Asynchronous layerwise deep learning with MCMC on low-power devices

M François, P Gay, S Lebeaud, S Loustau, J Palafox, F K Sow, N Tirel

Asynchronous layerwise deep learning with MCMC on low-power devices

M.François[1], P.Gay[1], S.Lebeaud[1], S.Loustau[1], J.Palafox[2], F.K.Sow[1], and N.Tirel[1]

[1]Université de Pau et des Pays de l'Adour, Laboratoire de Mathématiques et de leurs Applications, Green AI, France
[2]CY Tech, Département de Mathématiques, 2 Bd Lucien, 64000 Pau, France

October 2022

Abstract

We present a new architecture to learn a light neural network using an asynchronous layerwise bayesian optimization process deployed on low-power devices. The procedure is based on a sequence of five modules. In each module, an accept-reject algorithm allows to update real-valued - or binary - weights without any back propagation of gradients. The learning process is tested on two different environments and the electricity consumption is evaluated on several epochs, based on a homemade open source library using standard softwares and performance counters, and compared with a physical power meter. It shows that the decentralized version deployed on several low-power devices is more energy-efficient than the standard GP-GPU version on a dedicated server.

1 Introduction

1.1 Low-power Deep Learning

Deep learning is a subset of Machine Learning based on a very reduced set of computations that can be used to cover numerous supervised - as well as unsupervised - learning tasks. Starting from Image classification ([1]) and segmentation, deep learning architectures have spread many challenging problems in computer vision and natural language processing in general, such as automatic translation, as well as text generation. In the last two decades, the amount of energy - and time - needed to learn a given neural network has been reduced significantly thanks to GP-GPU. As a result, the size of the models have grown exponentially, leading to a generalization of deep learning to process a large amount of data in many applications. This is a cornerstone illustration of Jevons' paradox (see [2]), originally observed by Jevons in coal-burning factories : improved technology increases the efficiency to use a ressource but the falling cost of use increases it demands. With this in mind, this paper illustrates current researches in bayesian deep learning and parallel computing in order to deploy Convolutional Neural Networks (CNNs) in a tight constrained environment where each layer - or module - of the network is implemented in a particular micro-computer. It gives some insights into how to use layerwise deep learning on low-power devices.

Several approaches for reducing the computational effort of CNNs (training of smaller and faster models) have been proposed in the literature and show that interesting compromises are indeed possible. One can approximate real-valued convolutions with low bitwidth operations. In [3] (or more recently in [4]), CNNs are trained with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated. This drawback is mainly due to the use of the back-propagation algorithm, as well as the use of STE estimator (see [5]) in the procedure. In [6, 7], or more recently in [8], both filters and signal activations are binarized, leading to the so-called XNOR-nets where convolutions are approximated using primarily binary operations. Moreover, binarizing the input signals also provides a significant gain in the overall memory consumption, especially for large batch sizes. However since binarized networks can lead to poor approximations of real-valued convolutions, [9] proposes to use ternary weight networks whereas in [10], a more flexible low bitwidth approach uses different bitwidths (1-bit weights, 2-bit activations and more bits for gradients). [11] studies the effects of quantization for convolutional neural networks when the network complexity is changed. It shows a better resilience of deep nets when the number of layers is large. Finally, as low bitwidth convolutions can be implemented efficiently in standard CPU or GPU, but also on tight constrained field-programmable gate arrays (FPGAs) or even on Application Specific Integrated Circuits (ASICs), binarized networks are exploited on each specialized computer hardware. It leads in [12] to an energy-efficient and scalable CNN accelerator on ASICs, whereas [13] proposes a recent survey on hardware accelerators that uses FPGAs.

Other approaches to alleviate the computation of deep neural networks have appeared recently. Standard pruning methods (see [14] for the original paper, or more recently [15]) first train a feedforward or a convolutional neural network

