Distributed Adversarial Training with JointSpar-Lars

Caspar Meijer
Master student Computer Science
Delft University of Technology
Netherlands
c.j.meijer-1@student.tudelft.nl

Gijs Paardekooper
Master student Computer Science
Delft University of Technology
Netherlands
g.w.k.paardekooper@student.tudelft.nl

Felix Kaubek
Master student Computer Science
Delft University of Technology
Netherlands
f.kaubek@student.tudelft.nl

***Abstract*—This paper aims to reproduce the Distributed Adversarial Training (DAT) platform as introduced by Zhang et al [1]. Furthermore, the aim is to reproduce their results for CIFAR10-Ext with DAT-PGD and try to reduce the training time by combining it with JOINTSPAR [2]. This method enhances the efficiency of distributed neural network training by selectively sparsifying gradient communication. By skipping computations for gradients that do not need to be communicated, it also reduces the training time. Distributed experiments were conducted with 8 nodes, each having 1 Nvidia P100 GPU available. A baseline for DAT is established comparable to the original paper. Furthermore, JOINTSPAR is successfully reproduced and it achieves a 20% speedup during normal training on the CIFAR10-Ext dataset. Combining DAT with JointSpar shows marginal speedup, due to the large overhead of the PGD attack process. The code used in this paper is provided on GitHub [1].**

Index Terms*—Distributed Machine Learning, Distributed Adversarial Training, Layerwise Adaptive Rate Scheduling, Gradient Sparsification, Layer Freezing

I. INTRODUCTION

In recent years the size of neural networks has increased rapidly. This combined with the rise in popularity of neural networks due to models like GPT3 has given rise to a new class of methods researching how to make these models mores robust against adversaries. Adversaries could for example query a deployed model with malicious data knowing the model will use the query data for updating itself. This way, an adversary could possibly alter the behavior of a neural network in a negative way. Zhang et al introduced the Distributed Adversarial Training (DAT) framework for training a neural network to be more robust against adversaries in the distributed setting [1]. This is one of the first frameworks to implement Adversarial Training (AT) in the distributed setting. They prove that their method outperforms or equals the current best robust accuracy scores.

Although training a neural network in the distributed setting gives a speedup in training time, since more GPUs can be used for dividing the work to be done, a new overhead is introduced. Each machine needs to send its gradients to the other machines and this communication time can be of significant impact on the total time as denoted by the times in table 1 and 3 in the paper by Zhang et al by $C(s)$ which can be more than half of the training time itself. A recent method proposed by Liu and Mozafari [2] called JOINTSPAR reduces both the gradient computation and communication time by skipping the gradient computation for some of the layers and reducing the amount of gradients sent. There method is based on the EXP3 multi-armed bandit method to learn what gradients to skip during gradient computation.

The goal of this paper is to first reproduce the DAT framework and then incorporate JOINTSPAR to improve the overall training time giving an even faster and possibly more robust way to do AT. We will first talk about other relevant research and papers in section II to give the reader more background knowledge about the topic at hand. Section IV-A explains what experiments we did and their specific settings such as datasets and hardware used. Section IV dives into more detail regarding the experiments themselves followed up by section IV-B which shows the results of the experiments. In section V we discuss our overall work and its results. Section VI draws conclusions based on our paper. And finally in section VII we discuss some shortcomings and what can be done in the future to extend or improve our work.

II. RELATED WORKS

The main work of this project is based on the Distributed Adversarial Training [1] paper. The authors propose DAT, a large-batch framework implemented across multiple machines to address this. The attacks are simulated on each of the nodes used independently. They demonstrate the flexibility of DAT in terms of training with both labeled and unlabeled data, and also various attack generation methods. Convergence is proven under certain conditions, while empirical results showcase its improved training speed and performance.

