Prasanth Shaji, Deepak Venkataram

Training Neural Networks on Embedded Devices

Neural Network Frameworks vs Systems Programming Languages

UPPSALA
UNIVERSITET

There is great potential in enabling neural network applications in embedded devices and an important step in that is to allow for these device to perform the training of the neural network on board the device. Neural network inference is a popular and well supported functionality on these platforms however neural network training still has ways to go. In this project we take a closer look at this step and try to compare the performance capabilities of popular machine learning frameworks with straight forward implementation approaches. This report also contains a discussion on the nature of implementing neural network applications on top the fragmented embedded ecosystem.

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 process data generated within the device and to control the behaviour of the system. Embedded devices are a category of tiny devices with physical, computational, and power constraints that are programmed to perform dedicated tasks.

Like most of the automotive industry, Scania employs embedded devices 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 contain several ECUs to gather system data and transmit it to a central communicator where the sensor data is processed and the system 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 [15]. This large fleet of rolling vehicles that are connected though the communicators opens up new possibilities. These connected devices continuously monitor the state of the vehicle and this data can be used to accurately and efficiently schedule vehicle maintainance. For example, if a tire change is predicted to be required in 100 kms then the driver can plan the route smartly to reach the workshop before the vehicle breaks down. This opportunity can be realised by running smart algorithms on the hardware that is currently available.

Machine learning 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 machine learning application that Scania has been developing in their LOBSTR [17] and FAMOUS [16] projects is *anomaly and fault detection* on the vehicles. Anomalies in this context are patterns in the data that do not conform to the notion of normal behaviour. There are various machine learning techniques available for predicting anomalous behavior in trucks based

on the sensor data readings. The running anomaly detection models for fault prediction 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 are a type of machine learning technique that can learn intricate patterns across multiple data signals and time. Neural networks can learn from unstructured data and apply this understanding to unseen data points. Anomaly detection using neural network has the advantage of requiring less manual work in labelling data and can provide paradigms allowing for automation of discovering new and important data points.
- Federated learning techniques allow for the ECUs installed in Scania's distributed fleet of connected trucks to perform distributed training leveragin their computational capabilities. Each ECU individually trains the model with its own data and transmits the updated model parameters to a central server. This distributed learning approach enables early detection of faults or failures, reduces the network bandwith consumption by reducing the sensor data transmission, and ensures that critical data remains on the device.

Challenges to implementing Federated Learning on ECUs

- Neural network training is computationally demanding and achieving good performance on embedded devices require careful resource management.
- The potential of running machine learning applications on embedded platforms remains unattained due to the difficulties in creating these applications and running training of the model on-board. Approaches such as TensorFlow Lite (TFLite), Edge Impulse, and STM Cube AI implemented along with other 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 neural network models and training them on repurposed ECUs using different approaches and evaluating their performances.

Report Structure
This report is divided into three parts with multiple chapters containing several sections. The first part describes the background and introduces important information motivating the challenges in performing machine learning on embedded platforms. The second part details the bechmark applications that were implemented to evaluate the processes of training neural networks on a specific target platform. Lastly, the final part presents an analysis and discussion on implementing the training of a neural network.

1. Background

Developing and maintaining applications that rely on neural network models and run 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, 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 ARM 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 [8]. Linux is the preferred choice for such a guest operating system due to its configurability and rich support for application development.

1.1 Embedded Platforms

Embedded systems perform a varied mixture of applications including industrial automation, consumer electronics, automotive, etc. and have capabilities ranging from ultra low power wireless data logging to thrust vector control on rocket ships. Embedded boards for an application domain are generally designed in concert with multiple entities, each responsible for a different component in the hardware-software stack.

An example of such an arrangement could be as follows. The semiconductor and software design company ARM produces ARM processors that are used in a significant fraction of embedded devices due to their power efficiency and versatility. *Silicon vendors* such as Atmel, Qualcomm, NXP, etc. buy CPU core designs from ARM and adds a suite of peripherals to offer application processors. Ultimately, a *system maker* would select among these application processors, sensors, and other devices to design an embedded board for a particular application domain.There are several different practises that the industry adopts at creating embedded devices for different applications with the example presented here being one among several. The embedded device considered in this project derives from such a process.

1.2 Development Process for Embedded Linux

Building and maintaining Embedded Linux distributions with the Linux kernel and user mode applications require tools that can support multi-level build configurations, interface with or build a cross compiling toolchain, support `C` run times such as glibc or musl, and provide support for project management. There are several tools that provide this support such as OpenADK, The Yocto project, Buildroot, OpenWrt, etc., with The Yocto project and Buildroot being the most featureful and widely used Embedded Linux build systems. In comparison with Buildroot, the Yocto project supports a greater variety of hardware and also has faster incremental build times as

it caches the generated binaries [14]. The Yocto project was chosen as the primary build system and used to generate the Embedded Linux and application programs used within this project. The next chapter contains a section takes a look at a development environment that uses the Yocto project.

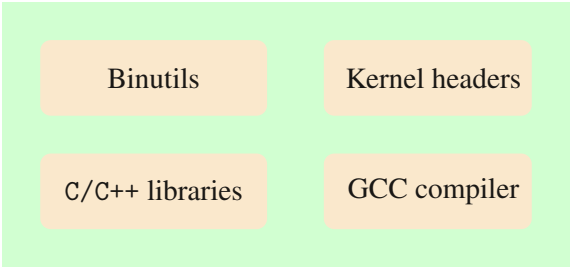
1.2.1 SDKs and Compiler Toolchains

Creating applications for 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, as well as a collection of programs to create and manage executable binary programs for a target device such as the commonly used GNU binary utilities, a.k.a binutils. The primary choice for a `C` compiler is the GNU Compiler Collection `GCC`, with LLVM's Clang being the closest alternative. To develop applications that interface with the Linux operating system APIs, the toolchain also contains 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.

1.2.2 Cross Compilers

The software development toolchains for embedded devices are generally run on a development machine that is different from the embedded device. In this configuration the compiler toolchain creates executables for a different platform that the one it is currently running on and is termed a *cross compiling* toolchain. A compiler toolchain that creates executables for the sample platform is termed a native compiler toolchain. Cross compilers are common due to several factors such as limited resources on embedded devices, ease of targetting multiple hardware platforms, etc. and they are ultimately an unavoidable part of creating programs for a new hardware platform. Most software that are run on embedded devices are created on a different computing platform. Such a computing platform in the context of cross compilation is referred to as a *development host* and the embedded device that the software ultimately runs on is the *target*.



A brief discussion on the ARM GNU toolchain

A toolchain can usually be described by a quadruple taking the general format `<arch>-<vendor>-<os>-<libc/abi>`. `<arch>` stands for the target CPU architecture for the binaries that will be produced by the toolchain, the system manufacturer or vendor who is responsible for the hardware is `<vendor>`, `<sys>` generally stands for the operating system or takes the special value "none" for bare-metal, and the target application binary interface or `C` runtime is `<libc/abi>`. This tradition of naming toolchains exists across build automation tools, compiler projects, etc in different ways under different names such as system definitions in autoconf, or the target triple in LLVM's Clang.

ARM GNU toolchain is a GNU toolchain for ARM architecture that is released and maintained by ARM and from open-source project GCC, Binutils, glibc, Newlib, and GDB. The toolchain

supports `C` and `C++` languages and supports CPUs based on the 3 main flavours of processor cores namely the Cortex A, Cortex R, and Cortex M.

Processor Core	ARM Architecture	ARM Processor Core
nRF51822	ARM v6-M	ARM Cortex M0
AM2732	ARM v7-R	ARM Cortex R5
AM3358	ARM v7-A	ARM Cortex A8
i.MX6S	ARM v7-A	ARM Cortex A9
Kryo 240	ARM v8-A	ARM Cortex-A73, A53

Table 1.1. A few ARM application processors with their processor cores and CPU architecture

1.2.3 Application development using QEMU

Another common alternative to cross compiling in this manner is by using native compilers via emulation. Emulation is some technique that allows a (host) computer system to simulate the behaviour of some other (guest) computer system. There are several software projects that allow for emulation in this manner with QEMU being by far the most commonly used emulator targetting several different hardware platforms. QEMU can also be used to create and test embedded applications before being deployed on the target hardware. Application development for embedded devices usually employs a combination of cross compiling toolchains and emulation software to create, test, and maintain the software.

An example Embedded Development Environment on Linux using QEMU

In this section let's consider a simple development environment capable of compiling and executing `C` programs for the `armv7l` ARM processor using a personal computer running linux. This setup leverages features of the Linux operating system running on the development host and the capabilities of the QEMU machine emulation system. The code listings in this section assumes a fedora workstation operating system and was tested using such a setup. The commands maybe specific to this environment, such as those for the fedora's package manager utility `dnf`. However the general principles maybe used to create a similar setup on other linux distributions as well.

The Linux kernel has a feature called `binfmt_misc` which stands for *Miscellaneous Binary Format*. This feature allows for interpreter programs to be associated and invoked upon using certain binary format files. Together with the `chroot` command, `binfmt_misc` can be leveraged to setup an environment that can use a QEMU emulator program to test and develop for a different architecture than that of the development host.

