To Send or to Not Send: A Case Study on Computer Vision for Low Power Edge Devices

Ajay Gopi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-90
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-90.html

May 29, 2020

Acknowledgement

I would like to thank Professor Prabal for giving me the opportunity to do research in his lab and guiding me through my entire research process. Through this process, I have learned many valuable skills and lessons that would have been hard to learn elsewhere. I would also like to thank Neal Jackson for mentoring me and providing help on every step of the way. Special thanks to Branden Ghena and Shishir Patil. I would like to thank Professor Kurt Keutzer for giving additional guidance and insight on the trends in this area. Last but not least, I would also like to thank my friends and family for motivating me through this process.

To Send or to Not Send: A Case Study on Computer Vision for Low Power Edge Devices

by

Ajay Gopi

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Prabal Dutta, Chair
Professor Kurt Keutzer

Spring 2020

To Send or to Not Send: A Case Study on Computer Vision for Low Power Edge Devices

by Ajay Gopi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Prabal Dutta
Research Advisor

5/28/2020

(Date)

* * * * * * *

Professor Kurt Keutzer
Second Reader

5/24/2020

(Date)

To Send or to Not Send: A Case Study on Computer Vision for Low Power Edge Devices

Abstract

To Send or to Not Send: A Case Study on Computer Vision for Low Power Edge Devices

by

Ajay Gopi

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Prabal Dutta, Chair

The Internet of Things (IoT) has been a growing area of recent times. With the large and ever-increasing number of IoT devices being deployed, there has been a rise in interest in incorporating machine learning on these devices. Low-power microcontrollers provide a low-cost compute platform to deploy intelligent IoT applications, but have extremely limited on-chip memory and compute capability. The use of machine learning for these applications highlights the tradeoff between local computation or sending the data to a more computationally powerful resource like the cloud. This paper explores this trade-off space through the computer vision tasks of people classification and people detection; people classification involves determining whether a human exists in an image while people detection involves providing bounding box information for all humans in an image. This paper uses existing models for these tasks and evaluates the tradeoff between running models locally and sending data to the cloud on the metrics of latency, energy, memory, and accuracy. The chosen models are run on the nRF52840 SoC, a low-power MCU system with protocol support for Thread and 802.15.4. Our findings confirm that local computation in low-energy constrained embedded systems makes sense for people classification in considering energy, memory, accuracy, and latency; however, these platforms are incompatible with more complex tasks like people detection due to fundamental memory limitations.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Professor Prabal for giving me the opportunity to do research in his lab and guiding me through my entire research process. Through this process, I have learned many valuable skills and lessons that would have been hard to learn elsewhere. I would also like to thank Neal Jackson for mentoring me and providing help on every step of the way. Special thanks to Branden Ghena and Shishir Patil. I would like to thank Professor Kurt Keutzer for giving additional guidance and insight on the trends in this area. Last but not least, I would also like to thank my friends and family for motivating me through this process.

Chapter 1

Introduction

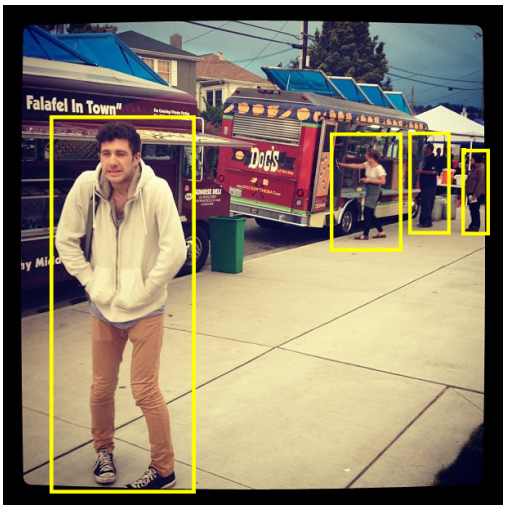
1.1 Overview

Machine learning on the edge is a growing field of interest. This growth has been fueled by the advent of the Internet of Things (IoT) and the ever growing supply of IoT devices. It is projected that there will be 41 billion such devices by 2027 [5]. Machine learning can be used to provide many valuable and intelligent applications on these devices. Some existing ML applications on IoT devices include traffic video analytics to improve traffic management in large cities [13, 1], voice trigger detection [28], and gesture recognition on canes [24]. Designers of machine learning applications for these devices have a choice between sending data to be processed elsewhere or locally. These devices introduce multiple constraints and as a result it is often not obvious whether to run locally or send the data to cloud. For example, limited memory makes sending all data to the cloud reasonable but limited energy makes running local algorithms to process some or all of the data necessary. In this paper we examine this tradeoff space on the tasks of people classification and people detection.

1.2 Task

Classification and detection are two prominent and interesting problems in the field of computer vision. Classification is the task of classifying an image into one of many categories. Detection is the task of finding and classifying a variable number of objects on an image. **For example, given an image, people classification involves determining if a person exists in the image while people detection involves providing bounding boxes for all people in the image.**

(a) Person classified



(b) Person detected with bounding boxes

Figure 1.1: People Classification vs. Detection
The left image represents people classification, which is just determining the image has a person. The right image represents people detection which involves generating bounding boxes for the people in the image.

1.3 Target Hardware

The target end-device for this paper is the Permacam, a variant of Permamote [15], an energy-harvesting sensor mote. The Permacam is built on the Nordic nRF52840 SoC which has the 32-bit 64 MHz ARM Cortex-M4 CPU with a floating point unit, 256 KB RAM, and 1 MB Flash. It has 802.15.4 support and utilizes OpenThread [23] to establish data backhaul. It also utilizes the low power Himax HM01B0 320x320 color sensor which can come in a monochrome version with QQVGA (160x120) internal downsampling. The motivation behind using this device is its low power and energy consumption, resulting in a long lifetime (greater than or equal to 10 years) with just a battery. While other devices such as CCTV cameras or Raspberry Pi's can offer similar functionality with less compute constraints, they require a larger, constant source of power and electrical supply, which makes it much more difficult to deploy these devices in the field.

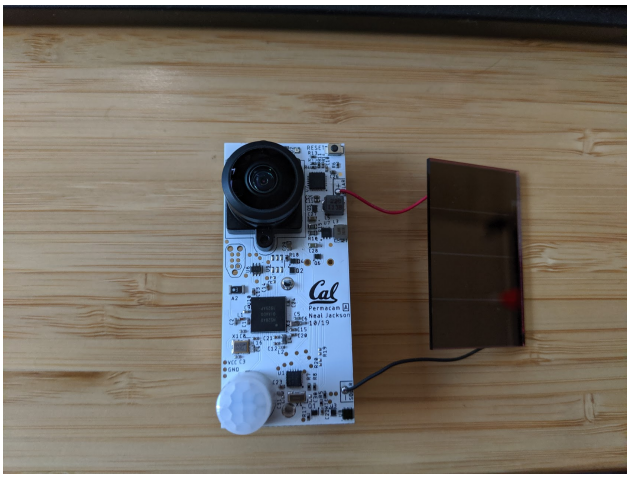


Figure 1.2: Permacam

Imagined Use Case

The imagined use case of people classification on this device is smart lights. People classification can be used to determine if a human exists in a room and can save energy from just using motion sensing approaches. The imagined use case of people detection is occupancy sensing and finer-grained lighting control.

1.4 Hypothesis

The central claim of this paper is that the feasibility of running machine learning models for computer vision locally is dependent on task. In other words, for the task of people classification, the benefits of local computation on severely resource constrained devices outweighs the benefits of sending images to the cloud in regards to energy, latency, accuracy, memory usage, and currently available technology; for the task of people detection, the benefits of sending images to the cloud outweigh the benefits of local computation in regards to the same factors, particularly memory.

Chapter 2

Background

2.1 Convolutional Neural Networks Overview

The following sections will give an overview of neural networks with an emphasis on convolutional neural networks.

Neural Networks Introduction

Neural networks are a set of algorithms that are used to approximate the human brain. Neural networks provide a technical framework for representing the concepts of learning and adjusting. The base component of a neural network is an artificial neuron. Artificial neurons take a number of input signals x_i and multiply them with weights w_i. The output is biased with w_b and the resulting output is passed through a non-linear activation function (f). A neuron's output can be represented as $y = f(\sum w_i * x_i + w_b)$. The weights w_i can be seen as a way of tuning the neuron's *reaction* to the input and their values can be modified to produce the desired output signal.

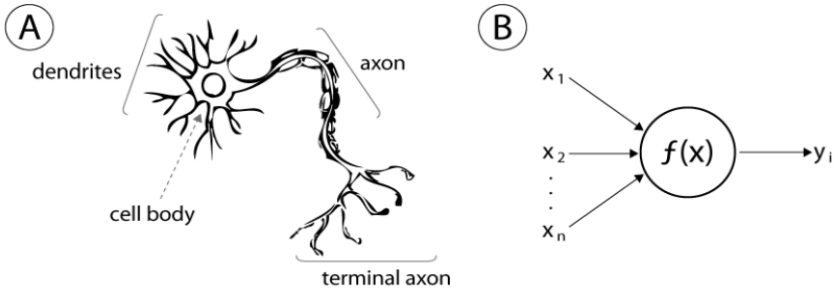


Figure 2.1: A biological neuron vs. neural network representation

Neural Network Structure

Neural networks consist of many interconnecting artificial neurons. Usually these neurons are in layers such that neurons only have connections with future layers and there are no cycles in the resulting structure. This structure is known as a *feed-forward network*. A *fully-connected network* is a feed-forward network such that each neuron in every layer has a connection to all neurons in the next layer.

Convolutional Neural Network Structure

Regular feed-forward networks can be used to solve multiple problems but certain issues arise in the image classification/detection field. '

1. Images are large in the number of input elements (pixels) they contain. For example, feeding a 196x196 grayscale image into a single layer fully-connected network results in **3,841,600** weights.

2. Fully-connected networks ignore the topology and structure of the input [18].

Convolutional neural networks address these issues with essentially sharing a set of parameters which are used throughout the image. Convolutional networks consist of multiple convolutional layers. The input to the convolutional layer is a volume with height h_{in}, width w_{in}, and depth d_{in} and the output is a volume with height h_{out}, width w_{out}, and depth d_{out}. The parameters of the layer are separated into filters, which each have a height k, width k, and a depth d_{in} (filters are generally square). Every filter is slid through the image depending on the stride and an elementwise product is computed with the input image and filter. d_{out} filters are employed to produce the output volume. Instead of having $(h_{in} * w_{in} * d_{in}) * (h_{out} * w_{out} * d_{out})$ parameters like fully-connected layers, convolutional layers have $k * k * d_{in} * d_{out}$ parameters. An illustration is shown in Figure 2.3.

