Real-time Object Classification and Complexity Evaluation of Lightweight Convolutional Neural Networks on Mobile Computing Platforms

Alin-Gabriel Cococi, Daniel-Mihai Armanda
Doctoral School in Electronics
Telecommunications and Information Technology,
University "Politehnica" of Bucharest, Romania
alin.gabriel_cococi@sdettib.pub.ro,
daniel.mihai_armanda@sdettib.pub.ro

Ioana Dogaru, Radu Dogaru
Natural Computing Laboratory,
Dept. of Applied Electronics and Information Eng.
University "Politehnica" of Bucharest, Romania
ioana.dogaru@upb.ro
radu_d@ieee.org

***Abstract*—: An optimized convolutional neural network architecture is implemented on a computational mobile platform and its performances are evaluated in comparison with MobileNet for widely considered character and object recognition datasets. The possibility of completely eliminating the hidden dense layer of neurons doubled by a proper optimization of the convolutional layers structure led to lightweight models providing better recognition speeds than obtained with widely known architectures such as Mobilenet when implemented on mobile platforms.**

Keywords— convolutional neural networks, resource constrained, mobile platforms, deep learning, image recognition.

I. INTRODUCTION

Interest in designing and implementing machine learning architectures and models for every device grows every year at an exponential rate. Their benefits are clear and the need for efficient data processing is only going to grow. Artificial intelligence (AI) is being now embedded in almost every application and the most popular field nowadays is image processing or computer vision. Many applications use machine learning architectures for image classification, object recognition and so on.

Deep learning has been the leading interest for resolving such image recognition problems. The problem with such complex architectures is that they are computationally demanding and require large power-consumption for the processors to run on. Deep learning models can have a very high accuracy in classification problems but are rather difficult to implement in constrained resources platforms like mobile phones, IoT platforms or FPGA. On the other hand such constrained resourced platforms are widely used for acquiring and processing images in remote or low power environments. Recently, many researches go in the direction of simplifying the deep-learning architectures in order to fit such resource-constrained platforms [1][2][3][4][5][11-16].

The most popular approach for image processing is using a convolutional neural network (CNN). These networks work very well in image classification consisting from multiple convolutional layers (each including nonlinear processing and pooling) followed by a multi-layer perceptron (MLP) with at least one hidden layer. The general approach for implementing such a neural network architecture to a constrained resource device or platform is to train the desired model on a high performance platform and then deploy the trained model to run on the limited resourced platform. However, if the resulting models are large they cannot fit the limited memory and due to the huge volume of computations associated with a large number of parameters, their latency (defined as the time needed to provide a decision when an image is applied at the input) is also large an may not fit into applications purpose.

Herein we consider the training and optimization of a light weight CNN architecture (called next L-CNN from Lite weight-CNN) which is the trainable version of the binary weight model proposed in [17] and developed in Keras[1]. Using a set of tools[2] [8] the optimized models are transferred into a mobile platform using Tensorflow Lite. On these platforms we evaluate the performances in comparison with the widely known MobileNet [2][6] (models were also trained in Keras using code available publicily[3]). TensorFlow is a free and open-source software library developed by the Google Brain team, and it is one of the widely used library for machine learning. TensorFlow Lite [7] is a set of tools to help developers run TensorFlow models on mobile devices. For this comparative research we created a mobile application that can load any TensorflowLite model and accepts multiple types of inputs like camera images, finger drawing input and whole databases as .csv files from the cloud or local storage. The loaded model can run on CPU, GPU or NNAPI and can also run on multiple threads. The mobile application project that we have created is open-source and can be downloaded or forked at [8].

Section II contains information about the architectures of the considered machine learning models for testing on a constrained resourced mobile platform. Section III provides the experimental results obtained from the mobile application

[1] https://keras.io/
[2] https://github.com/cococialin/devteam-neuro-lab
[3] https://github.com/arthurdouillard/keras-mobilenet

978-1-7281-2906-8/19/$31.00 ©2019 IEEE

running on various configurations on CPU, GPU or NNAPI[4] of the mobile platform. Section IV presents the conclusions and remarks about the experimental results.

