Prasanth Shaji, Deepak Venkataram

Benchmarking Training of Neural Networks on Embedded Devices

Comparing Training of Neural Network Frameworks vs Systems Programming Languages like C/C++

UPPSALA
UNIVERSITET

Training on embedded devices is an area that still requires a lot of attention

Contents

Part I:
Introduction

An embedded system is a combination of hardware and software components put together to achieve a specific task. Often, embedded systems are built into a larger device or system and are used to collect, store, process, and analyse data, as well as to control the device's behaviour. Embedded devices are a category of tiny devices with physical, computational and memory constraints that are programmable to perform dedicated tasks.

Like most of the automotive industry, Scania employs embedded systems called Electronic Control Units (ECUs) in their trucks to supervise and regulate essential subsystems like the engine, transmission, braking, and electrical systems. Each of these subsystems has one or more ECUs to gather system data and transmit it to a central communicator where the data is processed and the systems operations are monitored.

Scania currently runs a massive fleet of around 600,000 connected heavy vehicles. The company's truck sales make up 62% of its global sales and Scania has been adding 60,000 trucks to it's fleet annually [6]. This large fleet of rolling vehicles that are connected though the communicators opens up new possibilities. The different connected devices that monitor the health and state of the vehicle can be trained to predict system maintainance needs accurately and efficiently. For example, if the system can predict accurately that tire changes are required in 100kms then the driver can plan the route smartly to reach the workshop before the vehicle breaksdown. This opportunity can be realised by running smart algorithms on the hardware that is currently available.

Machine learning (ML) on embedded devices is becoming increasingly popular due to its ability to provide real-time insight and intelligence to devices. This technology can be used to automate tasks, improve efficiency, and make better decisions. But this technology presents a unique set of challenges due to the limited resources available on these devices. Embedded devices are designed to be power efficient, have limited memory and processing power, and require closely tailored algorithms, making it difficult to use pre-existing machine learning models. Furthermore, embedded devices are often expected to produce real-time results, which further complicates the development process. Despite these challenges, machine learning on embedded devices has potential applications in a variety of areas, such as in the fields of robotics and autonomous vehicles.

One such ML application Scania has been developing in their LOBSTR [8] and FAMOUS [7] projects is the anomaly and fault detection models in a federated learning environment. Targeting to run the anomaly detection models on the existing ECUs with limited resources has many benefits and challenges.

Benefits to performing Anomaly Detection on ECUs

- Scania is committed to promote a shift towards autonomous and eco-friendly transport systems. The latest addition of Scania's connected trucks and buses will be embedded with upgraded ECUs and communication devices. However, this upgrade will make the stock of older hardware devices to become obsolete and regarded as e-waste, which could be prevented. Exploring the possibility of repurposing existing ECUs to run ML models aligns with Scania's vision of leading the way towards a sustainable future.
- Neural networks (NNs) are a type of machine learning that can detect intricate patterns not only across multiple data signals but also over time. *include benefits to NN approach to Anomaly Detection*
- Federated learning methods facilitate the training of pre-trained anomaly detection models on the ECUs installed in Scania's distributed fleet of connected trucks. Each ECU individually trains the model with its data and transmits the updated model parameters to a central server. This distributed learning approach enables early detection of faults or failures and ensures that critical data remains on the device. Also dependency on network bandwidth is reduced as only the aggregated model updates are communicated over the network, instead of transmitting the entire data sample.

Challenges to implementing Federated Models

- To reap the best benefits of these approaches, training of the model needs to be performed on board. However much of the potential of running machine learning applications on these devices remains unattained due to the difficulties in creating these applications and running training on-board. Approaches such as TensorFlow Lite (TFLite), Edge Impulse, and STM Cube AI implemented along the TinyML frameworks, enable running ML models targeted for small resource devices. However these approaches are largely limited to inference capabilities and there is no adequate open source support in the existing infrastructure for training ML models.
- An Original Equipment Manufacturer (OEM) is responsible for the development and upkeep of the Scania ECU. However, the amount of information made available regarding the hardware design, memory layout, and operating system (OS) is restricted. To construct an embedded OS for a customized hardware, critical details such as the device tree, memory organisation, and boot flow are necessary. Obtaining this information from a functional board can be an enormous task requiring reverse engineering expertise.

Problem Description

The scope of the thesis is to repurpose the existing Scania ECU and explore the challenges of building targeted NN models and training them on repurposed ECU using different approaches and evaluating their performances.

