Evaluating Edge-Cloud Computing Trade-Offs for Mobile Object Detection and Classification with Deep Learning

W. F. Magalhães[1], M. C. de Farias[1], H. M. Gomes[1], L. B. Marinho[1], G. S. Aguiar[2], P. Silveira[2]

[1] Universidade Federal de Campina Grande, Brazil
whendell@copin.ufcg.edu.br, mainara.farias@ccc.ufcg.edu.br,
hmg@dsc.ufcg.edu.br, lbmarinho@dsc.ufcg.edu.br
[2] Hewlett Packard Enterprise, Brazil
glaucimar@hpe.com, plinio.silveira@hpe.com

Abstract. Internet-of-Things (IoT) applications based on Artificial Intelligence, such as mobile object detection and recognition from images and videos, may greatly benefit from inferences made by state-of-the-art Deep Neural Network (DNN) models. However, adopting such models in IoT applications poses an important challenge since DNNs usually require lots of computational resources (i.e. memory, disk, CPU/GPU, and power), which may prevent them to run on resource-limited edge devices. On the other hand, moving the heavy computation to the Cloud may significantly increase running costs and latency of IoT applications. Among the possible strategies to tackle this challenge are: (i) DNN model partitioning between edge and cloud; and (ii) running simpler models in the edge and more complex ones in the cloud, with information exchange between models, when needed. Variations of strategy (i) also include: running the entire DNN on the edge device (sometimes not feasible) and running the entire DNN on the cloud. All these strategies involve trade-offs in terms of latency, communication, and financial costs. In this article we investigate such trade-offs in real-world scenarios. We conduct several experiments using deep learning models for image-based object detection and classification. Our setup includes a Raspberry PI 3 B+ and a cloud server equipped with a GPU. Different network bandwidths are also evaluated. Our results provide useful insights about the aforementioned trade-offs. The partitioning experiment showed that, overall, running the inferences entirely on the edge or entirely on the cloud server are the best options. The collaborative approach yielded a significant increase in accuracy without penalizing running costs too much.

Categories and Subject Descriptors: I.2.6 [**Artificial Intelligence**]: Learning; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

Keywords: deep neural networks, edge-cloud collaboration, image classification, object detection

1. INTRODUCTION

Recent advances in the field of Deep Neural Network (DNN) have led to the rapid growth of AI-based solutions in a variety of segments, such as robotics, automobiles, health, and safety. In many DNN applications, the model is fully hosted in the cloud. In such a scenario, to perform object recognition on a mobile device, for example, acquired images are sent to a cloud service, which then perform the inferences and send them back to the edge device. From time to time, the model needs to be re-trained (also in the cloud) to accommodate new training data. However, there may be a considerable round trip time associated with successive API calls to a remote server.

Applications that require real-time inference may not be feasible in medium/high latency environments. For example, in the context of autonomous cars, a latency that is too high could significantly increase the risk of accidents. Also, unexpected events, such as crossing animals or pedestrians in forbidden places, may occur in a fraction of seconds, demanding a real-time response. Moreover,

when there are many devices connected to the same network, the effective bandwidth is reduced due to the inherent competition to use the communication channel. These problems can be significantly reduced if the more critical computing is performed at the edge [Liu et al. 2019].

Besides autonomous cars, there is a myriad of DNN application scenarios that could benefit from the collaboration between edge and cloud processing. Video surveillance is among those scenarios. In a typical smart city setup, the amount of network bandwidth to accommodate transferring video data from thousands of video cameras to a cloud server can be prohibitive and cost-ineffective. Let's suppose the problem of detecting violence (such as fights [Carneiro et al. 2019]) across the streets of a large city. Police response time needs to be as fast as possible. Therefore, it is better to run the DNN models at the edge and send only the (compact) resulting inferences to the cloud, instead of video streams. For further automated analysis, that could integrate multiple view-inferences for confirming/discarding the violent events. Also in a smart city scenario, visual inspections (e.g., power consumption readings [da Silva Marques et al. 2019]) performed at the edge could drastically reduce the costs of cloud processing, since the core processing (image analysis) is distributed among commodity smartphones. In the medical domain, for instance, existing approaches for voice pathology recognition [Marinus et al. 2018] and skin lesion classification [dos Santos and Ponti 2018] could be implemented for edge execution (e.g., in a smartphone) and compose a tool for pre-screening patients before a doctor sees them. Whenever inferences are below a given confidence level, a more robust model running in the cloud could be queried to confirm/discard those inferences.

Edge computing refers to computation performed locally on edge devices (e.g., desktops, Wi-Fi access points, mobile phones, and cameras). Instead of sending data to the cloud for processing and inference, data may be processed locally, e.g. on-device (when the model complexity is low) or sent to an edge server that handles requests from multiple nearby devices (when models are too complex to run locally).

A striking feature of DNN models is a large number of parameters that need to be stored or maintained in memory and the amount of computation required when making inferences. Thus, running these models on mobile edge devices with limited computational resources is not trivial. Among the possible strategies to tackle this challenge are: (i) DNN model partitioning between edge and cloud; and (ii) running simpler models in the edge and more complex ones in the cloud, with information exchange between models. Running the entire DNN on the edge device (sometimes not feasible) and running the whole DNN on the cloud are two possible variants of strategy (i).

Both strategies are part of an emerging concept in smart IoT applications, named collective intelligence. Within a broad and historical perspective, collaborative intelligence may be characterized by multi-agent (whether machine or human), distributed systems where each agent has specific functions and autonomy to contribute to solving a problem [Gill 2012].

In a mobile-cloud scenario, Eshratifar et al. [2019] explicitly defined collaborative intelligence as splitting the workload between the mobile devices and the cloud to reduce overall latency and energy consumption of the application, as in the strategy (i) investigated in this work, which is inspired by the work of Kang et al. [2017] on DNN model partitioning.

In a slightly different view of collaborative intelligence, Weerakoon et al. [2019] assumed that IoT might involve the deployment of resource-constrained node sensors with varying degrees of redundancy or overlap. In that sense, collaborative intelligence occurs when individual nodes adjust their inference mechanisms to incorporate such correlated observations from other nodes to improve accuracy and performance metrics, such as latency and energy overheads. A similar understanding is shown by [Pasandi and Nadeem 2020] as they propose exploring spatio-temporal correlations among sensors (cameras) to remove redundant frames and allow collaborative knowledge sharing among sensors. This view is outside the scope of this article.

In follow-up work, Misra et al. [2019] proposed that edge devices should first autonomously use

statistical analysis to identify potential collaborative IoT nodes. The IoT nodes should then perform real-time sharing of information to improve their inference accuracy. Strategy (ii) investigated in this article is related to the research mentioned above. However, we do not study functions related to the automatic probe of collaborative nodes. We consider a setup where there is a significant gap of expertise and complexity between mobile and cloud DNN models. Their communication occurs on-demand, based on the confidence level of the model running on the edge.

For investigations regarding the partitioning of DNNs, we conducted several experiments on different versions of YOLO (You Only Look Once) [Redmon et al. 2016; Redmon and Farhadi 2016], a state-of-the-art DNN for the object detection task with proven high speed and accuracy. To evaluate the collaborative approach between DNNs running on the mobile edge and in the cloud server, the MobileNet [Howard et al. 2017] and Xception [Chollet 2016] models were chosen for mobile and cloud execution, respectively. We performed both experiments using a Raspberry Pi 3 B+ device and a cloud server equipped with GPU, with different network bandwidths. The experimental results provided useful insights into the trade-offs, as mentioned earlier. We complement existing work on this area, such as Kang et al. [2017] and Misra et al. [2019], with the following contributions:

—We present one of the first investigations on partitioning YOLO, considering both its *full* and *tiny* versions. Given that YOLO full has skip connections, special care needs to be taken in the partitioning process.
—We exploit several network bandwidths in order to discover the best configuration for each one.
—We consider a Raspberry PI 3 Model B+ as edge device, which is very limited in terms of computational resources when compared to current smartphones, but is more cost effective.
—We conduct our partitioning experiments on video streams captured from real-world surveillance cameras, thus reflecting a realistic use case.
—We investigate the approach where two models of different complexities (MobileNet and Xception) are running on edge and cloud, as an alternative to DNN partitioning. Since communication between mobile and cloud nodes is performed only when needed, this is beneficial to reduce both latency and costs.

