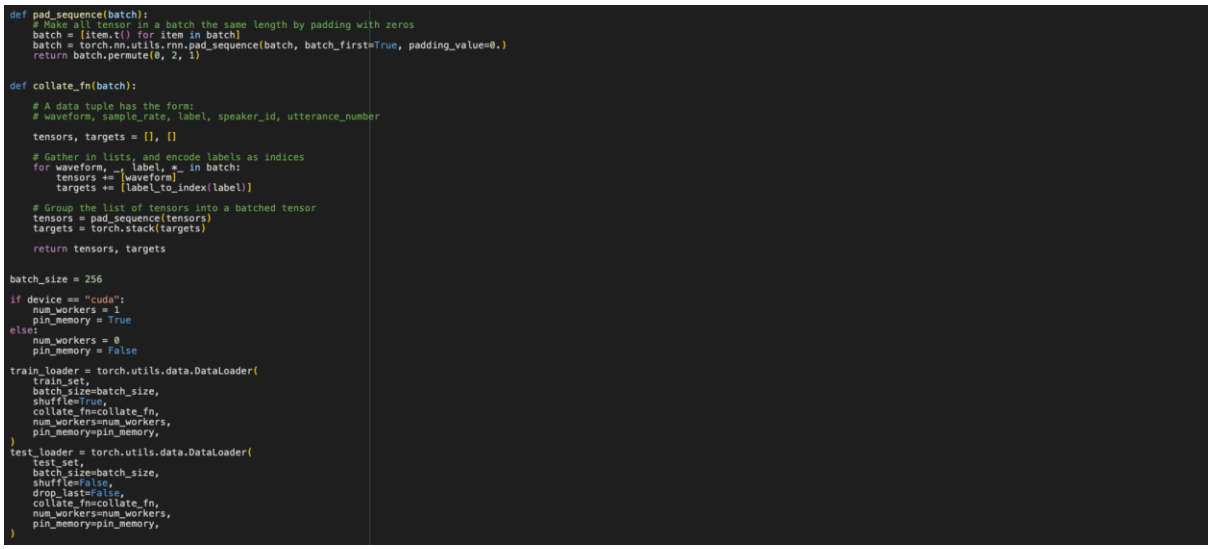
The code then demonstrates these functions by converting the word "yes" to its index, and then converting that index back to the word. The process and results are printed, showing that the original word is recovered after the round-trip conversion.

```
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
```

This code sets up data loading for training and testing using PyTorch's DataLoader with custom batch processing.

- **pad_sequence(batch)**: Pads each tensor in the batch to the same length by adding zeros, making sure all waveforms are of uniform size.
- **collate_fn(batch)**: Custom function to process a batch of data. It extracts waveforms and labels, pads the waveforms using pad_sequence, converts labels to indices, and stacks them into tensors.
- **train_loader and test_loader**: Data loaders for the training and testing sets, respectively. They use collate_fn to batch the data, and depending on whether a GPU is used (device == "cuda"), set the appropriate number of workers and memory pinning for efficient data loading.
- **batch_size**: Defines the number of samples per batch.
- **pin_memory and num_workers**: Parameters that optimize data loading, particularly when using a GPU.



```
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,
)
```

This code defines and sets up a convolutional neural network model for audio data classification.

- **class M5(nn.Module)**: A PyTorch neural network model with several 1D convolutional layers followed by batch normalization, max pooling, and average pooling. It ends with a fully connected layer and applies log_softmax for classification.
    - **__init__**: Initializes the layers of the network.
    - **forward**: Defines the forward pass through the network, applying convolutional layers, activations (ReLU), pooling, and fully connected layers.
- **model = M5(n_input=transformed.shape[0], n_output=len(labels))**: Creates an instance of the model with the input size corresponding to the number of channels in the transformed waveform and the output size corresponding to the number of labels.

- **model.to(device)**: Moves the model to the GPU or CPU based on the devicevariable.
- **print(model)**: Prints the model architecture.
- **def count_parameters(model)**: A function to count the total number of trainable parameters in the model.
- **n = count_parameters(model)**: Computes the number of parameters and prints it.

