**UCS749**

**Speech Recognition Project Report**

**MID-SEMESTER LAB EVALUATION**

**Submitted by:**

**(102103182) Yashmit Kapoor**

**BE Fourth Year, COE**

Under the mentorship of

**Dr. Raghav B. Venkataramaiyer**

Associate Professor, CSED Department



**Computer Science and Engineering Department,**

**Thapar Institute of Engineering and Technology, Patiala**

**September 2024**

**Introduction**

In the fast-paced field of Conversational AI, the ability to effectively process and interpret speech commands is essential for creating robust, user-friendly applications. This report is part of a broader evaluation designed to delve into the complexities of speech command classification—a key task in speech processing.

This assignment takes a comprehensive approach, starting with a succinct overview of a foundational research paper that provides the theoretical basis for our work. Following this, a thorough statistical analysis of the related dataset is conducted to establish a solid groundwork for training and assessing a speech command classifier. The report then outlines the steps involved in training a classifier to differentiate between various commands and describes the fine-tuning of this model using a custom dataset with new voice samples.

Our objective through this work is to showcase not only the effectiveness of our model but also the flexibility and scalability of the entire pipeline, which are crucial for practical applications. The results are analyzed against standard benchmarks, and the report concludes with a discussion on the strengths and possible limitations of our approach.

**Summary:**

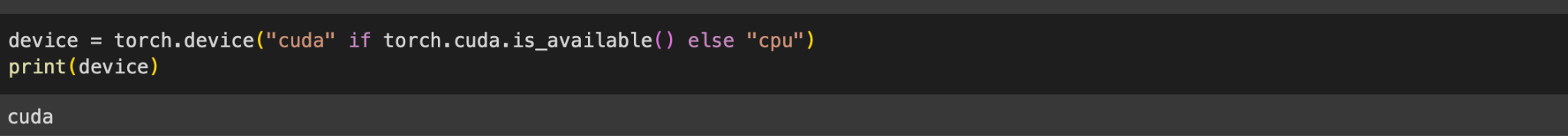
The document discusses Google's Speech Commands dataset, designed to aid limited-vocabulary speech recognition by training and evaluating keyword spotting systems. It outlines data collection, methodology, and evaluation metrics, emphasising efficient models for devices with limited computational resources. The dataset contains over 100,000 audio samples of 35 words, supporting reproducible and comparable accuracy for keyword recognition. Evaluation focuses on low-latency, low-energy keyword spotting.

**Code**:

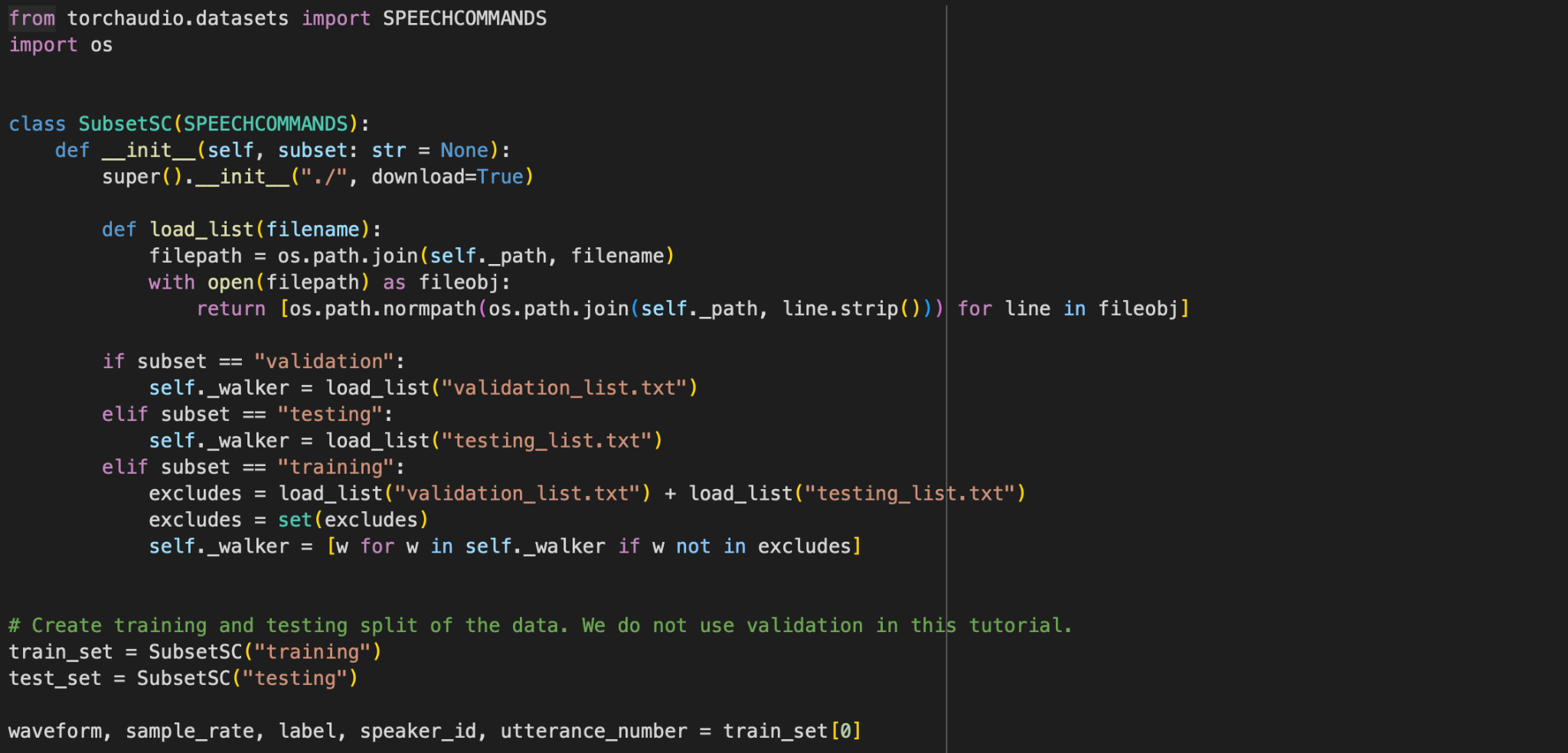
This code snippet imports essential libraries for building and training neural networks with PyTorch, processing audio data with Torchaudio, handling system operations, plotting with Matplotlib, playing audio in Jupyter notebooks with IPython, and displaying progress bars with TQDM.



