Profiling and optimization of Deep Neural Networks for embedded automotive applications

Eric Perraud¹, Loïc Cordone², and Jean-Marc Gabriel¹

Keywords — Deep Neural Network – Convolutional Neural Network - Profiling - Optimization – Embedded systems

I. Introduction

Over the past decade, deep neural networks (DNNs) have achieved dramatic accuracy improvements and are now used in a vast range of application domains, from object detection [9] to speech recognition [11]. Car manufacturers are now considering using deep neural networks (DNNs) primarily (but not only) in their advanced driver-assistance systems (ADAS) and ultimately in their autonomous driving (AD) solutions.

Nowadays, several open source frameworks offer building modules for designing, training, validating and sharing deep neural networks through a high-level programming interface (generally Python). We can cite TensorFlow [13], PyTorch and MXNet [14]. It makes sense for industrials which intend to use DNNs to leverage these frameworks: it allows the engineers to focus on searching for the best parameters of the DNN (rather than spending many valuable resources in implementing and training a DNN), it also allows them to rapidly reuse DNN models, to adapt them to their needs, thus reducing the time to market and the heavy cost of Research and Development.

However, integrating such algorithms in automotive embedded systems (the electronic control units, ECUs) represents an industrial challenge:

- The automotive manufacturer must integrate some software component they don’t control,
- The embedded CPUs has limited computing power, and the available electrical power is also limited,
- The latency is highly critical in many embedded applications, meaning that the inference shall be time-bounded,
- For mass market manufacturer, the cost of a DNN must be highly minimized.

It means that tuning a public DNN for automotive requirements within an open source framework may not be sufficient to guarantee a smooth integration in a shippable product. Additional information which is not provided off-the-shelf by frameworks is needed to facilitate the integration of DNNs in embedded systems.

In this paper, we investigate an approach combining deep learning profilers and deep learning compilers to:

- Assess the inference latency ³,
- Determine where the optimization effort (if needed) should focus,
- Compile the model for a fast and lightweight inference on the target ECU.

This paper can be considered as a first step toward a global methodology for dealing with embedding DNNs onto hardware with limited resources.

II. Scope of the study

To implement complex DNN in embedded solutions, the silicon manufacturers propose different types of architecture: multicore CPU (either ARM based or Intel based) or GPU. We are also witnessing an emerging trend in the use of custom hardware platforms, such as field-programmable gate arrays (FPGAs) and

¹ Renault Software Labs, Toulouse
² ENSEEIHT, Toulouse
³ It is assumed that the training step of a DNN is done offline, during the manufacturing. Therefore, it can be done with very powerful servers and the training time is not critical.
application-specific integrated circuits (ASICs) to accelerate the inference, with results of higher throughput and energy efficiency using neural networks compression techniques [12]. It is still unclear which HW architecture will be preferred by the automotive industry, and it is not even clear whether a given architecture will win as the universal HW solution for embedded DNNs. As shown in this paper, the proposed approach works on any type of HW architecture. We intended to validate it on two configurations: a CPU configuration with an Intel Xeon E5-2690 v2 and a GPU configuration with the same CPU and an NVIDIA RTX 2060 6GB. These high-end products are today too expensive to be implemented in an automotive application; but they allow to validate the approach we propose to deploy on both a CPU-based architecture or a GPU-based architecture. We have validated the approach with different types of DNN. We have considered three representative classes of neural networks which can be embedded in automotive systems: fully-connected deep neural networks (FC-DNNs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). FC-DNNs, also called multi-layer perceptron (MLP), can typically be used for a variety of small functions such as fault/intrusion detection, nonlinear regression, or classification. CNNs can typically be used to achieve a better understanding of the vehicle surrounding environment. They are used in a multitude of computer vision applications processing camera images such as character recognition, object detection and recognition (cars, signs, pedestrians, bicycles) or scene segmentation (identify and localize roads, sidewalks). RNNs are more suited to problems involving time series, such as trajectories prediction or voice recognition. They are quickly described below.

a. SteerDNN: nonlinear regression

We developed an automotive function to predict the front-wheels steering angle to keep a car at the center of its lane. The objective was to develop a solution faster than the current physics-based solution, while keeping a good accuracy. The neural network model must have a latency of 10ms, from the data ingestion to the output angle. We worked with a Renault dataset containing 6,000 kilometers of real driving data.