to convergence, and then network connections and / or neurons are pruned only subsequently. These techniques are applied and fined-tuned after training the entire network, where network connextions are learnt thanks to another gradient-based method. Recent advances propose two different strategies in order to avoid the training of the entire network. It leads to significant improvements and accelerate both training and inference since the overall skeletonization is estimated before or during the training. One can first prune the network at initialization, by estimating the important weights for a given task (see [16, 17] for moderate pruning levels up to 95%, or more recently [18] for higher level of compression, up to 99.5%). Another promising strategy is to select the connectivity of the network during the training. Interestingly, theoretical guarantees are proposed in [19] for an adaptive procedure selecting the architecture during training. It can be seen as a form of architecture design, from the most general purpose of automated machine learning (AutoML, see [20]) to the problem of aggregation and design of efficient neural networks in terms of latency, memory size or carbon footprint, which leads finally to search for device-specific CNNs. As a seminal example, [21] proposes a neural architecture search (NAS) called FBNet to construct hardware efficient CNNs for mobile phones (see also [22]). It uses a stochastic generator of architectures (that is a recurrent neural network named the controller) and train the proposed network with reinforcement learning. More recently, [23] expands the search space to number of filters and channels dimension without prohibitive memory and computational cost with FBNetv2.

Recently open sources libraries have been proposed in order to evaluate globally the electric consumption and the carbon footprint of deep learning algorithms as well as specific studies of particular algorithms. It has been shown for instance in [24] that significant gains in carbon emission could be done without endowing the generalization performances. However, open sources available libraries in [25, 26, 27, 28, 29] are designed for massive usage of a large community of researchers and thus constrain the estimations with strong hypotheses. As a consequence, we loose the specific impact of different hardware platforms, as well as the cooling technique of the data center used for extensive computations. Moreover, carbon footprints are usually based on national coefficient. Finally, as discussed in [30], electric consumption based on a software estimation does not reflect the entire consumption of the system. Some contributions (see [31, 30]) propose to conduct energy measurement experiments but are limited to a particular hardware distributor and specific algorithms.

1.2 From layerwise to asynchronous learning of neural networks

In most cases, when training a neural network, each layer updates its weights after the previous layer is updated. This procedure is synchronous and mainly based on the back-propagation of the gradient (sse [32]). Moreover, using such an architecture coupled with a stochastic gradient descent optimizer (see [33]) limits the flexibility of the learning process and the ability to perform model paralellism. Recently, a first contribution (see [34]) has shown that this end-to-end optimization is not necessary and a sequential greedy approach does not impact the learning performance. Indeed, learning each layer independently of the others can lead to the same performances as classical learning a large model. More recently, by using a replay buffer, [35] shows that this approach can be extended to asynchronous settings, where modules can operate with possibly large communication delays. In this paper, asynchronous learning will allow us a decoupling of the work of each device, depending on its computational power and amount of available energy. Another challenge in the use of asynchronous learning is to minimize the number of exchanges between each layer. In the classical end-to-end framework, a single weight update requires $2(L-1)$ connections, where $L \geq 1$ is the number of layers in the network. Asynchronous learning will allow several weight updates without a large amount of energy for data sending and receiving between the different devices.

The implementation of asynchronous learning architectures requires some additional considerations. In particular, the way in which data is sent from one layer to another. In [35], a local memory is used in order to move data from one layer to another. If a binarization of data is also used, it allows a significant reduction on the memory ressoucrs used. This method of transferring data between layers is unsuitable when the layers are located on different machines where the memory is not shared. In this context, the exploitation of the network is essential. For that purpose, we use TCP protocol, the most basic and reliable socket data transfer for our task. Finally, the way data are acquired and sended is another question in its own right. How to process when one layer is faster than another? In [35], a replay Last In First Out (LIFO / Queue) buffer is used in order to keep the data for training beforehand. In this way, a layer that is faster than the upstream layer will be able to resume training on the past data. These two aspects allow a good circulation of data from one layer to another, without leaving any at rest.

1.3 Bayesian Deep Learning

The optimizer presented in Section 2 is motivated by a strong the PAC-Bayesian theoretical framework. PAC-Bayesian bounds can be traced back to the work of Mac Allester (see [36] and [37]). It was first introduced in learning theory to give a theoretical framework for proving the generalization abilities of algorithms that combine the advantages of both PAC (Probably Approximately Correct), that is generalization bounds, and Bayesian statistics (that is, the introduction of prior domain knowledge on the set of candidate models). The introduction of the Bayesian approach in neural networks

is originally due to [38]. Recently, a scalable optimizer has been proposed for Deep Learning in [39], where a practical training of deep networks is proposed using natural-gradient Variational Inference (VI) methods. However, this approach leads to a Adam-like optimizer, with very similar properties as its stochastic gradient descent counterpart.

