



Object detection with YOLOv3 using PyArmNN and Debian packages

Version 21.08

Tutorial

Non-Confidential

Copyright © 2018, 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 02

102557_2108_02_en



Object detection with YOLOv3 using PyArmNN and Debian packages Tutorial

Copyright © 2018, 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	25 October 2018	Non-Confidential	First release for version 1.01
2108-01	1 October 2021	Non-Confidential	First release for version 21.08
2108-02	18 October 2021	Non-Confidential	Second release for version 21.08

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2018, 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web address

developer.arm.com

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.....	7
1.1 Conventions.....	7
1.2 Additional reading.....	8
1.3 Feedback.....	8
1.4 Other information.....	9
2 Overview.....	10
3 About PyArmNN.....	11
4 About object detection.....	12
5 Device specific installation.....	13
5.1 Raspberry Pi installation.....	13
5.2 Odroid N2 installation.....	14
6 Running the example.....	16
6.1 Setting up your workspace.....	16
6.2 Select and download the appropriate model.....	16
6.3 Run the application.....	17
7 Object detection application code overview.....	19
7.1 Import required Python package dependencies.....	20
7.2 Read from the video source.....	20
7.3 Create a video writer object for the output.....	20
7.4 Create the parser and import the network file.....	21
7.5 Optimize the graph for the compute device.....	21
7.6 Load the network.....	22
7.7 Create input and output binding information.....	22
7.8 Implement a class for the network graph.....	23
7.9 Pre-process the captured frame.....	24
7.10 Process the output into a list of detections.....	25
7.11 Prepare labels.....	27
7.12 Draw the bounding boxes.....	27

7.13 Execute inference on each frame of the input video.....	28
7.14 Release the input and output video.....	29
7.15 Run the application.....	29
8 Next steps.....	30
9 Related information.....	31
A Revisions.....	32

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
<i>italic</i>	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
monospace	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
<i>monospace italic</i>	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
monospace <u>underline</u>	Denotes 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: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the <i>Arm Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .
 Caution	This represents a recommendation which, if not followed, might lead to system failure or damage.
 Warning	This represents a requirement for the system that, if not followed, might result in system failure or damage.
 Danger	This represents a requirement for the system that, if not followed, will result in system failure or damage.

Convention	Use
 Note	This represents an important piece of information that needs your attention.
 Tip	This represents a useful tip that might make it easier, better or faster to perform a task.
 Remember	This is a reminder of something important that relates to the information you are reading.

1.2 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-2: Arm publications

Document Name	Document ID	Licensee only
None	-	-

1.3 Feedback

Arm welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title Object detection with YOLOv3 using PyArmNN and Debian packages Tutorial.
- The number 102557_2108_02_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.



Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

1.4 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 object detection with PyArmNN. This guide contains the following:

- An introduction to PyArmNN and the parsers that are compatible with PyArmNN.
- An Introduction to object detection and the differences between object detection and image detection.
- How to install the packages on Raspberry Pi and Odroid N2.
- How to run the example and an overview of the code used in running the example.

This guide is relevant to beginners and experienced application developers.

Before you begin

To work through this guide, you need a device with the following:

- An Armv7-A or Armv8-A processor
- Optionally, an Arm Mali GPU using the OpenCL driver or an Arm Ethos NPU.

This guide includes installation steps for both the Raspberry Pi and Odroid N2+.

3 About PyArmNN

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

PyArmNN is a Python package that provides a wrapper around the C++ Arm NN API. PyArmNN itself does not implement any computational kernels but instead delegates all operations to Arm NN. This means that you can access the power of Arm NN from Python.

PyArmNN enables rapid development which allows you to produce and test prototypes quickly.

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 then be optimized for Arm hardware.

You can find more information about PyArmNN and example code for PyArmNN in the [Arm Software GitHub repository](#).

4 About object detection

Object detection is the process of identifying real-world objects, like people, cars, or bottles, in still images or videos.

