Prasanth Shaji, Deepak Venkataram

Training Neural Networks on Embedded Devices

Neural Network Frameworks vs Systems Programming Languagess

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 devices 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 its fleet annually [19]. 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 maintenance. For example, if a tire change is predicted to be required in 100 km then the driver can plan the route smartly to reach the workshop before the vehicle breaks down. This opportunity can be realized 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 [21] and FAMOUS [20] 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 behaviour 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 machine learning 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 leveraging 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 bandwidth 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 full potential of implementing neural networks application on embedded devices has still not been realized. Approaches such as TensorFlow Lite, Edge Impulse, and STM Cube AI implemented along with other TinyML frameworks, enable running machine learning models targeted for small resource devices. However, these approaches are largely limited to inference capabilities. Approaches such as the Tiny Training Engine could potentially unlock the path to training neural network models on embedded devices.
- 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 is restricted. To construct an embedded operating system for a customized hardware, critical details such as the device tree, memory organization, 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 devices. The second part details the benchmark applications that were implemented to evaluate the processes of training neural networks on a specific target device. 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 learning 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 embedded device that is meant to run the neural network 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 [10]. The benchmark applications of interest in this report focuses on training neural network models on environments that use the Embedded Linux operating system.

This chapter introduces several terms and techniques associated with embedded application development and starts with a small section on ARM based embedded boards and hardware connection interfaces. The next section gives an overview of the terminologies and processes associated with Embedded Linux and application development for Embedded Linux boards. The subsequent section introduces a discussion on neural network application development for embedded devices. The final section on federated learning motivates the efforts to study training of the neural network on embedded devices contrasting further with the conventional model of training neural network models for embedded devices.

1.1 Embedded Devices

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. Depending on the application domain, embedded devices are generally designed in concert with multiple entities that are each responsible for a different component in the hardware-software stack.

Consider an example of such an arrangement accounted in this paragraph. 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, Texas Instruments, etc. buy CPU core designs from ARM and create application processors along with a suite of electrical components in an integrated circuit referred to as a *system on a chip* or SoC. Ultimately, a *system maker* would select among these application processors, sensors, and other peripheral devices to design an embedded *board* for a particular application domain. There are different practises that the industry adopts to create embedded devices for different applications and the example presented here is one among several. The embedded devices considered in this project derive from such a process.

The configuration and functionality of peripheral devices and how they can interact with the processor on the SoC are decided by the silicon vendor. There are several architectural factors that

go into this decision such as the computer organization, memory addressing model, supported instruction set architectures, etc. These decision in turn necessitates controller circuits such as for memory access, interrupt control, debug interfaces, etc. Another important component are controllers that enable communication to peripheral devices through standard serial communication buses such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Universal Serial Bus (USB), Universal Asynchronous Receiver-Transmitter (UART) etc. More information on these protocols and their corresponding hardware connectors will be given in the next part of this section. The sum of these decision factors determine the hardware software interface of the SoC. A simplified diagram of an ARM based embedded board is presented in Figure 1.1

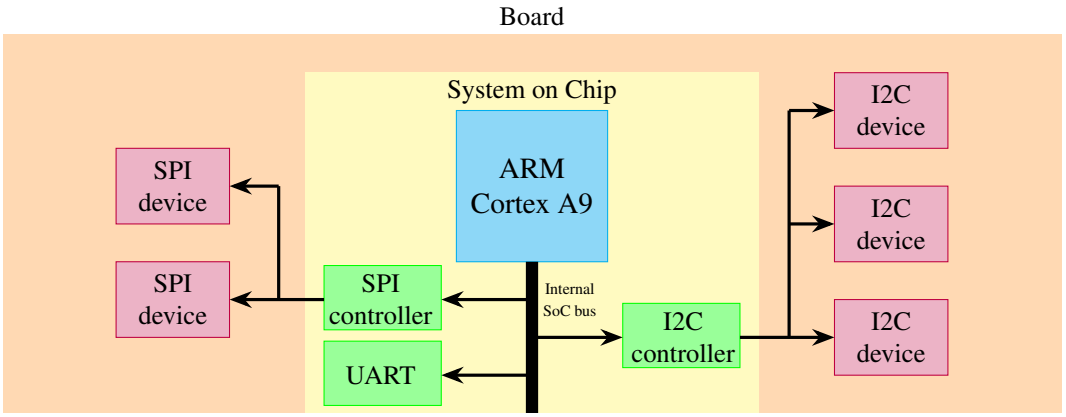


Figure 1.1. Simplified schematic view of an ARM Device

1.1.1 A Brief Overview of ARM Processor Families

ARM processor cores are ubiquitous in embedded systems and come in 3 primary flavours for their use in embedded devices. The architecture implemented by these different categories of processors also differ in their nature and complexity. The Cortex A series of processors implement the ARM Application profile (A-profile) architecture and are general purpose, high performance cores with capabilities such as virtual memory addressing, support for multiple processing units, secure boot through hardware enforced isolation, and more. The memory management unit present in these cores makes it easier to run modern operating systems on them. These embedded processor cores are primarily used in smartphones, laptops, network appliances, set-top boxes, medical devices, etc. The Cortex R series of processors implement the ARM Real-time profile (R-profile) architecture with its tightly coupled memory and fault tolerance features target real time applications such as solid state device controllers, media players, airbags, braking systems, etc. Cortex R processors do not have memory management units and instead implement a protected memory system architecture due to their requirement to support hard real time applications. A suitable modified Linux can be made to run on Cortex R, however the cores more commonly runs ARM's real time operating system Mbed OS. Lastly, the Cortex M series of processor cores implement ARM Microcontroller profile (M-profile) and are low-cost, energy-efficient cores targeting microcontroller applications. Cortex M cores are notably simpler compared to Cortex A or Cortex R and commonly run bare metal programs as well as Mbed OS.

The popular open source educational single board computer BeagleBone Black (Figure 1.2) has an ARM Cortex A8 based Texas Instruments AM3359 processor core. Table 1.1 lists further examples of different ARM based SoCs, their Application Domain, Processor Cores, and specific Architecture Profiles. Embedded Linux is commonly found on Cortex A based processor series such as the one present in the BeagleBone.

SoC	Application Domain	Architecture	Processor Core
nRF51822	Ultra low power, BLE	ARM v6-M	ARM Cortex M0
AM2732	Automotive	ARM v7-R	ARM Cortex R5
AM3358	Industrial / IoT	ARM v7-A	ARM Cortex A8
i.MX6S	Multimedia applications	ARM v7-A	ARM Cortex A9
Kryo 240	Smartphones	ARM v8-A	ARM Cortex-A73, A53

Table 1.1. A few ARM application processors with their domains, cores, and architecture

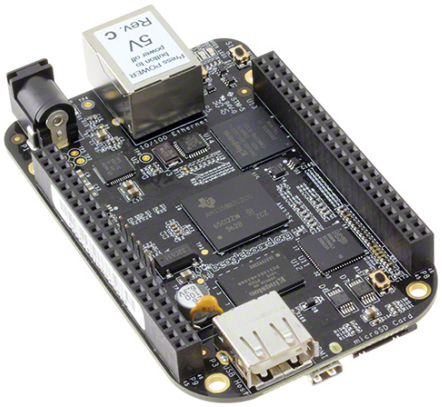


Figure 1.2. BeagleBone Black

1.1.2 Hardware Connection Interfaces for Embedded Devices

Embedded devices rely on a diverse array of hardware connection interfaces to facilitate communicating with other devices, debugging, or for user interaction. These interfaces enable devices to exchange data, configuration settings, and control signals. The connectors play a critical role in the development, testing, and operation of the embedded hardware device. Here, we explore some essential and common hardware connection interfaces for embedded devices.

General Purpose Input/Output (GPIO) pins

GPIO pins are versatile and fundamental components of embedded devices. They allow digital signals to be both input and output, enabling devices to interface with the external world. GPIO pins can be used for tasks such as reading sensors, controlling actuators, and communicating with other devices using protocols like I2C or SPI.

SPI and I2C are serial communication protocols that enable devices to exchange data with peripherals such as sensors, displays, and memory. SPI utilizes multiple data lines for high-speed, full-duplex communication. I2C, on the other hand, employs two wires and supports multiple devices on a shared bus. Both protocols offer efficient communication and are well-suited for connecting embedded systems to various external components.

UART (Universal Asynchronous Receiver-Transmitter)

UART is a simple and widely adopted asynchronous serial communication interface that facilitates the transmission of data between devices. It employs asynchronous communication, with data being sent as a stream of bits along with start and stop bits. The precise timing of the communication can be controlled by the communication channel such as USB. UART connections are commonly used for console output, firmware updates, and device configuration. They provide a straightforward method for devices to communicate over short distances.