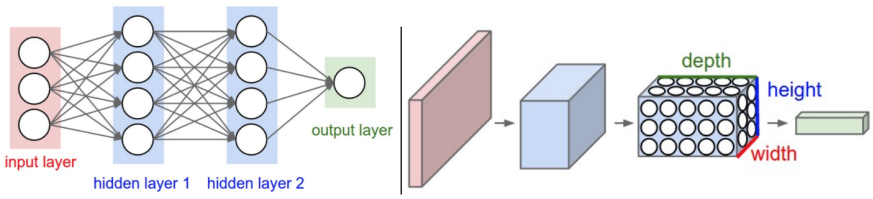


Figure 2.2: Fully-connected network vs convolutional neural network
The left shows a regular fully-connected network while the right shows a convolutional network. The convolutional network converts a 3D input volume into a 3D output volume. Visual is taken from [6]

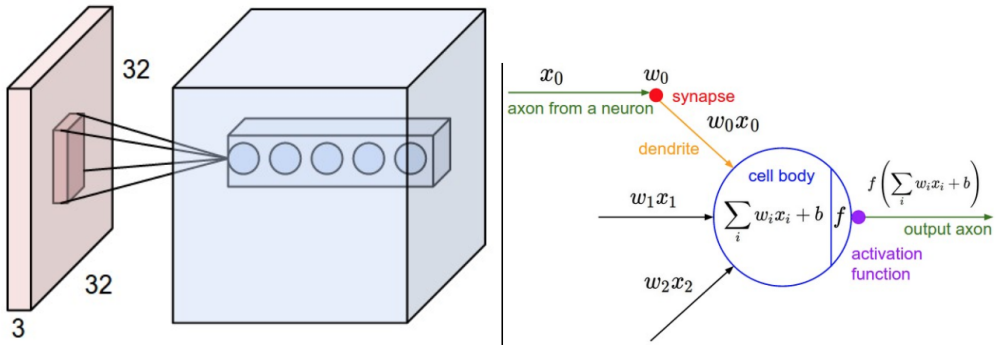


Figure 2.3: Convolution overview
Red represents input image. The filter slides through the input image to produce output (represented with blue)

Why does this work?

Convolutional neural networks (CNNs) are able to perform at a high level due to them using the locality of information in an image. Pixels in an image generally have a stronger relationship to nearby pixels rather than ones far away. The reuse of parameters through using the assumption of locality allows CNN filters to learn to detect distinct features as shown in Figure 2.4.

Depthwise Separable Convolutions

Convolutional networks tremendously reduce parameters from using fully-connected networks. Even with these reductions, it may be tough to deploy common architectures such as VGG-16 on an embedded system [29]. Depthwise separable convolutions address this issue [3]. Depthwise separable convolutions consist of one depthwise convolution and multiple pointwise convolutions. The depthwise convolution applies a single filter per input channel and the pointwise convolutions are used to create a linear combination of the output of the depthwise layer. While in a regular convolution d_{out} k by k by d_{in} filters are required, depthwise separable convolutions require one k by k by d_{in} filter and d_{out} 1 by 1 by d_{in} filters. While there are more filters in depthwise separable convolutions, by replacing k by k by d_{in} filters with 1 by 1 by d_{in} filters, depthwise separable convolutions significantly reduce the number of overall parameters.

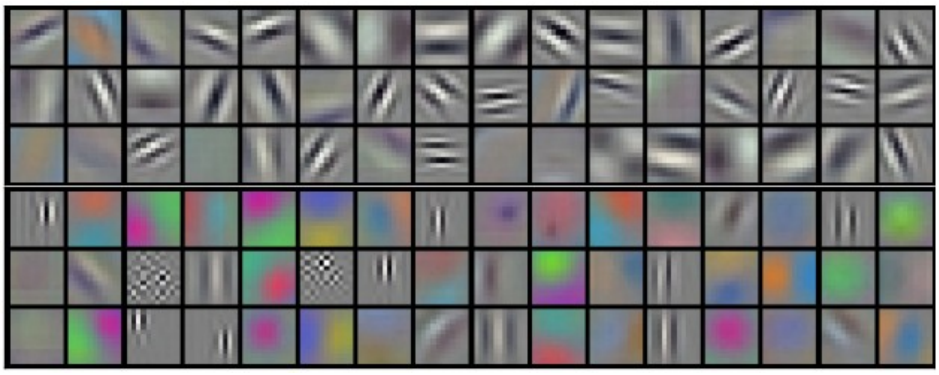
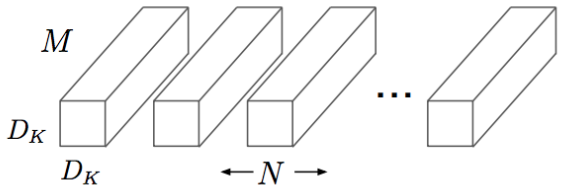
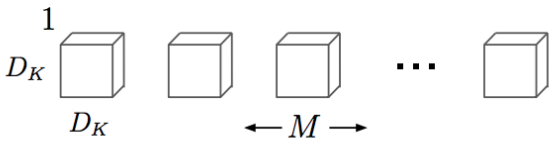


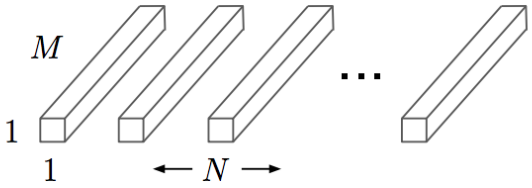
Figure 2.4: Filters learned in AlexNet
Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Some of these are simple edge detectors, while others are more complex.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2.5: Depthwise Separable Convolution
(a) Shows standard convolution (b) Depthwise convolution (c) Shows 1x1 pointwise convolutions The standard convolution is replaced with a depthwise convolution followed by a pointwise convolution in depthwise separable convolutions. Visual taken from [12].

2.2 Classification and Detection

This section will go over current models for image classification and object detection.

Top Performing Models for Classification

This section will briefly overview a set of models that are widely known in the area of image classification.

Classification Model Comparison				
Model Name	# of Conv Layers	# of params (millions)	MACS (millions)	ImageNet top-1 Accuracy
AlexNet	8	60	1140	62.5%
VGG16	16	138	15470	74.4%
GoogLeNet	22	7	1600	69.8%
ResNet	50	25.6	3870	77.15%
SqueezeNetv1.0	18	26	860	57.5%
MobileNetv1	8	4.2	569	70.6%
NASNetA	23	88.9	23800	82.7%

Table 2.1: Comparison of image classification models
This shows a comparison of image classification models for the image classification task on ImageNet, a dataset similar to COCO. MACS is the number of multiply accumulate operations in the forward pass. The top-1 accuracy tracks the percentage of correct labels assigned at first guess.

- **AlexNet**

  AlexNet was proposed by Alex Krishvesky et al. and is considered to be a breakthrough paper that drew attention to CNN architectures when it won the ImageNet challenge (ILSVRC) in 2012 [16]. It consists of a simple architecture of 5 conv layers and 3 fully connected layers. AlexNet uses ReLUs for the non-linearity layers and uses local response normalization. The network achieved a top-1 accuracy rate of 62.5 percent ILSVRC 2012.

- **VGG16**

  VGG16 was proposed by K. Simonyan and A. Zisserman from the University of Oxford and won part of the ILSVRC 2014 challenge. It replaces the large kernel filters in AlexNet with multiple 3x3 filters. It is widely used for its simple and regular architecture. VGG16 consists of 16 layers, which progressively expand the number of channels of the image from 3 to 4096 and reduce the height and width of the image from 224 by 224 to 7 by 7. VGG16 achieved a top-1 accuracy of 74.4 percent [29].

- **GoogLeNet**

  GoogLeNet was proposed by Christian Szegedy et al. and was published around the same time as VGG16. It won the ILSVRC 2014 and achieved a top-1 accuracy of 69.8 percent. Its top-5 accuracy was more impressive, with the model achieving a top-5 accuracy of 93.3 percent. The novel element in this architecture was a newly crafted module called the Inception module, which concatenates multiple convolutions on an input [30].

- **ResNet**

  ResNet was proposed by Kaiming He et al. and won the ILSVRC in 2015. It introduced the novel idea of skip connections which allowed the 152 layer model to train effectively. ResNet's skip connections solved the vanishing gradient problem that arose when training deep neural networks. The model achieved a top-1 accuracy of 77.15 percent [11].

- **SqueezeNetv1.0**

  SqueezeNetv1.0 was proposed by Forrest Iandola et al. at Berkeley in 2016. SqueezeNet, unlike the previous architectures discussed, did not focus on producing a state-of-the-art accuracy. It focused on optimizing latency and other constraints and the authors developed a network with an accuracy similar to AlexNet but with 50x less parameters. SqueezeNet introduced Fire modules which are comparable to GoogLeNet's Inception modules. It achieved a 57.5 percent top-1 accuracy and 80.3 percent top-5 accuracy. [14].

- **MobileNetv1**

  MobileNetv1 was proposed by Andrew G. Howard in 2017. It was built to address factors other than accuracy such as latency. MobileNetv1 consists of a lightweight architecture that uses depthwise convolutions. It used two parameters, α , the width multiplier, and ρ, the resolution multiplier, to reduce the number of channels per layer

and reduce input size respectively. It achieved an accuracy of 70.6 percent on ImageNet. [12]

- **NASNet**

  NASNet was proposed by Baret Zoph et al. at Google Brain in 2018. The central idea behind NASNet is to explore architectures through using Neural Architecture Search instead of manually creating architectures. NASNet is comprised of normal cells and reduction cells, which are searched through using a controller RNN. The weights of this RNN are updated through training. The NASNetA architecture resulted in a top-1 accuracy of 82.7 percent

Top Performing Models for Detection

This section will briefly overview a set of models that are widely known in the area of object detection.

