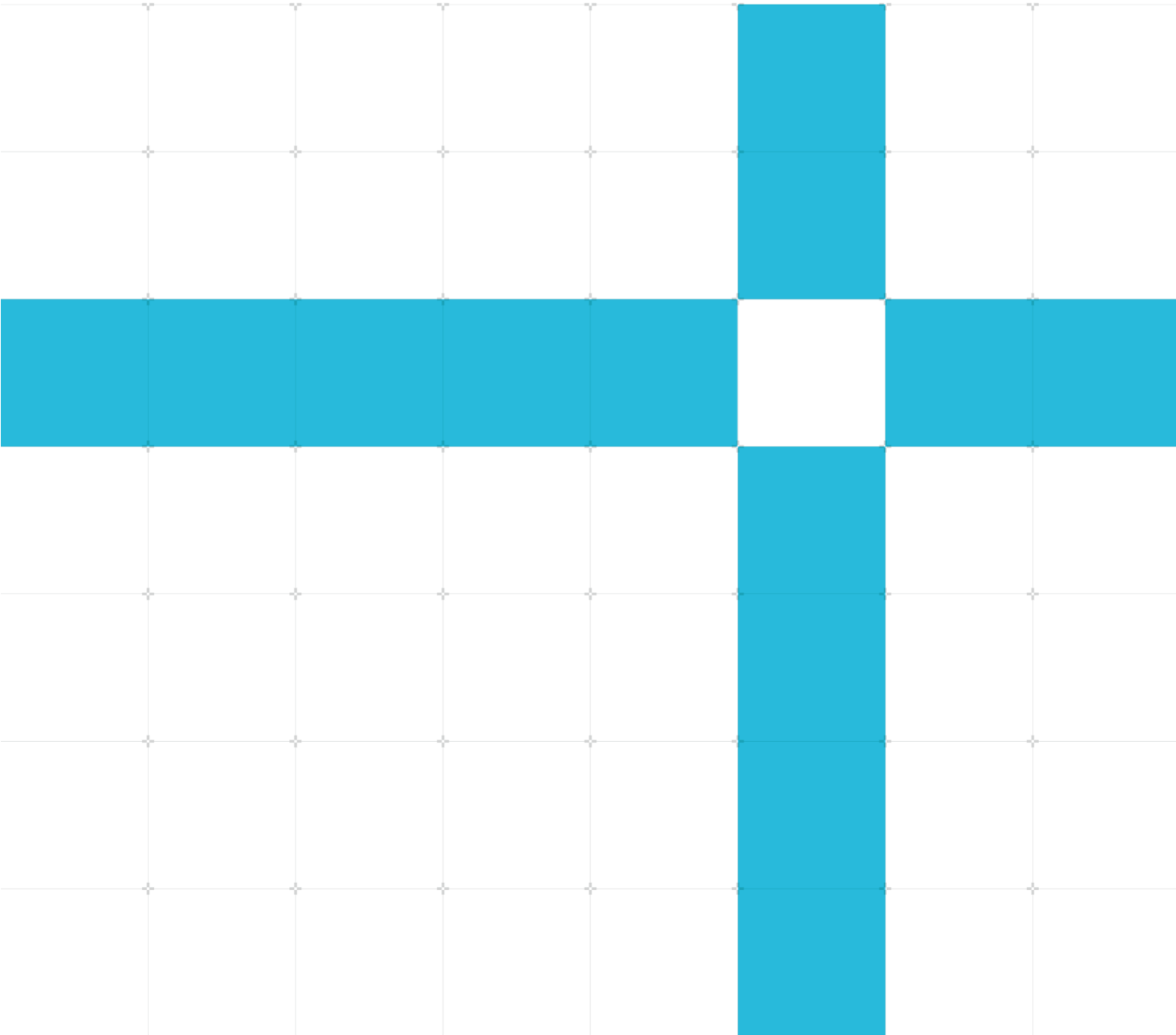
Object recognition with Arm NN and Raspberry Pi

Issue 0200
102274



Object recognition with Arm NN and Raspberry Pi

Release information

Document history

Issue	Date	Confidentiality	Change
01	14 October 2020	Non-Confidential	First release
02	05 February 2021	Non-Confidential	Update instructions for ArmNN 20.11

Non-Confidential Proprietary Notice

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.

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.