This article is an extension of a previous work by Magalhães et al. [2019], in which we included the evaluation of a collaborative approach between DNNs as an alternative to the partitioning of DNNs, since our overall results indicate that running the inferences entirely on the edge or entirely on the cloud server are the best options when considering partitioning the workload between the cloud and limited-resource edge devices. The details of the collaborative approach are further detailed in Section 5.

The remaining sections are organized as follows. Section 2 presents a review of related work. Section 3 introduces some concepts related to DNNs. In section 4, we present the DNN partitioning approach, describe the experimental setup, and discuss the experiments' results. The evaluation of the collaborative approach between DNNs of different complexities is presented in Section 5. Finally, we offer our conclusions and directions for future work in Section 6.

2. RELATED WORK

Some recent studies have investigated and proposed ways of distributing the computation of DNNs between mobile devices and edge/cloud servers to improve performance and make more efficient use of computing resources, as discussed next.

Kang et al. [2017] have investigated the benefits of partitioning DNN models at the layer level when considering three types of wireless connection technologies: 3G, 4G, and Wi-Fi. Their results show that, for some DNN models, partitioning can bring benefits in terms of latency and energy

consumption. In contrast, other models will suffer from high latency caused by the transmission of data generated by intermediate layers. A prediction model, named Neurosurgeon, is proposed to dynamically select the best DNN partition points of a given DNN architecture.

Differently from work of Kang et al. [2017], which used traditional DNN models, in this article, we focus our experiment of partitioning on a more recent and more complex model that is mainly designed for very fast and accurate inferences (YOLO). Moreover, instead of considering singleton input instances for measuring inference latency, we consider video stream data captured from a real-world video surveillance scenario. Additionally to 3G, 4G, and Wi-Fi we also consider cable network setups. Finally, instead of considering specialized mobile edge devices for DNN, as Kang et al. [2017] do, we consider generic and minimal ones (i.e. Raspberry Pi 3 B+), which pose an even more challenging scenario for DNN at the edge. As discussed in the next sections, this article contributes with new insights on the trade-offs of DNN edge/cloud partitioning in terms of latency and financial costs.

Teerapittayanon et al. [2017] propose a framework for partitioning DNN models and deploy the partial models on several edge devices. The approach proposed by the authors modifies the DNN model at a structural level by adding early exit points into the network. These changes require retraining the models for each setting. The results show that their proposed method can reduce network communication costs without harming accuracy.

A similar approach is proposed by Hadidi et al. [2019], which aggregates the existing computing power of edge devices in a local network environment by creating a collaborative network. In this scenario, edge devices cooperate to make inferences by applying different approaches for model-parallelism. The results of Hadidi et al. [2019] show that the collaborative network is enhanced by creating a distributed processing pipeline. Differently from these works, we are interested in two other approaches, partitioning of DNN models between edge and cloud, and running models of different complexities on edge and cloud (i.e., low complexity at the edge and high at the cloud), querying the cloud model when needed. This second approach aims to increase the overall accuracy of the application while reducing communication and costs associated with cloud services subscription.

While the studies mentioned above are interested in partitioning DNNs, others focus on compressing the data generated at intermediate layers. For instance, Shi et al. [2019] present a compression technique for partitioned models by applying a state-of-the-art 2-step pruning method to remove unimportant feature maps in each layer. The proposed framework generates a series of pruned DNN models and can automatically choose the best one with the corresponding partition point. The results show that the pruning method can improve end-to-end latency and maintain higher accuracy under limited bandwidth scenarios. In this context, we intend to investigate ways of enhancing the combination of compression with partitioning in future works.

3. BACKGROUND

Since the proposition of the first Artificial Neural Network (ANN) models in the early 1940s and 1950s, the ANN area has taken an enormous leap forward. Initial designs acted as mere Boolean function emulators, composed of just a few neurons and layers. Modern ANN designs encompass sophisticated architectures, containing millions of neurons, organized in tens to hundredths of layers, and capable of performing a wide range of sophisticated machine learning tasks [Bengio et al. 2013].

Deep learning is a term used to describe a broad class of machine learning methods, mostly based on ANN [LeCun et al. 2015]. The existence of multiple layers of neurons and large amounts of free parameters to learn is the most common characteristic of these methods. The term "deep" is associated with the large number of layers a model has. Layers are usually designed to extract and integrate features from the input in a progressive way. As information flows towards the network output, extracted features move from lower to higher abstraction levels.

Recurrent and convolutional networks are among the most popular DNN methods. These networks are playing a fundamental role in the unprecedented advances in signal processing and analysis, such as image, video, and language recognition [Ponti et al. 2017]. A convolutional network is usually formed by several blocks consisting of convolutional (for feature extraction) and abstraction layers (for dimensionality reduction). At the very end of the network, there might be one or more fully connected layers (for regression), which may be followed by a softmax layer (for classification) [Khan et al. 2020]. This type of network can be easily partitioned due to its inherent sequential nature.

4. PARTITIONING OF A DNN MODEL

In this section, we discuss our investigation on the trade-offs involved in the partitioning of complex DNN models. First, we present the target model architecture used in the experiments. Then, we introduce the proposed methodology, present the experimental setup, and discuss the results.

4.1 Model architecture

In this work, we investigated model partitioning using a popular object detection and classification network, named YOLO [Redmon et al. 2016], which is based on a convolutional design. Input images are divided into a grid system, and each grid cell represents a candidate region of detected objects. A grid cell predicts the number of bounding boxes for an object. Bounding boxes are represented by a 5-tuple (x, y, w, h, c), where (x, y) are the coordinates of the object center, (w, h) are the object's width and height, and c is a confidence score. Figure 1 [Redmon et al. 2016] presents the original YOLO architecture.

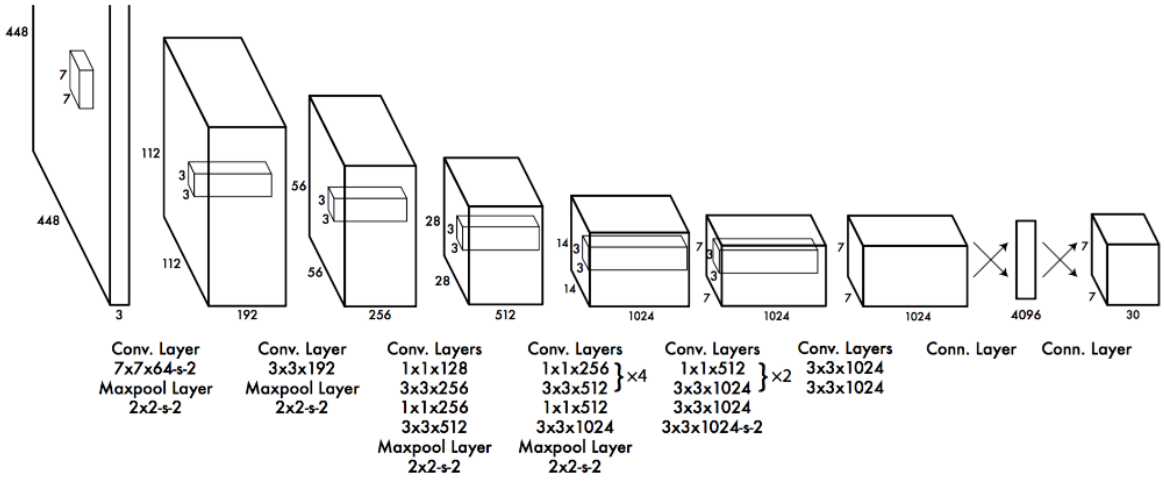

Fig. 1. Diagram of the original YOLO architecture [Redmon et al. 2016], where it is possible to see multiple convolution and maxpooling layers, followed by two fully connected layers near the output.