The second important paper this project uses, focuses more on the computation efficiency in distributed learning, trying to extend upon the previous standard of sparsifying gradients for communication [2]. With the rise of large batch training optimization, the computational powers of each machine become fully utilized, making the computation cost non-negligble. This lead to this papers introduction of JOINTSPAR, a novel algorithm using sparsification to reduce the communication cost and freezing non-relevant layers to reduce the computation cost. Existing methods compute all gradients before dropping some for communication. Unlike them JOINTSPAR tries to find the most important layers via a multi-armed-bandit and

[1]https://github.com/Capsar/dat

in doing so reduces the amount of layers that are calculated. In experiments the proposed methods achieve the same convergence rates as baseline methods, while also reducing training time by avoiding wasted computation. Experimental results further demonstrate that JOINTSPAR converges faster in terms of wall-clock time compared to various state-of-the-art compression methods.
There a various other papers solving similar problems with a similar solutions. For example, one introduces an algorithm they call FreezeOut [3]. Their motivation is a different one, namely reducing the necessary computation time, but their solution is very similar in that they also deactivate or freeze certain layers. The difference however is that FreezeOut freezes layers top down one by one after a certain iteration has been reached.

Another paper introduces another method that is even closer to JOINTSPAR, but still has some vast differences [4]. First the focus of the paper is even stronger on reducing the computation time. Furthermore the algorithm used to decide whether a layer is frozen differs vastly. Where JOINTSPAR uses a variant of the mulit-armed bandit method to determine the most important gradients, this paper uses the normalized gradient differences for all layers with weights in the model to determine which layers to Intelligently Freeze. The intelligently freezing method focuses on earlier layers to further decrease training times, because this stops the need to further compute the gradients for the rest of the model.

III. METHODOLOGY

The main aim of this section is to describe the process of implementing JOINTSPAR and to intergate it into the DAT framework. Furthermore, DAT had to be refactored quite extensively to bring it up to the PyTorch Distributed Framework standards to allow the experiments to be conducted [5].

A. Adjustments to DAT

The original DAT implementation required refactoring and extensions at several locations. First of all, it lacked the capability and explanation to be initialized dynamically, requiring manual initialization of each machine with their respective rank. Secondly, DAT was unable to support multiple GPUs on a single machine. It hard-coded PyTorch to use a the first GPU. Consequently, the provided implementation was not up to the standards of the PyTorch Distributed Framework [5].

To overcome these limitations, we have refactored the initialization and part of the training process of DAT. In short, our extensions consist of a more flexible and automated initialization process that allows DAT to run as a source distribution on Vertex AI, with the possibility of multi-node and multi-GPU support [6].

1) Multi-Node & Multi-GPU: Because of Vertex AI, each of the nodes/machines is launched with a given set of environment variables, specifically the total number of machines M and and their respective rank m. Furthermore, locally on the machine CUDA is used to identify the number of GPU/accelerators as A_m. For each accelerator on a single machine a new process is spawned given their local rank r, such that a specific process for a specific accelerator on a specific machine is defined as $A_{m,r}$. For each $A_{m,r}$ a PyTorch DistributedDataParallel (DDP) model is initialized and moved to its accelerator. As a result there are $\sum_{m=1}^{M} A_m$ processes running with their own specific model loaded to the accelerator.

B. Development of JOINTSPAR

In the absence of an openly accessible implementation of the JOINTSPAR algorithm, we implemented our own version from scratch. The main algorithm was built around the EXP3 multi-armed bandit method and it effectively disables layers by setting their gradient requirements to false. First will explain the hyperparameters that are later tuned in the experiments and secondly the ambiguities during implementation are described and how we solved them.

The algorithm uses two hyperparameters, the sparsity budget (s) and the minimal probability (p_{min}), where the first influences the amount of layers selected for the active set S, meaning the set of layers that gets their gradient calculated. The selection process works accordingly: Through the EXP3 algorithm every layer has a p value assigned. A random number $sample \in [0, 1)$ is generated. If $sample < p$ the layer will be added to the active set. As this selection process is stochastic in nature, the set value for the sparsity budget is not a strict value, but a guideline for the algorithm. p_{min} sets the minimum value that can be assigned to p.

Another point of confusion was step 10 in Algorithm 1 [2]. Here they instruct to setup a set of probabilities that have the minimal Kullback-Leibler divergence (D_{kl}) given the approximated weights w_t^m from the EXP3 multi-armed bandit method. The exact method of the constrained minimization of the D_{kl} is not explained.

For the implementation it was decided to use a gradient decent optimization on a tensor, while using the Kullback-Leibler divergence loss. After that the tensor was clipped with 1 and p_{min} to guarantee its correctness according to the constraints given.

Furthermore, the frequency at which the JOINTSPAR algorithm should re-evaluate its active set S remained unclear in the original paper. It simply mentioned a 'training iteration' without any specifics, causing ambiguity. After considering various alternatives, we chose to re-evaluate the active set S at every epoch, as this was the most in line with other methods we have seen and was confirmed to work the best on local test runs.

