AI on Edge with the NXP i.MX8

A project report submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Technology

by

Ishaan Sethi, Sri Amlan Anshu Mohanty, Tarun Kumar Das
(Entry No. 2020EEB1174, 2020EEB1211, 2020EEB1212)

Under the guidance of
Dr. Suman Kumar

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROPAR
April 2023

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

_____

Ishaan Sethi (2020EEB1174)

Sri Amlan Anshu Mohanty (2020EEB1211)

Tarun Kumar Das (2020EEB1212)

Date: _____

Abstract

This report details the work done by this team during the CP302 Course Project while working with Dr. Suman Kumar. The tasks allotted tackled the problem of using a USB camera to detect bees using deep learning models on Edge, i.e. AI processing at the microprocessor level. The report provides documentation of the building of a Linux image using the Yocto Project and the integration of the eIQ software package developed by NXP, along with demonstration of inference being performed using TensorFlow Lite on a Toradex Apalis i.MX8QM SoM.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The task of counting biodiversity or insect activity in the field is known to be a time-consuming and labor-intensive process. This project is one step towards making this activity easier and more effective.

The objective of this report is to provide documentation of the creation of an= Linux image using the Yocto project and the integration of the NXP eIQ layer into the Yocto build system. This will allow the Toradex Apalis i.MX8 board to count and detect bees (Apis Mellifera).

The various steps and research done to accomplish this goal, such as the tools used, programs written, difficulties faced, etc are described in the report.

Chapter 2

eIQ and Yocto

Our task is to detect bees using a machine learning model on an i.MX8-based system. For this purpose, libraries, inference engines, and hardware drivers are needed along with a operating system. We use the NXP eIQ ML software development environment, which is fully integrated with the Yocto Project, which is used to create customized Linux-based operating systems for embedded systems..

2.1 The NXP® eIQ® Platform

The NXP® eIQ ML software environment provides the necessary components to perform inference with neural network models on embedded systems and to deploy machine learning algorithms on NXP micro-controllers and SoMs.
The software environment combines inference engines, compilers, libraries and hardware abstraction layers to support popular inference engines such as TensorFlow Lite, ONNX Runtime, PyTorch, DeepView RT, and OpenCV.
We have used TensorFlow Lite for our work. It has been developed by Google to provide reduced implementations of TF models, and uses many techiniques to acheive low-latency and high-speed processing such as quantized kernels and fused activations to allow for small and fost models. TF Lite uses the Eigen library to speed-up matrix and vector arithmetic.
TF Lite uses a FlatBuffers-based model file format. FlatBuffers have a smaller memory footprint than TF protocol buffers, which enable greater use of cache lines and faster execution on NXP devices. TF Lite supports recurrent neural networks (RNNs) and long short-term memory networks (LSTM) network designs in addition to an optimized subset of the main neural

network operations.

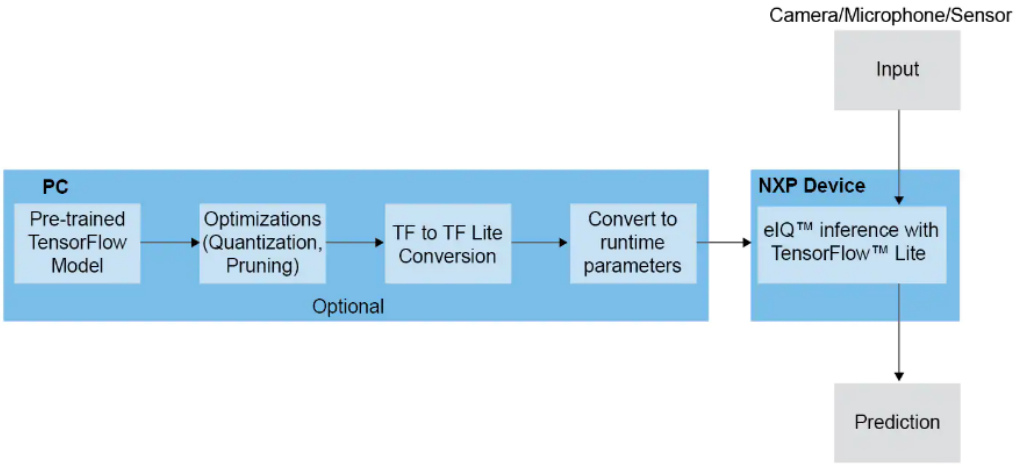

Figure 2.1: eIQ TensorFlow Lite workflow

2.2 The Yocto project

A convenient way to use an SoM like the Toradex i.MX8 module is to install a customized version of Linux on it, allowing the use of well-tested Linux solutions and applications. We have used the default operating system image provided by Toradex and customized it for our needs using the Yocto project.

The Yocto project is a collaborative open-source project with the goal to produce tools and processes that allow for the creation of custom Linux distributions for embedded software independent of the underlying hardware architecture. Both Toradex and NXP provide default operating system images and multiple layers. We are using the Toradex minimal-reference-image for this project.

2.3 Adding the eIQ layer to the Yocto build

The eIQ software is provided as a layer from NXP's Yocto repository: *meta-imx/meta-ml*. We provide the steps followed to add this layer to our Yocto build.

Cloning the Toradex BSP

We create a directory called *yocto-ml-build* and obtain a suitable version of the base repository used by Toradex, which will serve as the foundation of our Yocto build system.

```
mkdir -p ./yocto-ml-build/bsp-toradex && cd ./yocto-ml-build/bsp-toradex
repo init -u https://git.toradex.com/toradex-manifest.git -b
 ↪  refs/tags/5.7.0-devel-202206 -m tdxref/default.xml
repo sync
```

Adding the eIQ layer

In order to add the eIQ layer and recipes, we first move to the */bsp-toradex* directory, and enter the following commands -

```
git clone --depth 1 -b kirkstone-5.15.32-2.0.0
 ↪  git://source.codeaurora.org/external/imx/meta-imx ../meta-imx
git clone --depth 1 -b dunfell
 ↪  https://github.com/priv-kweihmann/meta-sca.git ../meta-sca
git clone --depth 1 -b kirkstone
 ↪  git://git.openembedded.org/openembedded-core
 ↪  ../openembedded-core-kirkstone
```

This code clones the required repositories to our current folder.

Setting up the environment

In order to setup the environment, we simply need to use one command -

```
source ./export
```

This command starts the bitbake environment, which is necessary for the next steps.