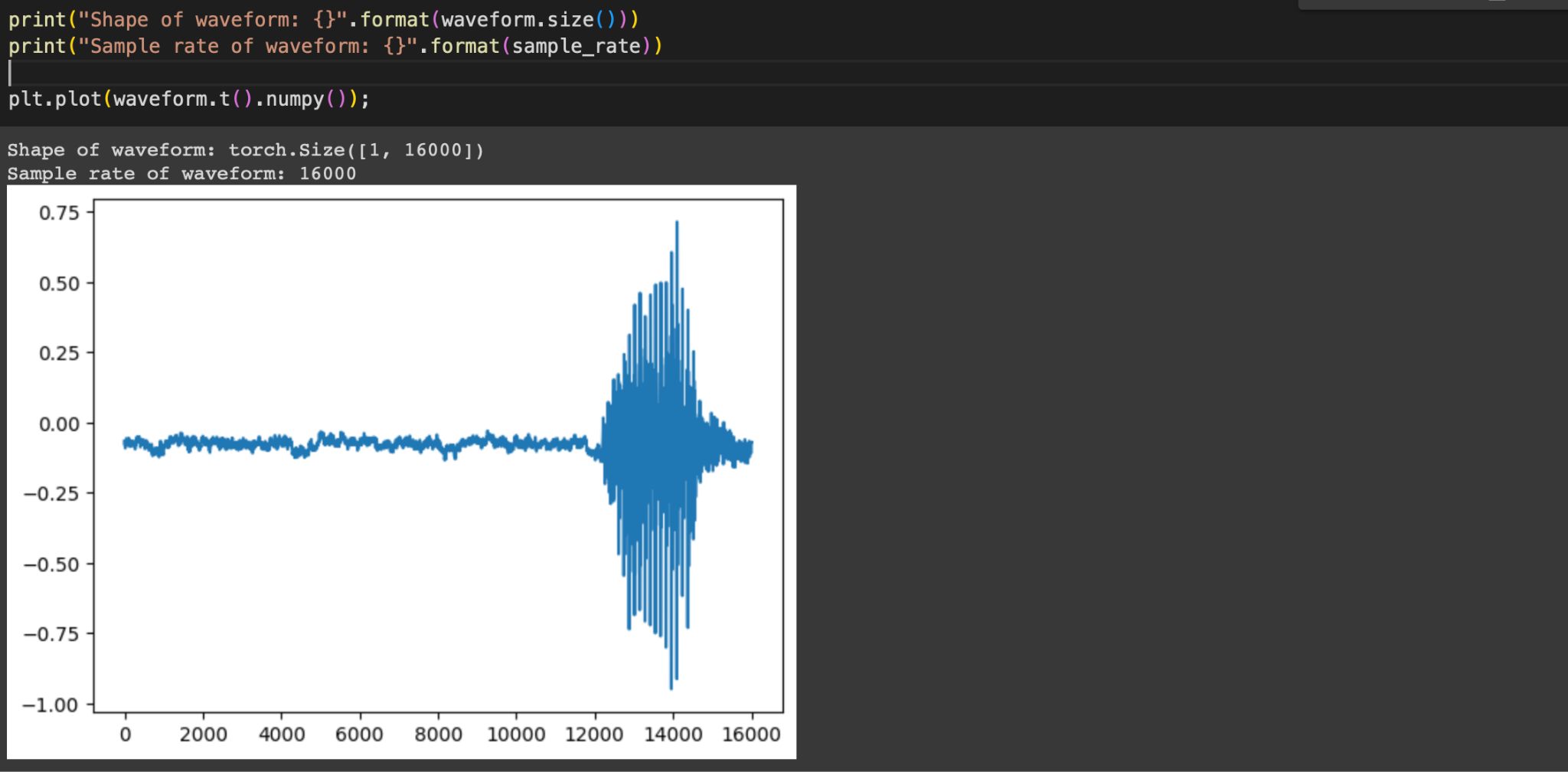
This code checks if a GPU is available (torch.cuda.is\_available()). If it is, it sets the device to use CUDA (GPU); otherwise, it defaults to the CPU. It then prints the selected device.



