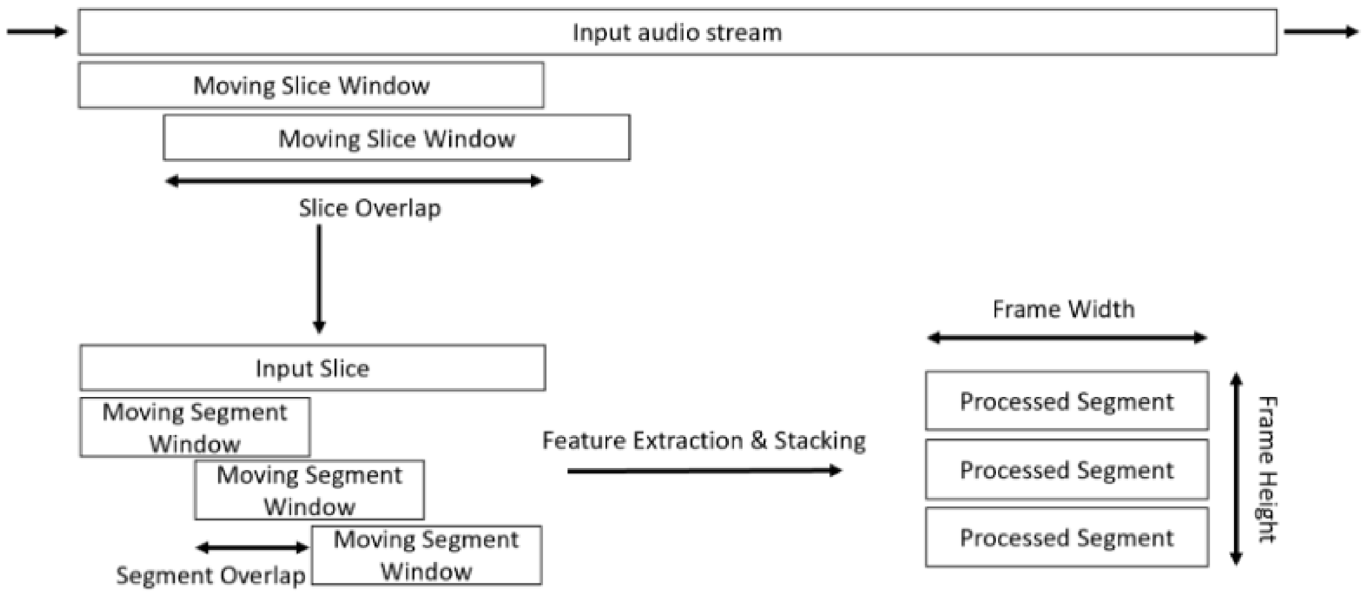
Lab 3: Speech Recognition

Learning outcome: Learn an efficient method for processing audio recordings using Mel-frequency cepstral coefficient (MFCC) and a convolutional neural network (CNN).

Introduction

Lab overview

In this lab you will learn an efficient method for processing audio recordings using Mel-frequency cepstral coefficient (MFCC) and a convolutional neural network (CNN).



The idea of MFCC feature extraction and network training is to segment one slice of a continuous input signal into (overlapping) segments, do some preprocessing, and stack the resulting processed segments over one-another to obtain a two-dimensional frame. On these frames the same principles are used as for ”real” images when it comes to machine learning.

Requirements

Software functions

Anaconda environment of lab 3 including the Python packages listed below and internet connection.

Python Packages

You will be using json, random, numpy, scipy, tensorflow tqdm (version: >4.50.2), zipfile, tempfile, simpleaudio. For a complete list of packages, refer to the `lab3_conda.yml` file. Install the required packages using Anaconda.

The "Hey Snips" Dataset

For this lab we are going to use the "Hey Snips" dataset from Couke et al. 2018 "Efficient keyword spotting using dilated convolutions and gating". The train dataset consists of utterances from 30 speakers, the test dataset consists of utterances from 10 speakers. The length of each utterance ranges between about 3 and 9 seconds, the data is labelled 1 if the sentence "Hey Snips!" was uttered and 0 if not. Visit the Dataset Repository and fill out the form to proceed with the dataset download. It is roughly 8.4GB.