JTAG (Joint Test Action Group) Debug Ports
JTAG is a standardized interface used for debugging and testing embedded systems. It allows developers to access and manipulate internal components of a device, including CPUs, memory, and other integrated circuits. JTAG enables operations like boundary scanning, which aids in identifying manufacturing defects, as well as in-circuit debugging to inspect and modify a device's state during runtime. Developer tools meant to configure and test an embedded board usually make heavy use of this interface.

Ethernet and Wi-Fi
For network connectivity, embedded devices can integrate Ethernet ports or Wi-Fi modules. Ethernet enables wired communication and is suitable for applications requiring high data rates and low latency. Wi-Fi, on the other hand, provides wireless connectivity, allowing devices to communicate over local or wide area networks. Both options enable embedded devices to connect to the internet, exchange data with remote servers, etc.

There are even more interfaces commonly found within embedded devices such as the USB, or the Controller Area Network (CAN) communication protocol which is commonly used in automotive applications. These hardware connection interfaces are an important part of interfacing with embedded devices. They enable integrations with other sensors and devices, communication with the outside world, and control of the system. Each interface caters to specific use cases, contributing to the versatility and functionality of embedded systems across industries and applications.

The upcoming sections explore the software development process for embedded devices that use the Embedded Linux operating system.

1.2 Embedded Linux and Application Development

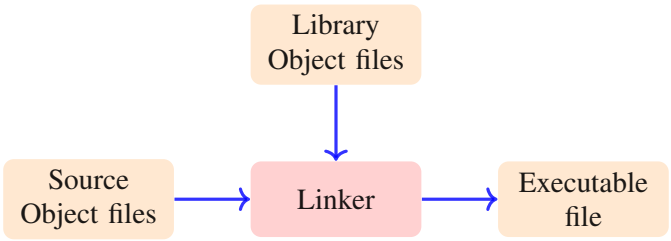
Creating application programs and building a particular distribution of an Embedded Linux kernel requires a suite of software programs. This section presents an overview of aspects of this software stack, focusing on the categories of critical tools that facilitate development in the domain of Embedded Linux. Communicating with an embedded system requires hardware specific connectors and matching software tools. Additionally, the concept of the cross compiling toolchain will be explored, highlighting their significance in enabling developers to build software for hardware architectures different from the development machine. After that, the section explores a software system that is used to generate a bootable executable file for an Embedded Linux kernel. The concluding part of the section demonstrate the use of QEMU emulator as a means of embedded application development.

1.2.1 Software Development Kits

Creating applications for embedded devices requires a set of software components that are usually collectively referred to as a *Software Development Kit* (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 an embedded device. A simplified account for that process now follows.

`C` and `C++` programming languages are implemented in such a way that source code written in either language has to be first read by a *compiler* program, which will then generate machine code in the form of an *object file*. The primary choice for a `C` compiler in the embedded domain is the GNU Compiler Collection `gcc`, with LLVM's `clang` being a close alternative.

The object files of a program generated via the compiler together with their required libraries, which are usually themselves object files of some kind, will then be combined by another program referred to as a *linker*. The linker merges the object files taking care to identify their interdependencies to generate another object file, or possible an *executable file*. The GNU project linker program `ld` is a popular choice used in the embedded domain, while for LLVM project has its own replacement utility called `LLD`.



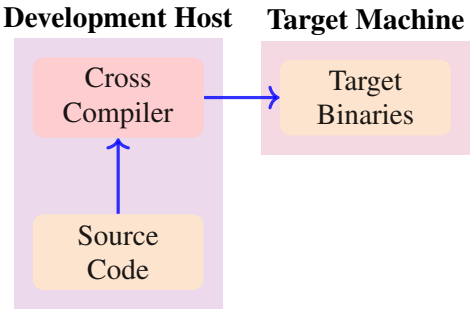
Finally, a *loader* program then takes this executable file and places it in memory such that it can begin its execution. A similar program to a compiler is the *assembler* which takes an assembly file as input before generating an object file. Another useful program in this context is the *debugger*, which is a tool used to test a program. Again, the GNU project's debugger is the `GDB` with the LLVM project alternative being `LLDB`.

The toolchains used for software development consists of a compiler, linker, libraries, and debuggers. Additionally, the toolchain will have a collection of programs to create and manage executable binary programs. An example for the same is the commonly used GNU binary utilities, a.k.a. binutils. To develop applications that interface with the Linux operating system APIs, a toolchain targeting a Linux operating system will also contain the necessary header files called *Linux kernel header files*. These header files encode the user level Linux API that user-level applications can use to interact with and utilize the different features of the Linux kernel. As an example, System level utility software programs make heavy use of this interface to provide their functionality.

The final important piece of a toolchain will be the `C` runtime library, which is the primary component that enables the execution of `C` and `C++` programs. It provides essential functionality in the form of functions and routines to interact with the operating system, manage memory, and perform various Input Output (I/O) operations. The most popular choice of a `C` runtime library is GNU's `glibc`. Alternatives include `musl libc` and `uClibc`.

1.2.2 Cross Compiling

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 device that the one it is currently running on and is termed a *cross compiling* toolchain. A compiler toolchain that creates executables for the sample device 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 devices, etc. and they are ultimately an unavoidable part of creating programs for a new hardware device. Most software that are run on embedded devices are created on a different computing device. Such a computing device in the context of cross compilation is referred to as a *development host* and the embedded device that the software ultimately runs on is called the *target*.



A short introduction to 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. Another naming convention used for the `<arch>` string when referring to ARM architectures is the use of strings such as `armv7` to denote the ARMv7 architecture found in ARM Cortex A (ARMv7-A), ARM Cortex R (ARMv7-R), and ARM Cortex M (ARMv7-M) cores.

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 A, R, and M profiles of the ARM architectures.

1.2.3 Embedded Linux Build Systems

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* libc, 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 feature full 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 [17] making it ideal for managing Embedded Linux builds for diverse hardware devices. The Yocto Project with its ability to create tailored and optimized Linux distributions, its comprehensive toolset, and its focus on reproducibility and flexibility was chosen as the primary build system for this project and was used to generate the associated Embedded Linux and application programs.

The Target Embedded Device

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 device. 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 *bootloader* program capable of that task as well. A bootloader is responsible for placing an operating system into memory and handing over the control of the processor. The technical details as to how the bootloader 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 playing 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 system maker's Yocto meta layer, memory mapping for the attached devices, source codes for boot ROM firmware or the bootloader, etc. The reverse engineering efforts to attain this information were dropped due to time constraints, the details of the attempt to uncover this information is laid out in Appendix I. Ultimately, the MCIMX6Q-SDB evaluation board (Figure 1.4) was chosen as an alternative to the ECU. The required information for the MCIMX6Q-SDB is publicly provided by processor chip vendor NXP.

A brief aside on Continuous Integration and Continuous Delivery

An important aspect for software development in general are the ideas behind Continuous Integration (CI) and Continuous Delivery (CD). CI/CD are modern software development practices that focus on automating and streamlining the process of building, testing, and deploying software. CI is the practise of frequently integrating code changes into a shared repository, performing automated tests to identify and fix issues that arise from such integration efforts, and promoting collaboration, early bug detection, and code quality improvements. CD extends the ideas in CI by automating the deployment of code to production environments. It enables rapid, reliable, and consistent software releases and aims to deliver code changes to end users quickly and efficiently. CI/CD practises foster a culture of automation and agility, allowing development teams to deliver high-quality software more frequently and with greater confidence. CI/CD has been quickly adopted to modern software development practises including embedded software development.

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, the number and the nature of Software Engineers using the tool, the CI/CD infrastructure, and the list of supported hardware devices. Consider a simple development scenario targetting the embedded hardware device MCIMX6Q-SDB, the primary hardware utilized for this report. The aim is to build and run an Embedded Linux distribution for the MCIMX6Q-SDB using a personal computer running Linux. Note that several of the code listings used in this discussion may break their interface, hence the emphasis is primarily on the essential concepts involved.

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. The configuration files, also called *recipes*, 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 device and distribution independent recipes and other metadata. 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 1.3. A set of recipes (teal) sharing a common purpose are arranged into a *layer* (cyan). Layers can depend on other layers, with multiple layers usually being used in a single Embedded Linux distribution. Note that a legend is also present in the bottom right of the diagram in Figure 1.3.

The primary version control system used within the Yocto Project is git. The Yocto Project has several other software packages that are required for its build system, all of which are laid out in its documentation (see docs.yoctoproject.org). Furthermore, there are minimum versions for the required packages for a given version of its release (see wiki.yoctoproject.org/wiki/Releases).