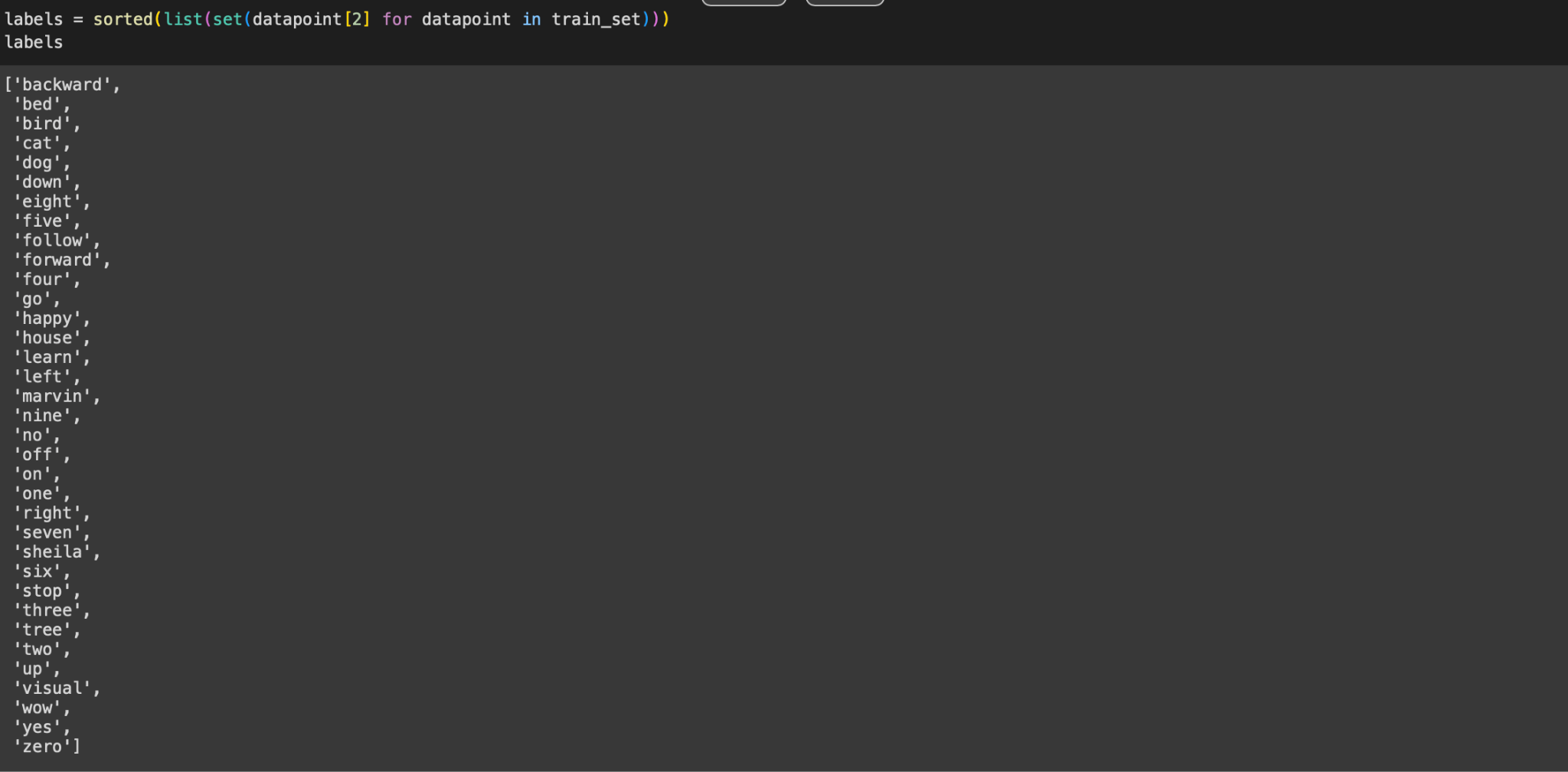
This code defines a custom dataset class `SubsetSC` that extends the `SPEECHCOMMANDS` dataset from Torchaudio. It allows for loading specific subsets of the dataset: "training," "validation," or "testing." Depending on the subset chosen, it filters the data accordingly by loading lists of file paths 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.