II. DEEP LEARNING ARCHITECTURES

A. The L-CNN Architecture

In [17] a lightweight model is proposed for constrained resources architectures. To promote aggressive reduction of computations, binary weights were proposed in the convolutional layers and in order to "train" those randomly generated weights an extreme learning machine with no hidden layer (ELM0) (using an improved implementation based on [9]) was used due to its very good speed in order to adjust the best set of binary kernels. In the end however, the ELM0 in the output the CNN is replaced with MLP0, i.e. a single output trainable layer in Keras (this strategy allowing a significant decrease in the number of parameters without sacrificing the accuracy performance). This paper uses the trainable version of the above system (L-CNN) i.e. where kernels have no binary but float32 weights and they are obtained during a (usually much longer) standard Keras training process. The code for both BCONV-ELM and L-CNN is provided in [17].

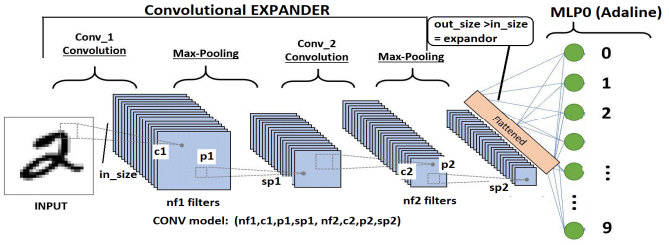

Fig. 1. Structure of the L-CNN architecture

The structure of the L-CNN is provided in Figure 1. The models are simply described as *(nf1,c1,p1,sp1,nf2,c2,p2,sp2)* where *nf1* and *nf2* are the number of filters per each layer, usually the main subject of model optimization, *c1* and *c2* are sizes of the convolutional kernels (typically they are chosen 3 to minimize computations, but sometimes size 5 is also used), *p1 and p2* are the pooling size (typically 2) and *sp1 sp2* are pooling strides size (downsampling the images) usually chosen to have a value of 2. A ReLU (rectified linear unit) nonlinearity is usually used in each of the convolutional layer although in the BCONV-ELM abs() nonlinearity was also found useful. Two types of convolutional layers were proposed in [6] and implemented also in the trainable version L-CNN namely the usual **conv2d()** convolution layer and the **depthwise_conv2d()** whih was particularly found the most convenient for the binary kernel approach. If not specified elsewhere, the trainable L-CNN uses the usual **conv2d()** convolution.

B. The MobileNet Architecture

The MobileNet model, detailed in [2] aims to reduce the complexity and thus generate slimmer architectures (also giving the user a possibility to control de size of the model for a given accuracy) and its performances rely on using depthwise separable convolution which is a form of factorized convolution into a depthwise convolution (with a certain number of 3x3 or 5x5 kernels, each corresponding to a filter, applied to input images) and a 1×1 convolution applied to all resulted channels called a pointwise convolution. For MobileNet the depthwise convolution applies a single filter to each input channel. Most computation is localized now in the pointwise convolution but in the end the number of computations is significantly reduced when compared with standard CNN architectures. Recently, improvements of Mobilenet were reported [5]. For comparisons, we used the code publicly provided and mentioned in the introductory section with the default structure provided by the authors. It is likely the optimization of that structure may lead to improvements but this was out of the scope of this paper.

[4] https://developer.android.com/ndk/guides/neuralnetworks

III. EXPERIMENTAL RESULTS

MNIST (handwritten figures) and CIFAR10 (various colour objects) datasets were considered for testing the two architectures. We used the mobile application [8] created by us on Android platform. The application can load any **tflite** file and run the model for any dataset provided as **.csv** file. The user has multiple options for running and testing the model. The app can run a model on CPU, GPU and NNAPI, it can also run on multiple threads. The app also contains options for manipulating the input images by rotating or flipping them. It is a useful tool for easy testing AI models on a constrained resourced platform. One can design a model in Keras and save the model to a **.h5** file and then, as seen in Fig. 2, export the model to a TensorflowLite model needed by the app. After this step no extra coding is required to run and test the model.

```
from tensorflow.contrib import lite
converter =
lite.TFLiteConverter.from_keras_model_
file( 'model.h5' ) # model's name
model = converter.convert()
file = open( 'model.tflite' , 'wb' )
file.write( model )
```

Fig. 2. Conversion code from .h5 format of the Keras models into .tflite model used as input by the mobile app.