The interpreter programs registered with `binfmt_misc` are usually user space applications such as emulators and virtual machines. For this section the intepreter program will be a QEMU user space emulator program, namely `qemu-arm`. QEMU has multiple operating modes apart from a full system emulation such as its user-mode emulation mode. In user-mode emulation, QEMU can perform system call translation and POSIX signal handling in linux which effectively allows for programs compiled for a different instruction set to be executed using QEMU. Cross compiling and cross debugging are common use cases for this user-mode emulation in QEMU. `qemu-arm` is a QEMU User space emulator program for executing programs compiled for ARM.

After an interpreter and binary file format pair are registerd, `binfmt_misc` recognises the associated binary files by matching some bytes at the begining of a file with a magic byte sequence that had been supplied. The usage of magic bytes at the begining of files is a UNIX tradition adopted as a means of incorporating file type metadata within the file.

As with some linux features, `binfmt_misc` must first be mounted at specific location after which it can be configured.

```
mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc
```

To register the binary format and interpreter pair to setup the development environment, a string of the form `:name:type:offset:magic:mask:interpreter:flags` needs to be send (`echo`ed) to the `register` file at the path mentioned previously. Assuming that `qemu-arm` is present at the path `/usr/bin`, running the following command will register `qemu-arm`

```
echo ":qemu-arm:M::\x7fELF\x01\x01\x01\x00\x00\x00"
 ↪  "\x00\x00\x00\x00\x00\x00\x02\x00\x28\x00:"
 ↪  "\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff"
 ↪  "\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff:"
 ↪  "/usr/bin/qemu-arm-static:F" > /proc/sys/fs/binfmt_misc/register
```

Note that the `echo` should produce a string without whitespaces. A new file named `qemu-arm` will show up in the path `/proc/sys/fs/binfmt_misc/` upon successful completion of the `echo` command.

Another linux feature is `chroot` which allows for changing the apparent root directory of a running process and its children. `chroot` system call interface started with UNIX and is useful for testing and developing software systems in a modified test environment.

Prior to running `chroot`, a root directory for the target environment must be prepared. A root directory is simply the top most directory in a hierarchy and can be prepared by sourcing programs and packages necessary for developing `C` programs for `armv7l`. Sourcing these programs can be a matter of approaching maintainers of the packages who may provide the appropriate binaries or building them from source using a cross compiler.

For example, versions of the fedora linux distribution provides these packages via its packages manager and the following command can source the necessary programs for that architecture.

```
dnf install --releasever=36 --installroot=/tmp/f36arm --forcearch=armv7hl
 ↪  --repo=fedora --repo=updates systemd passwd dnf fedora-release
 ↪  vim-minimal m4 cmake gcc-c++ tar gcc git make tmux -y
```

The target directory `/tmp/f36arm/` will then contain a simple root directory for the required development environment for `armv7l`. After configuring the environment in this manner, simply test changing the root directory and running a simple `C` program.

```
chroot /tmp/f36arm /usr/bin/bash
```

Since the `C` compiler for `armv7l` has been acquired for this environment using `dnf` command previously, `qemu-arm` will be able to run the compiler from the bash environment that is using QEMU's user-mode emulation. A simple hello world program as shown in the listing below can then be compiled and then executed in this environment

```c
#include <stdio.h>

int main()
{
        puts("Hello, World!\n");
        return 0;
}
```

Development environments such as these are easy to setup and can prove valuable for rapid prototyping. Another use case may be as part of a continuous integration and continuous delivery mechanism for embedded application development

1.2.4 Targeting Embedded Devices

Building an Embedded Linux kernel suited for a mainboard of an embedded device requires appropriate build configurations describing the kernel, its enabled feature, the device tree layout, i/o memory mapping, etc [2]. These parameters are a description of the devices on the mainboard, their interfaces to the processor, and the nature of the Embedded Linux that is to be managing the hardware platform. The collection of software and configurations required to get an operating system running on a board is referred to as a *Board Support Package* (BSP).

To port Linux onto a processor 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 and handing over the control of the processor. The technical details as to how the boot loader has to be configured will be based on the particulars of the hardware that it will be configured for.

The initial target machine for the project was an ECU filling the role of a communicator on the truck. The BSP source code for the board however was unavailable as well as certain critical support components for the board, such as the vendor's Yocto meta layer, memory mapping for the attached devices, source codes for boot ROM firmware or the boot loader, etc. The reverse engineering efforts to attain this information were dropped due to time constraints and ultimately a similiar board, namely the MCIMX6Q-SDB evaluation board, with the required information publicly provided by processor chip vendor NXP was chosen as the target platform. The details of the attempt at uncovering this information is layed out in Appendix II.

1.3 Neural Network Application Development

The most popular ways to write neural network models are by using machine learning frameworks such as Tensorflow, MXNet, PyTorch, Caffe, etc. all of which have Python as their primary programming language. The Development process for neural network application in industry has several steps from collecting and preparing the data, choosing a network architecture, implementing that model, training and evaluating the model, tuning the hyperparameters of the model, deploying the model to perform inference on new data, and monitoring and improving the model. Several software components, network resources, compute devices, engineerig personnel, etc. has to come together for the effective deployment such an application.

1.3.1 Choice of Programming Language and Machine Learning Framework

As most neural network applications are written in frameworks like PyTorch and Tensorflow, they have thriving ecosystems that provide rich developer support. Most neural network models are trained in a rich compute environments with either dedicated machine learning computer systems or general purpose computer systems with plentiful operational capacities. Machine learning based companies and their service offerings such as cloud machine learning platforms almost invariably target these platforms and provide software tools for developers to utilize. Developers in these platforms enjoy several resources such as productivity tools that allows for continuous integration and development, performance profiling tools, etc.

The programming environment for embedded devices however are not as featureful. Developer resources such as productivity tools for neural network application development and maintainance are lacking and the software stacks that are traditionally used are either too large or unsupported on the broader embedded hardware platforms.

Another aspect to consider is the programming language and software stack used to describe a neural network application. Most machine learning models at present are written in Python and frameworks like PyTorch and Tensorflow have richer interfaces for Python compared to other programming languages. This may be unfavourable to embedded devices where a Python application may take up higher memory and have longer latencies. The programming language of choice for embedded applications is `C` and `C++` which are supported by the ML frameworks but not to the same extent as their Python interfaces.

Machine learning frameworks also utilise multiple software libraries meant for specific aspects of performing machine learning calculations. For instance, a neural network model described in Python using Keras gets converted to a computational graph representation in Tensorflow. Then depending on the model, its invocation, and the compute platform its running on, Tensorflow determines the operations involved, execution order, etc. and execute the computation. At this stage Tensorflow may also use other software libraries such as XNNPACK, or Intel oneAPI Math Kernel Library to perform the calculations.

1.3.2 Neural Network Inference on Embedded Devices

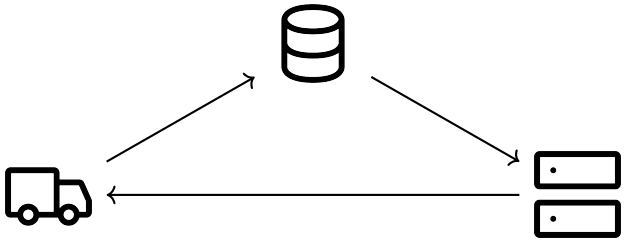
The typical deployment of neural network models on embedded devices follows a pattern of gathering sensor data from the embedded device onto an external data lake, training a neural network model using this data on workstations or cloud platforms, then implementing the neural network model inference on the embedded device. Preferably the implementation utilise math kernel libraries that are made specifically for the embedded platform and the popular machine learning frameworks may provide an avenue to transfer models written in them to target the embedded platform.

The possibilities in making embedded platforms more involved in the neural network development process has been explored in research avenues such as TinyML[18], and other efforts motivated by interests in getting the neural network applications ready for mobile devices such as Tensorflow Lite[6] and PyTorch Mobile [1]

However the primary approach in these cases is with a focus on making the neural network model inference step faster on these embedded platforms.

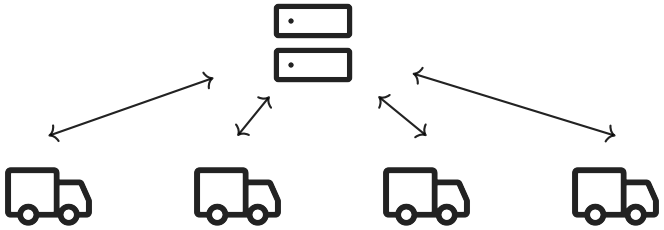
1.4 Federated Learning of Neural Networks

A significant problem in the traditional model for neural network application development in embedded devices is the consumption of network bandwith associated with continual transmission of sensor data to the data lake. The data from different devices are then combined together to form the dataset that will be used to further train the model. However this stream of data coming from the embedded platform exposes the device to computer security risks such as an attacker gaining access to the behavioural data of the vehicle.



One way to address this problem is to rely on alternative mechanisms to perform the continual training of the model in a decentralised manner. Federated learning is a technique of training ML models in such a distributed way, where each client device uses its own data set to train a local model. After this local training session, the new model may be send to a central server which will combine the different models to form a new global model. This may be then sent to the client devices for performing inference.

