https://drive.google.com/drive/folders/1WnIiuAiqCqwmQue003ZBuSjbYXvUePeY

The paper presents the Speech Commands dataset, designed to support keyword spotting tasks in limited-vocabulary speech recognition. It describes the dataset's collection, consisting of one-second audio samples of words useful for on-device recognition, highlights its applications, and provides baseline results for evaluating speech recognition models.

Statistical Analysis and Description of the Dataset of the Speech Command:

The Speech Commands dataset consists 105 829 audio files, classified into 35 distinct command words. Each audio sample spans about one second and has an average sampled at 16,000 Hz. The dataset is well suited for voice activity detection or keyword detection tasks and contains commands frequently found in voice user interfaces.

The Detailed Online Analysis:

Number of Samples:

Total audio files: 105,829.

Unique commands: 35.

Analysis of Specific Action Implicit in the Command Distribution:

The dataset contains a variety of commands with different sample sizes.

The most frequently used commands include the words ‘five’ with 4,052 samples, ‘yes’ with 4,044 samples and ‘zero’ with 4,052 samples.

Some commands, for instance, “follow” “learn” and “visual” which have 1,579 1,575 and 1,592 samples respectively may result in poor performance of the model on these commands as they have few samples.

Sample Duration:

Average duration: 0.99 seconds;

Duration range: 0.73 to 1 second.

Most samples’ length is about one second with minimal variations hence uniformity of the dataset for training.

Sample Rate:

All the samples were acquired at a constant sampling frequency of 16000 Hz which is the normal procedure in speech recognition activity.

Command Distribution Visualization:

The bar chart provided below shows the distribution of commands, highlighting commands with high and low sample counts, which is crucial for understanding dataset balance. Implications for Model Training: Balanced Commands: Commands like "five," "yes," and "zero" have ample representation, ensuring robust training for these categories. Underrepresented Commands: Commands such as "follow" and "learn" have fewer samples, which could lead to underperformance unless addressed with techniques like data augmentation. Consistent Sample Rate and Duration: The uniform sample rate and nearly identical duration of the samples make the dataset well-suited for neural network training without extensive preprocessing.

```
!pip install -U -q tensorflow tensorflow_datasets

import os
import pathlib

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import models
from IPython import display

# Set the seed value for experiment reproducibility.
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)
```

Import Speech Commands dataset

:

```
import urllib.request
import tarfile

# URL of the dataset
url =
'http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz'
filename = 'speech_commands_v0.02.tar.gz'

# Download the dataset
urllib.request.urlretrieve(url, filename)

# Extract the dataset
with tarfile.open(filename, 'r:gz') as tar:
    tar.extractall(path='./speech_commands')
print("Dataset downloaded and extracted.")
```

```
Dataset downloaded and extracted.
```

```
data_dir = pathlib.Path('./speech_commands')

commands = np.array(tf.io.gfile.listdir(str(data_dir)))
commands = commands[(commands != 'README.md') & (commands !=
'.DS_Store')]
print('Commands:', commands)
```