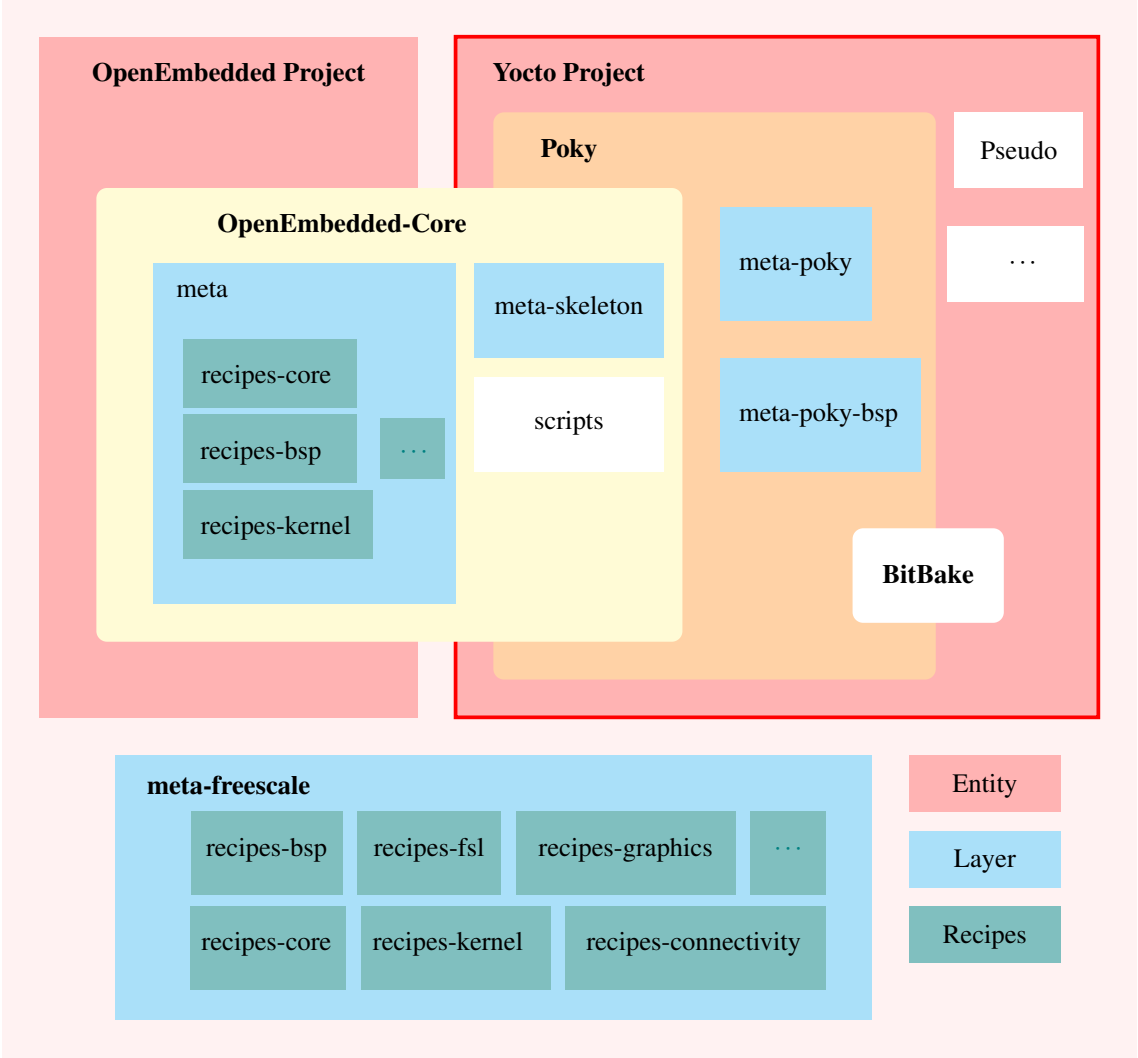


Figure 1.3. Overview of The Yocto Project Components

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
```

The above list is an incomplete example and relies on non-standard package names. Consulting the Yocto Project documentation provides a more accurate instruction as to the package names that are necessary to start with the Yocto Project on a given Linux distribution. An example install of the packages necessary on the Fedora 37 operating system as recommended by the Yocto Project documentation is given in the following code listing.

```
sudo dnf install gawk make wget tar bzip2 gzip python3 unzip perl patch
 ↪  diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath
 ↪  ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue
 ↪  perl-bignum socat python3-pexpect findutils which file cpio python
 ↪  python3-pip xz python3-GitPython python3-jinja2 SDL-devel rpcgen
 ↪  mesa-libGL-devel perl-FindBin perl-File-Compare perl-File-Copy
 ↪  perl-locale zstd lz4 hostname glibc-langpack-en
```

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 the different components by checking out the corresponding branches of the git repos. Using a different version of the Yocto Project would primarily mean checking out to a different branch or tag at this point.

```
git checkout kirkstone-4.0.11 # for poky
```

```
git checkout kirkstone # for meta-freescale
```

Once the correct branches have 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 (assumed that `$BUILDDIR` holds the pathname) 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 set up BitBake along with some additional tools and scripts in the current shell environment and also creates a few directories in `$BUILDDIR` along with some configuration files. Note that after sourcing the `oe-init-build-env` script, the shell switches to the `$BUILDDIR` directory. The first configuration file to edit will be the `$BUILDDIR/conf/bblayers.conf`. The `bblayers.conf` file contains a list of layers to be used in the Yocto Project set up. The `bitbake-layers` utility program can be used to list and modify the entries in the `bblayers.conf` file.

```
bitbake-layers show-layers # list layers in the Yocto Project set up
```

The above code listings shows `bitbake-layers` being used to list the layers of Yocto Project set up. The command should then list the `meta`, `meta-poky`, and `meta-yocto-bsp` layers. The names of the layers may change with the particular version of the Yocto Project being used. The path corresponding to those layers and a priority value attached to the layers will also be shown by the `show-layers` command of `bitbake-layers`. The priority is a means for BitBake to choose between layers for searching a recipe for particular software package, since multiple layers may have instructions for the same package. The `meta-freescale` layer that was cloned in the previous command to some path `$META_FREESCALE_PATH` can then be added to the Yocto Project configuration using the same utility.

```
bitbake-layers add-layer $META_FREESCALE_PATH
```

The next step is to configure the build system for the MCIMX6Q-SDB. The Note that this step may also be avoided by setting the `$MACHINE` environment variable.

```
# edit inside $BUILDDIR/conf/local.conf
MACHINE = "imx6qdlsabresd"
```

After `$MACHINE` is configured correctly, BitBake can create a minimal image that can boot the MCIMX6Q-SDB by building the packages for `core-image-minimal` recipe. BitBake can parse a recipe file to determine a series of tasks to generate a software package or some binary artefacts. To satisfy the `core-image-minimal` recipe, BitBake has to perform a series of operations. BitBake has to parse a list of recipe files that are associated with the `core-image-minimal` recipe, examine the dependencies defined in the recipes and recursively determine all the software components that are required.