We used TensorFlow to design, train and validate our models. The final model is a FC-DNN with 13 inputs (speed of the vehicle, transverse and longitudinal acceleration, position of the left and right lanes output by the Line-Keeping Assist (LKA) system), 4 layers of 64 neurons and 1 output, the steering angle.

After the training, the model achieved an MSE of 0.28° on our validation test, an accuracy we considered as good enough.

b. Mobilenet V1: object detection and recognition

Another automotive function that would be implemented in ADAS is object detection and recognition. We used a pre-trained state-of-the-art model, MobileNet V1 [7] + SSD [8]. This convolutional neural network provides an excellent accuracy (COCO AP@[.50:.05:.95] = 19.3%) while being composed of three times fewer operations than its competing models (1.2 billion mult-adds vs. 3.8 billion for Inception V2 [15]).

We have recovered the model trained with MXNet.

c. CS-LSTM: path prediction

To support and improve decision making and path planning, a recurrent neural network was developed internally to produce multimodal trajectory predictions over 0s-5s horizon. The model is based on a publicly available model (CS-LSTM [16]) and was implemented and trained in PyTorch. The network places the car in the center of a 13x3 space grid, and takes as input the past positions / speeds of the car and surrounding cars (up to 38 neighboring cars).

III. Profiling

Profiling refers to measuring the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

In the deep learning field, the programs are not directly written by hand, most models are trained and executed inside a framework. Therefore, a low-level profiling would be useless, since it is not easily possible to change the way the framework executes our code.

On the other hand, it is useful to profile at a high level: to measure the inference time, the memory consumption, the frequency and duration of the framework functions calls. This information allows to identify the bottlenecks of the full model itself, and to suggest optimization guidelines in order to rework the
model inside the framework environment. If the model can’t be modified, a second phase of optimization shall be pursued, on the bottlenecks identified. This optimization pass is presented at length in part IV.

This profiler measures approximately the duration of each operation and on which core they were placed. The total duration was longer, approximately 1.2ms, due to the profiling overhead.

It is possible to gather these measures with what is called “deep learning profilers”, which are profilers available in each deep learning framework.

### a. Deep Learning profilers

Deep learning profilers all work the same way: one shall insert profiling callbacks directly in the source code, and the profiler will register a profiling trace (for one step of training, or for one inference) during the next execution.

These profiling tools lead to overhead: registering the beginning and the end of each operation is intrusive and may lead to longer operations during the profiling phase. However, it is not a problem since we are not looking for very precise measures, but rather for a relative comparison of the different operation durations.

Each of the major deep learning has its profiler: TensorFlow/Keras, PyTorch/Caffe, MX-Net. They all register the duration of the operations and on which device (CPU, GPU) they are executed. They all rely on the tool chrome://tracing, a frontend for inspecting and analyzing profiling events. For each operation, the frontend indicates where it was really executed (which thread for a CPU, which stream for a GPU) and how much memory was requested. However, we found out that TensorFlow Profiler was the most user-friendly, being directly integrated within the TensorBoard.

In the following sections, the inference time represents the mean time of 500 identical inferences, after 50 dry-run inferences, and for the same random input.

### b. FC-DNN

First, we measured the inference time: around 1ms on CPU. We then profiled the FC-DNN inference with TensorFlow Profiler.

By analyzing the operations and their durations, we can divide the inference in three distinct parts: a) the memory read and parsing of the input, b) the data transformation (also called preprocessing), and c) the actual FC-DNN traversal. It is illustrated by Fig.1: each row represents a thread.

