Training on the Edge: The why and the how

Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Hückelheim, Nicola Ferrier, Paul Hovland, Gerard Gorman

Training on the Edge: The why and the how

Navjot Kukreja[1], Alena Shilova[2], Olivier Beaumont[2], Jan Hückelheim[1],
Nicola Ferrier[3], Paul Hovland[3], Gerard Gorman [1]

***Abstract*— Edge computing is the natural progression from Cloud computing, where, instead of collecting all data and processing it centrally, like in a cloud computing environment, we distribute the computing power and try to do as much processing as possible, close to the source of the data. There are various reasons this model is being adopted quickly, including privacy, and reduced power and bandwidth requirements on the Edge nodes. While it is common to see inference being done on Edge nodes today, it is much less common to do training on the Edge. The reasons for this range from computational limitations, to it not being advantageous in reducing communications between the Edge nodes. In this paper, we explore some scenarios where it is advantageous to do training on the Edge, as well as the use of checkpointing strategies to save memory.**

I. Introduction

Edge computing is a paradigm where computing capability is distributed across a large number of small devices instead of being concentrated in centralized "cloud" systems. This places more computing capability closer to either the user, or the source of the data [9] As it gets cheaper to have additional compute capability in devices like cellphones, cameras and environmental sensors, it becomes viable to do more processing on these devices. The ability to do processing at the Edge can be useful for many reasons. By running computations at the Edge, the latency on these computations can be reduced. This can be useful where a quick result is important, e.g. self-driving cars. By decreasing communication, we reduce the bandwidth and power requirements on the edge nodes - and thus the cost of infrastructure setup. The decentralization that comes with having a few hundred devices also increases the reliability of the system since a centralized system is likely to have a single point of failure. Another reason for doing more computation on the Edge is privacy, since in-situ processing avoids sending potentially sensitive information over a network [16]. For all these reasons, it is becoming commonplace to do inference on machine learning models on the Edge. In Section II, we discuss a project that uses this model extensively.

While it is common to do inference on the Edge today, performing training on the Edge is still not a common paradigm. There are many reasons for this. Firstly, if the information to be learned is relevant to the other Edge nodes, transferring a model update back and forth between the different nodes might introduce excessive communication and increase bandwidth requirements and latency. In this scenario, it might be more efficient to do the training centrally and only transfer the updated model to the Edge nodes. Secondly, machine learning models are not commonly trained on the Edge since the cost of transferring training data to the edge node is much higher than just transferring a trained model. This does not apply when the data is collected on the node itself, and automatically labelled so it can be used in training. We discuss such a scenario in Section III. Even if we can get enough prior training data onto the Edge node and any additional data being captured on the node is only relevant to training the current node, there is still the issue of limited computational capabilities of an Edge node. We detail this problem in Section III and solutions in Section IV.

II. Array of things and the Waggle platform

Array of Things is an Internet-of-Things project that uses an array of hundreds of sensors that work to collect data as a single unit, much like a telescope array, but with sensors collecting data about a city instead. Under this project, hundreds of smart-sensor nodes have been placed all over the city of Chicago. These nodes integrate air-quality sensors with cameras and on-board computational capability to create a distributed and integrated, city-wide network of smart-sensors that can be programmed and controlled as a single instrument to capture data for scientific research [4].

The individual Edge nodes of this network are based on the Waggle platform [2][1], which is designed as an embedded system with sensing and edge computing capabilities. It packages sensors for environmental measurements like pressure, temperature, humidity, light (IR/UV), sound level, gas levels, along with a camera, and three single-board computers (SBCs) into one node. Only one of these SBCs is meant to run edge computing payload, while the others are for node management and reliability. The current payload SBC is an ODROID XU4 based on the Samsung Exynos5422 CPU with four A15 cores, four A7 cores, a Mali-T628 MP6 GPU that supports OpenCL, 2GB LPDDR3 RAM, and attached flash storage. Due to the limited computational capacity of the Odroid platform, other options are being considered. These nodes are already running OpenCV, Caffe and Tensorflow.