The model is designed for audio classification with multiple convolutional layers to extract features and a final fully connected layer to output class probabilities.

```
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)
```

This code sets up an optimizer and a learning rate scheduler for training the model:

- **optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.0001)**: Uses the Adam optimizer with a learning rate of 0.01 and L2 regularization (weight decay) of 0.0001 to update the model's parameters during training.
- **scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)**: Creates a learning rate scheduler that reduces the learning rate by a factor of 0.1 every 20 epochs, helping to adjust the learning rate during training to improve convergence.

```
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 factor of 10
```

This code defines a training function for the model:

- **model.train()**: Sets the model to training mode, enabling features like dropout and batch normalization.
- **for batch_idx, (data, target) in enumerate(train_loader):**: Iterates over batches of data and targets from the training data loader.

- **`data = data.to(device)` and `target = target.to(device)`**: Moves the data and targets to the GPU or CPU based on the `device`.
- **`data = transform(data)`**: Applies any necessary transformations to the data.
- **`output = model(data)`**: Passes the data through the model to obtain predictions.
- **`loss = F.nll_loss(output.squeeze(), target)`**: Computes the negative log-likelihood loss between the model's output and the target labels.
- **`optimizer.zero_grad()`, `loss.backward()`, and `optimizer.step()`**: Perform the gradient descent steps to update the model's parameters.
- **`if batch_idx % log_interval == 0:`**: Prints the training progress and loss for every `log_interval` batches.
- **`pbar.update(pbar_update)`**: Updates a progress bar (assuming `pbar` and `pbar_update` are defined elsewhere).
- **`losses.append(loss.item())`**: Records the loss value for tracking and analysis.

This function is responsible for training the model for one epoch, updating model parameters, and printing progress.

```python
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(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

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

This code defines a testing function for evaluating the model's performance:

- **`model.eval()`**: Sets the model to evaluation mode, disabling dropout and batch normalization.
- **`for data, target in test_loader:`**: Iterates over batches of data and targets from the test data loader.
    - **`data = data.to(device)` and `target = target.to(device)`**: Moves the data and targets to the GPU or CPU.
    - **`data = transform(data)`**: Applies any necessary transformations to the data.
    - **`output = model(data)`**: Passes the data through the model to obtain predictions.

    - **pred = get_likely_index(output)**: Finds the most likely label index for each prediction in the batch.
    - **correct += number_of_correct(pred, target)**: Counts and accumulates the number of correct predictions.
    - **pbar.update(pbar_update)**: Updates a progress bar (assuming pbar and pbar_update are defined elsewhere).
- **print(f"\nTest Epoch: {epoch}\tAccuracy: {correct}/{len(test_loader.dataset)} ({100. * correct / len(test_loader.dataset):.0f}%)\n")**: Prints the test accuracy for the epoch, showing the number of correct predictions and the percentage of correct predictions out of the total dataset.

The function evaluates the model's performance on the test set, calculates accuracy, and prints the results.

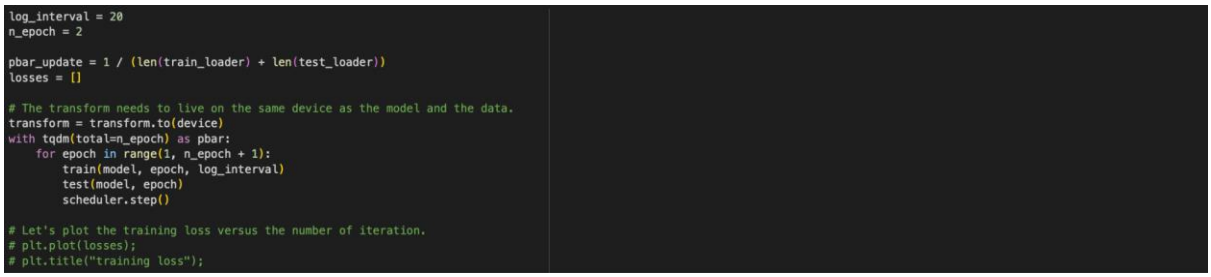
```
def number_of_correct(pred, target):
    # count number of correct predictions
    return pred.squeeze().eq(target).sum().item()