Federated learning has several different approaches differing in the manner in which distributed training can take place, the algorithm to combine different locally trained models, and other strategies used in completing the development loop. In the LOBSTR [17] and FAMOUS [16] projects Scania has developed statistical models and neural network models for anomaly detection using a



federated learning approach. The statistical models are lightweight and can easily be trained with limited computational resources.

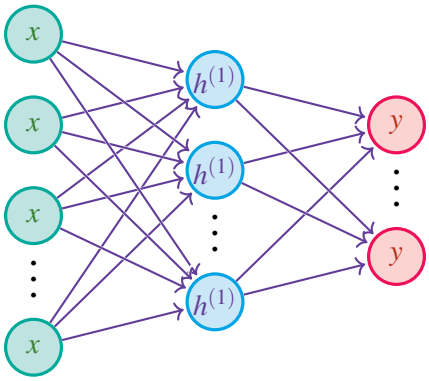
This thesis focuses on repurposing existing hardwared (ECU) to ML edge devices that are tailored to train neural network in the most efficient possible way. We try to reverse engineer the old communicator model to build a custom Yocto project tailored for ML. As alternative test ECU we use an evaluation board that has similar specifications as the commuicator to benchmark and experiment different neural network implementations and evaluate.

2. Theory

The first section in this chapter lays out an overview of the training process of neural networks. The following section introduces some terminology associated with software development for embedded devices, contextualised in Embedded Linux.

2.1 Neural Networks

A neural network consists of a collection nodes called *neurons* that are arranged into several layers with connecting edges that go between the layers. A connecting edge between two neurons describe an operation with the first neuron producing an output that is then consumed as input by the second neuron. The first layer and final layer are special and are called *input layer* and *output layer* respectively. There maybe zero or more layers that lie between them called *hidden layers*.



The connecting edges between the neurons are weighted and additionally a neuron may carry a weight of its own called *bias*. The neurons may have several incoming edges, except for the neurons in the input layer, and several outgoing edges, except for the neurons in the output layer. Each neuron in the network describes a computation in at least two steps, (1) multiply input data with corresponding edge weight and take their sum along with the bias value, (2) transform the value calculated earlier using an activation function.

An activation function σ is said to determine the activation of the neuron which can be thought of as the output that the neuron generates. There are several kinds of activation functions that are used in neural networks such as the sigmoid, ReLU, tanh, etc.

Combining these operations the neuron y_k has the output

$$y_k = \sigma\left(\sum_{j=0}^{m} w_{kj}x_j\right) \tag{2.1}$$

Where y_k is the k^{th} neuron in a layer with input values x_0 through x_m with corresponding weights w_{k0} through w_{km}. The first input x_0 is usually set to 0 and hence the corresponding weight w_k0 stands in for the bias b_k of the neuron. The complete neural network matrix multiplication pass from input to output is called the *feedforward*.

Neural networks can be constructed in a variety of ways with the choice for how many layers to use, the number of neurons in the layers, the connections between the layers, all of which can generate several different topologies. The neural network model can approximate a real world system by modelling that system as function that takes in some input and then generating an output.

The neural network can approximate this function better by changing the connections between neurons, dropping and or adding neurons, varying the weights encoded in the connections, or by varying the biases within the neurons.

2.1.1 Training a Neural Network

One of the most interesting characteristics of a neural network is its capacity to form probability weighted associations between a set of inputs and their correponding outputs. The process of forming this association is called *training* the neural network, the set of input patterns used for this purpose is called a *training set*, and the algorithm by which the network is trained is called the *learning algorithm*. After sufficient training, the network can also produce correct outputs to unseen inputs of the same kind.

The bulk of the mathematical operations involved in training a neural network are simply addition and multiplication instructions of floating point values. Hence a computing platform that is executing a learning algorithm for some neural network is issuing a series of floating point addition and multiplication instructions. Modern computers have optimised hardware features that allow for parallel executions of these multiply and add instructions, all in an effort to improve the training efficiency of neural networks. Google's TPU chip (Figure 2.1)

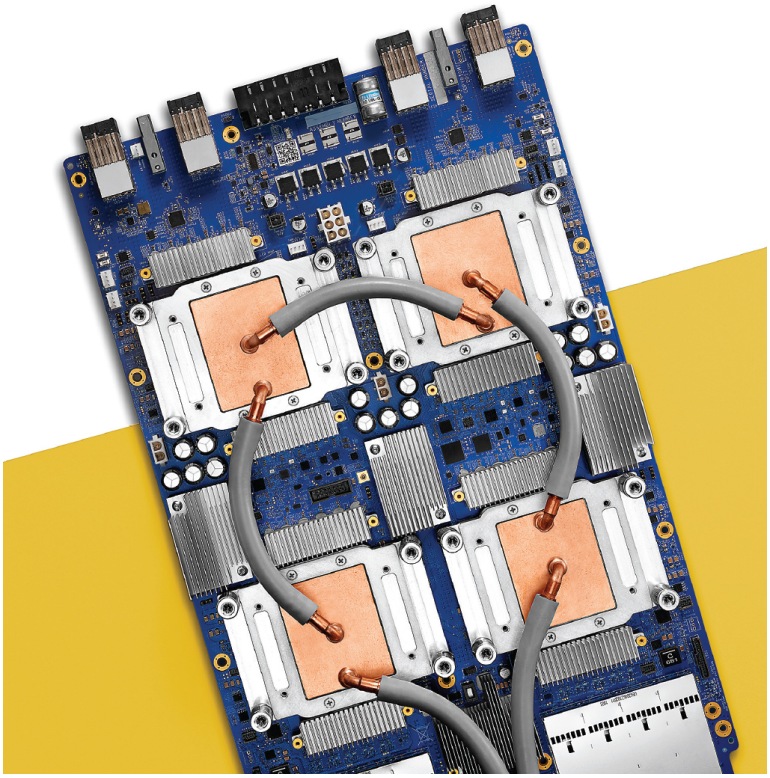


Figure 2.1. Google's TPU Chip

2.1.2 Training vs Inference

Once a neural network completes a training on a training set, it can then be used to look at data points that it has not seen previously. The neural network can be made to perform the feedforward calculations to produce some prediciton or output and this step is said to be a neural network *inference*.

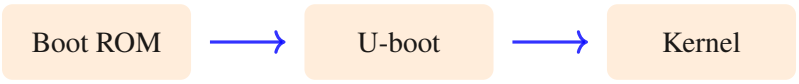
2.2 Embedded Linux

As presented in the preivous chapter, the development, deployment, and maintainance of Embedded Linux distributions and their user mode applications are usually managed using capable build systems. Configuring these systems requires understanding concepts such as boot loaders, device tree layouts, flash memory, cross compiling toolchains, board suport package, etc. with the later two having already been introduced in the preceding chapter.

Porting an Embedded Linux distribution on some embedded hardware completes successfully when the processor on the board is able to run the Linux operating system. Depending on hardware several paths may be taken by the processor to reach this stage after powering on. This process of starting the computer is called *booting* and the sequence of stages the board goes through is called *boot sequence*. Embedded devices are greatly varied and hence there is great variance in how boot sequence take place.

2.2.1 A Simplified Boot Sequence

After power up, the processor requires initialisation which for the kind of processors usually found in ECUs is performed by firmware placed in a special purpose memory called *Boot ROM*. This code is responsible for initialising peripheral devices, hardware busses, CPU registers, etc. and after hardware initialisation locates and loads a bootloader program. Bootloader programs are responsible for continuing the boot process and may have multiple stages with one loading another. The most commonly used open source boot loader for embedded devices is U-Boot. U-Boot will normally be stored in some storage medium that will be accessible by the Boot ROM code, usually some flash memory. Flash memory is a kind of non-volatile memory that can be electrically erased and reprogrammed and is commonly used for storing data and firmware. Flash memory comes in two kinds, NOR flash and NAND flash which have different performance characteristics and usage scenarios.



After getting control of the processor, U-Boot then has to take care of initialising the memory system, finding then loading the Linux kernel into an appropriate location in memory, generate boot parameters for the kernel, and copy other required data for the kernel. The kernel is also commonly stored on a flash memory on board. One of the configuration data that U-Boot has to pass to the kernel is the device tree, which is a data structure describing the hardware layout. Device trees were adopted in Linux and the embedded industry in general to allow mainline Linux and U-Boot to use the device tree to run on a particular board configuration, and to disuade the creation of U-Boot and Linux forks to target marginally different boards [5].

Once U-Boot completes and gives up control over the processor, the kernel then starts with more configuration steps such as configuring the memory, processor, peripherals, cache, and other hardware devices. The kernel then proceeds to complete its start up by setting up Stacks, initialising the Scheduler, setting up and allocating Pages, etc. and completes the start up after having spawned the `init` process, which is the first process to that starts after booting completes.

2.2.2 Porting Embedded Linux

Creating the binaries for an embedded linux and managing configurations for a particular board requires a build system capable of the task. This section takes a look at one such system, namely the Yocto project.