The main differences between YOLO v1 [Redmon et al. 2016] (original proposal) and v2 [Redmon and Farhadi 2016] (adopted in this article) is the addition of batch normalization layers after all convolutional layers and the removal of the last fully connected layers. An improved anchor boxes strategy is also used for object location. YOLO v2 architecture has a split point after the 13th convolutional layer, which imposes difficulties for partitioning the whole model into partial ones since the input of a given layer after the split point depends not only on the output of an immediately previous layer but also on the outputs of other previous layers. A condensed version of YOLO v2 (for

speed and memory improvements), named YOLO v2 tiny, has half of the original model's layers and receives a smaller input frame.

4.2 Methodology

The approach for partitioning DNN models, as proposed by Kang et al. [2017], consists of dividing a DNN into two partial models. The first partial model runs at the edge device and the second run at the cloud server. The output of the edge's partial model is transmitted and passed as input to the cloud's partial model, so that the output of the partial cloud model is equal to the output of the original DNN model, as shown in Figure 2.

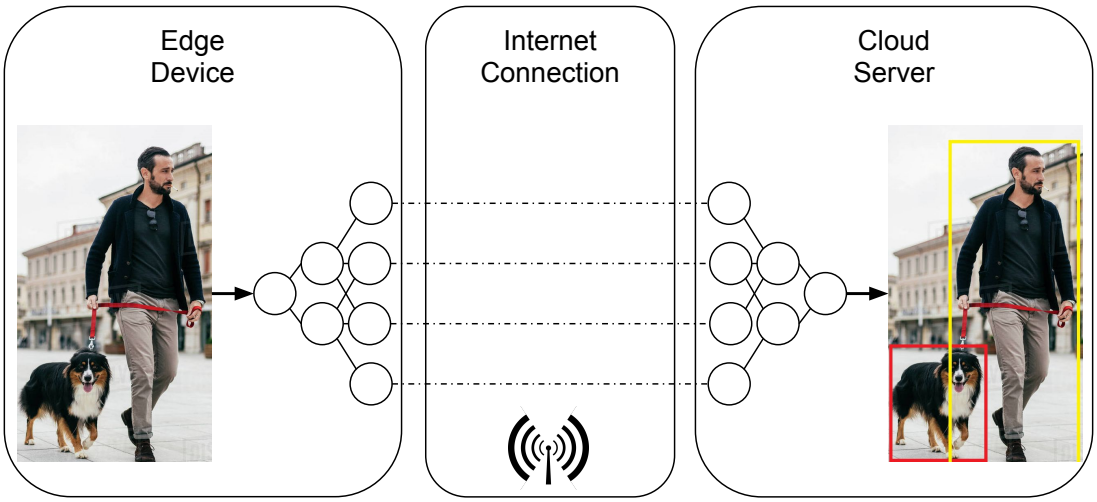

Fig. 2. Illustration of inference in a partitioned DNN model. The input and initial layers run on the edge device. The last layer's activations on the edge device are then transmitted via an internet connection to feed the remaining part of the network (running on the cloud server) to produce the model's output.

For arbitrarily selected DNN models and partitioning points, total inference time is mathematically defined by Equation 1:

$$\mathcal{T}_{d_e,d_c}(m,p,b) = \mathcal{T}_{d_e}(m,p) + \mathcal{T}_{net}(D_{d_e}(m,p),b) + \mathcal{T}_{d_c}(m,p) \tag{1}$$

where $\mathcal{T}_{d_e}$ is the execution time at the edge device, $\mathcal{T}_{net}$ is the time to transmit the data to the cloud server, $\mathcal{T}_{d_c}$ is the execution time at the cloud server, m is the DNN model, p is the partitioning point, d_e and d_c are, respectively, the edge device and the cloud server, D_{d_e} is the amount of data from the edge device output and b is the network bandwidth.

The partitioning method's main goal is to find the partitioning point p that minimizes the total inference time, given the model m, the bandwidth b, and the devices d_e and d_c. However, as seen in Kang et al. [2017], in most cases, the main factor responsible for the increase in total inference time is the network transmission time $\mathcal{T}_{net}$. The transmission time is defined as follows:

$$\mathcal{T}_{net}(D,b) = \frac{D}{b} \tag{2}$$

where D is the amount of data being transmitted. If we set the network bandwidth to an arbitrary value b, the transmission time will increase or decrease directly proportional to the size of D.

Like YOLO, convolutional networks have convolutional layers, which increase the data volume, and pooling layers, which reduce the data volume. Because of these characteristics and to optimize the search for the optimal value of p, we have selected both YOLO v2 and YOLO v2 tiny only the Max Pooling layers as potential partitioning points minimize $\mathcal{T}_{net}$ by reducing the amount of data to be transmitted. Figure 3 shows the amount of data produced as output by each layer in both YOLO v2 full and tiny. In the x-axis, block_n refers to the *nth* functional block composed by a Convolutional layer, a Batch Normalization layer and a Leaky ReLu activation layer, *pool_ n* refers to the *nth* Max Pooling layer and *conv_ n* refers to the *nth* Convolutional layer. To simplify the visualization, we omitted the layers after the split point of the YOLO v2 full architecture since we cannot partition the DNN model after this point.

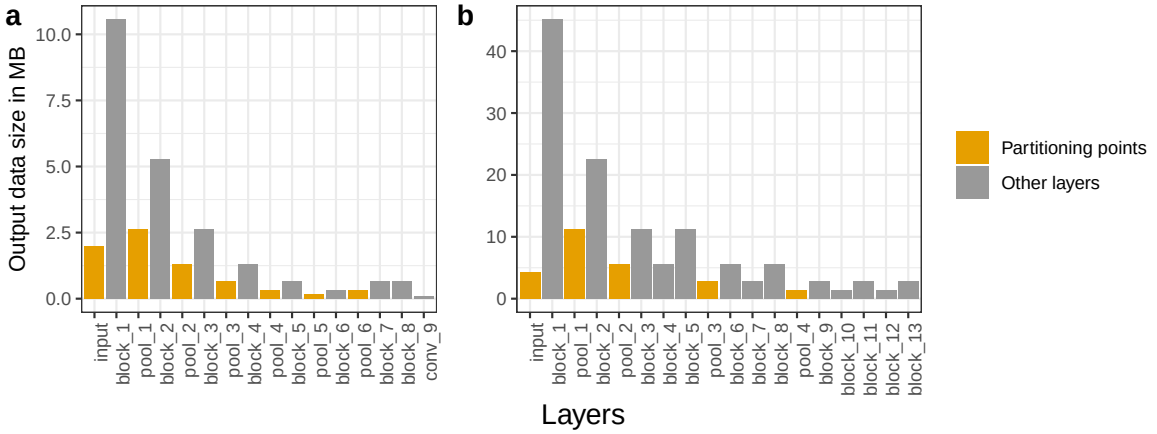

Fig. 3. Output data size for each block or layer (in MB), when considering the models: (a) YOLO v2 tiny and (b) YOLO v2 full. Yellow bars indicate the input layer as well as potential partitioning points (corresponding to smaller data sizes).

