# Lab 1: Run image classification model on Raspberry Pi for fire detection

In this lab, we are going to use Arm NN as the inference engine to run an image classification model for fire detection. We already installed Arm NN build environment for TensorFlow Lite on your Raspberry Pi /home/pi/armnn-dist.

From your Raspberry Pi console, add ArmNN library location to the system library environment variable by using the command below.

**$** export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/home/pi/armnn-dist/armnn/lib

armnnTfLiteParser is a library for loading neural networks defined by TensorFlow Lite FlatBuffers files into the Arm NN runtime. TensorFlow Lite operators supported by Arm NN SDK can be found [here](https://github.com/ARM-software/armnn/blob/branches/armnn_19_08/src/armnnTfLiteParser/TensorFlowLiteSupport.md).

All source code for this lab is under /home/pi/Documents/arm-workshop/lab1. Here is the structure of the directory.

* fire\_detection.tflite: Courtesy of this [tutorial](https://www.pyimagesearch.com/2019/11/18/fire-and-smoke-detection-with-keras-and-deep-learning/), we trained a TensorFlow Lite fire detection model.
* fire\_detection.cpp: Arm NN C++ template file. Attendees need to follow the instructions below to fill in the missing Arm NN parameters.
* fire\_detection.cpp.final: Final fire\_detection.cpp implementation.
* fire\_detection.py: Arm NN Python template file. Attendees need to follow the instructions below to fill in the missing parameters.
* fire\_detection.py.final: Final fire\_detection.py implementation.
* Makefile: Makefile for fire\_detection.cpp.
* labels.txt: Image classification label file.

## 1.1 Image classification model

In this exercise, we are going to use the TFLite parser provided by Arm NN to parse the different layers in the fire\_detection.tflite model for “Fire” vs. “Non-Fire” image classification.

## 1.2 Use Arm NN for on-device ML inference

Here are the steps to deploy and run a TensorFlow Lite model with the Arm NN SDK.

* Load the model output labels.
* Load and pre-process an input image.
* Create a parser and load the network.
* Choose backends, create runtime and optimize the model.
* Perform Inference.
* Interpret and report the output.

Now, we will walk you through each step.

### Load the model output labels

You must use the model output labels to interpret the outputs of the model. These labels are usually in a text file the model creator or distributor provides. In this file, each line contains the label or labels corresponding to each output node. In our lab, the file is located at /home/pi/Documents/arm-workshop/lab1/labels.txt. The file only contains 2 lines:

0:Non-Fire

1:Fire

In model\_output\_labels\_loader.hpp file, we define loadModelOutputLabels function to load the label file. The following code invokes the function.

const std::vector<CategoryNames> modelOutputLabels =

**LoadModelOutputLabels(programOptions.modelOutputLabelsPath);**

### Load and pre-process an input image

You must pre-process images before the model can use them as inputs. The pre-processing method that you use depends on the framework, model, or model data type you use.

Below is the structure of our fire detection model. Its input is a Separable Conversion 2D layer, with id ‘separable\_conv2d\_input’. Its output is a softmax activation layer, with id ‘activation\_6/Softmax’. The model properties are extracted by using Tensorboard, a visualization tool to inspect your models.

A screenshot of a cell phone

Description automatically generated

Input layer

A screenshot of a cell phone

Description automatically generated

Output layer

Under your device /home/pi/Documents/arm\_workshop/lab1, you can find a file with the name fire\_detection.cpp. That’s the file we will start with.

1. Resize the input images to match the dimensions of the input tensor of the model. In our case, our model accepts 128x128 input images.
2. Our model is a floating-point model. We must scale the input image values to a range of -1 to 1.
3. Use the C++ operation static\_cast to convert the input image values from floating point to 8-bit unsigned integer type.

In your fire\_detection.cpp, look for the “/\* Please define your model name and input image dimensions here \*/” and adjusted the parameters as below.

const std::string inputName = "**separable\_conv2d\_input**";

const std::string outputName = "**activation\_6/Softmax**";

const unsigned int inputTensorWidth = **128**;

const unsigned int inputTensorHeight = **128**;

const unsigned int inputTensorBatchSize = 1;

const armnn::DataLayout inputTensorDataLayout = armnn::DataLayout::NHWC;

The normParams variable determines how the input image is normalized. The following pseudocode shows how the image pre-processor within the PrepareImageTensor utility function calculates normalized image values:

out = ((in / scale) – mean) / stddev

Therefore, we define the normParams variable as follows:

// Prepare image normalization parameters

normParams.scale = 127.5;

normParams.mean = { 1.0, 1.0, 1.0 };

normParams.stddev = { 1.0, 1.0, 1.0 };

The following code loads and pre-processes an image the command-line option *imagePath* specifies:

// Load and preprocess input image

const std::vector<TContainer> inputDataContainers =

{ PrepareImageTensor<uint8\_t>(programOptions.imagePath,

 inputTensorWidth, inputTensorHeight,

 normParams,

 inputTensorBatchSize,

 inputTensorDataLayout) } ;

### Create a parser and load the network

The next step when working with Armn NN is to create a parser object that will be used to load your network file. Arm NN has parsers for a variety of model file types, including TFLite, ONNX, Caffe etc. Parsers handle creation of the underlying Arm NN graph so you don't need to construct your model graph by hand.

In this example, we will create a TfLite parser to load our TensorFlow Lite model from the specified path:

Search for “//Import the TensorFlowLite model” in the code to create a parser object and load the mordel.

// Import the TensorFlowLite model.

using IParser = **armnnTfLiteParser::ITfLiteParser**;

auto armnnparser(IParser::Create());

armnn::INetworkPtr network = **armnnparser->CreateNetworkFrom BinaryFile(programOptions.modelPath.c\_str())**;

Once created, the parser extracts the input information of the network.

The input binding info contains all the essential information about the input, it's a tuple consisting of integer identifiers for bindable layers (inputs, outputs) and the tensor info (data type, quantization information, number of dimensions, total number of elements).

The following code binds the input and output tensors to the data and selects the loaded network identifier:

// Find the binding points for the input and output nodes

using BindingPointInfo = armnnTfLiteParser::BindingPointInfo;

const std::vector<BindingPointInfo> inputBindings = { **armnnparser->GetNetworkInputBindingInfo(0, inputName)** };

const std::vector<BindingPointInfo> outputBindings = { **armnnparser->GetNetworkOutputBindingInfo(0, outputName)** };

### Prepare the output tensor

You must prepare a container to receive the output of the model. The output tensor size is equal to the number of model output labels, This is implemented as below:

// Output tensor size is equal to the number of model output labels

const unsigned int outputNumElements = **modelOutputLabels.size()**;

std::vector<TContainer> outputDataContainers = { std::vector<uint8\_t>(outputNumElements)};

Choose backends, create runtime and optimize the model

You must optimize your network and load it onto a compute device. The Arm NN SDK supports optimized execution backends on Arm CPU, Mali GPU and DSP devices. Backends are identified by a string that must be unique across backends, you can specify one or more backend in order of preference. In our code, Arm NN decides which layers are supported by the backend, first the CPU is checked, if one or more of the layers can't be run on CPU it will fallback first to the reference implementation.

After specifying the backend list you can create a runtime, and optimize the network in the runtime context. The backends may choose to implement backend-specific optimizations. Arm NN splits the graph into subgraphs based on backends, it calls a optimize subgraph function on each of them, and substitutes the corresponding sub-graph in the original graph with its optimized version when possible.

Once this is done, LoadNetwork creates the backend-specific workloads for the layers, it creates a backend specific workload factory and calls this to create the workloads. The input image is wrapped in a const tensor and bound to the input tensor.

// Create a runtime and optimize the network for a specific compute device,

// e.g. CpuAcc

armnn::IRuntime::CreationOptions options;

armnn::IRunTimePtr runtime(armnn::IRuntime::Create(options));

armnn::IOptimizedNetworkPtr optimizedNet = armnn::Optimize(\*network, {**armnn::Compute::CpuAcc, armnn::Compute::CpuRef**}, runtime->GetDeviceSpec());

// Load the optimized network onto the device

armnn::NetworkId networkId;

runtime->LoadNetwork(networkId, std::move(optimizedNet));

### Perform Inference

A compute device performs inference using the *EnqueueWorkload()* function. The following code runs a single inference on the test image:

runtime->EnqueueWorkload(networkId,

**armnnUtils::MakeInputTensors(inputBindings, inputDataContainers)**,

**armnnUtils::MakeOutputTensors(outputBindings, outputDataContainers)**);

### Interpret and report the output

The output of the model is a tensor of the same size as the number of output labels. The size of the our fire detection tensor is 2. You must interpret each value as the probability of the input image being classified as the corresponding label. To find the label that the model predicts most confidently, you must find the label of the output node with the highest output value.

