Adapting Deep Learning models to IoT environments

Sofien Resifi, Hassan Hassan, Khalil Drira

To cite this version:

Sofien Resifi, Hassan Hassan, Khalil Drira. Adapting Deep Learning models to IoT environments. 5th Conference on Cloud and Internet of Things (CIoT 2022), Mar 2022, Marrakech, Morocco. 10.1109/CIoT53061.2022.9766636 . hal-03622723

HAL Id: hal-03622723
https://laas.hal.science/hal-03622723

Submitted on 29 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Adapting Deep Learning models to IoT environments

Sofien Resifi
LAAS-CNRS
University of Toulouse
Toulouse, France
sresifi@laas.fr

Hassan Hassan
LAAS-CNRS
University of Toulouse
Toulouse, France
hhassan@laas.fr

Khalil Drira
LAAS-CNRS
University of Toulouse
Toulouse, France
khalil@laas.fr

***Abstract*—Deep Learning (DL) models are very efficient for many applications including, computer vision, natural language processing.... Yet DL models require important computation resources making it particularly difficult to deploy these applications in constrained environments such as the Internet of Things (IoT). Offloading DL models to the cloud is one solution to this problem but has a number of drawbacks related to the trade-off between efficiency and latency, and other privacy issues. In this paper we try to solve this problem using two approaches, first by sharing the DL model between the cloud and the device and second by optimising the execution of the model using early exiting where inputs do not need to execute the model entirely. Both approaches are optimized automatically in order to choose the best sharing point and the best exiting point according to input. The solutions proposed could be easily generalized and are independent of applications and offer a good alternative in order to execute DL models locally.**

***Index Terms*—Deep Learning, IoT, cloud computing, partitioning, optimization**

I. Introduction

Deep Learning (DL) applications know a wide success during recent years. They are used in image recognition, object tracking in videos, natural language processing for virtual assistants, reinforcement agents in robotics and many other fields. Although those models are very efficient, they require important resources to be executed. This is not an issue when it comes to offload applications to the cloud and uses its unlimited resources. Meanwhile making DL applications completely dependant on the cloud has many drawbacks. This includes for real time application a long response time, and for privacy, an important issue as data should transit on distant datacentres. One of the fields that are challenged by the deployment of DL applications is IoT environments. As IoT devices have very limited resources, small CPU capacity, limited memory and constrained energy power supply, DL applications require new strategies in order to run in an IoT environments. In this paper, we handle the adaptation of DL models in constrained IoT devices using two approaches: first by optimized model partitioning for cloud-device collaboration under CPU and energy limitations and second by executing DL models partially. The bandwidth between the device and the cloud is also used as an optimization parameter. Our approach applies to any model and adapts automatically to device capacities and network conditions in the IoT environment. Performances achieved could attain up to 4X speedup in latency of model execution and 40% energy reduction in some cases. The rest of this paper is organized as follows; first, we have on overview of related work, then we introduce model analysis to point out possible optimization strategies. Second, we explore optimized cloud collaboration and early exit methods in order to enhance the performance of DL models on IoT devices. After that, we show the results of the proposed methods and finally we conclude with the perspectives of this work

II. Related work

In the literature, we find many works handling the problem of adapting DL models to constrained devices under different angles. First we find approaches trying to accelerate the execution of DL models using "slimming" techniques to combine tensor and non-tensor layers in neural network models [8]. Although this approach claims a real gain in time execution and memory usage, the "slimming" technique is highly dependent on the model and cannot apply to all neural network models such as recurrent neural networks (RNN). The acceleration of the DL model execution can be achieved by model compression techniques as in [2], [4], [14]. In this case the goal is to reduce the computation, energy and storage cost by cutting unimportant neuron connections in the model. The performance of such approaches is conditioned by the neural network model and the dataset used to train it. Exploiting redundant operations as proposed in [3] can also be considered in Convolution Neural Network (CNN) due to some high redundancy in convolution computation but cannot be applied to other models. Other approaches tackle the partitioning of the model itself in order to get part of the model executed on the IoT device and other parts offloaded to the cloud as proposed in [6]. While in this case the main problem is how to decide about the good partitioning of the model according to the dataset used to train the neural network and based on network conditions. Choosing to execute the model partially is another approach as suggested in [12]. The same problem as previously is to decide about the right point to stop the model execution while getting an acceptable accuracy on predictions. Besides these approaches, a brand new technologies named under TinyML [13] aim to integrate Machine Learning capacities within small objects powered by

