Automated generation of Single Shot Detector embedded C library from a high level Deep Learning framework

Luca Ranalli, Luigi Di Stefano
CVLab - DISI, University of Bologna
Viale Risorgimento, 2 - 40135 Bologna, Italy
luca.ranalli@studio.unibo.it
luigi.distefano@unibo.it

Emanuele Plebani, Mirko Falchetto, Danilo Pau,
Advance System Technology,
STMicroelectronics,
Agrate Brianza, Italy
name.lastname@st.com
luca.ranalli@st.com

***Abstract*— When creating accurate high-performance artificial neural networks (ANN) topologies for embedded systems, weeks of engineering work are required using off-the-shelf python level deep learning frameworks to write new optimized layers both in term of memory and operations and test them in proper-scale experiments. Then, code implementing the validated layers needs to be mapped onto embedded systems, which requires many other months of engineering work. To shorthen this un-productive and long development procedure, an automated procedure was created and used. This work presents all the phases involved in the automated implementation of a Single Shot object Detector (SSD) trained model for an embedded mapping on which power consumption is a key point of focus and memory usage needs to be minimized. The work is focused on the implementation aspects dealing with mapping high-level functions and dynamic data structures into low-level logical equivalents (Ansi C). A brief explanation about all the validation measurements as well as a short summary on how memory usage is linked to the details of the considered detector is also presented.**

Keywords: Deep Learning, Neural Network, Single-Shot Detector, Ansi C automatized generation**.**

I. INTRODUCTION

Deep Neural Networks (DNN) are widely deployed in a growing number of applications dealing with different kind of sensors (inertial, environmemental, audio, imaging etc). DNN can solve problems related to classification, detection, recognition, analysis and, more recently, even to generation of synthetic signals in computer vision, signal processing, speech and audio applications, robotic motion, navigation, financial data analysis, medical diagnostics, and many more are appearing. When creating and developing DNN, developers face huge gaps and problems in targeting embedded systems in terms of productivity of software development, interoperability of file formats between cloud and embedded platforms and best usage of constrained memory and speed computational resources. We propose to address these issues by a new tool that automatizes the generation of C code from high level deep learning tools.

Thus, our tool makes it automatic and very fast the conversion of any pre-trained DNN into a memory and computationally efficient optimized library for a micro-processor to execute. Eventually, the library can be integated and executed in a vertical application.

II. CLIENT-SERVER TOOL

The tool is primarly meant to ease the mapping of a DNN into an embedded system featuring an ANSI C compiler such as as it is the case of every modern micro processor. The tool starts from a pre-trained DNN model created by a broad range of popular off the shelf deep learning frameworks (e.g. Caffe, Keras, Lasagne, etc), and maps it to an optimized DNN that is adapted to the memory and processing-power capabilities of a target micro processor. It features an efficient code generator system that can pack framework-dependent, pre-trained DNN into compact libraries. This system allow users to train their DNN with any of the most used deep learning frameworks so to produce high-performance codes that the user needs, automatically and on-demand. (An example of the flow is shown in Fig. 1).

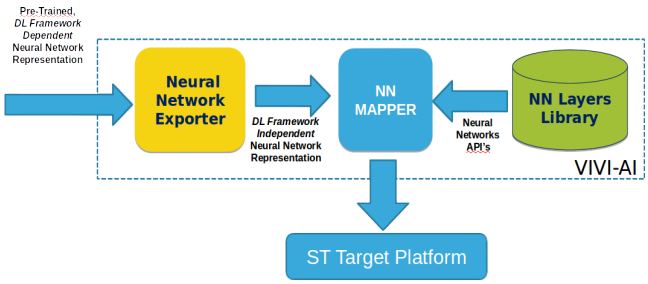

Figure 1 - Tool Architecture Overview

III. SSD

SSD is an object detector based on a single DNN. It includes a partial VGG-16 (truncated before any classification layer) as base network plus a set of auxiliary convolutional layers. This paper will not go in depth for what concerns the network training as well as data augmentation techniques as they can be found in the original paper [1]. We will sum up the workflow when inferring it while focusing on the elements that characterize its already trained implementation.

A. Why SSD

As stated in J. Huang et al. [2] , in which the authors compare the latest detectors meta-architectures over different feature extractors combination, SSD outperforms the other considered meta-architectures in terms of speed and memory usage over accuracy.
Moreover, it turned out that it is almost as accurate as Faster R-CNN [3] when detecting large objects while running much faster. On the other hand, being SSD performance even less sensitive to the quality of the feature extractor with respect to Faster R-CNN [3] and similar architectures, it turns out to be the best meta-architecture option for an embedded context, allowing to replace the feature extractor with a faster and lighter network.