The MNIST dataset [10] has N=10000 testing samples M=10 classes. Each sample from the MNIST dataset represents an image with n=28x28 pixels. CIFAR10 dataset has N=10000 testing samples and M=10 classes. Each sample from the CIFAR10 dataset represents an image with n=32x32 pixels.

The following tables present how well each model performs on the Android mobile app in different settings. The mobile computational platform used for running the Android 9.0 (Pie) application has a Octa-core (4x2.7 GHz Mongoose M3 & 4x1.8 GHz Cortex-A55) CPU with a 64 bit architecture, Mali-G72 MP18 GPU and 4 GByte of RAM. In terms of execution times, t1_UI represents the time of execution for *all samples* on the user interface thread, t2_UI represents the mean time of execution of *one sample* on the user interface thread, t1_BK represents the time of execution for *all samples* on the background thread and t2_BK the time of execution for *one sample* on the background thread. In terms of latency, as

reported by other papers devoted to resources constrained implementations, the last line is the most relevant. Tables I to VI summarizes the results obtained for the MNIST dataset where the following L-CNN model was used: *(nf1,c1,p1,sp1,nf2,c2,p2,sp2)* =(40,5,2,2,80,3,2,2) with more details in Figure 3. The MobileNet model provided by the publicly available code has around 3000000 trainable parameters (almost 42 times more than needed by the L-CNN model).

TABLE I. MODELS RUNNING ON CPU WITH 1 THREAD

MNIST DATASET	L-CNN	MobileNet
Accuracy [%]	99.52	99,73
t1_UI [seconds]	13.237	26.107
t2_UI [ms]	1.323	2.6107
t1_BK [seconds]	6.252	12.143
t2_BK [ms]	0,625	1.2143

TABLE II. MODELS RUNNING ON CPU WITH 3 THREADS

MNIST DATASET	L-CNN	MobileNet
t1_UI [seconds]	25.238	42.802
t2_UI [ms]	2.523	4.280
t1_BK [seconds]	11.739	19.908
t2_BK [ms]	1.173	1.990

TABLE III. MODELS RUNNING ON CPU WITH 6 THREADS

MNIST DATASET	L-CNN	MobileNet
t1_UI [seconds]	21.7601	64.530
t2_UI [ms]	2.1760	6.453
t1_BK [seconds]	10.121	30.014
t2_BK [ms]	1.0121	3.001

TABLE IV. MODELS RUNNING ON CPU WITH 9 THREADS

MNIST DATASET	L-CNN	MobileNet
t1_UI [seconds]	20.8163	62.152
t2_UI [ms]	2.08163	6.215
t1_BK [seconds]	9.682	28.908
t2_BK [ms]	0.9682	2.890

TABLE V. MODELS RUNNING ON GPU

MNIST DATASET	L-CNN	MobileNet
t1_UI [seconds]	14.3018	30.695
t2_UI [ms]	1.4301	3.069
t1_BK [seconds]	6.652	14.277
t2_BK [ms]	0.6652	1.427

TABLE VI. MODELS RUNNING ON NNAPI

MNIST DATASET	L-CNN	MobileNet
t1_UI [seconds]	159.693	252.663
t2_UI [ms]	15.9693	25.266
t1_BK [seconds]	74.276	117.518
t2_BK [ms]	7.4276	11.751

As seen from the above, CPU running in background gives the best latency performance for the L-CNN model tailored to MNIST dataset with 0.6 ms (almost 2 times less than MobileNet – however, a better accuracy is obtained with the MobileNet network). Running on GPU or NNAPI brings no significant advantages in this case. The next tables refer to the CIFAR dataset using the optimized L-CNN model: *(nf1,c1,p1,sp1,nf2,c2,p2,sp2)* =(60,3,2,2,60,3,2,2) with a total of 75450 parameters. Although MobileNET provides a plus of 9% in accuracy it requires 3,239,114 parameters, i.e. 43 times more than needed by the L-CNN. Accuracies are provided only in Tables I and VII since they not depend on the running environment.

