

Prasanth Shaji, Deepak Venkataram

Training Neural Networks on Embedded Devices

Comparing Training on 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	5
1 Background	7
1.1 Development Process for Embedded Linux	7
1.1.1 SDKs & Compiler Toolchains	7
1.2 Training on Device	7
1.2.1 Federated Learning	7
1.3 Neural Network Applications on Embedded Devices	8
1.3.1 Different Programming Paradigms	8
1.4 General Distribution of Work	8
2 Theory	9
2.1 Neural Networks	9
2.2 Support for Neural Networks in Embedded Hardware	9
2.3 Embedded Linux Environment	9
2.4 Performance Evaluation	9
Part II: Implementation	11
3 Design	13
3.1 Artificial Neural Network Development Process	13
3.1.1 Cross compilers & Build System	13
3.2 Handwritten Digit Recognition (HDR)	13
3.2.1 HDR-NN Training	14
3.2.2 Verifying Correctness	14
4 Development	15
4.1 iMX6 Custom Board Target	15
4.1.1 ECU / iMX6 Evaluation Board Overview	15
4.1.2 Testing on Device	15
4.2 HDR-NN Implementation	15
4.2.1 Python - Numpy	15
4.2.2 Tensorflow Lite	15
4.2.3 C	16
4.2.4 CPP - Eigen	16
Part III: Analysis	17
5 Results	19
5.1 HDR-NN comparisons	19
5.1.1 Python - Numpy	19
5.1.2 Tensorflow Lite	19
5.1.3 C	19
5.1.4 CPP - Eigen	20
5.2 CMSIS-NN based Optimisations to Training	20
5.2.1 Quantisation	20
5.2.2 Pruning the Network	20

6	Discussion	21
6.1	Developer Experience	21
7	Conclusion and Future Work	22
	References	23
	Appendixes	25
	Build Systems	27
	The Yocto Project	27
	Scania C300 Communicator	28

Part I: Introduction

Embedded devices are a category of tiny devices with physical, computational, and real-time computing constraints constituting an embedded system. An embedded system is a combination of hardware and software components put together to achieve specific tasks. Often, embedded systems are built into a larger device or system where they are used to collect, store, process, and analyse data, as well as to control the device's behaviour.

Like most of the automotive industry, Scania employs embedded devices called Electronic Control Units (ECUs) in their trucks to supervise and regulate essential subsystems such as 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 via the Controller Area Network (CAN) interface. The existing fleet of trucks will receive upgraded ECUs and communicators with higher performance hardware enabling faster and advanced data collection. This upgrade presents an opportunity to reuse the on-board hardware that is currently operational.

One such opportunity lies in performing machine learning (ML) applications on these ECUs. Machine learning has the ability to provide real-time insight and intelligence to devices however their implementation in these systems presents a unique set of challenges due to limited resources available on embedded devices. Embedded systems 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 systems often have expectations such as real-time behaviour which can complicate the ML development process. Despite these challenges, machine learning on embedded systems has potential applications in a variety of areas, such as in the fields of robotics and autonomous vehicles.

One such application Scania has been working on in their LOBSTR [\[1\]](#) and FAMOUS projects is anomaly and fault detection. Targeting to run the anomaly detection models on the existing ECUs with limited resources provides many benefits.

Scania currently runs a massive fleet of more than 100,000 trucks and has been adding another 60,000 trucks annually since 2014. The company's truck sales make up 62% of its global sales. Scania has a substantial fleet of connected trucks with electronic control units (ECUs) and communication devices that are due for an upgrade. However, this upgrade will result in a massive amount of e-waste, which could be prevented. Scania is actively working towards a sustainable and autonomous future by devising strategies to promote a shift towards eco-friendly transport systems.

As part of this commitment, Scania is exploring the possibility of repurposing existing ECU's to run machine learning models. This innovative approach aligns with Scania's vision of leading the way towards a sustainable future.

Among the ML approaches to take, Artificial Neural Networks are especially interesting for anomaly detection due to several factors such as minimal data preprocessing ... To get the best out of this approach, training of the model needs to be performed on-board. However much of the potential of running machine learning applications on these devices remain unattained due to the difficulties in creating these applications and running training on-board.

Problem Description

The scope of the thesis is to explore the challenges of building and training artificial neural networks on embedded devices using different approaches and evaluating their performance.

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 continual updates to the neural network, transfer data or model updates from the embedded devices to off-board analytics or machine learning pipelines, and not interfere with the other applications on the embedded device 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 devices to run these applications are the ECUs aboard a Scania truck which 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 automotive applications in addition to a guest operating system . Linux is

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 the linux kernel requires .An overview of these build systems are presented in [Appendix I](#).