Adding layers and copying recipes

Now we must add the layers from the downloaded directories to our build system. For that we use the bitbake program, which takes care of adding and keeping track of layers.

```
bitbake-layers create-layer ../layers/meta-ml
bitbake-layers add-layer ../layers/meta-ml
rm -rf ../layers/meta-ml/recipes-example
cp -r /media/awadh/AWADH_Work/AI_on_Edge_march23/yocto-ml-build/
meta-imx/meta-ml/recipes-* ../layers/meta-ml/
cp -r /media/awadh/AWADH_Work/AI_on_Edge_march23/yocto-ml-build/
meta-imx/meta-bsp/recipes-support/opencv
 ↪  ../layers/meta-ml/recipes-libraries/
cp -r ../../meta-sca/recipes-python/python-pybind11-native
 ↪  ../layers/meta-ml/recipes-libraries/
cp -r ../../openembedded-core-kirkstone/meta/recipes-devtools/cmake
 ↪  ../layers/meta-ml/recipes-devtools/
rm -rf ../layers/meta-openembedded/meta-oe/recipes-devtools/cmake
sed -i 's/require recipes-support\/opencv\/opencv_4.5.2.imx.bb/require
 ↪  backports\/recipes-support\/opencv\/opencv_4.5.2.imx.bb/g'
 ↪  ../layers/meta-ml/recipes-libraries/opencv/opencv_4.5.4.imx.bb
rm -rf ../layers/meta-openembedded/meta-oe/recipes-devtools/flatbuffers
for file in "../layers/meta-ml/recipes-libraries/arm-compute-library/
arm-compute-library_21.08.bb" "../layers/meta-ml/recipes-libraries/
tensorflow-lite/tensorflow-lite-vx-delegate_2.8.0.bb"
 ↪  "../layers/meta-ml/recipes-libraries/tim-vx/tim-vx_1.1.39.bb"
 ↪  "../layers/meta-ml/recipes-libraries/nn-imx/nn-imx_1.3.0.bb"; do
  echo 'COMPATIBLE_MACHINE:apalis-imx8 = "(apalis-imx8)" ' >> "$file"
  echo 'COMPATIBLE_MACHINE:verdin-imx8mp = "(verdin-imx8mp)" ' >> "$file"
done
sed -i 's/PACKAGECONFIG_VSI_NPU:mx8-nxp-bsp   =
 ↪  "vsi_npu"/PACKAGECONFIG_VSI_NPU:mx8-nxp-bsp   =
 ↪  "vsi_npu"\nPACKAGECONFIG_VSI_NPU:verdin-imx8mp   = "vsi_npu"/g'
 ↪  ../layers/meta-ml/recipes-libraries/onnxruntime/onnxruntime_1.10.0.bb
echo 'IMAGE_INSTALL_append += "tensorflow-lite tensorflow-lite-vx-delegate
 ↪  onnxruntime"' >> conf/local.conf
echo 'IMAGE_INSTALL_append += "opencv python3-pillow v4l2-"' >>
 ↪  conf/local.conf
echo 'IMAGE_INSTALL_remove += "packagegroup-tdx-qt5
 ↪  wayland-qtdemo-launch-cinematicexperience "' >> conf/local.conf
echo 'IMAGE_INSTALL_append += "packagegroup-tdx-graphical
 ↪  packagegroup-fsl-isp v4l-utils"' >> conf/local.conf
echo 'SCA_DEFAULT_PREFERENCE ?= "-1" ' >> conf/local.conf
echo 'PARALLEL_MAKE="-j 18" ' >> conf/local.conf
echo 'BB_NUMBER_THREADS="18"' >> conf/local.conf
echo 'ACCEPT_FSL_EULA = "1"' >> conf/local.conf
```

With these commands we have created the meta-ml layer, copied recipes into it, and made some necessary changes to the configuration files. There is one more change that is required - to set the Machine in the *local.conf* file. We have set the machine variable to *apalis-imx8*, which is the name of the machine we are using.

Building the image

After all the recipes have been added properly, we can build the image file using the bitbake utility -

```
bitbake tdx-reference-minimal-image
```

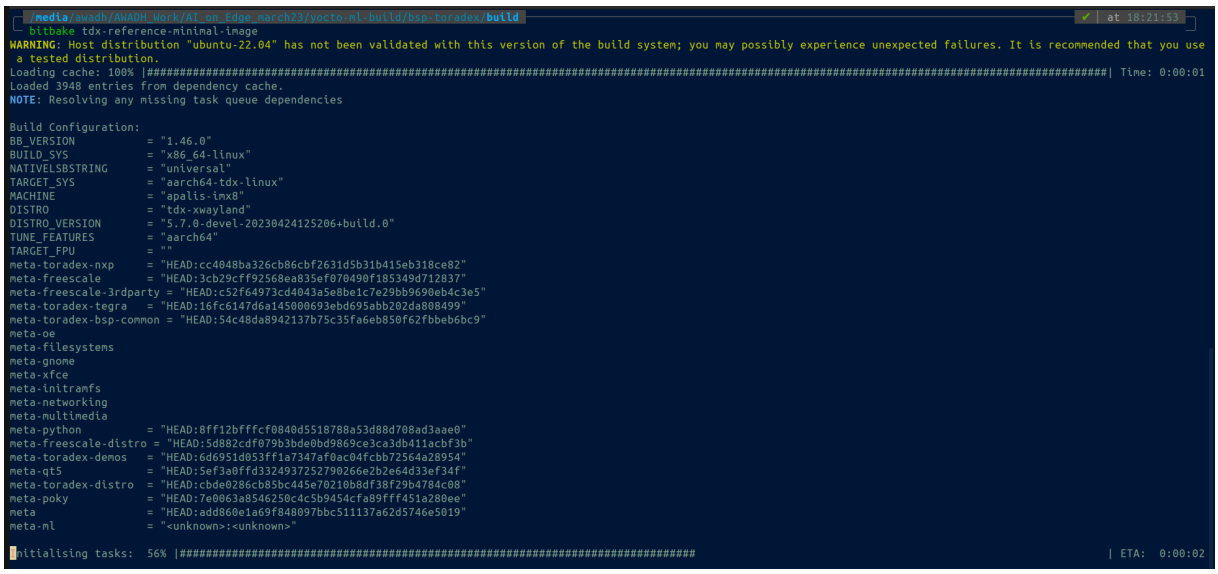

Figure 2.2: Building the reference minimal image

2.4 Flashing the image onto the i.MX8 board

Having built the image using `bitbake`, we untar the *.tar* file present in the `/build/deploy/images` and store it in a pendrive. This pendrive is connected to the upper connector of the X8 port.

We are now ready to install our operating system on the board. Toradex allows this using the Toradex Easy Installer.

