EPJ Web of Conferences **214**, 06025 (2019) https://doi.org/10.1051/epjconf/201921406025
CHEP 2018

Large-Scale Distributed Training Applied to Generative Adversarial Networks for Calorimeter Simulation

Jean-Roch Vlimant[1], *Felice* Pantaleo[2], *Maurizio* Pierini[2], *Vladimir* Loncar[2], *Sofia* Vallecorsa[2,3], *Dustin* Anderson[1], *Thong* Nguyen[1], and *Alexander* Zlokapa[1]

[1]California Institute of Technology, Pasadena, California, U.S.
[2]CERN, Meyrin, Switzerland
[3]Gangneung-Wonju National University, Korea

Abstract. In recent years, several studies have demonstrated the benefit of using deep learning to solve typical tasks related to high energy physics data taking and analysis. In particular, generative adversarial networks are a good candidate to supplement the simulation of the detector response in a collider environment. Training of neural network models has been made tractable with the improvement of optimization methods and the advent of GP-GPU well adapted to tackle the highly-parallelizable task of training neural nets. Despite these advancements, training of large models over large data sets can take days to weeks. Even more so, finding the best model architecture and settings can take many expensive trials. To get the best out of this new technology, it is important to scale up the available network-training resources and, consequently, to provide tools for optimal large-scale distributed training. In this context, our development of a new training workflow, which scales on multi-node/multi-GPU architectures with an eye to deployment on high performance computing machines is described. We describe the integration of hyper parameter optimization with a distributed training framework using Message Passing Interface, for models defined in keras [12] or pytorch [13]. We present results on the speedup of training generative adversarial networks trained on a data set composed of the energy deposition from electron, photons, charged and neutral hadrons in a fine grained digital calorimeter.

1 Introduction

Deep neural networks (DNN) are machine learning models with many parameters that are effectively trained using stochastic gradient descent methods. The power of DNN at executing challenging tasks learned from data is very attractive to the most diverse fields of science, business and society at large, and notably in High Energy Physics (HEP). In recent years, there have been a significant number of articles reporting promising results with applying deep learning to HEP challenges. DNN can be used in supervised learning, an approach with which one wants to learn, from a training data set, a mapping from a set of input features to a set of target features, in order to predict future target values on previously not seen data.

Within the context of unsupervised learning and generative models, multiple neural networks can be trained concurrently in the generative adversarial (GAN) scheme [1]. In this scheme a neural network is generating featured data starting from random numbers, and a

second neural network is set to distinguish between generated samples and the data from an initial data set. Upon convergence the generative model is able to generate new data that looks statistically identical to the presented training data. Training GAN with large input data sets and a large input space turns out to be very intensive, taking several hours per epoch. Such generative model have had tremendous publicity recently in the field of data science thanks to their great success in generating complex data (images mostly) and application of such models in the field of high energy physics are showing great promises [3].

Deep models require quite large data set so as to be trained (because of the large number of parameters to be adjusted) : even with the large boost given by general purpose graphical units (GP-GPU), training remains quite a computing intensive task that may last from days to weeks to converge, if not worse. Therefore we explore several parallelism approaches to the stochastic gradient descent method in order to speed up further the training of neural networks.

In training GAN, as in training other neural network models, the choice of some parameters of the models that cannot be learned with stochastic gradient (such as batch size, learning rate, number of hidden layers, ...) are left for optimization. Trial and errors optimization on such hyper parameters is time consuming and requires a rather high level of educated expertise. In this work, we explore the use of Bayesian optimization with a Gaussian process assumption for the prior, as well as an evolutionary algorithm on the hyper parameters.

This document is structured as follows, we provide the relevant details of artificial neural network and generative adversarial networks in section 2 as well as details on stochastic gradient descent in section 3. The problem of hyper parameter optimization is described in section 5. The directions of distributed training are explored in section 4 and results obtained at various facilities are provided in section 6. We conclude with some outlook on distributed training and optimization in section 7.