An excursion through Yocto Project's Embedded Linux build system

Embedded linux build systems can become fairly large and complicated software stacks depending on their supported development scenarios, number and nature of the software engineers using

the tool, the build and deploy infrastructure, and the list of supported target platforms. Consider a simple development scenario targetting the popular and open source embedded target platform MCIMX6Q-SDB. The aim is to build and run an embedded linux distribution for the MCIMX6Q-SDB using a personal computer running linux. Note that several components used in this discussion may break their interface so the emphasis will be on the essential concepts of the activites involved.

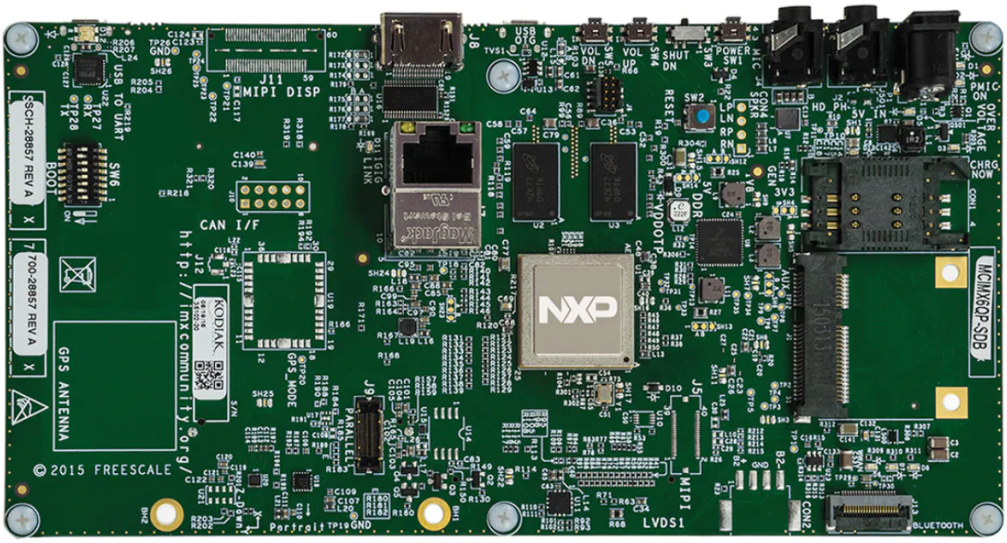


Figure 2.2. MCIMX6Q-SDB-BD

The Yocto Project is arranged into several components out of which the core 3 components are BitBake, OpenEmbedded-Core, and Poky. BitBake is a build engine that interprets configuration files to schedule and then perform tasks. These configuration files are called *recipes* and they describe how to build a particular package such as a shared library or application program. Recipes contain information as to where to obtain the source code for a package, instructions for compiling the source code, and installing or removing that package from a distribution. OpenEmbedded-Core is a set of platform and distribution independent recipes and other metadata while Poky is a reference system containing a collection of projects and tools that can be used to bootstrap a new distribution.

The Yocto Project came out of the work associated with OpenEmbedded build automation framework and shares maintainership of the central parts of the OpenEmbedded build system with the OpenEmbedded project. OpenEmbedded-Core came out of a split of the recipe metadata held within OpenEmbedded. The big picture of the Yocto project can be fairly complex and the amount of terminology and techniques associated with using the project causes the Yocto project to have a steep learning curve. A simplified illustration of the Yocto project is presented in Figure 2.3 below. A set of recipes (teal) sharing a common purpose are arranged into a *layer* (cyan) 2.3. Layers can depend on other layers, with multiple layers usually being used in an embedded linux distribution.

The primary version control system used within the Yocto project is git. The Yocto project has even several software packages requirements for its build system layed out in its documentation (see docs.yoctoproject.org). Furthermore there are minimum version for the required packages for a corresponding version of the Yocto project (see wiki.yoctoproject.org/wiki/Releases).

The following listing presents the typical required packages however, based on the linux distribution of the development host the command will vary.

```
PACKAGE_MANAGER install bc make automake gcc gcc-c++ chrpath cpio diffstat
↪  gawk git python texinfo wget zstd lz4
```

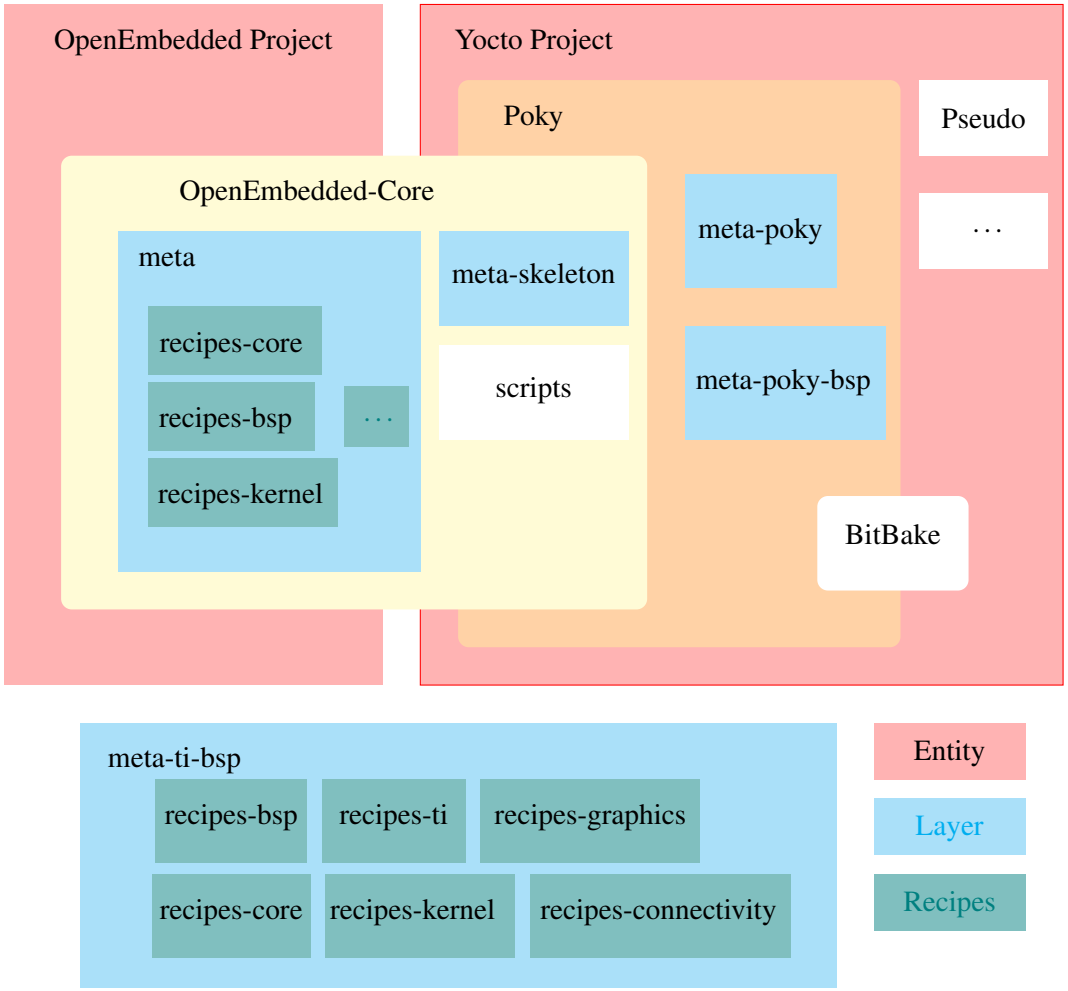


Figure 2.3. Overview of The Yocto Project Components

The above list is not complete and relies on non standard package names. Consulting the documentation above will provide more accurate instructions as to the packages that are necessary to start with the Yocto project.

For this section, consider the `kirkstone` Long Term Support (LTS) release of the Yocto project. To begin clone the Poky repository from git.yoctoproject.com

```
git clone https://git.yoctoproject.org/git/poky
```

The Poky repository contains the OpenEmbedded-Core as well as the BitBake tool that is required for the build. NXP (previously freescale) is the system maker and system vendor for the MCIMX6Q-SDB and they provide a BSP layer in Yocto via the meta-freescale repository.

```
git clone https://git.yoctoproject.org/git/meta-freescale
```

Checkout the kirkstone version of all the different components by checking out the corresponding branches.

```
git checkout kirkstone-4.0.10
```

Once the correct branches has been checked out, the next step is to bring the BitBake tool into the shell environment. The Poky repository contains scripts that facilitate this step. Create

a directory to manage the build configuration and output files and run the setup script inside the Poky repo.

```
source poky/oe-init-build-env $BUILDDIR
```

The script will setup BitBake along with some additional tools and scripts in the current shell environment and also creates a few directories in `$BUILDDIR` along with some configuration files. The first configuration file to edit will be the `$BUILDDIR/conf/bblayers.conf` to configure the build system for the MCIMX6Q-SDB.

```
        MACHINE = "imx6qdlsabresd"
```

After `$MACHINE` is configured correctly, BitBake can create a minimal image that can boot the MCIMX6Q-SDB building the packages for `core-image-minimal` recipe.

```
bitbake core-image-minimal
```

Build images will be generated in `$BUILDDIR/tmp/deploy/images/imx6qdlsabresd`. To boot the MCIMX6Q-SDB with the build image simply flash an SD card with the rootfs image and configure the board to boot from the SD card slot. SD-3 slot may be configured in this manner by toggling the dip switches to D1-OFF D2-ON D3,4,5,6-OFF D7-ON D8-OFF

Connecting to the board via USB-UART via a serial console terminal emulator program should then be possible after power on the MCIMX6Q-SDB.