In order to start the aforementioned program, we need to put the device in recovery mode and load the program onto RAM using a host PC. We remove the JP2 connector and connect our

host PC to the X9 USB OTG port. This allows the X9 port to be used as an OTG Client. The device setup looks like -

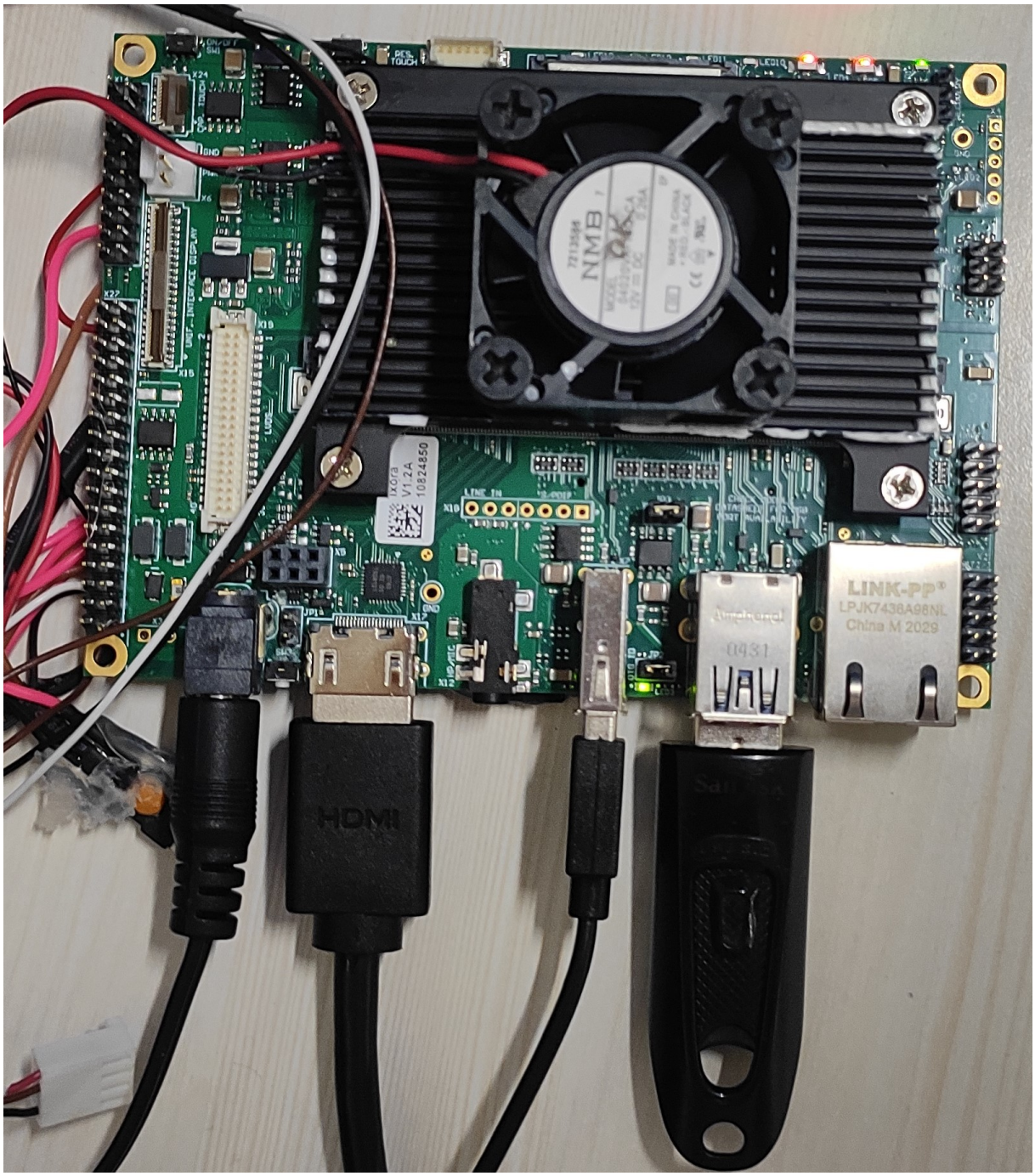

Figure 2.3: Setup for flashing image.

We short two pins on the board using a jumper, which forces the device to enter recovery mode. The jumper is removed after 6 seconds. Once the device is recognized by our host PC, we run the `./recovery-linux.sh` script and the Toradex Easy Installer is flashed onto the board.

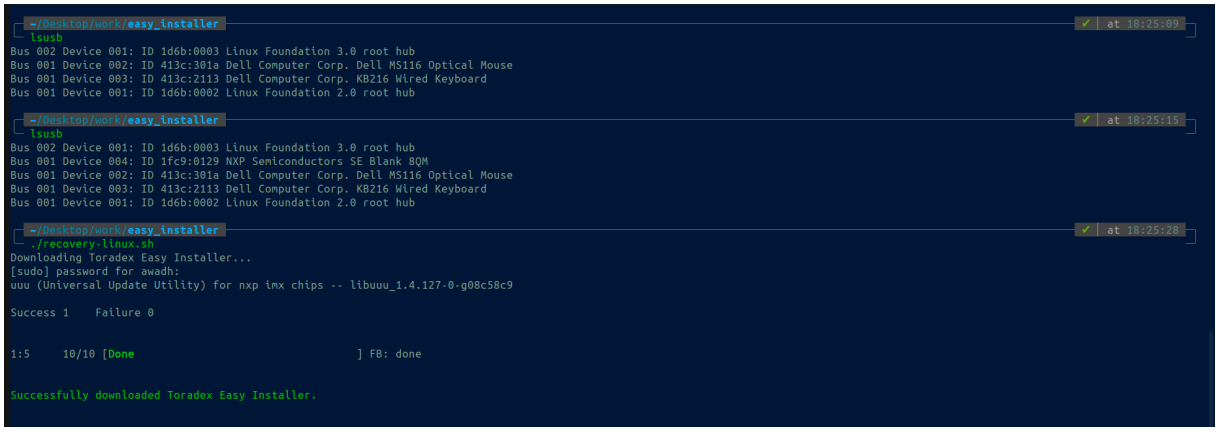

Figure 2.4: Loading the Toradex Easy Installer

On the monitor connected the board, we can see the Toradex Easy Installer.