Object Detection Model Comparison		
Model Name	mAP	Time (seconds)
R-CNN	62.4	49
Fast R-CNN	70.0	23
Faster R-CNN	78.8	.2
YOLO	63.7	.02
SSD	83.2	.02
YOLOv2	78.0	.025

Table 2.2: Comparison of object detection models
This shows a comparison of object detection models on the Pascal VOC dataset. mAP represents mean average precision.

- **RCNN**

  RCNN was invented by Ross Girshick et al. at UC Berkeley in 2014 [9]. The idea was to use a CNN as a feature extractor combined with an existing region proposal method known as selective search. The regions proposed by selective search were warped into a square image in order to conform to the feature extractor CNN's input size. The output of the CNN was fed into an SVM for determining the class of object and into a bounding box regressor to correct the proposed bounding box.

- **Fast RCNN**

  Fast RCNN was a continuation of RCNN by the same creator, Ross Girshick [8]. RCNN was slow due to the large amount of convolutions that had to be performed. The idea behind FastRCNN was to run the entire image through the convolutional neural network once and to map the regions proposed by selective search to the feature map produced by convolution. An ROI pooling layer was used to make the shape of these regions consistent and the output features were fed into a fully-connected network to determine the class. This reduced the test and training times by a factor of roughly 9 and 23 respectively.

- **Faster RCNN**

  Faster RCNN [8] by Ross Girshick addressed the issue from Fast RCNN where region proposal was taking longer than the convolutions used for classification. The idea behind Faster RCNN was to delegate the region proposal to a CNN, called the region proposal network. This ended up reducing detection time by an order of magnitude.

- **YOLO**

  YOLO (You only look once) was proposed by Joseph Redmon et al. in 2016. The main idea behind YOLO [26] was to treat the detection as a regression problem and not do processing for particular regions but to do it uniformly across the image. The input image was divided into a 7x7 grid and for each cell, C conditional class probabilities were predicted and for each bounding box type (B), 5 numbers were predicted: $x_{center}, y_{center}, H, W, and C$. This resulted in an output volume of shape $7*7*(5*B+C)$. Through this assumption, YOLO essentially reduced object detection networks in to a large convolutional neural network.

- **SSD**

  SSD [21] by Wei Liu, et al. is similar in its approach to YOLO in that it performed region proposal and detection in one shot (hence the name single shot detector). One key difference between SSD and YOLO is that SSD applied an object detection layer at different parts of the main convolutional networks layers. This resulted in SSD producing many more bounding box proposals than YOLO.

2.3 Embedded Systems Overview

Low power microcontrollers (MCUs) have to deal with many more constraints than regular hardware devices such as desktops, servers, or even phones. The main constraints that these MCUs have to deal with are energy, memory, computation, and communication. These constraints lead to interesting tradeoffs which need to be considered when designing a system. A prime example is that limited memory makes sending all the data to the cloud reasonable, but limited energy makes running local algorithms to process the data necessary.

Energy Constraints

Energy is a central constraint in embedded devices as these devices are often deployed in the edge. Devices that rely on batteries are limited by the energy storage of the battery. A typical coin cell battery allows an average power draw of $27\mu W$ for one year [7]. While certain devices are capable of scavenging energy from the environment, this amount is usually limited to microwatts. For example, in our target device , the maximum power that can be provided by the solar panel is $218\mu W$ at best. This is calculated as the typical irradiance in brightly lit indoor environments is $100\mu W/cm^2$ and solar cells are at best 20 percent efficient, so the $10.9cm^2$ solar panel used in the target device can provide at best $218\mu W$ [15, 10].

Limited Working Memory

Current microcontrollers have limited RAM (on the order of 100-300 kilobytes) and a megabytes or less of flash. Over the past decasdes, there has a been a significant increase in the memory capacity of these devices. For example, Table 2.3 shows the clock speed, RAM, and flash memory of two embedded platforms, the TelosB mote [25] and nRF52840DK [22]. Within a decade the RAM has increased from 10kB to 256kB. However, this memory is still extremely limited in comparison to the memory of traditional devices where machine learning is run, such as servers.

	TelosB	**nRF52840DK**
MCU	MSP430F1611	nRF52840
Sleep Current	$2.6\mu A$	$1.5\mu A$
Word Size	16-bit	32-bit
CPU Clock	8MHz	64MHz
Flash	48kB	1MB
RAM	10kB	256kB

Table 2.3: Comparison of embedded devices

Constrained Communication

Communication also presents a barrier for the majority of embedded devices. WiFi radios require hundreds of milliwatts or more and even low-power technologies such as Bluetooth Low Energy or 802.15.4 require around ten milliwatts. Due to these energy constraints in communication, embedded devices duty-cycle their radios, powering them between 0.1 and 1% of the time in many deployment scenarios. This limits communication to the range of kilobits and thus makes local processing more desirable in terms of latency and energy.

Computation Capabilities

In recent times, the computation capabilites of microcontrollers have reached hundreds of megahertz [27]. Dedicated hardware have been developed for exploring the viability of general purpose hardware that can efficiently accelerate operations for constrained devices, with systems such as a 14.6 A 0.62 mW ultra-low-power convolutional-neural-network face-recognition processor [2]. Even without special accelerators, it is now in the realm of possibility and beneficial to run machine learning models locally.

Even without specialized hardware, the low sampling rate of sensors in embedded systems, in order to meet energy constraints, make continuously running learning techniques possible.

Chapter 3

Approach

3.1 Model Training

The following sections will describe the steps taken to prepare models for people classification and people detection. The code used for producing these models can be found at `https://github.com/ajax98/PersonMLModels`.

People Classification

We first approach the easier task of people classification, and then extend our approach to people detection.

COCO Dataset Overview

Just like any other neural networks, large amounts of data are required to train convolutional neural networks. This data comes in the form of datasets. The COCO dataset is a widely used dataset to benchmark object detection and segmentation models. It is comprised of natural images of complex scenes that contain multiple objects. It has 91 object types with more than a million labeled instances in 328K images. COCO has fewer categories than the popular ImageNet dataset but has far more pictures per category. COCO addresses the issue of previous datasets by providing non-iconic views (e.g. objects can be occluded), multiple objects per image, and precise 2D localization of objects (bounding boxes) [20].

Dataset

The publicly available Visual Wake Words dataset was used [4]. The labels for the Visual Wake Words dataset were produced by modifying the COCO dataset for classifying people. Labels were created by sampling through the dataset and assigning each image a label 1 if a person existed in that image or a 0 if that was not the case. This was done by determining if an image had a bounding box corresponding to a human. In order to make training efficient

and have less outliers, only bounding boxes that occupied more than .5 % of the image area were considered. Approximately half of the images were labelled as human in the training and validation sets.

Model Training

The model architecture was based off MobileNetv1 [12] due to its lightweight architecture and parameters of α and ρ that allowed for tuning the size of the model. Vanilla MobileNetv1 gave a 90 % accuracy on the dataset. Through applying data augmentation tricks such as image flipping and random cropping, the accuracy increased to roughly 94 percent. Our model size was reduced by reducing the depth of each convolutional layer by 75 percent ($\alpha = .25$). This resulted in the architecture shown in Figure 3.1. This architecture reached an accuracy of roughly 85 percent. While the model parameter size was within the flash memory limits of our target system, it exceeded the limit when combined with our input image. Thus, in order to account for this, the input resolution was reduced from 224x224x3 to 96x96x1. This model gave an accuracy of roughly 78 percent. After applying post-training quantization, the model reached a size of roughly .23 MB with an accuracy of 76 percent. These models are summarized in Table 3.1.

Model Name	Input Size	Model Size (MB)	Accuracy
Vanilla MobileNet v1	224x224x3	16.4	.94
MobileNet v1 .25	224x224x3	.903452	.85
MobileNet v1 .25	96x96x1	.903452	.78
Quantized MobileNet v1	96x96x1	.237267	.76

Table 3.1: Classification architectures

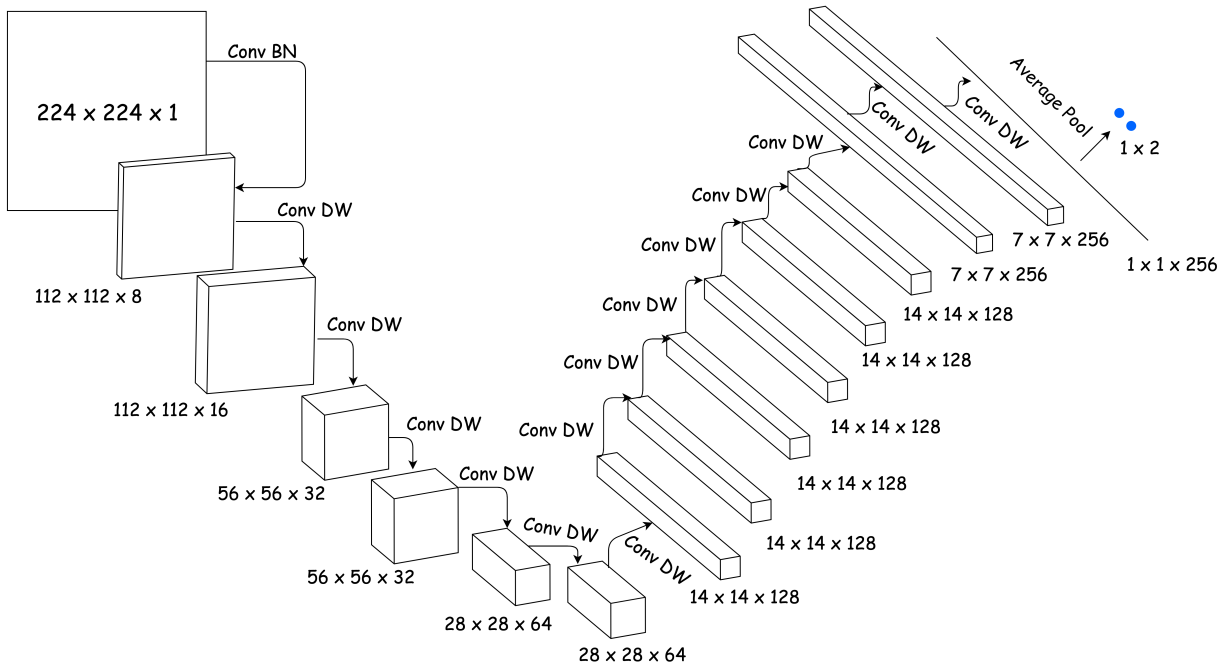


Figure 3.1: MobileNet Architecture
This shows the MobileNet Architecture $\alpha = .25$ applied to grayscale images. Conv BN represents a convolution followed with a batch normalization while Conv DW represents a depthwise separable convolution.