Microcontroller Units (MCUs). TinyML does not search to adapt DL models to small IoT devices but rather to implement some capacities in MCUs in order to make the execution of ML possible on constrained objects and open the way to some specific applications. Our work is different by searching to analyse common DL models and propose new strategies to deploy these models on constrained devices either by cloud-device collaboration or by early exiting techniques. In both approaches we provide an optimization method to make the decision of partitioning or early exiting automatically.

III. Model analysis

A. Output data size at each layer

When an image goes through a Deep Learning model, it goes through each layer separately (at each layer we have a matrix operation). The shape of the image will change passing from one layer to another. We benchmark with the state of the art Deep Learning models. The output data size will be a very important parameter in the optimization problem. We calculate the output data size at each layer, as the output data size depends on the type of the layer:

- **Convolutional Layers**: Those layers have many different parameters, the output size will be based on the following parameters:
  - Output Height Formula:

$$\frac{Input\ height + 2 * padding - kernel\ size}{stride} + 1 \tag{1}$$

  - Output Width Formula:

$$\frac{Input\ width + 2 * padding - kernel\ size}{stride} + 1 \tag{2}$$

- **Max Pooling Layer**: The same calculation for the convolutional layer can be applied to the Max Pooling layer.

where:

- **Kernel Size**: The kernel size is the size of the applied filter, it can be 3x3,5x5...
- **Stride**: Stride controls how the filter convolves around the input. the filter convolves around the input by shifting a specific number of unit at a time. The amount by which the filter shifts is the stride.
- **Padding**: Padding is a term relevant to CNNs as it refers to the amount of pixels added to an image when it is being processed by the kernel of a CNN.

B. Input/Output variables

This part is very important for the latency estimation model, in fact we provide the variables for each layer's type to consider:

- **Convolution Layers**:We include the input feature map dimension, number, size and stride of the filters. The latency estimation model for convolution layer is based on two variables:
  - the number of pixels (Input Dimension) in the input feature maps,
  - $(\frac{filter\ size}{stride})^2 * number\ of\ filters$, which represents the amount of computation applied to each pixel in the input feature maps
- **Pooling Layers**: we use the size of the input and output feature maps as the latency estimation model variables
- **Fully Connected Layers**:the input data is multiplied by the learned weight matrix to generate the output vector. We use the number of input neurons and number of output neurons as the latency estimation model variables. The activation functions layers are handled similarly.

Table I resumes the variables for each type of layer:

Layer Types / Variables	In	Out
Convolution Layer	Input number of pixels	$(\frac{filter\ size}{stride})^2 * \#\ filters$
Pooling Layer	In Dimension	Out Dimension
Fully Connected Layer	In Neurons	Out Neurons
Activation Layer	In Dimension	Out Dimension

TABLE I
USED VARIABLES FOR THE LATENCY ESTIMATION MODEL

For the activation layers the input dimension is the same as the output dimension, so we can consider them as one variable

C. Latency estimation model

We predict the latency (execution time) of a layer based on the input and output variables mentioned in III-B. We establish a linear regression model for estimating the latency per layer, the formulation of the problem depends on the type of the layer:

- **Convolution**, **Pooling** and **Fully connected** layers the formulation will be as follows:

$$\hat{y} = a1 * Input\ variable + a2 * Output\ variable + b \tag{3}$$

  where
  $\hat{y}$: The estimated latency.
  $a1$,$a2$: Coefficient of the linear regression
  b: Bias.
- **Activation Layers**: Since in activation layers the input dimension and the output dimension are the same, we consider them as one variable for the linear regression model:

$$\hat{y} = a1 * Input/Output\ variable + b \tag{4}$$

To create a linear regression model we need to create a dataset, a matrix containing samples of input/output variables, and we measure the latency for each couple of input/output variables. For this purpose we use the Python Library $Time$ [1]. First we vary the possible Input size (Input variable) for each layer then we calculate the output variable (whether it is the output data size or the computation per input pixel which depends on the type of the layer) based on the parameter of that layer, so for the same input variable we will have different output variable depending on the layer's parameters. As we can see each layer will be treated differently. The output of this process is the data and the measured latency for that data, the data will look different depending on the layer type:

- **Convolution**, **Pooling**, and **Fully Connected (FC)** Layers: The data matrix in this case will have three columns, samples of the input variable, samples of the output variables and the bias, the matrix and the measured latency look like:

Matrix Data= $\begin{pmatrix} IN_1 & OUT_1 & 1 \\ IN_2 & OUT_2 & 1 \\ . & . & . \\ . & . & . \\ . & . & . \\ IN_n & OUT_n & 1 \end{pmatrix}$

Measured Latency= $\begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix}$

- **Activation Layers**: The matrix data in this case will have two columns, samples of the input/Output variable(since they are the same), samples of the output variables and the bias, so the matrix and the measured latency look like:

Matrix Data= $\begin{pmatrix} IN_1 & 1 \\ IN_2 & 1 \\ . & . \\ . & . \\ . & . \\ IN_n & 1 \end{pmatrix}$, Measured Latency= $\begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix}$

The solution of this linear regression problem is presented in the next section.

D. *Solving the linear regression model*

Let us call the data matrix X and the measured latency Y. We need to figure out a linear regression relationship between X and Y:

$$X*? = Y \tag{5}$$

where X ia an n*3 matrix (depending on the layer's type but the approach is the same for all layers) and Y is a n*1 vector. In linear estimation we add a vector called A with suitable shape to create a relation between X and Y, our equation will look like this:

$$X * A = Y \tag{6}$$

Where A is 3*1 vector, A= $\left(a_1, a_2, a_3\right)$ or A= $\left(a_1, a_2\right)$ depending on the layer's type.
Our goal is to calculate the vector A. The intuitive thinking is to inverse the matrix X but we can not do that directly since X is a n*3 matrix which is not a square matrix. First we are going to multiply by X^T for both sides therefor we get this equation:

$$X^T XA = X^T Y \tag{7}$$

Now we can see we have $X^T X$ as a 3*3 matrix (3*n x n*3=3*3) which can be inverted. So we can calculate the vector A by inverting the $X^T X$ matrix:

$$X^T XA = X^T Y \rightleftharpoons A = (X^T X)^{-1} X^T Y \tag{8}$$

Finally we calculated the vector A. In fact once we have our vector A and we choose a layer with specified input and output variables, we can estimate that layer's latency by multiplying the vector on input and output variables with the vector A. Let's call L a vector containing the input and output variables L= $\begin{pmatrix} IN_l \\ OUT_l \\ 1 \end{pmatrix}$,the estimated latency can be calculated as follows:

$$\hat{Y} = LA = \left(IN_l, OUT_l, 1\right) \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = a_1 IN_l + a_2 OUT_l + a_3 \tag{9}$$

Now we can see the the output of the equation (9) looks exactly like the formulation in (4).

E. *Evaluation of latency estimation model*

In order to evaluate the efficiency of the latency estimation model, we compare the model output with the measured latency and calculate the percentage of error. The table below sums up the evaluation results for different models:

CNN Models / Latency	Measured Latency(s)	Estimated Latency(s)	Percentage of Error(%)
AlexNet v1 [7]	0.041	0.039	4
AlexNet v2 [7]	0.054	0.056	3
VGG16 [10]	0.114	0.118	3.3
VGG19 [11]	0.1439	0.1491	4

TABLE II
EVALUATION OF LATENCY ESTIMATION MODEL

Figure 1 shows the results of the latency estimation on AlexNet model.