In this paper, we adopt a different strategy by proposing a stochastic algorithm based on a markov chain procedure. The algorithm, described in Section 2.1.3, builds sequentially a Markov chain with an accept-reject procedure, without back-propagation of the gradient. The main objective is to extract a solution that optimizes a trade-off between accuracy over the training sample, and the complexity, or sparsity of the candidate learners. This trade-off is ensured by the bayesian procedure, thanks to a suitable prior design. This paradigm allows to derive a generic and flexible optimizer that can be used to learn binary weights, as well as continuous weights, or even hybrid weights, without any significant change in the procedure. Moreover, the optimizer only computes forward pass since no gradient is necessary to build the Markov chain. As a result, it can be used in a variety of framework, for example with non-differentiable loss functions. Finally, it has also theoretical guarantees of convergences and generalization proved in [40].

2 Network and cluster architectures

This section proposes to introduce the neural network architecture as well as the main hardwares that motivate the overall architecture of the cluster. We start with a description of the principle mathematical tools in order to describe the neural network architecture and the optimizer proposed for each module. Then, we detail the different devices and how we implement this neural network over these different hardwares and how data are moved from one module to another.

2.1 Neural Network architecture

In this paper, we learn a convolutional neural networks (CNNs) for a classification task thanks to a Bayesian approach. We then give some reminder about CNN and Bayesian learning and precise which architecture have been used for the experiments.

2.1.1 The neural network

A CNN is a sequence of convolutional and dense layers determined by a set of p weights $\mathbf{w} \in \mathbb{W}^p \subset \mathbb{R}^p$. For a $k \geq 1$ number of layers, the set of weights $\mathbf{w} := (w_l, b_l)_{l=1}^k$ where a couple (w_l, b_l) denotes the weights w_l and the bias b_l of layer $l = 1, \ldots, k$. The state space of each couple (w_l, b_l), $l = 1, \ldots, k$ depends on the encoding strategy on each layer (see Section 1.1 and the references therein). In the sequel, thanks to Algorithm 2.1.3, we can consider two cases: $\mathbb{W} = \{0, 1\}$ for binary layers, and $\mathbb{W} = \mathbb{R}$ for standard layers. The goal of the learning task is then to determine an optimal element of the parametric set $\{g_{\mathbf{w}} : \mathcal{X} \to \mathcal{Y},\ \mathbf{w} \in \mathbb{W}^p\}$ where $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space (for a classification task a label or a vector of probabilities). For a given $\mathbf{w} \in \mathbb{W}^p$, a CNN $g_{\mathbf{w}} : \mathcal{X} \to \mathcal{Y}$ is defined as:

$$\begin{cases} g_{\mathbf{w}}(x_0) &= softmax(w_k * \sigma(x_{k-1}) + b_k), \\ x_l &= w_l * \sigma(x_{l-1}) + b_l,\ l = 1, \ldots, k-1, \end{cases}$$

where for convolution layers, $*$ is the convolution operation[1], whereas it is a simple matrix multiplication for dense layers. The non-linear activation function σ is a Rectified Linear Unit (ReLU), and x_0 is the input data. In the cluster architecture defined below, the final architecture is a cascade of 5 modules of 2 layers made of convolutions and dense layers, where the number and the size of the filters are fixed. We also add an auxiliary dense layer in each module in order to perform layerwise learning (see Figure 2.3). Note that these hyper-parameters, namely the number of layers, the number and the size of the filters, could be selected by the optimizer adaptively, as proposed in [40].

2.1.2 Bayesian learning

Given a training dataset $\{(x_i, y_i),\ i \in \{1, \ldots, n\}\}$ and a loss function $\ell(.,.)$, learning a CNN among the family $\{g_{\mathbf{w}} : \mathcal{X} \to \mathcal{Y},\ \mathbf{w} \in \mathbb{W}^p\}$ involves solving an optimization problem. The Empirical Risk Minimization (ERM) principle aims at minimizing the empirical risk as follows:

$$\min_{w \in \mathbb{W}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, g_w(x_i)). \tag{1}$$

For an initialization $w_0 \in \mathbb{W}^p$, the gradient descent (GD) algorithm is based on the following first-order update:

$$w_{t+1} = w_t - \eta \nabla \left(\frac{1}{n} \sum_{i=1}^{n} \ell(y_i, g_w(x_i)) \right) [w_t], \tag{2}$$

[1] In the context of Computer Vision, $x \in \mathbb{R}^m$ represents an image of size $m = r_1 \times r_2$ and given a set of filters of size $s \times s$, the convolution $w * x$ corresponds to the matrix product of the filters and a relaxed form of the Toeplitz matrix of input signal x.

where $t \geq 0$ and $\eta > 0$ is the learning rate.

