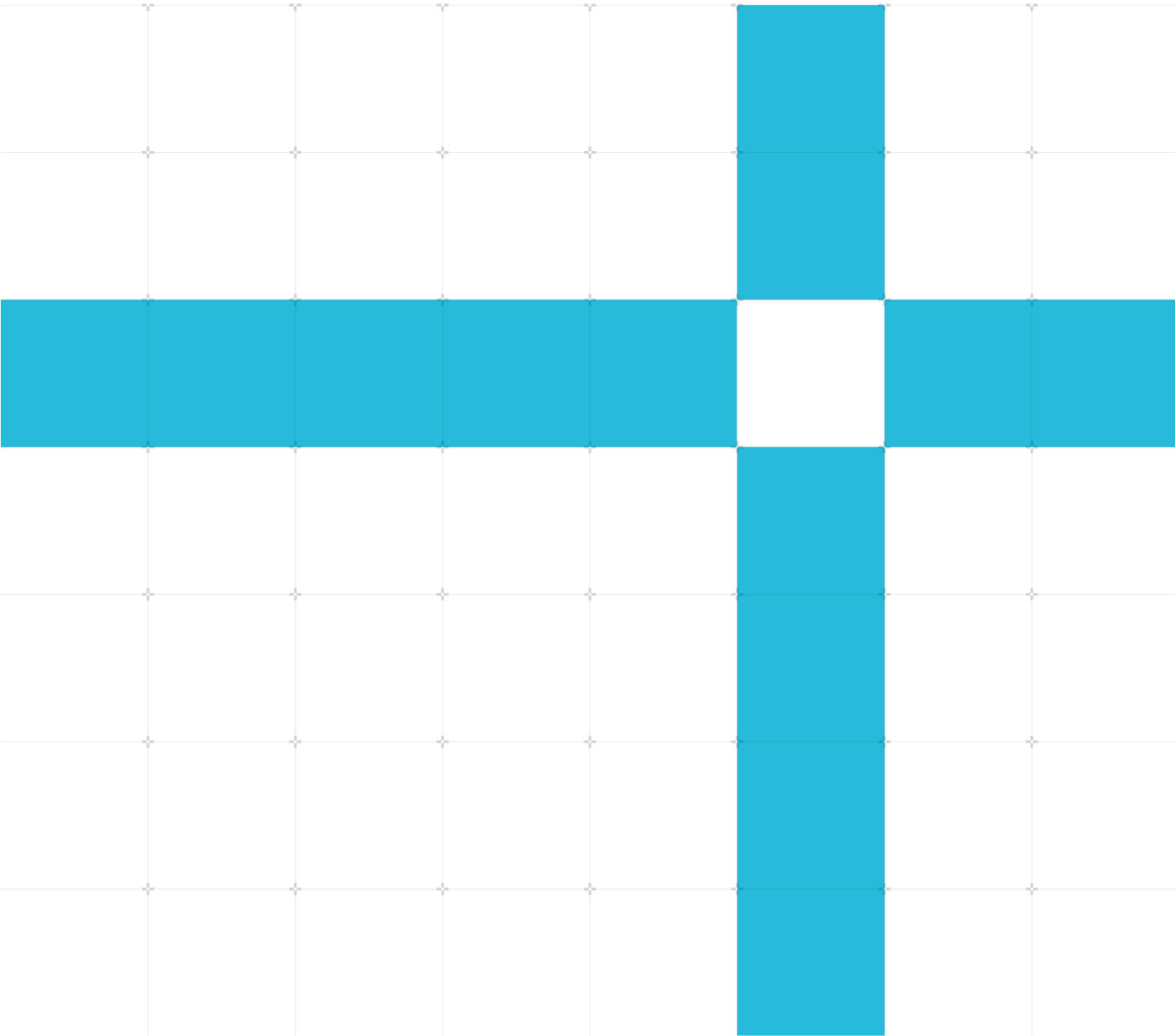
arm

Automatic speech recognition with wav2letter using PyArmNN

Non-Confidential

Issue 01

Document ID: 102470



Automatic speech recognition with wav2letter using PyArmNN

Release information

Document history

Issue	Date	Confidentiality	Change
01	June 2, 2021	Non-Confidential	First release

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email **terms@arm.com**.