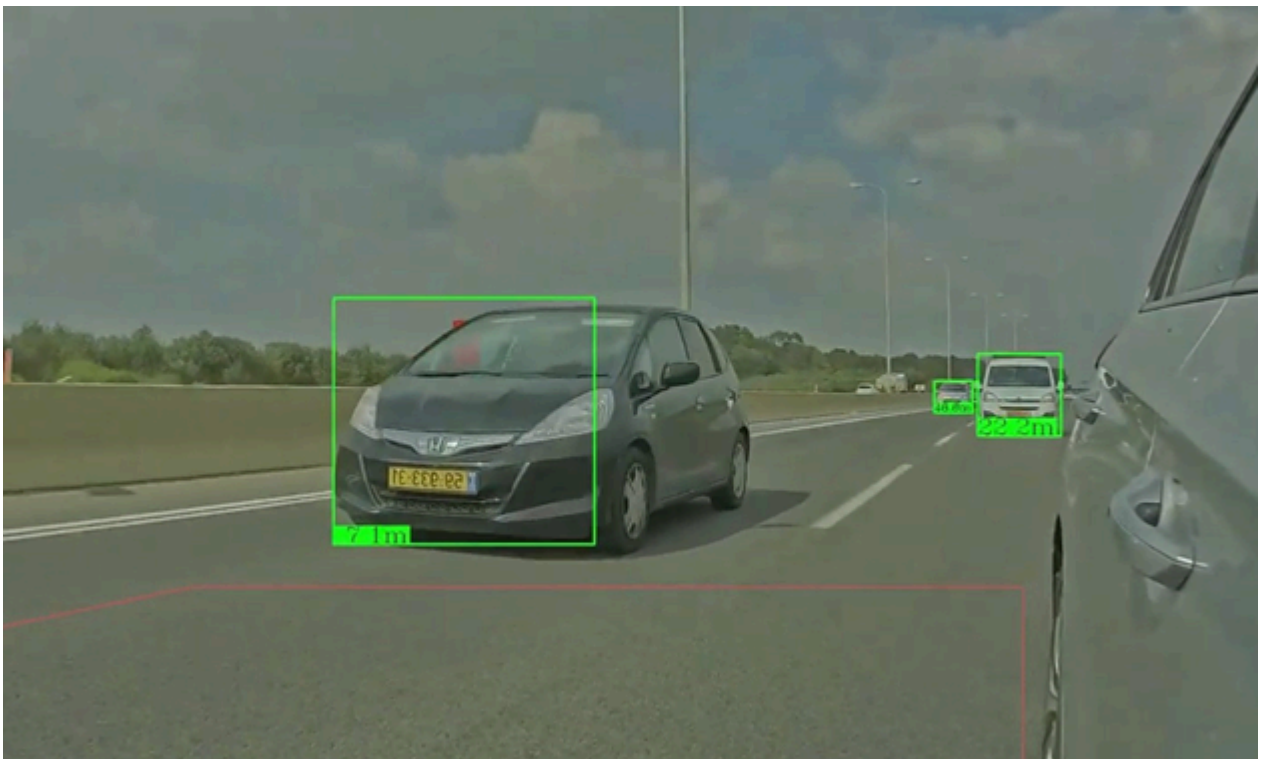
You can think of object detection as an extension to image classification. In image classification, the goal is to classify the object shown in an image, for example to identify a photo of an animal as either a dog or a cat. Object detection identifies and classifies multiple objects in a single image, drawing a bounding box around the detected objects.

Image classification answers the question “what object does this image show?” Object detection answers the questions “what objects are present in this image, and where are they?”.

Similarly, with image classification, an object detection model can only locate and classify objects that it has learned about from the training set.

The following image shows an example of object detection:

Figure 4-1: Object detection



5 Device specific installation

To complete this guide, you must first install Arm NN on your device. From Arm NN 20.11, we provide Debian packages for Ubuntu 64-bit. To use these packages, you must download a supported 64-bit Debian Linux Operating System. Arm NN supports 64bit Ubuntu versions 20.04, 20.10, and 21.04.

You can use Advanced Package Tool (APT) to install Arm NN and PyArmNN on your device. The Arm software stack has been available on Ubuntu Launchpad Personal Package Archive (PPA) since the Ubuntu 20.08 release and had a formal release in Bullseye Debian 11.

5.1 Raspberry Pi installation

The following section shows you how to install Arm NN and PyArmNN on a Raspberry Pi.

About this task

The easiest way to install Arm NN and PyArmNN on the Raspberry Pi is through the APT repository. The 64-bit Ubuntu 20.04 and 21.04 have official support for the Raspberry Pi and can be installed following the steps that we show here. This guide has been tested on both versions.

The packages required to run object detection are:

- Arm NN
- PyArmNN
- OpenCV

If you are using a different OS, or would like to install versions other than those that are available on the apt repository, follow the steps in our guides:

- [Configure the Arm NN SDK build environment](#)
- [Cross compiling Arm NN for the Raspberry Pi and tensorflow](#)

Alternatively, you can download the official release binaries from GitHub.

