Prasanth Shaji, Deepak Venkataram

Training Neural Networks on Embedded Devices

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

UPPSALA
UNIVERSITET

Short abstract

Contents

Part I: Introduction

Embedded systems are equipped with a combination of hardware and software components 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 systems are common in everyday applications due to their simplicity, flexibility and cost-effectiveness

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

Estimates put the number of tiny embedded systems devices north of 20 billion *TODO: attach a reference here* and the potential of running machine learning applications on that compute is enormous. Furthermore the notion of utilising the existing embedded infrastructure for the purpose of performing ML compute as a means for achiever greater utilisation and as opposed to deploying new specialised devices for those applications has an appeal from a sustainability standpoint. However much of the potential of running machine learning appilations on these devices remain unattained due to the difficulties in creating these applications

Among the approaches that would be salient on these platforms, neural network approaches are the most sought after owing to the unprecedented progress made in their practical applications. Tiny Machine Learning (Tiny ML) is a burgeoning field that looks at how this space of embedded devices can be made more suitable to create and explore the potential machine learning applications that it can support. An important feature of machine learning applications are their iterative improvement process. For neural network applications this happens during the training process which traditionally consumes a lot of compute resource

Why perform training and inference on ECUs?

In the world of embedded systems resources such as compute, memory, network bandwith etc. are all limited. The traditional model of sending data from embedded device sensors off-board to compute clusters on the cloud presents several challenges such as bandwith consumption, and privacy considerations making it attractive to move training on-board

Federated Learning

One approach to making this training loop take place from within these platforms is Federated Learning which cruicially allows for the data to remain on the device

1. Background

Hypothesis: Training and inference of (small) neural networks in embedded systems can be considerably improved compared to general purpose neural networks frameworks

The space of salient applications for automotive embedded systems is enormous. A classic example is anomaly detection within an automobile, a subcomponent of the automobile, or with the interactions between subsystems of the automobile

- *Introduce how an anomaly detection application could be run*
- *Introduce general information about artificial neural networks (ANNs), MLOps, etc - state that there is more information in the Theory chapter*
- *Include literature study elements from Federated Learning*

1.1 Anomaly Detection On Board

Introduce how the ANN application would be executing on the automobile

1.1.1 MLOps On Embedded Systems

Nature of (CAN) data generated on ECU systems and how they could be consumed - described from an MLOps viewpoint

1.1.2 Considerations Of Embedded Environments

- *State hardware requirements within the context of the ANN functionality - specifically contrast training with inference*
- *State intent to benchmark the training phase and motivate*

1.2 Development For Embedded Linux

Introduce build systems for embedded linux. Motivate the section in terms of targetting embedded hardware

1.2.1 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

1.2.2 Toolchains & Cross compilers

Describe their usage. Will show up again in Development chapter

1.3 Development Of Neural Network Application

Contrast general purpose frameworks - TFlite etc with handwritten applications

- *Include literature study elements from Tiny ML*

1.3.1 Different Programming Paradigms

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

2. Theory

2.1 Artificial Neural Networks

General introduction to ANNs. Explaining topics from inference, training, till federated learning systems

- *Explain the different ways of building out the ANN applications - Training on board vs off board, associated factors such as uploading data vs learned model*
- *Compare training with inference*

2.2 ANN Performance Optimisations Techniques

Contrast traditional implementations in resource rich environments and the constraints of embedded environment. Layout general strategies to acquire performance improvements with little losses to accuracy - Purning, Quantisation. State the emphasis on training

2.3 ARM's CMSIS-NN

Introduction to ARM CMSIS-NN kernels, mentioned again in Development chapter. Refer to the CMSIS-NN paper

2.4 ARM based Embedded Linux

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

2.5 Performance Evaluation

Describe and motivate performance measures used in the Results chapter

Part II: Implementation

ANN training presents an important gap in the current efforts in Tiny ML. This section contains the description of benchmark ANN 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 (HDRNN) on the MNIST dataset. MNIST is a popular database of handwritten digits commonly used for training image processing systems. It is a popular starting point for neural network implementations and has been used as the primary dataset in the benchmark experiments. The target embedded device is an Electronic Control Unit (ECU) board based on an iMX6 series processor

3.1 ANN Development Process

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

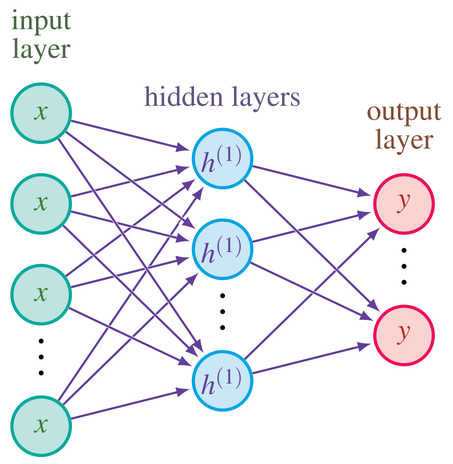
3.1.1 Compiler Toolchains & Yocto Recipes

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

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

3.2 Benchmark ANN - HDRNN

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 HDRNN 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. To verify their mutual correctness, all the implementations initialize HDRNN using the same PRNG

4. Development

The HDRNN 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 Targetting an i.MX6 based custom board

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 HDRNN 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 neuralnetworksanddeeplearning.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 HDRNN