Load the dataset

After downloading, extract the dataset folder `hey_snips_research_6k_en_train_eval_clean_ter` and put it in the same folder as the one where you have 'lab3.ipynb'.

Open Jupyter Notebook through Anaconda Prompt.

```
> jupyter notebook
```

Open 'lab4.ipynb' on the notebook. Then, execute the first code block to import the required packages and the json files of the dataset.

```
import json, random
import numpy as np
from scipy.io import wavfile
import tensorflow as tf
from tqdm import tqdm
import os
import zipfile, tempfile
import simpleaudio as sa

DataSetPath = "hey_snips_research_6k_en_train_eval_clean_ter/"

with open(DataSetPath+"train.json") as jsonfile:
    traindata = json.load(jsonfile)

with open(DataSetPath+"test.json") as jsonfile:
    testdata = json.load(jsonfile)
```

This is an example of how a data sample looks like. Information from the Dataset Repository.

```
{
  "id": "40084ea8-c576-4dba-a20b-fbda61f1de7d"
  "is_hotword": 1,
  "worker_id": 12,
  "duration": 1.86,
  "audio_file_path": "audio_files/40084ea8-c576-4dba-a20b-fbda61f1de7d.wav",
}
```

- `id`: A unique identifier located in the audio_files subdirectory of the dataset
- `is_hotword`: A 1 if the sample is a hotword and a 0 if not. This is what we are actually after in this classification model. Thus it acts as the label.

- `worker_id`: The unique identifier of the contributor - note that worker ids are not consistent across datasets 1 and 2 as they are re-generated for each one of them.
- `duration`: The duration of the audiofile in seconds.
- `audio_file_path`: The filepath to the respective audio sample.

But now we need to bring it into a format that we can actually learn in our machine learning model. So lets look at how that conversion looks like:

1. We read the .wav file under the `audio_path`, this returns the sample rate (in samples/sec) and data from an LPCM WAV file. For more information see the scipy documentation.
2. We zero-stuff the training and test samples so we have a constant slice length. As in the initial image.
3. Now we segment the slices. For simplifying this exercise, we will choose the segment overlap to be 0. We might also want our segments to have a certain length - Powers of 2 will enable us to do faster FFTs.

For time reasons we will also not load all training and test samples, but a smaller subset of them.

Preprocessing

We will load the data samples and slice them. Before proceeding, let's load a sample and hear it. Don't forget to turn on your speaker!

Execute the following code block:

```
testsize = 100 # Number of loaded testing samples
totalSliceLength = 10 # Length to stuff the signals to, given in seconds

fs = 16000 # Sampling rate of the samples
segmentLength = 1024 # Number of samples to use per segment

sliceLength = int(totalSliceLength * fs / segmentLength)*segmentLength

rand_idx = random.randrange(0, testsize)
fs, test_sound_data =
wavfile.read(DataSetPath+testdata[rand_idx]['audio_file_path'])
_x_test = test_sound_data.copy()
_x_test.resize(sliceLength)
_x_test = _x_test.reshape((-1,int(segmentLength)))
_x_test.astype(np.float32)
label = testdata[rand_idx]['is_hotword']

#Play the .wav file
wave_obj =
sa.WaveObject.from_wave_file(DataSetPath+testdata[rand_idx]['audio_file_path'])
play_obj = wave_obj.play()
play_obj.wait_done()

print('Data: ', _x_test) #Notice the zero stuffing
print('Label: ', label)
```

