Investigating Mobile Edge-Cloud Trade-Offs of Object Detection with YOLO

W. F. Magalhães[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 hmg@computacao.ufcg.edu.br lbmarinho@dsc.ufcg.edu.br
[2] Hewlett Packard Enterprise, Brazil
glaucimar@hpe.com plinio.silveira@hpe.com

Abstract. With the advent of smart IoT applications empowered with AI, together with the democratization of mobile devices, moving the computation from cloud to edge is a natural trend in both academia and industry. A major challenge in this direction is enabling the deployment of Deep Neural Networks (DNNs), which usually demand lots of computational resources (i.e. memory, disk, CPU/GPU, and power), in resource limited edge devices. Among the possible strategies to tackle this challenge are: (i) running the entire DNN on the edge device (sometimes not feasible), (ii) distributing the computation between edge and cloud or (iii) 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 a real-world scenario involving object detection from video surveillance feeds. We conduct several experiments on two different versions of YOLO (You Only Look Once), a state-of-the-art DNN designed for fast and accurate object detection and location. Our experimental setup for DNN model partitioning includes a Raspberry PI 3 B+ and a cloud server equipped with a GPU. Experiments using different network bandwidths are performed. Our results provide useful insights about the aforementioned trade-offs.

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

Keywords: deep neural networks, edge-cloud partitioning, object detection

1. INTRODUCTION

Recent advances in the field of Deep Neural Networks (DNNs) have led to the rapid adoption of AI-based solutions by industry in a variety of segments, such as robotics, automobiles, health and safety. In many DNN architectures, the model is fully hosted in the cloud. In an object detection application, for example, cameras send the images to a DNN model on a cloud service, which then perform the inferences and send them back to the client application.

From time to time, the model needs to be re-trained (also in the cloud) to accommodate the addition of new training data. There may be, however, 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. In the context of autonomous cars, for example, a latency that is too high could significantly increase the risk of accidents. In addition, unexpected events, such as the crossing of animals or pedestrians in forbidden places, may occur in a fraction of seconds, demanding real-time response. In addition, when there is a large number of devices connected to the same network, effective bandwidth is reduced due to the inherent competition to use the communication channel. These problems can be significantly reduced if the computing is performed at the edge.

Edge computing refers to computation that is performed locally on edge devices (e.g., desktops, Wi-Fi access points, mobile phones, and cameras). Instead of sending images to the cloud for processing

Copyright©2019 Permission to copy without fee all or part of the material printed in KDMiLe is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

and inference, these are sent to a device built into the camera itself, for example, which is responsible for performing the inferences.

A striking feature of DNN models is the large number of parameters that need to be stored or maintained in memory. Because of this, running these models on mobile edge devices with limited computational resources is not trivial as it incurs high performance and power overhead. Among the possible strategies to tackle this challenge are: (i) running the entire DNN on the edge device (sometimes not feasible), (ii) distributing the computation between edge and cloud, or (iii) 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 a scenario of real-world object detection from camera feeds.

We conduct several experiments on different versions of YOLO (You Only Look Once) [Redmon and Farhadi 2016], a state-of-the-art DNN with proven high speed and accuracy, running on a Raspberry Pi 3 B+ device and/or a cloud server equipped with GPU, with different network bandwidths, and provide useful insights about the aforementioned trade-offs. We complement existing works on this area, most notably Kang et al. [2017], 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, care must 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 even more limited in terms of computational resources than current mobile phones, but more cost effective.
—We conduct our experiments on video streams captured from real-world surveillance cameras, which reflect a realistic use case.

2. RELATED WORKS

Some recent studies have investigated and proposed ways of distributing the computation of deep neural networks between mobile devices and edge/cloud servers in order to improve performance and make more efficient use of computing resources.

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, while 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 the work of Kang et al. [2017], which used traditional DNN models, in this article we focus on a more recent and more complex model that is particularly 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 deep learning, as Kang et al. [2017] do, we consider generic and very limited ones (i.e. Raspberry Pi 3 B+) which pose an even more challenging scenario for deep learning on edge. As it will be 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 at 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 is able to reduce the network communication costs without harming accuracy. A similar approach is proposed by Hadidi et al. [2019], which aggregates existing computing power of edge devices in an 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 the partitioning of DNN models between edge and cloud.

While the aforementioned studies are interested in partitioning DNNs, some studies have tried to improve the benefits of this technique by compressing the data generated at intermediate layers. 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 together with the corresponding partition point. Results show that the pruning method can improve the end-to-end latency and maintain higher accuracy under limited bandwidth scenarios. We intend to investigate ways of improving 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 1940's and 1950's, the ANN area has taken a huge 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, organised in tens to hundredths of layers and capable of performing a wide range of complex machine learning tasks.

Deep learning is a term used to describe a broad class of machine learning methods, mostly based on ANN. The existence of multiple layers of neurons and large amounts of free parameters to learn are 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 deep learning methods. These networks are playing a fundamental role in the unprecedented advances on signal processing and analysis, such as image, video and language recognition. A convolutional network is usually formed by a number of 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).