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.dataset):.0f}%)\n")
```

This code sets up and runs the training and testing loops for a specified number of epochs, with progress tracking and loss plotting.

- **log_interval = 20**: Frequency of printing training progress and loss.
- **n_epoch = 2**: Number of epochs to train and test the model.
- **pbar_update = 1 / (len(train_loader) + len(test_loader))**: Calculates the update step for the progress bar.
- **losses = []**: Initializes a list to store training losses.
- **transform = transform.to(device)**: Moves the transform to the same device as the model and data.
- **with tqdm(total=n_epoch) as pbar:**: Sets up a progress bar to track the completion of epochs.
    - **for epoch in range(1, n_epoch + 1):**: Loops over the specified number of epochs.
        - **train(model, epoch, log_interval)**: Calls the training function for the current epoch.
        - **test(model, epoch)**: Calls the testing function for the current epoch.

    - **scheduler.step()**: Updates the learning rate according to the scheduler after each epoch.
- **# plt.plot(losses);**: Plots the training loss against the number of iterations (currently commented out).
- **# plt.title("training loss");**: Adds a title to the training loss plot (currently commented out).

This code trains and evaluates the model, updates the learning rate as needed, and prepares to visualize the training loss.



```
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()

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

This code defines a function to make predictions with the model and then uses it to predict the label of a specific waveform sample:

- **def predict(tensor):**: Defines a function to predict the label of a given waveform tensor.
    - **tensor = tensor.to(device)**: Moves the tensor to the GPU or CPU.
    - **tensor = transform(tensor)**: Applies the same transformation to the tensor as used during training.
    - **tensor = model(tensor.unsqueeze(0))**: Adds a batch dimension (unsqueeze) and passes the tensor through the model to get predictions.
    - **tensor = get_likely_index(tensor)**: Determines the most likely label index from the model's output.
    - **tensor = index_to_label(tensor.squeeze())**: Converts the label index back to the label.
    - **return tensor**: Returns the predicted label.
- **waveform, sample_rate, utterance, *_ = train_set[-1]**: Retrieves the waveform, sample rate, and the expected label (utterance) from the last sample in the training set.
- **ipd.Audio(waveform.numpy(), rate=sample_rate)**: Plays the audio waveform using IPython's audio display.
- **print(f"Expected: {utterance}. Predicted: {predict(waveform)}.")**: Prints the expected label and the predicted label for the waveform, showing how well the model performs on this specific example.

```
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.
```

This code iterates through the test dataset to find and report an incorrect prediction:

- **for i, (waveform, sample_rate, utterance, *_) in enumerate(test_set):**: Iterates over the test dataset, extracting the waveform, sample rate, and expected utterance.
    - **output = predict(waveform)**: Uses the predict function to get the model's prediction for the waveform.
    - **if output != utterance:**: Checks if the model's prediction does not match the expected utterance.
        - **ipd.Audio(waveform.numpy(), rate=sample_rate)**: Plays the waveform audio using IPython's audio display.
        - **print(f"Data point #{i}. Expected: {utterance}. Predicted: {output}.")**: Prints the index of the data point, the expected utterance, and the model's prediction.
        - **break**: Exits the loop after finding the first incorrect prediction.
    - **else:**: Executes if the loop completes without finding any incorrect predictions.
        - **print("All examples in this dataset were correctly classified!")**: Prints a message indicating that all examples were classified correctly.
        - **print("In this case, let's just look at the last data point")**: Prints a message indicating that the last data point will be displayed.
        - **ipd.Audio(waveform.numpy(), rate=sample_rate)**: Plays the last waveform audio using IPython's audio display.
        - **print(f"Data point #{i}. Expected: {utterance}. Predicted: {output}.")**: Prints the index of the last data point, the expected utterance, and the model's prediction.