TABLE VII. MODELS RUNNING ON CPU WITH 1 THREAD

CIFAR10 DATASET	L-CNN	MobileNet
Accuracy [%]	76.22	85.65
t1_UI [seconds]	17.709	21.484
t2_UI [ms]	1.770	2.148
t1_BK [seconds]	8.237	9.993
t2_BK [ms]	0,823	0,999

TABLE VIII. MODELS RUNNING ON CPU WITH 3 THREADS

CIFAR10 DATASET	L-CNN	MobileNet
t1_UI [seconds]	27.711	38.179
t2_UI [ms]	2.771	3.817
t1_BK [seconds]	12.889	17.758
t2_BK [ms]	1.288	1.775

TABLE IX. MODELS RUNNING ON CPU WITH 6 THREADS

CIFAR10 DATASET	L-CNN	MobileNet
t1_UI [seconds]	26.386	59.862
t2_UI [ms]	2.638	5.986
t1_BK [seconds]	12.273	27.843
t2_BK [ms]	1.227	2.784

TABLE X. MODELS RUNNING ON CPU WITH 9 THREADS

CIFAR10 DATASET	L-CNN	MobileNet
t1_UI [seconds]	25.438	57.529
t2_UI [ms]	2.543	5.752
t1_BK [seconds]	11.834	26.756
t2_BK [ms]	1.183	2.675

TABLE XI. MODELS RUNNING ON GPU

CIFAR10 DATASET	L-CNN	MobileNet
t1_UI [seconds]	18.924	26.073
t2_UI [ms]	1.892	2.607
t1_BK [seconds]	8.805	12.126
t2_BK [ms]	0.8	1.212

TABLE XII. MODELS RUNNING ON NNAPI

CIFAR10 DATASET	L-CNN	MobileNet
t1_UI [seconds]	168.858	235.362
t2_UI [ms]	16.885	23.536
t1_BK [seconds]	78.539	109.471
t2_BK [ms]	7.853	10.947

As for the MNIST, it is again observed that CPU running on 1 thread in the background gives a good latency (now only slightly smaller than for the MobileNet) but the increase in accuracy is now better for the MobileNet (considering that it

was trained with large objects images datasets it would be now surprise). The Android Neural Networks API (NNAPI) is an Android C API designed for running computationally intensive operations for machine learning on mobile devices. NNAPI is designed to provide a base layer of functionality for higher-level machine learning frameworks such as TensorFlow Lite. However, as seen from tables VI and XII, even though NNAPI is designed for running computationally intensive operations, we got the slowest execution times, thus this issues deserve further investigation.

MNIST:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 40)	1040
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 40)	0
conv2d_2 (Conv2D)	(None, 14, 14, 80)	28880
activation_1 (Activation)	(None, 14, 14, 80)	0
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 80)	0
activation_2 (Activation)	(None, 7, 7, 80)	0
flatten_1 (Flatten)	(None, 3920)	0
dense_1 (Dense)	(None, 10)	39210

Total params: 69,130

CIFAR10

Layer (type)	Output Shape	Param #
conv2d_59 (Conv2D)	(None, 32, 32, 60)	4560
max_pooling2d_59 (MaxPooling	(None, 16, 16, 60)	0
conv2d_60 (Conv2D)	(None, 16, 16, 60)	32460
activation_59 (Activation)	(None, 16, 16, 60)	0
max_pooling2d_60 (MaxPooling	(None, 8, 8, 60)	0
activation_60 (Activation)	(None, 8, 8, 60)	0
flatten_30 (Flatten)	(None, 3840)	0
dense_30 (Dense)	(None, 10)	38410

Total params: 75,430

Fig. 3. L-CNN models used for MNIST and CIFAR datasets

Figure 4 summarizes the dynamic performances of the Android implementations as resulted from the above tables and for the most critical parameter (the background running time per sample).