People Detection

As we were able to produce models for people classification with reasonable accuracy, we continued to develop models for people detection.

Dataset

Similar to the Visual Wake Words datset used for people classification, the dataset used for people detection was sampled from COCO. Additional information, such as bounding box coordinates of people in a frame were included in the annotations. These coordinates were normalized in order to conform to YOLO's abstraction that every grid cell is at the center and 1x1. See code in Appendix Listing 6.1 for details on how these annotations were produced.

Model Training

The model was based off of YOLOv3 tiny (Figure 3.2). By applying techniques such as inserting images of different sizes to increase robustness of grid prediction, the model achieved a mean average precision of 70.3 on our validation dataset. In order to compress the model architecture to fit in our target hardware, multiple convolutional layers in the architecture were replaced with depthwise separable convolutions. This itself reduced the size of our model by roughly an order of magnitude. The number of filters for each layer and number of layers were also reduced, allowing us to reach a model size of 1.5 MB. After training our model, we achieved a mean average precision of 36.9 on our validation set.

Model Name	Input Size	Model Size (MB)	mAP
YOLOv3 Tiny	416x416x1	33	70.3
YOLOv3 Ultra Tiny Quantized	416x416x1	1.5	36.9

Table 3.2: Detection architectures

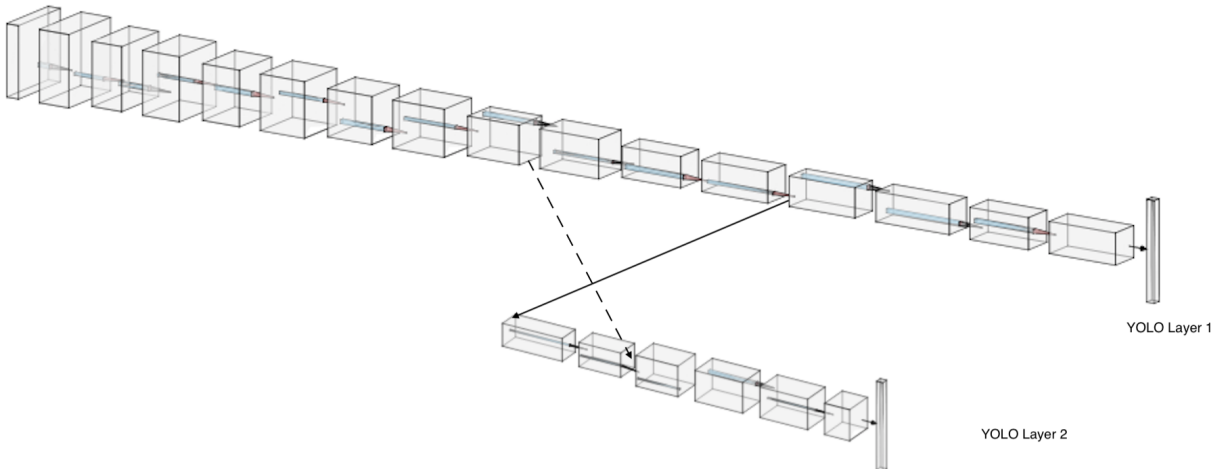


Figure 3.2: YOLO Architecture
This shows the rough architecture of YOLOv3 tiny. The dotted line represents a residual connection where the features are being concatenated while the solid line representing passing the output of a convolutional layer into a new convolutional layer. YOLO Layers are where the output is interpreted as grid cells and the loss is calculated and propagated backwards.

Figure 3.3: YOLO Ultra Tiny Output

3.2 Porting Model to Edge Device

After training models for people classification and people detection, the next step was to port these models to our target device. In order to port the model to the target device,

we used TensorFlow Lite. The code for the ported application can be found at `https://github.com/lab11/nrf52x-base/tree/tflite`.

Overview of TensorFlow Lite

TensorFlow Lite is a set of tools that facilitates running machine learning models on mobile, embedded, and IoT devices. It has features that allow for on-device machine learning inference with low latency and a small binary size. The main components of TensorFlow Lite are its interpreter and converter. The interpreter is essentially a library that takes a model file as input, executes the operations specified by the model, and provides access to the output. This interpreter supports a limited set of operations which can be expanded through defining custom operations. The converter is used to convert TensorFlow Models into a FlatBuffer, an efficient storage format which can be later run using the interpreter. See Figure 3.4 for details on the conversion of a model to the FlatBuffer format.

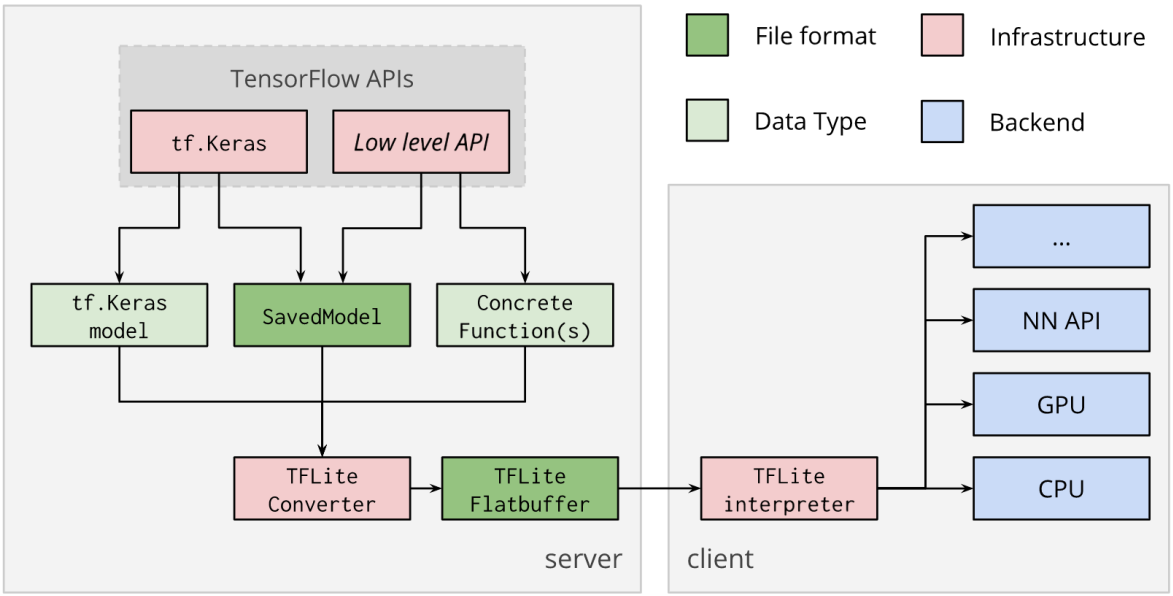


Figure 3.4: TensorFlow Lite Workflow
This diagram represents the steps on converting a TensorFlow model to run on TensorFlow Lite. The model can be trained using the tf.Keras or low level (computation graph) APIs . The trained model can be converted from its saved format to a TFLite Flatbuffer by the TFLite Converter. The server here represents any device used for training and the client represents the nRF52840.

FlatBuffer Model Representation

TensorFlow models are converted into a FlatBuffer representation before being run on the target device. FlatBuffers is a cross platform serialization library that can be used across many languages. What sets FlatBuffers apart is that it represents hierarchical data in a binary buffer such that it can accessed without parsing or unpacking. The first step in using FlatBuffers is to design a schema, which specifies the various objects that are being serialized and the relationship between these objects. The next step is to use the `flatc` compiler, which is used to generate a C++ header with helper classes to access and construct serialized data. Finally the `FlatBufferBuilder` is used construct a FlatBuffer binary. In the case of the TensorFlow Lite, the schema describes a model objects' relationship with various other objects. These objects and their components are outlined in Figure 3.5.

Compiling and Linking TensorFlow Lite Runtime

While TensorFlow Lite had compilation workflows for devices such as the Arduino Nano 33 BLE Sense, Sparkfun Edge, etc., none existed for our target device. This was designed through meeting a set of requirements. The requirements for porting a device are C++11 compatability, debug logging (printing strings to the debug console), math libray (libm.a), and global variable initialization. We used the **arm-eabi-gcc** compiler which satisfied the requirement of C++11 compatibility. Debugging logging was implemented based off the **fprintf** function. We included the math library while linking, satisfying that requirement. Finally, global variable initialization was satisfied by the compiler. For debugging, the workflow included **arm-eabi-gdb** and JLink. From this step, many integration issues arose which had to be addressed. Each of these issues were addressed through understanding various parts of the TensorFlow Lite internals and the approach to resolving them are detailed in the following section.

Resolving Issues

The section details various TensorFlow Lite issues resolved.

RAM region overflow

This error was due to too much space being allocated to the heap. As TensorFlow Lite does not use dynamic allocation, this was resolved by setting the `__HEAP_SIZE=0x2000`.

Failed to get registration for opcode d

In TensorFlow Lite, every operation/registration has a corresponding opcode. In order make sure the operations are registered and can be used, an `op_resolver` is used. In order to make sure that the operations defined in the graph were acceptable, we added `DEPTHWISE_CONV2D`, `CONV_2D`, and `AVERAGE_POOL2D` to the list of operations.