```
bitbake core-image-minimal # generate images for MCIMX6Q-SDB
```

At this point the Yocto Project has been configured to find the correct metadata for building software packages and the Linux kernel for the MCIMX6Q-SDB. BitBake will proceed by first going through the `core-image-minimal` recipe to determine the order and list of tasks to be executed. After listing some information on the particular build configuration, it will then proceed to execute the tasks. The generated output images can be found in `$BUILDDIR/tmp/deploy/images`. The path ends at the same name as the `$MACHINE` variable name and contains several binary images. One way to use the resulting binary images to boot the MCIMX6Q-SDB is to use an SD card and the *rootfs* image. The rootfs image should be named with the name of the machine, a time stamp, and the string `"rootfs"`.

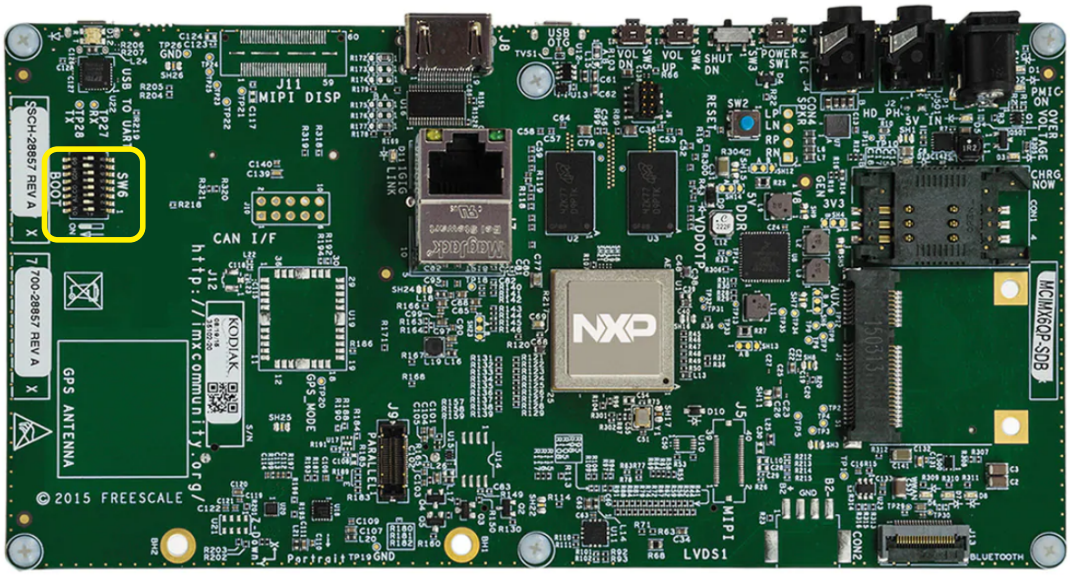


Figure 1.4. MCIMX6Q-SDB-BD

Booting the board using this image is one way to verify the build completed by BitBake. Obtain and flash an SD card with the *rootfs* image using a suitable program such as `dd`. Furthermore, the MCIMX6Q-SDB should be configured to boot from an SD card slot. The board can be configured to boot from the SD-3 slot by toggling the dip switches (shown in yellow highlights in Figure 1.4) to the configuration : `D1-OFF D2-ON D3,4,5,6-OFF D7-ON D8-OFF`.

Assuming the previous steps were completed in a correct manner, the board should boot using the Embedded Linux that was created by BitBake using the `core-image-minimal` recipe. The Micro-B USB plug beside the previously mentioned dip switches can then be used to start a serial terminal emulator program using USB UART using a 115200 baud rate. Alternatively, the board should be accessible via the Ethernet interface also present on the MCIMX6Q-SDB.

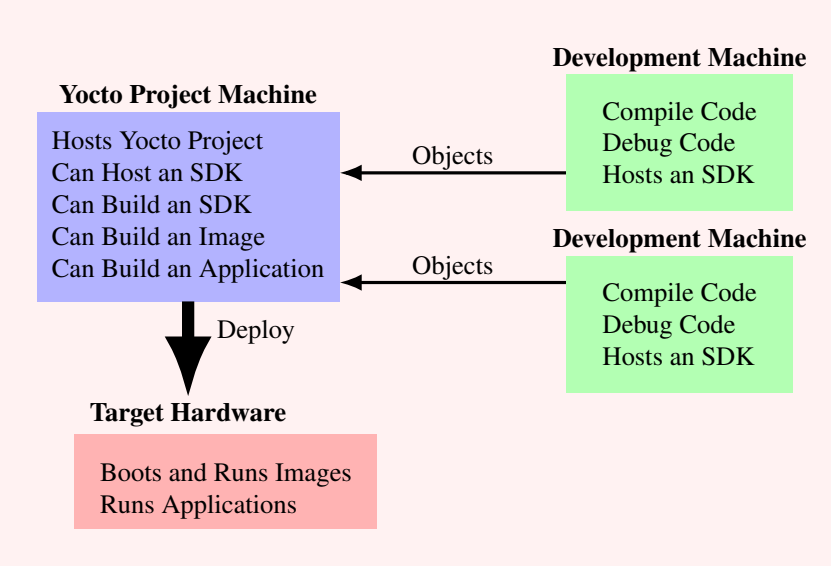


Figure 1.5. Simplified overview of an Application development practise using the Yocto Project

The Yocto Project can be used work on several aspects of Application and Kernel level development for Embedded Linux. Application development in the Yocto Project can be thought to have several practises such as arranging the information regarding the application packaging in a related metadata layer within BitBake recipes. The development process would also have multiple computer machines associated with it as shown in the diagram in Figure 1.5. In this model, the Yocto Project machine is the core environment for building the software stack, including the Linux kernel and user space applications. The development machine is used for coding, compiling, and debugging applications, while an embedded device runs the final images and executes the developed applications.

1.2.4 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 different hardware devices. 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 an `armv7` 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. As with the previous code listings within the section about the Yocto Project, some commands may break their interfaces, such as fedora's package manager utility `dnf`. However, the general principles maybe used to create a similar setup on Linux distributions.

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 set up 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 interpreter 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 registered, `binfmt_misc` recognizes the associated binary files by matching some bytes at the beginning of a file with a magic byte sequence that had been supplied. The usage of magic bytes at the beginning 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. If `binfmt_misc` is not already present at the default path location at `/proc/sys/fs/binfmt_misc`, it may be mounted by using the following command.

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

To register the binary format and interpreter pair to set up the development environment, a string of the form `:name:type:offset:magic:mask:interpreter:flags` needs to be sent (`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 `armv7`. 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 the required software packages for the `armv7hl` architecture via its package manager `dnf`. The following `dnf` command can source the necessary programs for the `armv7hl`.

```
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
```

Note that the *h* in `armv7hl` stands for *hard float* indicating that the architecture uses hardware acceleration for its floating point operations, while the *l* stands for *little endian* byte order. Endianness refers to the byte order in which multibyte data types are stored in computer memory.

It determines whether the most significant byte (MSB) or the least significant byte (LSB) comes first. If the MSB is stored in the smallest memory address, the system is said to be big endian. Otherwise, if the LSB comes first, then the system is said to be little endian. Endianness affects how data is interpreted and manipulated by different processors and architectures. ARM processor usually support both endianness while the little endian byte order is the most commonly used.

The target directory `/tmp/f36arm/` will then contain a simple root directory for the required development environment for `armv7`. Note that the choice of the fedora Linux release version (36) and the `/tmp/f36arm/` directory is somewhat arbitrary. 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 `armv7` 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

```
#include <stdio.h>

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

Development environments such as these are easy to set up and can prove valuable for rapid prototyping. Another use case may be as part of a CI/CD mechanism for a more complete embedded application development lifecycle.

1.3 Neural Network Application Development

The software development process for neural network application in industry has several steps. These steps include 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. This process requires several software components, network resources, computing devices, engineering personnel to come together for the effective deployment such an application.

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. 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 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 devices almost invariably target these hardware devices and provide software tools for developers to utilize on them. Developers in these devices enjoy several resources such as productivity tools that allows for CI/CD, performance profiling tools, etc [18].

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

1.3.1 Choice of Programming Language and Machine Learning Framework

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 machine learning frameworks but not to the same extent as their Python interfaces.

Machine learning frameworks also utilize 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 device 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 Intel's oneMKL [8] or XNNPACK [7] 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 devices, then implementing the neural network model inference on the embedded device. Preferably the implementation utilize linear algebra kernel libraries that are made specifically for the embedded device and the popular machine learning frameworks may provide an avenue to transfer models written in them to target the embedded device.

The possibilities in making embedded devices more involved in the neural network development process has been explored in research avenues such as TinyML [22], 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 devices. Efforts in enabling On-Device Training where the significant parts of the neural network model training happens on the embedded device is an area of active research.

1.3.3 Neural Network Training on Embedded Devices

Neural network training is a fundamental process in machine learning where a model learns to make predictions by adjusting its parameters based on input data. 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 device that is executing a learning algorithm for some neural network is issuing a series of floating point addition and multiplication instructions. Modern computers have specialized hardware features that allow for parallel vectorized executions of these multiply and add instructions, all in an effort to improve the training efficiency of neural networks. These hardware optimizations are particularly beneficial when dealing with larger datasets and complex network architectures.

Further specializations of hardware such as in the case of Application-Specific Integrated Circuits (ASICs) have improved upon this trend. Google's Tensor Processing Unit or TPU chip shown in Figure 1.6 is an example of such an ASIC meant to execute neural network operations, and other similar large matrix operations with great efficiency.

Most of the existing embedded hardware are not especially designed for machine learning or neural network acceleration like the ASICs. Hence, it becomes important to understand the difficulties in moving the training of a neural network to the embedded devices and the capabilities of modern software systems to complete these implementations efficiently [23]. On-Device training

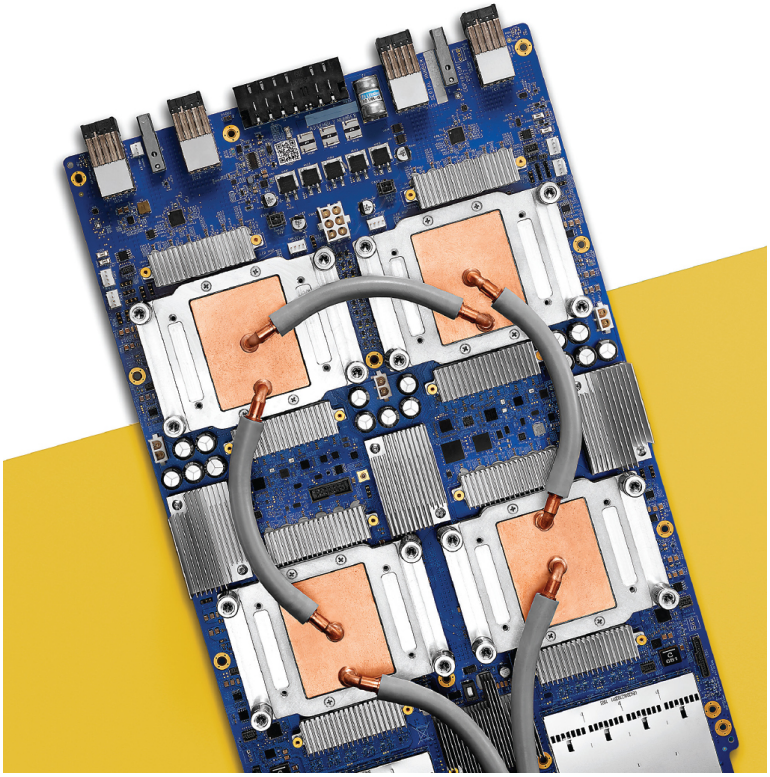
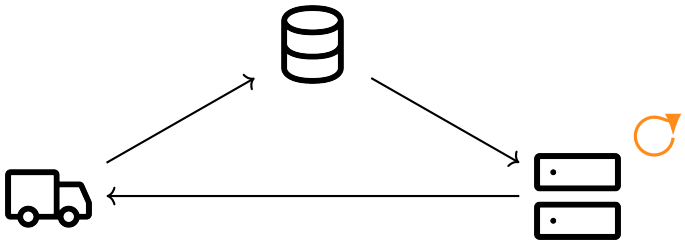


Figure 1.6. Google's TPU Chip

strategies also allow for the possibility of reducing the movement of data from embedded devices to achieve learning.