4.3 Experimental Setup

All the experiments were conducted using a Raspberry Pi 3 Model B+ as an edge device and a virtual machine at Google Cloud as a cloud server. The edge device has a Quad-core Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4 GHz processor, 1GiB LPDDR2 SDRAM memory and Raspbian Buster Linux Kernel v4.19.0 as the operating system. The cloud server has two vCPUs, 7.5 GiB DIMM memory, NVIDIA Tesla T4 GPU with 2560 CUDA cores, 16 GiB GDDR6 memory, and Ubuntu 16.04 with Linux Kernel v4.15.0 as the operating system.

The communication between the edge device and the cloud server was designed as a client-server application, using the Remote Procedure Call (RPC) protocol. All the code was written using the Python programming language. RPC was implemented using Apache Thrift v0.12.0[1] framework, Keras v2.2.4 [Chollet et al. 2015] and Tensorflow v1.14.0 [Abadi, M. et al. 2015] were used to implement the DNNs and the partitioned models. OpenCV v4.1[2] was used to capture the video streams and pass the frames as input to the DNN models. For the cloud server equipped with GPU, we used cuDNN and CUDA, NVIDIA's libraries that accelerate key DNN layers and optimize the execution time.

The data we have used was captured from an Intelbras VIP 1220 B G2 surveillance camera installed at the entrance hall of our research lab at the Federal University of Campina Grande. Camera specifications are as follows: main video channel with Full HD resolution, secondary video channel

[1] `https://thrift.apache.org/`
[2] `https://opencv.org/releases/`

with HD resolution, a maximum frame rate of 30 fps, RJ45 (10/100 BASE-T) network interface and network throughput of 15 Mbps. The video stream is fed into the YOLO object detection algorithm (YOLO v2 full or tiny) that is deployed according to the various partitioning schemes investigated in this article.

Figure 4 shows an example of an input frame used in our experiments. A bounding box indicates the location of a person that was detected. The input size of the frames passed as input for the YOLO v2 full is 608×608 pixels with 3 color channels and for the YOLO v2 tiny is 416×416 pixels also with three color channels.

Fig. 4. The left image is a sample frame from our setup and the right one is the same frame with a bounding box indicating a detected person with a given confidence level (near the top-left corner of the bounding box).

In our experiments, five random samples of 100 frames are used for assessing the partitioning setups in all network configurations considered. We then evaluate the average performance of each setup over these samples. As in Kang et al. [2017], the connection types used in this work refer to the link between the edge device and the Internet. For instance, for the Wi-fi connection, the edge device is connected to an access point, which then connects at higher speeds with other technologies (e.g. optics fiber) to the Internet. Nevertheless, the total throughput between edge-cloud is limited by the slower connection in the path, in this case, the Wi-fi one. For both 3G and 4G connections we use the mobile network from Claro mobile operator, The cable connection has a 1 Gbps link distributed through an HP 1920-24G switch and for Wi-fi we use a wireless router 802.11 b/g/n with 300 Mbps maximum link speed, connected to the switch. For estimating the transmission rates we used the SpeedTest CLI[3] and calculated the average of 10 test runs for each connection type, the average transmission rates for each internet connection setup we used are presented in Table I.

Table I. Average Transmission rates for the different internet connection setups.

Connection Type	Download (Mbps)	Upload (Mbps)
3G	2.12	2.73
4G	16.10	14.50
Wi-fi	29.03	37.08
Cable	48.46	91.22

[3]https://www.speedtest.net/pt/apps/cli

4.4 Results and Discussion

In Figure 5, we present the results for YOLO v2 tiny, where each bar represents the execution time considering all the partitioning points and the corresponding internet connection setups. Regarding the 3G internet connection, note that the better latencies occur when we execute the inferences entirely at the edge, followed by partitioning on the mp_5 layer, which is the best partitioning point for this setup.

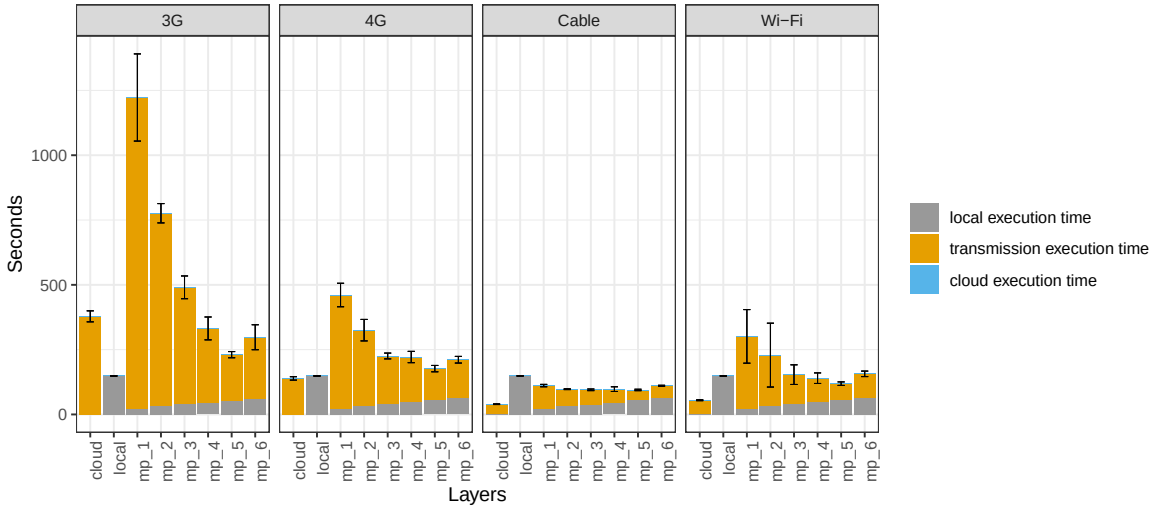

Fig. 5. Average total execution time in seconds using YOLO v2 tiny for each internet connection type (labels at the top of the bars) and partitioning point (horizontal axis). Local-only and Cloud-only execution time were included as well. Cloud execution time (blue) is barely visible in the bars since it is very small compared to the other measurements.

The choice of performing the inferences entirely on the cloud is hardly penalized by the transmission latency, which is a consequence of the low bandwidth of 3G. This behavior changes when we use a 4G connection, where executing either entirely at the edge or the cloud attains the best latencies compared to any partition strategy.

Notice that using a broader bandwidth connection, such as Wi-Fi or Cable, the best alternative is to run everything at the cloud. Also note that in all cases, the execution time at the cloud (blue bar on top of orange bars) corresponds to a tiny fraction of the execution time at the edge. Thus, the main bottleneck of any strategy that uses the cloud lies in the data transmission latency.

In Figure 6, we show the results considering the YOLO v2 full. Due to memory constraints, it is not possible to run YOLO full entirely at edge devices. Thus, any solution needs to either partition the model or to deploy it solely on the cloud. Regarding 3G and 4G internet connections, the results are similar to those observed for the YOLO tiny, with the main difference that the overall latencies are higher. Regarding the cable and Wi-Fi connections, note that the main bottleneck, at least concerning the last partitioning points (i.e., mp_3 onward), is the edge device's execution time. In YOLO v2 tiny, in contrast, the main bottleneck, in almost all cases, is related to the transmission latency. Note that YOLO full is more computationally demanding than its tiny version.