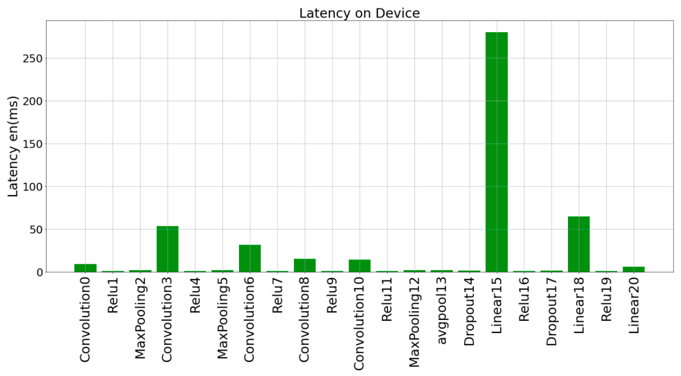

Fig. 1. Latency estimation model applied on AlexNet

IV. Partitioning algorithm

Based on the latency estimation of each layer on the device and on the server and taking into consideration the latency induced by the available bandwidth to transmit the data we search the best partitioning point for the model. The partitioning algorithm to make the collaboration between the device and the cloud is shown in Algorithm 1:

```
Algorithm 1: Partitioning algorithm
  Result: Partition Index
  Inputs:
    • CNN model
    • $D_i | i = 1...N$ : Data output size for each layer in the model
    • M($L_i$): Regression model predicting the latency for a specific layer.
    • B: current wireless network Uplink bandwidth
  For each i in 1...N:
  $TD_i \leftarrow M_{Device}(L_i)$
  $TS_i \leftarrow M_{Server}(L_i)$
  $TM_i \leftarrow \frac{D_i}{B}$
  PI= $\arg\min_{J=1...N}(\sum_{i=1}^{J} TD_i + \sum_{k=J+1}^{N} TS_K + TM_i)$
  Return PI
```

V. Early exit

A. Architecture

The second approach we present is early exit. In this approach the architecture is composed of a baseline model with some side branches as depicted in figure 2.

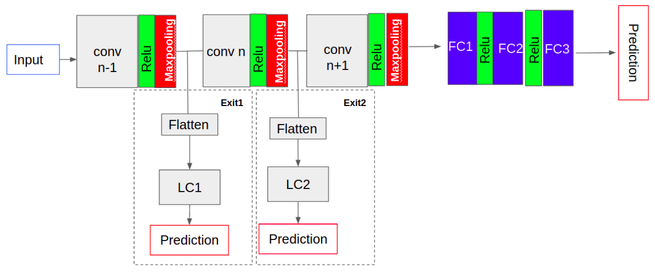

Fig. 2. Early exiting architecture

The input image goes through the architecture until arriving the (n-1) convolution layer, at that stage the features generated but the convolution layer number (n-1) will be fed to a linear classifier to generate a confidence level for a prediction. Based on that confidence level a decision will be made to exit the architecture at that stage or to proceed to the next stage after the next convolution layer, the exit decision is based on two criteria:

- If the linear classifier does not generate sufficient confidence level associated with any of the class labels or produce a sufficient confidence for more than one label, the input is deemed to be difficult to classify by the current stage and it is passed along to the next stage.
- If the linear classifier produces sufficient confidence associated with only one label, then the classification process is terminated at that stage and the corresponding label is produced as output of the framework.

B. Training phase

We use transfer learning in order to reduce training time on our models. Transfer Learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. Transfer learning is illustrated in figure 3.