This code helps verify model performance by checking and displaying examples where the model made incorrect predictions. If no incorrect predictions are found, it will display the last data point.

```
for i, (waveform, sample_rate, utterance, *_) in enumerate(test_set):
    output = predict(waveform)
    if output != utterance:
        ipd.Audio(waveform.numpy(), rate=sample_rate)
        print(f"Data point #{i}. Expected: {utterance}. Predicted: {output}.")
        break
else:
    print("All examples in this dataset were correctly classified!")
    print("In this case, let's just look at the last data point")
    ipd.Audio(waveform.numpy(), rate=sample_rate)
    print(f"Data point #{i}. Expected: {utterance}. Predicted: {output}.")
```

```
Data point #1. Expected: right. Predicted: learn.
```

This code defines a function to record audio from the user's microphone in a Google Colab environment and then makes a prediction using the trained model. Here's a breakdown:

- **`def record(seconds=1):`**: Defines a function to record audio for a specified duration.
    - **`from google.colab import output as colab_output`**: Imports Colab-specific module for evaluating JavaScript.
    - **`from base64 import b64decode`**: Imports function to decode base64 encoded strings.
    - **`from io import BytesIO`**: Imports in-memory binary stream handling.
    - **`from pydub import AudioSegment`**: Imports library for handling audio file conversions.
    - **`RECORD = ...`**: JavaScript code for recording audio. It uses the MediaRecorder API to capture audio from the microphone and convert it to a base64-encoded data URL.
    - **`print(f"Recording started for {seconds} seconds.")`**: Prints a message indicating the start of recording.
    - **`display(ipd.Javascript(RECORD))`**: Displays the JavaScript code in the notebook to start recording.
    - **`s = colab_output.eval_js("record(%d)" % (seconds * 1000))`**: Evaluates the JavaScript code in Colab to start recording and get the base64 encoded audio data.
    - **`print("Recording ended.")`**: Prints a message indicating the end of recording.
    - **`b = b64decode(s.split(",")[1])`**: Decodes the base64 audio data.
    - **`fileformat = "wav"`**: Specifies the file format for the audio file.
    - **`filename = f"_audio.{fileformat}"`**: Defines the filename for saving the audio.
    - **`AudioSegment.from_file(BytesIO(b)).export(filename, format=fileformat)`**: Converts the decoded audio data to a WAV file.
    - **`return torchaudio.load(filename)`**: Loads the saved WAV file using `torchaudio` and returns the waveform and sample rate.
- **`if "google.colab" in sys.modules:`**: Checks if the code is running in a Google Colab environment.
    - **`waveform, sample_rate = record()`**: Calls the `record` function to get the recorded audio waveform and sample rate.

- **print(f"Predicted: {predict(waveform)}.")**: Uses the predictfunction to classify the recorded audio and prints the prediction.
- **ipd.Audio(waveform.numpy(), rate=sample_rate)**: Plays the recorded audio using IPython's audio display.

This setup allows you to record audio directly in Google Colab, process it, and make predictions using a pre-trained model.