However, as mentioned in Section 1, we want to use a prior knowledge about the problem to enforce sparse solution. For that purpose, we need to use a pre-conditioning optimization problem. The bayesian paradigm solves the following optimization problem:

$$\min_{\rho \in \mathcal{P}(\mathbb{W}^p)} \left\{ \mathbb{E}_{w \sim \rho} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, g_w(x_i)) + \alpha \cdot \mathcal{K}(\rho, \pi) \right\}, \tag{3}$$

where $\mathcal{P}(\mathbb{W})$ is a particular set of probability distribution, π is a prior distribution and $\mathcal{K}(.,.)$ is the Kullback-Leibler divergence. In comparison to (1), the optimization problem lies on a set of probability distribution. Moreover, we penalize the averaged empirical risk by the KL divergence to the prior distribution π. Then, by a suitable choice of the prior π, it gives a sparse solution. Finally, by the duality formula of the Kullback-Leibler divergence, solution of (3) can be defined explicitly as a Gibbs measure as follows:

$$\hat{\rho}(dw) := C_{\alpha,n} \exp\left(-\frac{1}{\alpha n} \sum_{i=1}^{n} \ell(y_i, g_w(x_i)) \right) \pi(dw), \tag{4}$$

where $C_{\alpha,n}$ is the normalizing constant. It means that the solution concentrates on CNN g_w that are close to the minimum of the empirical risk. Unfortunately, the computation of (4) is untractable as $w \in \mathbb{W}^p$ with a potentially huge $p \geq 1$. This is a classical challenge in high-dimensional Bayesian statistics.

2.1.3 The optimizer

Thanks to the Bayesian optimization procedure defined below, we want to approximate the Gibbs posterior (4). Due to the high-dimensionality of $w \in \mathbb{W}^p$, variational inference (VI) method can be proposed (see [39]), leading to an optimizer called the natural gradient, very similar to the sequential procedure (2). In this article, we prefer to approximate the distribution (4) thanks to a MCMC method using Metropolis-Hastings algorithm. The optimizer used here is a simple accept-reject algorithm described in Algorithm 1.

Algorithm 1 MCMC Optimizer

Require: $\lambda > 0, \mathrm{L} \geq 1, N \in \mathbb{N}^*$, π, and a proposal distribution $w \mapsto \tilde{p}_w(\cdot)$

Generate $\hat{w}^{(1)}$ from π

for $k = 1, ..., N$: **do**

Random choice of a layer $\tilde{l} \in \{1, ..., L\}$,

Generate $\tilde{w} \sim \tilde{p}_{\hat{w}^{(k)}}$,

Compute acceptance ratio : $\rho(\tilde{w}, \hat{w}^{(k)}) = \dfrac{exp(-\lambda \sum_{i=1}^{n} \ell(y_i, g_{\tilde{w}}(x_i)))\pi(\tilde{w})\tilde{p}_{\hat{w}^{(k)}}(\hat{w}^{(k)})}{exp(-\lambda \sum_{i=1}^{n} \ell(y_i, g_{\hat{w}^{(k)}}(x_i)))\pi(\hat{w}^{(k)})\tilde{p}_{\tilde{w}}(\tilde{w})}$

Update

$$(\hat{w}^{(k+1)}, \hat{l}^{(k+1)}) = \begin{cases} (\tilde{w}, \tilde{l}) & \text{with proba } \rho(\tilde{w}, \hat{w}^{(k)}), \\ (\hat{w}^{(k)}, \hat{l}^{(k)}) & \text{otherwise.} \end{cases}$$

end for

The question of the convergence of the Markov chain generated by Algorithm 1 is discussed in [40]. We then use this algorithm on each device in order to select the set of weights associated to the given module. For the convolutional modules (see Figure 3 for the global architecture), Algorithm 1 select at random at each iteration the convolution or the auxiliary block and proposes to change the set of weights as described in Figure 1.