Now we wrap the previous preprocessing step in a helper function and perform it on a subset of the dataset. Execute the next code block:

```
def load_data():
    x_train_list = []
    y_train_list = []

    x_test_list = []
    y_test_list = []

    totalSliceLength = 10 # Length to stuff the signals to, given in seconds

    trainsize = 1000 # Number of loaded training samples
    testsize = 100 # Number of loaded testing samples


    fs = 16000 # Sampling rate of the samples
    segmentLength = 1024 # Number of samples to use per segment

    sliceLength = int(totalSliceLength * fs / segmentLength)*segmentLength

    for i in tqdm(range(trainsize)):
        fs, train_sound_data =
wavfile.read(DataSetPath+traindata[i]['audio_file_path']) # Read wavfile to extract
amplitudes

        _x_train = train_sound_data.copy() # Get a mutable copy of the wavfile
        _x_train.resize(sliceLength) # Zero stuff the single to a length of
sliceLength
        _x_train = _x_train.reshape(-1,int(segmentLength)) # Split slice into
Segments with 0 overlap
        x_train_list.append(_x_train.astype(np.float32)) # Add segmented slice to
training sample list, cast to float so librosa doesn't complain
        y_train_list.append(traindata[i]['is_hotword']) # Read label

    for i in tqdm(range(testsize)):
        fs, test_sound_data =
wavfile.read(DataSetPath+testdata[i]['audio_file_path'])
        _x_test = test_sound_data.copy()
        _x_test.resize(sliceLength)
        _x_test = _x_test.reshape((-1,int(segmentLength)))
        x_test_list.append(_x_test.astype(np.float32))
        y_test_list.append(testdata[i]['is_hotword'])

    x_train = tf.convert_to_tensor(np.asarray(x_train_list))
    y_train = tf.keras.utils.to_categorical(np.asarray(y_train_list), num_classes=2)

    x_test = tf.convert_to_tensor(np.asarray(x_test_list))
    y_test = tf.keras.utils.to_categorical(np.asarray(y_test_list), num_classes=2)
```

```
    return x_train, y_train, x_test, y_test
```

Feature extraction

The state-of-the art for the convolutional approach in speech recognition is structured around MFCC features, which is a non-linear mapping of the frequency spectrum to considerably fewer features. It is an effective feature extractor for audio classification. It converts conventional frequency to Mel scale, which is a logarithmic transformation that takes into account human perception at different frequencies. As MFCC has proven to be a very good feature for such audio tasks, there is an integrated function for it in TensorFlow, which we'll be using here. If you'd like more details, visit the [TensorFlow Documentation](#).

Define the feature extraction function, load the data, and compute the MFCC features by executing the next code blocks:

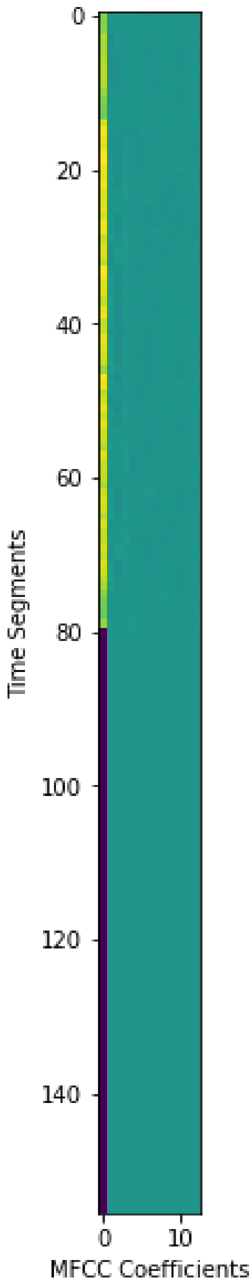
```
def compute_mfccs(tensor):
    sample_rate = 16000.0
    lower_edge_hertz, upper_edge_hertz, num_mel_bins = 80.0, 7600.0, 80
    frame_length = 1024
    num_mfcc = 13

    stfts = tf.signal.stft(tensor, frame_length=frame_length,
frame_step=frame_length, fft_length=frame_length)
    spectrograms = tf.abs(stfts)
    spectrograms = tf.reshape(spectrograms,
(spectrograms.shape[0],spectrograms.shape[1],-1))
    num_spectrogram_bins = stfts.shape[-1]
    linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(
      num_mel_bins, num_spectrogram_bins, sample_rate, lower_edge_hertz,
      upper_edge_hertz)
    mel_spectrograms = tf.tensordot(spectrograms, linear_to_mel_weight_matrix, 1)
    log_mel_spectrograms = tf.math.log(mel_spectrograms + 1e-6)
    mfccs =
tf.signal.mfccs_from_log_mel_spectrograms(log_mel_spectrograms)[..., :num_mfcc]
    return tf.reshape(mfccs, (mfccs.shape[0],mfccs.shape[1],mfccs.shape[2],-1))
```

```
x_train, y_train, x_test, y_test = load_data()

x_train_mfcc = compute_mfccs(x_train)
x_test_mfcc = compute_mfccs(x_test)

print('Training Data Shape: ', x_train_mfcc.shape)
print('Testing Data Shape: ', x_test_mfcc.shape)
```