Procedure

1. Install Ubuntu. See the installation instructions for the [Raspberry Pi 4 on the Ubuntu website](#).
2. Download and install the required packages. Install the Arm NN 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
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. Install the pip and OpenCV packages:

```
sudo apt-get install python3-opencv python3-pip
```

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

4. 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
```

5.2 Odroid N2 installation

The following section shows you how to install Arm NN and PyArmNN on an Odroid N2.

About this task

Odroid has official images for different versions of Ubuntu MATE. These can be downloaded from the Odroid wiki. The simplest way to install PyArmNN is to use the Ubuntu MATE 20.04 image and install via the APT repository.

The Odroid N2 comes with an Arm Mali GPU. To install PyArmNN with GPU support, you first install the OpenCL drivers. The official Odroid images come with OpenCL support, but the GPU is not recognized by clinfo as some libraries are missing.

The packages required to run object detection are:

- Arm NN
- PyArmNN
- OpenCV

Alternatively, if you are using a different OS or would like to install different versions than those available on the apt repository, follow the steps in our guides:

- [Configure the Arm NN SDK build environment](#)
- [Cross compiling Arm NN for the Raspberry Pi and tensorflow](#)

Another option is to download the official release binaries from the [Arm Software GitHub repository](#).

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 cp -r usr/* /usr/
sudo mkdir /etc/OpenCL
sudo mkdir /etc/OpenCL/vendors/
sudo bash -c 'echo "libmali.so" > /etc/OpenCL/vendors/mali.icd'
```

You can check your OpenCL installation by running the `clinfo` command:

```
clinfo
```

4. Remove the temporary folder:

```
cd ../
rm -rf temp-folder
```

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 libarmnn-latest-all
```

This guide uses the Arm NN TensorFlow Lite parser which is included in the `libarmnn-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
```

7. Install the pip and OpenCV packages:

```
sudo apt-get install -y python3-opencv python3-pip
```

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

6 Running the example

This section of the guide describes how to use PyArmNN to perform object detection using the example in the PyArmNN directory in the Arm NN repository.

6.1 Setting up your workspace

The following section describes how to set up the workspace.

About this task

The example inference application takes a sample video as the input. The application then runs inference on each frame to detect objects in the video. An output video is created in the same format as the input video, with bounding boxes drawn around detected objects. To run this example, you need a sample video.

The three main steps in the process are as follows:

1. Set up a workspace to contain the input video.
2. Select and download an appropriate model.
3. Run the example.

Procedure

1. Download the code:

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

2. Copy your example video into the object detection folder:

```
cp example.mp4 armnn/python/pyarmnn/examples/object_detection
```

This guide uses a video called `example.mp4`

6.2 Select and download the appropriate model

The following section shows you how to download the model and model label mappings.

About this task

The object detection example supports either use of YOLOv3 or Mobilenet-ssd. Arm Model Zoo supports both models. In this example, we use the YOLOv3 model.

Procedure

1. Clone the ML-Zoo repository:

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

2. Copy the model which in this example is YOLOv3 tiny:

```
cp ML-zoo/models/object_detection/yolo_v3_tiny/tflite_fp32/yolo_v3_tiny_dark\
net_fp32.tflite
armnn/python/pyarmnn/examples/object_detection/yolo_v3_tiny_darknet_fp32.tflite
```

3. Generate the label mapping for your model and copy them into the example folder:

```
cd ML-zoo/models/object_detection/yolo_v3_tiny/tflite_fp32/
./get_class_labels.sh
cd ../..
cp ML-zoo/models/object_detection/yolo_v3_tiny/tflite_fp32/labelmapping.txt
armnn/python/pyarmnn/examples/object_detection/yolo_v3_tiny_darknet_fp32.tflite
```

6.3 Run the application

The directory is now setup to run the example. The following section shows you how to install the required Python packages and run the application.

1. Change into the object detection directory.

```
cd armnn/python/pyarmnn/examples/object_detection
```

2. Install the Python packages.

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

3. Run the application:

```
python3 run_video_file.py \
--video_file_path example_video.mp4 \
--model_file_path yolo_v3_tiny_darknet_fp32.tflite \
--model_name yolo_v3_tiny \
--label_path labelmapping.txt \
--output_video_file_path output.mp4 \
--preferred_backends GpuAcc CpuAcc CpuRef
```

The preceding commands are for a device with an Arm Mali GPU and an Arm Cortex-A processor.