Web Address

www.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 shows you how to use Arm NN and PyArmNN to build and run a real-time object detection system.

The system runs on a Raspberry Pi 4 with Raspbian 10 operating system.

This guide examines the following sample applications that ship as part of Arm NN and PyArmNN:

- **PyArmNN Object Detection Sample Application**
- **Arm NN Object Detection Example**

These sample applications take a model and video file or camera feed as input. The applications then run inference on each frame. Finally, the applications draw bounding boxes around detected objects, with the corresponding labels and confidence scores overlaid.

By understanding how these sample applications perform object detection, you can learn to write your own Machine Leaning applications using Arm NN and PyArmNN.

1.1 Before you begin

This guide requires installing ArmNN on a Raspberry Pi. From ArmNN 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. Because Raspberry Pi OS 64-bit is still in beta, we recommend Ubuntu 64-bit 20.10 Groovy Gorilla. 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 LTS 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.
   Add the PPA to your sources the software-properties-common package:

   ```
   sudo apt install software-properties-common

   sudo add-apt-repository ppa:armnn/ppa

   sudo apt update

   sudo apt-get install -y python3-pyarmnn libarmnn-cpuacc-backend<version> libarmnn-cpuref-backend<version>
   ```

   Replace `<version>` with the latest supported version of `libarmn`, as listed on **our GitHub repository**.
   These packages provide the TensorflowLite parser for ArmNN, which is what this guide uses.

2 What is object detection?

Object detection is the process of identifying real-world objects, like people, cars, bottles, or sandwiches, in still images or videos. To understand an entire scene, a viewer must detect, identify, and locate multiple objects simultaneously.

Object detection is commonly used in applications like image retrieval, security, surveillance, and Advanced Driver Assistance Systems (ADAS).

There are several different approaches to performing object detection, including the following:

- Feature-based object detection
- Support Vector Machine (SVM) classification with Histogram of Oriented Gradient (HOG) feature descriptors
- Deep learning object detection

This guide focuses on deep learning object detection with Arm NN SDK.

Real-world use cases for object detection use cases include the following:

- Video surveillance

  Video surveillance is a natural application of object detection techniques. The ability to identify and track multiple instances of an object in a scene is a key requirement of video surveillance systems. Granular object tracking provides valuable insights that can assist, for example, crowd security, analysis of store traffic, or stock movement on the factory floors.

  The following image shows an example of object detection used in video surveillance:

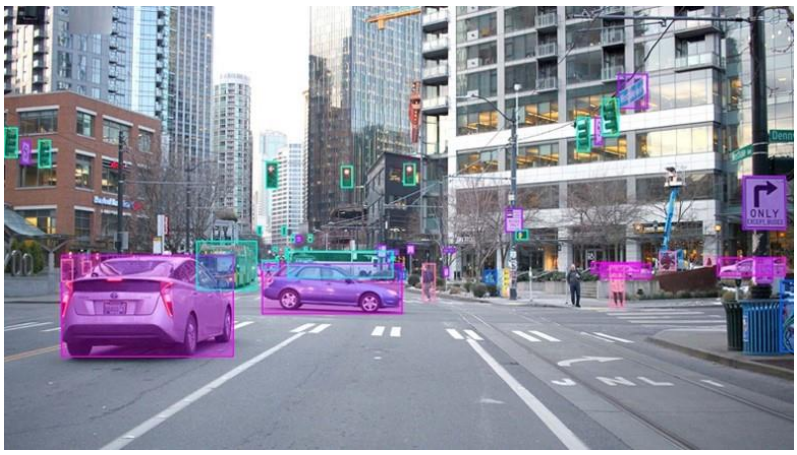
  

  Example of object detection for video surveillance

- Autonomous driving

Autonomous vehicles use real-time obstacle detection models to identify cars, pedestrians, bicycles, and traffic signs. The vehicle uses this information to decide whether to accelerate, decelerate, or turn.