1. Background

Developing and maintaining applications that rely on neural network models on a fleet of embedded devices has several considerations. The application deployment process should allow for continuous updates to the neural network, transfer data or model updates from the embedded devices to off-board analytics or machine learnining pipelines, and not interfere with the other applications on the embedded device, all the while maintaining correct representations in the neural network model. It is thus important to have an operating system that can support these applications with features such as process isolation, inter process communication mechanisms, multitasking etc.

The target embedded device to run these applications are the ECUs aboard a Scania vehicle. These ECUs have application processor cores that are capable of running rich operating systems such as linux distributions or real-time operating systems such as QNX, or VxWorks. All these operating systems also support hypervisors which allows for configurations where a host operating system runs standard automotive applications in addition to a guest operating system running the neural network application. This approach has the advantage of mitigating application crashes in the guest operating system and can provide a level of protection against software vulnerabilities [4]. Linux is the prefered choice for such a guest operating system due to its configurability and rich support for application development.

The next section looks at developing such an embedded linux environment and the process of developing neural network applications for that operating system.

1.1 Development For Embedded Linux

Building and maintaining embedded linux distributions with linux kernel and user mode applications require tools that can provide build configuration support at multiple levels, build a cross compiling toolchain or interface with one, support for several c run times, and provide support for project management. There are several tools that provide this support such as OpenADK, The Yocto project, Buildroot, OpenWrt, etc. Out of them Yocto and Buildroot are the most widely used and most featureful.

1.1.1 Toolchains & Cross compilers

Creating applications that are to be run on an embedded devices requires a set of software components that are usually collectively referred to as Software Development Kits (SDK). This suite of programs usually contain a toolchain that is capable of converting application source code, such as those in C or C++, into executables that can be run on the target embedded device.

Software development toolchains consists of a compiler, linker, libraries, debuggers etc to create and manage executable binary programs for a target device - the most common one being GNU binary utilities, a.k.a binutils. The primary choice for c compilers in this list of utility programs is GCC, with LLVM's Clang being another alternative. For developing applications that interface with the linux operating system APIs the toolchain also produces necessary header files called linux kernel header files. The last important piece of a toolchain will be the C runtime, with the most popular choice being GNU's glibc.

Building an embedded linux kernel requires several configuration parameters describing the kernel configuration, enabled feature, and more. To port linux onto a process on a particular board requires creating a boot loader capable of that task as well. A boot loader program is responsible

for placing an operating system into memory. This process will also need to be supported by the build system configurations. The majority of the details as to how the boot loader has to be configured will be based on the pariticular hardware that it will be configured for.

Supporting embedded hardware requires a software stack that includes several components that were just presented. The initial target machine was an ECU filling the role of a coordinator on the truck. However due to certain components missing from the stack layed out previously, namely board level support components such as Yocto meta layer, or the board level The details of the attempt at uncovering this information is layed out in Appendix II. Ultimately a similiar board, namely the iMX6SDB evaluation board, with the required information publicly provided by processor chip vendor NXP was chosen as the target platform.

1.1.2 Developing using QEMU

1.2 Neural Network Application Development

In general, neural network applications are written using machine learning frameworks or software libraries meant for scientific computing. The machine learning frameworks themselves are build on top of multiple software libraries meant for specific aspects of machine learning calculations. Several libraries exist that target specific subproblems in such computations and provide optimised implementations of several different processor architectures.

1.2.1 Choice of Software Stacks and Programming Languages

As most neural network applications are written in frameworks like PyTorch and Tensorflow, they have thriving ecosystems that provide rich developer support. Machine learning based companies and their service offerings such as cloud machine learning platforms almost invariably targets these platforms and provide several software tools for developers to utilize. Developers in these platforms enjoy several resources such as productivity tools that allows for continuous integration and development, maintainance and other resources with features for performance profiling, debugging, orchestration etc. ML and neural network service provides always target for these frameworks in their produces.

Another aspect to consider is the programming language in which the neural network will be written up in.

1.2.2 Embedded Software Stacks for Neural Networks

One complication caused by relying on software stack of the manner discribed before is software bloat. This causes a bigger problem in the context of embedded devices where resources are limited. There has been significant efforts made to clear this concern for the world of embedded devices, especially motivated by interests in getting neural network applications ready for mobile devices

1.3 Training on Device

The traditional model for machine learning applications on embedded devices have

1.3.1 Federated Learning

link to federated learning, mention FAMOUS again

2. Theory

In the following section an overview of the nature of computations involved in neural network applications is presented. Afterwards introductions are made to some terminology associated with software development for embedded devices, contextualised in embedded linux and its application development. The final section gives a short overview of conducting application performance evaluation.