In the a) part, TensorFlow is reading the inputs and the network’s weights from the memory, then it parses the inputs to feed them to the TensorFlow network. This step is done sequentially on one core and takes approximately 50% of the inference time. The b) part represents the preprocessing of the 13 inputs, which is a simple normalization (subtraction of training data means and division by the standard deviation). These operations, seemingly simple, represent 40% of the inference time. Finally, the actual network traversal represents less than 10% of the inference time. Our conclusions were the same when we modified the total number of neurons (from 32 to 256).

The profiling proves that for a FC-DNN developed in a framework, the actual network traversal represents a small fraction of the inference time. For an embedded device using such a fully-connected neural network, the inference optimization should focus on the data pipeline, not on the network, and is out of the scope of our work.

### c. CNN

We then profiled our object detection and recognition convolutional network, using MX-Net profiling callbacks and the chrome://tracing tool (fig.2). It is important to note that in MX-Net, the memory operations (green row) are separated from the computation operations (pink row).

We measured an inference time of 60ms (16 frames per second) on our CPU. By looking at the profiler output, we can...
divide the inference in three distinct parts: a) image resizing, b) the data preprocessing), c) the CNN traversal. The most time-consuming operations are the 27 convolutions (plain convolutions and depthwise convolutions [7]), which take about 65% of the inference time: the convolution operations take nearly 35% of the inference time and the batch normalization following each convolution, takes about 30% of the inference time.

The profiling traces also show that the convolutional operations are mainly processed by only one CPU and are not parallelized over multiple CPUs. Therefore, the convolutional operations make a poor usage of the available CPU resources. We had a similar when we profiled it within TensorFlow: the convolutional operations take most of the inference time and are not parallelized unlike the post-processing stage (Non-Max Suppression).

Thus, on CPU, the bottlenecks clearly are the convolutions. This model being state-of-the-art, we cannot easily remove or replace these operations. We noticed similar results on GPU, the convolution operations being the most time-consuming operations, even though the inference was much smaller (around 12ms).

MobileNet+SDD is a more complex network than an FC-DNN, extracting the weights of all tunable operations and implementing the inference in a low-level language is not trivial, and not recommended. If it is desired to reduce the inference time, it is then necessary to undertake an optimization phase which is focused on the convolution operations.

**d. RNN**

We then measured the duration of the operations during an inference of our internal RNN, using this time the PyTorch callbacks. The generated profiling trace is rather similar to the MX-Net one. But PyTorch offers a quick view of the most time-consuming operations, presented in Table 1.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>CPU total time (ms)</th>
<th>CPU total %</th>
<th>Number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>addmm</td>
<td>27.323ms</td>
<td>45.79%</td>
<td>335</td>
</tr>
<tr>
<td>sigmoid</td>
<td>6.172ms</td>
<td>10.34%</td>
<td>498</td>
</tr>
<tr>
<td>tanh</td>
<td>5.888ms</td>
<td>9.87%</td>
<td>338</td>
</tr>
<tr>
<td>mul</td>
<td>3.787ms</td>
<td>6.35%</td>
<td>515</td>
</tr>
<tr>
<td>add</td>
<td>3.742ms</td>
<td>6.27%</td>
<td>349</td>
</tr>
</tbody>
</table>

The network being mainly composed of LSTM cells and fully-connected layers, the matrix multiplication and addition functions are the most time-consuming, representing 60% of the inference time.

It is also interesting to note that the activation functions (sigmoid, tanh), are present inside each LSTM cell and after a fully-connected layer, amount to 20% of the inference time.

A fast way to reduce the inference time could be to use cheaper activation functions, such as hard sigmoid and hard tanh, or to change the LSTM cells for GRU cells, which are computationally cheaper. These changes would require retraining the model though, possibly with a precision loss.

**e. Profiling conclusions**

Thus, depending on the studied model, the focus shall be put on the data ingestion, on optimizing the model without modifying it or on changing some of the model’s operations.

Profiling the inference can thus suggest some guidelines for reducing the inference time directly within the framework. If the model can’t be modified (because it is a state-of-the-art model already trained), the profiling has identified the bottlenecks, which can be optimized thanks to a variety of new tools.

