Prasanth Shaji, Deepak Venkataram

Benchmarking Training of Neural Networks on Embedded Devices

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

UPPSALA
UNIVERSITET

There is great potential in enabling neural network applications in embedded devices.

Contents

Part I:
Introduction

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

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

Scania currently runs a massive fleet of around 600,000 connected heavy vehicles. The company's truck sales make up 62% of its global sales and Scania has been adding 60,000 trucks to it's fleet annually [11]. This large fleet of rolling vehicles that are connected though the communicators opens up new possibilities. These connected devices continuously monitor the state of the vehicle and this data can be used to accurately and efficiently schedule vehicle maintainance. For example, if a tire change is predicted to be required in 100 kms then the driver can plan the route smartly to reach the workshop before the vehicle breaks down. This opportunity can be realised by running smart algorithms on the hardware that is currently available.

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

One such ML application Scania has been developing in their LOBSTR [13] and FAMOUS [12] projects is anomaly and fault detection on the vehicles. Targeting to run the anomaly detection models on the existing ECUs with limited resources has many benefits and challenges.

Benefits to performing Anomaly Detection on ECUs

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

Challenges to implementing Federated Models

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

Problem Description

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

1. Background

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

The target embedded device to run these applications are the ECUs aboard a Scania vehicle. These ECUs have 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 [7]. Linux is the preferred choice for such a guest operating system due to its configurability and rich support for application development.

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

1.1 Development Process for Embedded Linux

Building and maintaining Embedded Linux distributions with the Linux kernel and user mode applications require tools that can support multi-level build configurations, interface with or build a cross compiling toolchain, support `C` run times such as glibc or musl, and provide support for project management. There are several tools that provide this support such as OpenADK, The Yocto project, Buildroot, OpenWrt, etc., with The Yocto project and Buildroot being the most featureful and presently the most 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 [10]. The Yocto project was chosen as the primary build system and used to generate the Embedded Linux and application programs used within this project.

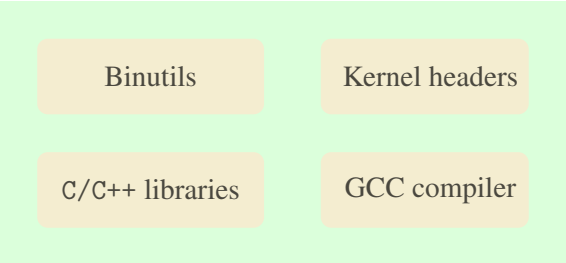
1.1.1 SDKs and Compiler Toolchains

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

Software development toolchains consists of a compiler, linker, libraries, debuggers, as well as a collection of programs to create and manage executable binary programs for a target device such as the commonly used GNU binary utilities, a.k.a binutils. The primary choice for a `C` compiler is the GNU Compiler Collection `GCC`, with LLVM's Clang being the closest alternative. To develop applications that interface with the Linux operating system APIs, the toolchain also contains necessary header files called Linux kernel header files. The last important piece of a toolchain will be the `C` runtime, with the most popular choice being GNU's glibc.

1.1.2 Cross Compiling and Application Development

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



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

1.1.3 Target 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 platform. The collection of software and configurations required to get an operating system running on a board is referred to as a Board Support Package (BSP).

To port Linux onto a processor on a particular board requires creating a boot loader capable of that task as well. A boot loader program is responsible for placing an operating system into memory and handing over the control of the processor. The technical details as to how the boot loader has to be configured will be based on the particulars of the hardware that it will be configured for.

The initial target machine for the project was an ECU filling the role of a communicator on the truck. The BSP source code for the board however was unavailable as well as certain critical support components for the board, such as the vendor's Yocto meta layer, memory mapping for the attached devices, source codes for boot ROM firmware or the boot loader, etc. The reverse engineering efforts to attain this information were dropped due to time constraints and ultimately a similiar board, namely the iMX6SDB evaluation board, with the required information publicly provided by processor chip vendor NXP was chosen as the target platform. The details of the attempt at uncovering this information is layed out in Appendix II.