2.1 Neural Networks

Training a machine learning model is fundamentally computing several floating point multiply and accumulate operations.

2.2 Embedded Linux Environment

As described in the previous chapter, embedded linux and user mode applications are using embedded build systems such as Buildroot or Yocto.

2.2.1 A Simplified Embedded Boot Sequence

2.3 Performance Evaluation

Roofline model for a simple matrix multiply application on Cortex-A9

Part II:
Implementation

Neural network inference has recieved a lot of attention in Tiny ML. The popular frameworks for machine learning such as PyTorch or Tensorflow do provide approaches for porting neural network application written using those frameworks but with a focus on allowing for model inference. Targetting even smaller devices with Tensorflow based neural network models is possible for inference only applications via Tensorflow Lite Micro [1]. Efforts to allow training as well in these frameworks require more effort due to the compute and memory intensive nature of the training process.

This section contains the description of benchmark training applications created to test the performance of an ANN training cycle on an embedded board. The neural network structure, learning algorithm, and the dataset remain the same but the implementations are completed in traditional general purpose neural network frameworks as well as straightforward implementations in C, C++, and Python. The design and development of these application and an overview of the target hardware to perform the benchmarking are covered in the following chapters.

3. Design

The benchmark applications test the training phase of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [3] dataset. MNIST is a popular dataset of handwritten digits commonly used for training image processing systems. It is a popular starting point for neural network implementations and has been used as the primary dataset in the benchmark experiments. The target embedded device is an Electronic Control Unit (ECU) with a Cortex-A9 processor.

3.1 Neural Network Development Process

The target environment necessitates the use of cross compilers and as part of the development process multiple build environments and systems were examined. Ultimately, the primary platform that ended up being used was the Yocto Project extensible SDK (eSDK) based application development process running on a standard linux based build environment. The QEMU emulator was also employed at various staged to check the build, and further test the application before moving onto tests on the actual hardware.

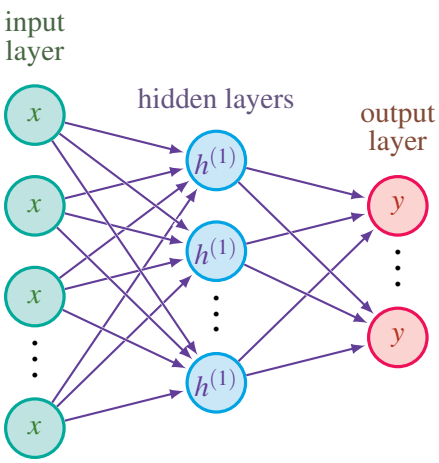
3.1.1 Compiler Toolchains, Python pakcages, & Yocto Recipes

The *meta-freescale* Yocto BSP layer by NXP supports the target processor and in combination with the Poky reference distribution provides an eSDK that was primarily used to test and develop the benchmark applications.

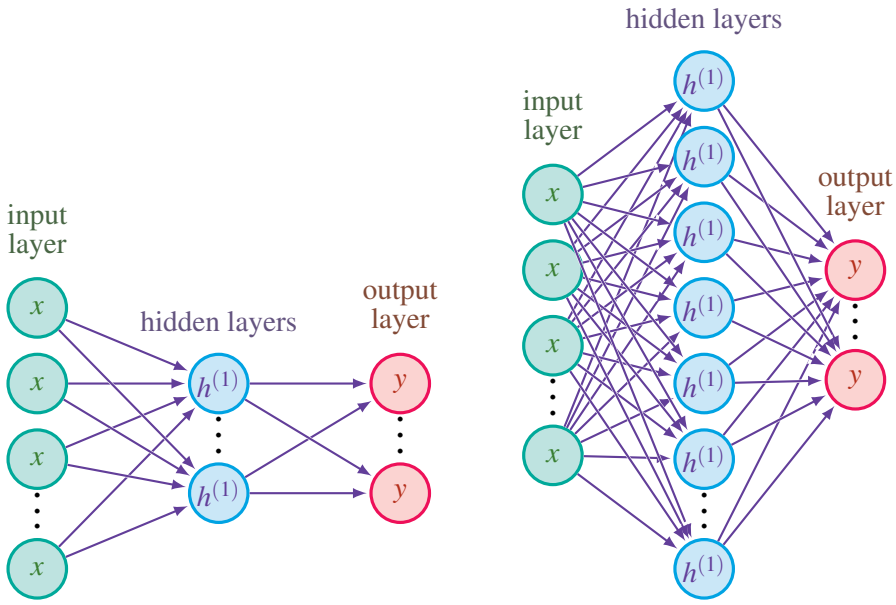
GCC based cross compilers and debuggers were usefull for the C, C++ programs. The *meta-python* layer provided by Open Embedded was also useful in allowing for applications using Python and Numpy. The general portability of the benchmark applications and the Yocto project allows for further experiments to be conducted on different target architectures as well. For further optimisations that relies on hardware specific features such as ARM's CMSIS-NN cannot be so easily ported however