The following image shows an example of real-time object detection used in an autonomous vehicle:

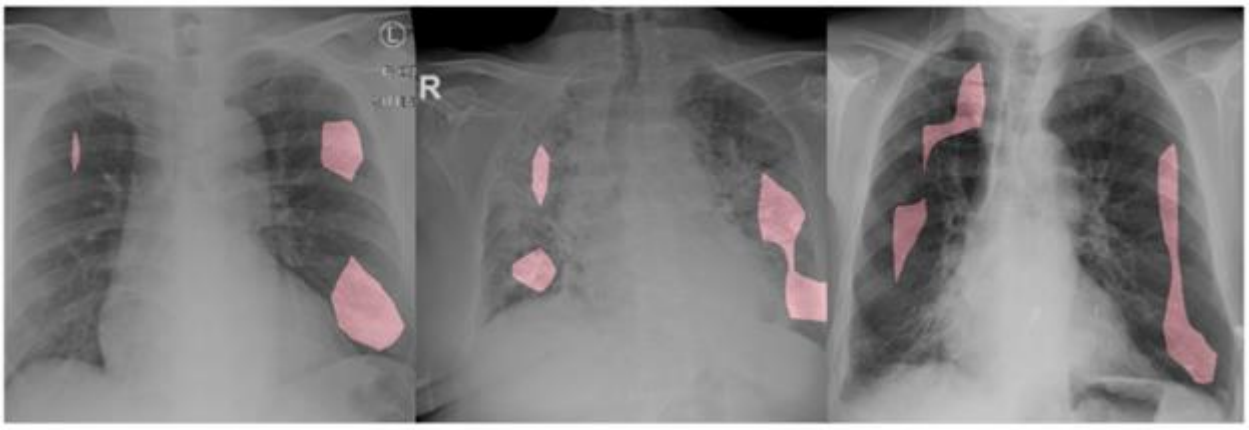


Example of object detection for autonomous vehicles (source: Mighty AI)

- Medical imaging

  In health care, object detection assists doctors in diagnosis and treatment planning. For example, Arm ecosystem partner Darwin.ai developed COVID-Net, a deep neural network which examines chest X-rays to help medical professionals rapidly screen for COVID-19 infections.

  The following image shows an example of object detection in medical X-rays:



Example of object detection in medical X-rays (source: DarwinAI)

- Manufacturing

Object detection helps to fully automate manufacturing systems. Object detection has applications in many manufacturing processes, including quality assurance, inventory management, and assembly line sorting.

3 Object detection model structure

Most deep learning-based object detection models have two parts:

- An encoder, which takes an image as input and runs it through a series of blocks and layers that extract features. The encoder then uses these features to locate and label objects.
- A decoder. Outputs from the encoder are then passed to a decoder. The decoder predicts bounding boxes and labels for each object.

The simplest decoder is a pure regressor. The regressor connects to the output of the encoder and directly predicts the location and size of each bounding box. The output of the model is the X, Y coordinate pair for the object and its extent. The disadvantage of using a pure regressor is that you must define the number of predicted objects ahead of time.

An extension of the regressor approach is a region proposal network. In this type of decoder, the model proposes regions of an image where it believes an object might reside. The pixels in these regions are fed into a classification network to determine a matching label. The region proposal network is a more accurate and flexible model that can process an arbitrary number of regions.

Single Shot Detectors (SSDs) seek to provide a middle ground between pure regressors and region proposal networks. Rather than using a subnetwork to propose regions, SSDs rely on a set of predetermined regions. A grid of anchor points is laid over the input image. At each anchor point, boxes of multiple shapes and sizes serve as regions.

For each box at each anchor point, the SSD model outputs:

- A prediction of whether an object exists within the region
- Modifications to the location and size of the box, to make the box fit the object more closely.

Because there are multiple boxes at each anchor point and anchor points might be close together, SSDs produce many potential detections that overlap. Post-processing must be applied to SSD outputs, to prune away most of these predictions, and pick the best one.

Object detectors output the location and label for each object. To benchmark the model performance, the most commonly-used metric is intersection-over-union (IOU). Given two bounding boxes, you compute the area of the intersection and divide by the area of the union. Metric values range from 0 (no interaction) to 1 (perfectly overlapping). For labels, you can use a simple percentage correct.

3.1 YOLO and MobileNet-SSD

Several models belong to the SSD family. The main differences between these variants are their encoders and the specific configuration of predetermined anchors.

YOLO v3 is a fast real-time object detection system. YOLO stands for You Only Look Once. Techniques like multi-scale predictions and improved backbone classifiers enable this fast performance. YOLO trains a single neural network model that directly predicts bounding boxes and class labels for each bounding box. You can find more details in the paper **YOLOv3: An Incremental Improvement**.