YOLO algorithm is based on a convolutional design, where an input image is 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 presents the original YOLO architecture.

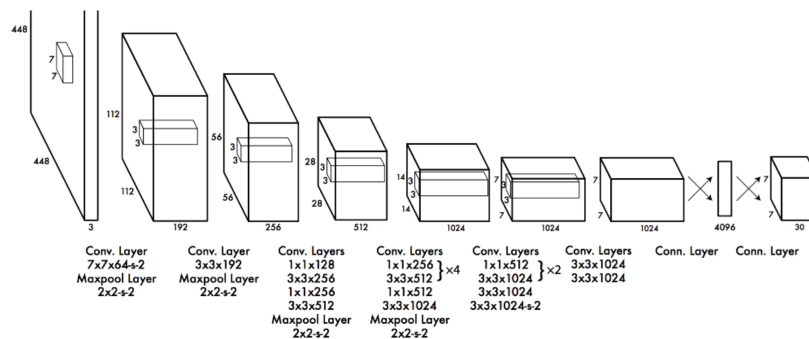

Fig. 1. Original YOLO architecture.

The main differences between YOLO v1 (original proposal) and v2 (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 layers of the original model and receives a smaller input frame.

4. METHOD

The technique 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 runs 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.

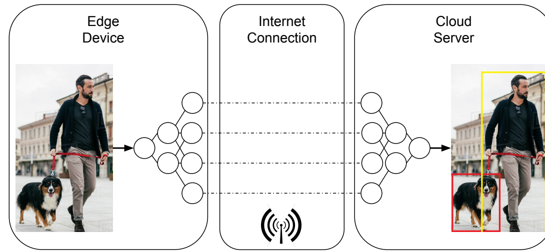

Fig. 2. Illustration of an inference in a partitioned DNN model.

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

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

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 main goal of the partitioning method 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:

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

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.

Convolutional networks, like YOLO, have convolutional layers, which increase the data volum, 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 for both YOLO v2 and YOLO v2 Tiny only the Max Pooling layers as potential partitioning points, in order to minimize $\mathcal{T}_{net}$ by reducing the amount

of data being 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.

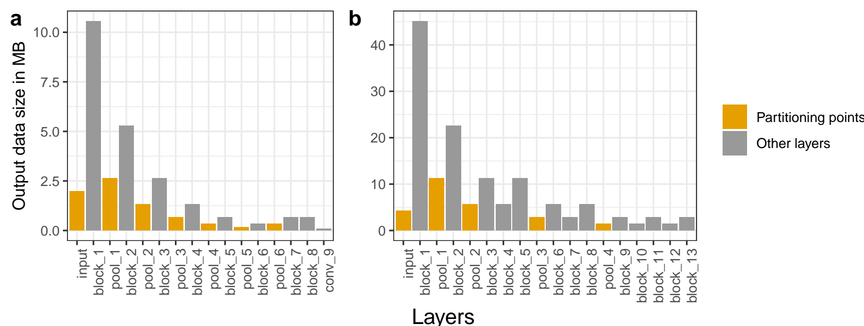

Fig. 3. Output data size for each block or layer (in MB): (a) YOLO v2 tiny and (b) YOLO v2 full. Yellow bars are the potential partitioning points and the input data size.

5. EVALUATION

In this section we evaluate the DNN partitioning technique in terms of performance gain and financial cost using the YOLO v2 full and YOLO v2 tiny models. In the following subsections, we describe the experimental setup and discuss the obtained results.

5.1 Experimental Setup

All the experiments were conducted using an Raspberry Pi 3 Model B+ as edge device and a virtual machine at Google Cloud as 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 operating system. The cloud server has 2 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 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 deep neural networks 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 an university laboratory. Camera specifications are as follows: main video channel with Full HD resolution, secondary video channel with HD resolution, 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.

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

Figure 4 shows an example of an input frame from our scenario. 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 with 3 color channels.

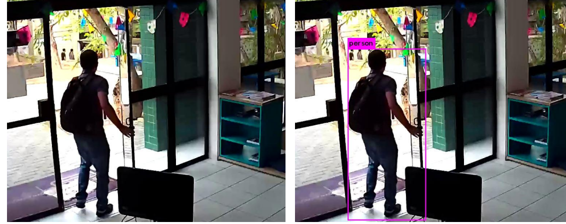

Fig. 4. Sample frame from our setup with a bounding box indicating a detected person.

In our experiments, 5 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. The average transmission rate for each internet connection setup we used are as follows:

—**3G:** Download: 2.12 Mbps; Upload: 2.73 Mbps;
—**4G:** Download: 16.10 Mbps; Upload: 14.5 Mbps;
—**Wi-fi:** Download: 29.03 Mbps; Upload: 37.08 Mbps;
—**Cable:** Download: 48.46 Mbps; Upload: 91.22 Mbps.

5.2 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 corresponding internet connection setups. Considering the 3G internet connection, it is noteworthy that the better latencies occur when we execute the inferences completely on the edge, followed by partitioning on the mp_5 layer, which is the best partitioning point for this setup.