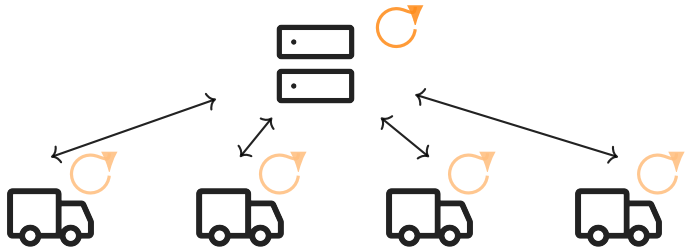
1.4 Federated Learning of Neural Networks

A significant problem in the traditional model for neural network application development in embedded devices is the consumption of network bandwidth associated with continual transmission of sensor data to the data lake. The data from different devices are combined to form the dataset that will be used to further train the model. However, this stream of data coming from the embedded device 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 decentralized manner. Federated learning is a technique of training machine learning 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 sent 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. There are several algorithmic strategies to

choose from to combine the local training sessions based on the differences in the data set on the embedded devices.



Federated learning implementations can thus differ based on how the distributed training takes place, the algorithm to combine different locally trained models, and other strategies used in completing the development loop. In the LOBSTR [21] and FAMOUS [20] 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.

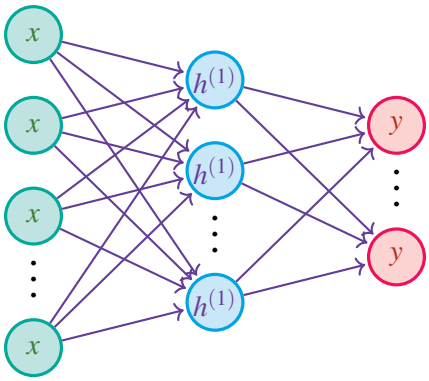
Federated learning allows for joint training of vast amounts of data without exchanging that data. Another aspect to consider is how the locally trained models are sent back to a central server for combining with other such models. Some form of gradient compressions strategies may be required to ensure that the model updates do not leak information about the data on board the embedded devices.

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, contextualized 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 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.

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 prediction or output and this step is said to be a neural network *inference*.

2.1.1 Neural Network Training

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 corresponding 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.

In the domain of neural network training, *Stochastic Gradient Descent (SGD)* is the cornerstone learning algorithm. This technique enables neural networks to learn from data, adjust their parameters, and improve their performance over time. Visualize a vast and undulating landscape, which represents the state space of values that the neural network model can take. The objective of learning algorithm is to reach the lowest point in this landscape that corresponds to the least error in the neural network models. While a complete view of the terrain is concealed, the direction of the steepest descent underfoot can be discerned. This mirrors the role of *gradient descent* in the learning algorithm.

The goal of the SGD algorithm is to minimize a *loss function*, which quantifies the discrepancy between the predicted outputs of the network and the actual target values. The gradient of this loss function provides the direction of the steepest ascent, and by moving in the opposite direction, we can descend toward the minimum. Mathematically, given a loss function $\mathscr{L}$ and parameters w, the gradient descent update step can be expressed as:

$$w_{new} = \theta - \eta \cdot \nabla \mathscr{L}(w) \tag{2.2}$$

Here, $\nabla \mathscr{L}(w)$ is the gradient of the loss with respect to the current parameters w, and η is the *learning rate*, a hyperparameter that controls the step size. The learning rate governs the trade-off between *convergence speed* and *stability*. Convergence

Calculating the gradient over the entire dataset for each update can be computationally expensive. SGD addresses this challenge by introducing randomness and *mini-batch sampling*. Instead of computing the gradient using all data points, SGD randomly selects a subset of the data called the *mini-batch* for each update. This not only speeds up the computation but also adds a form of noise that can help the learning algorithm escape local minima.

$$w_{new} = w - \eta \cdot \nabla \mathscr{L}(w; \textit{mini-batch}) \tag{2.3}$$

Another important improvement to how the learning algorithm works is in techniques used to calculate the gradient weights. The *backpropagation* algorithm is a critical mechanism used to update the weights of a neural network during the training process. It involves computing the gradients of the network's loss function with respect to its parameters by traversing the network in reverse order, from output to input layers. These gradients quantify the impact of each parameter on the overall error, enabling the network to adjust its parameters in a direction that minimizes the error. Backpropagation in SGD enhances the convergence of the optimization process by iteratively adjusting weights based on the computed gradients and a small learning rate, ultimately guiding the network towards better performance and improved ability to generalize to unseen data.

2.2 Embedded Linux

As presented in the previous chapter, the development, deployment, and maintenance of Embedded Linux distributions and their user mode applications are usually managed using capable build systems. Configuring these systems requires understanding concepts such as bootloaders, device tree layouts, flash memory, cross compiling toolchains, board support 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. The next part of this section introduces yet more concepts and terminology around the boot sequence for a commonly used embedded board running an Embedded Linux.

2.2.1 An Overview of the Standard Boot Sequence

After power up, the processor requires initialization of its constituent hardware components. The processors usually found in ECUs will have this function performed by *firmware* placed in a special purpose memory called *boot ROM*. Firmware is software that provides low level control of a devices specific hardware that is placed on embedded hardware by the silicon vendor or the system maker. The boot ROM, as the name suggests, is a type of read-only memory that is usually placed close to the processing unit. After powering on, the processor is said to come out of reset. The processor is designed to look up the memory mapped to the boot ROM after reset. The boot ROM code then starts executing on the processor and starts initializing peripheral devices, hardware busses, CPU registers, etc. After initializing the hardware, the boot ROM code is responsible for locating and loading a bootloader program. At this point the boot ROM may also perform additional verification procedures such as *checksum* validation. Checksum is a small block of data derived from another usually larger block of data that is used for the purpose of detecting errors or performing data integrity checks. The boot ROM may have some way of checking the data present inside a flash memory and computing a checksum calculation to verify based on some notion of data integrity. This location is usually on some external memory to the processor such as an *embedded MultiMediaCard* (eMMC). eMMC is one among several memory card standards for solid state memory storage.

Bootloader programs are responsible for continuing the boot process and may have multiple stages with one stage loading another. The first stage of booting may deal primarily with configuring the memory controller, and the main memory such as a *synchronous dynamic RAM* (SDRAM). The second stage of the boot process may then deal with identifying and preparing an operating system to complete the boot. The second stage may also have to configure additional peripheral devices, perhaps even provide an interface to select between different versions of operating systems available on across the peripheral memory devices. U-Boot is the most commonly used open source bootloader for embedded devices. 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. Different flash memory standards rely on either a NOR flash or NAND flash based scheme, for e.g. the eMMC standard uses NAND flash.

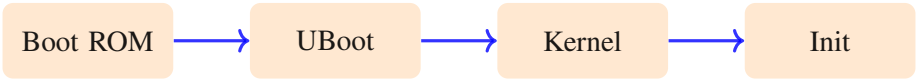
U-Boot may also be divided into multiple stages, as described previously, on different flash memory. One such multistage division for U-Boot is into the *Secondary Program Loader* (SPL) and a more fully featured U-Boot. SPL could then provide the first stage of the boot procedure and hand over control to the more fully featured U-Boot as the second stage. After getting control

of the processor, U-Boot then has to take care of initializing 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 dissuade the creation of U-Boot and Linux forks to target marginally different boards [5].

U-Boot implements a subset of the Unified Extensible Firmware Interface (UEFI) specification outlining the architecture of the device firmware used to boot and its interaction interface with the operating system. The full feature set of U-Boot includes an UEFI compatible boot manager utility, a device management utility, explicit memory handling to allow data copying and kernel execution, and a command line interface (CLI) to provide access to these features.