3.2 HDR-NN Benchmark Programs

The handwritten digit recognition neural network is a fully connected neural network and derives from the popular neural network textbook neuralnetworksanddeeplearning.com



The input layer has 784 neurons corresponding to 28 x 28 pixel images of the MNIST dataset and the output layer has 10 neurons corresponds to 10 different possible digits. The dimensions and depth of hidden layers of the network is configurable as well as other properties of the learning algorithm



3.2.1 The Learning Algorithm

The HDR-NN benchmark applications will all share the same standard training algorithm, namely Backpropagation with Stochastic Gradient Descent. Describing this algorithm in general purpose neural network frameworks is straight forward and plenty of general implementations of the algorithm exists in the wild, making the development process easier to target multiple programming paradigms. The configurable parameters of the learning algorithm in through out the implementations are the learning rate, the total number of epochs for training, and the batch size for gradient descent iterations

3.2.2 Verifying Correctness

The model structure can be configured in the same manner across the implementations, as well as the learning algorithm configuration. This means that the shape of the model, the input parameters, the connections between the neuron can be configured in the same manner across the implementations. Furthermore, the learning rate, the number of epochs, and the batch size are also configurable in the same manner. Once the different implementations are configured in a similar manner, the training of the model is completed and the resulting weights are compared.

4. Development

The HDR-NN benchmark application were completed in different programming languages and in neural network frameworks like Tensorflow. Details about the target environment and the benchmark implementations are layed out in this chapter

4.1 Target Hardware

Exploring the target ECU board involved several examinations of a known state of the board. The linux kernel binaries were made via the Yocto project however there was no access to source code such as the recipes or the meta-layers themselves

The i.MX SoCs have a special boot mode named Serial Download Mode (SDM) typically accessible through boot switches. When configured into this mode, the ROM code will poll for a connection on a USB OTG port

4.1.1 i.MX6 Overview

The iMX6 series is designed for high performance low power applications and target boards are configured with a single Cortex A9 core with the ARMv7 ISA. The processor supports NEON single-instruction multiple-data (SIMD) instructions, allowing for SIMD vector operations within the training program

4.2 HDR-NN Implementation

With the primary focus on training, MNIST dataset was primarily loaded in an easily readable format appropriate to the corresponding paradigms and the correctness verification routines and execution statistics measurement runs were seperated. The benchmark executions did not produce disk I/O after the dataset was read, unlike the correctness verification runs which produced the final weights from the execution runs that were subsequently compared with the other benchmark program execution output weights

4.2.1 The Reference HDR-NN in Python

This is the baseline implementation and follows close to the implementation exhibitied on neural-networksanddeeplearning.com. The implementation uses the n-dimensional array data structure present in the popular Python programming language library Numpy

4.2.2 Tensorflow Lite based HDR-NN

Developing ANNs on tensorflow using Keras is straightforward with good support and well documented APIs. Building the same model for a Tensorflow Lite (TFLite) was more involved however still straightforward

4.2.3 C based HDR-NN

The C implementation had the least amount of external dependencies and contained the network in float arrays within structs.

4.2.4 CPP based HDR-NN

The CPP implementation used the n-dimensional array data structure feature of Eigen

Part III:
Analysis

A hand digit recognition neural network (HDR-NN) model is implementated in C, C++ Eigen, Python Numpy and Pytorch. The performance of HDR-NN training implementations was evaluated on the iMX6SDB evaluation board, which was programmed with an Embedded Linux built using The Yocto Project. To gauge the effectiveness of the models, we compared model accuracy, execution time, and peak memory usage while altering the number of layers and neurons in each layer. The results of these measurements are presented in the following chapters along with discussions on the obstacles encountered in developing the NN model and compiling it to operate on the target hardware.