Besides, in the distributed setting there was a significant hurdle from the lack of clear instructions on sparsifying and encoding gradients for broadcasting. The existing implementation of LARS/LAMB does not support sparse gradients, thereby leading to contradictions, as JOINTSPAR mentions encoding gradients sparsely. To address this, we opted to allow PyTorch to manage this aspect by enabling the "*find_unused_parameters*" function within the DPP class.

This parameter introduces overhead by traversing the autograd graph to determine which gradients to skip waiting for.

IV. Experiments

A. *Experimental Setup*

We ran all experiments regarding a distributed setting on Google Cloud using the Vertex AI platform. Some of the JointSpar experiments were conducted locally to be better able to debug the process. This section includes both machine specifications as used on Vertex AI in Table I as local machine specifications in Table II. Furthermore, in Table III the hyper-parameters of the model, dataset, and adversarial settings are presented. The experiments explained and present are: experimenting with DAT as a baseline, running JointSpar locally, running JointSpar distributed, and at last the combination of DAT & JointSpar. All DAT models were put to the test on the test set t the end of the training process. With clean images and PGD attacked images in terms of accuracy, called clean accuracy (CA) and robust accuracy (RA) respectively.

TABLE I
Google Cloud Specifications

Specification	Value
Machine	n1-normal-4
CPU	Intel Xeon (Skylake)
vCPUs	4
RAM	15GB
GPU	1 x Nvidia Tesla P100
Storage	Google Cloud Storage Bucket

TABLE II
Local Machine Specifications

Specification	Value
CPU	AMD Ryzen 7950x
GPU	1 x Nvidia RTX 4090
RAM	64GB 4800 Mhz DDR5
SSD	4TB NVMe

TABLE III
Hyperparameters

Specification	Value
Model	ResNet-18
Batch size	2048
Dataset	CIFAR10 Ext [7]
Warm-up Epochs	5
Training Epochs	100
Attack Mode	PGD
Eplison (ϵ)	$8/255$
Attack steps	10
Sparsity budget	40
p_{min}	$1.25\text{e}{-2}$

1) Baseline DAT: The setup described in Table I is slightly different from the setup described in the DAT method in terms of machine count [1]. DAT describes a setup with 12 machines, each with a Nvidia P100 or V100 GPU. In our setup we tested DAT both with 4 and 8 machines as described in Table I.

2) Local JointSpar*:* Locally JointSpar was tested with the first set of hyperparameters in Table III without adversarial training. With these settings a hyperparameter search was conducted in the form of two experiments with the machine specifications in Table II. Firstly the sparsity budget was determined and afterwards the p_{min} parameter. Given that ResNet-18 has 54 layers, the sparsity budget was tested with the following values: $[30, 35, 40, 45]$ and compared with the baseline. Afterwards the $[1.25\text{e}{-1}, 1.25\text{e}{-2}, 1.25\text{e}{-3}, 1.25\text{e}{-4}, 1.25\text{e}{-5}]$ values were tested for p_{min}.

3) Distributed JointSpar*:* After having determined the hyperparameters for JointSpar locally it was time to test the JointSpar in the distributed setting. These experiments were also done without adversarial attacking to determine whether the speed up was also achieved in the cloud. 8 machines with the specifications from Table I were used.

4) DAT with JointSpar*:* The final experiment compares the baseline DAT against the DAT with additional usage of JointSpar. This was run on Vertex AI with 8 machines as described in Table I.

B. *Results*

1) Baseline DAT: In figure 1 the accuracy over the training set per second is presented. It can be seen that with double the number of machines, the total training time is halved. With 4 machines the seconds per epoch are about 600 seconds and with 8 machines it is about 300 seconds. The test accuracy results were for 4 machines: (CA) 89.4% & (RA) 52.5%, and for 8 machines: (CA) 88.0% & (RA) 52.0%.