U-Boot starts by locating the kernel to be loaded. The linux kernel is usually stored in some compressed form and the first step U-Boot takes is to decompress that kernel from flash memory, onto the main memory. After U-Boot completes its tasks, it gives up control over the processor to the kernel. The control ends up with the `start_kernel` method in the kernel code, which is an architecture independent starting point for the rest of the Linux kernel boot process. The kernel proceeds with yet more configuration steps such as configuring the memory, processor, peripherals, cache, and other hardware devices.

The *process* abstraction in Linux refers to the concept of treating running programs as independent and isolated entities that operate concurrently within the same system. Each process represents the execution of a program and is allocated its own view of memory and system resources like the CPU time and I/O operations. Processes in Linux are managed by the *scheduler*. After configuring the processor, memory, and other peripherals, the kernel proceeds to complete its start up by setting up kernel data structures in memory, initializing the scheduler, setting up and allocating *pages*, etc. A memory page is a fixed size block of data used in virtual memory systems serving as a unit for memory allocation, management, and data exchange between physical RAM and secondary storage devices. The virtual memory system is a memory management technique used by modern operating systems to provide an abstraction of a larger and more flexible memory space to applications than is physically available in the computer's physical RAM. It allows programs to use memory addresses that are independent of the actual physical addresses of the underlying hardware. This enables efficient memory usage, process isolation, and the ability to run larger programs than the available physical RAM would allow.



The kernel completes the start-up after initializing the root filesystem and spawning the `init` process, which is the first process to that starts after booting completes. The `init` process is responsible for bringing up several user level programs and system services. Processes are created through a mechanism called "forking," where an existing process (parent) generates a copy (child) with its own memory and execution state. The `init` process in this manner acts as the ancestor process for all other processes. Once the kernel has successfully initiated the init process, the user space environment is said to be instantiated and the higher-level processes and applications take over. At this point, the system becomes operational and ready for user interaction and application execution.

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 separately 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 memory and computation 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.

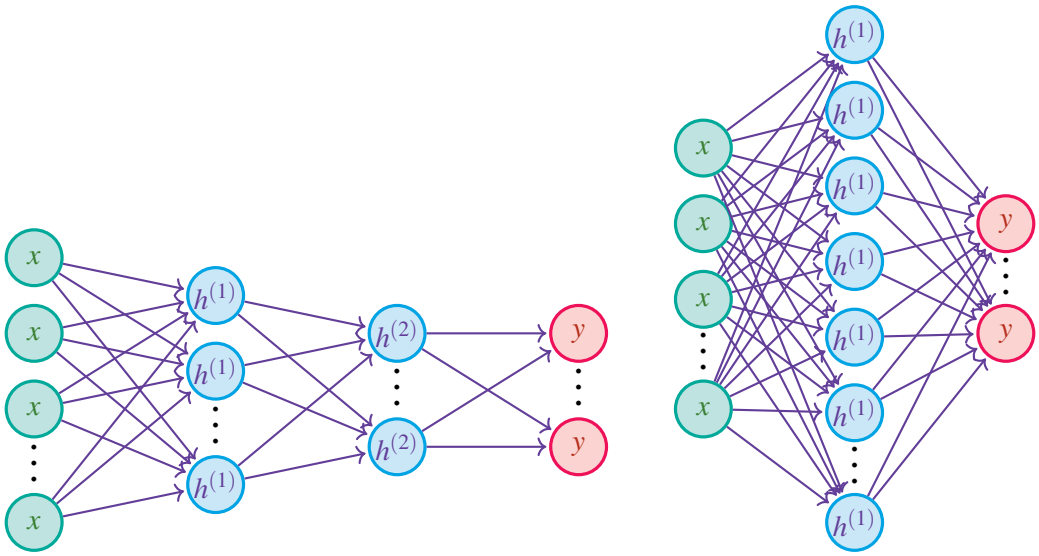
3. Design

The benchmark applications test the training of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [9] 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 characteristics 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 throughout 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 η 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) + ηΔw^(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 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 separated 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 laid 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 useful 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 separated. 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. 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++. The 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 maintained 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 implemented 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 handwritten digits. The model is trained using stochastic gradient descent, which is an optimization algorithm used to minimize a loss function. The back propagation 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 utilized 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 an Embedded Linux distribution for the `imx6qsabresd` machine. The NXP Yocto project guide [16] 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 preliminary 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 referred to as peak memory utilization 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 individually 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` optimization flag. Several other optimization opportunities that were possible and tried out however the data present for `-O2` was fixed for all the applications for uniformity.

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

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 in Appendix II revealing the areas where the applications were spending the most time on.

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. The resulting graphs are also presented in Appendix II

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 based on execution time and the peak memory usage.

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.

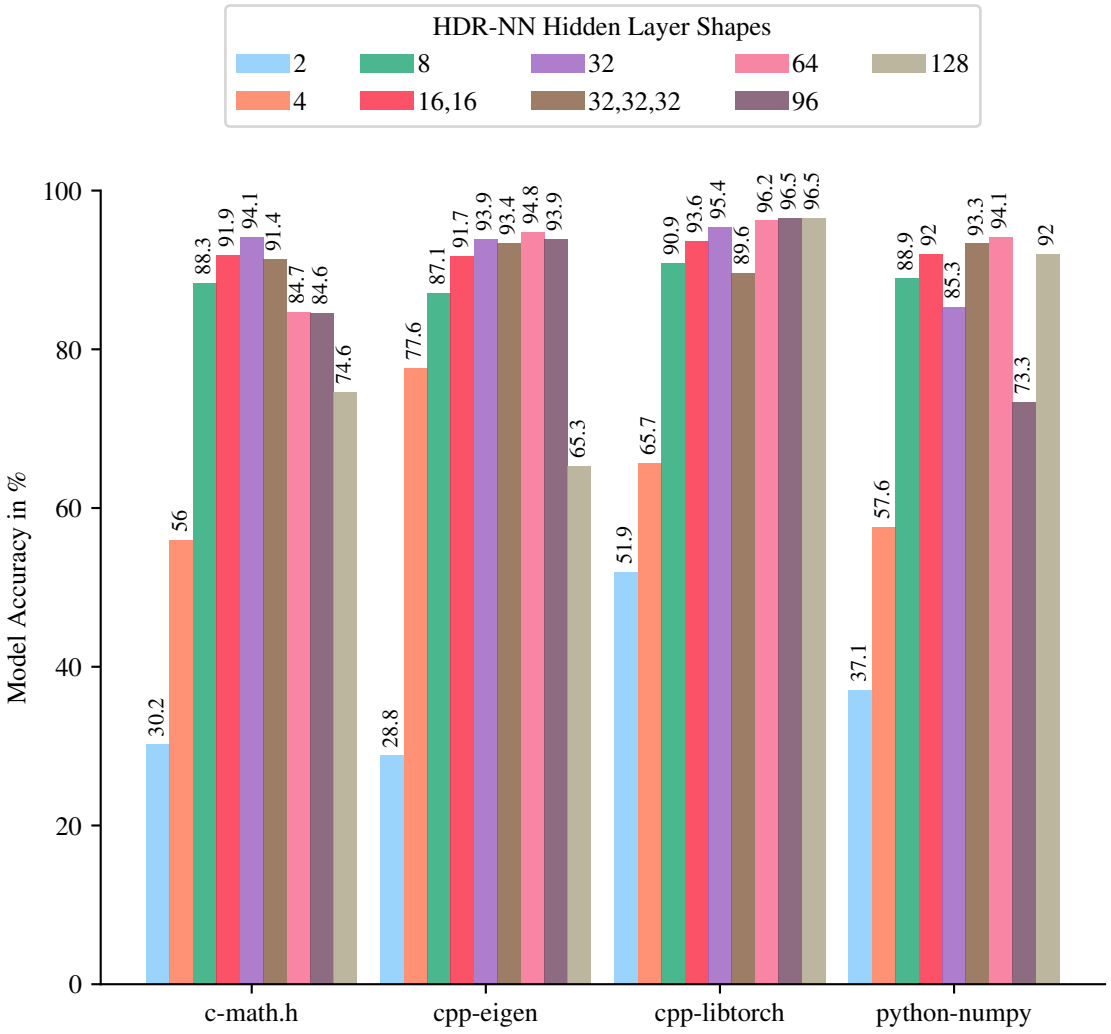


Figure 6.1. Comparing the accuracy of the different HDR-NN implementations.

6.2 Evaluating effectiveness

The primary measures used to evaluate the program performance was execution time and memory utilization 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 observed in Figure 6.2 where the execution time increases drastically as number of parameters increases.