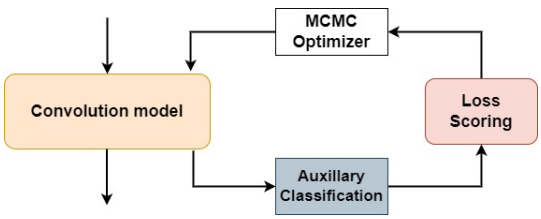

Figure 1: A model on a single convolutional module

In the next section, we plug these optimizers to solve a classification task on CIFAR10 dataset.

2.2 Hardware architecture

In this subsection we present the global architecture and gives some details about the hardware, and the model parallelism. We propose to use light-consumption devices to make the training and the inference on a cluster composed of Jetson Nano and Raspberry Pi cards. We also compared this configuration with our server equipped with an RTX 3090 card. Some characteristics are proposed in Table 1 below.

	Weight (g)	Dim. (mm)	Memory	Cuda cores	Transistors	Operation/sec	Power
RTX 3090	1565	318 x138	24GB	10496	28,3b	285 TFLOPS	350 W
Jetson Nano	138	45x70	2GB	128	2b	475 GFLOPS	10W
Raspberry Pi	40	56x85	0.5GB	-	4b	$\sim$ 700 MFLOPS	3W

Table 1: Hardware characteristics

Our cluster of calculus is composed of five devices connected in series as follows:

- a first Jetson Nano card of 4GB,
- three RaspBerry Pi cards of 8, 4 and 2 GB,
- the last one is a Jetson Nano of 2GB.

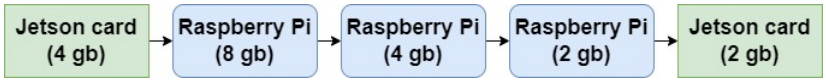

Figure 2: Global hardware interface

The choice of such an architecture is arbitrary but the idea is to let the maximal computational capabilities into the first devices. This allows us to minimize the delay to modify the overall architecture with fewer or more devices. Furthermore, the choice to run the cluster on ARM CPU has also been motivated by [41]. It allows us to easily move to binary calculation. The aspects of implementation and libraries available are detailed in the following section.

The Raspberry Pi cards are ARM - RISC processors (Reduced Instruction Set Computer) where quantization or binarization could be implemented more easily with Larq Compute Engine (see [41]). These methods are more efficient on that kind of CPU than on CPU x86 - CISC (Complex Instruction Set Computing).

On the Jetson Nano cards, we have the same CPU architecture but we recommend to test sparsity with pruning methods introduced in Section 1 thanks to the presence of a GPU.

For the first four devices, we implement a really simple model composed of two layers:

- a Convolution 2D layer, such that the outputs are connected to the convolution layer of the next device (followed by a flatten layer applied to the convolution outputs),
- a Fully Connected or Dense layer gives an intermediate output to give to the loss function.

The last device contains a model of two Fully Connected layers to obtain the final output of the model.

2.3 Deployment and limits

We compare the energy consumption of the network in Figure 3 with 2 different configurations:

- 5 independent instances on a single machine with an RTX 3090 GPU,
- the architecture of 5 microprocessors in series described above.

The first configuration on the RTX 3090 will hereafter be referred to the "server", while the second one on the low-devices will be referred to as the "cluster". It's important to note that the second configuration is a cluster with a sequential architecture, and the communication between them is on one side only.

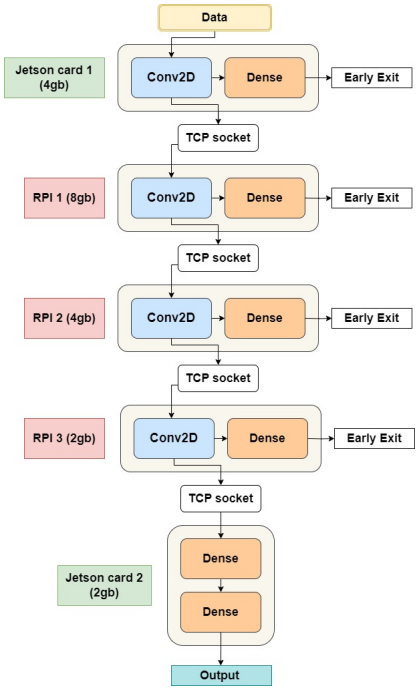

Figure 3: Distributed model

Morever, it's important to notice that we haven't been able to implement sparsity computation on Jetson Nano devices with GPU because CUDA version available on the hardware (Jetson Nano) was not matching the cuSPARSE library. Some details about the limitations and the real effect of pruning in neural networks have been studied in a previous work[2]. We also did not deploy binary training on Raspberry Pi devices either because the library used for format conversion wasn't suitable for our architecture [3].