With a camera and on-board computational capabilities, an obvious use is to run visual machine learning models, for example, to count the number of people in an area to understand the usage of streets and public spaces, or the number and types of cars passing by. This platform is also

[1]Department of Earth Science and Engineering, Imperial College London n.surname@imperial.ac.uk
[2] Inria Bordeaux, France name.surname@inria.fr
[3]Argonne National Laboratory, IL, USA

[1]http://wa8.gl/

being used for flood and ice detection. All these applications, however, currently only do inference on these Waggle nodes and no training. Some of the reasons for this have been explained in Section I. However, most of these visual models suffer from the viewpoint problem where the images on which these models are operating are from an angle which is specific to the installation of the particular camera/node (see Section III).

III. The viewpoint problem and in-situ student-teacher training

The viewpoint problem is a common problem in computer vision, faced when training an image classification or segmentation model on a data set. One example is a face recognition model. If all the facial images used to train the model are taken at eye level and with the subject directly facing the camera, the model will be trained to recognize faces in images taken at similar angles only and may not be effective on images taken from different angles. This model suffers from the viewpoint problem if used directly. In this approach, we use it as a "teacher" model instead.

In the context of the Waggle platform, although this teacher model may not be able to detect faces at certain skewed angles, it may still work at other angles that are closer to the original training angle. For example, let us assume that a subject walks from the left to the right edges of the frame, and the teacher model correctly identifies it in the last frame. Having received this identification, an object-tracking model [3] can be used to identify and segment all the previous frames which contain the same subject. These frames are then set aside, along with the identified label, as part of a new training set. Every such instance of the teacher model identifying a subject contributes tens of images to this new dataset, which can then be used to train a new "student" model. This student-teacher paradigm has previously been used to compress large networks into more parameter-efficient networks. [7]

Since these images will be used to train a convolutional neural network, they do not need to be stored at a very high resolution. At the standard resolution of 224×224, the size can be expected to be less than 10kb per image. Storing even about 100,000 of these images would require about 10GB of local storage, which is easily provided on an SD card - present on Waggle nodes. Since the training of the student model is not time critical, it can then be scheduled to run only when the node's CPU does not have a higher priority task. Doing this, we can specialise the model running at each node to its own viewpoint, automatically improving its accuracy.

Although in-situ training might be useful to address the viewpoint problem, there are computational issues - specifically, vision models typically require a large amount of memory. Tables I, II and III detail the amount of memory required for the whole model and activations for a few variations of ResNet, since that is a popular model for vision problems.

It can be seen in Table I that all models fit in 2GB memory, without taking into account the memory needed to perform computations. However, increasing the batch size to 3 makes it impossible to keep ResNet152 in memory and further increase makes even the smallest models require more than 2GB. Since training models using extremely small batch sizes is inefficient because of a large number of minibatches per epoch [14], finding a way to increase the batch size while keeping the model in memory can improve training performance.

	ResNet$_x$				
batch_size	$x = 18$	$x = 34$	$x = 50$	$x = 101$	$x = 152$
1	230.05	413.00	620.27	1027.21	1410.62
3	340.05	580.42	1091.11	1732.33	2405.14
5	450.06	747.85	1561.94	2437.45	3399.67
10	725.07	1166.42	2739.04	4200.25	5885.98
30	1825.13	2840.70	7447.42	11251.43	15831.23
50	2925.18	4514.97	12155.79	18302.62	25776.48

TABLE I: Memory requirement for each model to keep all weights and activations for the standard size of image (224×224), the amount is given in MB. The shaded values correspond to the cases when the model cannot fit in memory.

	ResNet$_x$				
image width/height	$x = 18$	$x = 34$	$x = 50$	$x = 101$	$x = 152$
224	230.05	413.00	620.27	1027.21	1410.62
350	309.83	534.96	964.66	1543.72	2139.75
500	449.21	749.73	1570.93	2472.72	3458.50
650	639.07	1039.08	2387.54	3682.00	5161.76
1100	1496.10	2346.95	6073.06	9208.30	12961.96
1500	2628.70	4075.07	10944.42	16515.11	23277.27

TABLE II: Memory requirement for each model to keep all weights and activations for the batch_size = 1, the amount is given in MB. The shaded values correspond to the cases where the model cannot fit in memory.

At the same time problems with memory could also emerge for images with higher resolution than the standard one (224×224) as it follows from Table II, even for the smallest batch_size. The situation becomes worse for batch_size = 8, when one cannot use a neural network with more than 50 layers even for the smallest possible image size.