5. Measurement

The benchmark applications were executed on an embedded linux operating system and the measurements were taken primarily based on the *times* system call and perf_events linux API. The primary tools for current measurement values given in the following chapter were taken using the GNU time. GNU Time provides timing statistics such as the elapsed real time between invocation and termination, the user CPU time, and the system CPU time, the later two via the *times* system call API. GNU Time also provides output lots of useful information on other resources like memory, I/O and IPC calls where available.

The priliminary measurements for the different executions completed with different learning algorithm parameters and model shapes across implementations were timing statistics and maximum resident set size (alternatively refered to as peak memory utilisation in the following chapter)

5.1 Benchmark Application Parameters

6. Results

A hand digit recognition application is implemented in different paradigms, specifically C, C++, Python, and Pytorch, which are the benchmark applications. Each this application is a fully connected feedforward neural network composed of multiple layers of neurons connected in a directed graph. The model has a constant input size of 784 and output size of 10. The hidden layer sizes vary depending on the implementation:

- C and C++ Eigen: 2, 4, 8, 32, 128, (32,16), and (128,16)
- Python-Numpy: 2, 8, 32, (32,16)
- Tensorflow/Pytorch:

The MNIST dataset is selected to train the model. This dataset contains 60,000 training images and 10,000 test images of hand-written digits. The model is trained using stochastic gradient descent, which is an optimization algorithm used to minimize a loss function. The backpropagation algorithm is used to calculate the gradients of the loss function with respect to the weights of the network. Finally, the mean square error loss function is used to measure the difference between the predicted output and the actual output of the network. The values of the biases and weights are initialized randomly with the PNGR random generator and a starting seed which are chosen to be identical for the different benchmark applications. The training hyperparameters are set to 30 epochs with a batch size of 10, a learning rate of 3 and sigmoid activation.

It is essential that the hardware utilsed for benchmarking closely resembles the Scania ECU's IMX6 processor, as this will make it easier to replicate the experiment on a repurposed ECU and will also provide the most precise results. The IMX6Q-SABRE Smart Devices evaluation board, which is armed with four 32-bit Cortex A9 cores, is an ideal choice. The Cortex A9 core is equipped with ARM V7 instruction set architecture and a powerful VFPv3 floating point unit with NEON SIMD capabilities. The processor has 32 KB instruction and data L1 caches, 1 MB L2 cache and 1 GB DDR3 SDRAM memory. The benchmark applications are designed to be run on a single core of the IMX6 processor, although it supports quad-core, to ensure the experiment is straightforward and easier to manage. This will also guarantee that the results are precise and accurate.

The yocto project is used to create a custom embedded linux distribution for the imx6qsabresd machine. The NXP yocto project guide (link) provides the instructions for building the Linux image, and additional packages such as cmake, python3 are installed during the build. The resulting image file, which used to flash the hardware, has a size of 300Mb.

The accuracy of the model is evaluated after each training epoch on the MNIST test set. After the training of the model for 30 epochs, the final weights and biases of the network and the accuracy on the test set are saved for analysis. This data is used to verify the correctness of the NN model in each benchmark application. The GNU time program is a great tool for monitoring the performance of applications. It allows us to measure the execution time and peak memory usage, which is used to compare the effectiveness of training the neural network model on the custom hardware implemented with different paradigms.

The python script created runs the experiment, executing each of the benchmark applications (C, C++, Python, Pytorch) one after the other. Every benchmark application is designed to be repeated 10 times, and all the measurements for each of the hidden layer configurations are saved for each of these iterations. The average values of the model accuracy, execution time and peak memory usage across all iterations are utilized for the analysis.

(we also reason the impact of execution time in the decision to skip some network configurations and add a table of the experiments conducted)

6.1 Evaluating Correctness

6.1.1 Accuracy

As the benchmark applications are developed to be identical by keeping the same structure and configurations, the model accuracy is expected to be similar. The (figure 6.1) showcases that the different implementations perform similarly irrespective of the number of parameters.