In particular the optimization guidelines may be fully different. As a matter of fact, we have shown that:
- For the analyzed FCC-DNN, the data preprocessing is the highest contributor in the overall inference budget and it is where the optimization should be focused if needed,
- For the analyzed CNN, the traversal of the convolutional layers is the highest contributor in the overall inference budget,

IV. Optimization

Optimizing a neural network means reducing its number of operations or changing the way they are performed. In this section, we will present the techniques currently implemented within the frameworks, and another approach based on the compilation of neural networks.

a. Different levels of optimization

In deep learning frameworks, the operations constituting a neural network are represented as a graph, called a dataflow graph. The nodes represent compute units, and the edges represent the data consumed or produced by the nodes. The dependencies between compute units are thus explicitly defined.

Each operation in the graph has an implementation, different for each hardware.

We can therefore define three scales of optimizations:

Model optimization. The techniques that directly modify the structure of the network, or the representation of the data within the network. This includes network pruning techniques where unnecessary (low-weight) neurons are suppressed and quantization-based neurons, where the idea is to encode network weights on a reduced number of bits. Encoding the 8-bit instead of 32-bit weights reduces the memory footprint by 4x ratio, and up to 4 times the inference time (if suitable hardware instructions exist) [6]. These techniques directly modify the neural network, and thus modify the accuracy of it. The gains can be high but require some expertise and shall be performed by skilled engineers. Some documents even argue that pruning is ineffective [7].

Graph optimization. Graph representation allows neural networks to take advantage of all the existing mathematical graph optimizations. It includes graph simplification (reducing dependencies, minimizing data transformations), and merging operations.

Operation optimization. The implementation of an operation can be optimized considering the specificities of the target hardware. An operation will be parallelized differently on a graphics card (thousands of hearts) and on a processor (a dozen hearts at most). Memory management will also be different depending on the size of the memory caches for the processors, and the amount of video memory for the graphics cards. Such optimizations would need very skilled engineers who have a very deep understanding of the target hardware architecture and of the mathematics behind the DNN operations.

The operations optimizations are the source of many neural network operations libraries proposed by the hardware manufacturers: the NVIDIA cuDNN library, the Intel MKL-DNN library and the ARM ComputeLib library. Operations are optimized for the target hardware, but they are opaque and proprietary. The execution of a neural network by a framework therefore consists in calling the optimized operations specific to each hardware, as shown in fig.3, and then in optimizing the graph.

But the optimized operations are not very scalable, while the operations composing the neural networks evolve at a high speed.

This split into proprietary calls also strongly limits the graph optimizations, where it becomes impossible to merge several operations if each operation makes a different call.

Optimized operations do not consider the differences in the characteristics between two hardware of the same family. So, an operation will be executed in the same way on a low-end NVIDIA GPU and on a top-of-the-range NVIDIA GPU, even as the GPU high-end has twice as many cores and four times more memory. Finally, in the case of a development
on an exotic hardware (FPGA, ASIC), it becomes impossible to use the model developed under a framework, since there are no libraries of operations.

All these reasons have led many researchers to develop compilers specifically dedicated to neural networks, usually called deep learning compilers or deep learning inference engines.

b. Deep learning compilers / inference engines

The idea of the deep learning compilers is to recover an already trained model of a framework and compile it for a target hardware to execute the inference phase. The compiled model will be fixed but lightweight and ideally optimized for the target hardware. Also, the compiled model will no longer need the framework to run (only a light runtime, i.e. a minimal framework capable of running the compiled model) and can be integrated into programs in any language.

For a little over a year, compilers (with integrated optimizations) have appeared in various manufacturers and major IA companies: NVIDIA TensorRT, Intel nGraph [19], TensorFlow XLA [20] (Google), Glow [21] (Facebook), TVM (Amazon) [4]. We are therefore seeing a real enthusiasm of the different actors for this approach, and all these compilers have excellent results. But apart from Glow and TVM, they remain hardware-dependent (software-dependent for TensorFlow XLA) and not very scalable. Glow, the Facebook compiler, is not available yet.