Contents

1 Overview

This guide reviews a sample application that performs Automatic Speech Recognition (ASR) with the **PyArmNN API**. The guide explains how the speech recognition application works, then gives instructions for running the application on a Raspberry Pi.

1.1 Before you begin

This guide is for experienced ML developers and assumes an understanding of most ML concepts.

To complete this guide, you must first install Arm NN on a Raspberry Pi. From Arm NN 20.11, we provide Debian packages for Ubuntu 64-bit. To use these packages, download a supported 64-bit Debian Linux Operating System. Arm NN supports 64bit Ubuntu versions 20.04, 20.10 and 21.04. This guide has been tested on Ubuntu 64bit 20.10. However, this version of Linux requires a Raspberry Pi 4 with at least 4GB of RAM, and runs better on 8GB. If you do not need a full desktop environment or more long-term support, you can install 20.04 Server, as explained on **linuxhint**.

To run the example in this guide:

1. Install Ubuntu. See installation instructions for the **Raspberry Pi4 on the Ubuntu website**.
2. Download and install the required packages.

   Install the Arm NN Personal Package Archive (PPA) and software-properties-common with the following commands:

   ```
   sudo apt install software-properties-common
   sudo add-apt-repository ppa:armnn/ppa
   sudo apt update
   export ARMNN_MAJOR_VERSION=23
   sudo apt-get install -y python3-pyarmnn libarmnn-cpuacc-
   backend${ARMNN_MAJOR_VERSION}
       libarmnn-cpuref-backend${ARMNN_MAJOR_VERSION}
   ```

   In the `apt-get` command shown in the preceding script, `23` is the `libarmnn` version. You can replace it with the latest supported version, as listed on our **GitHub repository**.
   These packages provide the TensorflowLite parser for Arm NN, which is what this guide uses.
3. Install Git:

   ```
   sudo apt-get install git
   ```
4. Install Git Large File Storage (LFS):

   ```
   sudo apt-get install git-lfs
   ```

5. Initialize Git LFS:

```
git lfs install
```

6. Install Pip:

```
sudo apt install python-3-pip
```

2 Wav2Letter speech recognition

This section of the guide explains how the Wav2Letter application process a WAV file and generates a transcript of human speech.

The application takes a WAV file of human speech as input, then uses the Mel-Frequency Cepstral Coefficients (MFCC) class to generate the input for the model. MFCC converts audio into waveforms, which are easier for a convolutional neural network to parse. The model assigns each chunk of converted audio the correct phoneme, which the application then converts into a correctly spelled word.

The Wav2Letter application performs the following steps:

1. Configure the application:
    a. Point the application to the sample audio, model, and label file.
    b. Build the dictionary of phonemes to spelling.
2. Create a network: Convert the Network to be run by Arm NN.

The application contains an automatic speech recognition pipeline to process the audio file. The pipeline performs the following steps:

1. Feed the audio into the MFCC conversion process.
2. Input the sample into the model for processing.
3. Read output of the phonemes that the model processed.
4. Use the dictionary to convert to the correct spelling.
5. Output the words as string.

2.1 Initialization

The application parses the supplied user arguments, which include a path to an audio file. The application loads the audio file into the `AudioCapture` class, which initializes the audio source. The `ModelParams` class sets the sampling parameters that the model requires.

The `AudioCapture` class captures chunks of audio data from the source. Using Automatic Speech Recognition (ASR) on the audio file, the application creates a generator object to yield blocks of audio data from the file. Each block has a minimum sample size. The number of samples that are required depends on your sample rate and your window size in milliseconds (ms). The following formula is used to calculate the minimum sample size:

```
num_samples_per_inference = ((self.model_input_size - 1) * self.stride) +
self.__mfcc_calc.mfcc_params.frame_len
```

To interpret the inference result of the loaded network, the application loads the labels that are associated with the model. Each label represents a phoneme. The `dict_labels()` function creates a dictionary that is keyed on the classification index at the output node of the model. The values of the dictionary are the characters that correspond to the appropriate phonemes.

2.2 Creating a network

A PyArmNN application must import a graph from a file using an appropriate parser. These parsers are libraries for loading neural networks of various formats into the Arm NN runtime. Arm NN provides parsers for various model file types, including `TFLite`, `TF`, and `ONNX`.