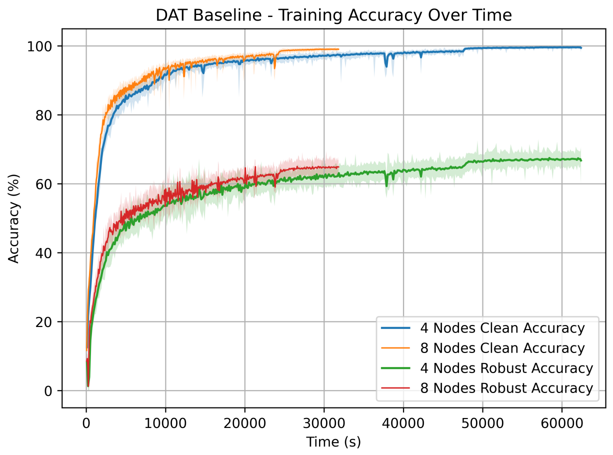

Fig. 1. DAT Comparison between JointSpar and Baseline of Training Accuracy during Training.

2) Local JointSpar*:* In Figure 2 the effects of the different sparsity budgets are presented. As can seen the sparsity budgets of 40 and 45 reach about the same test accuracy, while 30 and 35 do not reach the same level. From this 40 was chosen because it had the same accuracy levels and the baseline and was the lowest budget. Afterwards with a sparsity budget of 40, the p_{min} hyperparameter was experimented with as presented in Figure 3. The values $p_{min} < 1.25\text{e}{-2}$ showed bad convergence rates and thus the best performing p_{min} is chosen as $1.25\text{e}{-2}$.

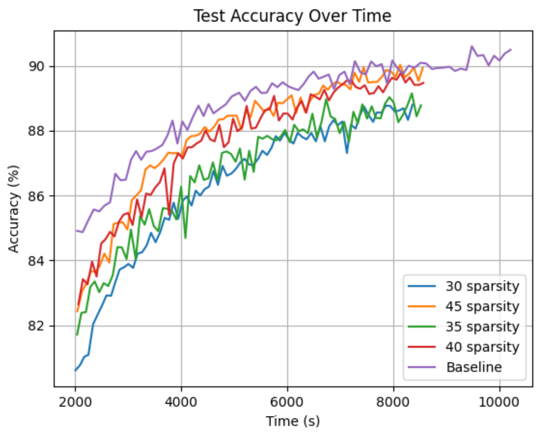

Fig. 2. Local comparison of sparsity budgets over time (zoomed in).

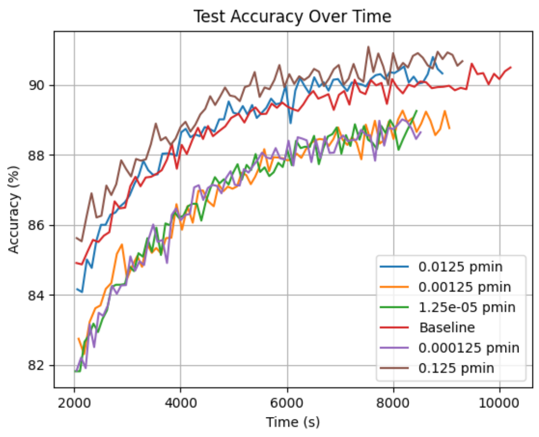

Fig. 3. Local comparison of p min values over time (zoomed in).

3) Comparison Normal Training Baseline & JOINTSPAR*:* In Figure 4 and 5 it can be seen that also in the cloud JOINTSPAR performs well. On average there is a 20% speed-up compared to the baseline. Furthermore it maintains the same accuracy on the test set as the baseline.

4) Comparison DAT Baseline & JOINTSPAR*:* In figure 7 it is presented that JOINTSPAR is on average 1 second faster than the baseline. The improvements in the total run time are non-existing as can be seen in figure 6. In terms of test accuracy, the baseline achieved a CA of 88.1% and a RA of 52.0%. The JOINTSPAR run achieved a CA of 87.5% and a RA of 51.9%.