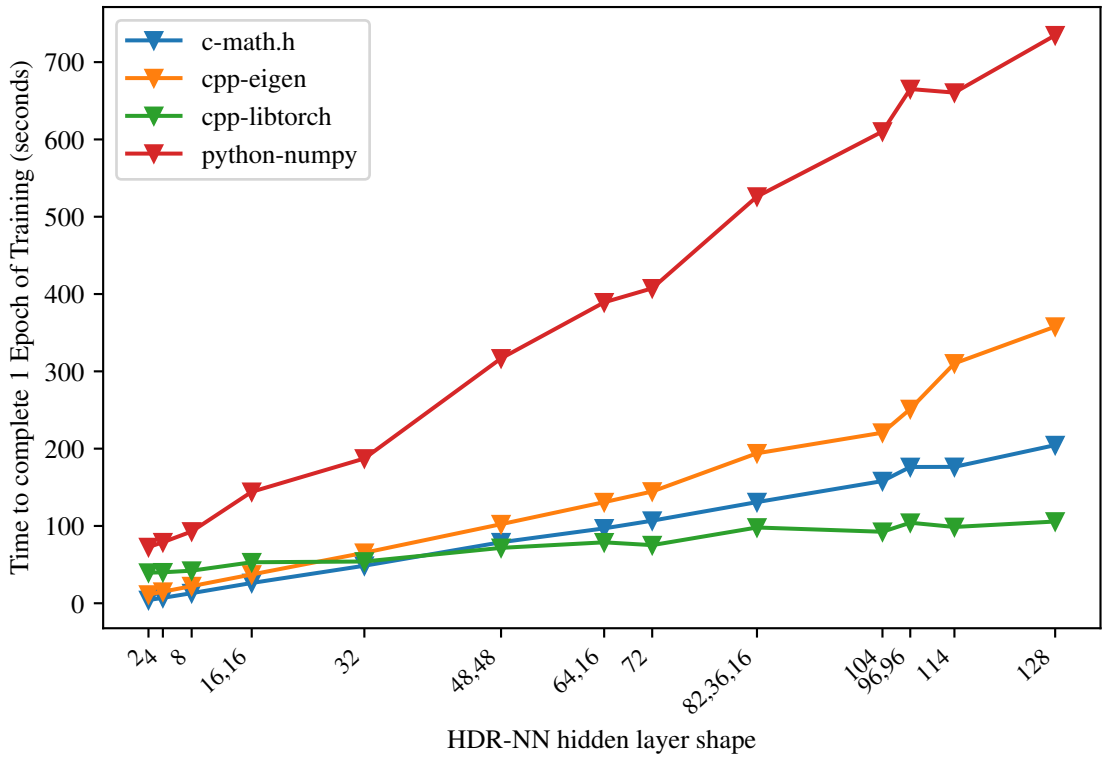


Figure 6.2. Comparing total run time for training the different HDR-NN programs

6.2.2 Peak Memory Usage

Regardless of the hidden layer sizes, the peak memory utilization 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.

Note that the device RAM is 1024 MB however the peak memory utilization for both `C` and NumPy are higher than this value. This can be explained by over allocation of memory by the operating system utilizing the swap space. The peak memory utilization 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.

Program	**1600**	**6370**	**13002**	**40522**	**57250**	**85642**	**101770**
PyTorch	1171.8	1171.8	1171.8	1173.8	1172.9	1172.9	1174.8
NumPy	1303.7	1303.7	1303.7	1303.7	1303.7	1303.7	1303.7
Eigen	752.6	752.7	754.3	755.8	757.1	758.6	759.9
C	1079.0	1079.2	1079.5	1080.8	1081.6	1082.9	1083.7

Table 6.1. Peak Memory Utilized for a single epoch during training based on Maximum Resident Set Size measure (MB)

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 Comparing Benchmark Applications

As the benchmark applications share identical structures and configurations, it is expected that the model accuracy will be consistent. Figure 6.1 illustrates that the various implementations perform comparably across different parameter counts. However, an anomaly arises when the parameter count exceeds 101,770, as seen in the accuracy drop of the `C` implementation, possibly due to an unidentified bug.

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 7.1. Percentage difference in accuracies between the implementations.

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%

Figure 7.2. Percentage difference in peak memory utilized between the implementations

Figures 7.1 and 7.2 shows the comparison between `C`-`C++` Eigen, `C++` Eigen-`Python` Numpy and `C`-`Python` Numpy. It is calculated as the percentage of 1 - $(\frac{x}{y})$ where x is performance value of the application which has lower value and y is performance value of the application that has higher number.

7.1.1 C vs Eigen

For smaller models, `C` exhibits superior speed over Eigen (e.g., 38% faster for a network size of 1600). As the model complexity grows, `C++` Eigen's performance surpasses `C`, with execution

time differences less than 15%. However, there are noticeable differences that emerge at larger network sizes. Both the C and C++ Eigen proves 26% faster than Eigen. Further investigation is needed to pinpoint the source of this anomaly. Eigen excels in memory utilization, outperforming C by 30%.

7.1.2 C vs Numpy

Notably, C consistently outperforms Numpy. In the case of a network size of 101,770, C exhibits a remarkable 62% speed advantage. Numpy, however, consumes 17% more runtime memory compared to C.

7.1.3 Eigen vs Numpy

Similar to the C comparison, Eigen consistently outpaces Numpy across various network sizes, maintaining similar speed differentials. Eigen shines in memory efficiency, surpassing Numpy by 42

7.2 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 components that have varying degrees of maintenance and create complicated interrelationships over time that are difficult or unwieldy to maintain. This has lead to a lack of good software infrastructure that can support multi-hardware, multi-architecture neural network frameworks.

7.2.1 Reverse Engineering Scania ECU

The Scania Electronic Control Unit (ECU) is akin to a mysterious black box with its inner workings covered in secrecy. It divulges minimal information, leading to the ambitious task of porting a customized Embedded Linux seem like threading a needle in the dark. The little insight available from this device comes primarily in the form of a custom encased hardware, housing Ethernet over modem and a UART interface. Supplementing this, a hardware circuit schematic document along with certain scripts adds a touch of clarity to an otherwise murky puzzle. Regrettably, the information regarding processor specifications, memory unit, bootloader specifics, the boot sequence, and of its peripherals remain elusive.

The Scania ECU proves to be a formidable challenge for any would-be developer, given its status as a production unit. The absence of development tools on the device itself strips away the convenience of on-device programming, rendering ambitions of porting packages or applications a fruitless endeavour. Even the basic functionality of executing common commands that would yield insights into the system's inner workings has been stymied by the intentional disabling of many features on the bootloader and kernel.

The repurposing of Scania ECU involves a dual-pronged approach encompassing: figuring out how it works by taking it apart (reverse engineering) and applying the learning to build and port a custom Embedded Linux. The veil was partially lifted, revealing key details such as the processor type, architecture specifications, input/output interfaces on the board, and the essential boot flow process. Similarly, valuable software information, such as bootloader specifics, the kernel and compiler details, the C library version was successfully extracted. Despite these triumphs, the path forward was muddied by the continued absence of a clear memory layout and the essential device tree configuration.

Endeavours to install a custom Embedded Linux system uncovered a complex web of challenges that proved resistant to resolution. Divergent outcomes arose from booting the ECU under normal circumstances compared to booting it through serial download mode, each beset by their unique set of issues and subsequent failures. These attempts, unfortunately, yielded the disheartening result of rendering the ECU non-operational, a state colloquially referred to as being "bricked". The scarcity of available boards further made the experimentation process difficult.

The process of flashing within a regular boot sequence was constrained by the absence of development tools. The scope reduced to replacing the bootloader with the MTD partition and hoping to reach the bootloader shell. Failures during this phase could be attributed to a plethora of factors: erroneous u-boot images, inconsistencies within the device tree configuration, discrepancies between the versions of the bootloader and kernel, checksum mismatches, or complications arising from oversized files overwriting critical memory regions. The precise cause could not be identified, as there were no logs enabled during the board's booting process.

Serial download mode, on the other hand, emerged as an even more intricate endeavour. Successful flashing hinged on procuring precise information regarding the memory load addresses and entry points for essential components like the bootloader, kernel and root file system. These crucial details remained elusive, casting a shadow over the feasibility of the flashing process.

Appendix I contains a summary of the efforts involved in reverse engineering the Scania ECU. The project focus shifted to the MCIMX6Q-SDB board after having spent considerable time on the board.

7.3 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 ECU. 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 are version controlled in two separate 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 ECU 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 Scania ECU

Repurposing the Scania ECU is a technical challenge that was left incomplete during the project. The experience as detailed in Appendix II and throughout 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 On Device Training

Multi-framework software infrastructure that allows On Device training are being developed my the community in parts such as Modular [14], EdgeImpulse, etc. There are innovative approaches to bring down the memory requirements of deep learning neural network models such as the Tiny Training Engine [12] and TinyTL [3] by changing the learning algorithm that will be running on-board.

The fundamental bottleneck of implementing neural network training is the amount of memory required. The memory requirements for training a neural network is larger than for neural network inference due to the requirement for storing intermediate activations of the neural network layers. The backpropagation algorithm requires these intermediate calculations of these values to be present while executing the backward pass. Ideas that work for implementing efficient neural network inference on embedded hardware, such as improving parameter efficiency by reducing the number of parameters in the neural network model, do not work to bring down on the total number of activations values required to be stored during the training process.