The commands may vary slightly depending on your device, your model, and your video.

The following table displays the options for the command:

Option	Description	Required
--video_file_path	Path to the video file to run object detection on.	Yes
--model_file_path	Path to .tflite, .pb, or .onnx object detection model.	Yes
--model_name	The name of the model being used. Assembles the workflow for the input model. The examples support the model names: ssd_mobilenet_v1 yolo_v3_tiny.	Yes

Option	Description	Required
<code>--label_path</code>	Path to labels file for the specified model file.	Yes
<code>--out\put_video_file_path</code>	Path to the output video file with detections added in.	No
<code>--preferred_backends</code>	You can specify one or more backend in order of preference. Accepted backends include <code>CpuAcc</code> , <code>GpuAcc</code> , <code>CpuRef</code> . Arm NN decides which layers of the network are supported by the backend, falling back to the next layer if it is unsupported. The command defaults to <code>'CpuAcc', 'CpuRef'</code> .	No

If your device does not have an Arm Mali GPU, remove `GpuAcc` from the `--preferred_backends` option.

The application exports an output video called `output.mp4` with bounding boxes and labels to a folder called `output` in the object detection directory.

7 Object detection application code overview

This section of the guide explains the scripts and the steps that are required to build your own object detection application.

The object detection script performs the following steps:

1. Initialize the application:
 - Import required Python package dependencies.
 - Read from the video source.
 - Create a video writer object for the output.
2. Create a network:
 - Create the parser and import the network file.
 - Optimize the graph for the compute device.
 - Load the network.
 - Create input and output binding information.
 - Implement a class to store information about the network graph and provide a method to execute inference.
3. Initialize the object detection pipeline:
 - Pre-process the captured frame.
 - Execute inference.
4. Process the input and output video:
 - Process the output into a list of detections.
 - Prepare labels.
 - Draw the bounding boxes onto the frame.
 - Execute inference on each frame of the input video.
5. Release the input and output video.
6. Run the application.

The following subsections describe each of these steps.

7.1 Import required Python package dependencies

The script must import the required Python package dependencies. These packages are PyArmNN, NumPy, OpenCV, and other inbuilt Python packages to help with the inference.

To install the Python package dependencies, run the following commands:

```
import cv2
import pyarmnn as ann
import numpy as np

from pathlib import Path
from typing import List, Tuple
import argparse
```

7.2 Read from the video source

The script loads the input video and sets the output video location.

The script uses `argparse` to allow the input video and output video names to be set from the command line.

Once the video has been imported, the script calculates the number of frames in the video. The script uses this information later to loop through each individual frame.

The following code shows how the script reads from the video source:

```
parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument(
    '--input', help='File path of input video file', required=True)
parser.add_argument(
    '--output', help='File path of output video file', required=True)
parser.add_argument(
    '--model_file_path', help='File path of output video file', required=True)
args = parser.parse_args()

input_video_loc = Path(args.input)
output_video_loc = Path(args.output)
Model_file_path = Path(args.model_file_path)

video = cv2.VideoCapture(input_video_loc.as_posix())
frame_count = range(int(video.get(cv2.CAP_PROP_FRAME_COUNT)))
```

7.3 Create a video writer object for the output

The script creates a video writer to use for storing the output video. The video writer is an OpenCV class which lets you output video frame by frame to a defined location.

To create a video writer, we must specify the following attributes:

- `filename` - the output video filename, supplied earlier from the command-line arguments.

- `fourcc` - the definition of the output video data format.
- `fps` - the number of frames in each second.
- `frameSize` - the width and height of the output video.

This information is obtained from the input video to ensure that the input and video have the same format.

The following code shows how the script creates the video writer object:

```
video_encoding = video.get(cv2.CAP_PROP_FOURCC)
fps = video.get(cv2.CAP_PROP_FPS)
frame_size = (int(video.get(cv2.CAP_PROP_FRAME_WIDTH)),
              int(video.get(cv2.CAP_PROP_FRAME_HEIGHT)))

video_writer = cv2.VideoWriter(filename = output_video_loc.as_posix(),
                               fourcc = video_encoding,
                               fps = fps,
                               frameSize = frame_size
                              )
```

7.4 Create the parser and import the network file

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

The `CreateNetworkFromBinaryFile()` function creates the parser and loads the network file. The parser then constructs the underlying Arm NN graph from the network file.