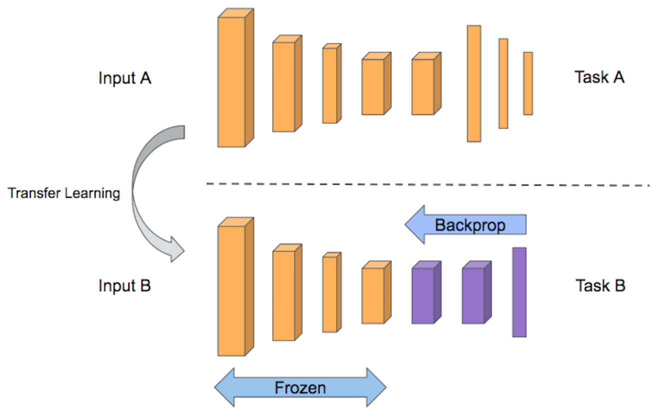

Fig. 3. Transfer learning

The training phase of this approach can be illustrated in some steps as follows

- **Training the Baseline**: In our work we used the baseline VGG16 [10] implemented with Pytorch [9] and pre-trained on the ImageNet dataset [5], we chose an example of dataset which is cat-dog dataset and we applied **Transfer Learning** to the pre-trained version of VGG16 to get a good baseline training with 98% accuracy.
- **Creating Exits**: We create exits and we add the linear classifiers to the baseline model. With the notion of **Transfer Learning**, we fix the weights of the convolutional layers of the baseline model and we only train the linear classifiers for each exit separately (The metric used to train the exits and the baseline is Cross Entropy).

C. Deployment phase & Decision making

Once the baseline and all exits are trained, we push the architecture to the deployment phase. The steps of the deployment phase are shown in algorithm 2.

In summary, the presented approach modulates implicitly the number of layers used for classification based on the input and produces an optimal DL model. The user defined threshold, P, for the confidence level can be adjusted during runtime to achieve the best trade-off between accuracy and efficiency improvements comparing to the baseline model. Thus, this approach is systematic and hence can be applied to all image recognition applications.

D. Improving early exiting approach

The early exit approach presented in the previous section requires a validation at each step before passing to the next. We improve it by adding an other component to the architecture which is called Automated Decision Making. Previously we

Algorithm 2: Deployment phase

```
Result: Early Exiting Model
Inputs:
    • CNN model as a baseline.
    • Input image
Step:
    • First Step: Launch the model for an input image until
      arriving to the first exit.
    • Second Step: Execute the first exit and get the vector
      of classes' probabilities.
    • Third Step: If the confidence value of the output is
      beyond a certain threshold P (user defined), then
      TERMINATE testing at that exit.
    • Fourth Step: Repeat the same steps until finishing all
      exits.
```

had the validation process, in fact each input goes through the exit before it is validated, if it is not validated it moves to the next exit. this process can be improved by the automated decision making.

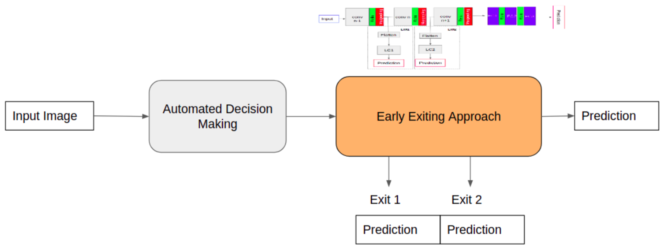

Fig. 4. Automated decision making

Instead of validating the input at each exit, with the automated decision making we predict the index of the exit so that the input goes directly to the exit where it is going to leave the architecture without passing by all the previous exits. The Decision Making component is a Deep Learning (DL) model which we train on a modified version of the data set that the baseline was trained on. In order to prepare the decision making component we have two steps:

- **Dataset Preparation**: To prepare the dataset, for each image we apply the early exiting approach to get a vector of exits' indexes which will be the new target for the decision making model. In this example we fixed the probability threshold to 0.9.
- **Training the Decision Making**: Once the data is ready we train the decision making model using the technique of Transfer Learning to obtain high accuracy.