IV. Related work on reducing memory consumption for backpropagation

Training a neural network through backpropagation has a characteristic data flow pattern where the data corresponding

	ResNet$_x$				
image width/height	$x = 18$	$x = 34$	$x = 50$	$x = 101$	$x = 152$
224	0.60	0.98	2.22	3.41	4.78
350	1.22	1.93	4.90	7.45	10.47
500	2.31	3.60	9.63	14.69	20.76
650	3.79	5.86	15.99	24.13	34.06

TABLE III: Memory requirement for each model to keep all weights and activations for the batch_size = 8, the amount is given in GB. The shaded values correspond to the cases where the model cannot fit in memory.

to the neuron activations is generated while propagating forward through the network. This is followed by a backpropagation pass that calculates derivatives, using the activation values computed earlier. This means that a simple implementation of backpropagation would require all the activations to be stored in memory during the forward pass, in order to use them again during backpropagation.

Sometimes, the memory required to do this is not available. Memory issues are not uncommon, even when training on the largest available commodity GPUs today - the batch size is often adjusted so that a single batch can fit in memory - however the batch size also affects the convergence properties of the training. This problem is especially intensified when training on the Edge when the available batch sizes might be as low as 1-2 (if at all). [12] In recent times, multiple studies have addressed these memory concerns. One technique is model parallelism, where a big model is split over multiple nodes in a cluster [17], [13]. However this is only applicable when the training is done on a cluster with a high-speed interconnect as otherwise communication overheads quickly dominate.

Another technique that has garnered attention recently is checkpointing, also sometimes called reforwarding. In this approach, only a subset of activations is stored during the forward pass, and the rest discarded. The discarded data can be recovered by rerunning the forward propagation from the last available "checkpoint". [8] Although most major neural network training packages today have some implementation of checkpointing [6], [11], [5] these implementations are very basic and do not take advantage of the research that was done on this topic in the fields of high performance computing and automatic differentiation. This means that some of these implementations might be doing more computations or using more memory than strictly necessary. We discuss this in Section VI. While the suboptimality of these implementations might not be immediately obvious when training on a cluster, more efficient implementations are required when dealing with hardware that is highly limited in its computational powers - not only does it not have enough memory to store the entire model, its CPU is small enough that the effect of suboptimal recomputation will be more obvious.

V. Existing checkpointing implementations in machine learning frameworks

PyTorch is a fast-evolving Python package widely applied in deep learning. It is developed by Facebook's artificial-intelligence research group. It shares some features with another popular package called TensorFlow. Both use Tensors as a basic class and all operations are performed on them. The structure of Tensors is similar to the one used in NumPy library with the same basic functions and operations, while also allowing to work with them on GPU. A key difference is Pytorch's ability to dynamically define the computational graph of the model. That renders models flexible, allowing them to be changed during training. Additionally, PyTorch is considered more transparent and more Python oriented.

PyTorch is actively being developed and is designed to be memory efficient to allow executing larger models. To achieve this memory efficiency several techniques are applied, including data-flow analysis, data parallelism and checkpointing, as discussed in Section IV. The current implementation called *checkpoint_sequential* divides the whole network in parts that are all equal except the last one. The number of such parts is determined by the parameter *segments*. Hence, during the forward propagation phase only the inputs of first segments are saved for the backward propagation phase and the last segment is treated as usual, i.e. all activations are stored. So during the backward phase the last segment could be processed immediately while for others it is necessary to recompute activations starting from checkpoints in order to proceed.

For better understanding of how much memory consumption could be reduced it is enough to consider the following example. Assuming that there is a neural network which could be divided in l homogeneous blocks in terms of operation costs and activation sizes and s corresponds to the number of segments described above, then the total memory taken by all activations could be defined by

$$\text{Memory} = s - 1 + \big(l - \lfloor l/s \rfloor (s-1)\big).$$

It can be seen that there is a lower bound $L = 2\sqrt{l}$ and for any $s \geq 2$ it is not possible to reduce $Memory$ below this value. On the other hand, for Edge devices it can be crucial if large models are used (see Section III). Therefore, in case of bigger models another approach should be applied like binomial checkpointing discussed in more details in Section VI.