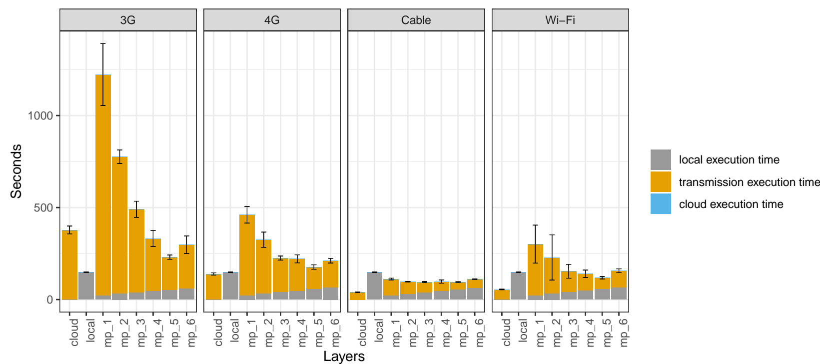

Fig. 5. Average total execution time in seconds using YOLO v2 tiny for each internet connection type and partitioning point. Local-only and Cloud-only execution time were included as well.

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 on the edge or on the cloud attain the best latencies in comparison to any partition strategy.

Notice that by using a wider bandwidth connection, such as Wi-Fi or Cable, the best alternative is to run everything on the cloud. Also note that in all cases, the execution time in the cloud (blue bar on top of orange bars) corresponds to a tiny fraction of the execution time on the edge. Thus, the main bottleneck of any strategy that uses the cloud lies on 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 on the edge device. Thus, any solution needs to either partition the model or to deploy it entirely on the cloud. Regarding 3G and 4G internet connections, the results are similar to the ones observed for the YOLO tiny, with the main difference that the overall latencies are higher. Regarding the cable and Wi-Fi connections, it is noteworthy that the main bottleneck, at least concerning the last partitioning points (i.e., mp_3 onwards), is the execution time on the edge device, whereas in YOLO v2 tiny, the main bottleneck, in almost all cases, is related to the transmission latency. This is due to the fact that YOLO full is more computationally demanding than its tiny version.

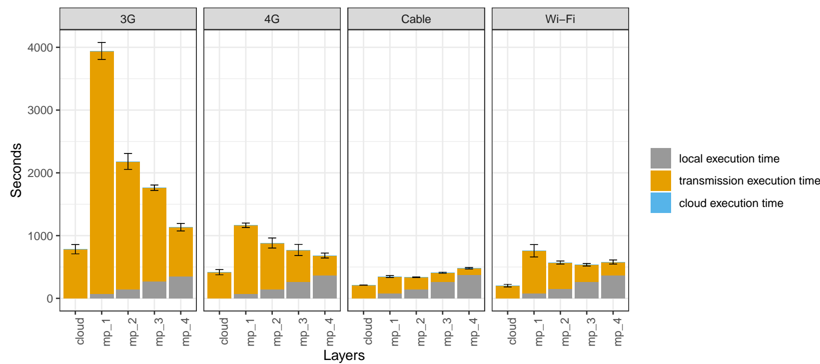

Fig. 6. Average total execution time in seconds using YOLO v2 full for each internet connection type and partitioning point. Cloud-only execution time was included as well. Due to Raspberry Pi 3 B+ memory limitations, Local-only execution was not possible.

Besides performance, another key 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[3] uninterruptedly for a period of one year is USD 10,465.73 (leaving network usage out of the equation), 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 of time 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.

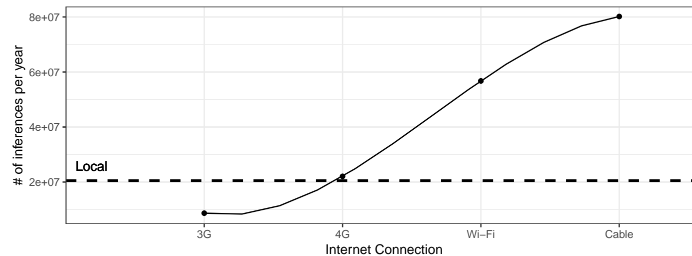

Fig. 7. Estimated number of inferences per year using YOLO v2 tiny object detection model. The number of local-only inferences is also shown (dashed line).

[3] https://cloud.google.com/products/calculator/

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 and executing the workload on 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.

For a performance evaluation from 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[4] 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, of course, unrealistic. So the conclusion is that on realistic scenarios, edge processing is more cost effective.

6. CONCLUSIONS

In this work we have investigated trade-offs of applying DNN partitioning to object detection using YOLO v2, a state-of-the-art algorithm for this task, with real-world data and a Raspberry Pi 3 B+ which is more resource-limited and less costly than actual smartphones, but has the same processor architecture. Despite the reduction of the execution time for some partitioning points, overall results indicated that executing the inferences entirely on the edge or entirely on the cloud server is still better. 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 can fit the entire DNN. Future work might 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 device, 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.

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

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.

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.

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

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.

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.

[4]calculated as the number of inferences per year divided by the annual cost