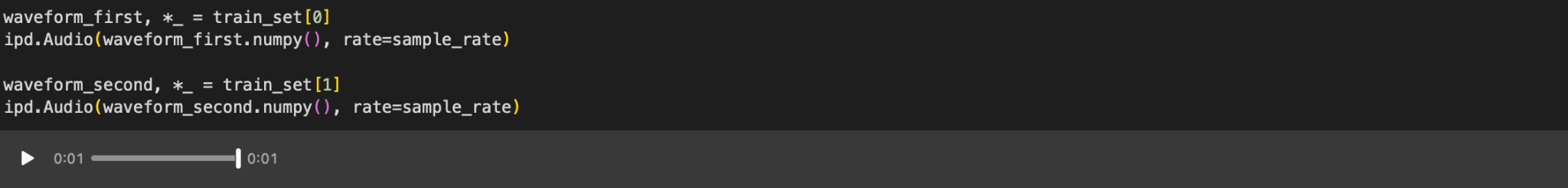
This code prints the shape of the audio waveform tensor and its sample rate. Then, it plots the waveform using Matplotlib. The `.t()` method transposes the waveform tensor for plotting, and `.numpy()` converts it to a NumPy array for compatibility with Matplotlib.



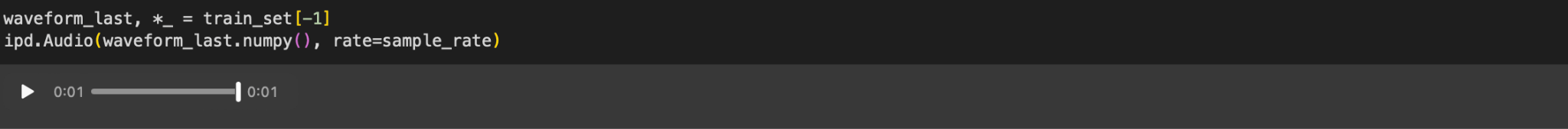
This code extracts all unique labels from the `train\_set`, sorts them alphabetically, and stores them in the `labels` list. The labels represent the different classes or categories in the dataset.



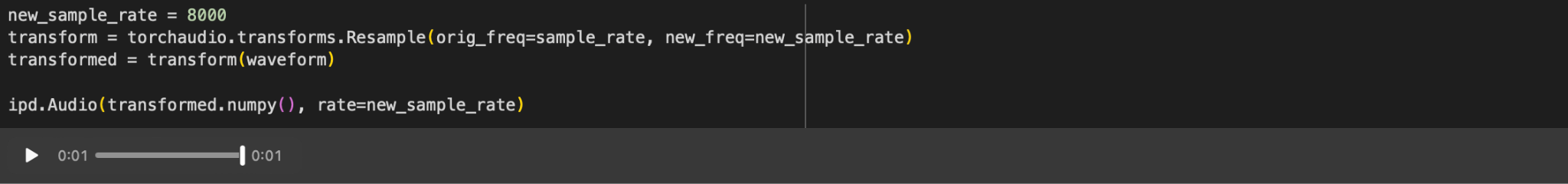
This code extracts the waveforms from the first and second audio samples in the `train\_set` and plays them as audio using IPython's `Audio` display function. The `\*\_` syntax is used to ignore the other data elements (e.g., label, speaker ID) returned by the dataset. The waveforms are converted to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



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 `\*\_`.