2 Neural Networks and Generative Adversarial Network

Among the many mathematical models in the field of machine learning, Artificial neural network (ANN) are a type of model initially inspired from the biological functioning of the brain. Such a model is composed of an input layer with as many nodes (neurons) as desired input features, an output layer with a number of neurons in adequacy with the problem that one wishes to solve and one or multiple internal layers composed of internal nodes arranged in a variety of topology [2]. The only restriction to the topology and computation will be clear from section 5 with the necessity of having a differentiable computation graph.

ANN can be used for supervised learning, a task for which one possesses a target feature (boolean for classification, continuous for regression) which one wishes to learn, from a corpus of training data, and then predict on unseen similar data. On the other hand, with unsupervised learning, one may wish to train a model that learns the structure of a data set so as to be able to generate new samples, that statistically resemble the original data. Such models are particularly attractive because, even though they might be hard to train, as a one time process, generating new data can be significantly faster, by several orders of magnitude, than existing simulation software. In the context of the resource constraint computing model for the High Luminosity Large Hadron Collider, these models, even with reduced fidelity, could enable the production of the large data set of simulated collisions required during analysis.

One class of generative model is the so called Generative Adversarial Network [1] composed of two ANN. The first one is the generative model (generator) which produces new data from numerical vectors drawn at random from pre-determined distributions and aiming at mimicing sample drawn from the original data set. It is however hard to train such model

with traditional methods because of the absence of a tractable metric to estimate the goodness of identity of the generated data and the original data. The second ANN, the so called discriminator, is set to classify properly the generated data and the original data. It therefore takes input from the output space of the generator and produces a label on the origin of the input (fake or original). The two ANN are trained in an alternating fashion, the discriminator is exposed to the original data and generated samples while the generator is trained with the loss of the discriminator with generated samples labelled as original. During this procedure the generator is trained to fool the discriminator. Further in depth details on GAN can be found in [1].

3 Model Training with Stochastic Gradient Descent

ANN parameters are learned from data through an optimization procedure aiming at maximizing likelihood or minimizing errors of the model, cast as a minimization of a loss function. A standard method in convex optimization is gradient descent, during which the parameter space if navigated by taking infinitesimal steps on the opposite direction of the gradient of the loss with respect to the parameters of the model. By having ANN models and loss function fully differentiable, one can compute the gradient analytically and is left with only the evaluation of these gradients. Mini-batch Stochastic gradient descent (SGD) is an algorithm that computes the direction of stepping not from the gradient of a single sample but from a collection of samples ("batch"). The optimization of the loss function of an ANN is not a convex optimization problem, however stochastic gradient descent have shown great success in finding good parameters for ANN [4]. An epoch indicates a cycle of the SGD where all batches of the training data set have been taken into account. Several epochs are usually needed to achieve convergence of the model.

Further manipulations can be performed on the batch gradients in the optimizer algorithm in order to reach faster convergence: these algorithms are driven by various parameters, such as the learning rate (measuring the step size in the parameter space). Further information on optimizers are available in [5].

4 Distributed Training

In this section we present several ways for parallel computation of the gradients needed for SGD. One can leverage these levels of parallelism on high performance computing (HPC) centers composed of many nodes with high bandwidth connectivity and obtain a shorter time to solution. The computation and communication is orchestrated using the MPI [6] framework, abstracting the communication protocols from the computation. An MPI program is executed over multiple processes, running on multiple physical hosts on the cluster. We call each process a worker, and it does not matter a priori if they get executed on the same physical node. Depending on the topology of the HPC, there can be more than one GP-GPU per physical host, and we enforce that we do not get more than one process associated with one GP-GPU. De-facto, in the following, each worker is referring to a process with at least one dedicated GP-GPU attached. Results of applying the following techniques are reported in section 6.