Part II:
Implementation

The traditional model for deploying neural network applications on embedded devices has over time developed the neural network inference step. The popular frameworks for machine learning such as PyTorch and Tensorflow provide approaches for porting neural network models written using those frameworks with a focus on allowing for model inference on embedded platforms. Under this paradigm, training would be completed on more powerful hardware and the data would have to be collected seperately to allow for the continual training of the neural network model. Targetting even smaller devices with Tensorflow based neural network models is possible for neural network inference applications via Tensorflow Lite Micro [4] further expanding the range of platforms that this paradigm applies. Efforts to allow training as well in these frameworks require more effort due to the compute and memory intensive nature of the training process. Another reason for the delay in adopting training on board is fragmented nature of embedded ecosystem and the difficulties involved in streamlining features for any machine learning framework that would have to support the varied hardware platforms in the ecosystem.

This part of the report contains the description of multiple application programs that train a neural network model for the purpose of benchmarking the training step on an embedded device. The neural network structure, the learning algorithm, and the dataset remain the same across the implementations however they are completed in a traditional general purpose neural network frameworks as well as straightforward implementations in `C`, `C++`, and Python. The design of the neural network is based on a textbook example and is kept configurable to allow for testing across different neural network architectures. Greater detail on the design is presented in the first chapter, followed by another chapter describing the different implementations.

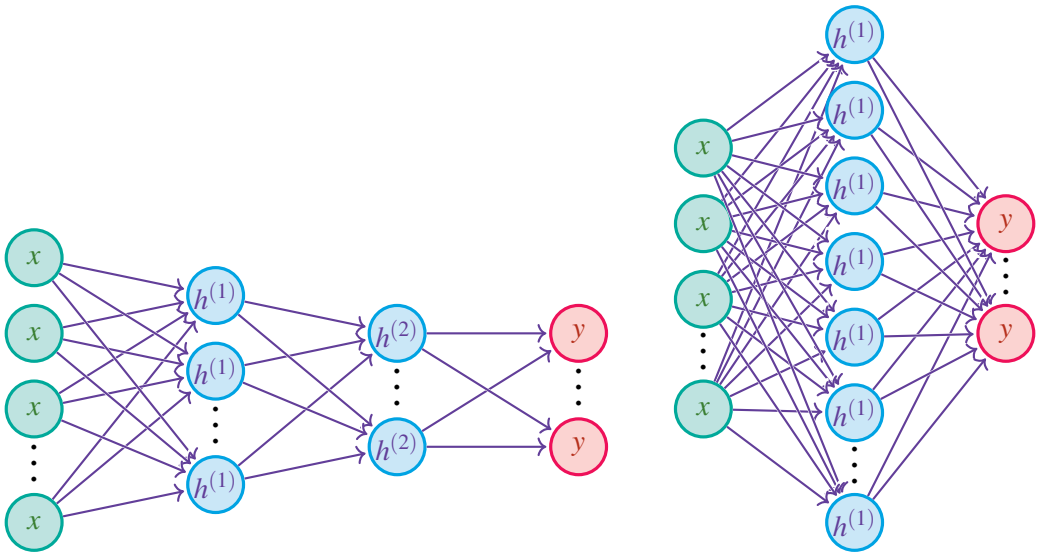
3. Design

The benchmark applications test the training of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [7] 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 for the benchmark experiments.

3.1 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. The network uses the sigmoid activation function and uses a 32 bit floating point representation for its weights and biases.



The benchmark programs would also compute charactersitics of the neural network such as accuracy and are also capable of producing a dump of the network values such as the hidden layer structure, weight and bias values, etc. into a `.nn` binary file format. The `.nn` binary format can then be consumed by any of the implementations to perform an image inference. For instance, the `C` based benchmark application could produce a `.nn` file after training the network and then the Python based benchmark application can then take that file as input to complete a feedforward pass for an MNIST 28 x 28 image in PGM format. The feedforward pass between different implementations of the benchmark applications can be compared in this manner as well.

3.1.1 The Learning Algorithm

The HDR-NN benchmark applications all share the same standard training algorithm listed below (1). Describing this algorithm in general purpose neural network frameworks is straight forward

and plenty of general implementations of the algorithm exists, making the development process easier to target multiple programming paradigms. The configurable parameters of the learning algorithm through out the implementations are the learning rate, the total number of epochs for training, and the batch size for gradient descent iterations.

Algorithm 1 Mini Batch Gradient Descent with learning rate γ and the Mean Squared Error (`MSE`) cost function

Require: initial weights $w^{(0)}$, number of epochs E, batch size B, training data with T entries
Ensure: final weights $w^{(E*T)}$

```
for e = 0 → E − 1 do
    for b = 0 → T/B do
        for t = b ∗ B → (b + 1) ∗ B do
            estimate ∇ℒ(w^(t))                    ▷ ℒ here is MSE
            compute Δw^(b) += −∇ℒ(w^(t))
        end for
        w^(e+1) := w^(e) + γΔw^(e)
    end for
end for
return w^(T)
```

3.1.2 Training Configurations

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.

UX for configuring learning algorithm parameters and other configurations
All benchmark applications upon invocation without any arguments presents a helpful usage string

```
./hdrnn

nothing to perform
Usage: ./hdrnn <command> [<args>]

Commands:
        infer [-i, --image IMAGE_PATH] [-n, --net hdr.nn]
        train [-s, --shape 32] [-e, --epochs 30]
                        [-q, --quiet] [-bs, --batch_size=10]
                        [-lr, --learning_rate 3.0] [-n, --net hdr.nn]
```

Running a training pass of the network can be as simple as

```
./hdr train
```

This execution will run for 30 epochs with a learning rate of 3.0 and batch size of 10 on a neural network with shape [784, 32, 10]. After the training completes, the program will then output a network description file `hdr.nn` with the shape, weights and biases of the network. This file can then be later used to perform an inference only run by running the following command.

```
./hdr infer --image 7.pgm --net hdr.nn
```

The C version of the benchmark application can also produce sketch of the image on to stdout, the shell for example, along with the prediction that the network makes and would produce the following output

```
                                  x x x x x
                                  x x x x x
                            x x x x x x
                            x x x x x x x x x
                            x x x x x x x x x x x
                      x x x x x x x x x x x
                      x x x x x x x x x x x x x
               x x x x x x x x     x x x x x x
               x x x x x x           x x x x x
               x x x x x             x x x x x
               x x x                 x x x x x
               x x x                 x x x x x
               x x x               x x x x x x x
               x x x               x x x x x x
               x x x             x x x x x x
               x x x x x x x x x x x x x x
               x x x x x x x x x x x x x x
               x x x x x x x x x x x x x x
                       x x x x x x x x x
                       x x x x x x x
Network predicts 0
```

To change the network shape and epoch parameters, for example, run:

```
./hdr train --shape 16,16 --epoch 4 -q
```

The program will then train a new network of shape [784, 16, 16, 10] for 4 epochs and silently exit without producing the `hdr.nn` file as the -q quiet flag was present. The complete list of program parameters are given below.

Silent invocation

Use silent invocation using the `-quiet` or `-q` flag

When used with the 'train' command, the final weights and biases files won't be generated

```
./hdr train --quiet
```

Epochs

The number of epochs to train the network, specified by `-epochs` or `-e` followed by a natural number

```
./hdr train --epochs 21
```

Network Shape

The size and shape of the HDR-NN network as comma seperated numbers after the `-shape` or `-s` flag

```
./hdr train --shape 16,16
```

Learning Rate

The learning rate for the Stochastic Gradient Descent as a floating point value after the `-lr` or `-learning_rate` flag

```
./hdr train --learning_rate 3.1
```

Batch Size

The batch size while performing mSGD as an integer after the `-batch_size` or `-bs` flag

```
./hdr train --batch_size 64
```

4. Development

The HDR-NN benchmark applications were completed in different programming languages and in PyTorch. Details about the target environment and the benchmark implementations are layed out in this chapter

4.1 Targeting MCIMX6Q-SDB

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.

4.1.1 Compiler Toolchains & 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.

4.1.2 Building PyTorch for armv7hl

PyTorch project provides LibTorch as a binary distribution of all the headers, libraries, CMake configurations required to use PyTorch. However the PyTorch project does not provide these binaries for MCIMX6Q-SDB. The source code could however, with some effort, be used to generate these binaries and for this project a QEMU based user-mode emulation environment was used for native compilation of the libtorch binaries.

4.1.3 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 Implementations

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. The following listing shows the function that performs feedforward pass in the Network.

```
class Network(object):

def __init__(self, sizes):
        """Initialise neural network"""
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
        for x, y in zip(sizes[:-1], sizes[1:])]

def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
                a = sigmoid(np.dot(w, a)+b)
        return a
```

4.2.2 PyTorch based HDR-NN

Developing ANNs using PyTorch is straightforward with good support and well documented APIs. There were two primary choices for programming language to write the neural network in PyTorch, namely Python and C++. PyTorch project does not provide binaries for either language choice for `armv7hl` however with the source code of the project being available, the libtorch binaries were generated as mention in the previous section.