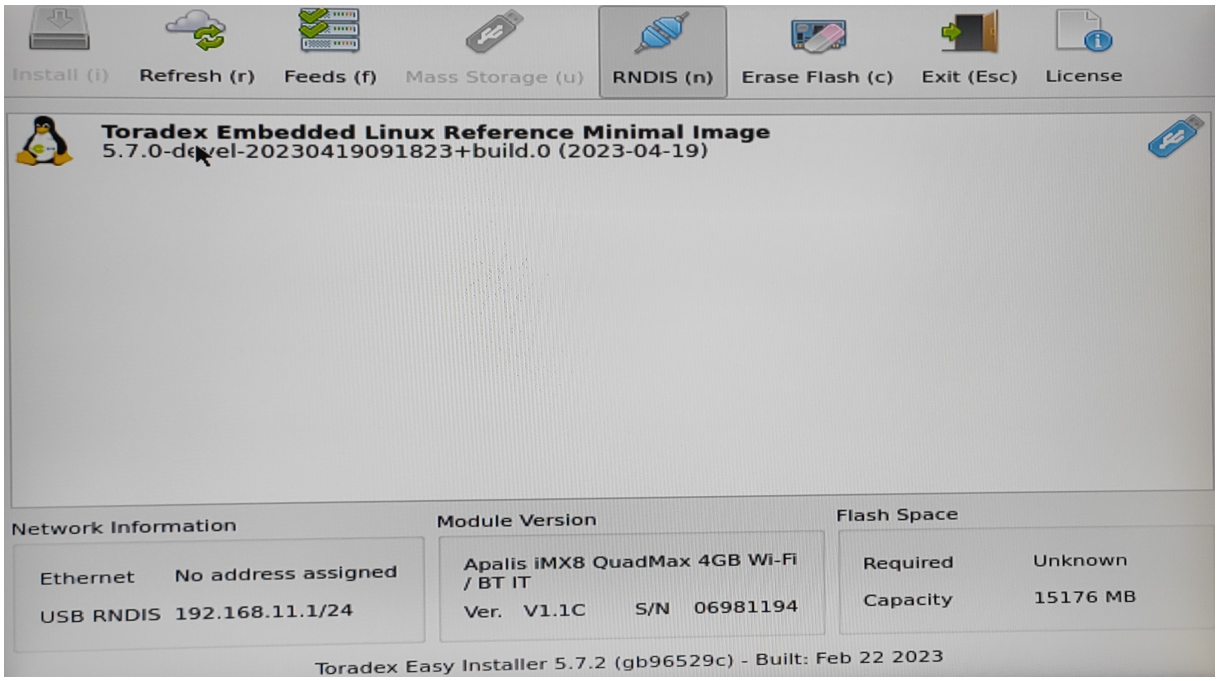

Figure 2.5: The Toradex Easy Installer

We select the reference image option, and the image is installed onto the board.

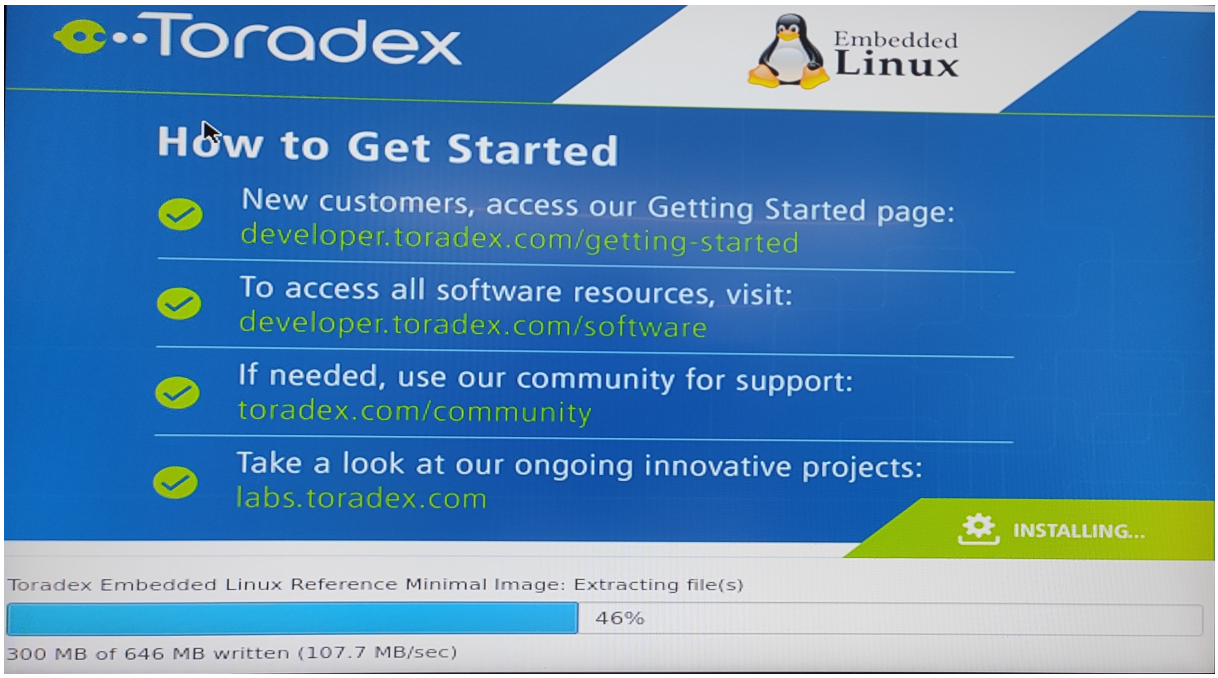

Figure 2.6: Image installation

We are now ready to run our python script and perform inference using TensorFlow Lite.

Chapter 3

Tensorflow Lite on Edge

We are now ready to perform inference on the device using TensorFlow Lite. TensorFlow Lite is based on the well-tested TensorFlow library, and provides convenient methods to create *.tflite* models. We are using the EfficientDet models that have been tested and optimized for microprocessors by TensorFlow developers. For a start, we have worked with the EfficientDet0 model which is lightweight and thus can be run with minimal computation cost on the device. This model was trained on a dataset of Apis M. bees and is not very accurate due to the small model size. We can see the performance of some models below -

EfficientDet Models in TFLite			
Complexity Level	Size in MB	Latency in ms	Accuracy (average)
EfficientDetLite0	4.4	37	25.69%
EfficientDetLite1	5.8	49	30.55%
EfficientDetLite2	7.2	69	33.97%
EfficientDetLite3	11.4	116	37.70%
EfficientDetLite4	19.9	260	41.96%

