CS498IoT Lab 1 Report

Tanya Verma: tanyav2,
Andrew Chen: andrew6
Michael Rechenberg: rchnbrg2
Vivek Gupta: vivekg2

February 1, 2019

1 Important Links

Github Repository: https://github-dev.cs.illinois.edu/tanyav2/Lab1
Wiring Diagram: Google Drive Link to Fritzig Wiring Diagram
Demo Videos: Final Demo, Object Detection, Tensorflow, Network and Logs

2 Design Considerations

2.1 Computer Vision System

The code for the computer vision system is stored within the raspi_code/ directory, with driver.py serving as the main entry point for using object recognition with the Pi. In the main thread, we continuously loop to perform the following: use Picamera to capture an image from the camera module on the Pi at 1280x720 resolution, use OpenCV to resize the captured image to 300x300 (to make object recognition occur fast enough that the FPS acceptable), use a pretrained Tensorflow model (ssdlite_mobilenet_v2_coco_2018_05_09) to perform the object recognition and draw bounding boxes on the detected objects, and then send the annotated image over UDP sockets to the head unit. A separate thread continually listens over UDP for distance updates from the distance sensor Arduino and after receiving an update, the Pi decides whether to brake or not, sends the braking decision to the brake unit via a UDP socket, and then forwards the distance read to the head unit over a different UDP socket.

We separated the object recognition and brake signaling into 2 different threads for a few reasons. For one, the object recognition is a computationally intensive task whereas communication with the brake unit is more I/O focused, so separating those two helped boost performance. Also, we wanted the latency between the Pi deciding to brake and the brake LED being turned on to be as short as possible (stopping the car is more important than rendering the head

unit faster), so having the braking-decision-and-Arduino-networking code running within a separate thread rather than occurring serially after we processed an image in the main thread helped with that latency.

2.1.1 Why are quantized models better for resource constrained devices?

Quantized models, by definition, take the floating-point weights generated from model training (e.g. floating-point in float32) and discretizes those weights to a less precise but more compact data representation (e.g. fixed-point representation in uint8). Operations for fixed-point arithmetic can execute faster and draw less power by utilizing different hardware instructions (e.g. SIMD instruction sets) than required for floating-point arithmetic.

In addition, the fact that a weight in a quantized model requires less bits to represent than a weight in a non-quantized model means that the overall storage requirement for a quantized model is smaller; this makes model updating and storage on limited-storage-capacity devices more feasible. With all these considerations, quantized models are better for devices with constrained storage, computational power, and power supply like the Raspberry Pi in our lab.

2.1.2 Would hardware acceleration help in image processing? If so, have the packages mentioned above leveraged it? If not, how could you properly leverage hardware acceleration?

Hardware acceleration would help in image processing. By design, general-purpose CPUs are meant to execute a variety of different types of instructions at an acceptable efficiency, so the CPU architecture has to make compromises to support all those instructions. However, if your application performs just a few types of operations, a specific processor architecture can be optimized to only support those few operations so as to allow designers to make those efficient to compute (see CS233, CS433).

One of the backbone operations for neural network training and image processing is matrix multiplication, so hardware optimized for matrix multiplication will speed up the image processing. This is part of the reason why GPUs typically outclass CPUs for image processing: matrix multiplication can be computed in a parallel fashion efficiently, so often the parallelism means hundreds of slower cores of a GPU can compute matrix multiplication faster than the few but faster cores of a CPU (for sufficiently large amounts of data... if the data is too small, then synchronization overhead nullifies any performance gains from the parallel execution).

Tensorflow leverages hardware acceleration to speed up model training and inference and OpenCV supports hardware acceleration. However, for the lab we are just using the Raspberry Pi's quad-core CPU.

2.1.3 Would multithreading help increase (or hurt) the performance of your program?

Multithreading can help the performance of the program, but reckless multithreading could waste CPU cycles on busy waiting or result in unacceptable synchronization issues. One possible design for this lab is to have one thread execute a loop that does the following in serial order: (block until distance sensor information is received from the head unit, capture image from camera, run object recognition model, decide to brake or not, send signal to brake unit, block to send information to the head unit).

This is simpler to write code for, but it serializes I/O operations (socket communication with brake unit and head unit) with computationally intensive operations (performing inference with the object recognition model). This also means that the latency to send a brake a signal is bounded by network speed AND the time needed to perform object recognition.

Our code uses multiple threads to decouple some of the I/O operations from the compute-intensive operations. Thread A continuously gets the image from the camera and then performs the object recognition, while Thread B does the network communication with the brake unit. The two threads communicate over global variables. We neglect explicit synchronization between these two threads for performance (it's important that Thread B can signal to turn on the brake LED as fast as possible) and simplicity, and justify it because Thread A only ever writes the results of the object recognition (detected classes and probabilities) and Thread B only ever reads those results and uses the distance from the brake sensor. Thread A performs its writes much less frequently since the object recognition takes around 800ms to complete for one image whereas Thread B receives information from the brake sensor multiple times a second.

In addition, the physical motion of objects captured by the camera will be continuous (a stop-sign doesn't teleport around, a pedestrian walks in a continuous line in space), so it's more acceptable that Thread B may read detected classes from the current or just previous frame (or partially from each frame if the operations are interleaved in such a manner) since any detected objects would have just translated within the image between those two frames and we aren't using any positional information of the objects when deciding to turn on the brake.