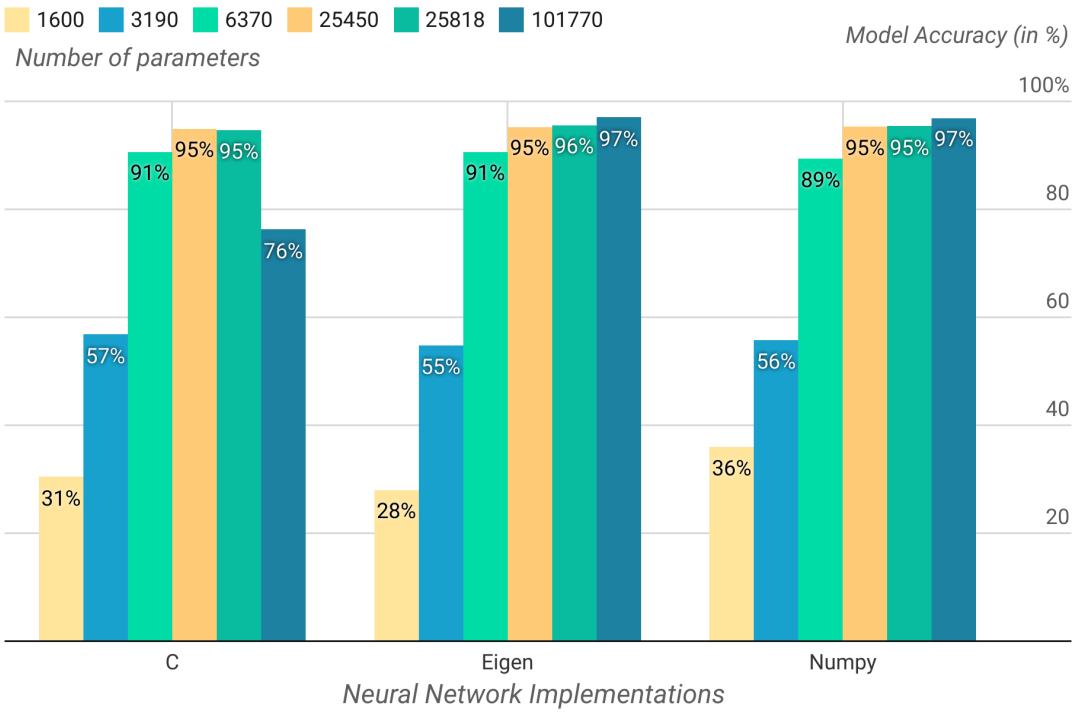


Figure 6.1. Comparing the accuracy of the different HDR-NN implementations.

Further, an abnormal behaviour can be observed when the number of parameters exceeds 101770. The accuracy of the C implementation decreases due to (an unknown bug).

(TODO: evaluate the mean squared error in accuray between the benchmark applications)

6.1.2 Weights and Biases

(TODO: evaluate the mean squared error in the generated weights and biases between the different implementation. Also, reason how the data structure in each of the implementation influence the error.)

6.2 Evaluating effectiveness

6.2.1 Execution Time

The training time of the neural network applications increases exponentially as the network size increases by the power of 2 because the number of parameters in a fully connected network increases exponentially as the number of neurons increases. This leads to an increase in the amount of calculations needed for the network to learn, resulting in a longer run time for the training process. This behaviour can be observeed in (figure 6.2) where the execution time increases drastically as number of parameters increases.