MobileNet-SSD models feature a MobileNet-based encoder. SSDs are a good choice for models that are destined for mobile or embedded devices. For more information, see **MobileNetV2 + SSDLite with Core ML**.

3.2 Region-Based Convolutional Neural Network

The Region-Based Convolutional Neural Network (R-CNN) family of methods do the following:

3. Generate candidate bounding boxes
4. Extract features from each candidate region using a deep convolutional neural network
5. Classify the features as one of the known classes

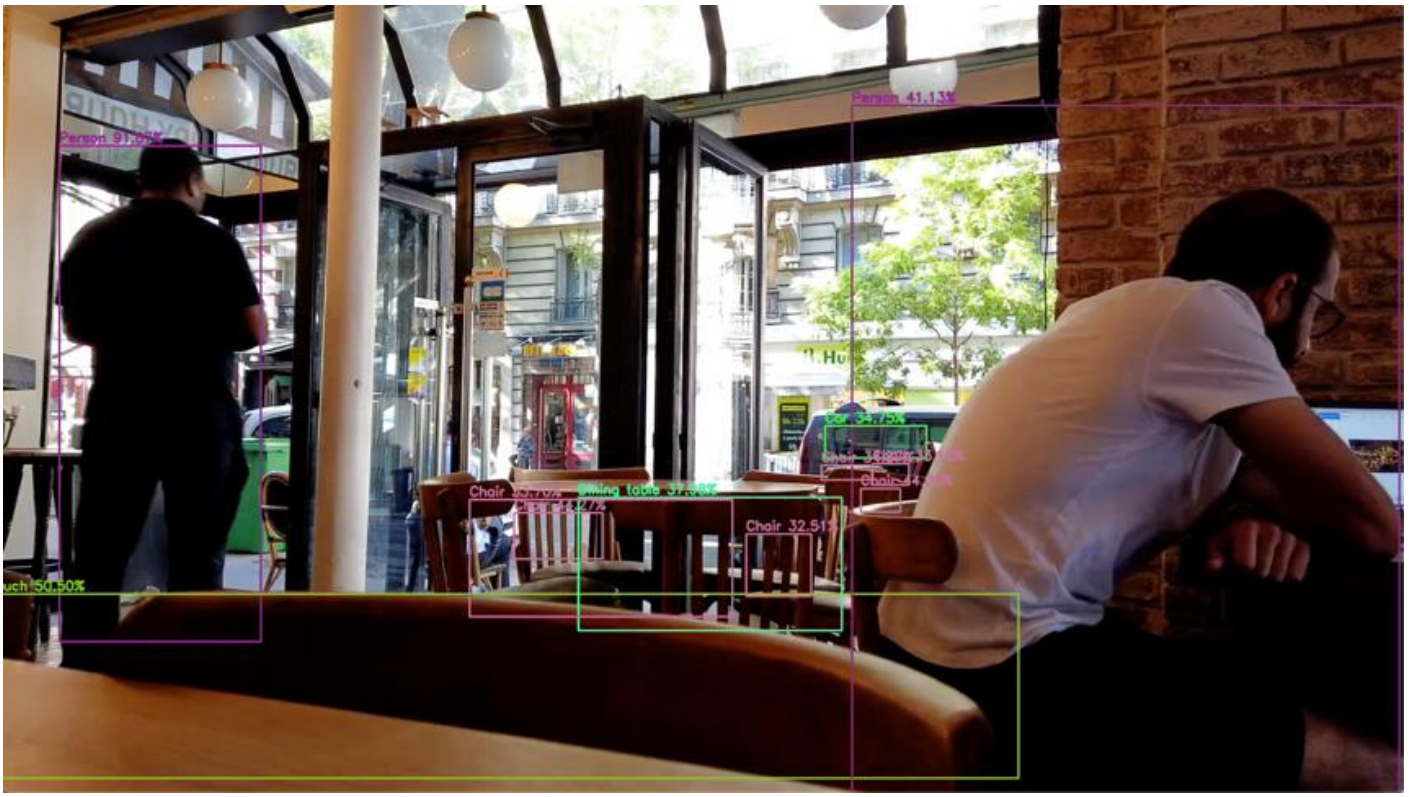
R-CNN is a relatively simple and straightforward approach. Several popular object detection models belong to the R-CNN family, including Fast R-CNN and Mask R-CNN.

4 PyArmNN object recognition

This section of the guide describes how to use PyArmNN to perform object recognition, using the **PyArmNN Object Detection Sample Application** as an example.