1.2 Neural Network Application Development

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

1.2.1 Choice of Programming Language and Machine Learning Framework

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

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

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

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

1.2.2 Neural Network Inference on Embedded Devices

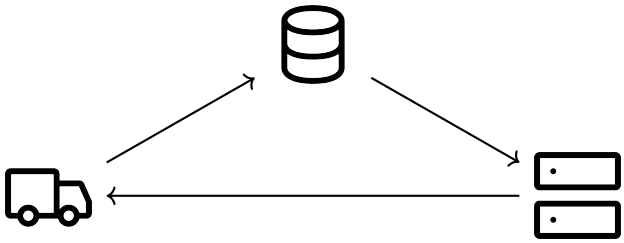
The typical deployment of neural network models on embedded devices follows a pattern of gathering sensor data from the embedded device onto an external data lake, training a neural network model using this data on workstations or cloud platforms, then implementing the neural network model inference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

The possibilities in making embedded platforms more involved in the neural network development process has been explored in research avenues such as TinyML[14], and other efforts motivated by interests in getting the neural network applications ready for mobile devices such as Tensorflow Lite[5] and PyTorch Mobile [1]

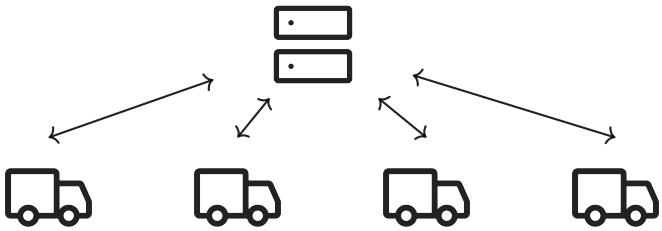
However the primary approach in these cases is with a focus on making the neural network model inference step faster on these embedded platforms.

1.3 Federated Learning

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



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



Federated learning has several different approaches differing in the manner in which distributed training can take place, the algorithm to combine different locally trained models, and other strategies used in completing the development loop. In the LOBSTR [13] and FAMOUS [12] projects Scania has developed statistical models and NN models for anomaly detection using a federated learning approach. The statistical models are lightweight and can easily be trained with limited computational resources.

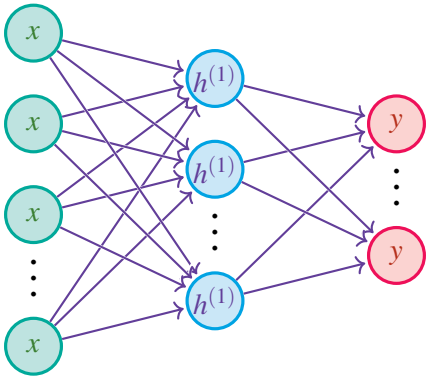
This thesis focuses on repurposing existing hardwared (ECU) to ML edge devices that are tailored to train NN in the most efficient possible way. We try to reverse engineer the old communicator model to build a custom Yocto project tailored for ML. As alternative test ECU we use an evaluation board that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

2. Theory

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

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

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

Combining these operations the neuron y_k has the output

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

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

Neural networks can be constructed in a variety of ways with the choice for how many layers to use, the number of neurons in the layers, the connections between the layers, all of which can

generate several different topologies. The neural network model can approximate a real world system by modelling that system as function that takes in some input and then generating an output. The neural network can approximate this function better by changing the connections between neurons, dropping and or adding neurons, varying the weights encoded in the connections, or by varying the biases within the neurons.

2.1.1 Training a Neural Network

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

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

2.2 Embedded Linux

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

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

2.2.1 A Simplified Boot Sequence

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

After getting control of the processor, U-Boot then has to take care of initialising the memory system, finding then loading the Linux kernel into an appropriate location in memory, generate boot parameters for the kernel, and copy other required data for the kernel. The kernel is also commonly stored on a flash memory on board. One of the configuration data that U-Boot has to pass to the kernel is the device tree, which is a data structure describing the hardware layout. Device trees were adopted in Linux and the embedded industry in general to allow mainline Linux and U-Boot to use the device tree to run on a particular board configuration, and to disuade the creation of U-Boot and Linux forks to target marginally different boards [4].

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

2.3 Performance Evaluation

Performance engineering is the act of employing a variety of techniques to understand and improve some performance aspect of a computer program or software system. Embedded systems are generally resource constrained with limited memory and processing power and once an application can meet its functional specifications, there is a natural pressure to ensure improved resource utilisation and performance efficiency.

The first step to improve these metrics is to measure them and there are several tools that are used by the engineers conducting performance engineering. These tools are generally termed profling of software systems use that rely on the underlying hardware, the operating system environment, etc.