This architecture has been implemented in Python, and data transfers are enabled by the *asyncio* package and its asynchronous TCP socket implementation. The one side communication allows us to implement 2 servers and 3 clients to communicate our data rather than create on each device 1 client and 1 server.

We've implemented a buffer different from [35], which ensures that data is received and sent correctly, but does not implement a replay system. Indeed, the decreasing architecture used ensures that no downstream machine is faster than an upstream one. Regarding the input size of each layer in Section 2.2, no device should run faster than a previous one. This asynchronous implementation of the MCMC architecture can be found on GitHub[4].

3 Comparison of the energy consumption

3.1 Experimental design

We run our architecture on a classification task thanks to the CIFAR10 dataset made of 60 000 images of size 32x32 with RGB channels and 10 different labels. the protocol is designed as follows:

- the dataset is loaded on the first Jetson card 1 (4gb) and mini-batches of size 512 are sent during 5 epochs,
- for each mini-batch, $N = 10$ iterations of Algorithm 1 are run by the first layer,
- at the end of each minibatch, the output of the convolutional layer is sent to the next device,
- the next device starts $N \leq 10$ iterations of Algorithm 1 with the new given input, and stops when $N = 10$ or if a new output is sent by the previous device,
- the data and I/O are disseminated in this way along the network until the last layer has received all the data and has made all its accept-reject iterations.

[2]see: `https://medium.com/@yanis.chaigneau/pruning-in-neural-networks-541af4f9a899`
[3]see: `https://medium.com/@fkinesow/binary-neural-network-part-2-cecbe5761b78`
[4]`https://github.com/GreenAI-Uppa/deep_learning_mcmc/tree/asynchrone_version`

Moreover, we realized the same protocol for 1 epoch, 5 times, in order to compute the average consumption of 1 epoch and have its standard deviation.

Since we stop the generator after 5 epochs, we only observe a decrease in the loss on each device, without reaching a convergence to an interesting solution.

3.2 Results

We measure electricity consumption during the entire protocol described above thanks to our open source library AIPowerMeter[5] and compare the results with a power meter[6]. In Figure 4 below, we compare the energy consumption of the low-devices cluster described in Section 2.2 with the same architecture deployed in a standard PC equipped with a RTX 3090. The vertical blue line on the right part of the graph shows the true size of the left graph. In this case, each module is a process of the same machine. We observed two significant differences between those two ways of running MCMC architecture. Firstly, the experiment time is longer for the cluster experience. We ran our five epochs in 10 minutes on the server while it needs one hour on the cluster. However, interestingly, the server consumed almost two times more energy than the cluster. The total server's consumption is 141,073 Joules while the cluster's total consumption is 79,260 Joules. Moreover, the average consumption of one epoch reaches 16,000 Joules with a standard deviation of 350 in the cluster, whereas it is 37,000 Joules with a standard deviation of 2,000 in the server (5 repetitions).