IV. THE OBJECTIVE OF THIS WORK

The goal is to start from the original Caffe (C++ and Python) implementation of the network provided by the authors with some examples and reproduce the end-to-end behavior of the network by rewriting the interested and missing layers into building blocks to extend the Neural Network Internal Library of the tool shown in figure 1. Being the SSD network made out mainly of convolution layers that are well known and already implemented into the library, as well as data concatenation and reshape operations, the focus is on the logical key layers that characterize the model: the *Box Generation Layer* and the *Detection Output Layer*.

A. Box Generation Layer

This layer is responsible for the generation of the default boxes over different feature map shapes. It takes a set of parameters of the network such as the list of aspect ratios of the boxes to generate, scale factors, etc. All these parameters do affect the output tensor shape, but it can be estimated from the model allowing to statically allocate the data structures (memory usage grows linearly with the number of boxes to be generated). Under a practical viewpoint, this layer check the shapes of its input to discriminate over which layer output it is generating boxes and outputs a tensor containing in the encoded coordinates of the boxes for that feature map.

B. Detection Output Layer

While the implementation of the previous layer is pretty straightforward, due to the minimum presence of dynamic data structures as well as complex data manipulation, this is not the case of the considered layer. Before getting into deep implementation details, let us describe what it does intuitively.
This layer takes as input 3 tensors that are the concatenation results of all the predictions and default boxes which we will call: *Confidence Tensor* - containing the predictions for the classes; *Location Tensor* - containing the offset predictions for the boxes; *Boxes Tensor* - containing the default boxes.

The *Boxes Tensor* has shape *[BATCH IDX, 2, N BOXES*4]*, contains 4 coordinates for each default box calculated by the *Box Generation Layer* and a variance for each coordinate.
The *Confidence Tensor* has shape *[BATCH IDX, N BOXES * N CLASSES]* and contains a confidence score for each box prediction relative to each class. The *Location Tensor* shape depends on the value of the share flag, if **true** the shape is *[BATCH IDX, N BOXES * 4]*. It can contain either a list of offsets relative to each default box or (if share is **false**) 4 offsets for each class and for each default box. See Figure 2.

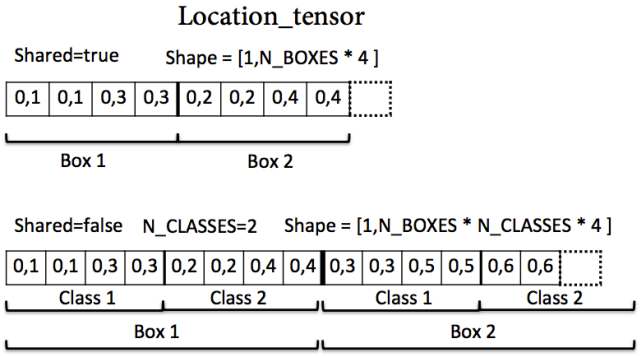

Figure 2 – Location tensor data order

Furthermore, the layer needs some other values as internal parameters. The original C++ Caffe code, organizes all the described data into maps of vectors clustered by class (key=class), hence, for each class, stores a vector of boxes, a vector of confidences relative to each box, a vector of localization regressions relative to each box. Then, for each default box, it computes the final boxes prediction by multiplying the regressions with the variance and by adding the result to the default generated values. After that, all the computed box predictions are filtered with a Non Maximum Suppression strategy.

C. Non Maximum Suppression

For each class, we keep all the indexes of the boxes which have a corresponding confidence greater than the **confidence threshold**, we then filter them by their Jaccard overlap factor. So, if specified, we might want to take the previous results, sort them by confidence score, and keep the K best detections. Once done, we're able to prepare the output tensor with data in the form *[BATCH IDX, CLASS, CONFIDENCE, X MIN,Y MIN,X MAX,Y MAX]*, hence, 7 float values per detection will be used.

D. Integration Testing

In order to test the layers with meaningful data and check their behavior in the network pipeline, after the layer passed the translated caffe unit tests, we used a pre-trained SSD caffe model example. We extracted the model parameters that are meaningful for our layers and edited the original python script to redirect to file intermediate data tensors before and after the layers to test. After that, an ad-hoc test has been created.

It fills out input tensors with the collected data stored on file (4 files, approximately 300k elements) triggers the layer, and checks the expected output elementwise. The layers turned out to reproduce the original behavior without any loss in precision.