The implementation used the MNIST made available by Torchvision library which is mainted under the PyTorch project and configured a Module to implement the same learning algorithm as that outlined in the Numpy based Python implementation.

```
struct Net : torch::nn::Module
{
        Net(std::vector<unsigned int> shape)
        {
                // size of image in row vector form
                unsigned int previous_dim = IMAGE_SIZE;
                unsigned int index = 1;
                torch::nn::Linear fc{nullptr};

                // iterate over the dimensions required in the hidden
                ↪  layer
                for (auto c_dim : shape)
                {
                        // Construct and register Linear submodules.
                        fc = register_module(
                                "fc" + std::to_string(index),
                                torch::nn::Linear(previous_dim, c_dim));
                        previous_dim = c_dim;
                        index++;
                        fc_list.push_back(fc);
                }
```

```
                // finally, add the last layer
                fc = register_module(
                        "fc" + std::to_string(index),
                        torch::nn::Linear(previous_dim, DIGITS));
                fc_list.push_back(fc);
        }

        // Implement the Net's algorithm.
        torch::Tensor forward(torch::Tensor x)
        {
                // Use one of many tensor manipulation functions.
                x = x.reshape({x.size(0), 784});
                for (auto fc : fc_list)
                        x = torch::sigmoid(fc->forward(x));

                return x;
        }

        // Use one of many "standard library" modules.
        std::vector<torch::nn::Linear> fc_list{};
};
```

4.2.3 C based HDR-NN

The `C` implementation had the least amount of external dependencies, other than the `C` standard library. The data structures of the neural network contain float arrays within structs as shown in the listing below. The learning algorithm was implemented to remain identical with those used in the other implementations.

```
/* HDR Neural Network */
typedef struct
{
        float bias;
        float *weights;
        float *nabla_w;
} Neuron;

typedef struct LayerT
{
        int size;
        int incidents;
        Neuron *neurons;
        float *activations;
        float *z_values;
        float *nabla_b;
        struct LayerT *next;
        struct LayerT *previous;
} Layer; // Network layers except for input

typedef struct
```

```
{
        Layer *layers;
        int depth;
} Network; // HDRNN
```

4.2.4 C++ based HDR-NN

The C++ implementation used the n-dimensional arrays of popular C++ linear algebra library, Eigen. It is a straight forward port of the same structure and learning algorithm as the Numpy based Python implementation. A portion of the learning algorithm is shown in the listing below.

Compared to the C version, the C++ version uses the richer language feature set of C++ such as namespaces, classes, etc. The C++ implementation uses the popular build automation tool CMake instead of the GNU autotools setup adopted by the C implementation.

```
void mini_batch_sgd()
{
        // Initialize Nabla matrixes
        std::vector<nabla> nablas;
        for (std::size_t i = 0; i < network.size(); i++)
                nablas.push_back(
                                        nabla(network[i].weights.rows(),
                                                network[i].weights.cols())
                                        );

        // Go through the training data by batches
        for (std::size_t i = 0; i < mnist_loader::train.size()
                                ; i += BATCH_SIZE)
        {
                // Perform Backpropagation on the batch
                for (std::size_t j = 0; j < BATCH_SIZE; j++)
                        back_propogate(nablas,
                                        mnist_loader::train[i+j].data,
                                        mnist_loader::train[i+j].label);

                // Update the weights and biases of the network
                for (std::size_t j = 0; j < network.size(); j++)
                        network[j].update(nablas[j]);

                // Zero out the nabla matrixes
                for (std::size_t j = 0; j < nablas.size(); j++)
                        nablas[j].zero_out();
        }
}
```

Part III:
Analysis

A hand digit recognition neural network (HDR-NN) model is implementated in `C`, `C++` using Eigen, Python using Numpy, and Pytorch via its `C++` interface. The performance of HDR-NN training implementations was evaluated on the MCIMX6Q-SDB 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 neural network model and compiling it to operate on the target hardware.

Benchmark Overview

HDR-NN is implemented in different paradigms, specifically `C`, `C++`, Python, and Pytorch. Each of these applications contain 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 which correspond to the 28 x 28 pixel dimensions of the images of the MNIST dataset. The output size of the network is 10, corresponding to the 10 possible digits that the image contains. The number and size of the hidden layers are configurable in each of the different implementations.

The MNIST dataset was selected to train the model, which 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 for the benchmark runs are set to 1 epoch with a batch size of 10, a learning rate of 3, with the network using the sigmoid activation function.

It is essential that the hardware utilzed for benchmarking closely resemble the i.MX6 processor on Scania ECUs, as this will make it easier to replicate the experiment on a repurposed ECU and will also provide the most precise results. The MCIMX6Q-SDB 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 i.MX6 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 [13] 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 $\sim$ 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 neural network 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.

A python script was developed to run 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.

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 additional information on other resource usage such as the 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

The benchmark applications all had the same configurable parameters for their learning algorithm and network structure. Initial testing of the different benchmark applications were completed seperately with different configurations. These test runs were used to come up with estimates as to how long each run of the training sequence with the different parameters would take and then used to come up with the network shape sizes, and other learning algorithm parameters.

Initially, the network's single hidden layer shape was varied according to powers of two, with the `C` and `C++` variants tested with 2, 4, 8, 32, 128, 256, 512, 1024 and so on before adding another hidden layer. The additional hidden layer varied sizes from (16, 16), to (32, 16) and (16, 32), then (128, 16), etc. The final list of shapes and their corresponding shapes for which the measurements results are state in the next chapter are shown in the following table.

Hidden layer shape	**HDR-NN parameters** (total)
2	1600
4	3190
8	6370
16,16	13002
32	25450
48,48	40522
64,16	51450
72	57250
82,36,16	68120
96,96	85642
104	82690
114	90640
128	101770

The section on UX in the Design chapter describes how the shape and epochs parameters were configured.

5.2 Compilation Options

All the compiled benchmark programs used the GCC compiler and its `-O2` optimisation flag. Several other optimisation possibilities were possible and tried out however the data present for `-O2` was fixed for all the applications for uniformity.

The general suite of compiler optimisations from GCC 11.3 are clever enough to use SIMD, inlining, strength reduction, and a suite of powerful optimisations.

5.3 Profiling Analysis

A `perf_events` based record was conducted using the `perf` utility for all the programs. Flame graphs were generated for the same as shown attached below revealing the areas where the applications were spending the most time on.

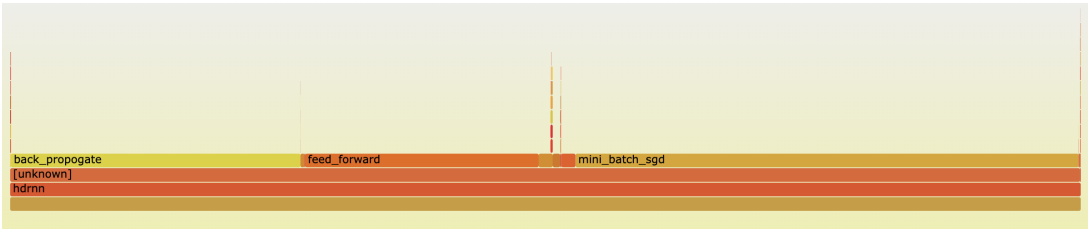


Figure 5.1. c-math.h

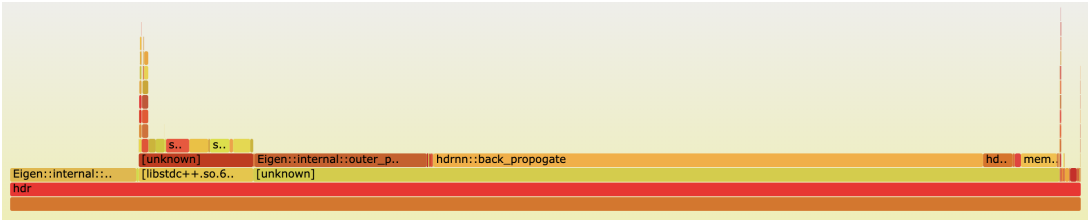


Figure 5.2. cpp-eigen

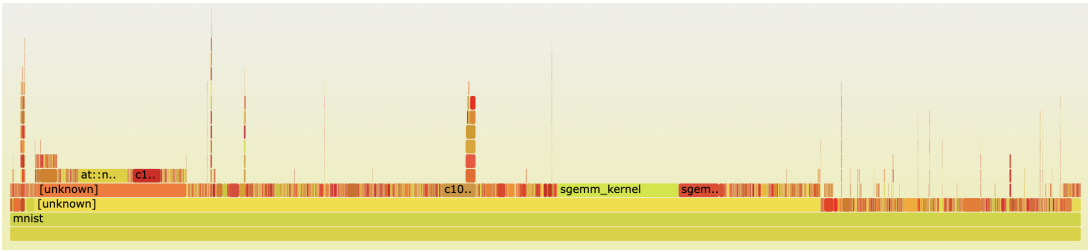


Figure 5.3. cpp-libtorch

5.3.1 Memory Profile

Heaptrack is an alternative to the popular valgrind programming tool for memory profiling that comes under the KDE gear of applications by the KDE community. Heaptrack was used to profile the memory of the benchmark application programs under execution