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 HDRNN

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

4.2.4 CPP based HDRNN

4.3 CMSIS-NN based Optimisations

4.4 General Distribution of Work

Part III:
Analysis

Results from the implementation

5. Results

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

5.1 HDR-NN comparisons

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

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

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

5.1.1 Python Numpy based HDR-NN

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

5.1.2 Tensorflow-Lite based HDR-NN

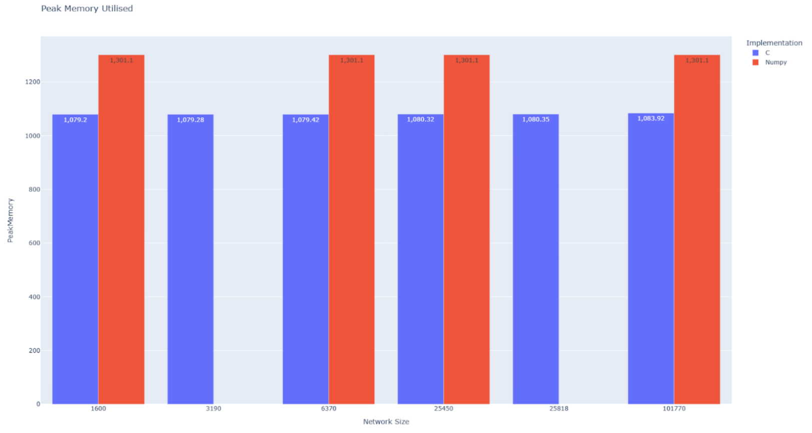
Benchmark pending …

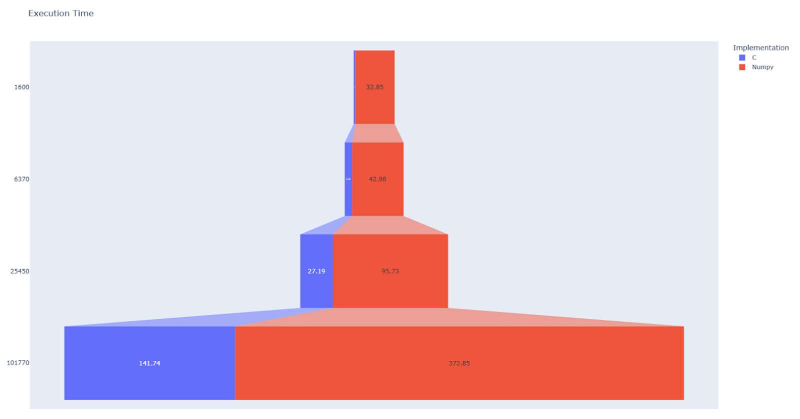
5.1.3 C based HDR-NN

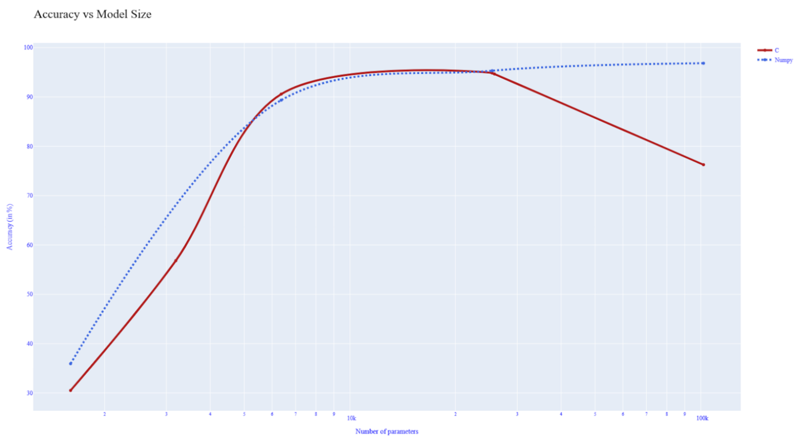
C implementation had lower execution times and memory usage

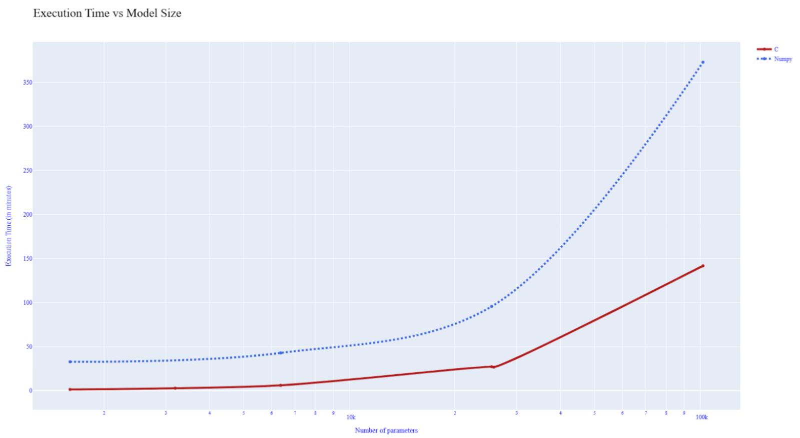
5.1.4 CPP based HDR-NN

Benchmark pending . . .









5.2 CMSIS-NN based Optimisations to Training

Further breakdown of the performance achieved from different optimisation techniques

5.2.1 Quantisation

future: Training Network with Quantized weights

5.2.2 Pruning the Network

future

6. Discussion

- *Contrast development process for the ML programming paradigms*
- *Which optimisation approaches gave the most in improvement?*

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, 2020.