The primary metrics in this mode of evaluation are the execution time, memory utilisation, energy consumption, and system responsiveness. Embedded linux has several interfaces that allow for the measurement of these features for both the kernel and user-level program performance.

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. Targetting even smaller devices with Tensorflow based neural network models is possible for inference only applications via Tensorflow Lite Micro [3]. Efforts to allow training as well in these frameworks require more effort due to the compute and memory intensive nature of the training process.

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

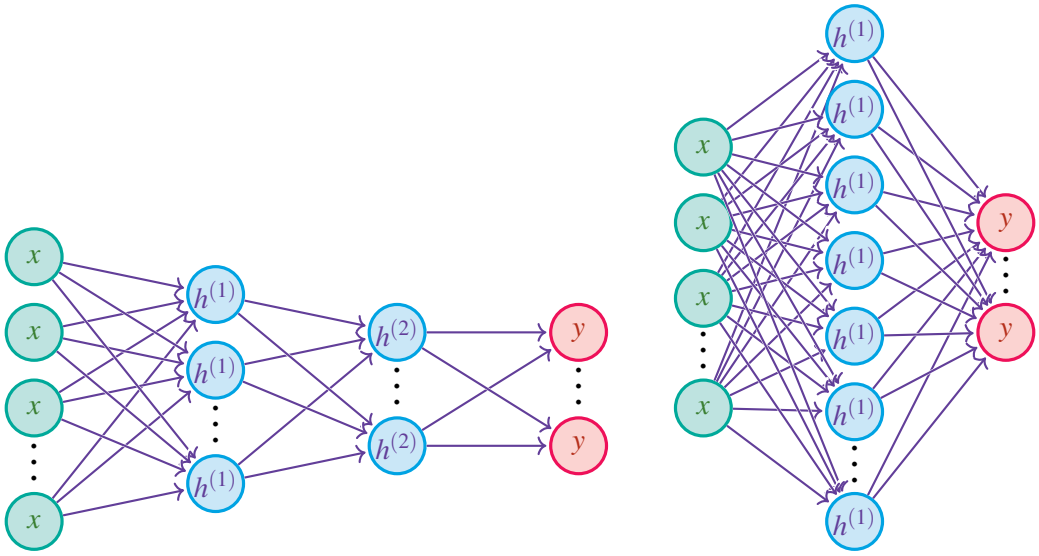
3. Design

The benchmark applications test the training phase of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [6] dataset. MNIST is a popular dataset of handwritten digits commonly used for training image processing systems. It is a popular starting point for neural network implementations and has been used as the primary dataset in the benchmark experiments. The target embedded device is an Electronic Control Unit (ECU) with a Cortex-A9 processor.

3.1 HDR-NN Benchmark Programs

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

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



3.1.1 The Learning Algorithm

The HDR-NN benchmark applications all share the same standard training algorithm listed below (1). Describing this algorithm in general purpose neural network frameworks is straight forward and plenty of general implementations of the algorithm exists, making the development process easier to target multiple programming paradigms. The configurable parameters of the learning algorithm through out the implementations are the learning rate, the total number of epochs for training, and the batch size for gradient descent iterations.

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

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

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

4. Development

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

4.1 Targeting iMX6SDB

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

4.1.1 Compiler Toolchains & Yocto Recipes

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

GCC based cross compilers and debuggers were usefull for the `C`, `C++` programs. The *meta-python* layer provided by Open Embedded was also useful in allowing for applications using Python and Numpy. The general portability of the benchmark applications and the Yocto project allows for further experiments to be conducted on different target architectures as well.

4.1.2 Building PyTorch for iMX6SDB

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 iMX6SDB. The source code could however, with some effort, be used to generate these binaries and for this project a QEMU based user-mode emulation 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 Implementation

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

4.2.1 The Reference HDR-NN in Python

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

4.2.2 PyTorch based HDR-NN

Developing ANNs using PyTorch is straightforward with good support and well documented APIs. There were two primary choices for programming language to write the neural network in PyTorch, namely Python and C++. PyTorch project does not provide binaries for either language choice for iMX6SDB however with the source code of the project being available, the libtorch binaries were generated as mention in the previous section.