The automated decision making comes with an additional cost, as the new component will induce additional time execution and energy consumption. In order to reduce the impact of this extra component we propose two solutions:

- **First Solution**: This first solution is mainly about reducing the Deep learning model used for the decision making. In our experiments we have seen that this solution is not efficient. In fact when we reduce the deep learning model (choosing a smaller DL model), we start to make some losses on the accuracy. This solution impacts negatively the performance of the approach besides the percentage of energy and latency gain has slightly improved comparing the early exiting approach without decision making.
- **Second Solution**: This second solution is to launch the decision making on the cloud to unburden the device with its costs. Since we are speaking about cloud computing, we must take into account the bandwidth of the Uplink connection. Figure 5 presents the new architecture of the early exiting approach:

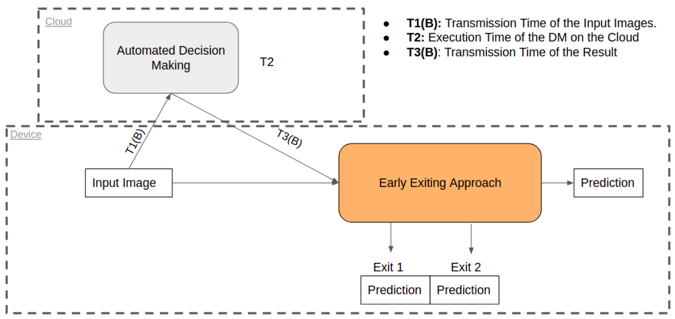
Cloud
Automated Decision Making
T2
• T1(B): Transmission Time of the Input Images.
• T2: Execution Time of the DM on the Cloud
• T3(B): Transmission Time of the Result
Device
T1(B)
T3(B)
Input Image
Early Exiting Approach
Prediction
Exit 1
Exit 2
Prediction
Prediction

Fig. 5. Offloading decision making to the cloud

When we execute the decision making component on the cloud we actually have some costs to add, the first one is $T_1(B)$ which is the transmission time of the input images, the second one is T_2 which the time execution of the decision making model on the cloud and finally $T_3(B)$ which is the reception time of the results (the exit indexes for the input images).

VI. Results

A. Partitioning algorithm results

In order to visualize the output of our partition algorithm, we fix the value of the bandwidth and the input image size. We choose the value $B = 0.05MB/s$ for the bandwidth and the size of $128 * 128 * 3$ for the input.

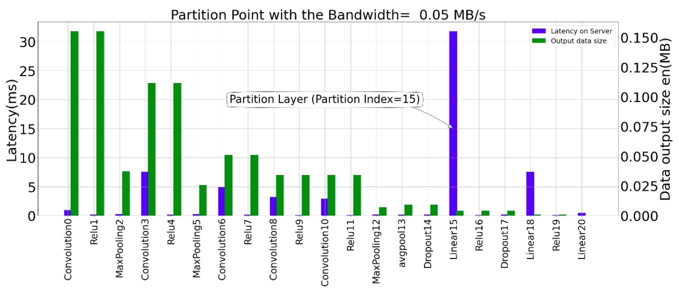

Fig. 6. Output of partitioning algorithm

Figure 6 presents how the partitioning algorithm works for different models for the specified bandwidth (B=0.05 MB/s).

Exit / Characteristics	Accuracy	F1 Score	Latency(s)	Energy Consumption
Exit1	0.9	0.89	19.62	53.16
Exit2	0.93	0.927	21.81	77.07
Baseline	0.98	0.98	34.68	138.94

TABLE III
RESULTS OF EACH EXIT SEPARATELY

The partitioning index is highly dependent on the bandwidth. In figure 7, we compare between three approaches, the device-only approach, the cloud-only approach and the partitioning approach. In order to have a good evaluation of this method we vary the bandwidth and observe how the latency of the decomposed model evolve in comparison to the latency of other approaches. In the case of the model AlexNet V2, we can notice that the latency of the model in the device (red line with star marker) is constant because it is not impacted by the bandwidth, while the latency on the server (blue line with lozenge marker) is decreasing when the bandwidth is increasing because in the cloud-only approach the input image will be sent to the cloud so the latency of transmission decreases with the bandwidth. Now if we look at the partitioning approach latency we remark that it is decreasing with the variation of the bandwidth, when the bandwidth is smaller then 2.3 MB/s the partitioning approach has the best latency which proves that this method is efficient. When the bandwidth is large enough we notice that the partitioning approach and the cloud-only approach are superposed, in other words the partition algorithm knows exactly when to send the model to the cloud, we call that bandwidth B_c which is the transition bandwidth to the cloud. Finally we have calculated the average speedup of this method by comparing the latency of the model on the device and the latency of the model with the partitioning for each value of the bandwidth. The speedup will be the average across the possible values of the bandwidth. Our calculation shows that we have 1.5X up to 3X speedup, actually the speedup depends on the server capacity, and with a more powerful server we would get a higher speedup.