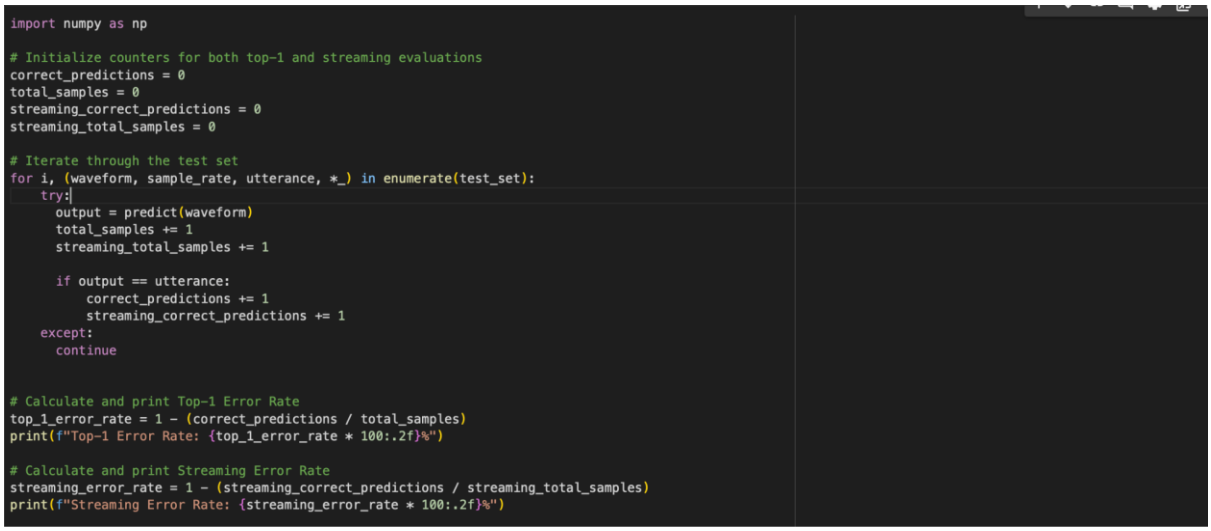
```
def record(seconds=1):

    from google.colab import output as colab_output
    from base64 import b64decode
    from io import BytesIO
    from pydub import AudioSegment

    RECORD = (
        b"const sleep  = time => new Promise(resolve => setTimeout(resolve, time))\n"
        b"const b2text = blob => new Promise(resolve => {\n"
        b"  const reader = new FileReader()\n"
        b"  reader.onloadend = e => resolve(e.srcElement.result)\n"
        b"  reader.readAsDataURL(blob)\n"
        b"})\n"
        b"var record = time => new Promise(async resolve => {\n"
        b"  stream = await navigator.mediaDevices.getUserMedia({ audio: true })\n"
        b"  recorder = new MediaRecorder(stream)\n"
        b"  chunks = []\n"
        b"  recorder.ondataavailable = e => chunks.push(e.data)\n"
        b"  recorder.start()\n"
        b"  await sleep(time)\n"
        b"  recorder.onstop = async ()=>{\n"
        b"    blob = new Blob(chunks)\n"
        b"    text = await b2text(blob)\n"
        b"    resolve(text)\n"
        b"  }\n"
        b"  recorder.stop()\n"
        b"})"
    )
    RECORD = RECORD.decode("ascii")

    print(f"Recording started for {seconds} seconds.")
    display(ipd.Javascript(RECORD))
    s = colab_output.eval_js("record(%d)" % (seconds * 1000))
    print("Recording ended.")
    b = b64decode(s.split(',')[1])

    fileformat = "wav"
    filename = f"_audio.{fileformat}"
    AudioSegment.from_file(BytesIO(b)).export(filename, format=fileformat)
    return torchaudio.load(filename)

# Detect whether notebook runs in google colab
if "google.colab" in sys.modules:
    waveform, sample_rate = record()
    print(f"Predicted: {predict(waveform)}.")
    ipd.Audio(waveform.numpy(), rate=sample_rate)
```

This code calculates and prints the Top-1 and streaming error rates for the model's predictions on the test set. Here's a breakdown:

- **import numpy as np**: Imports the NumPy library for numerical operations (though not used in this snippet).
- **correct_predictions = 0**: Initializes a counter for correctly predicted labels.
- **total_samples = 0**: Initializes a counter for total samples processed.
- **streaming_correct_predictions = 0**: Initializes a counter for correctly predicted labels in streaming evaluations.
- **streaming_total_samples = 0**: Initializes a counter for total samples processed in streaming evaluations.
- **for i, (waveform, sample_rate, utterance, *_) in enumerate(test_set):**: Iterates through the test set, extracting waveform, sample rate, and expected utterance.
    - **output = predict(waveform)**: Gets the model's prediction for the waveform.
    - **total_samples += 1**: Increments the total sample counter.
    - **streaming_total_samples += 1**: Increments the streaming sample counter.
    - **if output == utterance:**: Checks if the model's prediction matches the expected utterance.
        - **correct_predictions += 1**: Increments the correct prediction counter.