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

```
len(commands)
```

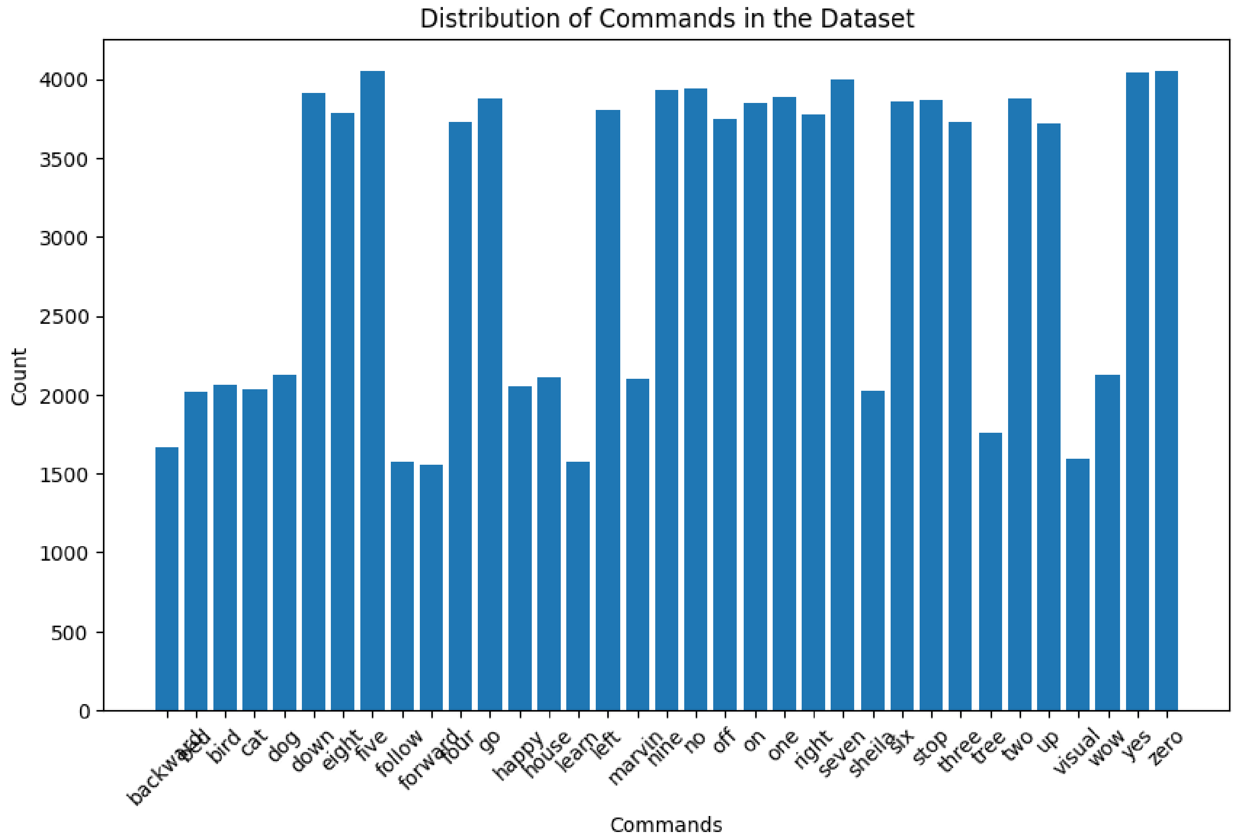
```
39
```

```
import matplotlib.pyplot as plt
from scipy.io import wavfile

# Step 2: List all files and commands
all_files = []
commands = []

# Traverse the directory to gather all .wav files
for root, dirs, files in os.walk(data_dir):
    if "_background_noise_" not in root:  # Skip background noise
folder
        for file in files:
            if file.endswith(".wav"):
                all_files.append(os.path.join(root, file))
                commands.append(root.split('/')[-1])

# Step 3: Statistical Analysis
# Get unique commands and count occurrences
unique_commands, counts = np.unique(commands, return_counts=True)

# Display statistical summary
print(f"Total number of audio files: {len(all_files)}")
print(f"Unique commands: {len(unique_commands)}")
print("Commands and their counts:")
for command, count in zip(unique_commands, counts):
    print(f"{command}: {count}")

# Step 4: Analyze Duration and Sample Rates of Files
durations = []
sample_rates = []
```

```python
for file in all_files[:100]:  # Limiting to first 100 files for quick analysis
    sample_rate, audio_data = wavfile.read(file)
    sample_rates.append(sample_rate)
    durations.append(len(audio_data) / sample_rate)

# Display basic statistics
print(f"\nAverage Sample Rate: {np.mean(sample_rates):.2f} Hz")
print(f"Average Duration: {np.mean(durations):.2f} seconds")
print(f"Duration Range: {min(durations):.2f} - {max(durations):.2f} seconds")

# Step 5: Visualize Distribution of Commands
plt.figure(figsize=(10, 6))
plt.bar(unique_commands, counts)
plt.xticks(rotation=45)
plt.title("Distribution of Commands in the Dataset")
plt.xlabel("Commands")
plt.ylabel("Count")
plt.show()
```

```
Total number of audio files: 105829
Unique commands: 35
Commands and their counts:
backward: 1664
bed: 2014
bird: 2064
cat: 2031
dog: 2128
down: 3917
eight: 3787
five: 4052
follow: 1579
forward: 1557
four: 3728
go: 3880
happy: 2054
house: 2113
learn: 1575
left: 3801
marvin: 2100
nine: 3934
no: 3941
off: 3745
on: 3845
one: 3890
right: 3778
seven: 3998
sheila: 2022
```

```
six: 3860
stop: 3872
three: 3727
tree: 1759
two: 3880
up: 3723
visual: 1592
wow: 2123
yes: 4044
zero: 4052

Average Sample Rate: 16000.00 Hz
Average Duration: 0.99 seconds
Duration Range: 0.73 - 1.00 seconds
```



Preprocessing

```
train_ds, val_ds = tf.keras.utils.audio_dataset_from_directory(
    directory=data_dir,
    batch_size=64,
    validation_split=0.2,
    seed=0,
    output_sequence_length=16000,
```

```
    subset='both')

label_names = np.array(train_ds.class_names)
print()
print("label names:", label_names)
```

```
Found 105835 files belonging to 36 classes.
Using 84668 files for training.
Using 21167 files for validation.

label names: ['_background_noise_' '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']
```

```
train_ds.element_spec
```

```
(TensorSpec(shape=(None, 16000, None), dtype=tf.float32, name=None),
 TensorSpec(shape=(None,), dtype=tf.int32, name=None))
```

```
def squeeze(audio, labels):
  audio = tf.squeeze(audio, axis=-1)
  return audio, labels

train_ds = train_ds.map(squeeze, tf.data.AUTOTUNE)
val_ds = val_ds.map(squeeze, tf.data.AUTOTUNE)

test_ds = val_ds.shard(num_shards=2, index=0)
val_ds = val_ds.shard(num_shards=2, index=1)

for example_audio, example_labels in train_ds.take(1):
  print(example_audio.shape)
  print(example_labels.shape)
```

```
(64, 16000)
(64,)
```