V. DISCUSSION

We were unable to achieve the same training time per epoch as presented in the DAT paper [1], we achieved a 300 and 600 seconds per epoch with only 8 and 4 machines respectively. In the DAT paper a 450 second epoch with 12 machines was presented, from our results we would expect to achieve this

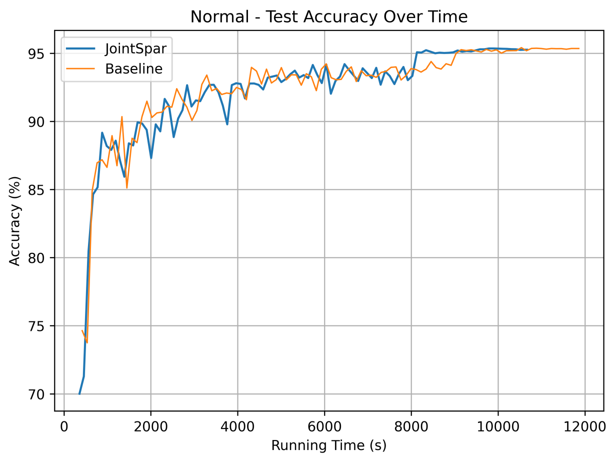

Fig. 4. Comparison between Baseline & JOINTSPAR of Training Accuracy during Normal Training.

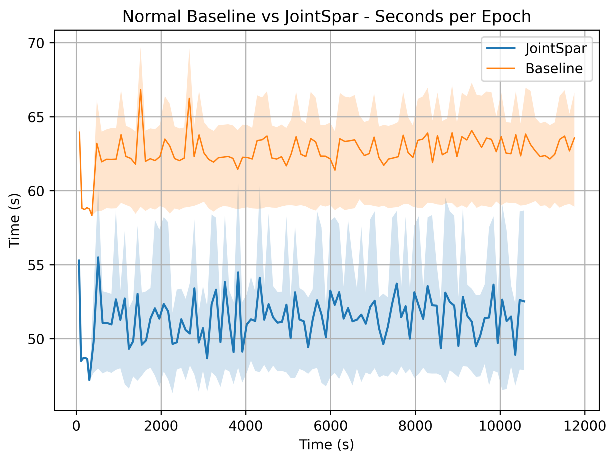

Fig. 5. Comparison between Baseline & JOINTSPAR of Epoch Duration during DAT.

with only 6 machines. That would mean a 2x improvement, sounds to-good-to-be-true. We suspect that this caused by either their 1Gigabit bandwidth setup or some other bottleneck in their training epochs, as our results in terms of Clean and Robust Accuracy is the same as in the paper.

Before implementing both methods we hypothesized that JOINTSPAR might help the robustness accuracy since it act similar to a dropout probability, disabling some layers at random. This disabling of layers at random could possible reduce the bias of the model towards the training data and therefore lead to a better robustness accuracy. Given our time limitations we did not manage to look into this any further but this could prove interesting as a future area of research regarding the topic of adversarial training.

Regarding the implementation of JOINTSPAR, the empirical results show that the speed up of about 20% is comparable to their results, even with 2x less machines and 8x less GPUs. Tuning the sparsity budget and p_{min} is not as straight forward as it seems, as the sparsity budget has to be equal to the sum

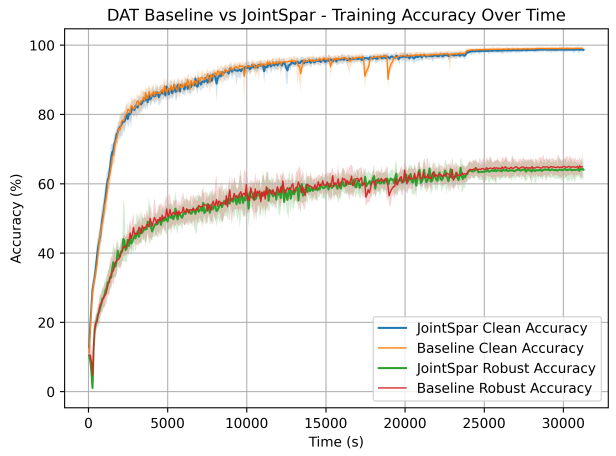

Fig. 6. Comparison between Baseline & JOINTSPAR of Training Accuracy during DAT.

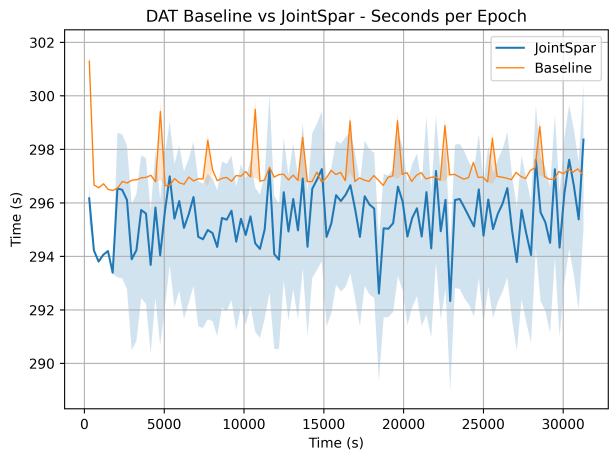

Fig. 7. Comparison between Baseline & JOINTSPAR of Epoch Duration during DAT.