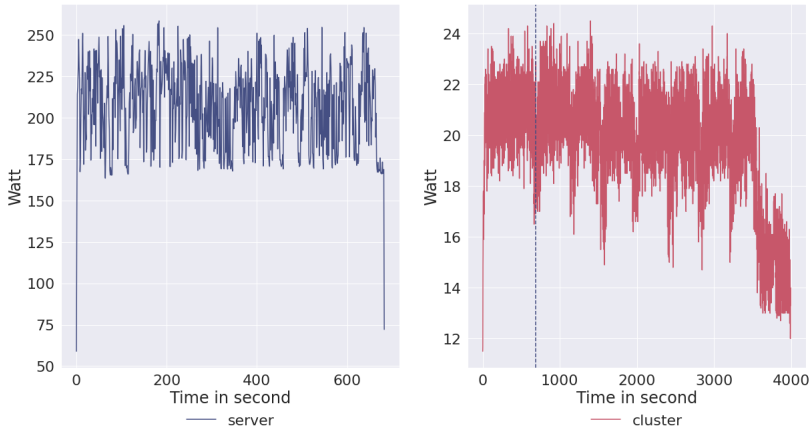

Figure 4: Cluster and server's energy consumption in W per second

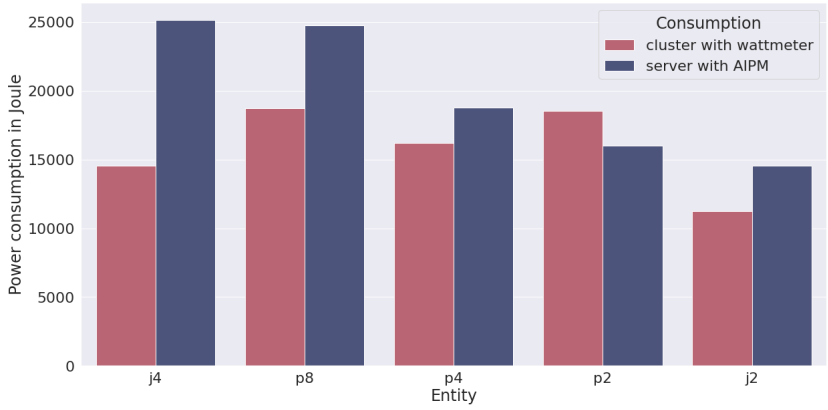

Figure 5: Cluster and server energy consumption per entity

Finally, we compare the consumption of each module for both cluster and server. First of all, as we can see in Figure 5 in red, each Raspberry Pi of the cluster consumes more energy than each Jetson Nano while they have lighter hardware configurations. This difference can be explained by the use of GPU on Jetson and not on Raspberry Pi, which is more suitable for convolution calculation. Furthermore, these three Raspberry Pi seem to have the same consumption on that

[5]Github link here `https://greenai-uppa.github.io/AIPowerMeter/`

[6]Details about the power meter is available on the documentation here `https://greenai-uppa.github.io/AIPowerMeter/experiments/schneiderbox.html`

experiment for a similar input size ($512 \times 8 \times 8 \times 32$) while they all have different RAM sizes. Finally, the Jetson Nano 4GB RAM (j4) consumes more than j2, since it deals with more data and has a bigger configuration.

We've also been able to measure the consumption of each entity on the server thanks to the library Aipowermeter(AIPM[7]) in blue in Figure 5. It allows us to measure the GPU and CPU consumption for each experiment on the device thanks to RAPL (see [42]) and Nvidia-Smi statistics. The distribution is less uniform than expected by our team, it decreases in the processes launched order.

The gap between the server and cluster in the experiment is less important than explained before because of using AIPM that produces a lower bound on the entire consumption given by the wattmeter measurements.

4 Conclusion and perspectives

In this article, we present a new architecture in order to train a neural network in an asynchronous way, layer by layer. The method uses a MCMC procedure at each layer, where an accept-reject algorithm is implemented in order to select the weights of the network. This new optimization design is deployed on embedded low-power devices. This implementation allows us to custom each layer of the network with a particular device, without any back propagation of gradients.

We compare the power consumption of the procedure deployed on these embedded devices to the same structure on a server. As excepted, the cluster consumes less energy but is slower than the server.

In a future work, several improvments could be done in order to fully enjoy this parallel and hardware-dependant layerise learning. For example, depending on the hardware, we can propose continuous and sparse weights for the first and the last module by using Jetson Nano, and binary weights for the other layers, as proposed in [8]. However, for the moment, a lack of PyTorch and Cuda libraries was problematic in order to implement sparsity and binarization. Finally, our network architecture does not use pooling layers in order to adapt the size of data to transfer on each device.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[2] D. Bauer, K. Papp, J. Polimeni, K. Mayumi, M. Giampietro, and B. Alcott, "John polimeni, kozo mayumi, mario giampietro & blake alcott, the jevons paradox and the myth of resource efficiency improvements," *Sustainability: Science, Practice and Policy*, vol. 5, no. 1, pp. 48–54, 2009.

[3] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, pp. 3123–3131, 2015.

[4] W. Tang, G. Hua, and L. Wang, "How to train a compact binary neural network with high accuracy?," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 1, 2017.