References

[1] Meta AI. Pytorch mobile. https://pytorch.org/mobile/home/. [Online; Accessed on August 11, 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 August 11, 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 August 11, 2023].

[7] Andri Kulik Juhyun Lee Frank Barchard Ming Guang Yong Chao Mei Jared Duke Erich Elsen Marat Dukhan. Xnnpack. https://github.com/google/XNNPACK, 2019. [Online Accessed on August 11, 2023].

[8] Mariia Krainiuk, Mehdi Goli, and Vincent R. Pascuzzi. oneapi open-source math library interface, 2021.

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

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

[11] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

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

[13] 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.

[14] 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 August 11, 2023].

[15] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2018. [Online Accessed on August 11, 2023].

[16] 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 August 11, 2023].

[17] 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.

[18] Alejandro Saucedo. Awesome Production Machine Learning. https://github.com/EthicalML/awesome-production-machine-learning, 2023. [Online Accessed on August 11, 2023].

[19] 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 August 11, 2023].

[20] 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 August 11, 2023].
[21] 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 August 11, 2023].
[22] Pete Warden and Daniel Situnayake. *TinyML: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.
[23] Shuai Zhu, Thiemo Voigt, JeongGil Ko, and Fatemeh Rahimian. On-device training: A first overview on existing systems, 2023.

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 activities 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 build from the Yocto Project was flashed on MTD partitions with the device information that was available. However, the broad was bricked

presumably 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

Profiling Benchmark Applications

Flame graphs capturing application performance during a single epoch are listed below

Flame graphs from perf Measurements

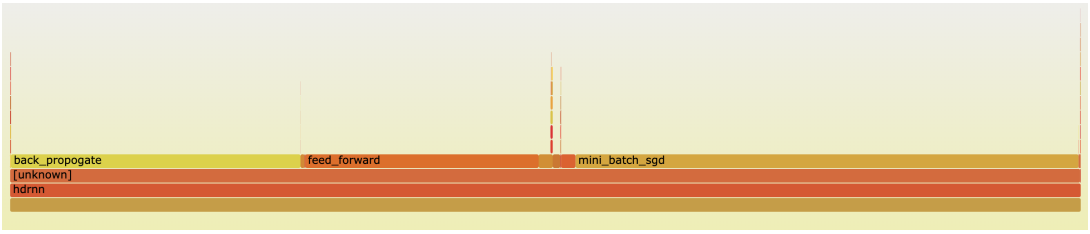


Figure 8.1. c-math.h

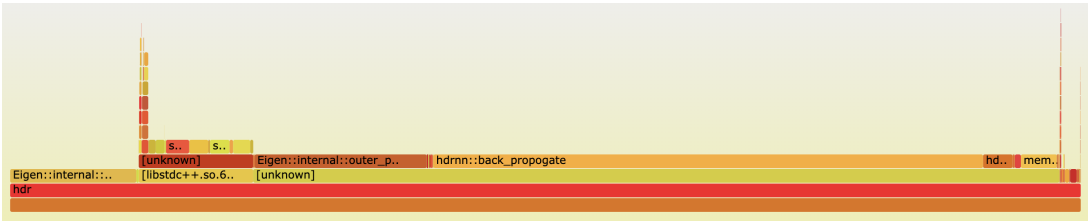


Figure 8.2. cpp-eigen

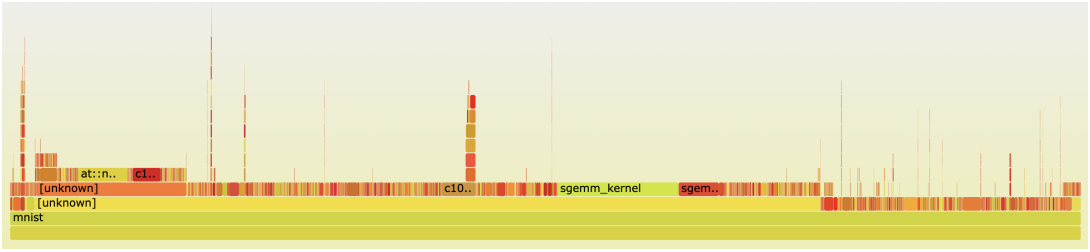


Figure 8.3. cpp-libtorch

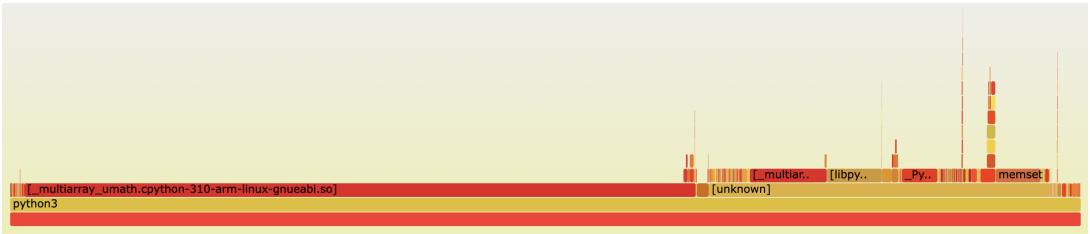


Figure 8.4. python-numpy

Memory Profile through Heaptrack

Heaptrack measurement for C based implementation of HDR-NN running for 4 epochs under different sizes are presented below

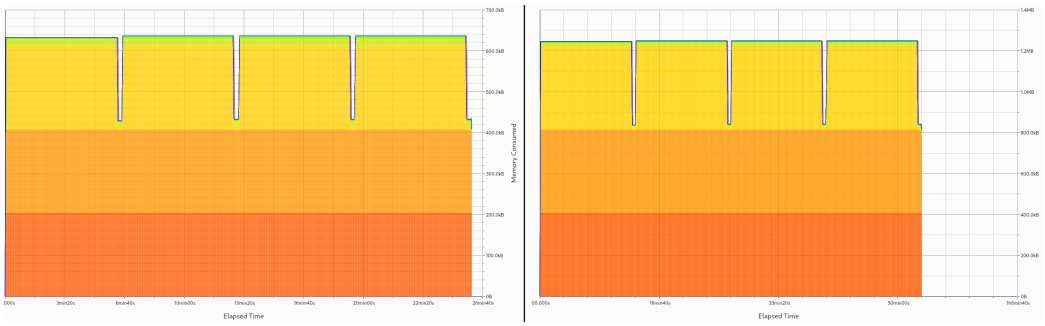


Figure 8.5. c-math.h with 2 different shapes

The memory allocations of different size as tracked by Heaptrack for a single epoch execution run of the C based implementation is given below

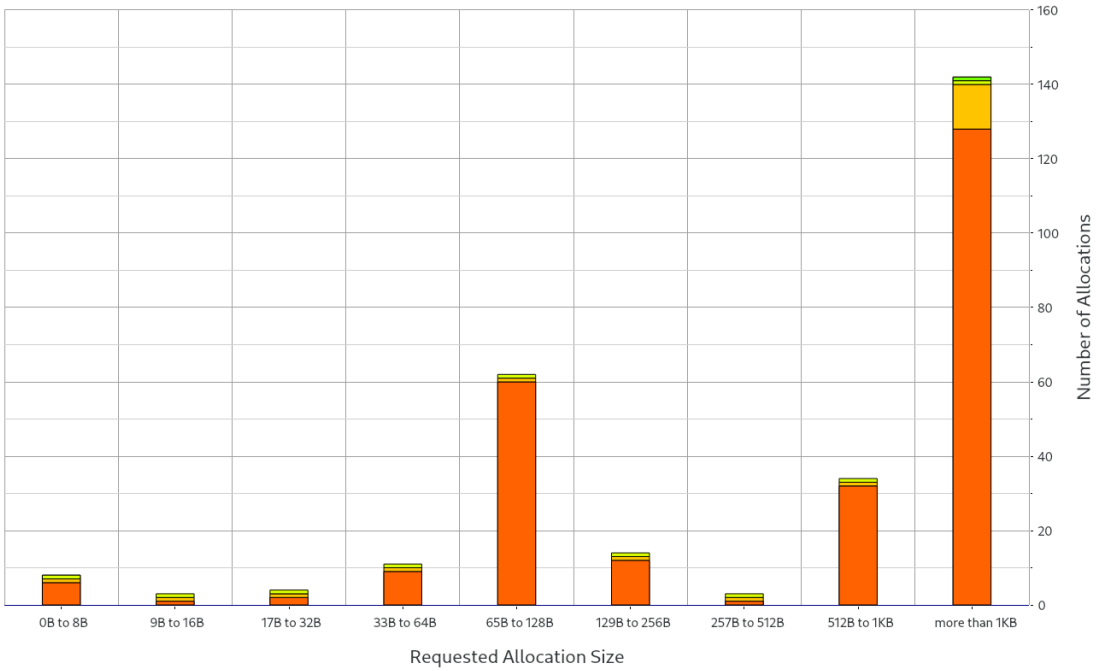


Figure 8.6. c-math.h allocations