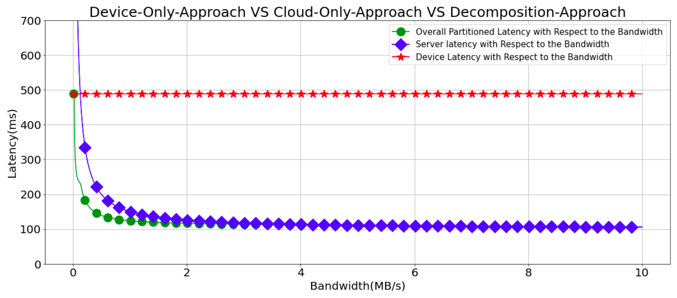

Fig. 7. Comparison of device-only, cloud-only, and partitioning approaches

B. Early exiting results

We evaluate the results of the early exit approach by working on a batch of images (it could be a batch of 100,200... images), in the results presented in table III the batch size is 150 images.

1) Evaluate each exit separately: Table III illustrates the performance, the latency, and the energy consumption of each exit on a batch of 150 images for probability threshold P=0.75. What we notice here that the baseline model is the most accurate, yet its latency and energy consumption is remarkably higher then exit1 and exit2 therefor obviously we will encounter a trade-off problem.

2) Image distribution per exit: Actually the image distribution depends on the threshold of probabilities that we choose, the bar plot show in figure 8 presents the different distribution of the images per exit for different threshold values. We remark that when the threshold is 0.5, the images tend to leave the architecture from the first exit since the confidence level demanded is low, while the number of images in exit 2 are around 20 images and fewer images get to complete the whole architecture. When the probability threshold is 0.9 which is a high confidence level we notice that a big portion of the images leave the architecture from the first exit which proves that for some samples of batch images, it is enough to leave the architecture at the first exit with a high confidence level. We notice also that the portion of the images that completed the baseline architecture is now important since the confidence level is high.

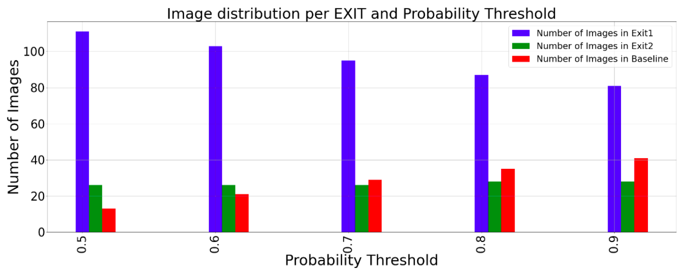

Fig. 8. Image distribution

3) Variation of energy gain, latency gain and accuracy loss: We examined before the distribution of the images per exit as a function of the probability threshold, now we look at the energy gain percentage and latency gain percentage and the accuracy loss percentage.