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

4.2.3 C based HDR-NN

The `C` implementation had the least amount of external dependencies and contained the data structures of the neural network in float arrays within structs 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 CPP based HDR-NN

The CPP implementation used the n-dimensional array data structure feature of Eigen. It shares the same structure as the Numpy based Python implementation with the same learning algorithm show below.

```
/* Mini Batched Stochastic Gradient Descend Algorithm
*
* Reference implementation from
* http://neuralnetworksanddeeplearning.com
*/
void mini_batch_sgd()
{
  // Initialize Nabla matrixes
  std::vector<nabla> nablas;
  for (std::size_t i = 0; i < network.size(); i++)
    nablas.push_back(
          nabla(network[i].weights.rows(),
            network[i].weights.cols())
          );

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

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

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

Part III:
Analysis

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

5. Measurement

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

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

5.1 Benchmark Application Parameters

6. Results

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

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

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

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

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

The accuracy of the model is evaluated after each training epoch on the MNIST test set. After the training of the model for 30 epochs, the final weights and biases of the network and the accuracy on the test set are saved for analysis. This data is used to verify the correctness of the NN model in each benchmark application.

The GNU time program is a great tool for monitoring the performance of applications. It allows us to measure the execution time and peak memory usage, which is used to compare the effectiveness of training the neural network model on the custom hardware implemented with different paradigms.

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.

6.1 Evaluating Correctness

6.1.1 Accuracy

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

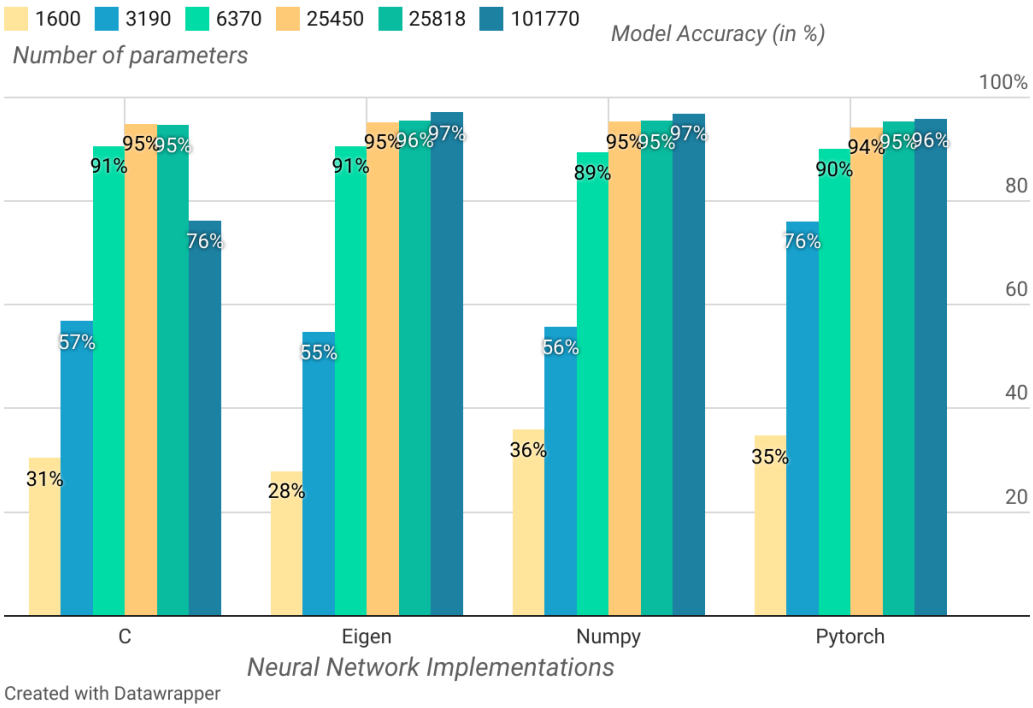


Figure 6.1. Comparing the accuracy of the different HDR-NN implementations.

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

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

6.1.2 Weights and Biases

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

6.2 Evaluating effectiveness

6.2.1 Execution Time

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

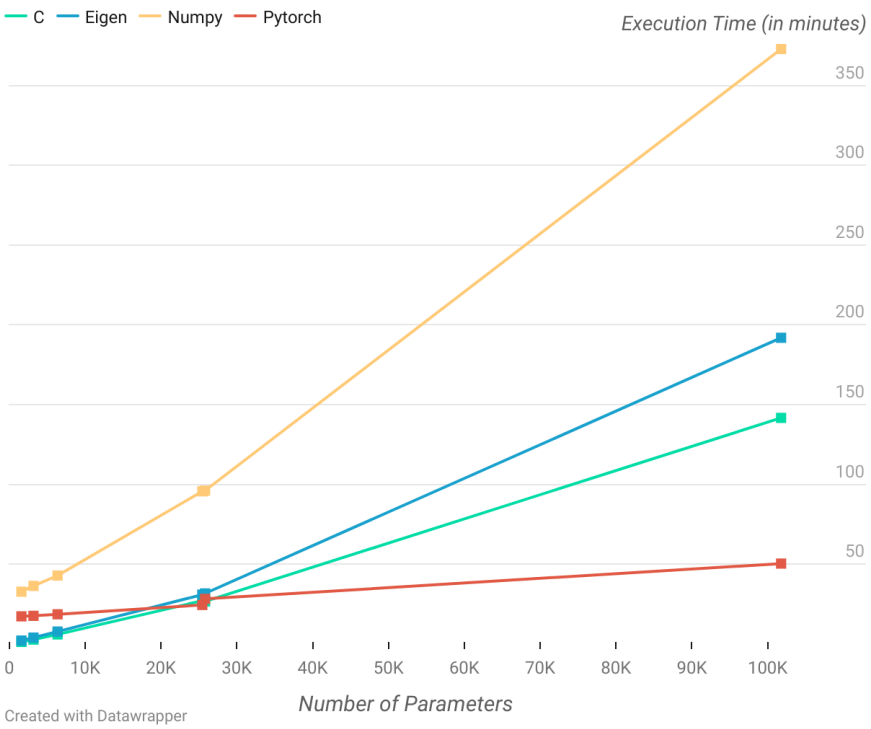


Figure 6.2. Comparing total run time for training the different HDR-NN programs