Table 3.1: Model Architectures

The model architecture for using TFLite through eIQ is shown below -

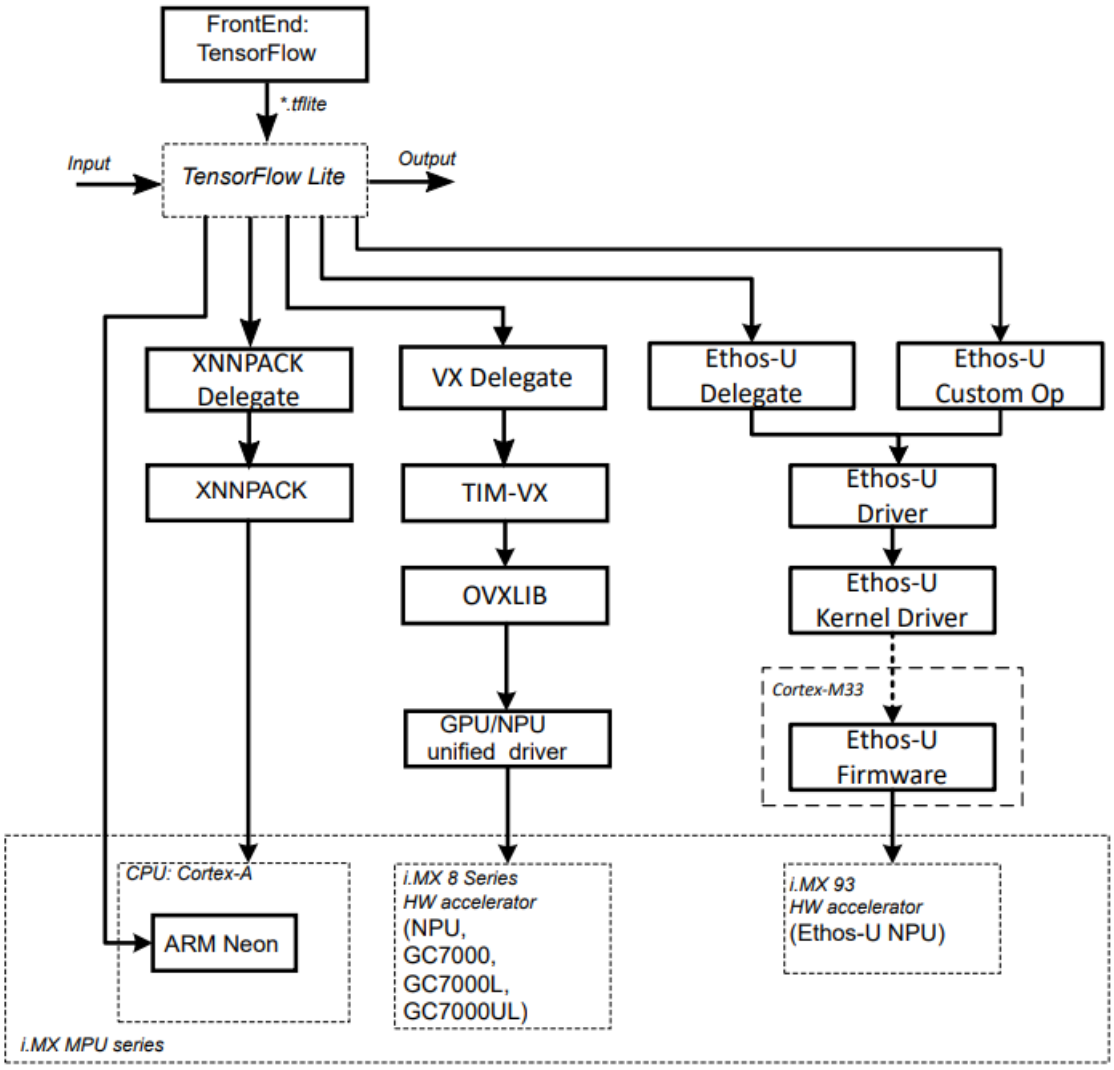

Figure 3.1: eIQ TFLite software architecture for NXP processors

Inference is done using a python script that utilises the `tflite_runtime` library to perform processing and give us an output. The model outputs coordinates of bounding boxes where it has detected bees, and we have used OpenCV to draw boxes and save the output images in a directory.

3.1 Images from a Video

We first tested the script by splitting a video into frames, performing inference, then rejoining the frames together. Using OpenCV, we can split a video into constituent frames -

```
import cv2
import os
import time
st = time.time()
video_path = './vid1.avi'
frames_path = './frames'
video = cv2.VideoCapture(video_path)
while True:
        image_exists, image = video.read()
        cnt = 0
        while image_exists is True:
            if not os.path.isdir(frames_path):
                os.makedirs(frames_path)
            cv2.imwrite(frames_path+'/input%03d.png'%cnt, image)
            image_exists, image = video.read()
            cnt += 1
        else:
                break
end=time.time()
print(end-st)
```

Having obtained a folder of images, we run the TensorFlow code.

3.1.1 Unoptimized Code

First, we import required libraries.

```
import os
import warnings
import cv2
import numpy as np
import tflite_runtime.interpreter as tflite
from PIL import Image
import csv
import time
warnings.filterwarnings('ignore')
cwd = os.getcwd()
```

Then, the script goes through each image, performs inference and saves the output. An interpreter object is created, which when invoked gives the output as a tensor object.

```
MODEL_PATH = './model2apis.tflite'
MODEL_NAME = 'model2apis'
DETECTION_THRESHOLD = 0.01
INPUT_PATH = './frames/input{:03}.png'
OUTPUT_PATH = './frames/output{:03}.png'

bees_per_frame=[]

start = time.time()

for i in range(0,101):

    interpreter =tflite.Interpreter(model_path=MODEL_PATH)
    interpreter.allocate_tensors()

    detection_result_image, cnt = create_output(INPUT_PATH.format(i),
    ↪  interpreter, threshold=DETECTION_THRESHOLD)

    Image.fromarray(detection_result_image).save(OUTPUT_PATH.format(i))

    bees_per_frame.append(cnt)
    print('Image {} done'.format(i))
    os.remove(INPUT_PATH.format(i))

with open('counts', 'w') as myfile:
    wr = csv.writer(myfile, delimiter = '\n',quoting=csv.QUOTE_ALL)
    wr.writerow(bees_per_frame)
end = time.time()
print(end-start)
```

We used the following custom functions to simplify the code -

```
def preprocess_image(image_path, input_size):
    img = cv2.imread(image_path)[:,:, ::-1]
    img = img.astype(np.uint8, copy=False)
    resized_img = (np.expand_dims(cv2.resize(img, input_size,
    ↪  interpolation=cv2.INTER_LINEAR), axis=0)).astype(np.uint8, copy =
    ↪  False)
    return resized_img, img
```

```
def detect_objects(interpreter, image, threshold):
    signature_fn = interpreter.get_signature_runner()

    output = signature_fn(images=image)
    count = int(np.squeeze(output['output_0']))
    scores = np.squeeze(output['output_1'])
    classes = np.squeeze(output['output_2'])
    boxes = np.squeeze(output['output_3'])

    results = [{'bounding_box': boxes[i], 'class_id': classes[i], 'score':
    ↪  scores[i]} for i in range(count) if scores[i] >= threshold]
    return results


def create_output(image_path, interpreter, threshold=0.5):
    _, input_height, input_width, _ =
    ↪  interpreter.get_input_details()[0]['shape']
    preprocessed_image, original_image =
    ↪  preprocess_image(image_path,(input_height, input_width))

    results = detect_objects(interpreter, preprocessed_image,
    ↪  threshold=threshold)

    original_image_np = original_image.astype('uint8')
    for obj in results:
        ymin, xmin, ymax, xmax = obj['bounding_box']
        xmin = int(xmin * original_image_np.shape[1])
        xmax = int(xmax * original_image_np.shape[1])
        ymin = int(ymin * original_image_np.shape[0])
        ymax = int(ymax * original_image_np.shape[0])

        cv2.rectangle(original_image_np, (xmin, ymin), (xmax, ymax),
        ↪  (255,0,0), 1)

        y = ymin - 15 if ymin - 15 > 15 else ymin + 15
        label = "{}: {:.0f}%".format('APIS', obj['score'] * 100)
        cv2.putText(original_image_np, label, (xmin, y),
        ↪  cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,0), 1)
    original_image_np = original_image_np.astype('uint8')
    return original_image_np, len(results)
```

preprocess_image simply resizes the input based on the input tensor shape desired by the model. *detect_objects* performs inference by calling the appropriate functions. *create_output* takes the output tensor and draws boxes on the input image.