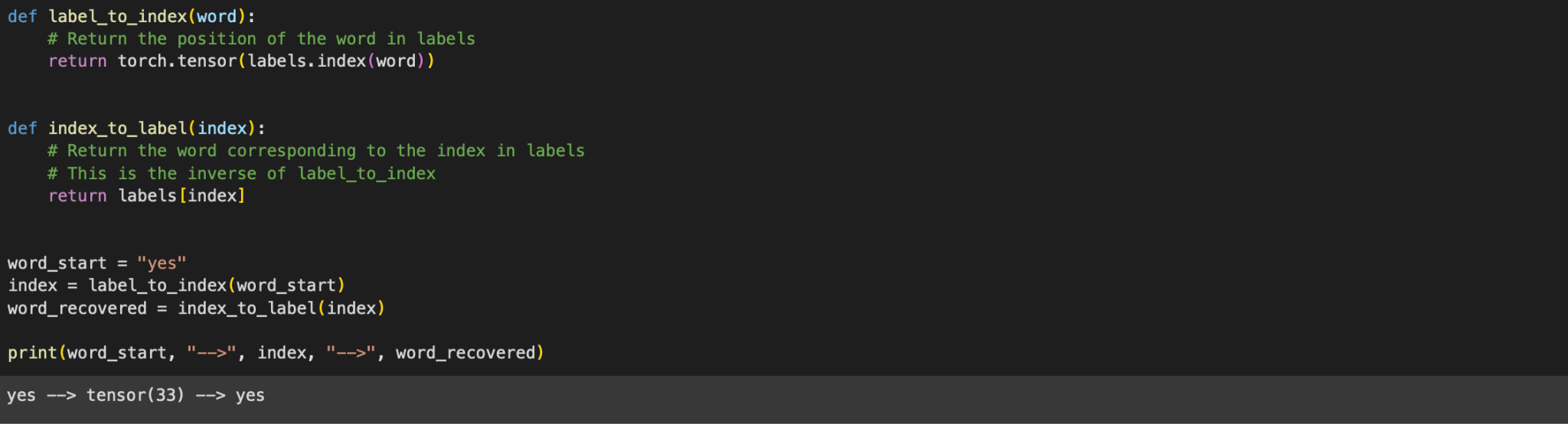
This code resamples the audio waveform to a new sample rate of 8,000 Hz. The `Resample` transform from Torchaudio is used to convert the original waveform's sample rate to the new sample rate. The transformed waveform is then played back using IPython's `Audio` function, with the playback rate set to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



This code defines two functions: label\_to\_index and index\_to\_label.

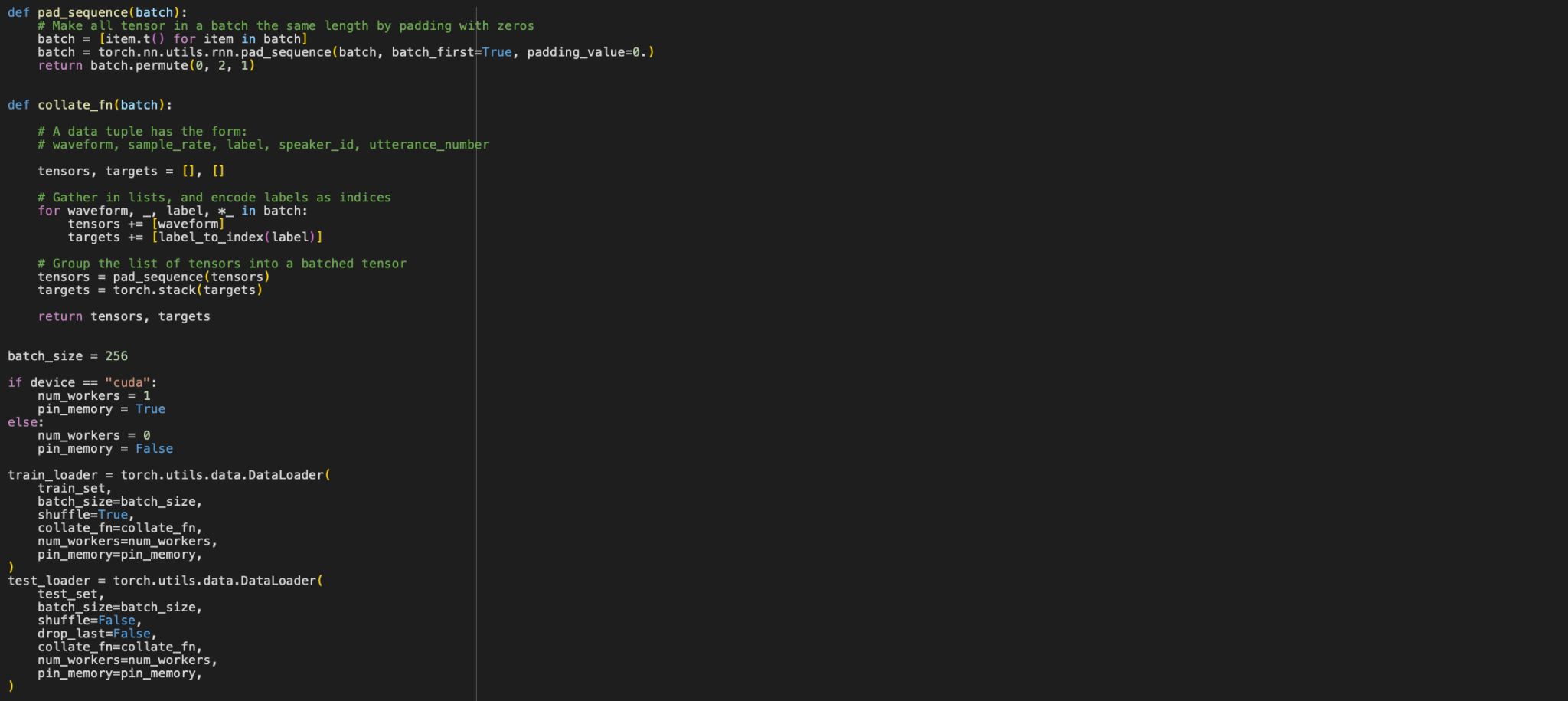
* label\_to\_index(word) converts a label (word) into its corresponding index in the labels list and returns it as a tensor.
* index\_to\_label(index) converts an index back into the corresponding label (word) from the labels list.

