Automatic Speech Recognition with Wav2Letter using PyArmNN and Debian Packages

Version 22.11

Tutorial

Non-Confidential

Issue 04

102603_2211_04_en

Automatic Speech Recognition with Wav2Letter using PyArmNN and Debian Packages

Tutorial

Release information

Document history

Issue	Date	Confidentiality	Change
2108-01	9 August 2021	Non-Confidential	First release for 21.08
2108-02	20 October 2021	Non-Confidential	Second release for 21.08
2108-03	5 November 2021	Non-Confidential	Third release for 21.08
2211-04	12 January 2023	Non-Confidential	First release for 22.11

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` <u>`underline`</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.

Convention	Use
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
AI and machine learning content from Arm	-	Non-Confidential
Arm Community	-	Non-Confidential
Arm NN Github	-	Non-Confidential
Arm NN Product Documentation	-	Non-Confidential
Arm's Machine Learning blog	-	Non-Confidential
Object recognition with Arm NN and Raspberry Pi	102274	Non-Confidential
PyArmNN API	-	Non-Confidential
PyArmNN repository	-	Non-Confidential

Non-Arm resources	Document ID	Organization
The research paper from Facebook AI Research	–	Facebook AI Research

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.

- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. 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 or an Odroid N2 Plus.

2.1 Overview of PyArmNN

The Arm NN library optimizes neural networks for Arm hardware. Arm NN provides significant performance increases compared with other frameworks when running on an Arm Cortex-A.

PyArmNN is a Python package that provides a wrapper around the C++ Arm NN API. PyArmNN does not implement any computational kernels. PyArmNN delegates all operations to Arm NN, meaning you access the power of Arm NN from Python.

PyArmNN enables rapid development allowing you to produce and test prototypes in minutes.

Both PyArmNN and Arm NN use parsers to import models from different external frameworks. Available parsers include the following:

- TensorFlow Lite
- ONNX
- PyTorch via ONNX

The parser converts the imported model into an Arm NN network graph, which can be optimized for Arm hardware.

You can find more information about PyArmNN and example code for PyArmNN in the Arm Software GitHub repository.

2.2 Speech recognition

Speech recognition is the process of a program recognizing and translating spoken language into a written format or a format understood by an application.

Many speech recognition applications can be broken down into two steps. The first step is the pre-processing of the raw audio data which usually involves applying some signal processing to the audio data such as a Fast-Fourier transform. The second step is a model which takes the processed data as input, this model is often a neural network.

There are some recent advancements in neural networks which use the raw data as input and perform all the analysis. However, these networks tend to be large as they have all the feature extraction and analysis built within them.

Speech recognition has many use-cases, these mainly fall under the umbrella of hands-free dictation context. Some examples include enabling car drivers to perform tasks without taking their hands off the driving wheel, automated caption generation making media more accessible, and virtual assistants to help with day-to-day tasks.

2.3 Before you begin

This guide requires installing PyArmNN on your device. From Arm NN 20.11, we provide Debian packages for Ubuntu 64-bit. These packages are the easiest way to install PyArmNN and are what we recommend you use. To use these packages, you must download a supported 64-bit Debian Linux operating system.

We provide installation steps for different devices in the Device-specific installation section.

3. Device-specific installation

In this section, we provide installation steps to set up your Raspberry Pi and Odroid N2 Plus devices.

3.1 Install on Raspberry Pi

The following steps cover setting up your Raspberry Pi for this example.

Before you begin

To run the example in this guide, you must install a 64-bit operating system on your Raspberry Pi. Ubuntu has official support for 64-bit 20.04 and 21.04 on the Raspberry Pi.

Procedure

1. Install Ubuntu. See installation instructions for the Raspberry Pi4 on the Ubuntu website. This guide has been tested on Ubuntu 20.04 and Ubuntu 21.04.
2. Download and install the required packages. Add the PPA to your sources from the software-properties-common package as shown in the following commands:

   ```
   sudo apt install software-properties-common
   sudo add-apt-repository ppa:armnn/ppa
   sudo apt update
   sudo apt-get install -y python3-pyarmnn armnn-latest-cpu armnn-latest-ref
   ```

   This command installs the latest version of Arm NN with the CpuAcc and CpuRef backends. It also provides the Arm NN TensorFlow Lite parser, which this guide uses. The command also installs PyArmnn which contains Python bindings for Arm NN. PyArmnn allows you to use Arm NN in Python code.
3. Enter the following commands to install Git and Git Large File Storage to download models from the model zoo:

   ```
   sudo apt-get install git git-lfs
   git lfs install --skip-repo
   ```
4. Install pip using the following command:

   ```
   sudo apt install python3-pip
   ```

3.2 Install on Odroid N2 Plus

The Odroid has both an Arm Cortex-A CPU and an Arm Mali GPU. This means you can configure your setup to utilize both the CPU and GPU. The following steps cover running the example in this guide.