All inference engines optimize models on the three scales presented previously: they offer optional functions for doing quantization, they perform an extensive optimization of the graph (fusion of operations, transformation of the data, management of the memory) and optimize each operation using code generation.

In our case studies, we limited ourselves to the optimizations of the graph and the operations, to keep the network performances unchanged.

We chose to use TVM, an open-source Deep Learning compiler for specialized CPUs, GPUs, and accelerators (FPGAs, ASICs) which aims to bridge the gap between the productivity-focused Deep Learning frameworks and the hardware backends (CUDA for NVIDIA GPUs, LLVM for Intel CPUs) based on performance or efficiency.

TVM can compile deep learning models in Keras, MXNet, PyTorch, Tensorflow, CoreML, DarkNet into minimum deployable modules and sets up an infrastructure to automatically generate and optimize tensor operators on more backends with better performance than the handcrafted operations libraries (such as cuDNN, etc.). More details can be found in the paper [4].

V. Results

Using the profiling results, we compiled and optimized our models with TVM. We did not optimize the FC-DNN because, as we have seen, the network traversal only takes a small part of the inference time.

In TVM, the whole model is compiled, but it is possible to define specific optimizations for an operation, depending on the target hardware.

![Figure 4: Inference time before and after optimization, on CPU and GPU.](image-url)
The profiling enables targeting these operations.

a. CNN

We optimized our CNN Mobilenet+SSD for a CPU target and for a GPU target. We used the convolution optimizations available in TVM, for both targets. These optimizations change the implementation of each convolution operation according to the hardware specificities (number of threads, cache memory size, video memory available). On CPU, this optimization step searches for the optimal trade-off between minimizing the cache miss rates, improving the register stationarity of the vector instructions and the level of parallelism. The optimization of the convolution operation for x86 CPUs can be found in [17], and for NVIDIA GPUs in [18].

We present the results obtained in the fig.4. We measure the inference time when the model is executed within the framework but without any optimization, when the model is compiled with TVM and when the model is compiled with TVM with hardware-specific optimizations for the convolution operations.

We observed a division by 2 of the inference time, on both targets. More precisely, an is 40% shorter after TVM optimizations, which here allow to process 21 frames per second instead of 13 frames per second.

Deep learning compilers such as TVM, Intel GraphN and NVIDIA TensorRT can therefore reduce their inference time without altering their precision of such CNN.

b. RNN

We then optimized our RNN for a CPU target. TVM provides optimizations for the dense layers, but not for the other operations. We first compiled the model, and then applied the specific optimizations on the dense layers. The results before and after compilation are presented in fig.5, for each number of neighboring vehicles (cf section II.C). The architecture varies with the number of neighboring vehicles, which leads to 39 different models.

For each model, the inference has been divided by 5 after compilation. The worst-case scenario, where there are 38 neighboring vehicles, induces an inference time of 36ms in PyTorch, against only 8ms after compilation with TVM. It is worth noting that the specific optimizations on the dense layers contribute only to a negligible reduction in inference time, of around 1%.

Thus the compiler allowed us to achieve excellent results here due the diverse nature of operations composing our model, without impacting the accuracy of the DNN.

VI. Conclusions

Deep learning frameworks are promising tools: it is now possible to use them for developing intelligent embedded functions and they take care of optimizing the executed code to the target hardware. The presented approach combined the analysis of the high-level profiling trace to identify whether the
optimization of the network itself is relevant and to identify bottlenecks or conception errors with the use of deep compilers to optimize the model for a fast and lightweight inference on a diversity of hardware. It supports a complete separation between the DNN design and its porting on embedded systems.

It also allows to resolve some conflicting requirements of many industries, aka how to adapt state-of-the-art DNN algorithms to hardware targets which have limited computing power and memory, without degrading the accuracy performances of these algorithms.

More work will be achieved in order to plug all the observations on a more global methodology supporting the engineering teams in using both profiler and compiler in an efficient way.

References
[18] "How to optimize convolution on GPU", https://docs.tvm.ai/tutorials/optimize/opt_conv_cuda.html