2.1.4 How would you trade-off between the frame rate and detection accuracy?

We could trade off between the frame rate and detection accuracy (the accuracy of the object recognition model... does it detect the stop sign correctly?) through various means. In our current setup of our lab, the camera captures images at 1280x720 resolution, then we resize the image to 300x300 resolution

before feeding the resized image to the object recognition model to keep the FPS to at least 1FPS.

To increase the detection accuracy, we could resize the image to a larger resolution or forego resizing altogether so that the object recognition model has as much input data from the image to work with. However, this would tank the FPS since more input data means that the object recognition requires longer to complete.

We could also use a quantized model to speed up the inference (we did not use a quantized model for logistic reasons... see Discord) at the cost of lower detection accuracy.

2.2 CAN based Arduino to get distance and power LED

2.2.1 Physics Calculation

The HC-SR04 Ultrasonic Module sends a 40,000 Hz ultrasound and waits for the it to bounce off of the nearest object back to the sensor. The time for this is measured and so the distance is calculated by the speed of sound (343m/s) times the measured time divided by two since the signal covers double the actual distance when traveling to the closest object and back.

2.3 CAN to IP Gateway

The CAN to IP Gateway is an Arduino that is used to convert CAN packets to IP packets and vice versa. This is done using some boilerplate code from the Seeed CAN library and the Ethernet library, as well as by outlining a protocol for transmission of the data. The purpose of this gateway is outlined below.

- Raspberry pi (ADAS) sends brake signals to the Arduino that serves as the ECU. These brake signals are sent as IP packets to the CAN to IP gateway, which converts them to CAN packets that is transmitted over the CAN bus to the ECU arduino.
- The ECU arduino also sends the distance data to the raspberry pi (ADAS). The raspberry pi also runs a socket server, and hence this data needs to be converted to an IP packet

2.3.1 Would you consider UDP or TCP?

UDP is used in this case. This is because we control the entire network, ie, the endpoints as well as the transmission channel. Because the transmission channel uses Gigabit ethernet links, the probability of packets dropping is really low. There is also no multicast going on that would cause reordering of packets.

2.3.2 Can you realize full-duplex communication between the CAN ECU and the Raspberry Pi?

Currently we have half-duplex communication overall. However, we have full duplex between the CAN to IP gateway and the raspi. This is because the communication is happening between different ports. The CAN to IP gateway uses port 8888 to receive packets from the raspi, and the raspi uses port 9000 to receive packets from the CAN to IP gateway.

The reason we don't have full duplex overall is because the ECU Arduino switches between transmitting CAN packets and receiving CAN packets in the loop() code. It is not possible to transmit and receive at the same time as only one serial line (Serial Pin 10) is connected from the ECU Arduino to the MPC2515 CAN controller.

2.4 The Head Unit and The Network

Our head unit uses 2 UDP sockets in 2 different threads to listen distance and image updates from the Pi, along with a Flask server to serve that information to our web-based client that serves as our UI. Each of the UDP-socket-listening threads, upon receipt of new information, writes the received data to their respective global variables. When the web client requests an update, Flask reads from those global variables in order to return to the client the most recently received data from the Pi. The Javascript in our web-based client for the head continuously polls the Flask server for updates .

The Flask part of the head unit listens only to connections from the ADAS. It receives distance data and the video frame containing the recognized object after the Tensorflow model has acted on the image. It then renders these two things on the webpage.

2.4.1 How does sending data over the network influence the Pi's performance?

Sending data over the network influences the Pi's performance because the latency of network I/O eats up processing time that could be used for performing object recognition at a faster FPS. Currently, the main thread updates the head unit after performing inference, so the I/O of that update introduces a delay between the processing of the next captured image; but the delay was not that noticeable (around 1.1 FPS processing to around 1.2 FPS processing). We also make another network call to update the head unit with the new distance reading after sending the brake signal to the brake unit, adding a delay in that thread's main loop. However, the delay was seemingly negligible to the overall performance of turning on the brake LED; this is probably because our network was small and reliable (all the devices had wired connections to a fast ethernet switch).

2.4.2 How should we interconnect them in the Ethernet network?

We have an Ethernet switch that we use to assign static IPs to each of the IP devices on the network. The head unit has IP 10.0.0.4/24, the raspberry pi is 10.0.0.2/24, and the CAN to IP gateway is 10.0.0.3/24.

3 End To End Performance

3.1 Video Frame Rate

We measured three different situations and took five samples of each situation.

We measured the time difference between when the object was first seen by the camera to when the frame containing the bounding boxes identifying the object was displayed on the head unit.
All readings are in seconds.

Stop Sign on a screen:
Readings: 3.56, 3.38, 2.20, 2.43, 2.72
Average: 2.85

Person walking by (distance roughly the length of a large table):
Readings: 1.97, 1.79, 1.40, 2.45, 2.20
Average: 1.96

In the next case, we observed the total number of frames that contained an image of a car moving in a youtube video to the number of frames in which said car was detected by the model.

Car moving in youtube video (Frames that contained the car vs total number of frames):
8/14, 7/14, 8/15, 8/15, 9/16
Readings: 0.57, 0.50, 0.53, 0.53, 0.56
Average: 0.54

3.2 Detection Accuracy

We measured the response time of the brake LED. We started a timer as soon as an object was placed within 15cm (threshold of activation of brake LED) of the ultrasonic sensor, and stopped it as soon as the brake LED turned on.

Readings (in seconds) : 1.08, 0.8, 0.98, 0.90, 1.15
Average (in seconds) : 0.98

Next, we measured how long it took for the distance from an object to be

updated on the head unit.

Readings (in seconds) : 1.30, 1.23, 1.10, 1.06, 1.20
Average (in seconds) : 1.17