The std::distance() function in the following example code is used to find the index of the largest element in the output. This function is equivalent to the argmax() function from the NumPy library.

std::vector<uint8\_t> output = boost::get<std::vector<uint8\_t>>(outputDataContainers[0]);

size\_t labelInd = std::distance(output.begin(), std::max\_element(output.begin(),output.end()));

std::cout << "Prediction: ";

for (const auto& label : modelOutputLabels[labelInd])

{

  std::cout << label << ", ";

}

std:: cout << std::endl;

### Compile the code and detect fire

To compile the code, just run make from command line. This will generate fire\_detection binary.

**$** make

**$** export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/home/pi/armnn-dist/armnn/lib

Under the images/ folder under our lab1 directory, we already included some images for you to run inference with. We you launch your application, you can pass on the image filenames as command line parameters for ArmNN to parse.

A group of people in a field with a mountain in the background

Description automatically generated

**$**./fire\_detection -m ./fire\_detection.tflite -d images/opencountry\_land660.jpg -p labels.txt

ArmNN v20190800

Running network...

Prediction: Non-Fire,

Ran successfully!

A lot of smoke around it

Description automatically generated

**$**./fire\_detection -m ./fire\_detection.tflite -d images/350.jpg -p labels.txt

ArmNN v20190800

Running network...

Prediction: Fire,

Ran successfully!

## 1.3 Converting to PyArmNN API (Beta Release)

A lot of developers are more comfortable developing with Python APIs. To give these developers the opportunity exploring the power of Arm NN, we are adding Python APIs to Arm NN.

### Import pyarmnn module

import PIL

from PIL import Image

import pyarmnn as ann

import numpy as np

import cv2

print(f"Working with ARMNN {ann.ARMNN\_VERSION}")

Use the variables below to define the location of our model, image and label file.

### Load and pre-process an input image

Load the image specified at IMAGE\_PATH, resize it to the model input dimension. The input image is wrapped in a const tensor and bound to the input tensor.

parser = argparse.ArgumentParser(

formatter\_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add\_argument(

'--image', help='File path of image file', required=True)

args = parser.parse\_args()

# Load an image

img = cv2.imread(args.image)

img = cv2.resize(img, (**128, 128**))

img = np.array(img, **dtype=np.float32**) / **255.0**

print(img.size)

### Create a parser and load the network

Create a parser object that will be used to load your network file.

parser = **ann.ITfLiteParser()**

network = **parser.CreateNetworkFromBinaryFile('./fire\_detection.tflite')**

### Get Input Binding Info

Once created, the parser is used to extract the input information for the network.

We can extract all the input names by calling *GetSubgraphInputTensorNames*() and then use them get the input binding information. For this example, since our model only has one input layer, we use input\_names[0] to obtain the input tensor, then use this string to retrieve the input binding info.

The input binding info contains all the essential information about the input, it is a tuple consisting of integer identifiers for bindable layers (inputs, outputs) and the tensor info (data type, quantization information, number of dimensions, total number of elements).

graph\_id = 0

input\_names = parser.GetSubgraphInputTensorNames(graph\_id)

input\_binding\_info = parser.GetNetworkInputBindingInfo(graph\_id, input\_names[0])

input\_tensor\_id = input\_binding\_info[0]

input\_tensor\_info = input\_binding\_info[1]

print(f"""

tensor id: {input\_tensor\_id},

tensor info: {input\_tensor\_info}

""")

Choose backends, create runtime and optimize the model

Specify the backend list you can optimize the network.

options = ann.CreationOptions()

runtime = ann.IRuntime(options)

# Backend choices earlier in the list have higher preference.

preferredBackends = [**ann.BackendId('CpuAcc'), ann.BackendId('CpuRef')**]

opt\_network, messages = ann.Optimize(network, preferredBackends, runtime.GetDeviceSpec(), ann.OptimizerOptions())

Load optimized network into the runtime

Load the optimized network in the runtime context. *LoadNetwork*() creates the backend-specific workloads for the layers.

# Load the optimized network into the runtime.

net\_id, \_ = runtime.LoadNetwork(opt\_network)

print(f"Loaded network, id={net\_id}")

input\_tensors = ann.make\_input\_tensors([input\_binding\_info], [img])

Get output binding info and make output tensor

Similar to the input binding info, we can retrieve from the parser the output tensor names and get the binding information.

In our sample, it is considered that an image classification model has only one output, hence it's used only the first name from the list returned, it can easily be extended to multiple output looping on the output\_names.