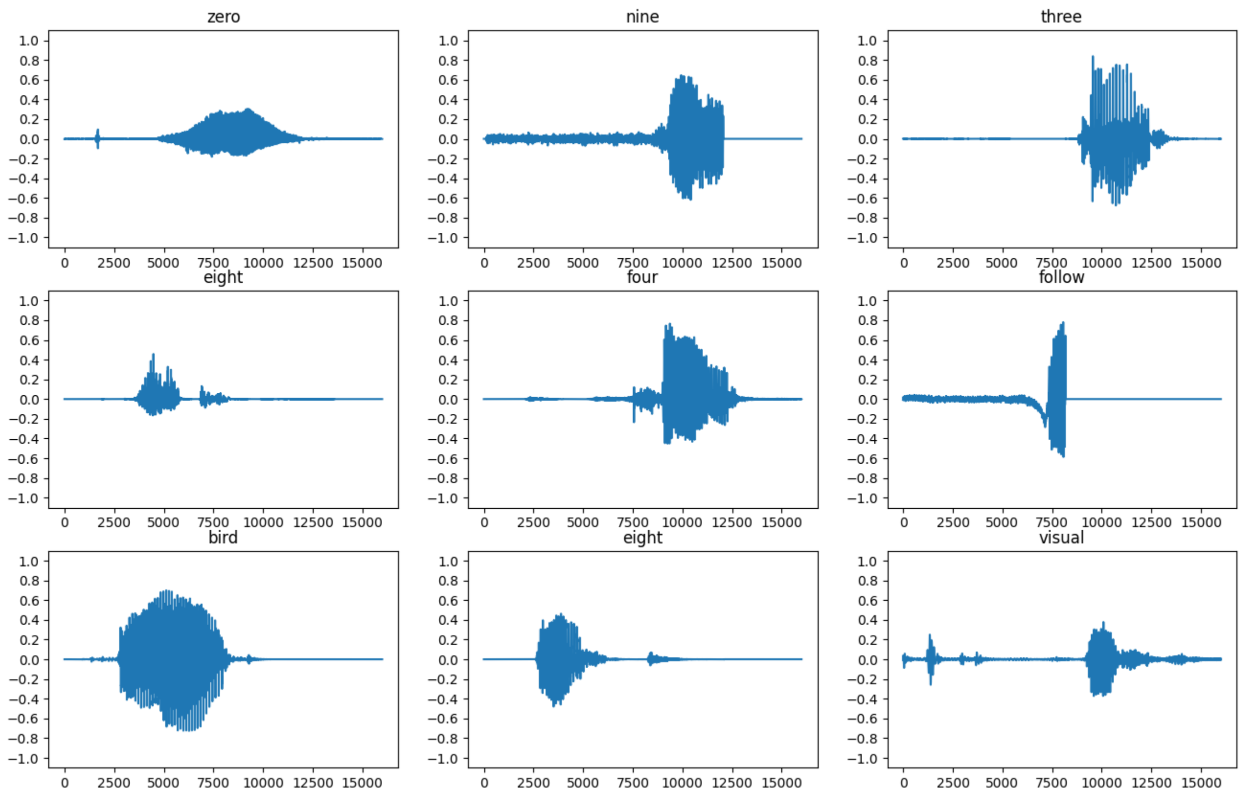
Let's plot a few audio waveforms:

```
label_names[[1,1,3,0]]
```

```
array(['backward', 'backward', 'bird', '_background_noise_'],
dtype='<U18')
```

```
plt.figure(figsize=(16, 10))
rows = 3
cols = 3
n = rows * cols
for i in range(n):
  plt.subplot(rows, cols, i+1)
```

```
  audio_signal = example_audio[i]
  plt.plot(audio_signal)
  plt.title(label_names[example_labels[i]])
  plt.yticks(np.arange(-1.2, 1.2, 0.2))
  plt.ylim([-1.1, 1.1])
```