4.1 Batch Parallelism

In the scheme of the SGD, with a small batch size, the effective gradient is very noisy due to statistical fluctuations over the small number of gradients averaged. It is therefore necessary to have a sufficiently large batch size. Even with ever growing memory in GP-GPU, the

amount of input data and network information might become larger than the available memory, preventing from doing efficient computation on GP-GPU. When faced with this situation, the batch can be divided in sub-batches and distributed to multiple workers. The gradients then are efficiently gathered for the averaging over the batch. We use the Horovod [7] library, developed by Uber, efficiently implementing this process, with modifications that we brought to the library interface. With these modifications, sub-groups of ranks can be individually initialized and work in concert for batch parallelism. This therefore allowed each sub-group to work on the computation of the gradient of one batch. HPC topology with many GP-GPU per node are well suited for this method since the communication can be implemented very efficiently with GPU-2-GPU fast communication such as Nvlink [8]. In summary, batch parallelism enables running SGD with significant batch size when the GP-GPU memory is a limiting factor.

4.2 Data Parallelism

The SGD algorithm is sequential in the successive batches used to compute updates to the model parameters. The computation of the gradient for multiple batches can however be distributed from a master process to multiple processes so as to calculate them in parallel. One of the immediate points, to be noted about doing so, is that all workers are calculating the gradient evaluated for the set of model parameters, over different batches of data. After successive update of the parameter of the master model with the gradients computed in this manner, one updates a set of hyper parameters using gradients calculated on outdated hyper parameter values. This generates, if not mitigated algorithmically, the staleness of the gradients and loss of convergence at fixed number of epochs. Observation of such effect is available in [6] and can be mitigated with dedicated algorithm like the one in [10]. There is a trade-off between the speedup obtained with computing the gradient from multiple batches and the degradation of the convergence rate of the model. In summary, data parallelism is having the gradient of many batches done in parallel and integrated to a master model.

4.3 Model Parallelism

Deep neural networks can turn up very large, with billions of parameters. It might be grown at a point at which the amount of memory required is too large (assuming a fixed batch size that itself fits on the device, see section 4.1 otherwise). By virtue of the chain rule in calculating the gradient of the loss, the calculation can be factorized by layers of the ANN. It is therefore effectively possible to distribute part of the computation graphs corresponding to successive layers of the model to several devices, with the need to only communicate the activation of the layer at the boundaries. We use the native functionality of tensorflow [11] to put part of the computation graph on different device, in the case of multiple GP-GPU per node. In summary, model parallelism allows to spread the forward and backward passes of SGD over mutliple devices.

5 Model Parameter Optimization

There are however parameters of the models that cannot be adjusted using SGD and that have significant impact on model performance. Such parameters are for example learning rate of the optimizer or the number of nodes in internal layers of the ANN. These parameters are often called hyper-parameters to differentiate them from the other parameters. The hyper parameters are often scanned by a developer while looking for a set with good performance

after training, and such scanning is a lengthy, painful and sometimes random process. Such search can be replaced with a full grid search but is computationally prohibitive in large hyper-parameter space dimension. Gradient descent is often not a possible algorithm because of the discreetness of hyper parameters. Below, we give details of two methods commonly used for hyper parameter optimization. It should be noted that it is not necessary to use the loss function previously defined as the performance metric of the model. Any other metric may be use to quantify the performance of the model, and be used as figure of merit (FOM) during hyper parameter optimization. Specifically, this figure of merit does not need to be differentiable. We eventually provide details of the cross validation procedure, as a must for model comparison.

5.1 Bayesian Optimization using Gaussian Process Prior

A method commonly used for hyper parameter optimization is using Bayesian optimization with modeling the FOM with Gaussian processes. Details of the algorithms can be found in [14]. We use the scikit-optimize implementation [15] of the optimizer, which one queries for values of the hyper parameters to evaluate the performance for. With successive sampling of the hyper parameter space, the optimizer gets better at providing suggestions close to the optimal hyper parameters. The complexity of this algorithms grows as the cube of the number of sampling points and can become prohibitive for large hyper parameter space in which the convergence is taking multiple sampling iterations. While the process of trial and error is essentially sequential, multiple set of hyper parameters can be evaluated simultaneously.