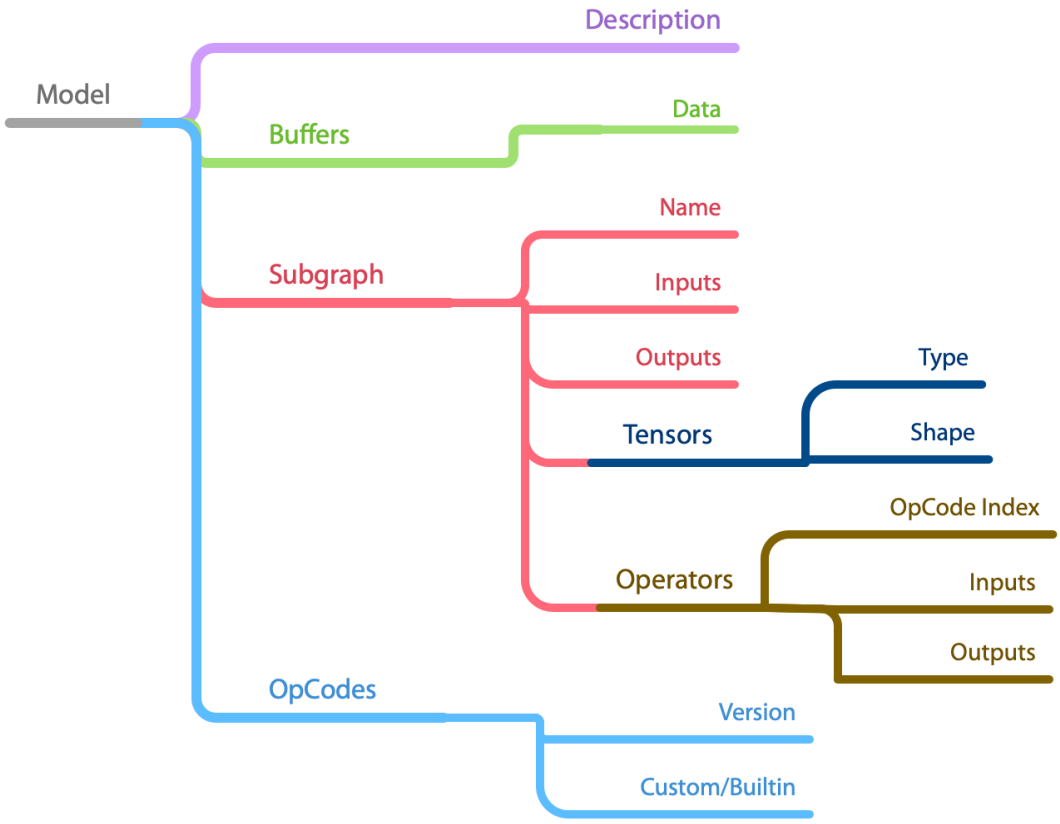


Figure 3.5: Flatbuffer Model Representation

The model was represented as the following in TensorFlow Lite. A model is composed of a subgraph which represents the computation graph and operators which represent operations such as convolution. The subgraph has multiple tensors, which are essentially multidimensional arrays which represent inputs and outputs to various nodes of the graph. The operators represent particular instantiations of operators which contain information on the input and output to the output. The inputs and outputs in a subgraph are numbers that index the list of tensors for tensors that are the overall input and output to the graph.

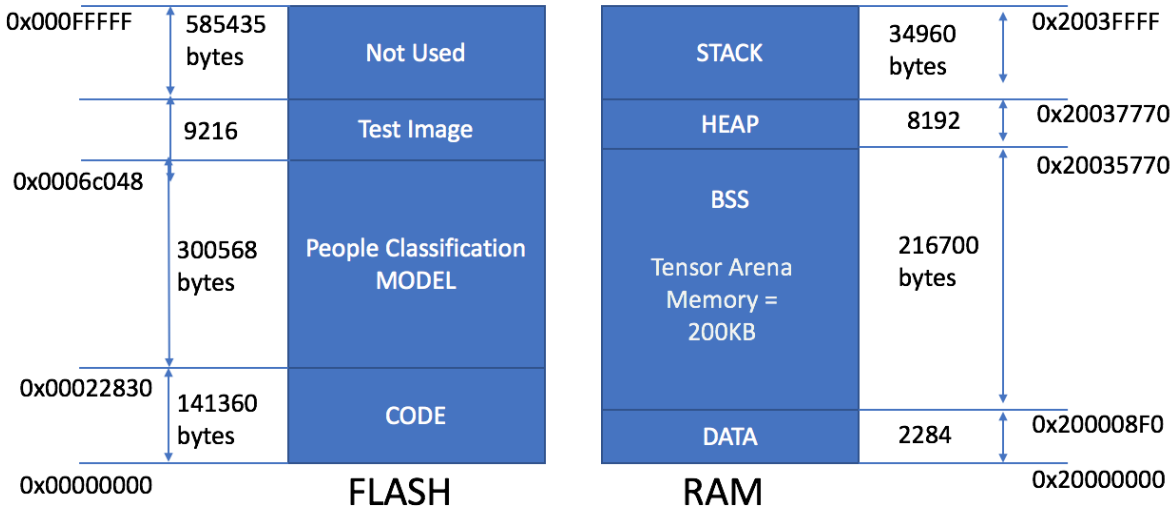


Figure 3.6: Memory Map

This represents the memory usage of running the people classification model on the target device. The flash contains `const` objects such as the people classification model and test image as well as the code segment. The RAM consists of the stack, heap, bss, and data segments. The bss mainly consists of the tensor arena, an area used by TensorFlow Lite for calculating the forward pass of the model. As the diagram shows, RAM is generally more of a constraint than Flash.

Tensor Arena not sufficient

This error was complex in that it had many possible causes and was solved by understanding the internals of TensorFlow Lite. Initially it was assumed that the target device did not have enough memory available for running the model. In order to debug this, we looked through the elf file created using `nm` and constructed the memory map shown in Figure 3.6. This showed us that this was not the issue. The next step that was taken was to quantize the model using the TensorFlow Lite Converter, and this was performed by adding `converter.optimizations = [tf.lite.Optimize.DEFAULT]`. As this did not solve the issue, we then looked through the tensors of the graph and realized that some of tensors were still not quantized. This error was solved by setting `converter.target_spec.supported_ops` to `[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]` and setting the `converter.inference_input_type` and `converter.inference_output_type` to `tf.uint8` in the converter, which set the input and output to `uint8` and limited the set of operations to `int8`, thus quantizing all tensors. This solved this issue and output was produced in terms of a person and no person score.

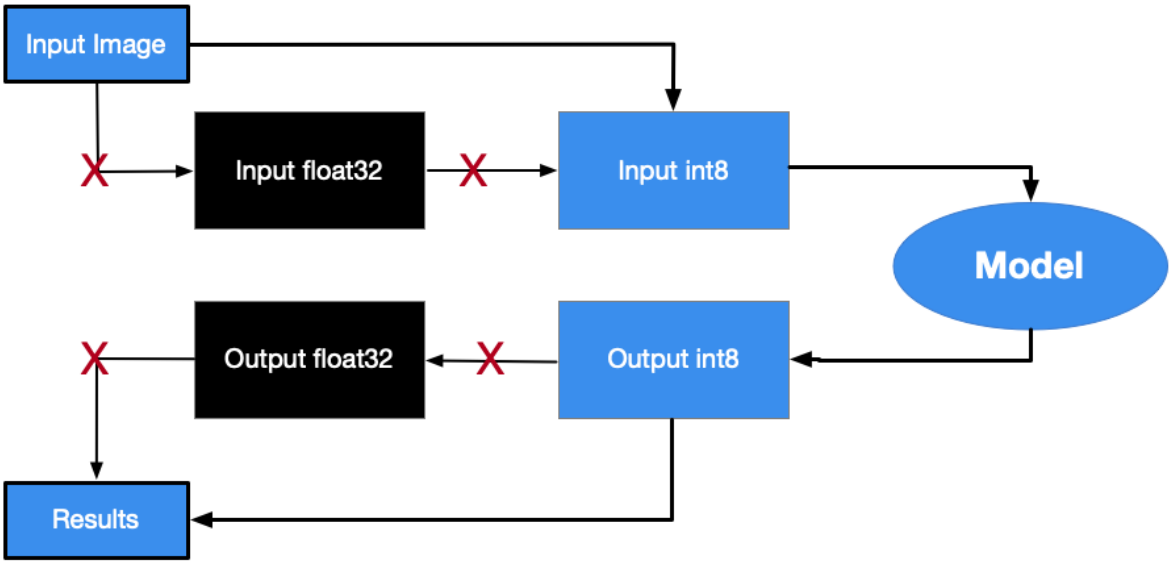


Figure 3.7: Flatbuffer Converter vs. Interpreter Mismatch

Flatbuffer Converter vs. Interpreter Mismatch

While the issue seemed to be solved, there was a hidden issue. By making the converter input and output type `uint8`, the TensorFlow Lite Converter added extra nodes rather than converting the existing input nodes and output nodes type from `float` to `tf.uint8` as shown in Figure 3.7. This resulted in unnecessary memory usage. In order to resolve this we initially tried to modify the FlatBuffer model structure. However, this structure was static and as a result, we modified the processing of the model in TensorFlow Lite so that the additional input and output nodes were ignored. This resolved this issue.

Chapter 4

Results

The following sections will overview the testing of the approach, the measurement of the energy, latency, accuracy, and memory across different models, and a comparison between running people classification model locally and sending data to the cloud.

4.1 Experiment Setup

In order to explore the tradeoff between accuracy, energy, latency and memory, we trained our model on 4 different input sizes, 48x48, 72x72, 96x96, and 120x120. These sizes were chosen so the model could fit on the target device while maintaining a reasonable factor of difference in the image resolutions. The energy, latency, and memory usage of these models were measured. These energy and latency results were compared with the energy and time required to send data to a cloud AWS instance. It is assumed that the cloud instance does not have memory constraints and accuracy-wise performs as well as the best possible model for people classification.

4.2 Testing

In order to test that TensorFlow Lite on the target device worked, we compared results of 5 different images across 3 different settings: TensorFlow Python, TensorFlow Lite on Linux, and TensorFlow Lite on target device (`nrf52840`). As the output of TensorFlow Python (the reference) was in floats, a dequantization node was implemented in the TensorFlow Lite on the target device implementation that simply divided by the scale and added the zero point to the quantized output. Images a and b were taken from the COCO validation dataset and images c to e were taken from the target device, the Permacam. The results on these images across these three settings are shown below.