The sample application takes a model and video file or camera feed as input. The application then runs inference on each frame. Finally, the application draws bounding boxes around detected objects, with the corresponding labels and confidence scores overlaid.

The following image shows an example of one frame from the resulting video file with bounding boxes and confidence scores:



The PyArmNN Object Detection Sample Application performs the following steps:

1. Initialization
    a. Read from the video source
    b. Prepare labels and model-specific functions
2. Create a network
    a. Create the parser and import a graph
    b. Optimize the graph for the compute device
    c. Create input and output binding information

3. Object detection pipeline
    a. Preprocess the captured frame
    b. Make the input and output tensors
    c. Execute inference
4. Postprocessing
    a. Decode and process the inference output
    b. Draw the bounding boxes
    c. Get the example code
    d. Run the application

The following subsections describe these steps.

4.1 Read from the video source

The application parses the supplied user arguments and loads the specified video file or stream into an OpenCV `cv2.VideoCapture` object. The application then uses this object to capture frames from the source with the `read()` function.

The `VideoCapture` object also provides information about the source, like the frame rate and resolution of the input video. The application uses this information to create a `cv2.VideoWriter` object. This object is used at the end of every loop to write the processed frame to an output video file.

4.2 Prepare labels and model-specific functions

To interpret the inference result on the loaded network, an application must load the labels that are associated with the model. In the sample application, the `dict_labels()` function creates a dictionary that is keyed on the classification index at the output node of the model. The values in the dictionary map each label to a randomly generated RGB color. This mapping means that each class has a unique color, which is useful when plotting the bounding boxes of detected objects in a frame.

The user-specified model accesses and returns functions to decode and process the inference output, along with a resize factor. This resize factor is used when plotting bounding boxes, to ensure that they are scaled to their correct position in the original frame.

4.3 Create the parser and import a graph

A PyArmNN application 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.

Because both the Yolo v3 and SSD models are in the TFLite format, the sample application uses the TFLite parser `armnnTfLitePaser` to process the models.

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

4.4 Optimize the graph for the compute device

Arm NN supports optimized execution on multiple CPU, GPU, and Ethos-N NPU devices. Before executing a graph, the application must select the appropriate device context by using `IRuntime()` to create a runtime context with default options.

We can optimize the imported graph by specifying a list of backends in order of preference and implementing backend-specific optimizations. A unique string identifies each one of these backends. For example:

- `CpuAcc` represents the CPU backend.
- `GpuAcc` represents the GPU backend.
- `CpuRef` represents the CPU reference kernels.

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.

4.5 Create input and output binding information

Parsers extract the input information for the network. The `GetSubgraphInputTensorNames()` function extracts all the input names and 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

- Tensor information including:
  - Data type
  - Quantization information
  - Number of dimensions
  - Total number of elements

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.

4.6 Preprocess the captured frame

Each frame that is captured from the video source is read as a `ndarray` in BGR format. Each frame must then be preprocessed before being passed into the network.

This preprocessing step consists of the following:

1. Swap channels. In this example, swap BGR to RGB.
2. Resize the frame to the required resolution
3. Expand the dimensions of the array and perform data type conversion to match the model input layer.

You can read `input_binding_info` to obtain information about the shape and the data type of the input tensor. For example, SSD MobileNet V1 takes an input tensor with shape `[1, 300, 300, 3]` and data type `uint8`.

4.7 Make the input and output tensors

The `make_input_tensors()` function produces the input workload tensors.

The `make_output_tensors()` function produces the output workload tensors.

4.8 Execute inference

After creating the workload tensors, the compute device performs inference for the loaded network using the `EnqueueWorkload()` function of the runtime context. Calling the `workload_tensors_to_ndarray()` function obtains the inference results as a list of `ndarrays`.

4.9 Decode and process the inference output

The output from inference must be decoded to obtain information about detected objects in the frame.

The examples includes implementations of two networks, but you can implement your own network decoding solution. For more information, see **Implementing Your Own Network**.

For SSD MobileNet V1 models, the application decodes the results to obtain the bounding box positions, classification index, confidence, and number of detections in the input frame.

For YOLO v3 tiny models, the application decodes the output and performs non-maximum suppression. This suppression filters out any weak detections below a confidence threshold and any redundant bounding boxes above an intersection-over-union (IoU) threshold.