[5] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," 2013.

[6] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[7] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, pp. 525–542, Springer, 2016.

[8] A. Bulat and G. Tzimiropoulos, "Xnor-net++: Improved binary neural networks," *arXiv preprint arXiv:1909.13863*, 2019.

[9] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[10] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[11] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.

[7] `https://greenai-uppa.github.io/AIPowerMeter/`

[12] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2017.

[13] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2018.

[14] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," in *Proceedings of the 1st International Conference on Neural Information Processing Systems*, pp. 107–115, 1988.

[15] Z. Yang, M. Moczulski, M. Denil, N. De Freitas, A. Smola, L. Song, and Z. Wang, "Deep fried convnets," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1476–1483, 2015.

[16] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[17] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," *arXiv preprint arXiv:2002.07376*, 2020.

[18] P. de Jorge, A. Sanyal, H. S. Behl, P. H. Torr, G. Rogez, and P. K. Dokania, "Progressive skeletonization: Trimming more fat from a network at initialization," *arXiv preprint arXiv:2006.09081*, 2020.

[19] G. Bellec, D. Kappel, W. Maass, and R. Legenstein, "Deep rewiring: Training very sparse deep networks," in *International Conference on Learning Representations*, 2018.

[20] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *arXiv preprint arXiv:1908.00709*, 2019.

[21] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

[22] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

[23] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen, *et al.*, "Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions," *arXiv preprint arXiv:2004.05565*, 2020.

[24] T. Parcollet and M. Ravanelli, "The Energy and Carbon Footprint of Training End-to-End Speech Recognizers." working paper or preprint, Apr. 2021.

[25] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.

[26] P. Henderson, J.-R. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *ArXiv*, vol. abs/2002.05651, 2020.

[27] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Quantifying the carbon emissions of computation," *Advanced science*, 05 2021.

[28] L. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," p. arXiv preprint https://arxiv.org/abs/2007.03051, 07 2020.

[29] D. A. Patterson, J. Gonzalez, Q. V. Le, C. Liang, L. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *CoRR*, vol. abs/2104.10350, 2021.

[30] Q. Cao, A. Balasubramanian, and N. Balasubramanian, "Towards accurate and reliable energy measurement of NLP models," in *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, (Online), pp. 141–148, Association for Computational Linguistics, Nov. 2020.

[31] C. F. Rodrigues, G. Riley, and M. Luján, "Synergy: An energy measurement and prediction framework for convolutional neural networks on jetson tx1," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 375–382, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2018.

[32] Y. Lecun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems (NIPS 1989), Denver, CO* (D. Touretzky, ed.), vol. 2, Morgan Kaufmann, 1990.

[33] L. Bottou, "Online algorithms and stochastic approximations," in *Online Learning and Neural Networks* (D. Saad, ed.), Cambridge, UK: Cambridge University Press, 1998. revised, oct 2012.

[34] E. Belilovsky, M. Eickenberg, and E. Oyallon, "Greedy layerwise learning can scale to imagenet," 2019.

[35] E. Belilovsky, M. Eickenberg, and E. Oyallon, "Decoupled greedy learning of CNNs," in *Proceedings of the 37th International Conference on Machine Learning* (H. D. III and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 736–745, PMLR, 13–18 Jul 2020.

[36] D. A. McAllester, "Some pac-bayesian theorems," *Machine Learning*, vol. 37, no. 3, pp. 355–363, 1999.

[37] D. A. McAllester, "Pac-bayesian stochastic model selection," *Machine Learning*, vol. 51, no. 1, pp. 5–21, 2003.

[38] R. M. Neal, *Bayesian Learning for Neural Networks, Vol. 118 of Lecture Notes in Statistics*. Springer-Verlag, 1996.

[39] K. Osawa, S. Swaroop, M. E. E. Khan, A. Jain, R. Eschenhagen, R. E. Turner, and R. Yokota, "Practical deep learning with bayesian principles," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[40] A. Chee, P. Gay, and S. Loustau, "Sparsity regret bounds for xnor-nets++," 2021.

[41] T. Bannink, A. Hillier, L. Geiger, T. de Bruin, L. Overweel, J. Neeven, and K. Helwegen, "Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 680–695, 2021.

[42] K. Khan, M. Hirki, T. Niemi, J. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, 01 2018.