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.



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.



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.



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

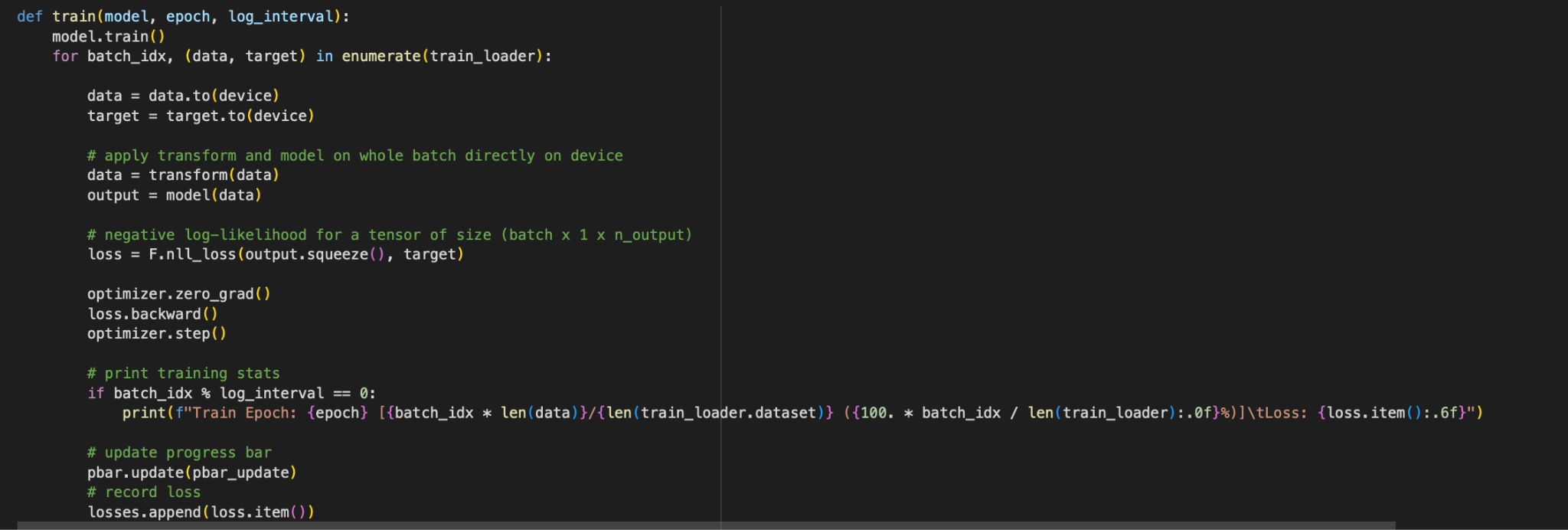
* **optimizer = optim.Adam(model.parameters(), lr=0.01, weight\_decay=0.0001)**: Uses the Adam optimizer with a learning rate of 0.01 and L2 regularization (weight decay) of 0.0001 to update the model’s parameters during training.
* **scheduler = optim.lr\_scheduler.StepLR(optimizer, step\_size=20, gamma=0.1)**: 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.