Besides performance, another critical aspect to determine the suitability of any solution for real-world applications is the financial cost. The estimated cost to keep our cloud server running on Google Cloud[4] uninterruptedly for one year is USD 10,465.73 (leaving network usage out of the equation),

[4] `https://cloud.google.com/products/calculator/`

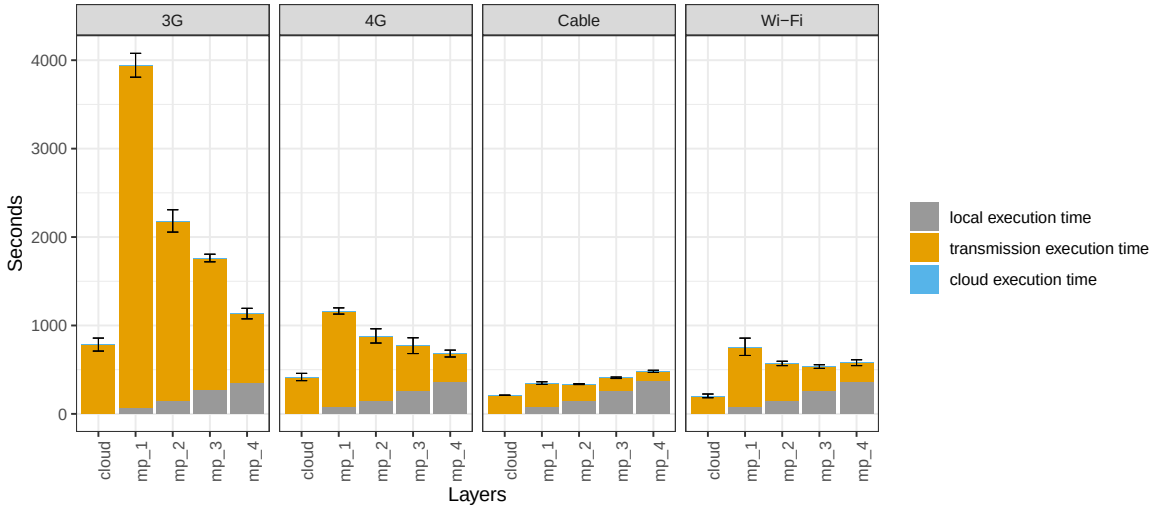

Fig. 6. Average total execution time in seconds using YOLO v2 full for each internet connection type (labels at the top of the bars) and partitioning point (horizontal axis). Cloud-only execution time was included as well. As before, the cloud execution (blue legend) is very small compared to the other measurements). Due to Raspberry Pi 3 B+ memory limitations, Local-only execution was not possible.

of which USD 1,080.37 corresponds to the CPU, RAM and storage costs, and USD 9,385.36 is the GPU usage cost. On the other hand, the cost to acquire and keep our edge device running for the same period is USD 99.29, out of which USD 93.37 is the price of the Raspberry Pi 3 Model B+ kit and USD 5,92 is the estimated energy consumption cost for powering the device in our lab using a standard energy rate from the city of Campina Grande, Brazil.

Figure 7 shows the estimated number of inferences per year using YOLO v2 tiny for each network configuration when transmitting the data from the Raspberry to execute at the cloud server. Notice that these values were estimated considering synchronized procedure calls to the cloud server, which under-utilizes the cloud server computational power due to the high transmission latency.

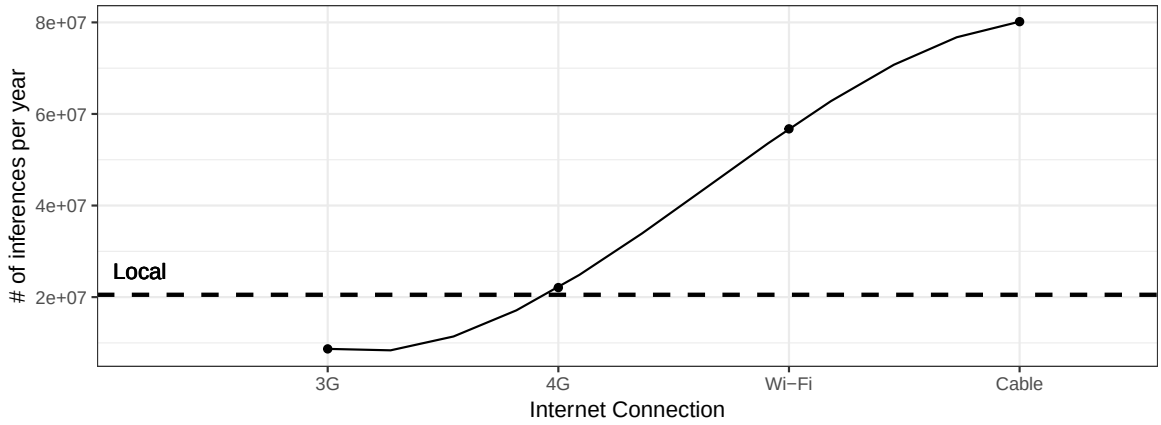

Fig. 7. Estimated number of inferences per year (vertical axis) using YOLO v2 tiny object detection model, for the different evaluated internet connections (vertical axis). The number of local-only inferences is shown as a dashed line.

For a performance evaluation from a computing-intensive perspective, we estimated the number of inferences per USD for edge and cloud counterparts. We found out that running YOLO v2 tiny

on a cloud-only approach, where the data are generated at edge devices and processed remotely on the cloud, we achieve ≈823 inferences per USD[5] invested using 3G, ≈2,090 inferences using 4G, ≈5,371 inferences using Wi-Fi and ≈7,587 inferences per USD using Cable internet connection. In contrast, the Raspberry Pi 3 Model B+ makes ≈ 206,582 inferences per USD invested, and the cloud server makes ≈ 318,205 inferences per USD, both cases running the inferences on data stored locally. Therefore, cloud processing (measured as the number of inferences) is more cost-effective than edge processing in a latency-free scenario, which is unrealistic. So the conclusion is that in realistic scenarios, edge processing is more cost-effective.

5. COLLABORATION BETWEEN DISTINCT DNN MODELS

In this section, we present our experimental evaluation of the proposed collaboration approach between two models: one (simpler) running at the edge and the other (more complex) running at the cloud.

To choose suitable models for edge and cloud inference (in terms of complexity), we analyzed existing Keras Applications models, pre-trained with the ImageNet [Deng et al. 2009] dataset. As the client's goal is to run on edge devices, the model chosen must be light, so we selected MobileNet [Howard et al. 2017; Sandler et al. 2018], which is 16 MB in size, Top-1 accuracy of 0.704 and Top-5 accuracy of 0.895. Differently from the model used in the edge device, the model used by the server must be more robust and accurate, so we chose the Xception [Chollet 2016] with 88 MB in size, Top-1 accuracy of 0.790 and Top-5 accuracy of 0.945. All of this information is available in the Keras Application documentation[6].

In the following subsections, we describe the models architecture, method used, experimental setup and discuss the obtained results.

5.1 Models architectures

MobileNet is a model based on a simplified DNN architecture that uses depth-wise separable convolutions [7] to build lightweight DNNs. It has a reduced number of Multiply-Accumulates (MACs)[8] and a small number of parameters, thus being more compact and less complex than its competitors [Howard et al. 2017; Sandler et al. 2018]. MobileNet is particularly tailored for applications at the edge since the speed, and power consumption are proportional to the number of MACs. This model has 27 convolutional layers, where the last three layers are: average pooling, fully connected, and softmax, respectively. Table II contains the sequence of the layer types, filter shapes, and input sizes of the MobileNet architecture.

As aforementioned, the second model used in this experiment was the Xception [Chollet 2016] (see Figure 8), which, like MobileNet, uses depth-wise separable convolutions. Xception is an adaptation of the Inception [Szegedy et al. 2014] model, in which the main change was the replacement of the Inception modules with depth-wise separable convolutions. The original Inception hypothesis has the following two main features: (i) cross-channel (or cross-feature map) 1x1 convolutions capture correlations; and, consequently, (ii) spatial correlations within each channel are captured via the regular 3×3 or 5×5 convolutions. Therefore, in Xception, 1×1 convolutions are performed for every channel, and then 3×3 convolutions are performed for each output. Moreover, in Xception, the modified depthwise separable convolution perform the 1×1 convolution before the channel-wise spatial convolution. Finally, there is no intermediate ReLU non-linearity in the depth-wise separable convolution adopted by the Xception model.