Arm NN supports optimized execution on multiple CPU, GPU, and Ethos-N devices. Before executing a graph, the application uses `IRuntime()` to create a runtime context with default options that are appropriate to the device. We can optimize the imported graph by specifying a list of backends in order of preference, and then implement backend-specific optimizations. Each backend is identified by a unique string: `CpuAcc`, `GpuAcc`, and `CpuRef` represent the accelerated CPU, accelerated GPU, and the CPU reference kernels, respectively.

Arm NN splits the entire graph into subgraphs based on these backends. Each subgraph is then optimized, and the corresponding subgraph in the original graph is substituted with its optimized version.

The `Optimize()` function optimizes the graph for inference, then `LoadNetwork()` loads the optimized network onto the compute device. The `LoadNetwork()` function also creates the backend-specific workloads for the layers and a backend-specific workload factory.

Parsers extract the input information for the network:

- The `GetSubgraphInputTensorNames()` function extracts all the input names.
- The `GetNetworkInputBindingInfo()` function obtains the input binding information of the graph. The input binding information contains all the essential information about the input. This information is a tuple consisting of integer identifiers for bindable layers and tensor information: data type, quantization info, dimension count, and total elements.

To get the output binding information for an output layer, the parser retrieves the output tensor names and calls the `GetNetworkOutputBindingInfo()` function.

For this application, the main point of contact with PyArmNN is through the `ArmnnNetworkExecutor` class, which handles the network creation step for you.

The following code shows the network creation step:

```
# common/network_executor.py

# The provided wav2letter model is in. tflite format so we use TfLiteParser() to import
the graph
```

```
if ext == '.tflite':
    parser = ann.ITfLiteParser()
network = parser.CreateNetworkFromBinaryFile(model_file)
...
# Optimize the network for the list of preferred backends
opt_network, messages = ann.Optimize(
    network, preferred_backends, self.runtime.GetDeviceSpec(), ann.OptimizerOptions()
    )
# Load the optimized network onto the runtime device
self.network_id, _ = self.runtime.LoadNetwork(opt_network)
# Get the input and output binding information
self.input_binding_info = parser.GetNetworkInputBindingInfo(graph_id, input_names[0])
self.output_binding_info = parser.GetNetworkOutputBindingInfo(graph_id, output_name)
```

2.3 Automatic speech recognition pipeline

To extract the MFCCs from a given audio frame, to be used as features for the network, we use the `MFCC` class. MFCCs are the result of computing the dot product of the Discrete Cosine Transform (DCT) matrix and the log of the mel energy.

After extracting the MFCCs that are needed for an inference from the audio data, we compute the first and second MFCC derivatives with respect to time. The computation convolves the derivatives with one-dimensional Savitzky-Golay filters. The MFCCs and the derivatives are concatenated to make the input tensor for the model.

The following code shows the MFCC extraction and derivative computation:

```
# preprocess.py
# Extract MFCC features
log_mel_energy = np.maximum(log_mel_energy, log_mel_energy.max() - top_db)
mfcc_feats = np.dot(self.__dct_matrix, log_mel_energy)
...
# Compute first and second derivatives (delta and delta-delta respectively) by passing
a
# Savitzky-Golay filter as a 1D convolution over the features
for i in range(features.shape[1]):
            idelta = np.convolve(features[:, i], self.__savgol_order1_coeffs, 'same')
            mfcc_delta_np[:, i] = (idelta)
            ideltadelta = np.convolve(features[:, i], self.savgol_order2_coeffs,
'same')
            mfcc_delta2_np[:, i] = (ideltadelta)
# audio_utils.py
# Quantize the input data and create input tensors with PyArmNN
```

```
input_tensor = quantize_input(input_tensor, input_binding_info)
input_tensors = ann.make_input_tensors([input_binding_info], [input_tensor])
```

`ArmnnNetworkExecutor` has already created the output tensors for you.

After creating the workload tensors, the compute device performs inference for the loaded network by using the `EnqueueWorkload()` function of the runtime context.