# Get output binding information for an output layer by using the layer name.

output\_names = parser.GetSubgraphOutputTensorNames(graph\_id)

output\_binding\_info = parser.GetNetworkOutputBindingInfo(0, output\_names[0])

output\_tensors = ann.make\_output\_tensors([output\_binding\_info])

Perform Inference

Performance Inference EnqueueWorkload() function of the runtime context executes the inference for the network loaded.

runtime.EnqueueWorkload(**0, input\_tensors, output\_tensors**)

output, output\_tensor\_info = ann.from\_output\_tensor(output\_tensors[0][1])

print(f"Output tensor info: {output\_tensor\_info}")

Run the Python script from command line:

A large green field with trees in the background

Description automatically generated

**$** python3 fire\_detection.py --image ./images/opencountry\_land663.jpg

Working with ARMNN 20190800

tensor id: 15616,

tensor info: TensorInfo{DataType: 1, IsQuantized: 0, QuantizationScale: 0.000000, QuantizationOffset: 0, NumDimensions: 4, NumElements: 49152}

(128, 128, 3)

Output tensor info: TensorInfo{DataType: 1, IsQuantized: 0, QuantizationScale: 0.000000, QuantizationOffset: 0, NumDimensions: 2, NumElements: 2}

[0.9967675, 0.00323252]

Non-Fire

In our example, class 0’s possibility is 0.9967675, vs. class 1’s possibility is 0.00323252, fire is not detected in the image.

### 1.4 PyArmNN vs. TensorFlow Lite Performance Benchmarking

As the next step, we benchmark PyArmNN and TensorFlow Lite Python APIs performance on the Raspberry Pi.

For performance benchmarking, inference was carried out with our fire detection model. We only run inference once. We can also run the model multiple times and take the average inferencing time.

In fire\_detection.py, extend it to assess the inference time. Look for

**from timeit import default\_timer as timer**

**start = timer()**

runtime.EnqueueWorkload(0, input\_tensors, output\_tensors)

**end = timer()**

**print('Elapsed time is ', (end - start) \* 1000, 'ms')**

Run the script again.

**$** python3 predict\_pyarmnn.py --image ./images/opencountry\_land663.jpg

Working with ARMNN 20190800

(128, 128, 3)

tensor id: 15616,

tensor info: TensorInfo{DataType: 1, IsQuantized: 0, QuantizationScale: 0.000000, QuantizationOffset: 0, NumDimensions: 4, NumElements: 49152}

Loaded network, id=0

**Elapsed time is 28.630815009819344 ms**

Output tensor info: TensorInfo{DataType: 1, IsQuantized: 0, QuantizationScale: 0.000000, QuantizationOffset: 0, NumDimensions: 2, NumElements: 2}

[0.9967675, 0.0032325124]

Non-Fire

TensorFlow Lite uses interpreter to perform an inference. The interpreter uses a static graph ordering and a custom(less-dynamic) memory allocator. To understand how to load and run a model with Python API, please refer to TensorFlow Lite [documentation](https://www.tensorflow.org/lite/guide/inference).

Under the same directory, detect\_tflite.py is a script that uses TFLite interpreter to parse the our fire detection model. Run the script as below and compare its inference performance with PyArmNN.

**$** python3 predict\_tflite.py --image ./images/opencountry\_land663.jpg

**Elapsed time is 36.57340400968678 ms**

[[0.9967675 0.00323252]]

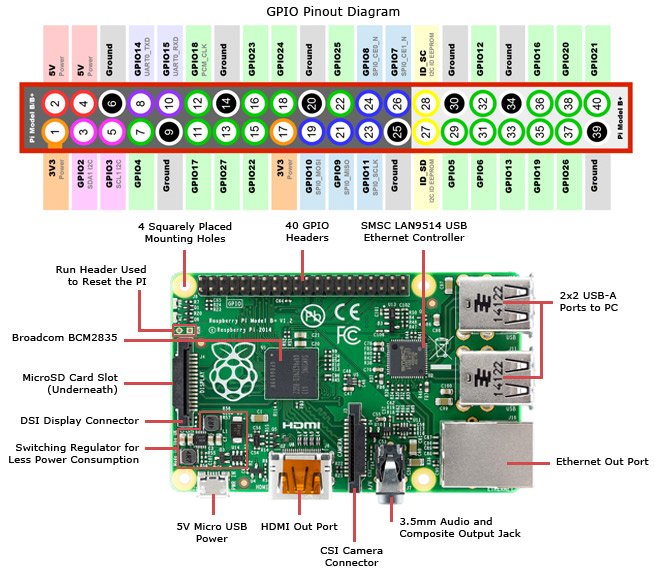
Non-Fire

# Lab 2: Run time series ML model on Raspberry Pi for prediction

In this lab, we will collect temperature data from the sensor, write it to local InfluxDB and use Grafana for visualization. We will use Arm NN as the inference engine to run a time series ML model for sensor data prediction.