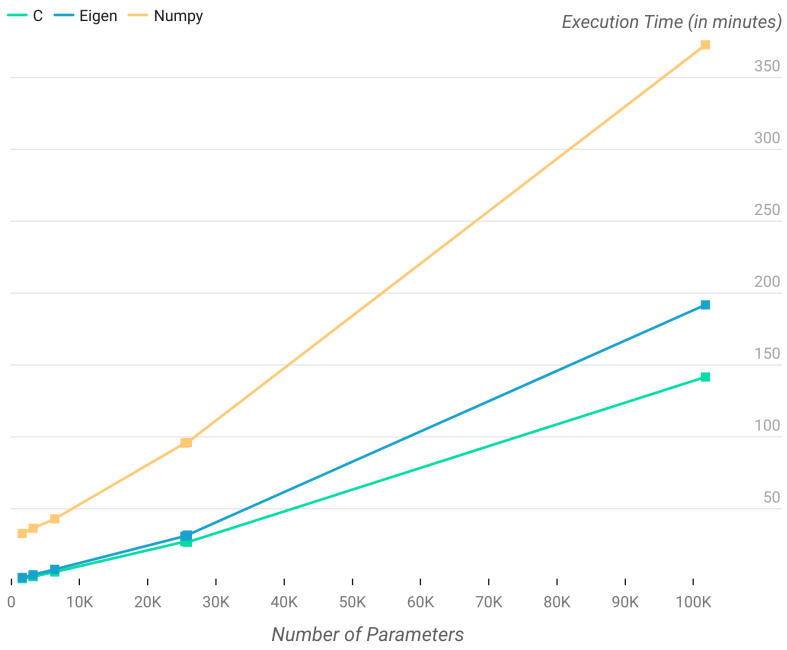


Figure 6.2. Comparing total run time for training the different HDR-NN programs

6.2.2 Peak Memory Usage

Regardless of the hidden layer sizes, the peak memory utilisation remains constant for the NN application across all implementations. The C++ Eigen implementation has the lowest run time memory footprint, while Python Numpy is the least efficient.

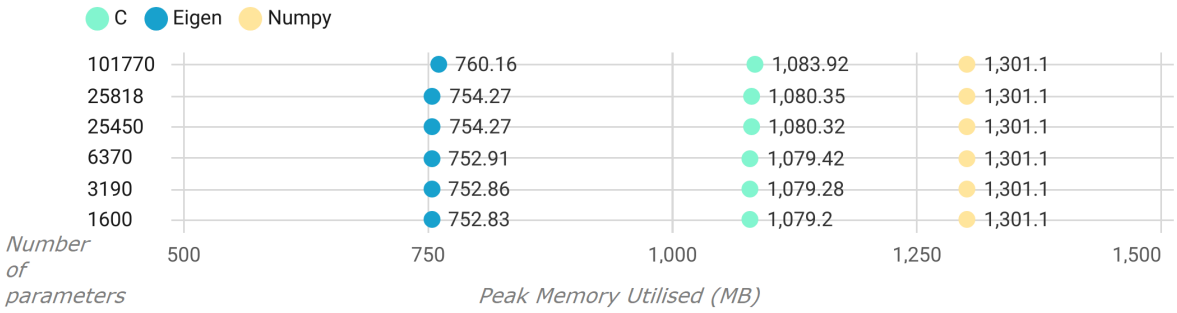


Figure 6.3. Peak Memory Utilized during training with different model sizes remain similar within the same implementation

(here we evaluate the percentage/ratio/cov between the applications.)

6.3 Comparing the implementations

The tables in (figure 6.4 and figure 6.5) shows the comparison between C-Eigen, Eigen-Numpy and C-Numpy. It is calculated as the percentage of 1 - (x/y) where x is performance value of the application which has lower value and y is performance value of the application that has higher number.

Network	C - Eigen	Eigen - Numpy	C - Numpy
1600	38%	93%	96%
3190	31%	89%	92%
6370	23%	82%	86%
25450	12%	68%	72%
25818	16%	71%	76%
101770	26%	49%	62%

Figure 6.4. Percentage difference between the implementations. Example: C is 38% faster than Eigen for the network size of 1600.

Network	C - Eigen	Eigen - Numpy	C - Numpy
1600	30%	42%	17%
3190	30%	42%	17%
6370	30%	42%	17%
25450	30%	42%	17%
25818	30%	42%	17%
101770	30%	42%	17%

Created with Datawrapper

Figure 6.5. Percentage difference between the implementations

6.3.1 C vs Eigen

For smaller models, it can be observed that C is faster than Eigen (for the network size of 1600, C is 38 percent faster than Eigen). But as model becomes complex, Eigen perform better and the difference is execution time is less than 15 percent. Again the abnormal behaviour can be observed in case of network size 101770 where C is 26 percent faster than Eigen. More test on the C implementation needs to conducted to identify the bug causing this behaviour.

With regards to memory utilisation Eigen perform better than C by 30%.

6.3.2 C vs Numpy

C constantly performs much better than Numpy. For the network size 101770, C is 62% faster. Numpy utilises 17% more runtime memory than C.

6.3.3 Eigen vs Numpy

Similar to C, irrespective of network size, Eigen is faster than Numpy by similar margins.

Eigen is efficient in memory utilisation and 42% better than Numpy.

6.3.4 Profiling

(TODO: perform profiling of the benchmark applications and note the results)

6.3.5 Failure/Fault testing

(TODO: perform the failure tests such as increases the network size until the application fails or system hangs.)

6.4 Repurposing Scania ECU

Scania ECU is like a black box with no information. A custom encased hardware that supports ethernet over modem and an UART interface along with hardware circuit schematic document was the only information available regarding the ECU. There is no information regarding the processor, memory support, bootloader. As the ECU is a production unit, there is no development tools on device and no support to port packages and application to the ECU. Many features on the bootloader, kernel were disabled making it futile to execute the common commands that provide system information.