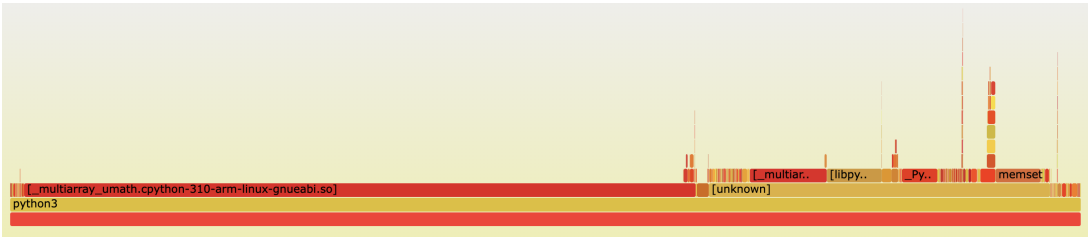


Figure 5.4. python-numpy

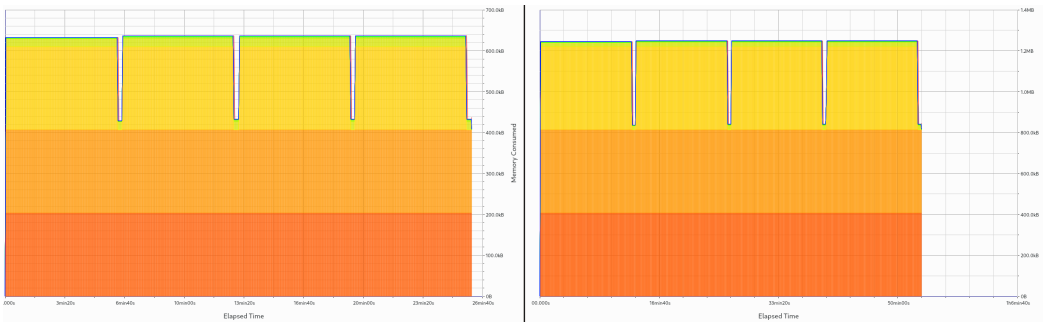


Figure 5.5. c-math.h with 2 different shapes

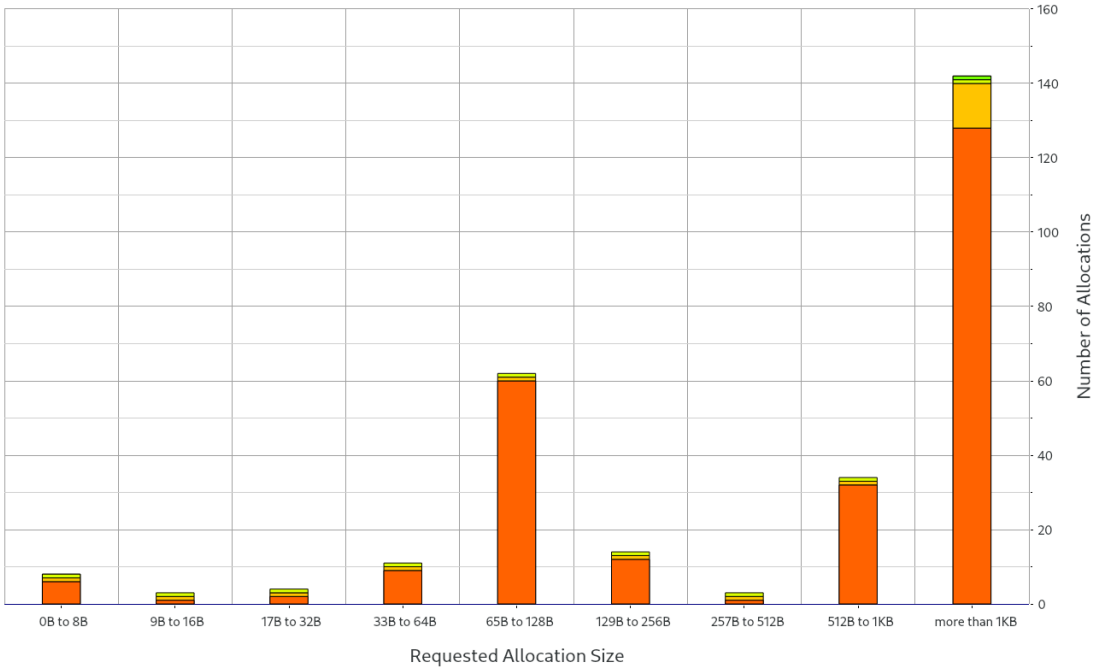


Figure 5.6. c-math.h allocations

6. Results

This chapter presents the results of the benchmark application runs. The first section contains a brief look at the analysis on the neural network accuracies for the different implementations. The second section considers the performance of the benchmark applications followed by the third section that details their differences.

6.1 Evaluating Correctness

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

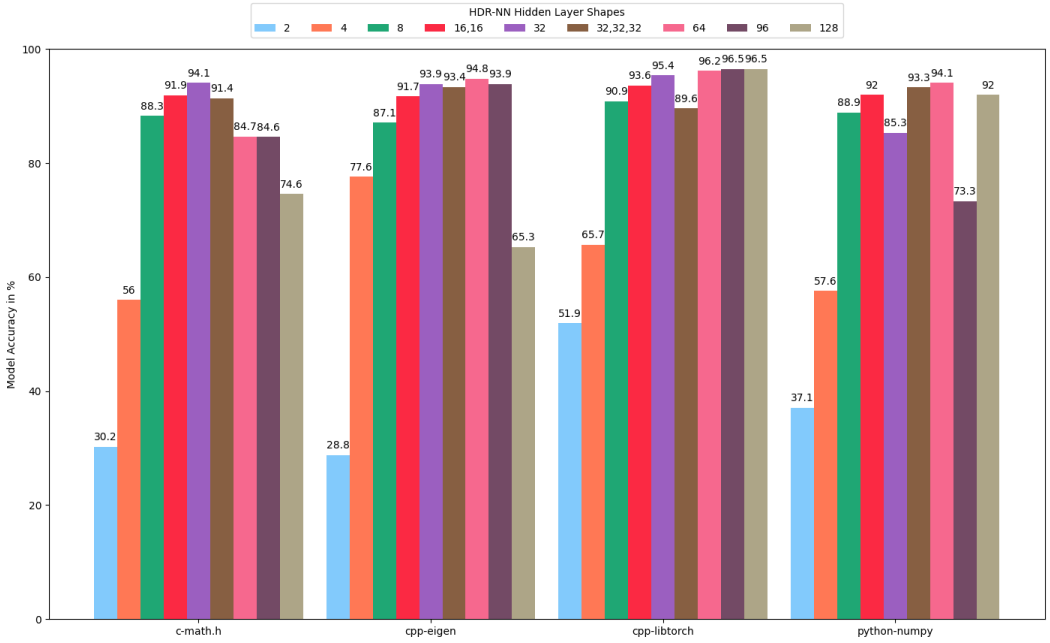


Figure 6.1. Comparing the accuracy of the different HDR-NN implementations.

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).

6.2 Evaluating effectiveness

The primary measures used to evaluate the program performance was execution time and memory utlisation while the applications completed their neural network training.

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.