(a)


(b)


(c)


(d)


(e)

Image	Python 96	Linux 96	Device 96
a	.8	.31	.31
b	3.26	3.31	3.31
c	-0.84	-0.71	-0.71
d	-1.73	-1.65	-1.65
e	0.43	0.21	0.21

Table 4.1: Testing of Model

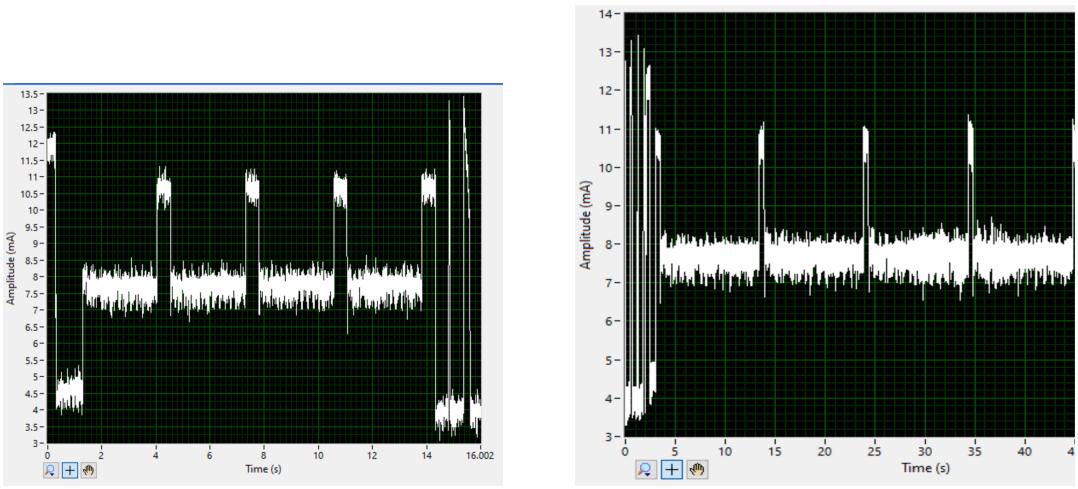
The images *a* to *e* were run through the model on TensorFlow Python locally, TensorFlow Lite locally (Linux), and TensorFlow Lite for the target device. The output represents a person score; if the score is greater than 1 (represented with color green), there is a person otherwise there is not (represented with color red). The scores differ between TensorFlow Python and the other two due to quantization and rounding errors that come with representing floats with ints. Images *c* to *e* are taken from the target device, the Permacam.

4.3 Measurement

The 4 aspects measured across the models are energy, latency, memory and accuracy.

Energy

The energy was measured through using the ADS1115, a data acquisition device, along with an Arduino. The measurement was repeated across different versions of the model and all instances gave a current of roughly 7.5 milli Amperes and power of 0.034 W. See Appendix for details on power measurement.



(a) 48x48 Current Usage **(b)** 96x96 Current Usage

Figure 4.1: Energy Measurement Setup
(a) and (b) show the current measured for the 48x48 model and 96x96 model respectively. The bumps in current usage represent the completion of a detection and are directly due to toggling an LED. The average current usage is 7.5 milliAmperes across models.

Latency

The latency for running each model was measured through using the real-time counter (RTC). The RTC runs on the low frequency clock and the frequency of counting is determined by $f_{RTC} = 32.768/(PRESCALER + 1)$. By setting the `PRESCALER` to 327, the count was incremented every 10 milisecond.

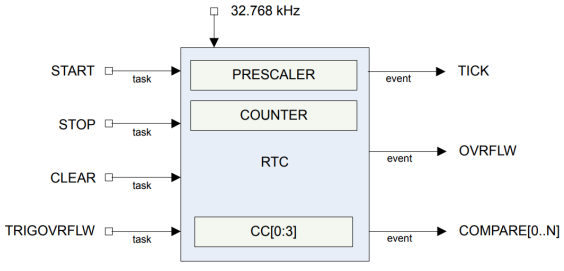


Figure 4.2: Real Time Clock Diagram

Peak Memory Usage

The peak memory usage of the target device was measured by looking through the tensors of the model. TensorFlow Lite processes models through first creating structures that reference the Tensors (inputs/outputs of Operators) and structures that keep track of memory; in other words, it uses memory for bookkeeping. While processing an the model, TensorFlow Lite allocates memory for the input and output the current operator it is processing, the current node usage. Therefore the peak memory usage is nothing but the sum of the current input, current output, and bookkeeping memory.

Accuracy

The accuracy of the model was measured through running each model through the COCO validation dataset.

Cost of Sending Data to Cloud

Other members of our research lab determined the cost of sending data to cloud. The data was sent using OpenThread with CoAP (constrained application protocol). The target device was able to achieve a goodput of **20 kbps** using a block size of 512 bytes at a 0 dBm radio power mode. At the 0 dBm power mode, the radio consumes 4.8 mA and allows for 10-30 meters range of communication. Increasing the transmit power to +4dBm increases the range but also increases the current consumption to 9.6 mA.