VI. Proposed Improvements

In this section, we evaluate the practical advantages of optimal checkpointing.

In order to establish this, we will base our analysis on the memory requirements of different ResNet networks, different batch size and different image sizes, as given in Tables I and II. To simplify the analysis here, we will denote by LinearResNet$_x$ a linear homogeneous network built by analogy to ResNet$_x$. The memory needed to store all network weights is the same in LinearResNet$_x$ and in ResNet$_x$, and the size of the forward activation for a given image size in LinearResNet$_x$ is defined as the overall activation weights for ResNet$_x$ divided by the depth of ResNet$_x$. Thus, we obtain a linear homogeneous version LinearResNet$_x$ of ResNet$_x$, that has approximately the same memory requirements as ResNet$_x$, both in terms of weights and activations.

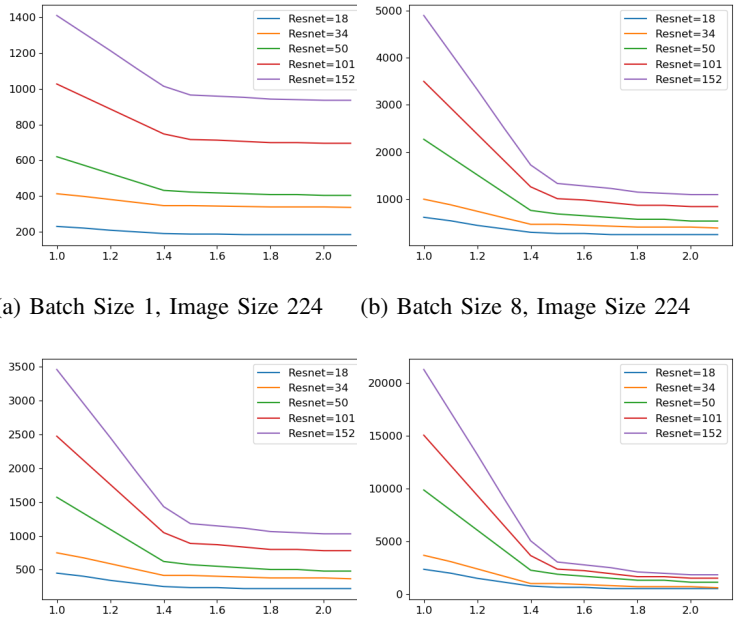
Considering a LinearResNet of depth l, let us denote by M_C the size of the memory of the Edge device, by M_W the memory needed to store all the weights, and by M_A the memory to store the result of any forward step (remember that all the layers in the LinearResNet have the same size) with a unit batch size, and let us denote by k the batch size. Then, $n_{\max} = \frac{M_C - M_W}{k \times M_A}$ represents the depth of the largest ResNet that can be trained in the given amount of memory for the associated batch size k. In order to increase

$n_{\max}$, the solution that is used in practice often consists in using a smaller batch size k, which may affect the time to complete an epoch. On the other hand, for $n_{\max}$ greater than 3, it is possible to rely on checkpointing in order to perform training. In fact, given $n_{\max}$, it is possible to determine in polynomial time the optimal dynamic sequence of checkpoints, using the dynamic programming algorithm Revolve [10], [15], [1]. The produced computation schedule is recursive, in the sense that the same memory slot is used to store activations from different layers at different times. We refer the reader to [15] for the details of the algorithm, and our goal in this section is rather to show that optimal checkpointing is very efficient in drastically limiting required memory, while only reasonably increasing the processing time.

Let us denote by ρ the increase in the time for a single backpropagation, i.e. the recompute factor, that is acceptable in our specific context. Thus, the maximal number of forward and backward computations is $2\rho l$. Combining PyRevolve, that computes the optimal schedule that minimizes the time to solution given a number of checkpoint slots to an elementary binary search, we can easily compute the minimal number of checkpoint slots so that the time to solution is smaller than $2\rho l$. Figure 1 depict the evolution of the memory footprint with ρ for different LinearResNet networks. Each plot corresponds to a specific image size, either small (224×224) for a respective batch size of 1 (Figures 1a) and 8 (Figures 1b)and medium (500×500) for a respective batch size of 1 (Figures 1c) and 8 (Figures 1d).