The following code selects the TFLite parser and converts the model file into an Arm NN network graph:

```
parser = ann.ITfLiteParser()
network = parser.CreateNetworkFromBinaryFile(model_file_path.as_posix())
```

7.5 Optimize the graph for the compute device

The model is now ready to be optimized for Arm hardware. Arm NN supports optimized execution on multiple CPU, GPU, and Ethos-N NPU devices. To optimize the model, select the appropriate Arm NN backend for the target hardware.

An Arm NN backend defines how the network graph is mapped to the hardware. Arm NN includes several built-in backends. However, it is also possible to write your own custom backend.

The currently available built-in backends are:

- `CpuAcc` represents the CPU backend.
- `GpuAcc` represents the GPU backend.

- `CpuRef` represents the CPU reference kernels.
- `EthosNAcc` represents the Ethos-N NPU backend.

We optimize the imported graph by specifying a list of backends in order of preference. For example, if your device has an Arm GPU you might select the following preferred order of backends:

1. `GpuAcc`
2. `CpuAcc`
3. `CpuRef`

This list would mean that `GpuAcc` was used for all operations that are supported by the Arm GPU. For any operations not supported in the `GpuAcc` backend, `CpuAcc` and `CpuRef` are used as fallbacks.

The following code defines the preferred backends and then calls the `optimize()` function to optimize the graph for inference:

```
preferred_backends = [ann.BackendId("CpuAcc"), ann.BackendId("CpuRef")]

options = ann.CreationOptions()
runtime = ann.IRuntime(options)

opt_network, messages = ann.Optimize(network,
                                     preferred_backends,
                                     runtime.GetDeviceSpec(),
                                     ann.OptimizerOptions()
                                    )
```

7.6 Load the network

The network is now ready to be loaded into the Arm NN runtime context.

The `LoadNetwork()` function loads the optimized network, and creates the backend-specific workloads for the layers:

```
net_id, _ = runtime.LoadNetwork(opt_network)
```

7.7 Create input and output binding information

To input new data from our video and use the output, we must find the input and output binding information from the network graph.

The input binding information has one tensor, and the output binding information contains four tensors.

The script obtains the output binding information by looping through the output names. This information can also be found in the README.md for your model on the Arm ML Zoo.

The `GetSubgraphInputTensorNames()` function extracts all the input names and the `GetNetworkInputBindingInfo()` function obtains the input binding information of the graph.

Similarly, we can get the output binding information for an output layer by using the parser to retrieve output tensor names and calling the `GetNetworkOutputBindingInfo()` function.

The following code creates the input and output binding information:

```
graph_id = parser.GetSubgraphCount() - 1

#input is just one frame
input_names = parser.GetSubgraphInputTensorNames(graph_id)
input_binding_info = parser.GetNetworkInputBindingInfo(graph_id, input_names[0])

#output names is a vector
output_names = parser.GetSubgraphOutputTensorNames(graph_id)
output_binding_info = []

#loop through each output name and get the binding
for output_name in output_names:
    out_bind_info = parser.GetNetworkOutputBindingInfo(graph_id, output_name)
    output_binding_info.append(out_bind_info)
```

7.8 Implement a class for the network graph

The script implements a class to store information about the network graph and provide a method to execute inference.

The class is initiated with information about the network graph and creates a variable for the output tensors. The inference function `run()` takes as input the input tensors from the video and runs inference using the `EnqueueWorkload()` function.

The following code shows how the class is implemented:

```
class ArmnnNetworkExecutor:
    def __init__(self, net_id, runtime, input_binding_info, output_binding_info):
        """
        Creates an inference executor for a given network and a list of backends.
        """
        self.network_id = net_id
        self.runtime = runtime
        self.input_binding_info = input_binding_info
        self.output_binding_info = output_binding_info

        self.output_tensors = ann.make_output_tensors(self.output_binding_info)

    def run(self, input_tensors: list) -> List[np.ndarray]:
        """
        Executes inference for the loaded network.
        Args:
            input_tensors: The input frame tensor.
            output_tensors: The output tensor from output node.
            runtime: Runtime context for executing inference.
            net_id: Unique ID of the network to run.
        Returns:
            list: Inference results as a list of ndarrays.
        """
        self.runtime.EnqueueWorkload(self.network_id, input_tensors,
                                     self.output_tensors)
```

```

        output = ann.workload_tensors_to_ndarray(self.output_tensors)

        return output

    executor = ArmnnNetworkExecutor(net_id = net_id,
                                    runtime = runtime,
                                    input_binding_info = input_binding_info,
                                    output_binding_info = output_binding_info)

```

7.9 Pre-process the captured frame

For each frame inference, the script must perform preprocessing to ensure that the frame is of the right format for the network. The network in this example requires images to have height and width of 300 pixels and to be in 32-bit floating-point format.