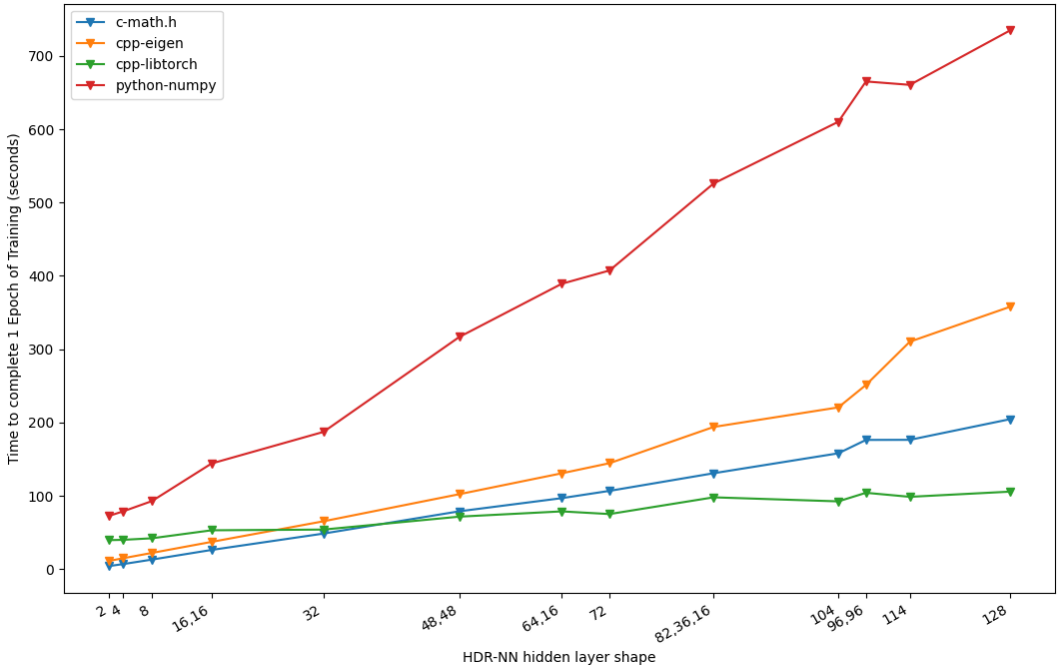


Figure 6.2. Comparing total run time for training the different HDR-NN programs

6.2.2 Peak Memory Usage

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

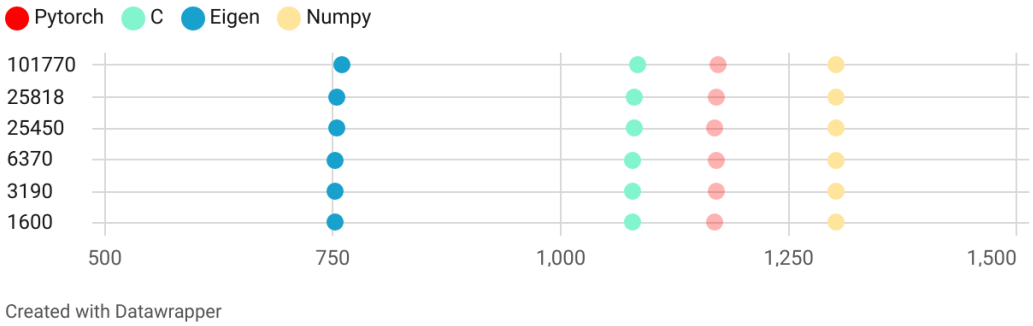


Figure 6.3. Peak Memory Utilized during training with different model sizes remain similar within the same implementation

Note that the device RAM is 1024 MB however the peak memory utilisation for both `C` and Numpy are higher than this value. This can be explained by over allocation of memory by the operating system utilising the swap space. The peak memory utilisation measure is using the Maximum resident set size measure, which is roughly the total amount of physical memory assigned to a process at a given point in time. It does not count pages that have been swapped out, or that are mapped from a file but not currently loaded into physical memory.

6.2.3 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 and the different network model shapes within the same implementations.

6.3 Comparing the implementations

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

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.

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.

7. Discussion

This chapter contains discussions on the experience while working on the project with the first section examining the process of developing the benchmark applications. The last section describes in brief the distribution of work on the project between the two authors.

7.1 Developer Experience

Embedded environments are greatly varied and working on these platforms are different compared to the more general platforms with comparably reduced support. The greatest challenge in writing the benchmark applications were in sourcing the binaries for the general purpose neural network frameworks. Both Tensorflow and PyTorch do not target the ARMv7 environment that was the primary target for this project. There were are older community tensorflow binaries that are available for the platform that are currently unmaintained, at the time of writing this report. Finally the PyTorch source code was successfully compiled for our target environment using QEMU user-mode emulation.

The embedded hardware and software ecosystems are greatly varied and fragmented leading to less support from the machine learning software ecosystem. The embedded machine learning communities largely contain sprawling technologies that have several different components that have varying degrees of maintanance and create complicated inter relationships over time that are difficult or unwieldy to maintain. This has lead to a lack of good software infrastructure that can support multi-hardware, mult-architecture neural network frameworks.

7.1.1 Reverse Engineering Scania C300

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.

Appendix II contains a summary of the efforts involved in reverse engineering the C300 hardware. The project focus shifted to the MCIMX6Q-SDB board after having spend considerable time on the C300.

7.2 General Distribution of Work

The reverse engineering efforts were done in unison between the authors by suggesting then attempting different ideas, with Deepak Venkataram working on soldering and other physical manipulations on the Scania C300. The programming of the benchmark applications was completed by Prasanth Thomas Shaji while the testing of these applications on the MCIMX6Q-SDB were completed by Deepak Venkataram. The primary responsibilities of most activities were divided between the authors however they were performed in concert were possible. The benchmark applications and the report were version controlled in two seperate repositories containing commits from both authors and provides the primary accounting of the distribution of work.

8. Conclusion and Future Work

Developing of neural network implementations for the embedded environment is considerably more challenging than a traditional general purpose personal computer or server computers. The lack of support of on these platforms for the traditional machine learning frameworks stems from the fragmented nature of the embedded ecosystem. There are several challenges in the way for developing a machine learning framework for embedded devices. The current mature frameworks have some means of providing inference passes on embedded platforms however training on-board is still limited to larger platforms.

The development and testing of the benchmark application HDR-NN revealed the difficulties in targetting the `armv7hl` architecture. The performance of the benchmark applications showed that PyTorch had considerably higher performance than expected for larger shapes than the hand made `C` implementation. This could however change by working on performance engineering of the `C` implementations. However leveraging tool support for performance engineering on these systems are still difficult and requires heavy investment in time.

8.1 Future Work

The initial mandate on the project was to repurpose the Scania C300 to provide a means of targetting the platform reliably. There are yet more technical hurdles to achieving that goal. On-Board Training of neural network models is an active area of reach in TinyML with several efforts in both academia and industry to achieve efficient training algorithms.

8.1.1 Porting to C300

Repurposing the Scania ECU is a technical challenge that was left incomplete during the project. The experience as detailed in Appendix II and through out the report. The attempt concluded without having ported the embedded linux onto the board. Further information on the device tree layout may be extracted by leveraging the `/proc/device-tree` linux interface and a binary disassembly of the bootloader on-board the device could reveal more information about the memory layout. These efforts were dropped in favour of continuing the work on the MCIMX6Q-SDB.

8.1.2 TinyML research on On-Device Training

Multi-framework and multi-architecture based software infrastructure that allows are being developed my the community in parts such as Modular [11], EdgeImpulse, etc. There are innovative approaches to bring down the memory requirements of deep learning neural network models such as TinyTL [3] by changing the learning algorithm that will be running on-board.

References

[1] Meta AI. Pytorch mobile. https://pytorch.org/mobile/home/. [Online; Accessed on June 14, 2023].

[2] Alexandre Belloni (Bootlin). Porting linux on an arm board. https://bootlin.com/pub/conferences/2015/captronic/captronic-porting-linux-on-arm.pdf. [Online Accessed on June 14, 2023].

[3] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce activations, not trainable parameters for efficient on-device learning, 2021.

[4] 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.

[5] David Gibson and Benjamin Herrenschmidt. Device trees everywhere. *OzLabs, IBM Linux Technology Center*, 2006.

[6] Google. Tensorflow lite. https://www.tensorflow.org/lite. [Online; Accessed on June 14, 2023].

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

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

[9] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory, 2022.

[10] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data, 20–22 Apr 2017.

[11] Mojo. *i.MX* yocto project user's guide. https://www.nxp.com/docs/en/user-guide/IMX_YOCTO_PROJECT_USERS_GUIDE.pdf, 2023. [Online Accessed on June 14, 2023].

[12] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2018. [Online Accessed on June 14, 2023].

[13] NXP. *i.MX* yocto project user's guide. https://www.nxp.com/docs/en/user-guide/IMX_YOCTO_PROJECT_USERS_GUIDE.pdf, 2023. [Online Accessed on June 14, 2023].

[14] Otavio Salvador and Daiane Angolini. *Embedded Linux Development using Yocto Projects: Learn to leverage the power of Yocto Project to build efficient Linux-based products*. Packt Publishing Ltd, 2017.

[15] 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 June 14, 2023].

[16] 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 June 14, 2023].

[17] 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 June 14, 2023].

[18] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

Appendixes

Scania C300 Communicator

Scania ECU was set to be the target hardware to benchmark the training of a machine learning model. The ECU is developed by an external system maker Actia who designed the board in association with Scania and with an i.MX6 series processor and a number of on-board peripherals including CAN controllers. Much of the build system configuration files and Yocto BSP layers used for the board are not made available by the system maker, instead providing embedded linux binaries and SDKs to Scania. The initial mandate included creating such a BSP layer for the C300 which would facilitate the repurposing of the hardware on board. For this purpose, some examinations on the board were conducted and this part of the report will account the activites made in this effort.

Development Tools

The U-Boot bootloader on a vanilla C300 is silent. The documentation contained little information on the device tree and no memory layout information. The boot flow information of the processor was available from NXP but no information was available on any modifications made by the system vendor. The lack of these resources made it difficult to port or install any tool/package on the device to reveal certain basic information. The information that was available about the processor were the number of cores, the instruction set architecture, the supported memory units, and the basic kernel information from the name, version, and distribution. There was visibility into the file system however, and to the presence of the bootloader code as well

Approach

The first naive approach taken was of flashing the MTD partitions that houses the bootloader. This was to verify that it is possible to flash the same bootloader and to then 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 with the little device information that was available and sample

by examing the board. However the broad was bricked, pressumably due to the u-boot image having incomplete or wrong device tree structure or perhaps the loading kernel failed as version mismatch between bootloader and kernel, or a checksum failure, or perhaps the size of the file was big enough to overwrite a different crucial region.

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