[5] calculated as the number of inferences per year divided by the annual cost
[6] `https://keras.io/applications/`
[7] i.e., a depth-wise convolution followed by a point-wise convolution
[8] measure of the number of fused Multiplication and Addition operations

Table II. Layers of the MobileNet architecture described in terms of type/stride, filter shape and input size. Data move from top to bottom layers of the table.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

5.2 Method

In the proposed approach, communication occurs when the model running on the edge device has low confidence, so the client node makes a request to the server in the cloud by sending the input data and waiting for the result of its prediction. When using the proposed hybrid approach, our goals are to increase the overall accuracy and reduce communication delays and costs, compared to making all inferences on the edge device. As stated before, a relevant characteristic of this approach is that communication between edge and cloud is on-demand.

Besides accuracy, the total elapsed time is also an essential factor. As in the partitioning experiment, the transmission time is the main factor for increasing the total inference time. As shown in Equation 2, the transmission time is directly proportional to the amount of data transmitted.

In this experiment, the main parameter that influences the amount of data transmitted is the confidence threshold. This parameter indicates when the client should query the server for aid. Therefore, several experiments were carried out with different thresholds to determine which one produces significant improvement in accuracy at a reasonable cost. All the results are shown and discussed in the following sections.

5.3 Experimental Setup

This experiment was conducted using the same settings as in the partitioning experiment (cf Section 4.3). However, the input data was a new one. Instead of video frames captured from a surveillance camera, images came from the ILSVRC2017[9] validation set. The input size of the images passed as input to the MobileNet is 224×224 pixels with 3 color channels. The images used by Xception are 299×299 pixels with 3 color channels.

[9]http://image-net.org/challenges/LSVRC/2017/

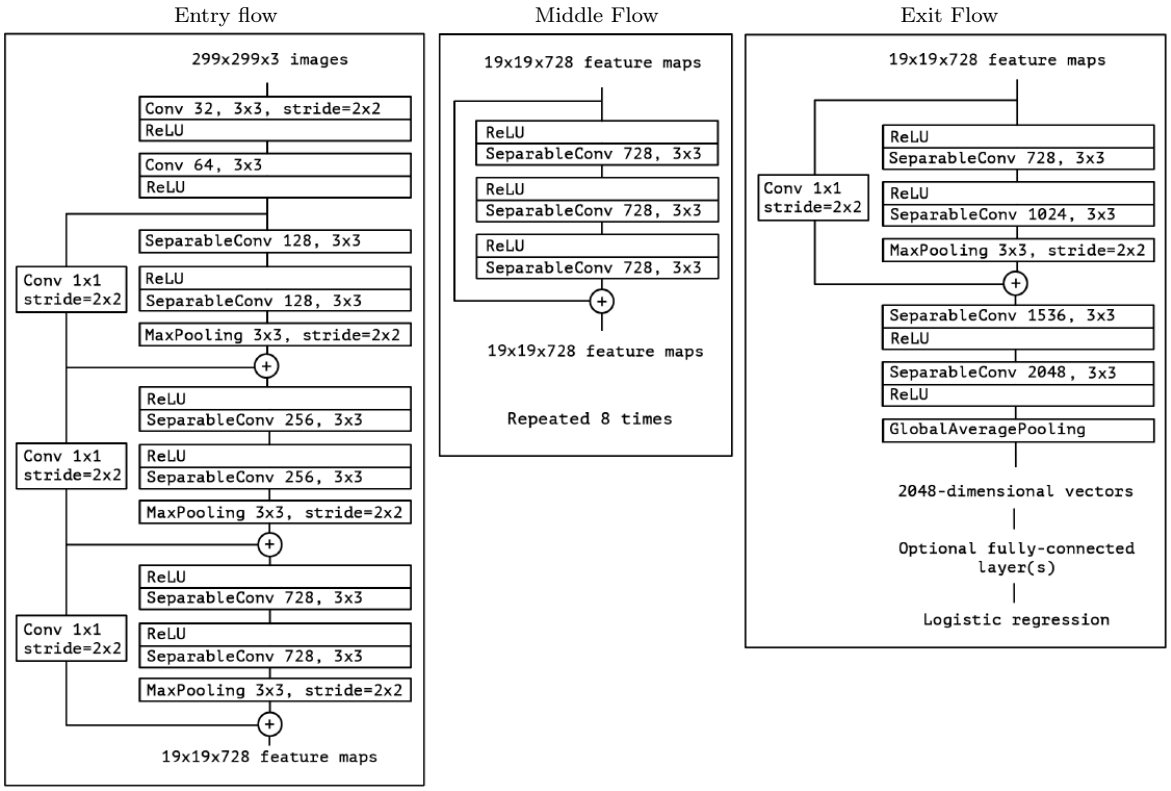

Fig. 8. Diagram of the Xception [Chollet 2016] architecture, depicting the different layer types and dimensions. Input data advances from top to bottom within each flow and from left to right between the flows.

In our experiment, 500 random samples from the ILSVRC2017 validation set are used to assess the final hybrid application's performance, exploring all threshold variations. Thresholds were varied from 0 to 100%, with a step of 5%.

5.4 Results and Discussion

Although accuracy is the primary metric to be analyzed in this experiment, it is also relevant to evaluate the amount of data transmitted, the total execution time, and the financial cost of the application. In Figure 9, we present the amount of data transmitted for all confidence thresholds. As expected, the amount of data transmitted grows as the threshold increases. This happens because the edge device makes more requests to the server hosted in the cloud. However, after the threshold of 30%, the number of bytes transmitted grows almost linearly. Only when the threshold reaches 100%, i.e., when all inferences are made in the cloud, there is a more substantial leap.

Figure 10 shows how the total execution time responds to threshold variation. When observing the first and last bar of the figure, which represents execution only at the edge and only at the cloud, it is possible to notice an increase in the execution time. However, it is the transmission time that causes the greatest impact on the total application time (making the application more costly). Also, the variation in time at the other thresholds is minimal if only the execution time is isolated, with transmission time once again being the major contributor to the increase in total application time.

Differently from the partitioning experiment, accuracy is an important metric to be analyzed in this experiment. In Figure 11, we present the accuracy of the application for all confidence thresholds. As expected, the accuracy improves as the threshold increases. This happens because more inferences

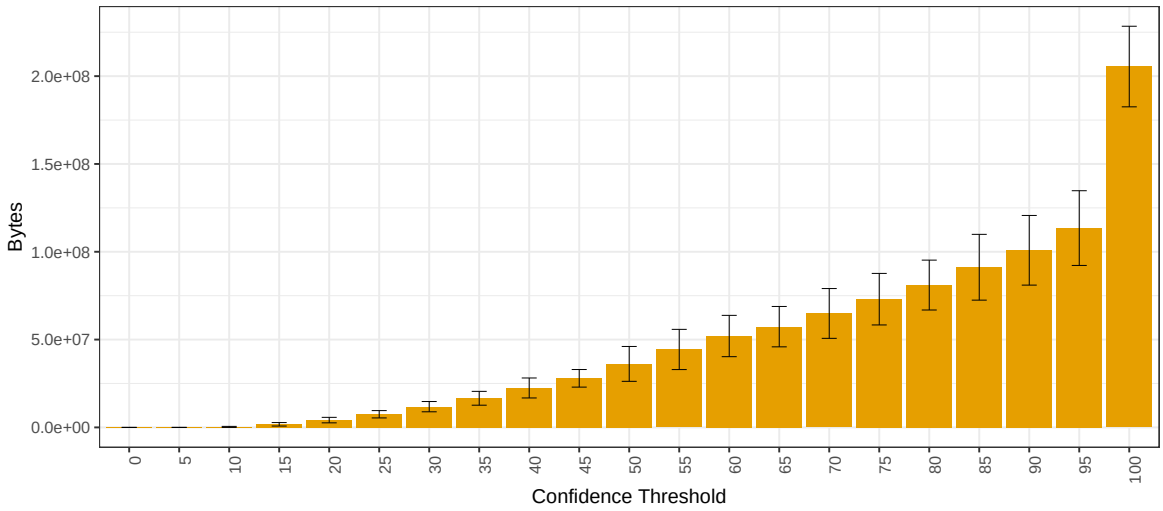

Fig. 9. Bar chart with confidence intervals of the average total amount of data transmitted from edge to cloud (vertical axis), when using MobileNet (edge) and Xception (cloud) for each confidence threshold (horizontal axis). Threshold 0% represents edge-only execution, while threshold 100% represents cloud-only execution.