Convert waveforms to spectrograms

```
def get_spectrogram(waveform):
  # Convert the waveform to a spectrogram via a STFT.
  spectrogram = tf.signal.stft(
      waveform, frame_length=255, frame_step=128)
  # Obtain the magnitude of the STFT.
  spectrogram = tf.abs(spectrogram)
  # Add a `channels` dimension, so that the spectrogram can be used
  # as image-like input data with convolution layers (which expect
  # shape (`batch_size`, `height`, `width`, `channels`).
  spectrogram = spectrogram[..., tf.newaxis]
  return spectrogram

for i in range(3):
  label = label_names[example_labels[i]]
  waveform = example_audio[i]
  spectrogram = get_spectrogram(waveform)
```

```
  print('Label:', label)
  print('Waveform shape:', waveform.shape)
  print('Spectrogram shape:', spectrogram.shape)
  print('Audio playback')
  display.display(display.Audio(waveform, rate=16000))
```

```
Label: zero
Waveform shape: (16000,)
Spectrogram shape: (124, 129, 1)
Audio playback

<IPython.lib.display.Audio object>

Label: nine
Waveform shape: (16000,)
Spectrogram shape: (124, 129, 1)
Audio playback

<IPython.lib.display.Audio object>

Label: three
Waveform shape: (16000,)
Spectrogram shape: (124, 129, 1)
Audio playback

<IPython.lib.display.Audio object>
```

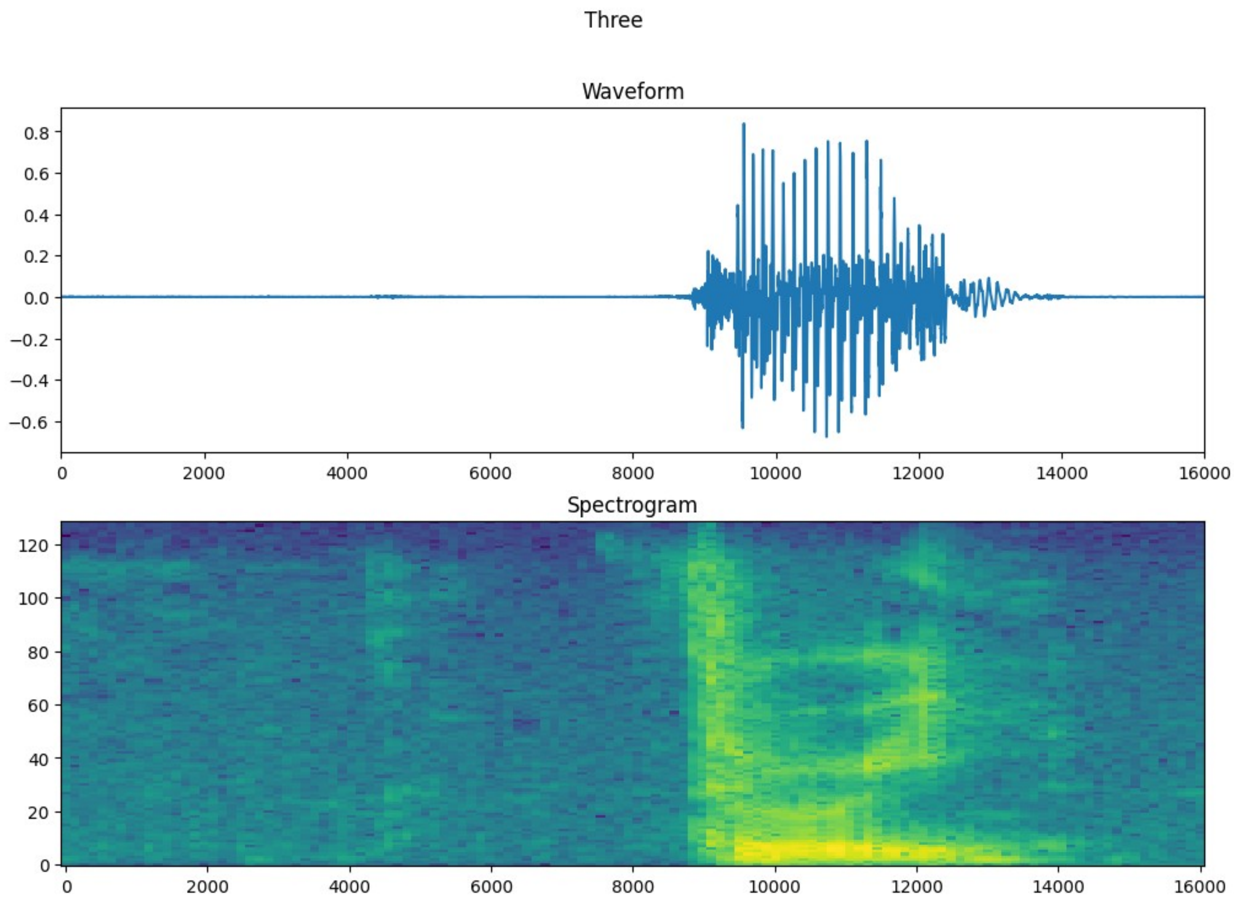
function for displaying a spectrogram:

```
def plot_spectrogram(spectrogram, ax):
  if len(spectrogram.shape) > 2:
    assert len(spectrogram.shape) == 3
    spectrogram = np.squeeze(spectrogram, axis=-1)
  # Convert the frequencies to log scale and transpose, so that the
time is
  # represented on the x-axis (columns).
  # Add an epsilon to avoid taking a log of zero.
  log_spec = np.log(spectrogram.T + np.finfo(float).eps)
  height = log_spec.shape[0]
  width = log_spec.shape[1]
  X = np.linspace(0, np.size(spectrogram), num=width, dtype=int)
  Y = range(height)
  ax.pcolormesh(X, Y, log_spec)
```