Experiment with different threshold values for confidence and IoU to achieve the best visual results.

Detection results are returned as a list with the following form:

```
[class index, [box positions], confidence score]
```

Where `[box positions]` contains bounding box coordinates in the following form:

```
[x_min, y_min, x_max, y_max]
```

4.10 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 labels dictionary that we created in **Preparing labels and model-specific functions** uses the class index of the detected object as a key to return the associated label and color for that class. The resize factor that we defined in **Preparing labels and model-specific functions** scales the bounding box coordinates to their correct positions in the original frame.

The processed frames are then written to file or displayed in a separate window.

4.11 Get the example code

You can find code for our example application, and more instructions, on our **GitHub repository**.

To use the example:

1. Install git:
   ```
   $ sudo apt install git
   ```
2. Clone the repo:
   ```
   git clone https://github.com/ARM-software/armnn.git
   ```

3. Move to the object detection example:
```
cd armnn/python/pyarmnn/examples/object_detection/
```
4. Install the following dependencies on your system
```
$ sudo apt update
$ sudo apt install libopencv-dev python3-opencv python3-venv
```
5. Create a virtual environment:
```
$ python3.8 -m venv devenv --system-site-packages
$ source devenv/bin/activate
```
6. Install the following dependencies on the virtual environment:
```
$ pip install -r requirements.txt
```
7. Download the object detection model from our **GitHub repository**.
8. Download a video of your choice as an MP4.

4.12 Run the application

Run the model on the video you downloaded:

```
$ python run_video_file.py --video_file_path <video_file_path> --model_file_path
<model_file_path> --model_name <model_name>

$ python run_video_file.py --video_file_path <video_file_path> --model_file_path
ssd_mobilenet_v1.tflite --model_name ssd_mobilenet_v1 --label_path labels.txt

$ python run_video_file.py --video_file_path <video_file_path> --model_file_path
yolo_v3_tiny_darknet_fp32.tflite --model_name yolo_v3_tiny --label_path labels.txt
```

5 Arm NN C++ API object recognition

This section of the guide describes how to use the Arm NN public C++ API to perform object recognition. We use the **Arm NN Object Detection Example** to illustrate the process.

The sample application takes a model and video file or camera feed as input. The application then runs inference on each frame. Finally, the application draws bounding boxes around detected objects, with the corresponding labels and confidence scores overlaid.

The Arm NN Object Detection Example performs the following steps:

1. Initialization
    a. Read from the video source
    b. Prepare labels and model-specific functions
2. Create a network
    a. Create the parser and import a graph
    b. Optimize the graph for the compute device
    c. Create input and output binding information
3. Object detection pipeline
    a. Preprocess the captured frame
    b. Make the input and output tensors
    c. Execute inference
4. Postprocessing
    a. Decode and process the inference output
    b. Draw the bounding boxes
    c. Run the application

The following subsections describe these steps.

5.1 Read from the video source

After parsing user arguments, the application loads the chosen video file or stream into an OpenCV `cv::VideoCapture` object. The main function uses the **IFrameReader** interface and the OpenCV-specific implementation **CvVideoFrameReader** to capture frames from the source using the `ReadFrame()` function.

The `CvVideoFrameReader` object also provides information about the input video. Based on this information and the application arguments, the application creates one of the implementations of the **IFrameOutput** interface: `CvVideoFileWriter` or `CvWindowOutput`. The created object is used at the end of every loop to do one of the following:

- **CvVideoFileWriter** uses `cv::VideoWriter` with an `ffmpeg` backend to write the processed frame to an output video file.
- **CvWindowOutput** uses the `cv::imshow()` function to write the processed frame to a GUI window. See the `GetFrameSourceAndSink` function in **Main.cpp** for more details.

5.2 Prepare labels and model-specific functions

To interpret the result of running inference on the loaded network, the application must load the labels associated with the model. In the provided example code, the `AssignColourToLabel` function creates a vector of `[label, color]` pairs. The vector is ordered according to the object class index at the output node of the model. Labels are assigned with a randomly generated RGB color. This ensures that each class has a unique color which is helpful when plotting the bounding boxes of various detected objects in a frame.

Depending on the model that is being used, the `CreatePipeline` function returns a specific implementation of the object detection pipeline.

5.3 Create a network

All operations with Arm NN and networks are encapsulated in **ArmnnNetworkExecutor** class.