4.4 Results

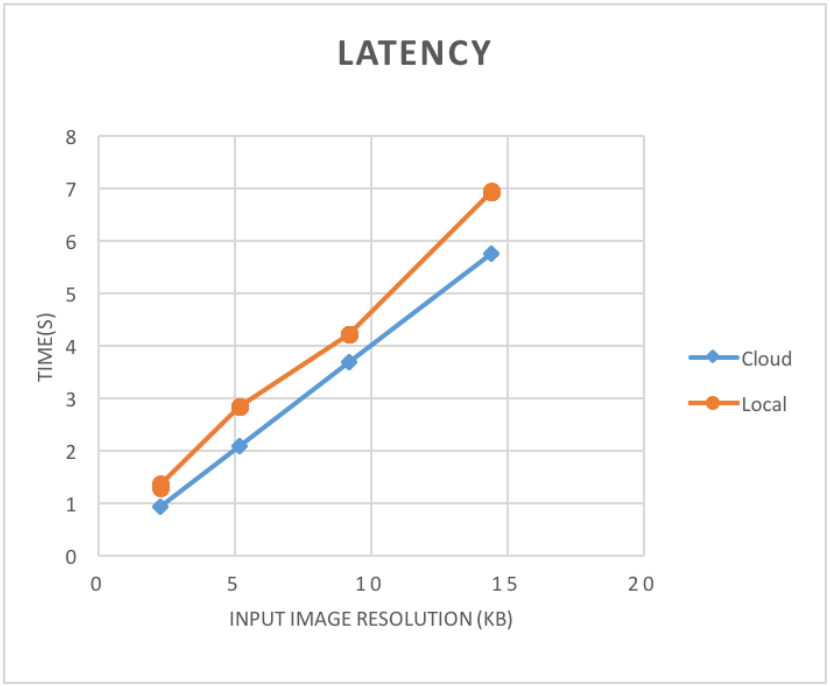


Figure 4.3: Input Image Resolution vs Latency

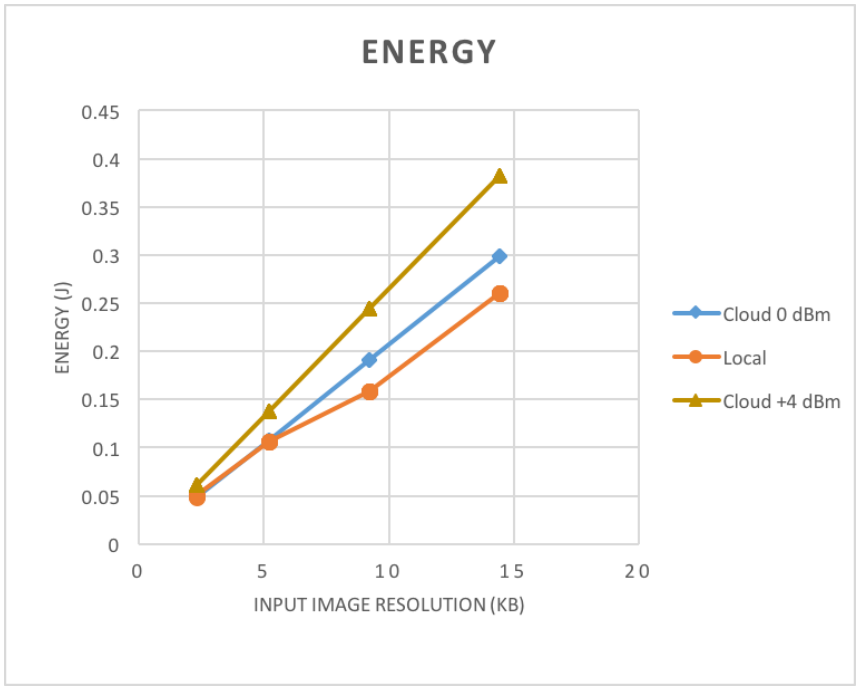


Figure 4.4: Input Image Resolution vs Energy

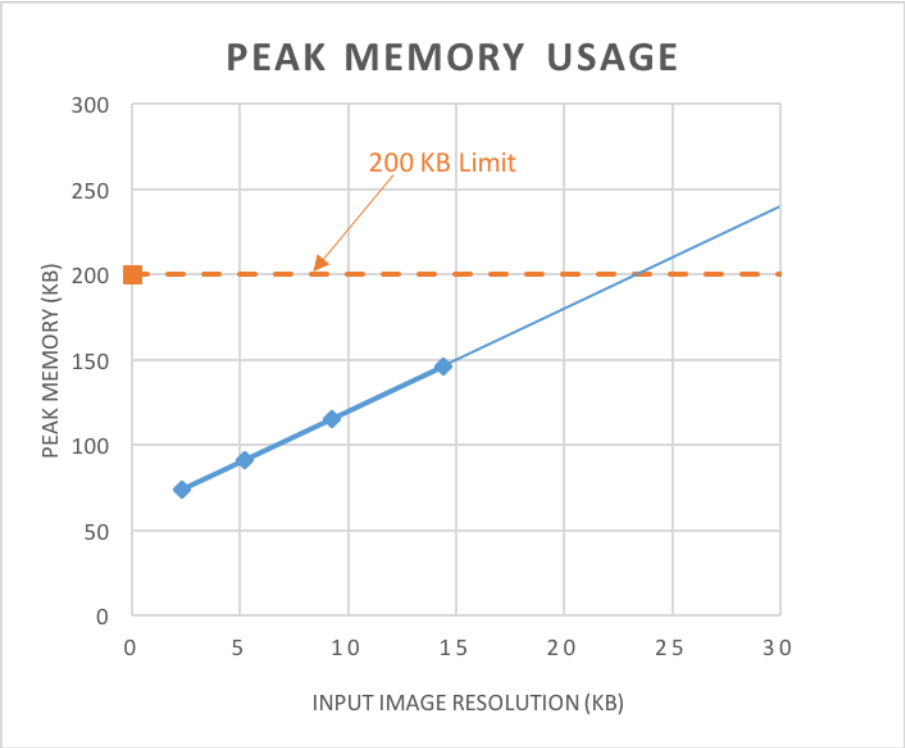


Figure 4.5: Input Image Resolution vs Memory

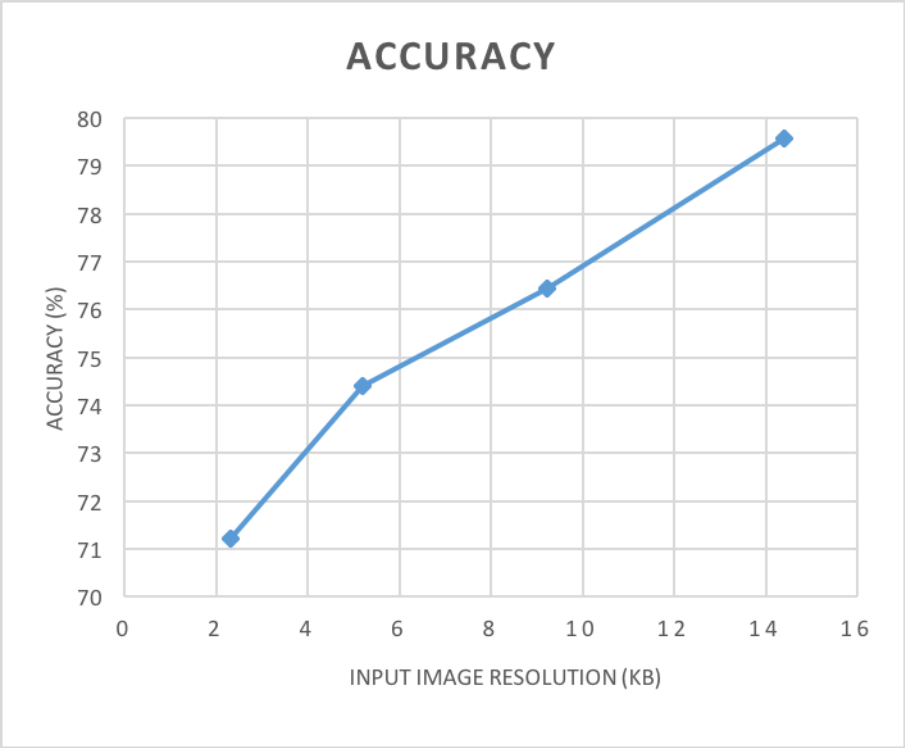


Figure 4.6: Input Image Resolution vs Accuracy

4.5 Analysis for People Classification

The following graphs (4.3,4.4,4.5,4.6) show that the decision between cloud and local compute is complex. Latency-wise and energy-wise it is a close call between cloud and local processing. Sending data to the cloud takes less time than local processing, a difference that only increases with input size. However, sending data to the cloud requires more energy due to the extra power needed for the wireless radio. Factoring in a few more variables makes the decision for local processing more clear. First, the TensorFlow Lite convolutions run are not optimized and can be sped up through using efficient neural network kernels through ARM-CMSIS NN [17]. Secondly, the model architecture itself can be optimized through using Neural Architecture Search that factors in energy and latency [31]. Furthermore, the energy and latency cost for sending data to the cloud shown in Figure 4.4 and Figure 4.3 is the best case scenario. Any packet loss can increase the time and therefore energy required to transmit data to cloud. Also, in scenarios where the communication distance is higher (greater than 30 meters), more devices need to be added to the thread network or the transmit power needs to be increased. Increasing the transmit power however increases the current and energy consumption as shown in the graphs above. Considering these factors and the factors of energy and latency, it does make sense to people classification locally.

Local processing is however not without sacrifice, as running locally does not provide the same accuracy as running in the cloud. The best trained model for local processing had an 80 % accuracy on the validation dataset. While the accuracy of this model is far from state-of-the-art accuracy, it can be improved through finding better model architectures or having multiple devices running this model simultaneously and aggregating their results (essentially ensemble models). Increasing input size could be a solution to improve accuracy as shown in Figure 4.6; however, this is constrained by the available RAM.

In running models locally, memory, particularly RAM, is the greatest constraint. While getting model parameters to compress is possible through quantization, during inference each operation needs sufficient space to store its input and output. This is not trivial, as while convolutional neural networks generally reduce the height and width of the input image through each hidden layer, the depth of the output of these hidden layers increases. These increases in depth can easily push the input/output to operators beyond the limits of the available RAM. For the task of people classification,this turns out to not be an issue and models can meet RAM constraints.

Therefore, by analyzing latency, energy, memory, and accuracy, we can see that running models locally is more advantageous than sending data to the cloud for people classification with the condition that the input image resolution must be constrained. In the case of the Permacam, the input image resolution constraint can be met through using the monochrome version of the HiMax sensor which provides QQVGA (160x120) monochrome output.

4.6 Analysis for People Detection

Experiments were unable to be run for people detection due to memory constraints. The YOLOv3 Tiny model parameters size was 22 MB (well beyond the memory of the target device). While modifications made through replacing convolutions with depth separable convolutions and quantization allowed the model parameters to fit in the flash, inference was not possible due to the lack of RAM. This once again points to RAM being the greatest constraint in running models locally. **Therefore, through analyzing the memory constraints we can see sending data to the cloud for people detection is more reasonable than running locally at this current point of time.** Future technology innovations can change how much can be accomplished locally versus in the cloud.

Chapter 5

Conclusion and Future Work

In this thesis, we explore the viability of running computer vision machine learning models on constrained embedded devices. Our experiments confirm our initial hypothesis that the feasibility is dependent on task. In other words, local computation in low-energy constrained embedded systems is viable for people classification while considering energy, memory, and latency, but does not make sense for "tougher" problems such as people detection due to memory limitations.

Future work would be to explore compression methods and novel machine learning architectures for the tasks of people classification and people detection. Compression methods such as pruning [19] and DeepIoT [32] could be effective ways of reducing architecture size and compute. Another approach would be to look into neural architecture search which takes into consideration the constraints of latency, energy and memory in order to produce novel architectures for both people classification and people detection [31].

Chapter 6

Appendix

6.1 YOLO Dataset Code

```
import numpy as np
from pycocotools.coco import COCO
import json
from PIL import Image



#Category of person is 1
foreground_img_ids = coco.getImgIds(catIds=1)
background_img_ids = coco.getImgIds(catIds=range(2,81))

#Iterate through images that are labelled as Person
for img_id in foreground_img_ids:
        img = coco.imgs[img_id]
        height = img['height']
        width = img['width']
        name = img['file_name'].replace('jpg', 'txt')
        img_area = height * width

        f = open(output_path+name,"w+")

        #Iterate through annotations in the image
        for ann_id in coco.getAnnIds( imgIds=img_id,  catIds=
   foreground_class_id ):


            ann = coco.anns[ann_id]
            ann_area = ann['area']
            norm_area = ann_area/img_area

            #If area of object is greater than .005 consider it and modify
    the bounds (for YOLO processing)
```

```
31
32              if norm_area > .005:
33
34                  bbox = ann['bbox']
35                  from_left = bbox[0]
36                  from_top = bbox[1]
37                  ann_width = bbox[2]
38                  ann_height = bbox[3]
39                  x_center = from_left + ann_width/2
40                  y_center = from_top + ann_height/2
41                  norm_x_cent = x_center/width
42                  norm_ann_width = ann_width/width
43                  norm_y_cent = y_center/height
44                  norm_ann_height = ann_height/height
45
46                  f.write("0 %f %f %f %f\n" % (norm_x_cent , norm_y_cent ,
    norm_ann_width , norm_ann_height))
47
48
49          f.close()
```

Listing 6.1: YOLO

6.2 Energy Measurement

The energy used locally through the board was measured through using the ADS1115, a data acquisition device, along with an Arduino. The nRF52840 board was connected to a power supply through a 5 volt connection to V_{IN}. The ground of the board was connected to a 50 Ohm resistor and the voltage across this resistor was measured using the ADS1115. The voltage across the 50 Ohm resistor was converted into current by dividing by the resistance and power was found by multiplying the voltage and current. The energy used was determined by integrating power usage across time. This measurement was repeated across different versions of the model and all instances gave a current of roughly 7.5 milli Amperes.

An image of the setup and circuit diagram are shown below.