Plot the example's waveform over time and the corresponding spectrogram (frequencies over time):

```
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(waveform.shape[0])
axes[0].plot(timescale, waveform.numpy())
```

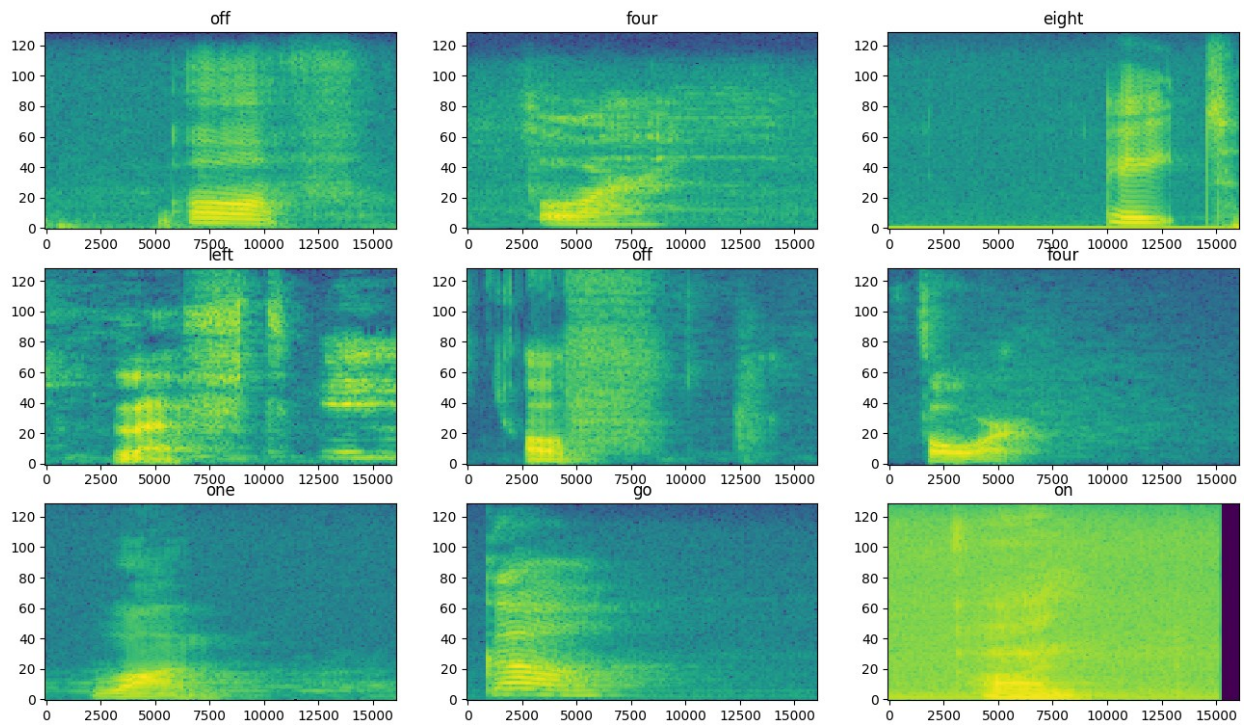
```
axes[0].set_title('Waveform')
axes[0].set_xlim([0, 16000])

plot_spectrogram(spectrogram.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.suptitle(label.title())
plt.show()
```



spectrogram datasets from the audio datasets:

```
def make_spec_ds(ds):
  return ds.map(
      map_func=lambda audio,label: (get_spectrogram(audio), label),
      num_parallel_calls=tf.data.AUTOTUNE)

train_spectrogram_ds = make_spec_ds(train_ds)
val_spectrogram_ds = make_spec_ds(val_ds)
test_spectrogram_ds = make_spec_ds(test_ds)

for example_spectrograms, example_spect_labels in
train_spectrogram_ds.take(1):
  break
```

```
rows = 3
cols = 3
n = rows*cols
fig, axes = plt.subplots(rows, cols, figsize=(16, 9))

for i in range(n):
    r = i // cols
    c = i % cols
    ax = axes[r][c]
    plot_spectrogram(example_spectrograms[i].numpy(), ax)
    ax.set_title(label_names[example_spect_labels[i].numpy()])

plt.show()
```



Building and training the model

Add `Dataset.cache` and `Dataset.prefetch` operations to reduce read latency while training the model:

```
train_spectrogram_ds =
train_spectrogram_ds.cache().shuffle(10000).prefetch(tf.data.AUTOTUNE)
val_spectrogram_ds =
val_spectrogram_ds.cache().prefetch(tf.data.AUTOTUNE)
test_spectrogram_ds =
test_spectrogram_ds.cache().prefetch(tf.data.AUTOTUNE)
```

For the model, you'll use a simple convolutional neural network (CNN), since you have transformed the audio files into spectrogram images.