The script uses the following two functions to perform the preprocessing:

- The `resize_with_aspect_ratio()` function resizes the input frame to the correct size, keeping the aspect ratio.
- The `preprocess()` function uses the resizing function and sets the correct data types.

The following code performs this preprocessing:

```

def resize_with_aspect_ratio(frame: np.ndarray, input_binding_info: tuple):
    """
    Resizes frame while maintaining aspect ratio, padding any empty space.
    Args:
        frame: Captured frame.
        input_binding_info: Contains shape of model input layer.
    Returns:
        Frame resized to the size of model input layer.
    """
    aspect_ratio = frame.shape[1] / frame.shape[0]
    model_height, model_width = list(input_binding_info[1].GetShape())[1:3]

    if aspect_ratio >= 1.0:
        new_height, new_width = int(model_width / aspect_ratio), model_width
        b_padding, r_padding = model_height - new_height, 0
    else:
        new_height, new_width = model_height, int(model_height * aspect_ratio)
        b_padding, r_padding = 0, model_width - new_width

    # Resize and pad any empty space
    frame = cv2.resize(frame, (new_width, new_height), interpolation=cv2.INTER_LINEAR)
    frame = cv2.copyMakeBorder(frame, top=0, bottom=b_padding, left=0,
                              right=r_padding,
                              borderType=cv2.BORDER_CONSTANT, value=[0, 0, 0])
    return frame

def preprocess(frame: np.ndarray, input_binding_info: tuple):
    """
    Takes a frame, resizes, swaps channels and converts data type to match
    model input layer. The converted frame is wrapped in a const tensor
    and bound to the input tensor.
    Args:
        frame: Captured frame from video.
        input_binding_info: Contains shape and data type of model input layer.
    Returns:
        Input tensor.
    """
    # Swap channels and resize frame to model resolution

```



```
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
resized_frame = resize_with_aspect_ratio(frame, input_binding_info)

# Expand dimensions and convert data type to match model input
data_type = np.float32 if input_binding_info[1].GetDataType() ==
ann.DataType_Float32 else np.uint8
resized_frame = np.expand_dims(np.asarray(resized_frame, dtype=data_type), ax\
is=0)
assert resized_frame.shape == tuple(input_binding_info[1].GetShape())

input_tensors = ann.make_input_tensors([input_binding_info], [resized_frame])

return input_tensors
```

7.10 Process the output into a list of detections

After the script has run inference for each frame, we process the output into a usable format. Three functions are used to process the output of a YOLOv3 model:

- The `iou()` function takes the coordinates of two bounding boxes and calculates the intersection over union between the two. Boxes with higher IOU have more overlap. This means that one should be filtered out, so that there are not too many bounding boxes in the same place.
- The `process_output()` function takes the predictions from the model and reformats them into a list of different detections, filtering out low scoring predictions and high IOU scoring bounding boxes. The process keeps repeating, until there are no two pairs of bounding boxes that have an IOU over a certain threshold between 0 and 1.
- The `resize_factor()` function calculates a multiplier to use when rescaling the bounding boxes to their correct position in the frame.

The output vector is converted so that it has the following form:

```
[[class, [box positions], confidence], ...]
```

