Prasanth Shaji, Deepak Venkataram

Benchmarking Training of Neural Networks on Embedded Devices

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

UPPSALA
UNIVERSITET

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

Contents

Part I:
Introduction

An embedded system is a combination of hardware and software components put together to achieve a specific task. Often, embedded systems are built into a larger device or system and are used to collect, store, process, and analyse data, as well as to control the device's behaviour. Embedded devices are a category of tiny devices with physical, computational and memory constraints that are programmable to perform dedicated tasks. Like most of the automotive industry, Scania employs embedded systems called Electronic Control Units (ECUs) in their trucks to supervise and regulate essential subsystems like the engine, transmission, braking, and electrical systems. Each of these subsystems has one or more ECUs to gather system data and transmit it to a central communicator where the data is processed and the systems operations are monitored.

Scania currently runs a massive fleet of around 600,000 connected heavy vehicles. The company's truck sales make up 62% of its global sales and Scania has been adding 60,000 trucks to it's fleet annually [5]. This large fleet of rolling vehicles that are connected though the communicators opens up new possibilities. The different connnected devices that monitor the health and state of the vehicle can be trained to predict the system maintainance accurately and efficiently. (For example, if the system can predict accurately that tire changes are required in 100kms then the driver can plan the route smartly to reach the workshop before the vehicle breaksdown.) This opportunity can be realised by running smart algorithms on the hardware that is currently available.

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

One such ML application Scania is has been developing in their LOBSTR [7] and FAMOUS [6] projects is the anomaly and fault detection models in a federated learning environment. Targeting to run the anomaly detection models on the existing ECUs with limited resources has many benefits and challenges. Some of them are described below:

Benefits :

- 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. Achieving this level of performance is computationally demanding, and training NNs on embedded devices requires careful resource management. However, being able to efficiently train NNs on embedded devices with high accuracy could unlock significant potential for future advancements. (could include benefits of NN for 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 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. But these approaches are limited to inference and there is no 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.

A big gap can be observed in the development of frameworks that support the training of NN models on embedded devices and there is a need to investigate the solutions.

Problem Description

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

1. Background

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

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

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

1.1 Development For Embedded Linux

Detail embedded linux development process An overview of these build systems are presented in Appendix I.

1.1.1 Toolchains & Cross compilers

Creating applications that are to be run on an embedded devices requires a set of software components that are usually collectively referred to as 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.

Such software development toolchains consists of a compiler, linker, libraries, debuggers etc to create executable programs for a target device.

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

1.2 Training on Device

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

1.2.1 Federated Learning

link to federated learning, mention FAMOUS again

1.3 Neural Network Applications on Embedded Devices

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

1.3.1 Different Programming Paradigms

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

1.4 General Distribution of Work

2. Theory

2.1 Neural Networks

2.2 Support for Neural Networks in Embedded Hardware

2.3 Embedded Linux Environment

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

2.4 Performance Evaluation

Performance evaluation of programs

Part II:
Implementation

Neural network inference has recieved a lot of attension in Tiny ML. The popular with efforts such as providing neural network application programmers a framework to port. Targetting even smaller devices with Tensorflow based neural network models is possible for inference only applications via Tensorflow Lite Micro [1].

This section contains the description of benchmark training applications created to test the performance of an ANN training cycle on an embedded board. The neural network structure, learning algorithm, and the dataset remain the same but the implementations are completed in traditional general purpose neural network frameworks as well as straightforward implementations in C and other languages

3. Design

The benchmark applications test the training phase of a Handwritten Digit Recognition Neural Network (HDR-NN) on the MNIST [3] dataset. MNIST is a popular dataset of handwritten digits commonly used for training image processing systems. It is a popular starting point for neural network implementations and has been used as the primary dataset in the benchmark experiments. The target embedded device is an Electronic Control Unit (ECU) board based on an iMX6 series processor

3.1 Neural Network Development Process

The target environment necessitates the use of cross compilers and as part of the development process multiple build environments and systems were examined. Ultimately, the primary platform that ended up being used was the Yocto Project extensible SDK (eSDK) based application development process running on a standard linux based build environment

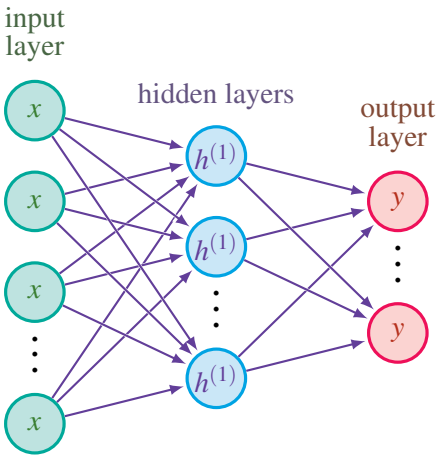
3.1.1 Compiler Toolchains & Yocto Recipes