Finally, we join back the images using another script -

```python
import cv2
import os
image_folder = './'
video_name = 'video.mp4'
images = [img for img in os.listdir(image_folder) if img.endswith(".png")]
frame = cv2.imread(os.path.join(image_folder, images[0]))
height, width, layers = frame.shape

video = cv2.VideoWriter(video_name, 0, 60, (width,height))

for image in images:
    video.write(cv2.imread(os.path.join(image_folder, image)))

video.release()
```

We can see the output of these scripts on the device:

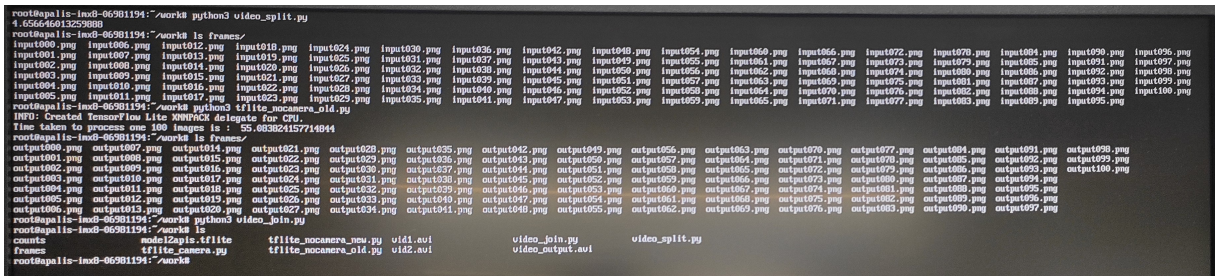

Figure 3.2: Processing images using the Unoptimized script.

We see that the script takes 55.08 seconds to process 100 images, hence 0.55 seconds per image. We can increase this by optimizing our looping code.

3.1.2 Optimized Code

We have optimized the above code by using variables for methods instead of calling them multiple times, which wastes memory and is slower since new objects are created each time a method is called. We have also removed the functions since calling functions multiple times creates new objects, hence slowing down processing.

We can see the new script below -

```
import os
import warnings
import cv2
import numpy as np
import tflite_runtime.interpreter as tflite
import csv
import time
import timeit
warnings.filterwarnings('ignore')
cwd = os.getcwd()

start = time.time()
MODEL_PATH = './model2apis.tflite'
MODEL_NAME = 'model2apis'
DETECTION_THRESHOLD = 0.05
INPUT_PATH = './frames/input{:03}.png'
OUTPUT_PATH = './frames/output{:03}.png'

bees_per_frame=[]

interpreter = tf_lite.Interpreter(model_path=MODEL_PATH)
interpreter.allocate_tensors()
signature_fn = interpreter.get_signature_runner()
detection_result_image, cnt = [], 0
_, input_height, input_width, _ =
↪  interpreter.get_input_details()[0]['shape']

image_read = cv2.imread
image_resize = cv2.resize
expand_dims = np.expand_dims
uint8 = np.uint8
typecast = np.ndarray.astype
squeeze = np.squeeze
draw_rect = cv2.rectangle
draw_text = cv2.putText
fontt = cv2.FONT_HERSHEY_SIMPLEX
append_bees = bees_per_frame.append
```

```
save_image = cv2.imwrite

for i in range(0,101):
    img = image_read(INPUT_PATH.format(i)).astype(uint8)

    resized_img = expand_dims(image_resize(img, (input_height,
    ↪  input_width), interpolation=cv2.INTER_LINEAR), axis=0)

    output = signature_fn(images=resized_img)
    count = int(squeeze(output['output_0']))
    scores = squeeze(output['output_1'])
    boxes = squeeze(output['output_3'])
    results = [{'bounding_box': boxes[i], 'score': scores[i]} for i in
    ↪  range(count) if scores[i] >= DETECTION_THRESHOLD]
    for obj in results:
        ymin, xmin, ymax, xmax = obj['bounding_box']
        xmin = int(xmin * img.shape[1])
        xmax = int(xmax * img.shape[1])
        ymin = int(ymin * img.shape[0])
        ymax = int(ymax * img.shape[0])

        draw_rect(img, (xmin, ymin), (xmax, ymax), (255,0,0), 1)
        y = ymin - 15 if ymin - 15 > 15 else ymin + 15
        label = "{}: {:.0f}%".format('APIS', obj['score'] * 100)
        draw_text(img,label, (xmin, y), fontt, 0.5, (0,0,0), 1)

    save_image(OUTPUT_PATH.format(i), img)

    append_bees(len(results))
    os.remove(INPUT_PATH.format(i))

write_csv = csv.writer
with open('counts', 'w') as myfile:
    wr = write_csv(myfile, delimiter = '\n',quoting=csv.QUOTE_ALL)
    wr.writerow(bees_per_frame)
end = time.time()
print(end-start)
```

We can see the speed of the new code -

Figure 3.3: Processing images using the optimized script.

We see that processing one image takes 0.32 seconds, which is a large improvement from the previous script.

3.2 Images from the Camera

Now that we have been able to perform inference on images obtained from a video, we then take frames directly from a USB Camera attached connected to the device. We used two cameras - auto-focus and fixed-focus.

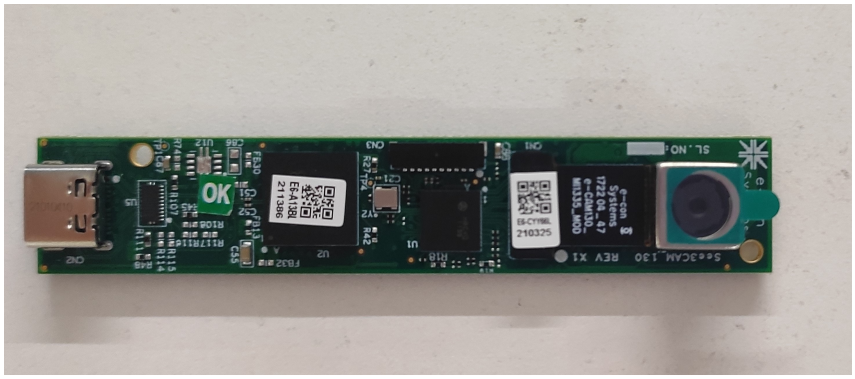

Figure 3.4: Auto-focus camera.

Figure 3.5: Fixed-focus camera.

A while loop is set up and frames are taken from the camera every loop. Every 60 loops we run inference on the frame and save it as output. Since the camera provides frames at 60 frames per second, we do not want to needlessly waste memory saving hundreds of images with largely the same content.