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.

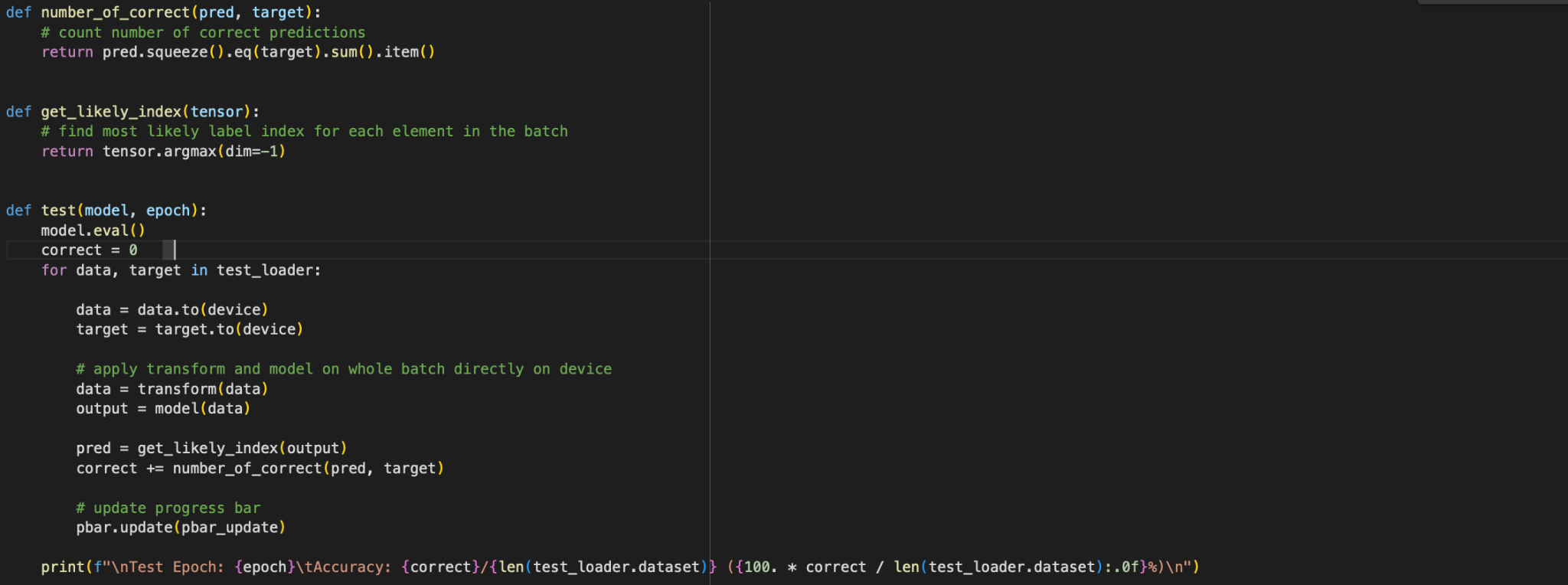
This function is responsible for training the model for one epoch, updating model parameters, and printing progress.



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.

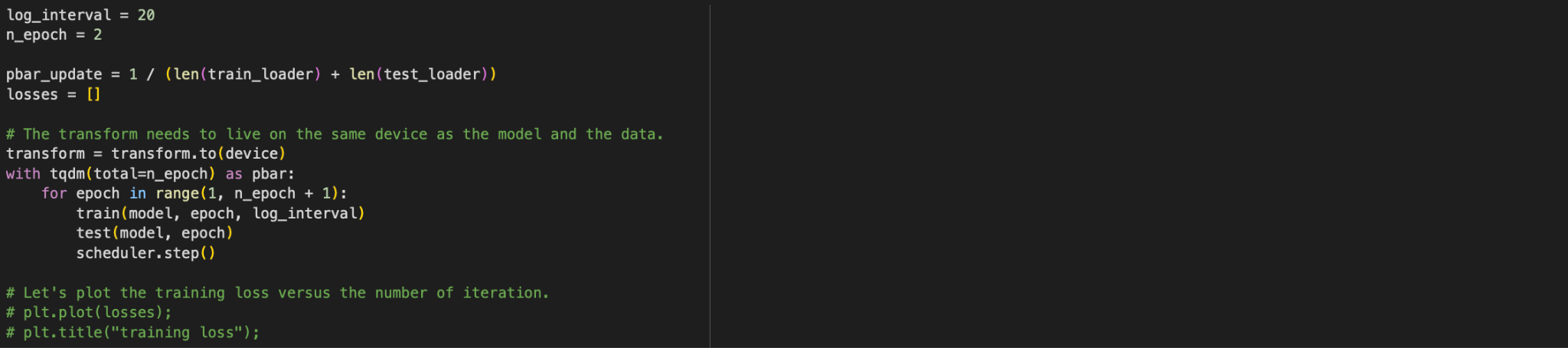
The function evaluates the model’s performance on the test set, calculates accuracy, and prints the results.