The task of repurposing the Scania ECU comprised of reverse engineering and obtaining the required hardware/software information and flashing a custom operating system to benchmark the neural network applications. The first task was partially successfully as hardware information such as processor, architecture, I/O interfaces, device tree and software information such as kernel, compiler, glic and versions was obtained. But information regarding the memory layout and boot flow could not be concretely reverse engineered. The second task was not achieved as flashing custom embedded linux always resulted in the ECU being bricked. Experiments conducted from booting the normal operation and from serial download mode had different issues and failed. While flashing from the normal boot, only the bootloader is replaced in the mtd partition. This could have failed because of incorrect u-boot image with wrong device trees or loading kernel failed as version mismatch between bootloader and kernel or checksum failure or size of the file is big overwriting a different region with crucial data. Serial download mode flashing required some crucial information regarding the memory load address and entry point for bootloader, kernel, root file system which is configured in the custom device tree. This information could be obtained from reverse engineering.

7. Discussion

7.1 Developer Experience

7.2 Early stopping

The training for all the implementations were executed by configuring the number of epochs as 30. This leads to the accuracy of model dropping significantly due to overfitting, which could be avoided if early stopping was implemented. But, early stopping is not implemented as the performance would be completely different and there wouldn't be a standard setting to compare the implementations.

7.3 General Distribution of Work

8. Conclusion and Future Work

What does it all mean? Where do we go from here?

References

[1] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems, 2021.

[2] Google. Tensorflow lite. https://www.tensorflow.org/lite. [Online; Accessed on March 30, 2023].

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition, 1998.

[4] Ning Li, Yuki Kinebuchi, and Tatsuo Nakajima. Enhancing security of embedded linux on a multi-core processor, 2011.

[5] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2018. [Online Accessed on March 30, 2023].

[6] Scania. Driving the shift: Annual and sustainability report 2022. https://www.scania.com/content/dam/group/investor-relations/annual-review/download-full-report/scania-annual-and-sustainability-report-2022.pdf. [Online Accessed on March 30, 2023].

[7] Vinnova. Famous : Federated anomaly modelling and orchestration for modular systems. https://www.vinnova.se/en/p/famous---federated-anomaly-modelling-and-orchestration-for-modular-systems/. [Online Accessed on March 30, 2023].

[8] Vinnova. Lobstr : Learning on-board signals for timely reaction. https://www.vinnova.se/en/p/lobstr---learning-on-board-signals-for-timely-reaction/. [Online Accessed on March 30, 2023].

Appendixes

Scania C300 Communicator

Scania ECU was set to be the target hardware to benchmark the training of a machine learning model. In order to be able to port the different implementation applications and it's dependencies on the Scania ECU and successfully execute the programs, certain information regarding the hardware, kernel, supported compilers and libraries is needed. Since the existing ECU is developed by an OEM, obtaining all the information and source files is not possible. It can be achieved by replacing the existing software with a custom developed embedded linux distribution, thus repurposing the custom hardware. The biggest challenge for repurposing an custom board with no information is reverse engineering to obtain the required information for flashing a custom linux kernel. The reverse engineering learnings are stated in this section.

(TODO: No development tools, no bootloader access, no serial console prints from the bootloader, no device tree info, no memory layout info, no boot flow info, challenging to port or install any tool/package on device basic information to gather: processor, number of core, architecture, memory units supported, kernel info (name, version, distribution), file system, bootloader, system boot flow)

The naive approach of flashing the mtd partition that houses the bootloader. To verify that it possible to flash and boot the board, a dump of the existing bootloader was taken and flashed in the same partition. This worked and the device booted successfully. Next, the u-boot bootloader developed from the yocto project was flashed on the bootloader mtd partition. The broad was bricked (reasons: incorrect u-boot image with wrong device trees or loading kernel failed as version mismatch between bootloader and kernel or checksum failure or size of the file is big overwriting a different region with crucial data).

(TODO: Using mfgtools on board, to collect information from bootloader.) (TODO: Using uuu tool in SDM mode to flash custom image.) (TODO: bootloader flashing using dd command)

(TODO: results from varying bootloader environment parameters) (TODO: results from mfgtools experiment) (TODO: results of unbricking the board) (TODO: reasoning not enough memory layout infomation)