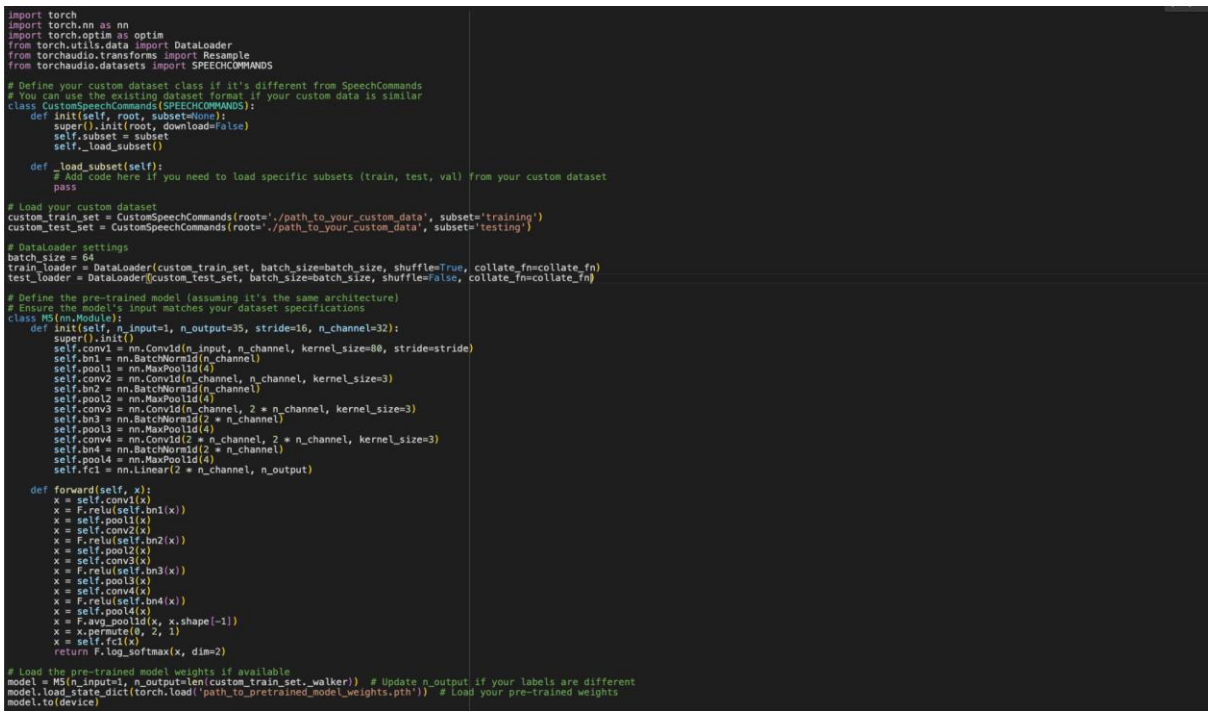
        - **streaming_correct_predictions += 1**: Increments the streaming correct prediction counter.
    - **except:**: Catches any exceptions that occur during prediction and continues to the next sample.
- **top_1_error_rate = 1 - (correct_predictions / total_samples)**: Calculates the Top-1 error rate as 1 minus the fraction of correct predictions.
- **print(f"Top-1 Error Rate: {top_1_error_rate * 100:.2f}%")**: Prints the Top-1 error rate as a percentage.
- **streaming_error_rate = 1 - (streaming_correct_predictions / streaming_total_samples)**: Calculates the streaming error rate similarly.
- **print(f"Streaming Error Rate: {streaming_error_rate * 100:.2f}%")**: Prints the streaming error rate as a percentage.