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

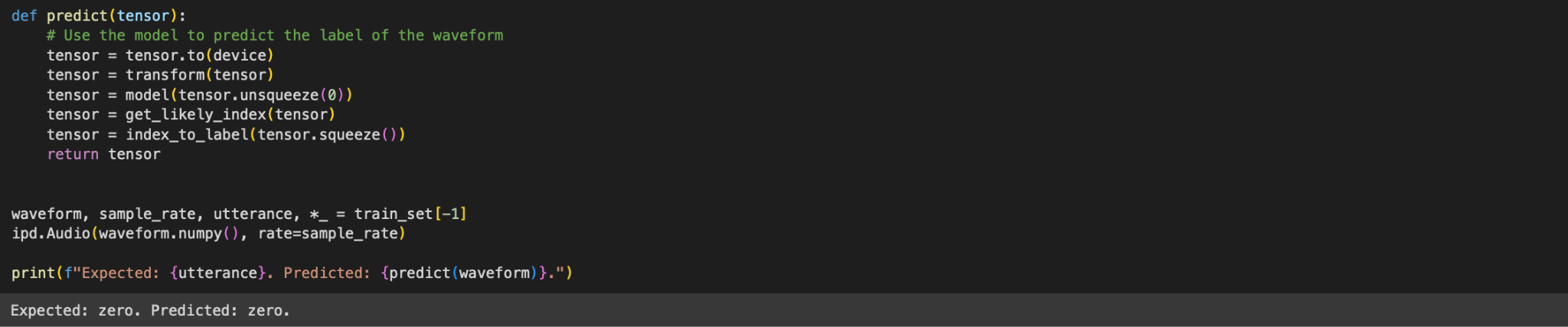
* **log\_interval = 20**: Frequency of printing training progress and loss.
* **n\_epoch = 2**: Number of epochs to train and test the model.
* **pbar\_update = 1 / (len(train\_loader) + len(test\_loader))**: Calculates the update step for the progress bar.
* **losses = []**: Initializes a list to store 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.



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

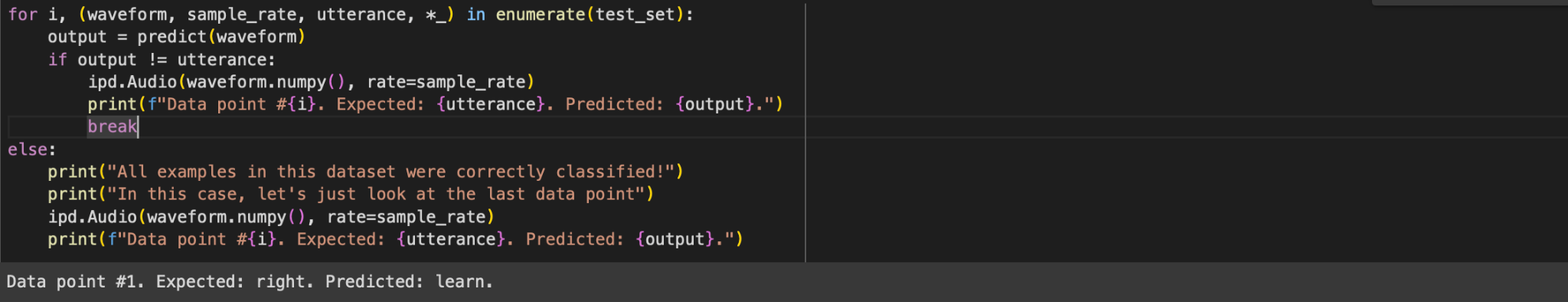
* **def predict(tensor):**: Defines a function to predict the label of a given waveform tensor.
  + **tensor = tensor.to(device)**: Moves the tensor to the GPU or CPU.
  + **tensor = transform(tensor)**: Applies the same transformation to the tensor as used during training.
  + **tensor = model(tensor.unsqueeze(0))**: 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.



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.



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.