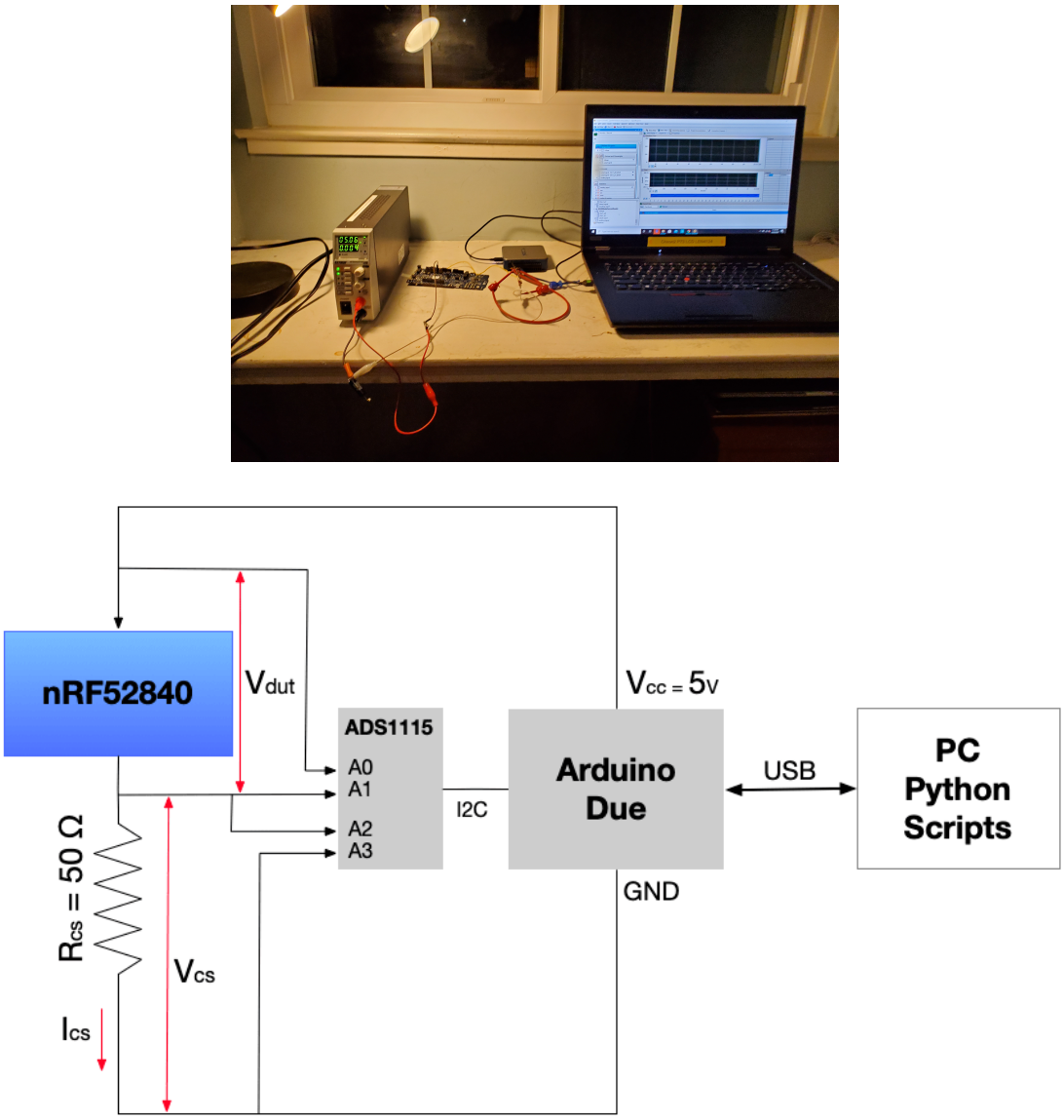


Figure 6.1: Energy Mesasurement ADS1115 + Arduino Due

The following equations display how the energy used was determined. E_{dut} and P_{dut} represent the energy and power consumed by the nRF52840 respectively. I_{cs}, V_{cs}, and R_{cs} represent the current, voltage, and resistance of current sense resistor respectively.

$$I_{cs} = V_{cs}/R_{cs} \tag{6.1}$$
$$V_{dut} = V_{cc} - V_{cs} \tag{6.2}$$
$$P_{dut} = V_{dut} * I_{cs} \tag{6.3}$$
$$E_{dut} = P_{dut} * T \tag{6.4}$$

Due to calibration errors and lack of more precise equipment, there may be some errors in the detected values but the values are roughly around what should be expected. Some other more precise methods for measuring current and power would be to use the NI DAQ USB-6009 or the High Voltage Power Monitor from Monsoon Solutions. Circuit diagrams for these approaches are provided below.

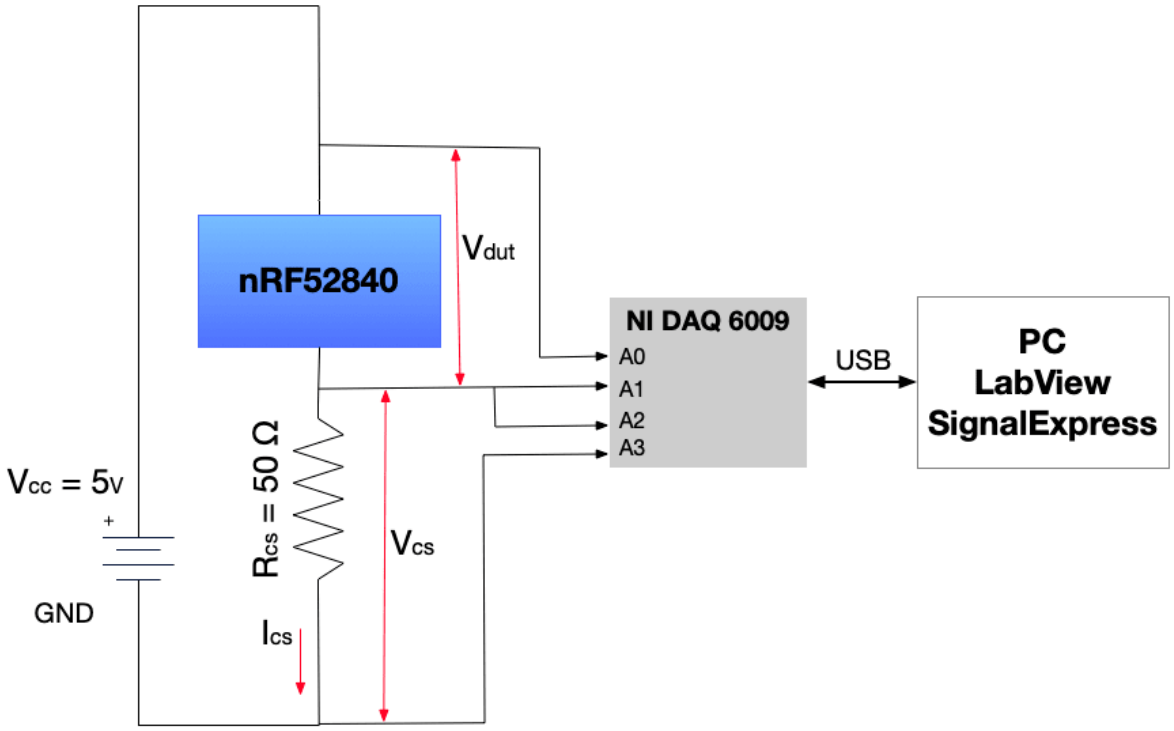


Figure 6.2: Energy Mesasurement NI DAQ USB-6009

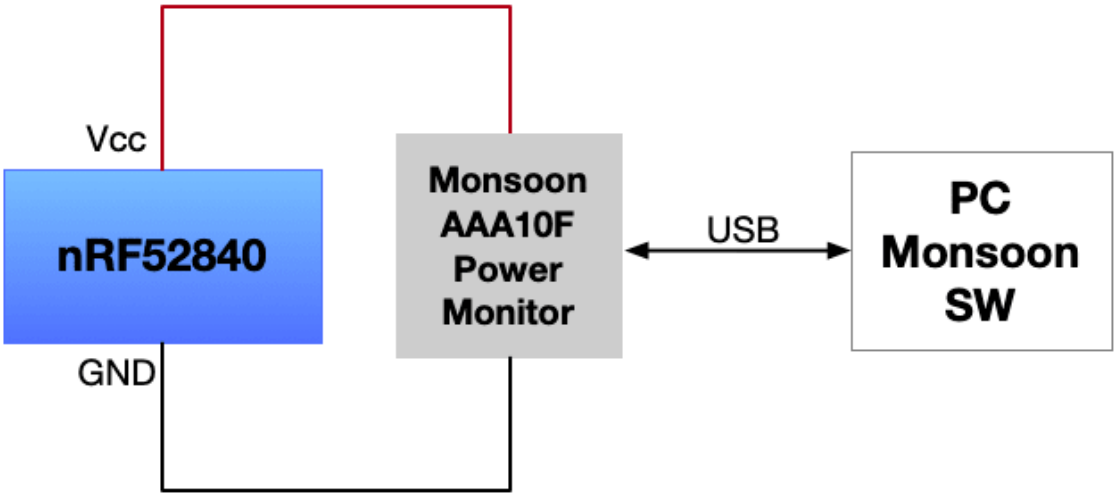


Figure 6.3: Energy Mesasurement Monsoon

Bibliography

[1] Romil Bhardwaj et al. "Autocalib: automatic traffic camera calibration at scale". In: *ACM Transactions on Sensor Networks (TOSN)* 14.3-4 (2018), pp. 1–27.

[2] Kyeongryeol Bong et al. "14.6 A 0.62 mW ultra-low-power convolutional-neural-network face-recognition processor and a CIS integrated with always-on haar-like face detector". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2017, pp. 248–249.

[3] François Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[4] Aakanksha Chowdhery et al. "Visual wake words dataset". In: *arXiv preprint arXiv:1906.05721* (2019).

[5] *Comprehensive Guide to IoT Statistics You Need to Know in 2020*. Apr. 2020. URL: https://www.vxchnge.com/blog/iot-statistics.

[6] *Convolutional Neural Networks*. Apr. 2020. URL: https://cs231n.github.io/convolutional-networks/.

[7] *CR2032 Datasheet*. Apr. 2020. URL: http://data.energizer.com/pdfs/cr2032.pdf.

[8] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[9] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014.

[10] Maria Gorlatova et al. "Energy harvesting active networked tags (EnHANTs) for ubiquitous object networking". In: *IEEE Wireless Communications* 17.6 (2010), pp. 18–25.

[11] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[12] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[13] Kevin Hsieh et al. "Focus: Querying large video datasets with low latency and low cost". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 269–286.

[14] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[15] Neal Jackson, Joshua Adkins, and Prabal Dutta. "A long-lifetime sensor platform for a reliable internet of things: demo abstract". In: *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. 2019, pp. 331–332.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[17] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus". In: *arXiv preprint arXiv:1801.06601* (2018).

[18] Yann LeCun et al. "Object recognition with gradient-based learning". In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[19] Hao Li et al. "Pruning filters for efficient convnets". In: *arXiv preprint arXiv:1608.08710* (2016).

[20] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[21] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[22] *nRF52840DK*. Apr. 2020. URL: https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK.

[23] *Open Thread*. Apr. 2020. URL: https://openthread.io/.

[24] Shishir G Patil et al. "Gesturepod: Enabling on-device gesture-based interaction for white cane users". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 2019, pp. 403–415.

[25] Joseph Polastre, Robert Szewczyk, and David Culler. "Telos: enabling ultra-low power wireless research". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE. 2005, pp. 364–369.

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[27] *SAME7 ARM Cortex-M7 Microcontrollers*. Apr. 2020. URL: http://www.atmel.com/Images/Atmel-11296-32-bit-Cortex-M7-Microcontroller-SAM-E70Q-SAM-E70N-SAM-E70J_Datasheet.pdf.

[28] Siddharth Sigtia et al. "Efficient Voice Trigger Detection for Low Resource Hardware." In: *Interspeech*. 2018, pp. 2092–2096.

[29] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[30] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[31] Bichen Wu et al. "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[32] Shuochao Yao et al. "Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework". In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 2017, pp. 1–14.