## 2.1 Collect Sensor Data, Integrate with InfluxDB and Grafana

### Wire up your DHT11 temperature & humidity sensor



Connect the GND of your sensor to pin 9, Vcc to pin 1 and data pin to pin 8.

### Read sensor data

Under /home/pi/Documents/arm\_workshop/lab2 directory on your Raspberry Pi, sensor.py collects temperature, humidity sensor data every 3 secs.

Run the script and you can see real-time sensor data:

**pi@raspberrypi**:**~/Documents/arm\_workshop/lab2 $** python3 sensor.py

Temp:18.0 C Humidity:51.0%

Temp:18.0 C Humidity:48.0%

Temp:18.0 C Humidity:48.0%

Temp:18.0 C Humidity:48.0%

Temp:18.0 C Humidity:48.0%

### Write real-time sensor data to local InfluxDB

On your Raspberry Pi , we have pre-installed InfluxDB and Grafana for data storage and visualization.

Under /home/pi/Documents/arm\_workshop/lab2 directory on your Raspberry Pi, sensor\_influxdb.py extends the script above, and writes the sensor reading into local InfluxDB.

#!/usr/bin/python

import Adafruit\_DHT

import time

DHT\_SENSOR = Adafruit\_DHT.DHT11

DHT\_PIN = 14

while True:

humidity, temperature = Adafruit\_DHT.read(DHT\_SENSOR, DHT\_PIN)

if humidity is not None and temperature is not None:

print('Temp:{0:0.1f} C Humidity:{1:0.1f}%'.format(temperature, humidity))

else:

print("Sensor failure. Check wiring.")

time.sleep(3)

### Configure Grafana for visualization

To start Grafana as a background service, type in the commands below from your Raspberry Pi console. The last command verifies Grafana has been started properly and its status should show as “active(running)”

**$** sudo systemctl daemon-reload

**$** sudo systemctl start grafana-server

**$** sudo systemctl status grafana-server

**●** grafana-server.service - Grafana instance

Loaded: loaded (/usr/lib/systemd/system/grafana-server.service; disabled; vendor preset: enabled)

Active: **active (running)** since Thu 2019-11-21 21:04:12 PST; 2 days ago

Docs: http://docs.grafana.org

Main PID: 2838 (grafana-server)

Tasks: 15 (limit: 1599)

Memory: 25.9M

CGroup: /system.slice/grafana-server.service

└─2838 /usr/sbin/grafana-server --config=/etc/grafana/grafana.ini --pidfile=/var/run/grafana/g

Nov 21 21:04:13 raspberrypi grafana-server[2838]: t=2019-11-21T21:04:13-0800 lvl=info msg="Initializing U

Nov 21 21:04:13 raspberrypi grafana-server[2838]: t=2019-11-21T21:04:13-0800 lvl=info msg="Initializing C

Nov 21 21:04:13 raspberrypi grafana-server[2838]: t=2019-11-21T21:04:13-0800 lvl=info msg="Initializing N

Nov 21 21:04:13 raspberrypi grafana-server[2838]: t=2019-11-21T21:04:13-0800 lvl=info msg="Initializing p

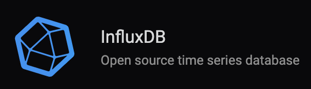
Launch a browser on your Raspberry Pi, and access Grafana web admin portal at localhost port 3000: http://localhost:3000. Use admin/admin to log in.

Select Configuration button from the left pane to configure Data Source

A screenshot of a cell phone screen with text

Description automatically generated

Click A screen shot of a person

Description automatically generated button and select InfluxDB time series database

Add configuration details as below:

* Name should be configured as your InfluxDB measurement name, which is “prediction”
* URL links to the local HTTP InfluxDB REST API interface, which should be <http://localhost:8086>
* Database name is arm\_workshop
* User / password is root/root

A screenshot of a video game

Description automatically generated

Click the “Save and Test” green button at the bottom to verify Grafana’s connection to the database. You should be able to see the message "Data Source is working" if everything is configured properly.