Procedure

1. Install Ubuntu Mate 20.04. Use the official Ubuntu Mate images on the Odroid website.

2. Do a basic `apt update` and `apt upgrade`. The following commands ensure everything is up to date.

```
sudo apt update
sudo apt upgrade
```

3. Install the OpenCL drivers to include GPU support using the following commands. The Ubuntu Mate images come with official support. However, some users have reported problems. If you have already installed these drivers, you can skip this step.

```
mkdir temp-folder
cd temp-folder
sudo apt-get install clinfo ocl-icd-libopencl1
sudo apt-get download mali-fbdev
ar -xv mali-fbdev_*
tar -xvf data.tar.xz
sudo rm /usr/lib/aarch64-linux-gnu/libOpenCL.so*
sudo cp -r usr/* /usr/
sudo mkdir /etc/OpenCL
sudo mkdir /etc/OpenCL/vendors/
sudo bash -c 'echo "libmali.so" > /etc/OpenCL/vendors/mali.icd'
```

4. Check your OpenCL installation by running the `clinfo` command.

```
clinfo
```

5. Download and install the required Arm NN packages to run the example using the following command:

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:armnn/ppa
sudo apt update

sudo apt-get install -y python3-pyarmnn armnn-latest-all
```

   This guide uses the Arm NN TensorFlow Lite parser which is included in the `armnn-latest-all` package. The package also provides CPU and GPU accelerators. The command also installs PyArmnn which contains Python bindings for Arm NN. PyArmnn allows you to use Arm NN in Python code.

6. Install Git and Git Large File Storage to download models from the model zoo using the following commands:

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

7. Install pip using the following command:

```
sudo apt install python3-pip
```

   We install pip, so we can later install Python packages required to run the example.

4. Running the application

This section explains how to retrieve and run all the code and models that you require to use the Automatic Speech Recognition application. By the end of this section, you should have a text output from a WAV file.

4.1 Initializing the project

The following steps cover initializing the project on your device.

About this task

Get the example code from GitHub.

Procedure

1. Create a workspace for the project with the following command:

```
mkdir ~/workspace && cd ~/workspace
```

2. Clone the Arm NN repository with the following command:

```
git clone https://github.com/ARM-software/armnn/
```

3. Navigate to the example folder with the following command:

```
cd python/pyarmnn/examples/speech_recognition
```

4. Install the PortAudio package with the following command:

```
sudo apt-get install libsndfile1 libportaudio2
```

5. Install the required Python modules with the following command:

```
pip3 install -r requirements.txt
```

6. To get the model from the Arm Model Zoo, navigate back to the example folder with the following command:

```
cd ~/workspace
```

7. Clone the Model Zoo repository with the following command:

```
git clone https://github.com/ARM-software/ML-zoo
```

8. Copy the model file to the example application with the following commands:

```
cd armnn/python/pyarmnn/examples/speech_recognition
cp -r ~/workspace/ML-zoo/models/speech_recognition/wav2letter/tflite_int8 .
```

4.2 Get an audio file for the example

To run this example, you need a WAV file. We have provided an audio file for use with this guide, available from the PyArmNN repository.

To download the file from the PyArmNN repository, use `wget` with the download link of your file. For example, to get the file quick_brown_fox_16000khz.wav use the following command:

```
wget
https://git.mlplatform.org/ml/armnn.git/tree/python/pyarmnn/examples/
speech_recognition/tests/testdata/quick_brown_fox_16000khz.wav
```

4.3 Run the example

The following steps cover running the example on your device.

Procedure

1. To run this example, use the following command:

   ```
   python3 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.

2. Optional flags: use `--preferred_backends` to run with a specific backend. You can enter multiple values in preference order, separated by whitespace. For example, pass `CpuAcc CpuRef` for ["CpuAcc", "CpuRef"]. To see all available options, use `--help`
   The available values are:

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

   The following 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 tests/
   testdata/wav2letter_labels.txt --preferred_backends CpuAcc CpuRef
   ```

Results

After the script has finished running, the output is displayed.

5. Code deep dive

This section of the guide explains how the Wav2Letter application processes 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. Initialization:
    a. Point the application to the sample audio, model, and label file.
    b. Build the dictionary of phonemes to spelling.
2. Creating a network: Convert the network to be run by Arm NN.
3. Automatic speech recognition pipeline:
    a. Feed the audio into the conversion process (MFCC).
    b. Input the sample into the model.
    c. The model processes the sample.
    d. Read output of the phonemes the model processed.
    e. Use the dictionary to convert to correct spelling.
    f. Output the words as strings.

5.1 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.

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.

5.2 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 which 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. This information includes 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)
```

5.3 Automatic speech recognition pipeline

The network uses the Mel-Frequency Cepstral Coefficients (MFCCs) of a given audio frame to convert speech to text. The MFCC class performs this speech to text conversion. 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 all the MFCCs that the application needs 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])
```

Note

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.

6. 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.

An important step to improve the accuracy of the generated output sentences is to provide cleaner data to the network. You can improve the accuracy 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 can even try adding a language model at the end to try to correct any spelling mistakes the model produces.

Appendix A Revisions

This appendix describes the technical changes between released issues of this book.

Table A-1: First release for version 21.08

Change	Location
First release	—

Table A-2: Second release for version 21.08

Change	Location
Updates command in step 2	Install on Raspberry Pi
Updates command in step 5	Install on Odroid N2 Plus
Adds additional resources	[Related information]

Table A-3: Third release for version 21.08

Change	Location
Removes non-inclusive language	Initializing the project

Table A-4: First release for version 22.11

Change	Location
Fixes installation command	Install on Odroid N2 Plus