5.4 Create the parser and import a graph

Arm NN SDK imports the graph from a file using the appropriate parser.

Arm NN SDK provides parsers for reading graphs from various model formats. The example application focuses on the `.tflite`, `.pb`, and `.onnx` models.

Based on the extension of the provided model file, the corresponding parser is created, and the network file loaded with the `CreateNetworkFromBinaryFile()` method. The parser creates the underlying Arm NN graph.

The example application accepts `.tflite` format model files, using `ITfLiteParser`:

```
#include "armnnTfLiteParser/ITfLiteParser.hpp"

armnnTfLiteParser::ITfLiteParserPtr parser =
      armnnTfLiteParser::ITfLiteParser::Create();

armnn::INetworkPtr network = parser->CreateNetworkFromBinaryFile(modelPath.c_str());
```

5.5 Optimize the graph for the compute device

Arm NN supports optimized execution on multiple CPU and GPU devices. Before executing a graph, the application must select the appropriate device context. The example application creates a runtime context with default options with `IRuntime()`, as shown in the following code:

```
#include "armnn/ArmNN.hpp"

auto runtime = armnn::IRuntime::Create(armnn::IRuntime::CreationOptions());
```

The application optimizes the imported graph by specifying a list of backends in order of preference and implementing backend-specific optimizations. A unique string identifies each of the backends, for example `CpuAcc`, `GpuAcc`, `CpuRef`.

For example, the example application specifies backend optimizations, as shown in the following code:

```
std::vector<armnn::BackendId> backends{"CpuAcc", "GpuAcc", "CpuRef"};
```

Internally and transparently, Arm NN splits the graph into subgraphs based on the specified backends. Arm NN optimizes each of the subgraphs and, if possible, substitutes the corresponding subgraph in the original graph with its optimized version.

The application uses the `Optimize()` function to optimize the graph for inference, then loads the optimized network onto the compute device with `LoadNetwork()`. The `LoadNetwork()` function creates:

- The backend-specific workloads for the layers
- A backend-specific workload factory which creates the workloads.

The example application contains the following code:

```
armnn::IOptimizedNetworkPtr optNet = Optimize(*network,
                                              backends,
                                              m_Runtime->GetDeviceSpec(),
                                              armnn::OptimizerOptions());

std::string errorMessage;

runtime->LoadNetwork(0, std::move(optNet), errorMessage));

std::cerr << errorMessage << std::endl;
```

5.6 Create input and output binding information

Parsers can also extract input information for the network. The application calls `GetSubgraphInputTensorNames` to extract all the input names, then `GetNetworkInputBindingInfo` binds the input points of the graph. The example application contains the following code:

```
std::vector<std::string> inputNames = parser->GetSubgraphInputTensorNames(0);

auto inputBindingInfo = parser->GetNetworkInputBindingInfo(0, inputNames[0]);
```

The input binding information contains all the essential information about the input.

This information is a tuple consisting of:

- Integer identifiers for bindable layers
- Tensor information including:
  - Data type
  - Quantization information
  - Number of dimensions
  - Total number of elements

Similarly, the application gets the output binding information for an output layer by using the parser to retrieve output tensor names and calling `GetNetworkOutputBindingInfo()`.

5.7 Object detection pipeline

The generic object detection pipeline contains the following three steps:

1. Perform data pre-processing.
2. Run inference.
3. Decode inference results in the post-processing step.

See **ObjDetectionPipeline** and the implementations for **MobileNetSSDv1** and **YoloV3Tiny** for more details.

5.8 Preprocess the captured frame

The application reads each frame captured from source as a `cv::Mat` in BGR format. The channels are swapped to RGB in frame reader code, as follows:

```
cv::Mat processed;
...
objectDetectionPipeline->PreProcessing(frame, processed);
```

The preprocessing step consists of resizing the frame to the required resolution, padding, and converting the data types to match the model input layer. For example, the example application uses SSD MobileNet V1 which takes an input tensor with shape `[1, 300, 300, 3]` and data type `uint8`.

The preprocessing step returns a `cv::Mat` object containing data ready for inference.