The script is given below -

```
import os
import warnings
import cv2
import numpy as np
import tflite_runtime.interpreter as tflite
import csv
import time
import timeit
import sys
warnings.filterwarnings('ignore')
cwd = os.getcwd()

cap = cv2.VideoCapture("/dev/video2", cv2.CAP_V4L)

MODEL_PATH = './model2apis.tflite'
MODEL_NAME = 'model2apis'
DETECTION_THRESHOLD = 0.05
OUTPUT_PATH = './frames1/output{:03}.png'

interpreter = tflite.Interpreter(model_path=MODEL_PATH)
interpreter.allocate_tensors()
signature_fn = interpreter.get_signature_runner()
detection_result_image, cnt = [], 0
```

```
_, input_height, input_width, _ =
↪  interpreter.get_input_details()[0]['shape']

image_resize = cv2.resize
expand_dims = np.expand_dims
uint8 = np.uint8
typecast = np.ndarray.astype
sqee = np.squeeze
draw_rect = cv2.rectangle
draw_text = cv2.putText
fontt = cv2.FONT_HERSHEY_SIMPLEX
save_image = cv2.imwrite

if cap.isOpened():
    ctr=0
    while True:
        ret_val, img = cap.read()
        start=time.time()
        if ctr%60==0:
            img = img.astype(uint8)
            resized_img = expand_dims(image_resize(img, (input_height,
            ↪  input_width), interpolation=cv2.INTER_LINEAR), axis=0)

            output = signature_fn(images=resized_img)
            count = int(sqee(output['output_0']))
            scores = sqee(output['output_1'])
            boxes = sqee(output['output_3'])
            results = [{'bounding_box': boxes[i], 'score': scores[i]} for
            ↪  i in range(count) if scores[i] >= DETECTION_THRESHOLD]

            for obj in results:
                ymin, xmin, ymax, xmax = obj['bounding_box']
                xmin = int(xmin * img.shape[1])
                xmax = int(xmax * img.shape[1])
                ymin = int(ymin * img.shape[0])
                ymax = int(ymax * img.shape[0])

                draw_rect(img, (xmin, ymin), (xmax, ymax), (255,0,0), 1)
                y = ymin - 15 if ymin - 15 > 15 else ymin + 15
                label = "{}: {:.0f}%".format('APIS', obj['score'] * 100)
                draw_text(img,label, (xmin, y), fontt, 0.5, (0,0,0), 1)

            save_image(OUTPUT_PATH.format(ctr//60), img)
        end=time.time()
        print(end-start)

        ctr+=1
```

```
        cv2.waitKey(1)
else:
    print("camera open failed")
cap.release()
cv2.destroyAllWindows()
```

We take input from /dev/video2, which is the port number of the USB camera. It is a v4l2-based device, so we are using the appropriate function. We have created global variables to speed up the loop and avoid wastage of memory.

The output of this script can be seen below -

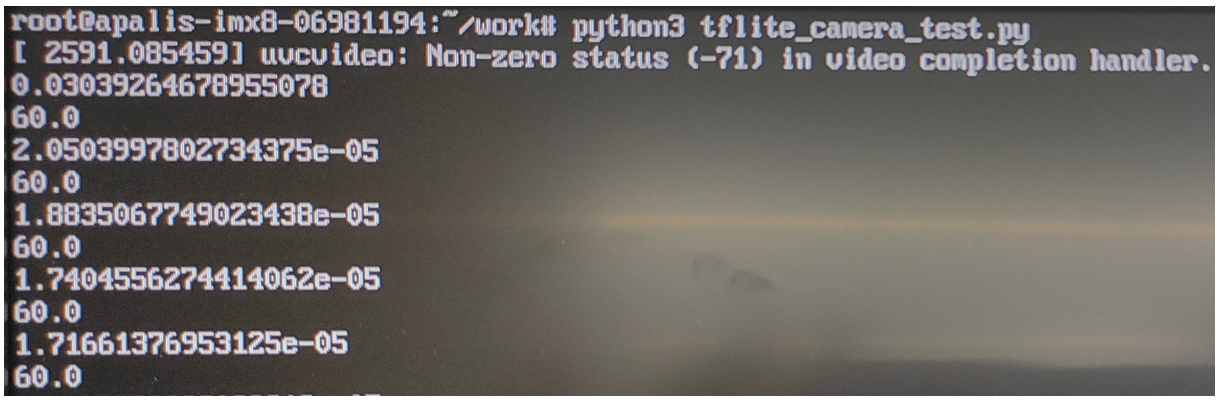

Figure 3.6: Output of python script using the Camera as input.

3.3 Output images

The input image is:

Figure 3.7: Input image to the model.

The output obtained using the EfficientDet0 model is:

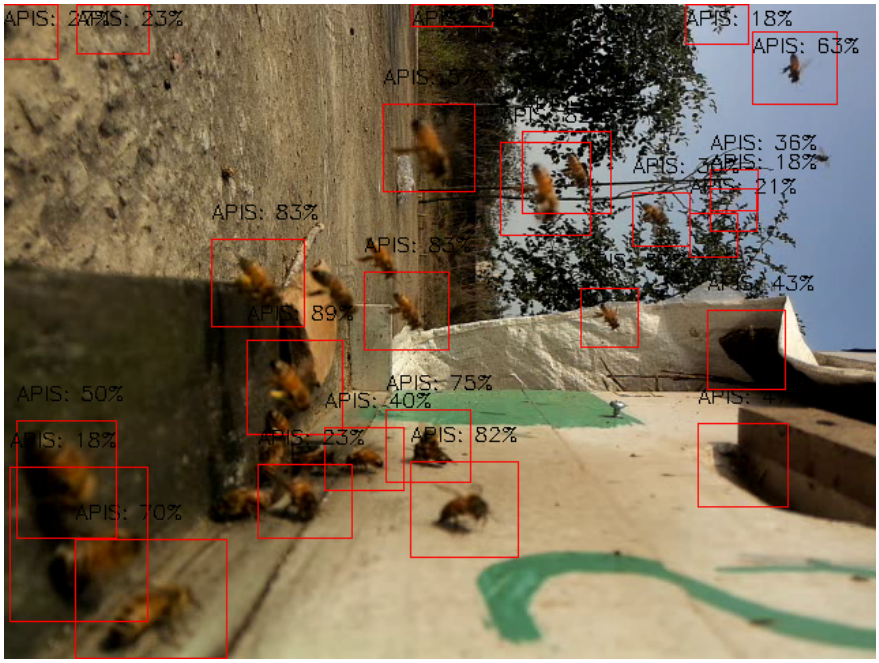

Figure 3.8: Model output image.

Chapter 4

AI on Cloud

We were also assigned work with the Machine Learning team. This team deploys the Neural Network on the Cloud and performs processing and inference there.

4.1 Creation of Apis Mellifera Dataset

Apis Mellifera is also called the Western Honey Bee, and is the most common honey bee in the world. The task given was to create a dataset of research-grade images, sorted year-by-year from. The data was obtained from the site Inaturalist, which is network of naturalists, scientists, and biologists. It is based on the idea of mapping and sharing observations of biodiversity across the globe.

In order to obtain high-quality images, a python program was used. It is a python script that uses the *csv* and *requests* modules in python to download images from a .csv file that contained URLs of an image dataset.

The script creates a directory where the images will be downloaded, then opens the .csv file and reads each row to get the image URL. It then extracts the filename from the URL and downloads the image using the requests module. Finally, it writes the image content to a file in the download directory.

This was not a very efficient or reliable method, especially when dealing with a large amount of images. So, we started using SeoTools, which is an Excel extension that uses a function *DownloadFile()*. The arguments of this function are filename, url and directory. This function then instantaneously saves the images directly without any other processing required. Through this method, more than 1 Lakh images were processed and saved as a dataset.