6.2.2 Peak Memory Usage

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

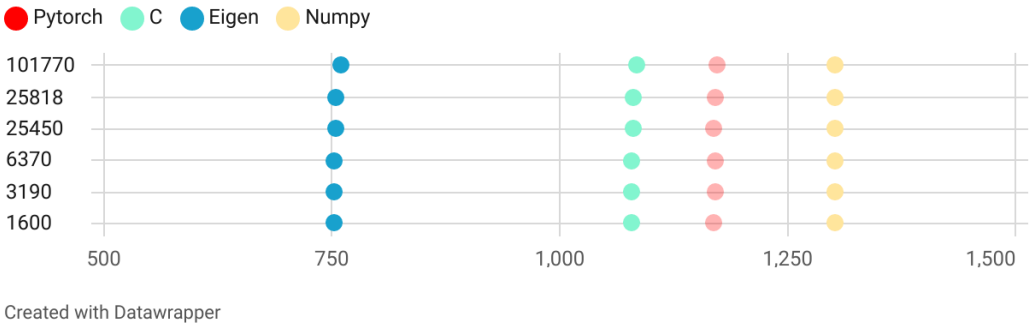


Figure 6.3. Peak Memory Utilized during training with different model sizes remain similar within the same implementation

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

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

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

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

Created with Datawrapper

Figure 6.5. Percentage difference between the implementations

6.3 Comparing the implementations

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

6.3.1 C vs Eigen

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

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

6.3.2 C vs Numpy

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

6.3.3 Eigen vs Numpy

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

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

6.3.4 Profiling

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

6.3.5 Failure/Fault testing

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

6.4 Repurposing Scania ECU

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

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

7. Discussion

7.1 Developer Experience

7.2 Early stopping

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

7.3 General Distribution of Work

8. Conclusion and Future Work

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

References

[1] Meta AI. Pytorch mobile. https://pytorch.org/mobile/home/. [Online; Accessed on April 19, 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 April 19, 2023].

[3] 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.

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

[5] Google. Tensorflow lite. https://www.tensorflow.org/lite. [Online; Accessed on April 19, 2023].

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

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

[8] 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.

[9] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2018. [Online Accessed on April 19, 2023].

[10] 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.

[11] 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 April 19, 2023].

[12] 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 April 19, 2023].

[13] 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 April 19, 2023].

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

Appendixes

Scania C300 Communicator

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

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

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

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

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

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