Your `tf.keras.Sequential` model will use the following Keras preprocessing layers:

- `tf.keras.layers.Resizing`: to downsample the input to enable the model to train faster.
- `tf.keras.layers.Normalization`: to normalize each pixel in the image based on its mean and standard deviation.

For the `Normalization` layer, its `adapt` method would first need to be called on the training data in order to compute aggregate statistics (that is, the mean and the standard deviation).

```
input_shape = example_spectrograms.shape[1:]
print('Input shape:', input_shape)
num_labels = len(label_names)

# Instantiate the `tf.keras.layers.Normalization` layer.
norm_layer = layers.Normalization()
# Fit the state of the layer to the spectrograms
# with `Normalization.adapt`.
norm_layer.adapt(data=train_spectrogram_ds.map(map_func=lambda spec,
label: spec))

model = models.Sequential([
    layers.Input(shape=input_shape),
    # Downsample the input.
    layers.Resizing(32, 32),
    # Normalize.
    norm_layer,
    layers.Conv2D(32, 3, activation='relu'),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Dropout(0.25),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_labels),
])

model.summary()
```

```
Input shape: (124, 129, 1)

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇
```

```
━━━━━━━━━━━━━━━━━━━┩
│ resizing (Resizing)                  │ (None, 32, 32, 1)             │
0 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ normalization (Normalization)        │ (None, 32, 32, 1)             │
3 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ conv2d (Conv2D)                      │ (None, 30, 30, 32)            │
320 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 28, 28, 64)            │
18,496 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 14, 14, 64)            │
0 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ dropout (Dropout)                    │ (None, 14, 14, 64)            │
0 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ flatten (Flatten)                    │ (None, 12544)                 │
0 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ dense (Dense)                        │ (None, 128)                   │
1,605,760 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ dropout_1 (Dropout)                  │ (None, 128)                   │
0 │
├──────────────────────────────────────┼───────────────────────────────┼
───────────────────┤
│ dense_1 (Dense)                      │ (None, 36)                    │
4,644 │
└──────────────────────────────────────┴───────────────────────────────┴
───────────────────┘

 Total params: 1,629,223 (6.21 MB)

 Trainable params: 1,629,220 (6.21 MB)

 Non-trainable params: 3 (16.00 B)
```

Configure the Keras model with the Adam optimizer and the cross-entropy loss:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'],
)
```

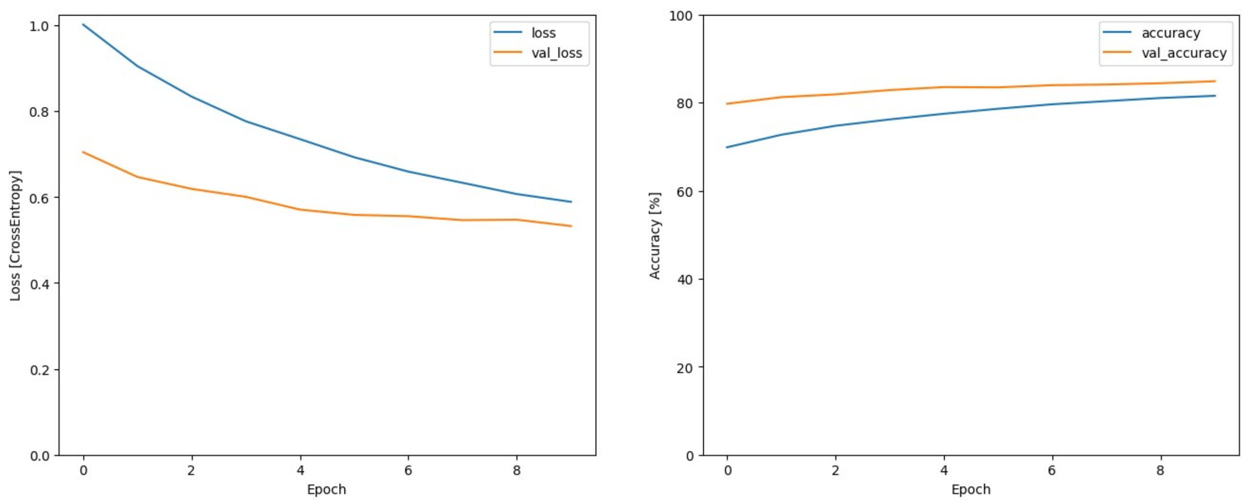
Train the model over 10 epochs for demonstration purposes:

```
EPOCHS = 10
history = model.fit(
    train_spectrogram_ds,
    validation_data=val_spectrogram_ds,
    epochs=EPOCHS,
    callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=2),
)