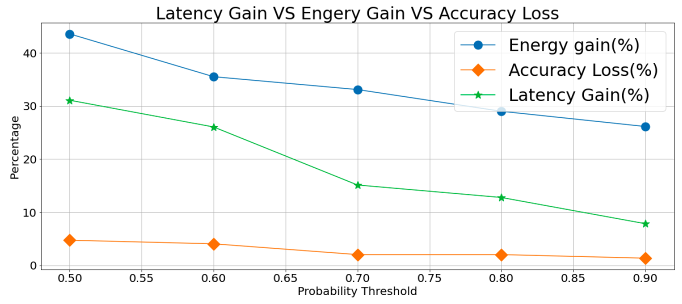

Fig. 9. Latency and energy gain vs accuracy loss

In figure 9 we can see that the percentage of gain latency and energy is decreasing with the variation of the probability threshold and that can be explained by the distribution of the images per exit. In fact when the probability threshold is 0.5 the most of image samples leave the architecture from exit 1, which is presented in section VI-B2. With that 0.5 confidence level we can reduce the time execution by 31% and the energy consumption by 44% with only 4% Loss in the accuracy. When we increase the probability threshold we can obviously see that we have a trade-off, for example when the threshold is 0.9 we have 0% loss in the accuracy yet the latency gain is only 9% and the energy gain is around 24%. To sum up, the choice of the probability threshold is very dependent on the task, for example if we are using the Deep Learning model to classify objects, it would be better to choose a low threshold while in medical diagnosis context the confidence level must be higher.

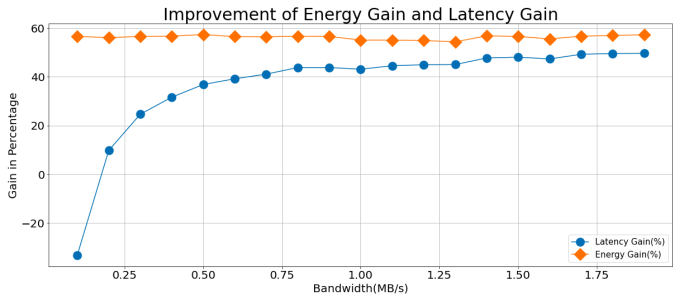

Fig. 10. Energy gain vs latency gain

4) Results of early exiting with decision maker: Since the accuracy loss is not impacted by the bandwidth, we chose not to present it on the figure 10, actually the accuracy loss was around 5% compared to the baseline. In fact the energy gain is not impacted by the variation of the bandwidth because the same model will be executed only the time is not the same. Here we can see that the energy gain is 57% which a very good result, previously with old version of the early exiting approach the maximum energy gain percentage was 40% when the threshold was 0.5 (low confidence level). Now with a high confidence level we got a higher energy gain percentage. When looking at the latency gain we notice that we have a negative gain when the bandwidth is too small(the negative gain is caused by the transmission latency), but when we have a bandwidth higher then 0.5 MB/s, which is a very large bandwidth, we start to get a latency gain percentage equal to 40% while previously the maximum latency gain was 37% with a low confidence level.

VII. Conclusion

In this paper we presented two approaches to enhance DL models performance in IoT environments. First we presented a collaboration method between the device and the cloud based on partitioning approach with an algorithm to determine the best point to partition a DL model. Second we presented an early exiting approach that aims to execute the DL model partially and exit the model as soon as the result respect the required accuracy. Both methods offer the possibility to IoT devices to execute complex DL models locally and make it possible to deploy high performance applications in IoT environments. In future work we will extend these approaches to video streaming applications in IoT environments.

References

[1] Python Software Foundation. Time calculation per instruction.

[2] Shupeng Gui, Haotao Wang, Haichuan Yang, Chen Yu, Zhangyang Wang, and Ji Liu. Model compression with adversarial robustness: A unified optimization framework. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 1283–1294, 2019.

[3] Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. Ghostnet: More features from cheap operations. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1577–1586, 2020.

[4] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[5] ImageNet. Imagenet dataset containing 1000 classes.

[6] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor N. Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 615–629. ACM, 2017.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[8] Dawei Li, Xiaolong Wang, and Deguang Kong. Deeprebirth: Accelerating deep neural network execution on mobile devices. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2322–2330. AAAI Press, 2018.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[12] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. *CoRR*, abs/1709.01686, 2017.

[13] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly Media, Incorporated, 2020.

[14] Yixing Xu, Yunhe Wang, Hanting Chen, Kai Han, Chunjing Xu, Dacheng Tao, and Chang Xu. Positive-unlabeled compression on the cloud. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 2561–2570, 2019.