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 a Software Development Kits (SDK). This suite usually contains a toolchain that is capable of converting C or C++ source code into executables that can be run on the target embedded device

Supporting embedded hardware requires a software stack that includes several components covered in the preceding section. 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 , with the required information publicly provided by NXP - the vendors that provide the processor chip on the intial target ECU

1.2 Training on Device

Mention traditional offboard training and onboard inference architecture vs approaches with training on board. Reference Tiny ML research

1.2.1 Federated Learning

link to federated learning, mention FAMOUS again

1.3 Neural Network Applications on Embedded Devices

Neural network applications are generally written using machine learning libraries such as TensorFlow or PyTorch

1.3.1 Different Programming Paradigms

Approaches to doing Machine Learning in Embedded Environments. Emphasis on how these applications are developed - e.g TFLite

1.4 General Distribution of Work

2. Theory

2.1 Neural Networks

2.2 Support for Neural Networks in Embedded Hardware

2.3 Embedded Linux Environment

Describe the process of boot flow introducing concepts such as Boot ROM, eMMC, IVT, boot-loader, kernel, file systems etc.

2.4 Performance Evaluation

Performance evaluation of programs

Part II: Implementation

Neural network inference has recieved a lot of attension in Tiny ML. The popular with efforts such as providing neural network application programmers a framework to port. Targetting even smaller devices with Tensorflow based neural network models is possible for inference only applications via Tensorflow Lite Micro [\[2\]](#). 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 and other languages

3. Design

The benchmark applications test the training phase of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [4] 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) board based on an iMX6 series processor

3.1 ANN 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

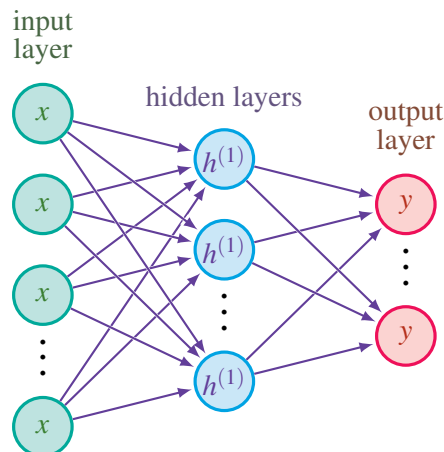
3.1.1 Compiler Toolchains & Yocto Recipes

The *meta-freescale* Yocto BSP layer by NXP supports the target processor and in combination with Poky can provide 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 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

4.1.2 Testing on Device

The benchmark tests were performed on ... using the perf tool

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 exhibited 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

Part III: Analysis

Results from the implementation

5. Results

The HDR-NN training implementations were benchmarked on the iMX6SDB evaluation board. Model accuracy, execution time and peak memory utilised during the training of the model is compared while varying the number of layers and the neurons in each layer

5.1 HDR-NN comparisons

The execution times for HDR-NN training were recorded for different network configurations such as differing hidden layer sizes of 2, 8, 32, and 128. The run times increased exponentially with the number of parameters. This is due to the fact that the amount of calculation in a fully connected network increases with the number of neurons, leading to longer training times

Further, when the number of neurons in a single layer exceeds 32, the accuracy of the model is observed to decrease due to overfitting. To improve accuracy, adding another layer with 16 neurons is found to be beneficial without significantly increasing the time required for computation. In fact, for larger network sizes, it is observed to even reduce the computation time required

Regardless of the hidden layer sizes, the peak memory utilisation remains constant for the same application regardless of the network configuration

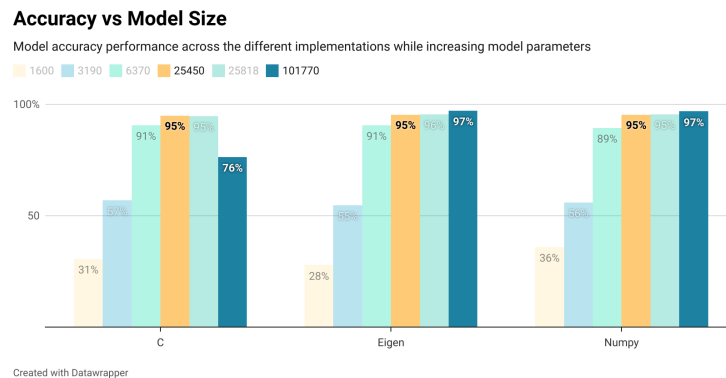


Figure 5.1. Comparing the accuracy of the different HDR-NN implementations

5.1.1 Python Numpy based HDR-NN

The Numpy implementation consistently took longer duration to perform the same training cycle as compared to the C implementation

5.1.2 Tensorflow-Lite based HDR-NN

Benchmark pending ...