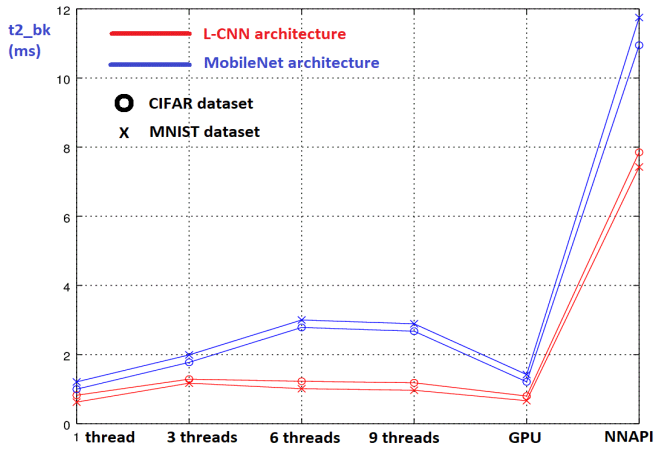

Fig. 4. Dynamic properties of Android implementations for the considered architectures and datasets.

IV. CONCLUSIONS

A flexible framework to integrate on mobile platforms various light-weight deep learning models trained in Keras was developed, with its code made publicly available [8]. Using tools and methods exposed in [17] we performed comparisons and evaluations for different datasets and for both L-CNN and MobileNet [2] architectures in order to evaluate the possibilities of the mobile platform. Although the accuracies were slightly smaller for the L-CNN, the latencies (execution times on mobile platforms) are faster for L-CNN than for the MobileNet architecture. The best execution time is when the models run on CPU with 1 thread. This happens because both models are not designed to run on multiple threads and except the execution thread, the rest of them are just slowing down the mobile device. Further research will consider the integration on mobile platforms of the BCONV-ELM model [17] (which is not stored in the standard Keras format) and possibly the development of some specialized functions to support aggressive quantization (as in binary kernel networks) or mobile GPU support as a convenient method to improve the speed while maintaining the accuracy in reasonable limits. For medium complexity datasets such as MNIST, L-CNN provides a good alternative to more sophisticated models (such as [2],[11]) which are optimized for computational speed but less for memory (note that L-CNN requires 42 times less memory than MobileNet models).

REFERENCES

[1] S. Yao et al., "Deep Learning for the Internet of Things," in Computer, vol. 51, no. 5, pp. 32-41, May 2018.

[2] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", Google Inc., 2017

[3] M. Lin, Q. Chen, and S. Yan, "Network in Network," in Proc. of ICLR, 2014.

[4] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in Proc. of CVPR, 2017.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications," in arXiv:1704.04861, 2017.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in arXiv:1801.04381v3, 2018.

[7] "Tensorflow Library", Available: https://www.tensorflow.org/ , 30.08.2019

[8] "Android aplication for testing tflite models", Available: https://bitbucket.org/devteamromania/elm-android

[9] R. Dogaru and I. Dogaru, "Optimized Extreme Learning Machine for Big Data Applications Using Python," 2018 International Conference on Communications (COMM), Bucharest, 2018, pp. 189-192.

[10] L. Deng. The MNIST database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), November 2012.

[11] Z. Qin, Z. Zhang, X. Chen, and Y. Peng, "FD-MobileNet: Improved MobileNet with a Fast Downsampling Strategy," in arXiv:1802.03750, 2018.

[12] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated Residual Transformations for Deep Neural Networks," in Proc. of CVPR, 2017.

[13] L. Sifre, "Rigid-motion Scattering for Image Classification, Ph.D. thesis, 2014.

[14] L. Sifre and S. Mallat, "Rotation, Scaling and Deformation Invariant Scattering for Texture Discrimination," in Proc. of CVPR, 2013.

[15] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in arXiv:1707.01083, 2017.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proc. of CVPR, 2016.

[17] R. Dogaru and I. Dogaru, "BCONV-ELM: Binary Weights Convolutional Neural Network Simulator based on Keras/Tensorflow, for Low Complexity Implementations", ISEEE 2019 (this Proceedings), in press.