Visualize the features

You can visualize a MFCC “image” by executing the next code block:

```
import matplotlib
import matplotlib.pyplot as plt
matplotlib.rcParams['figure.figsize'] = (10,10)

mfcc_data=x_test_mfcc[rand_idx,:,:,0].numpy()
fig, ax = plt.subplots()
ax.imshow(mfcc_data)
ax.set_xlabel("MFCC Coefficients")
ax.set_ylabel("Time Segments")
plt.show()
```



CNN Classification

Hyperparameter and input scaling

Define the hyperparameters for the CNN training, such as the batch size, the number of epochs, the learning rate, and apply a scaling to the MFCC input features by executing the following code block:

```
batchSize = 10
epochs = 20
lr = 0.001

train_set = (x_train_mfcc/512 + 0.5)
train_labels = y_train

test_set = (x_test_mfcc/512 + 0.5)
test_labels = y_test
```

CNN Architecture

We are using a CNN to architecture to fuse information over multiple channels efficiently. Such an architecture has proven to be very effective in such a problem setting. Define the architecture by executing the following code block:

```
model = tf.keras.models.Sequential()

input_shape = (train_set.shape[1],train_set.shape[2],train_set.shape[3])
model.add(tf.keras.layers.InputLayer(input_shape=input_shape))
model.add(tf.keras.layers.Conv2D(filters=3,kernel_size=(3,3),padding="same",
activation='relu', input_shape=input_shape))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.Conv2D(filters=16,kernel_size=(3,3),strides=(2,2),padding=
'same', activation='relu'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.MaxPool2D((2,2)))

model.add(tf.keras.layers.Conv2D(filters=32,kernel_size=(3,3),strides=(2,2),padding=
'same', activation='relu'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.MaxPool2D((2,2)))

model.add(tf.keras.layers.Conv2D(filters=48,kernel_size=(3,3),padding='same',strides
=(2,2), activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
```

```
model.add(tf.keras.layers.GlobalAveragePooling2D())

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(8, activation='relu'))

model.add(tf.keras.layers.Dense(2, activation='softmax'))

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 156, 13, 3)	30
batch_normalization (BatchNo	(None, 156, 13, 3)	12
conv2d_1 (Conv2D)	(None, 78, 7, 16)	448
batch_normalization_1 (Batch	(None, 78, 7, 16)	64
max_pooling2d (MaxPooling2D)	(None, 39, 3, 16)	0
conv2d_2 (Conv2D)	(None, 20, 2, 32)	4640
batch_normalization_2 (Batch	(None, 20, 2, 32)	128
max_pooling2d_1 (MaxPooling2	(None, 10, 1, 32)	0
conv2d_3 (Conv2D)	(None, 5, 1, 48)	13872
batch_normalization_3 (Batch	(None, 5, 1, 48)	192
global_average_pooling2d (Gl	(None, 48)	0
flatten (Flatten)	(None, 48)	0
dense (Dense)	(None, 8)	392
dense_1 (Dense)	(None, 2)	18

Total params: 19,796
Trainable params: 19,598
Non-trainable params: 198

Model compilation and training

Compile the model by defining the loss functions, the optimizer, and the metrics to be used. We then train the model. Execute the next code block:

```
model.compile(loss='categorical_crossentropy',
optimizer=tf.keras.optimizers.Adam(learning_rate=lr), metrics=['accuracy'])
model.fit(train_set, y_train, batchSize, epochs, validation_data=(test_set, y_test))
```

Epoch 1/20
100/100 [==============================] - 7s 22ms/step - loss: 0.4341 - accuracy: 0.9100 - val_loss: 0.5034 - val_accuracy: 0.8000
Epoch 2/20
100/100 [==============================] - 1s 13ms/step - loss: 0.1842 - accuracy: 0.9410 - val_loss: 0.5778 - val_accuracy: 0.8000
Epoch 3/20
100/100 [==============================] - 1s 13ms/step - loss: 0.1371 - accuracy: 0.9400 - val_loss: 0.7783 - val_accuracy: 0.8000
Epoch 4/20
100/100 [==============================] - 1s 13ms/step - loss: 0.1018 - accuracy: 0.9540 - val_loss: 0.5329 - val_accuracy: 0.8000
Epoch 5/20
100/100 [==============================] - 1s 14ms/step - loss: 0.1003 - accuracy: 0.9560 - val_loss: 0.5746 - val_accuracy: 0.8000
Epoch 6/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0844 - accuracy: 0.9640 - val_loss: 0.1327 - val_accuracy: 0.9800
Epoch 7/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0787 - accuracy: 0.9690 - val_loss: 0.1515 - val_accuracy: 0.9000
Epoch 8/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0660 - accuracy: 0.9760 - val_loss: 3.1921 - val_accuracy: 0.8000
Epoch 9/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0515 - accuracy: 0.9820 - val_loss: 11.4461 - val_accuracy: 0.8000
Epoch 10/20
100/100 [==============================] - 1s 14ms/step - loss: 0.0605 - accuracy: 0.9760 - val_loss: 4.0709 - val_accuracy: 0.8000
Epoch 11/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0520 - accuracy: 0.9850 - val_loss: 4.6924 - val_accuracy: 0.8000
Epoch 12/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0501 - accuracy: 0.9840 - val_loss: 3.7893 - val_accuracy: 0.8000
Epoch 13/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0407 - accuracy: 0.9870 - val_loss: 3.6819 - val_accuracy: 0.2300
Epoch 14/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0445 - accuracy: 0.9820 - val_loss: 1.9372 - val_accuracy: 0.8000
Epoch 15/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0311 - accuracy: 0.9870 - val_loss: 2.1957 - val_accuracy: 0.8000
Epoch 16/20
100/100 [==============================] - 1s 14ms/step - loss: 0.0449 - accuracy: 0.9860 - val_loss: 89.5371 - val_accuracy: 0.8000
Epoch 17/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0408 - accuracy: 0.9880 - val_loss: 63.7609 - val_accuracy: 0.8000
Epoch 18/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0400 - accuracy: 0.9860 - val_loss: 0.1507 - val_accuracy: 0.9300
Epoch 19/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0257 - accuracy: 0.9930 - val_loss: 0.0521 - val_accuracy: 0.9700
Epoch 20/20
100/100 [==============================] - 1s 13ms/step - loss: 0.0258 - accuracy: 0.9880 - val_loss: 25.1634 - val_accuracy: 0.8000

Model evaluation

Evaluate the trained model on the remaining data by executing the following line of code:

```
score = model.evaluate(test_set, y_test, batch_size=batchSize)
```

10/10 [==============================] - 0s 6ms/step - loss: 25.1634 - accuracy: 0.8000

Save the model

Once you are satisfied with the performance of your model, you can save it by executing the following line of code:

```
model.save('MFCCmodel.h5')
```

The ‘MFCCmodel.h5’ can then be deployed on the MCU using Cube.AI as you’ve learned in lab 2.