Epoch 1/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.6950 - loss:
1.0117 - val_accuracy: 0.7973 - val_loss: 0.7046
Epoch 2/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.7270 - loss:
0.9032 - val_accuracy: 0.8123 - val_loss: 0.6469
Epoch 3/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.7462 - loss:
0.8341 - val_accuracy: 0.8187 - val_loss: 0.6191
Epoch 4/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.7634 - loss:
0.7684 - val_accuracy: 0.8284 - val_loss: 0.6006
Epoch 5/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.7762 - loss:
0.7241 - val_accuracy: 0.8352 - val_loss: 0.5709
Epoch 6/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.7873 - loss:
0.6827 - val_accuracy: 0.8346 - val_loss: 0.5586
Epoch 7/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.7996 - loss:
0.6478 - val_accuracy: 0.8395 - val_loss: 0.5555
Epoch 8/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8054 - loss:
0.6221 - val_accuracy: 0.8410 - val_loss: 0.5464
Epoch 9/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8147 - loss:
0.5984 - val_accuracy: 0.8439 - val_loss: 0.5473
Epoch 10/10
1323/1323 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.8141 - loss:
0.5872 - val_accuracy: 0.8485 - val_loss: 0.5325
```

Let's plot the training and validation loss curves to check how your model has improved during training:

```
metrics = history.history
plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(history.epoch, metrics['loss'], metrics['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.ylim([0, max(plt.ylim())])
plt.xlabel('Epoch')
plt.ylabel('Loss [CrossEntropy]')

plt.subplot(1,2,2)
plt.plot(history.epoch, 100*np.array(metrics['accuracy']),
100*np.array(metrics['val_accuracy']))
plt.legend(['accuracy', 'val_accuracy'])
plt.ylim([0, 100])
plt.xlabel('Epoch')
plt.ylabel('Accuracy [%]')
```

```
Text(0, 0.5, 'Accuracy [%]')
```



Evaluate the model performance

Run the model on the test set and check the model's performance:

```
model.evaluate(test_spectrogram_ds, return_dict=True)
```

```
166/166 ━━━━━━━━━━━━━━━━━━━━ 28s 169ms/step - accuracy: 0.8522 - loss:
0.5157

{'accuracy': 0.8556613326072693, 'loss': 0.5092366337776184}
```

Display a confusion matrix

Use a confusion matrix to check how well the model did classifying each of the commands in the test set:

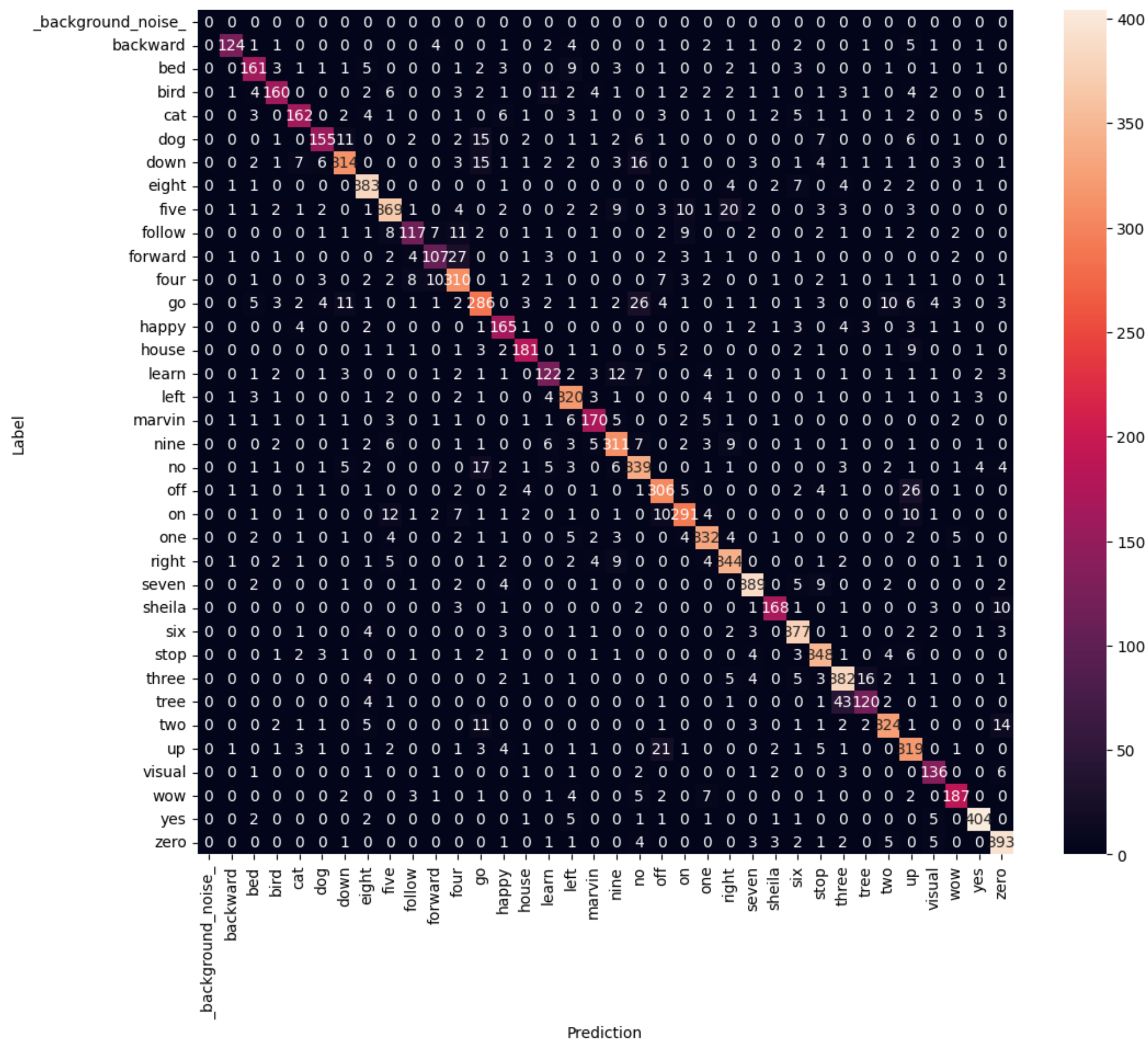
```
y_pred = model.predict(test_spectrogram_ds)
```