V. CHALLENGES AND RESOURCES USAGE

When reproducing the Detection Output layer, the naive approach of mapping each function and each data structure onto its C equivalent (dynamic array of structs containing arrays), turned out to be very complex and introducing too much overhead. Therefore, we opted to represent all the data as flattened arrays and, when possible, leveraging on the array index to encode the class we are referring to when storing data. Purposely, we managed to analyze the original code to figure out the contribution of every intermediate function in terms of pointers to the input tensors data. In other words, through the definition of accessory macros which takes as arguments the parameters on which the inputs shape and order depends, we are able to compute either the right confidence or the right location pointer to the input data, as well as an index to access the right boxes predictions. For example, the formula to compute the offset of the *i-th* box for the label c of the Location Tensor is: *(i*tot class*4)+(c*4)*[1]. For the sake of simplicity, then, we wanted to loop over these conditions and create logically sorted array of pointers, thereby enabling the sorting operations of the filtering phase to work with pointers.

For what concerns the output shape and allocation, it is not a problem if the model specifies the number of best detections to keep as we can consider that value to be an upper bound to allocate memory for the output. Otherwise, a default size must be defined at compile time.

Despite we had to rely on "worst-case" statical memory allocation which often takes more memory than is needed, our C implementation of the layers uses about 60% less memory than the original C++ version. Analyzing stack and heap usage, as shown in Fig. 2, it turned out that the use of memory is directly proportional to both the number of priors and the number of classes, while the final number of detections is negligible[2] .

Even if the author of the model provided the best ratios and, accordingly, the best number of default boxes to maximize the accuracy, it could be worth to consider how these parameters affect memory usage and train the model accordingly. For what concerns CPU usage, the layers execution times are really low compared to the time taken by the feature extractor network. Indeed, with the considered parameters, the Detection Output Layer took barely 3 ms to run on a CPU thus fulfilling the requirements for a 30-60 FPS realtime detector. In Tab. 1 a data[3] comparison between the well known VGG16 and the SSD networks is reported, so to give an extimation of the network complexity before and after the optimization process.

[1] valid if *share location* is **false**

[2] Considering the case with N CLASSES = 5, N PRIORS=8732 the Detection Output Layer, requires approximately 500 KB of RAM

[3] Verified data taken by Netscope CNN Analyzer

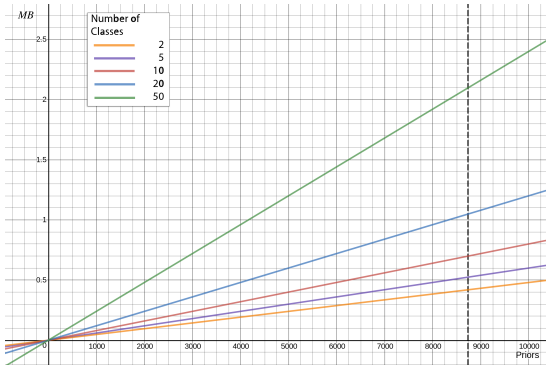

Figure 3 - Memory usage (MB) of the Detection Output Layer w.r.t. different network parameters.

			Optimized memory requirements	
	VGG16	SSD	VGG16	SSD
MACC	15.5 G	31.4 G	Same	Same
Activations	28.8 M	55 M	~16 MB	~24.2 MB
Weights	138 M	26 M	552 MB	104 MB

Table 1 - MACC, Parameters and Activations comparison between the complete VGG16 and SSD. M = Million - G=Billion

VI. CONCLUSIONS AND FUTURE WORKS

We have performed an optimized implementation of the main SSD layers that meet the requirements set forth by the considered HW/SW platform. Analysis on memory usage highlighted that our pointer-based approach mixed with few static allocations is affordable with respect to the available constrained resources. The benchmarks shows that the time spent on the execution of these layer is negligible compared to the time required to run the feature extractor network, pointing that a different base network could be considered for further optimizations. Although there are still margins of improvement, for the current state of the art, the SSD detector turns out to be one of the best real time detectors to use within an embedded context. Related work [4] shows that the model could be further improved for the detection of small object by using clever concatenation/merge of high feature maps to catch more semantic informations. Fine tuning, as well as the implementation of different feature extractors base networks, should be addressed and benchmarked in future work in order to evaluate the best tradeoff for embedded object detection applications.

VII. REFERENCES

[1] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.- Y. Fu, and A. C. Berg. SSD: Single Shot Multibox Detector. In *European Conference on Computer Vision*, 2016.

[2] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, K. Murphy. Speed/Accuracy trade-offs for modern convolutional object detectors. *arXiv*:1611.10012v3, 2017

[3] S.Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 2015.

[4] G. Cao, X. Xie, W. Yang, Q. Liao, G. Shi, J. Wu. Feature-Fused SSD: Fast Detection for Small Objects. *arXiv: 1709.05054v2*, 2017