The following code shows calling the `workload_tensors_to_ndarray()` function to obtain the inference results as a list of `ndarrays`:

```
# common/network_executor.py
status = runtime.EnqueueWorkload(net_id, input_tensors, self.output_tensors)
self.output_result = ann.workload_tensors_to_ndarray(self.output_tensors)
```

The output from the inference must be decoded to obtain the recognized characters from the speech. A simple greedy decoder classifies the results by taking the highest element of the output as a key for the labels dictionary. The value returned is a character. The character is appended to a list, and the list is filtered to remove unwanted characters. The produced string is displayed on the console.

3 Running the application

This section explains how to get and run all the code and models that are required to use the Wav2Letter application. By the end of this section, you will have the output of the model for a WAV file.

3.1 Initializing the project

To initialize the project, do the following:

1. Get the example code **from GitHub**
2. Create a workspace for the project:
```
mkdir workspace && cd workspace
```
3. Clone the Arm NN repository:
```
git clone https://github.com/ARM-software/armnn/
```
4. Check out the Arm NN master:
```
cd armnn && git checkout master
```
5. Navigate to the example folder:
```
cd armnn/python/pyarmnn/examples/speech_recognition
```
6. Install the PortAudio package:
```
$ sudo apt-get install libsndfile1 libportaudio2
```
7. Install the following Python modules:
```
$ pip install -r requirements.txt
```
8. Navigate back to the example folder:
```
cd ~ && cd workspace
```
9. Clone the Model Zoo repository:
```
git clone https://github.com/ARM-software/ML-zoo
```
10. Copy the `tflite_int8` model file to the example application:
```
cd armnn/python/pyarmnn/examples/speech_recognition

cp -r ~/workspace/ML-zoo/models/speech_recognition/wav2letter/tflite_int8 .
```

3.2 Get an audio file for the example

To run this example, you need a WAV file. For the application to process the audio file correctly, the audio file must have a sample rate of 16000Hz. We have provided an audio file for use with this guide, available from the ArmNN repository.

3.3 Run the example

The following command runs the Python script with the location of the audio, model, and label files:

```
python run_audio_file.py --audio_file_path <path/to/your_audio> --model_file_path
<path/to/your_model> --labels_file_path <path/to/your_labels>
```

The label file is the `tests/testdata/wav2letter_labels.txt` file in your example repository.

Additional script options are as follows:

a) To run with a specific backend, use `--preferred_backends`. You can enter multiple values in preference order, separated by whitespace. For example, pass `CpuAcc CpuRef` for `["CpuAcc", "CpuRef"]`.

   The available values are:

   - `CpuAcc` for the CPU backend
   - `GpuAcc` for the GPU backend
   - `CpuRef` for the CPU reference kernels

b) To see all available options, use `--help`

Here is an example, with the file `quick_brown_fox_16000khz.wav`:

```
python3 run_audio_file.py --audio_file_path tests/testdata/quick_brown_fox_16000khz.wav
--model_file_path tflite_int8/wav2letter_int8.tflite --labels_file_path
tflite_int8/labels.txt
```

When the script finishes, the output is displayed in the terminal window. This example produces the following output:

```
the quick brown fox juhmped over the llazy dag
```

4 Related information

Here are some resources related to the material in this guide:

- **Accelerated ML inference on Raspberry Pi with PyArmNN**
- **AI and machine learning content from Arm**
- **Arm Community machine learning blog**
- **Object recognition with Arm NN and Raspberry Pi**
- **Wav2Letter: an End-to-End ConvNet-based Speech** – This research paper provides full details on how this model works.
- **Mel-frequency cepstrum Wikipedia**

Other Arm resources:

- **Arm Community** - ask development questions, and find articles and blogs on specific topics from Arm experts.

5 Next steps

Now that you understand how to perform automatic speech recognition with PyArmNN, you can take control and create your own application. We suggest implementing your own network, which you can do by updating the parameters of `ModelParams` and `MfccParams` to match your custom model. The `ArmnnNetworkExecutor()` class handles the network optimization and loading for you.

To improve the accuracy of the generated output sentences, an important step is to provide cleaner data to the network. You can do this by including more preprocessing steps, like noise reduction of your audio data.

In this application, we used a greedy decoder to decode the integer-encoded output. However, you can achieve better results by implementing a beam search decoder. You may even try adding a language model at the end to try to correct any spelling mistakes the model may produce.