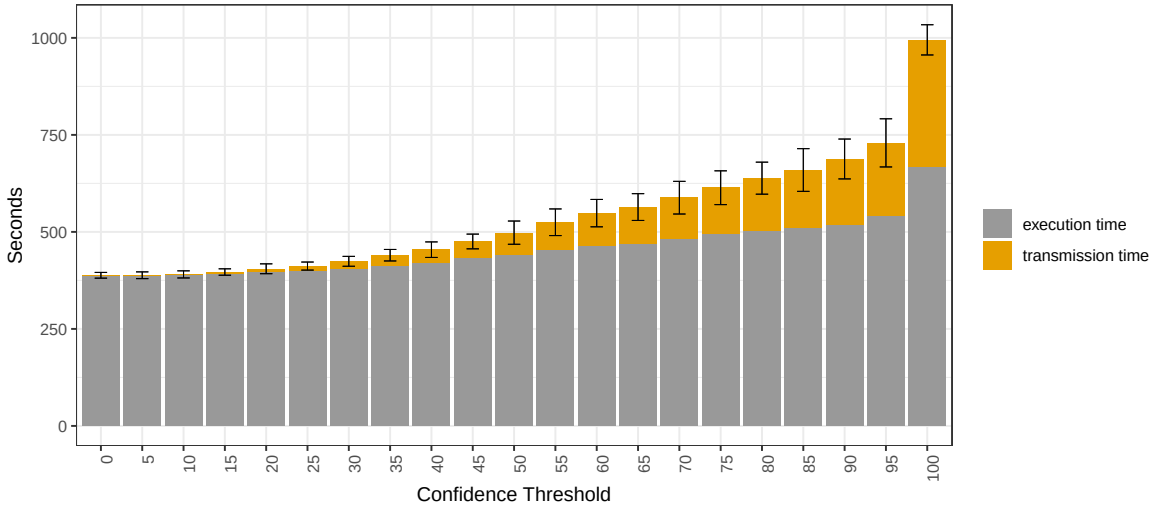

Fig. 10. Bar chart with confidence intervals of the average total execution time in seconds (vertical axis) using MobileNet (edge) and Xception (cloud) for each confidence threshold (horizontal axis). The error bars are split into transmission and execution time, as indicated in the figure legend. Threshold 0% represents edge-only execution, while threshold 100% represents cloud-only execution.

will be executed by the more sophisticated model hosted in the cloud. Analyzing just the accuracy, thresholds above 35% already show an absolute improvement of approximately 3%. Moreover, all thresholds above 40% have improved accuracy in all executions. Note that the dashed line that represents the accuracy of the application running only at the edge is below the standard deviation range after 40%.

The choice of the threshold depends on the context of the application. More specifically, in the case where the MobileNet and Xception models were used, when choosing the threshold of 80%, it is already possible to obtain a general joint accuracy of the application that is the same obtained by

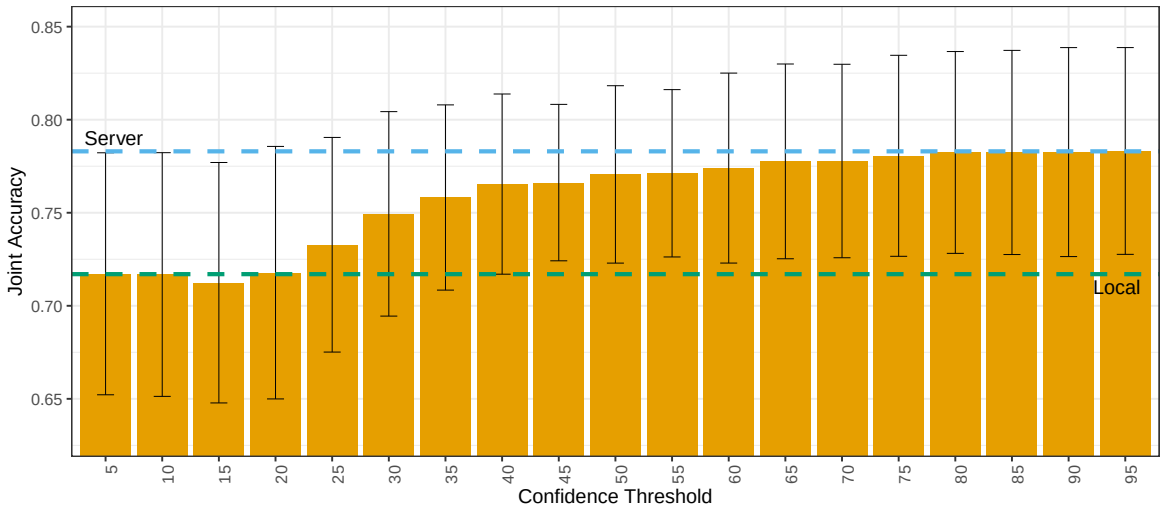

Fig. 11. Bar chart with confidence intervals of the average joint accuracy (vertical axis) using MobileNet (edge) and Xception (cloud) for each confidence threshold (horizontal axis). Local-only accuracy and Server-only accuracy are shown as dashed lines.

the more complex model hosted in the cloud, but with a lower cost, as the volume of data transferred is much smaller (Figure 9). If such an accurate application is not required, and a lower price is desired, a threshold of 50% would suffice (5% increase in accuracy and a small amount of data to be transmitted).

Another determining factor for the viability of an application is its financial cost. As both the hardware and software configurations used were the same for the partitioning experiment, the estimated annual costs for the cloud server and the edge device are the same. The details of the calculations have already been explained in Subsection 4.4. As the collaborative intelligence approach differs from the partitioning approach in that it does not require the cloud server to be queried for every single inference so that the cloud server can be seen as an on-demand inference service. In this way, the cloud computing costs are strictly related to the confidence threshold adopted, which will depend on the level of criticality of the specific application. However, costs considering an on-demand approach have not been estimated since no cloud computing platform offers a service modality that allows deep learning inferences to be executed on-demand.

Figure 12 shows the estimated number of inferences per year using both models MobileNet and Xception for each confidence threshold, when performing inferences on edge, transmitting the data from the Raspberry to the cloud, and executing the workload on the cloud server. In this scenario, the number of inferences per year decreases as the confidence threshold increases. This is because the inference performed on edge is always executed, i.e., if the edge model queries the cloud model, two inferences will be performed. Therefore, the choice of the threshold should also take this measure into account. However, the gain in accuracy at thresholds close to 50% is already quite relevant. Therefore, depending on the application, the proposed hybrid approach is consistent.