The following code processes the output into a list of detections:

```
def iou(box1: list, box2: list):,
    """
    Calculates the intersection-over-union (IoU) value for two bounding boxes.

    Args:
    box1: Array of positions for first bounding box
    in the form [x_min, y_min, x_max, y_max].
    box2: Array of positions for second bounding box.

    Returns:
    Calculated intersection-over-union (IoU) value for two bounding boxes.
    """
    area_box1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area_box2 = (box2[2] - box2[0]) * (box2[3] - box2[1])

    if area_box1 <= 0 or area_box2 <= 0:
        iou_value = 0
    else:
        y_min_intersection = max(box1[1], box2[1])
        x_min_intersection = max(box1[0], box2[0])
        y_max_intersection = min(box1[3], box2[3])
```

```
x_max_intersection = min(box1[2], box2[2])

area_intersection = max(0, y_max_intersection - y_min_intersection) *\
max(0, x_max_intersection - x_min_intersection)
area_union = area_box1 + area_box2 - area_intersection

try:
    iou_value = area_intersection / area_union
except ZeroDivisionError:
    iou_value = 0

return iou_value

def process_output(output: np.ndarray, confidence_threshold=0.40, iou_thresh\
old=0.40):
    """
    Performs non-maximum suppression on input detections. Any detections
    with IOU value greater than given threshold are suppressed.

    Args:
        output: Vector of outputs from network.
        confidence_threshold: Selects only strong detections above this value.
        iou_threshold: Filters out boxes with IOU values above this value.

    Returns:
        A list of detected objects in the form [class, [box positions], confidence]
    """
    if len(output) != 1:
        raise RuntimeError('Number of outputs from YOLO model does not equal 1')

    # Find the array index of detections with confidence value above threshold
    confidence_det = output[0][:, :, 4][0]
    detections = list(np.where(confidence_det > confidence_threshold)[0])
    all_det, nms_det = [], []

    # Create list of all detections above confidence threshold
    for d in detections:
        box_positions = list(output[0][:, d, :4][0])
        confidence_score = output[0][:, d, 4][0]
        class_idx = np.argmax(output[0][:, d, 5:])
        all_det.append((class_idx, box_positions, confidence_score))

    # Suppress detections with IOU value above threshold
    while all_det:
        element = int(np.argmax([all_det[i][2] for i in range(len(all_det))]))
        nms_det.append(all_det.pop(element))
        all_det = [*filter(lambda x: (iou(x[1], nms_det[-1][1]) <= iou_threshold),
[det for det in all_det])]

    return nms_det

def calculate_resize_factor(video: cv2.VideoCapture, input_binding_info: tuple):
    """
    Gets a multiplier to scale the bounding box positions to
    their correct position in the frame.

    Args:
        video: Video capture object, contains information about data source.
        input_binding_info: Contains shape of model input layer.

    Returns:
        Resizing factor to scale box coordinates to output frame size.
    """
    frame_height = video.get(cv2.CAP_PROP_FRAME_HEIGHT)
    frame_width = video.get(cv2.CAP_PROP_FRAME_WIDTH)
    model_height, model_width = list(input_binding_info[1].GetShape())[1:3]
    return max(frame_height, frame_width) / max(model_height, model_width)

resize_factor = calculate_resize_factor(video, input_binding_info)
```

7.11 Prepare labels

To interpret the inference result on the loaded network, the script must load the labels that are associated with the model.

The script imports and reformats the `labelmappings.txt` into a format for the model. The `labelmappings.txt` contains a list of classes. The row number, starting from 0, relates to the class predicted in the model.

Each output is assigned an associated bounding box color and class name. To assign the bounding box and class name, the script creates a dictionary which maps output index to a list of class names and colors.

The following code maps the output labels and colors:

```
def dict_labels(labels_file_path: str, include_rgb=False) -> dict:
    """Creates a dictionary of labels from the input labels file.
    Args:
        labels_file: Path to file containing labels to map model outputs.
        include_rgb: Adds randomly generated RGB values to the values of the
            dictionary. Used for plotting bounding boxes of different colours.
    Returns:
        Dictionary with classification indices for keys and labels for values.
    Raises:
        FileNotFoundError:
            Provided `labels_file_path` does not exist.
    """
    labels_file = Path(labels_file_path)
    if not labels_file.is_file():
        raise FileNotFoundError(
            errno.ENOENT, os.strerror(errno.ENOENT), labels_file_path
        )

    labels = {}
    with open(labels_file, "r") as f:
        for idx, line in enumerate(f, 0):
            if include_rgb:
                labels[idx] = line.strip("\n"), tuple(np.random.random(size=3) *
255)
            else:
                labels[idx] = line.strip("\n")

    return labels

labels = utils.dict_labels(label_path.as_posix(), include_rgb=True)
```

7.12 Draw the bounding boxes

The `draw_bounding_boxes()` function takes the inference results and draws bounding boxes around detected objects. This function also adds the associated label and confidence score.