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

GCC based cross compilers and debuggers were usefull for the C, C++ programs. The general portability of the benchmark applications and the Yocto project allows for further experiments to be conducted on different target architectures as well. For further optimisations that relies on hardware specific features such as ARM's CMSIS-NN cannot be so easily ported however

3.2 HDR-NN Benchmark Programs

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



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

3.2.1 The Learning Algorithm

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

3.2.2 Verifying Correctness

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

4. Development

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

4.1 Target Hardware

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

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

4.1.1 i.MX6 Overview

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

4.1.2 Testing on Device

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

4.2 HDR-NN Implementation

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

4.2.1 The Reference HDR-NN in Python

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

4.2.2 Tensorflow Lite based HDR-NN

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

4.2.3 C based HDR-NN

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

4.2.4 CPP based HDR-NN

Part III:
Analysis

A hand digit recognition neural network (HDR-NN) model is implementated in straightforward approaches such as C, C++ Eigen, Python Numpy and also in a general purpose framework 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.

Furthermore, we address the obstacles encountered in developing the application NN model and compiling it to execute on the target hardware.

(discuss some results regarding the reserve engineering of ECUs)

(efforts into the implementations such as Tensorflow, Tensorflow Lite, Rust, CMSIS-NN which did not yeild the desired results will also be discussed)

5. Results

The benchmark HDR-NN applications implemented in all paradigms have a constant input size of 784 and output size of 10. The hidden layer sizes vary depending on the implementation:

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

The model accuracy, training time, and peak memory usage are measured for each implementation and all its hidden layer settings. This experiment is repeated 10 times and the final values are the average of all the iterations.

5.1 Accuracy

It is not surprising that the different implementations yield similar model accuracy, as the models have the same structure and configurations.

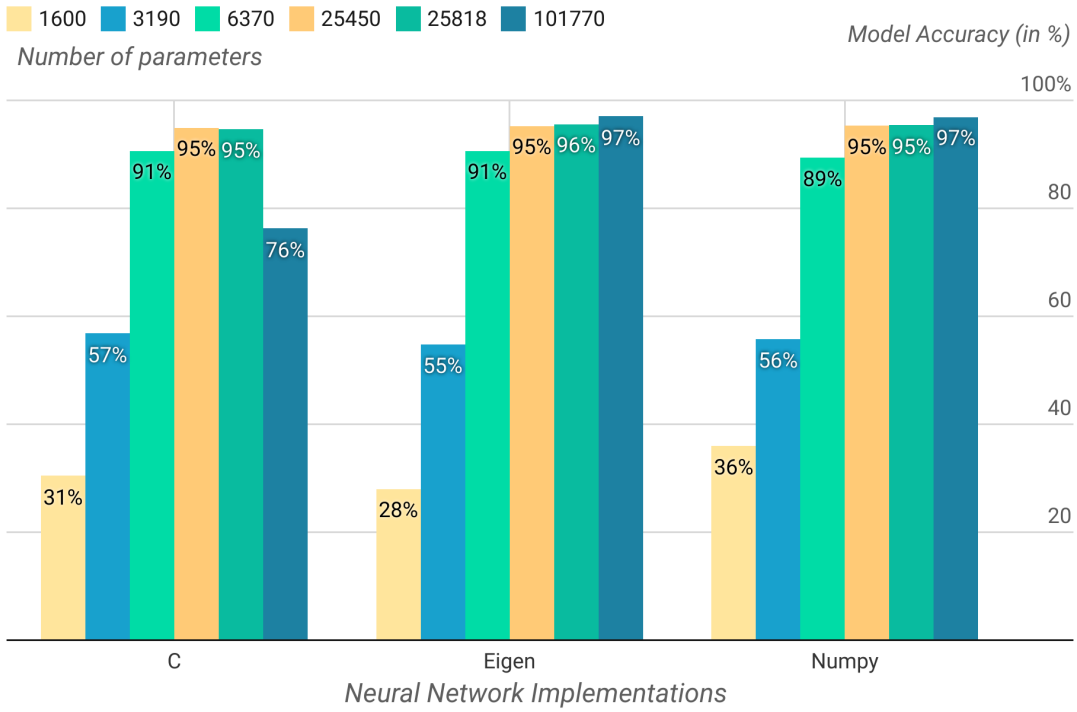


Figure 5.1. Comparing the accuracy of the different HDR-NN implementations.

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

5.2 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.