A well-curated dataset of images enables deep learning models to learn and generalize from a wide range of visual examples. By presenting the model with a diverse set of images, it is able to better identify and extract features relevant to the task at hand. Moreover, a large dataset ensures that the ML model has sufficient training examples to build robust and accurate models. It reducing over-fitting and provides the model with more variations to learn from.

4.2 Dataset annotation on Roboflow platform

The next task given was to annotate datasets on Roboflow. Roboflow is a data management and annotation platform targeted towards machine learning and computer vision applications. It has support for object identification, semantic segmentation and keypoint labelling among other annotation tools and workflows that enable users to label their data efficiently and reliably.
On the Roboflow platform, annotations can be made manually or partially automatically using machine learning models. The platform offers tools for checking and validating the annotations once they are finished to make sure they are accurate and consistent. It also offers choices for exporting the annotated data in a number of different formats for use in training machine learning models.
We annotated images of bees using this platform in order to enhance the quality of datasets used. Dataset quality and size is an essential factor in determining the accuracy of any machine learning model.

4.3 Understanding of the machine learning models being used

We were assigned to study the various machine learning models being used for image recognition.
TensorFlow Lite (TFLite) was employed, which is a lightweight version of the popular TensorFlow framework designed for mobile and embedded devices. It is intended to offer low-latency on-device machine learning inference with a tiny memory footprint, making it perfect for executing machine learning models on mobile devices, microcontrollers, and other resource-constrained systems. It is compatible with a variety of platforms, including Android, iOS, Linux, Windows, and others.
Also, the Cloud team was using the YoloV5 model, which is a state-of-the-art Neural Network

model trained on the COCO dataset, and is suitable for large-scale processing. It collects and labels a dataset of images from the Roboflow. Each image should have one or more objects labeled with bounding boxes.

4.4 Work on reducing false positives

We were given a task to study and try to reduce the false positives being detected by our model. False positives are predictions generated by a model that wrongly classifies an object as belonging to one class when it does not. In the instance of object detection, a false positive might result in inaccurate detection. This can lead to unintended actions or judgements being made based on incorrect information.

We studied some ways to reduce false positives -

Tighter bounding boxes:

The bounding box is a region in a picture that encloses the object of interest in object detection. By tightening the bounding box, the model becomes more selective in its predictions, considering just the areas that contain the object of interest. This can help to reduce false positives by excluding predictions that are not related to the object of interest.

Training on blanks: Another approach to reducing false positives is to train the model on images that do not contain the object of interest. By doing this, the model learns to distinguish between what is and what is not the object of interest. This approach is often used in combination with data augmentation techniques such as flipping and rotating the images to provide the model with a more comprehensive set of negative examples.

4.5 Front-end

We were told to study and observe the front-end work being done in the lab.

We can see some graphs developed with the aid of other interns:

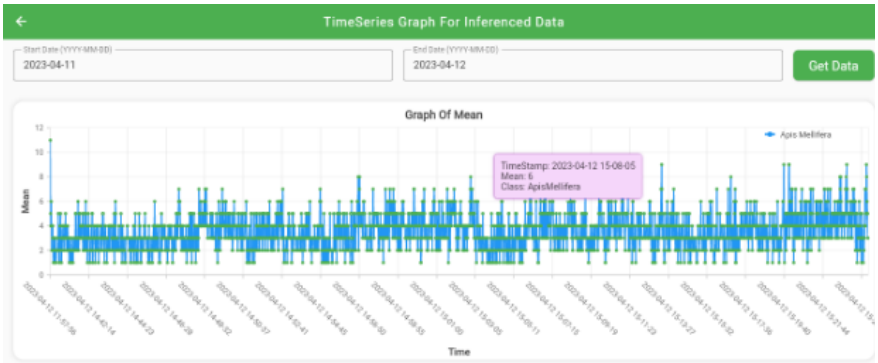

Figure 4.1: Movement of insects.

This graph displays the movement of insects over time. Another graph shows the mean Insect count -

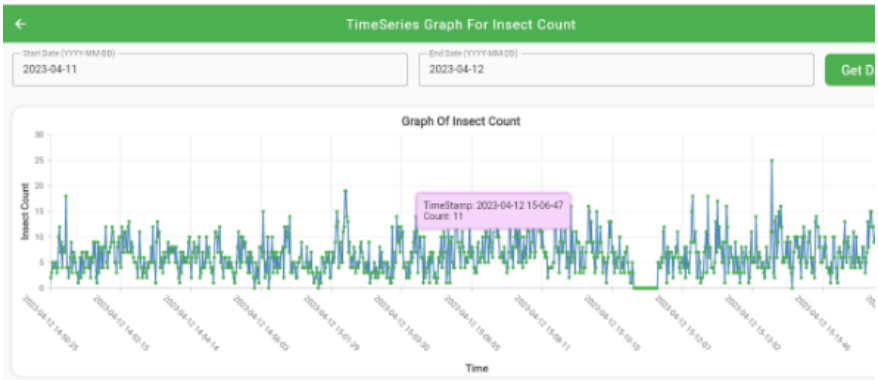

Figure 4.2: Insect Mean graph.

4.6 Image detection

We also performed image detection using the Cloud model -

Figure 4.3: Image detection-1.

Figure 4.4: Image detection - 2.

Chapter 5

Results

Our projects' findings show that our model was effective at identifying the location and number of bees frame-by-frame, through the camera connected to the device. After running the image through our model for inference, we returned bounding boxes, the bees species, and a confidence score expressed as a percentage.

These results show the efficacy of this method at recognizing bees through a lightweight model running on an SoM, and hold promise for future use in the conservation and population management of bees.

We also studied online techniques for bee classification and state that those methods are also beneficial for this task.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this project report we have demonstrated the creation of an object detection model based on TensorFlow Lite which can detect bees of the species Apis Mellifera. The model was trained with a set of bee photos collected from various sources from the internet. This model was then deployed on a Toradex Apalis i.MX8QM SoM running a custom Linux distribution, which was built using the Yocto Project. Our report demonstrates the efficacy of such a method in the monitoring of bee populations at different locations.
The usefulness of deep learning approaches for object detection, both on Cloud and on the Edge, has been demonstrated. The benefit of AI on Edge has been shown, with complex processing being done on hardware reducing the amount of compute required on the Cloud, along with using the full potential of the i.MX8 processor.

6.2 Future work

Future work can be done on various aspects -

1. Testing of higher complexity models to find a suitable spot between model accuracy and processing cost.

2. Further optimization of the TFLite processing workflow to reduce RAM and Memory usage.

3. Incorporation of various image pre-processing techniques to greatly increase model accuracy and reduce the number of false positives.

Chapter 7

References

1. TensorFlow Lite
2. TensorFlow Lite Python API
3. NXP eIQ
4. Yocto Project
5. Adding meta-ml layers
6. BSP Layers and Reference Images for Yocto Project Software
7. Build a reference image with Yocto
8. i.MX Machine Learning User's Guide
9. Embedded Linux for i.MX processors
10. Roboflow
11. YoloV5