As the next step, we will add a dashboard for visualization. From the left pane of your Grafana portal, select +/Create Dashboard.

A screenshot of a cell phone

Description automatically generated

Click Add Query button.

A screenshot of a cell phone screen with text

Description automatically generated

You should be able to observe the real-time sensor readings from Grafana dashboard.

A screenshot of a video game

Description automatically generated

## 2.2 Use time series model for prediction

### Connect to the InfluxDB

InfluxDB uses port 8086 to run HTTP service. We are going to use HTTP to retrieve the last 3 historical data. Database name is “arm\_workshop” and user/password are *root*/*root*.

influxdb\_cpp::server\_info server("127.0.0.1", 8086, "arm\_workshop", "root", "root");

### Define input and output elements

We are going to feed 3 historical data into our ML model and generate 1 prediction, so we define inputNumElements as 3, and outputNumElements as 1.

// Specify the size and type of inputs and outputs

static constexpr size\_t inputNumElements = 3;

static constexpr size\_t outputNumElements = 1;

### Create a parser and load the network

In this example, we are using TFLite model so we create the TfLite parser to load the model from the specified path.

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

armnn::INetworkPtr network = parser->CreateNetworkFromBinaryFile("fire\_prediction.tflite");

Once created, the parser extracts the input information of the network.

Our model has just one input layer called "dense\_1\_input ", and an output layer called "dense\_4/BiasAdd".

The following code binds the input and output tensors to the data and selects the loaded network identifier:

// Find the binding points for the input and output nodes

using BindingPointInfo = armnnTfLiteParser::BindingPointInfo;

const std::vector<BindingPointInfo> inputBindingInfo = { parser->GetNetworkInputBindingInfo(0, "dense\_1\_input") };

const std::vector<BindingPointInfo> outputBindingInfo = { parser->GetNetworkOutputBindingInfo(0, "dense\_4/BiasAdd") };

Choose backends, create runtime and optimize the model

Now, you can specify the backend list, create a runtime, and optimize the network in the runtime context. This part of the code is identical to lab 1.

// Create a runtime and optimize the network for a specific compute device,

// e.g. CpuAcc

armnn::IRuntime::CreationOptions options;

armnn::IRunTimePtr runtime(armnn::IRuntime::Create(options));

armnn::IOptimizedNetworkPtr optimizedNet = armnn::Optimize(\*network, {**armnn::Compute::CpuAcc, armnn::Compute::CpuRef**}, runtime->GetDeviceSpec());

// Load the optimized network onto the device

armnn::NetworkId networkId;

runtime->LoadNetwork(networkId, std::move(optimizedNet));

### Retrieve historical data from InfluxDB and performance reference

for(;;) {

influxdb\_cpp::query(response, "select temperature from fireprediction order by time desc limit 3", server);

std::vector<std::pair<std::string, int>> values = parse(response);

// Populate data containers with loaded data

std::vector<TInputContainer> inputDataContainers = { {

static\_cast<float>(values[0].second),

static\_cast<float>(values[1].second),

static\_cast<float>(values[2].second)

} };

std::cout << "Historical Data: " << values[0].second << " " << values[1].second << " " << values[2].second << " " << std::endl;

std::vector<TOutputContainer> outputDataContainers = { { 0 } };

// Run a single inference on the test image

armnn::Status ret = runtime->EnqueueWorkload(networkIdentifier,

MakeInputTensors(inputBindingInfo, inputDataContainers),

MakeOutputTensors(outputBindingInfo, outputDataContainers));

// Print output

std::cout << outputDataContainers[0][0] << std::endl;

usleep(3000000);

}

### Compile the code and predict

To compile the code, just run make from command line. This will generate prediction binary.

**$** make

**$** make test

LD\_LIBRARY\_PATH=:/home/pi/armnn-dist/armnn/lib ./prediction

Humidity = 79.0 % Temperature = 30.0 \*C (86.0 \*F)

Humidity = 40.0 % Temperature = 31.0 \*C (87.8 \*F)

Historical Data: 31 22 22 22

23.7596

Humidity = 35.0 % Temperature = 31.0 \*C (87.8 \*F)

Historical Data: 31 31 22 22

27.5774

Humidity = 33.0 % Temperature = 30.0 \*C (86.0 \*F)

Historical Data: 30 31 31 22

30.7919

A screenshot of a video game

Description automatically generated