6. CONCLUSIONS AND FUTURE WORK

In this work we have investigated trade-offs of applying different approaches to Mobile Edge-Cloud applications. The edge device used was a Raspberry Pi 3 B+ which is more resource-limited and less costly than current smartphones, but has the same processor architecture.

Regarding the partitioning experiment, it is possible to observe that, despite a decrease in execution

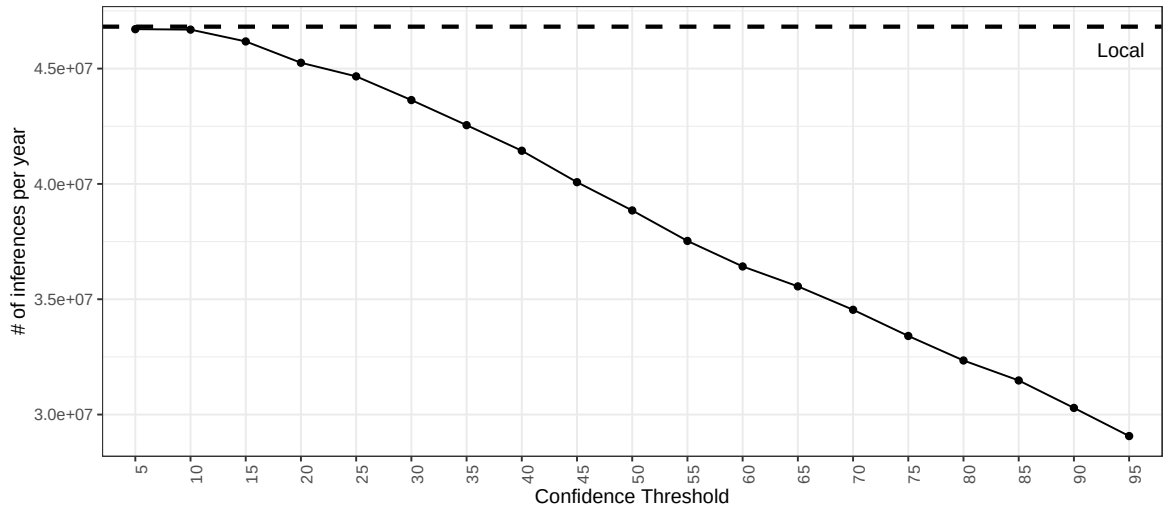

Fig. 12. Estimated number of inferences per year (vertical axis) using the collaboration between the models MobileNet (edge) and Xception (cloud) for each confidence threshold (horizontal axis). The number of inferences for local-only execution is also shown (dashed line).

time for some partitioning points, overall results indicated that running the inferences entirely on the edge or entirely on the cloud server are the best options. If we consider financial costs trade-offs, under the current prices of GPU solutions served on the cloud and realistic scenarios, processing all the workload on the edge device might be more cost effective if the edge device memory can accommodate the entire DNN model.

From the second experiment, in which simpler models running on edge may query a more complex cloud model, whenever their confidence is too low, it is possible to note that this scheme produces a significant increase in accuracy, even for lower confidence thresholds, which makes the inferences more robust without penalizing running costs too much.

Future work will focus on evaluating how model compression, data compression, parallel and distributed computing techniques can be used to improve the overall performance of the object detection task on resource-constrained edge devices, with the aim of achieving real-time requirements.

ACKNOWLEDGMENT

This work was supported by a cooperation between UFCG and Hewlett Packard Enterprise (Hewlett-Packard Brasil Ltda.) using incentives of Brazilian Informatics Law (Law No. 8.2.48 of 1991).

REFERENCES

Abadi, M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Bengio, Y., Courville, A., and Vincent, P. Representation learning: A review and new perspectives. IEEE transactions on pattern analysis and machine intelligence 35 (8): 1798–1828, 2013.

Carneiro, S. A., da Silva, G. P., Guimaraes, S. J. F., and Pedrini, H. Fight detection in video sequences based on multi-stream convolutional neural networks. In 2019 32nd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI). IEEE, pp. 8–15, 2019.

Chollet, F. Xception: Deep learning with depthwise separable convolutions, 2016.

Chollet, F. et al. Keras. `https://keras.io`, 2015.

da Silva Marques, R. C., Serra, A. C., França, J. V. F., Diniz, J. O. B., Junior, G. B., de Almeida, J.

D. S., da Silva, M. I. A., and Monteiro, E. M. G. Image-based electric consumption recognition via multi-task learning. In 2019 8th Brazilian Conference on Intelligent Systems (BRACIS). IEEE, pp. 419–424, 2019.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09, 2009.

dos Santos, F. P. and Ponti, M. A. Robust feature spaces from pre-trained deep network layers for skin lesion classification. In 2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI). IEEE, pp. 189–196, 2018.

Eshratifar, A. E., Esmaili, A., and Pedram, M. Towards collaborative intelligence friendly architectures for deep learning. In 20th International Symposium on Quality Electronic Design (ISQED). IEEE, pp. 14–19, 2019.

Gill, Z. User-driven collaborative intelligence: social networks as crowdsourcing ecosystems. In CHI'12 Extended Abstracts on Human Factors in Computing Systems. pp. 161–170, 2012.

Hadidi, R., Cao, J., Ryoo, M. S., and Kim, H. Collaborative execution of deep neural networks on internet of things devices. CoRR vol. abs/1901.02537, 2019.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, Xi'an, China, pp. 615–629, 2017.

Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. A survey of the recent architectures of deep convolutional neural networks. Artificial Intelligence Review, 2020.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. nature 521 (7553): 436–444, 2015.

Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y., and Shi, W. Edge computing for autonomous driving: Opportunities and challenges. Proceedings of the IEEE 107 (8): 1697–1716, 2019.

Magalhães, W., Gomes, H., Marinho, L., Aguiar, G., and Silveira, P. Investigating mobile edge-cloud trade-offs of object detection with yolo. In Anais do VII Symposium on Knowledge Discovery, Mining and Learning. SBC, pp. 49–56, 2019.

Marinus, J. V. d. M. L., de Araújo, J. M. F. R., and Gomes, H. M. Reconstructed phase space and convolutional neural networks for classifying voice pathologies. In Iberoamerican Congress on Pattern Recognition. Springer, pp. 792–801, 2018.

Misra, A., Jayarajah, K., Weerakoon, D., Tandriansyah, R., Yao, S., and Abdelzaher, T. Dependable machine intelligence at the tactical edge. In Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications. Vol. 11006. International Society for Optics and Photonics, pp. 1100608, 2019.

Pasandi, H. B. and Nadeem, T. Convince: Collaborative cross-camera video analytics at the edge. arXiv preprint arXiv:2002.03797, 2020.

Ponti, M. A., Ribeiro, L. S. F., Nazare, T. S., Bui, T., and Collomosse, J. Everything you wanted to know about deep learning for computer vision but were afraid to ask. In 2017 30th SIBGRAPI conference on graphics, patterns and images tutorials (SIBGRAPI-T). IEEE, pp. 17–41, 2017.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 779–788, 2016.

Redmon, J. and Farhadi, A. YOLO9000: better, faster, stronger. CoRR vol. abs/1612.08242, 2016.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 4510–4520, 2018.

Shi, W., Hou, Y., Zhou, S., Niu, Z., Zhang, Y., and Geng, L. Improving device-edge cooperative inference of deep learning via 2-step pruning. CoRR vol. abs/1903.03472, 2019.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions, 2014.

Teerapittayanon, S., McDanel, B., and Kung, H. T. Distributed deep neural networks over the cloud, the edge and end devices. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). Institute of Electrical and Electronics Engineers (IEEE), Atlanta, USA, pp. 328–339, 2017.

Weerakoon, D., Jayarajah, K., and Misra, A. Resilient collaborative intelligence for adversarial iot environments, 2019.