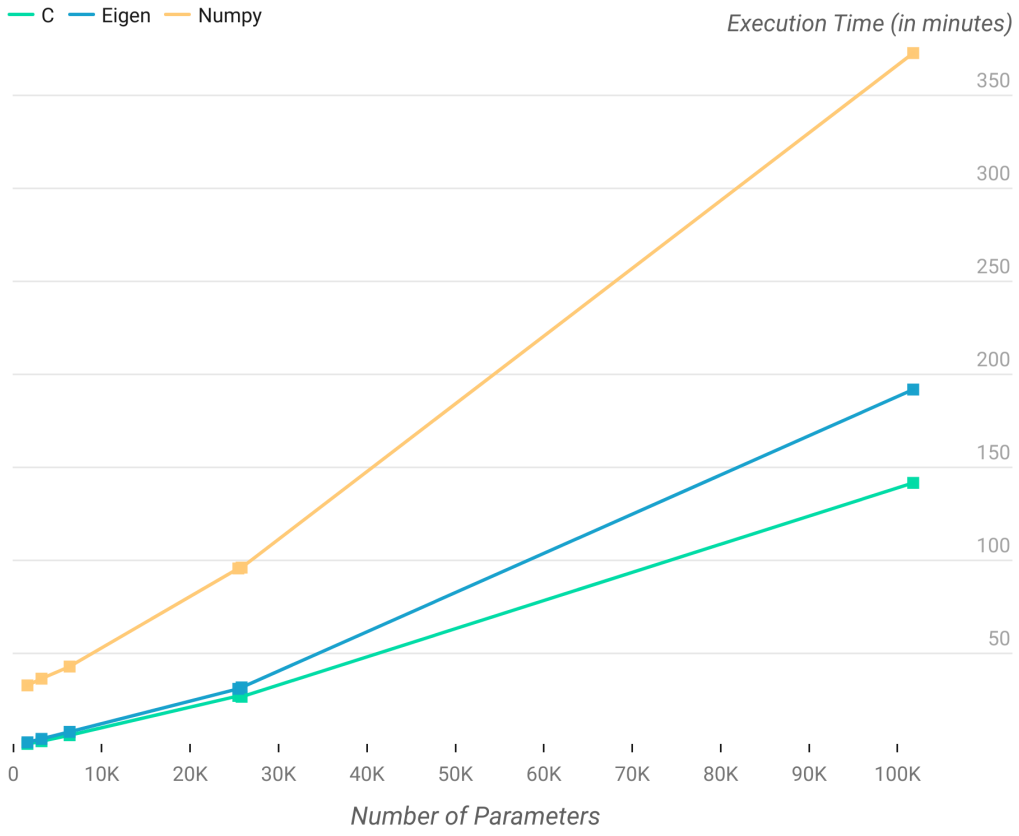


Figure 5.2. Comparing total run time for training the different HDR-NN programs

(percentage or ratio between the different execution time of the different implementations)

5.3 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.

(percentage or ratio between the different run time memory usage of the different implementations)

5.4 Python Numpy based HDR-NN

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

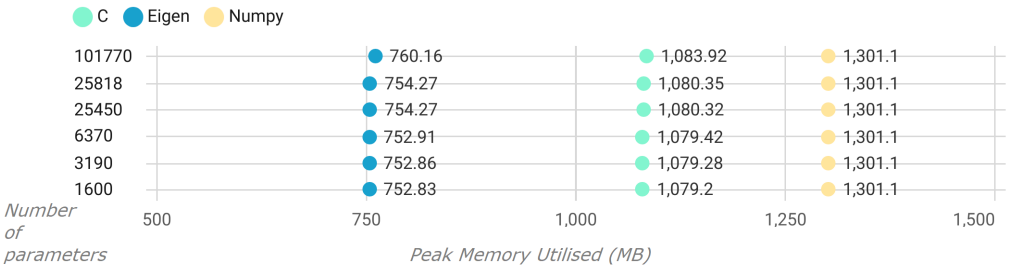


Figure 5.3. Peak Memory Utilized during training with different model sizes remain similar within the same implementation

5.4.1 Tensorflow-Lite based HDR-NN

Benchmark pending . . .

5.4.2 C based HDR-NN

C implementation had lower execution times and memory usage

5.4.3 CPP based HDR-NN

Benchmark pending . . .

5.5 CMSIS-NN based Optimisations to Training

Further breakdown of the performance achieved from different optimisation techniques

5.5.1 Quantisation

future: Training Network with Quantized weights

5.5.2 Pruning the Network

future

5.6 Coefficient of variation

A total of 10 iterations were conducted to ensure that the results remained consistent. To assess the degree of variability among the various trials, the mean and standard deviation were calculated across all runs, and their ratio was determined. This ratio indicates the level of variation between the different tests.

6. Discussion

6.1 Developer Experience

6.2 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. Conclusion and Future Work

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

References

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

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

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

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

[5] 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, 2018. [Online Accessed on March 30, 2023].

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

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

Appendixes

Build Systems

The Yocto Project

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

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

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

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

Scania C300 Communicator

The communicator contains iMX6 series processors