```
166/166 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
```

```
y_pred = tf.argmax(y_pred, axis=1)
```

```
y_true = tf.concat(list(test_spectrogram_ds.map(lambda s,lab: lab)),
axis=0)
```

```
confusion_mtx = tf.math.confusion_matrix(y_true, y_pred)
plt.figure(figsize=(12, 10))
sns.heatmap(confusion_mtx,
            xticklabels=label_names,
            yticklabels=label_names,
            annot=True, fmt='g')
plt.xlabel('Prediction')
plt.ylabel('Label')
plt.show()
```

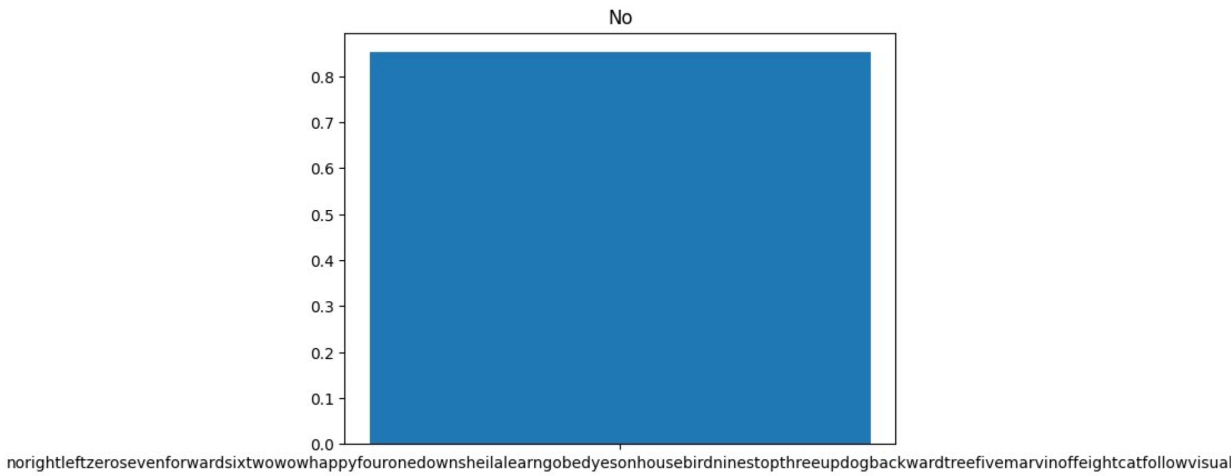


Run inference on an audio file

Finally, verify the model's prediction output using an input audio file of someone saying "no". How well does your model perform?

```
x = data_dir/'no/01bb6a2a_nohash_0.wav'
x = tf.io.read_file(str(x))
x, sample_rate = tf.audio.decode_wav(x, desired_channels=1,
desired_samples=16000,)
x = tf.squeeze(x, axis=-1)
waveform = x
x = get_spectrogram(x)
x = x[tf.newaxis,...]

prediction = model(x)
```

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

plt.bar(x_labels, tf.nn.softmax(prediction[0]))
plt.title('No')
plt.show()

display.display(display.Audio(waveform, rate=16000))
```




```
<IPython.lib.display.Audio object>
```

As the output suggests, your model should have recognized the audio command as "no".

Save And Export the model

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

model.save("/content/drive/MyDrive/keras_model.keras")

ok wonderful..

  File "<ipython-input-9-d853894db0ac>", line 1
    ok wonderful..
       ^
SyntaxError: invalid syntax
```