5.1.3 C based HDR-NN

C implementation had lower execution times and memory usage

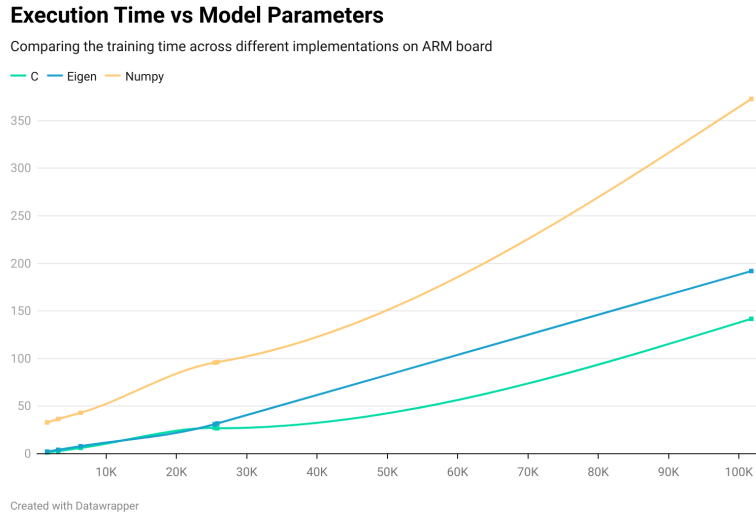


Figure 5.2. Comparing total run time for training the different HDR-NN programs

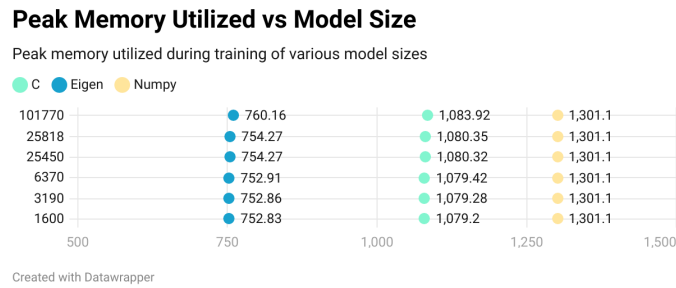


Figure 5.3. Peak Memory Utilized during training with different model sizes remain similar within the same implementation

5.1.4 CPP based HDR-NN

Benchmark pending ...

5.2 CMSIS-NN based Optimisations to Training

Further breakdown of the performance achieved from different optimisation techniques

5.2.1 Quantisation

future: Training Network with Quantized weights

5.2.2 Pruning the Network

future

6. Discussion

6.1 Developer Experience

7. Conclusion and Future Work

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

References

- [1] Juan Carlos Andresen. Lobstr learning on-board signals for timely reaction, 2020.
- [2] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezheng Wang, and Pete Warden. Tensorflow lite micro: Embedded machine learning on tinymml systems, 2021.
- [3] Google. Tensorflow lite. Accessed on March 21, 2023.
- [4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition, 1998.

Appendixes

Build Systems

The Yocto Project

The Yocto Project is an open source collaborative project that provides users with a set of tools to create custom Linux-based systems for embedded products. It's based on the OpenEmbedded framework and is backed by the Linux Foundation. The Yocto Project works with hardware vendors, open source communities, and hundreds of developers to provide a robust development environment for embedded products.

Yocto Project allows developers to create unique Linux-based systems for embedded devices. Yocto Project provides developers with the tools to customise their embedded Linux systems to meet the specific needs of their products. Yocto Project is used by many companies for their embedded products. It is especially useful for those developing custom embedded products, as it allows users to quickly create a customised Linux-based operating system.

Yocto Project provides many features that make it a great choice for embedded Linux development. These features include:

1. **Open Source** Yocto Project is an open source project backed by the Linux Foundation. This means it is free to use and developers can access the source code to customise their systems as needed
2. **Compatibility** Yocto Project is compatible with many types of embedded hardware, including ARM, PowerPC, MIPS, and x86. This makes it easy to use for any type of embedded project
3. **Robust Development Environment** Yocto Project provides a robust development environment for embedded Linux development. It includes libraries, tools, and debugging support to make development easier
4. **High Performance** Yocto Project provides an optimised development environment for embedded systems. This helps developers to create high-performance products quickly and easily
5. **Flexibility** Yocto Project provides developers with the flexibility to create custom Linux-based systems for their embedded devices. This allows developers to tailor their systems to meet the specific needs of their products
6. **Time Savings** Yocto Project makes it easier and faster to create custom Linux-based systems. This helps to reduce development time and save money

Scania C300 Communicator

The communicator contains iMX6 series processors