5.9 Execute inference

The following code shows how the application executes inference:

```
od::InferenceResults results;

...

objectDetectionPipeline->Inference(processed, results);
```

The inference step calls the `ArmnnNetworkExecutor::Run` method that prepares input tensors and executes inference. A compute device performs inference for the loaded network using the `EnqueueWorkload()` function of the runtime context. For example:

```
//const void* inputData = ...;

//outputTensors were pre-allocated before


armnn::InputTensors inputTensors = {{
     inputBindingInfo.first,armnn::ConstTensor(inputBindingInfo.second, inputData)}};

runtime->EnqueueWorkload(0, inputTensors, outputTensors);
```

The application allocates memory for output data once and maps it to output tensor objects. After successful inference, the application reads data from the pre-allocated output data buffer. See **ArmnnNetworkExecutor::ArmnnNetworkExecutor** and **ArmnnNetworkExecutor::Run** for more information.

5.10 Decode and process the inference output

The application must decode the output from inference to obtain information about the detected objects in the frame. The example application contains implementations for two networks, or you can implement your own network decoding solution.

For SSD MobileNet V1 models, the application decodes the results to obtain the bounding box positions, classification index, confidence, and number of detections in the input frame. See **SSDResultDecoder** for more details.

For YOLO V3 Tiny models, the application decodes the output and performs non-maximum suppression. This suppression filters out weak detections below a confidence threshold and any redundant bounding boxes above an intersection-over-union (IoU) threshold. See **YoloResultDecoder** for more details.

Experiment with different threshold values for confidence and IoU to achieve the best visual results.

The detection results are always returned as a vector of `DetectedObject`, with the box positions list containing bounding box coordinates in the following form:

```
[x_min, y_min, x_max, y_max]
```

5.11 Draw the bounding boxes

The post-processing step accepts a callback function which is invoked when decoding finishes. The application uses this callback function to draw detections on the initial frame. The example application uses the output detections and the `AddInferenceOutputToFrame` function to draw bounding boxes around detected objects and add the associated label and confidence score. The following code shows the post-processing step in detail:

```
objectDetectionPipeline->PostProcessing(results,
   [&frame, &labels](od::DetectedObjects detects) -> void {
        AddInferenceOutputToFrame(detects, *frame, labels);
   });
```

The processed frames are written to a file or displayed in a separate window.

5.12 Run the application

After building the application executable, you can run it with the following command-line options:

- `--video-file-path`: Specifies the path to the video file. This option is required.
- `--model-file-path`: Specifies the path to the object detection model. This option is required.
- `--label-path`: Specifies the path to the label set for the model file. This option is required.
- `--model-name`: Specifies the name of the model used for object detection. Valid values are `SSD_MOBILE` and `YOLO_V3_TINY`. This option is required.
- `--output-video-file-path`: Specifies the path to the output video file. This is optional. The default is `/tmp/output.avi`.
- `--preferred-backends`: Specifies the backends in preference order, separated by a comma. Valid values include `CpuAcc`, `CpuRef`, and `GpuAcc`. This is optional. The default is `CpuRef`, the reference kernel on CPU.
- `--help`: Displays all the available command-line options.

To run object detection on a video file and output the result to another video file, use the following commands:

```
LD_LIBRARY_PATH=/path/to/armnn/libs:/path/to/opencv/libs
./object_detection_example --label-path /path/to/labels/file
      --video-file-path /path/to/video/file --model-file-path /path/to/model/file
      --model-name [YOLO_V3_TINY | SSD_MOBILE]
      --output-video-file-path /path/to/output/file
```

To run object detection on a video file and output the result to a window GUI, use the following commands:

```
LD_LIBRARY_PATH=/path/to/armnn/libs:/path/to/opencv/libs
./object_detection_example --label-path /path/to/labels/file
      --video-file-path /path/to/video/file --model-file-path /path/to/model/file
      --model-name [YOLO_V3_TINY | SSD_MOBILE]
```

6 Related information

Here are some resources related to material in this guide:

- Joseph Redmon, Ali Farhadi. **YOLOv3: An incremental improvement**
- Joseph Redmon, **You only look once: Unified, real-time object detection.** CVPR 2016
- Wei Liu et al. **SSD: Single Shot MultiBox Detector.** ECCV 2016

7 Next steps

In this guide, we examined sample applications that use the Arm NN C++ API and the PyArmNN Python extension to create a real-time object detection system.

You can now use the knowledge that you have gained to write your own applications.

To learn more, you can read our many **guides related to AI and Machine Learning** to learn more about PyArmNN and Arm NN.