5.2 Evolutionary Algorithms

Another class of algorithms used for hyper parameters optimization is based on genetic evolution[16]. We implement a simple version where the hyper parameters are trivially encoded as the chromosone vector and the FOM is trivially mapped to the fitness of a chromosome The algorithm starts with an initial population of chromosomes taken at random within the allowed space, a fraction of chromosome with best fitness are kept to carry on producing the next generation. The population of the next generation is created from random linear interpolation from fittest chromosomes and additional mutation are obtained by moving the chromosome at random within an infinitesimal volume. The fitness of the next generation is evaluated and the process is repeated for a certain number of iterations. This algorithm is expensive in the number of evaluation calls (training of a model to convergence) but can have an advantage over the previous method when the hyper optimization requires many iterations. The algorithm is sequential in the successive generation, but fully parallel in evaluating the fitness of a given generation.

5.3 Cross Validation

The training of ANN is subject to some level of fluctuation due to the initial random weights and numerical rounding happening during SGD. There is another stochastic component in batch parallelism (see section 4.2) where gradients are considered in an order driven by computation time on each node. The performance of a model is usually evaluated on the validation set, distinct from the training one: its finite size and choice introduces a bias in the measured performance. One can estimate both performance uncertainties using the K-split cross validation algorithm in which the initial data set is plit into multiple parts that are use to concurrently evaluate the performance of a choice of model hyper parameters. The initial datset is divided in K parts (a.k.a. splits): K-1 splits are used for training, the remaining one

is left out for validation. One can quote the average performance over the K FOM values obtained with the K-splits cross validation. The training and evaluation of the different models are trivially parallel, and one can leverage large number of nodes at an HPC to run cross validation in the same time it would take to train one model, with the advantage of having a better estimate of the performance.

6 Results

We report here on scaling performance obtained in training a GAN on 3D calorimetry simulated data on several supercomputers and using libraries implementing different schemes of distributed training. Further details on the model and data set are available in [17].

We report scaling performance using the mpi-learn [19] package adapted to train GAN models, and with the extension for performing hyper-parameter optimization in mpi-opt [22]. The original paper reported quasi-linear scaling up to 8-15 workers [19]. The GAN training was performed on PiZ Daint [20] and Titan [21] supercomputers equipped with NVIDIA®V100 GP-GPU and K20 respectively. As can be seen in figure 1 the preliminary results on scaling for GAN is not great with a factor of about 1:2 speedup per worker up to 20 workers, and 20x speedup with 100 workers. We have observed no degradation of the fidelity of the trained model up to using 15 workers. In the current setup this is not the most efficient use of the resource as the speed up is not linear, but still provide a significant improvement over using a single node. We hypothesize that the deviation from a linear speed-up is due to the fact that the workload for the workers is too small and that most time is spend with the master handling the weights update and communication to the workers. Better understanding of the scaling would require further in depth analysis, which the authors are planning as future work.

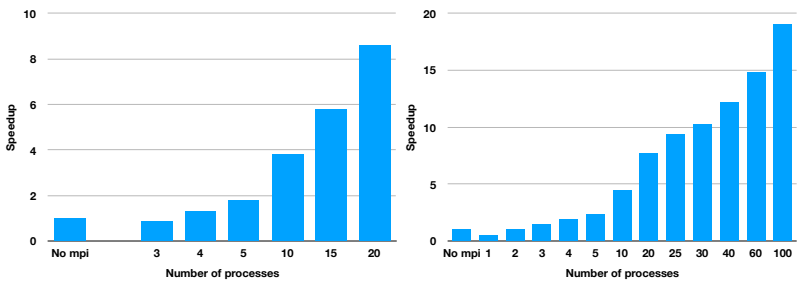

Figure 1. Speed up of training a 3D calorimeter energy deposition generative adversarial network, as a function of the number of training processes and with respect to training with one process only. Results obtained when running on CSCS Piz Daint (left) and ORNL Titan (right).