$\rho = 1$ corresponds to the case with no checkpointing. In this specific case, we can observe that all models and activations can fit into the 2GB limit only if the image size is 224. In all other cases (larger batch sizes or larger image sizes), the memory is too limited to store and run the models. On the other hand, considering a value of ρ between 1.5 and 2 dramatically changes the situation. The lower memory consumption can be used to consider larger batch sizes. For instance, when the batch size is 8 (Figure 1d), all models fit into the 2GB memory with $\rho > 1.6$, whereas in the same context, even ResNet_{18} does not fit into the 2GB limit.

Moreover, the effective increase in the total time to solution is likely to be smaller than what is shown in above results because a larger batch size will enable fewer batches per epoch. Also, on the typical multi-threaded vector architectures (such as GPUs), having a larger batch-size enables to increase the computational efficiency, and therefore, the time to process 8 times a batch size of 1 is expected to be much larger than the time to process a batch size of 8.



(a) Batch Size 1, Image Size 224 (b) Batch Size 8, Image Size 224

(c) Batch Size 1, Image Size 500 (d) Batch Size 8, Image Size 500

Fig. 1: Peak memory requirement vs recomputate factor for different image sizes. The recompute factor is the ratio between the extended time to solution due to recomputations induced by the memory-saving checkpointing and the original time to solution. We can see that for small recompute factors, the memory requirement is often prohibitively high, especially for an Edge node

VII. Conclusion

We have considered the opportunity of performing training on Edge devices, especially in the context of the viewpoint problem. A student-teacher model pair is a possible approach whereby the viewpoint problem can be addressed by in-situ training of a model specialized to each camera's viewpoint. This approach does not require any additional data to be transferred to the node beyond the original teacher model. We have also shown that the peak memory footprint, which is a crucial factor for training on Edge devices, can be reduced by checkpointing strategies. We show that the current implementations of checkpointing in popular neural network packages can be improved by taking advantage of full binomial checkpointing and that the impact of this improvement would be most useful for training on the Edge.

Acknowledgments: This paper benefited greatly from discussions with Paul Kelly and Prasanna Balaprakash. This work was partly funded by the Intel Parallel Computing Centre at Imperial College London and EPSRC EP/R029423/1. This work was partly funded by HPC-BigData INRIA Project LAB (IPL).This work was funded in part by a grant from U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH1135.

References

[1] `https://gitlab.inria.fr/adjoint-computation/disk-revolve-public`.

[2] P. Beckman, R. Sankaran, C. Catlett, N. Ferrier, R. Jacob, and M. Papka. Waggle: An open sensor platform for edge computing. In *SENSORS, 2016 IEEE*, pages 1–3. IEEE, 2016.

[3] A. Brunetti, D. Buongiorno, G. F. Trotta, and V. Bevilacqua. Computer vision and deep learning techniques for pedestrian detection and tracking: A survey. *Neurocomputing*, 300:17–33, 2018.

[4] C. E. Catlett, P. H. Beckman, R. Sankaran, and K. K. Galvin. Array of things: a scientific research instrument in the public way: platform design and early lessons learned. In *Proceedings of the*

2nd International Workshop on Science of Smart City Operations and Platforms Engineering, pages 26–33. ACM, 2017.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] E. J. Crowley, G. Gray, and A. J. Storkey. Moonshine: Distilling with cheap convolutions. In *Advances in Neural Information Processing Systems*, pages 2893–2903, 2018.

[8] J. Feng and D. Huang. Cutting down training memory by re-fowarding. *arXiv preprint arXiv:1808.00079*, 2018.

[9] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[10] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[11] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[12] J. Hanlon. How to solve the memory challenges of deep neural networks, Aug 2018.

[13] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.

[14] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[15] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman. High-level python abstractions for optimal checkpointing in inversion problems. *arXiv preprint arXiv:1802.02474*, 2018.

[16] G. Paul and J. Irvine. Privacy implications of wearable health devices. In *Proceedings of the 7th International Conference on Security of Information and Networks*, page 117. ACM, 2014.

[17] M. Wang, C.-c. Huang, and J. Li. Supporting very large models using automatic dataflow graph partitioning. *arXiv preprint arXiv:1807.08887*, 2018.