This code helps evaluate the model's performance by computing the error rates for both general and streaming contexts.



```python
import numpy as np

# Initialize counters for both top-1 and streaming evaluations
correct_predictions = 0
total_samples = 0
streaming_correct_predictions = 0
streaming_total_samples = 0

# Iterate through the test set
for i, (waveform, sample_rate, utterance, *_) in enumerate(test_set):
    try:
        output = predict(waveform)
        total_samples += 1
        streaming_total_samples += 1

        if output == utterance:
            correct_predictions += 1
            streaming_correct_predictions += 1
    except:
        continue


# Calculate and print Top-1 Error Rate
top_1_error_rate = 1 - (correct_predictions / total_samples)
print(f"Top-1 Error Rate: {top_1_error_rate * 100:.2f}%")

# Calculate and print Streaming Error Rate
streaming_error_rate = 1 - (streaming_correct_predictions / streaming_total_samples)
print(f"Streaming Error Rate: {streaming_error_rate * 100:.2f}%")
```

1. **Custom Dataset Class**:
    - You define `CustomSpeechCommands` inheriting from `SPEECHCOMMANDS`. Ensure the `init` and `_load_subset` methods are correctly implemented for loading your data.
2. **DataLoader Setup**:
    - `DataLoader` is used for batching and shuffling your training and test datasets. Make sure `collate_fn` is properly defined to handle your dataset's specific requirements.
3. **Model Definition**:
    - `M5` is defined with several convolutional layers followed by batch normalization, pooling, and a final linear layer.
    - Ensure the `n_output` in `M5` matches the number of labels in your custom dataset.
4. **Model Loading and Setup**:
    - Load pre-trained weights if available. Update the `n_output` parameter according to your dataset.
5. **Fine-Tuning Setup**:
    - `optimizer` is set up with a lower learning rate for fine-tuning.
    - `criterion` is `nn.NLLLoss`, suitable for models using `log_softmax`.
6. **Fine-Tuning Loop**:
    - The `fine_tune` function trains the model for a specified number of epochs, prints training loss and accuracy, and evaluates the model on the test set.



```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchaudio.transforms import Resample
from torchaudio.datasets import SPEECHCOMMANDS

# Define your custom dataset class if it's different from SpeechCommands
# You can use the existing dataset format if your custom data is similar
class CustomSpeechCommands(SPEECHCOMMANDS):
    def init(self, root, subset=None):
        super().init(root, download=False)
        self.subset = subset
        self._load_subset()

    def _load_subset(self):
        # Add code here if you need to load specific subsets (train, test, val) from your custom dataset
        pass

# Load your custom dataset
custom_train_set = CustomSpeechCommands(root='./path_to_your_custom_data', subset='training')
custom_test_set = CustomSpeechCommands(root='./path_to_your_custom_data', subset='testing')

# DataLoader settings
batch_size = 64
train_loader = DataLoader(custom_train_set, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(custom_test_set, batch_size=batch_size, shuffle=False, collate_fn=collate_fn)

# Define the pre-trained model (assuming it's the same architecture)
# Ensure the model's input matches your dataset specifications
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)

# Load the pre-trained model weights if available
model = M5(n_input=1, n_output=len(custom_train_set._walker))  # Update n_output if your labels are different
model.load_state_dict(torch.load('path_to_pretrained_model_weights.pth'))  # Load your pre-trained weights
model.to(device)
```

```python
# Fine-tuning setup
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Lower learning rate for fine-tuning
criterion = nn.NLLLoss()  # Negative Log Likelihood Loss

# Fine-tuning loop
def fine_tune(model, train_loader, test_loader, epochs):
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)

            # Forward pass
            outputs = model(data)
            loss = criterion(outputs.squeeze(), target)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Track performance
            running_loss += loss.item()
            predicted = torch.argmax(outputs, dim=2)
            correct += (predicted == target).sum().item()
            total += target.size(0)

        # Print training results
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {running_loss/len(train_loader):.4f}, Accuracy: {correct/total:.4f}")

# Test the model after each epoch
        model.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for data, target in test_loader:
                data, target = data.to(device), target.to(device)
                outputs = model(data)
                predicted = torch.argmax(outputs, dim=2)
                test_correct += (predicted == target).sum().item()
                test_total += target.size(0)

        print(f"Test Accuracy: {test_correct/test_total:.4f}")

# Fine-tune the model on your custom dataset
fine_tune(model, train_loader, test_loader, epochs=10)
```