7 Discussion

In this article, we review the technicalities of training neural networks on distributed systems. We present several ways to parallelise training of neural networks, including Generative Adversarial Networks. We describe methods to perform hyper parameter optimization, implemented in a python library for deployment on HPC resource. We report results on scaling of

distributed training on two supercomputers, obtaining promising speedup without noticeable degradation in model fidelity.

The authors continue to work on detailed validation and optimization of these preliminary results, subject to resource availability (large scale training benchmarking are expensive on resource allocation), toward releasing a turn-key software for distributed training and optimization of neural networks using keras, tensorflow and pytorch.

Acknowledgement

Part of this work was conducted on Piz Daint at CSCS under the allocations d59 (2016) and cn01 (2018). Part of this work was conducted on Titan at OLCF under the allocation csc291 (2018). Part of this work was conducted at "*iBanks*", the AI GPU cluster at Caltech. We acknowledge NVIDIA, SuperMicro and the Kavli Foundation for their support of "*iBanks*". Part of the team is funded by ERC H2020 grant number 772369.

References

[1] Generative Adversarial Networks, Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, https://arxiv.org/abs/1406.2661

[2] The Neural Network Zoo, Fjodor van Veen, http://www.asimovinstitute.org/neural-network-zoo/

[3] Machine learning at the energy and intensity frontiers of particle physics, Alexander Radovic, Mike Williams, David Rousseau, Michael Kagan, Daniele Bonacorsi, Alexander Himmel, Adam Aurisano, Kazuhiro Terao, Taritree Wongjirad, https://www.nature.com/articles/s41586-018-0361-2

[4] A stochastic approximation method, Robbins, H. and S. Monro (1951), The annals of mathematical statistics, 400–407.

[5] An overview of gradient descent optimization algorithms, Sebastian Ruder, https://arxiv.org/abs/1609.04747

[6] MPI Forum. MPI: a message passing interface standard. (1994). Technical Report (1994).

[7] Horovod: fast and easy distributed deep learning in TensorFlow, Alexander Sergeev and Mike Del Balso, https://arxiv.org/abs/1802.05799

[8] https://www.nvidia.com/en-us/data-center/nvlink/

[9] NVIDIA Collective Communications Library (NCCL), https://developer.nvidia.com/nccl

[10] Gradient Energy Matching for Distributed Asynchronous Gradient Descent, Joeri Hermans, Gilles Louppe, https://arxiv.org/abs/1805.08469

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

[12] Keras, François Chollet and others, https://keras.io

[13] Automatic differentiation in PyTorch, Adam Paszke et al., https://pytorch.org

[14] Practical Bayesian Optimization of Machine Learning Algorithms Jasper Snoek, Hugo Larochelle and Ryan P. Adams Advances in Neural Information Processing Systems, 2012

[15] scikit-optimize python library, https://scikit-optimize.github.io/

[16] Genetic Algorithms, Numerical Optimization, and Constraints, Zbigniew Michalewicz, In: Morgan Kaufmann, 1995, pp. 151–158.

[17] Calorimetry with Deep Learning: Particle Classification, Energy Regression, and Simulation for High-Energy Physics, Benjamin Hooberman et al.,https://dl4physicalsciences.github.io/files/nips_dlps_2017_15.pdf

[18] Intel®Math Kernel Library for Deep Neural Networks, https://github.com/intel/mkl-dnn
[19] An MPI-Based Python Framework for Distributed Training with Keras, Dustin Anderson, Jean-Roch Vlimant, Maria Spiropulu, https://arxiv.org/abs/1712.05878https://github.com/vlimant/mpi_learn
[20] https://www.cscs.ch/computers/piz-daint/
[21] https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/
[22] https://github.com/vlimant/mpi_opt