The following code draws the bounding boxes onto a frame from a video:

```
def draw_bounding_boxes(frame: np.ndarray, detections: list, resize_factor, labels:
dict):
```

```
"""
    Draws bounding boxes around detected objects and adds a label and confidence
    score.
    Args:
        frame: The original captured frame from video source.
        detections: A list of detected objects in the form [class, [box positions],
        confidence].
        resize_factor: Resizing factor to scale box coordinates to output frame
        size.
        labels: Dictionary of labels and colors keyed on the classification index.
    """
    for detection in detections:
        class_idx, box, confidence = [d for d in detection]
        label, color = labels[class_idx][0].capitalize(), labels[class_idx][1]

        # Obtain frame size and resized bounding box positions
        frame_height, frame_width = frame.shape[:2]
        x_min, y_min, x_max, y_max = [int(position * resize_factor) for position in
box]

        # Ensure box stays within the frame
        x_min, y_min = max(0, x_min), max(0, y_min)
        x_max, y_max = min(frame_width, x_max), min(frame_height, y_max)

        # Draw bounding box around detected object
        cv2.rectangle(frame, (x_min, y_min), (x_max, y_max), color, 2)

        # Create label for detected object class
        label = f'{label} {confidence * 100:.1f}%'
        label_color = (0, 0, 0) if sum(color)>200 else (255, 255, 255)

        # Make sure label always stays on-screen
        x_text, y_text = cv2.getTextSize(label, cv2.FONT_HERSHEY_DUPLEX, 1, 1)[0]
[:2]

        lbl_box_xy_min = (x_min, y_min if y_min<25 else y_min - y_text)
        lbl_box_xy_max = (x_min + int(0.55 * x_text), y_min + y_text if y_min<25
else y_min)
        lbl_text_pos = (x_min + 5, y_min + 16 if y_min<25 else y_min - 5)

        # Add label and confidence value
        cv2.rectangle(frame, lbl_box_xy_min, lbl_box_xy_max, color, -1)
        cv2.putText(frame, label, lbl_text_pos, cv2.FONT_HERSHEY_DUPLEX, 0.50,
                    label_color, 1, cv2.LINE_AA)
```

7.13 Execute inference on each frame of the input video

The main inference loop performs the following tasks:

1. Feed in the next frame from the input video.
2. Preprocess the input frame.
3. Execute the network for the input.
4. Process the output into a list of detections.
5. Draw the bounding boxes onto the frame.
6. Write the frame to the output video.

Many of these steps are examined in further detail in other sections of this guide.

The following code shows the main inference loop:

```
for _ in frame_count:
    # import the frame to an array
    frame_present, frame = video.read()

    # if a frame is not opened continue to next frame
    if not frame_present:
        continue

    # preprocess the input frame
    input_tensors = preprocess(frame, executor.input_binding_info)

    # get output result using the run() method
    output_result = executor.run(input_tensors)

    # process the output in "detections"
    detections = process_output(output_result)

    # convert detections into bounding boxes
    draw_bounding_boxes(frame, detections, resize_factor, labels)

    # write to video file
    video_writer.write(frame)
```

7.14 Release the input and output video

The final step is for the script to release the input video and output video. The following code shows how to do this:

```
video.release(), video_writer.release()
```

7.15 Run the application

To run the video file with the YOLOv3 model with PyArmNN, use the following command:

```
python3 object_detection.py --input example.mp4 --output video_with_boxes.mp4 --mod\
el_file_path yolo_v3_tiny_darknet_fp32.tflite
```

8 Next steps

This guide has shown you how to a run real-time object detection app with YOLOv3 using PyArmNN on Odroid N2 and Raspberry Pi. You will now know how to install the packages on your specific device, how to run the application and the underlying code used to create the application.

- For other guides related to PyArmNN and machine learning on Arm NN, visit [AI and Machine Learning How to guides](#).

9 Related information

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

- [Arm NN on Arm Developer](#)
- [Configure the Arm NN SDK build environment](#)
- [Cross compiling Arm NN for the Raspberry Pi and TensorFlow](#)

Other Arm resources:

- [Arm Community](#) - ask development questions and find articles and blogs on specific topics from Arm experts.
- [Arm NN Github](#) - raise queries or issues associated with the Arm NN how-to guides.
- [Arm NN Product Documentation](#) - find out more about the latest Arm NN features.

Appendix A Revisions

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

Table A-1: First release for version 1.01

Change	Location
First release	—

Table A-2: First release for version 21.08

Change	Location
Fixes typos	—

Table A-3: Second release for version 21.08

Change	Location
PyArmNN is added to a command in section 2 and 3	Running the example
Updates command in step 3	Raspberry Pi installation
Updates command in step 4	Odroid N2 installation
Fixes formatting issues	—
Adds additional resources	Related information