of all of the probabilities, while p_{min} can be set in the range of $[0, 1)$. This upper-bound of 1 is constrictive enough, as the true upper-bound should be the sparsity budget divided by the total number of layers. Because otherwise there is the scenario where the p_{min} is set such that all probabilities summed is larger than the sparsity budget and breaking the algorithm. Furthermore, one would expect the EXP3 multi-armed bandit method to be able to correctly determine which layers should receive high probabilities to be selected such that a p_{min} variable is not required, but on the other hand its also nice to set a lower-bound probability to not leave out whole layers from the training process.

Furthermore, the marginal improvement in the wall-clock time of combining DAT and JOINTSPAR could be attributed to the significant computational time consumed by the PGD attack component of the Adversarial Training (AT) process. Even though the model had its layers disabled during the attack phase, the other components of PGD eat up most of the time.

VI. CONCLUSION

In this paper, we investigated the effectiveness of the Distributed Adversarial Training (DAT) algorithm and explored the performance of JOINTSPAR, and further the combination of both. DAT had to be refactored to work with Vertex AI and be up to the standards put forward by PyTorch and Vertex AI. Overall the implementation was challenging but the results shown in this paper are comparable in terms of accuracy and even better in terms of wall-clock time with the original author [1]. Furthermore, the JOINTSPAR implementation was straight forward, but with a couple of challenges. These challenges were overcome and the performance shown in this paper are comparable to the original [2]. We have shown the performance of both DAT and JOINTSPAR and their combination. The performance increase of the combination is marginal. Moving forward, we think the right direction for future research is enhancing the efficiency of the attack simulation component. Our analysis revealed that the attack simulation process consumed a substantial portion of the training time. Therefore, if the intend is to speed up the algorithm, efforts should be directed towards addressing this current bottleneck of the system. Furthermore, exploring novel techniques and algorithms that optimize attack simulation, especially in a distributed setting, could open doors to further advancements in the field.

In conclusion, this paper confirms the findings of both the original DAT paper [1], as well as the original JOINTSPAR paper [1]. Although the additional improvements of combining these techniques were not substantial, the effect is still noticeable and showcases the flexibility and generality of JOINTSPAR.

VII. LIMITATIONS

On the one hand we encountered a variety of implementation problems. For example, even though the DAT-paper provided a GitHub and in doing so a code base to start from, it was ultimately very outdated and missing some of the features that were mentioned in the paper itself. Additionally it was difficult to get it running with the current PyTorch library. Furthermore it was unclear how the communication time was measured, as there wasn't any inclusion of this in the code.

Similarly the paper describing JOINTSPAR, did not include actual code, but merely pseudocode which in some parts was hard to understand. Specifically Step 10 of the Algorithm 1. calls for a minimization of the Kullback-Leibler Divergence, which is not further explained on how to achieve it.

On the other hand some of our limitations were budget constraints. First in the context of GPU's we had available or were made available to us by Google Cloud. Second in the amount of credits we had available to run said servers for experiments.

References

[1] G. Zhang, S. Lu, Y. Zhang, *et al.*, *Distributed Adversarial Training to Robustify Deep Neural Networks at Scale*, arXiv:2206.06257 [cs], Sep. 2022. DOI: 10.48550/arXiv.2206.06257. [Online]. Available: http://arxiv.org/abs/2206.06257 (visited on 05/10/2023) (cit. on pp. 1, 3–5).

[2] R. Liu and B. Mozafari, "Communication-efficient Distributed Learning for Large Batch Optimization," en, in *Proceedings of the 39th International Conference on Machine Learning*, ISSN: 2640-3498, PMLR, Jun. 2022, pp. 13 925–13 946. [Online]. Available: https://proceedings.mlr.press/v162/liu22n.html (visited on 06/08/2023) (cit. on pp. 1, 2, 5).

[3] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, *FreezeOut: Accelerate Training by Progressively Freezing Layers*, arXiv:1706.04983 [cs, stat], Jun. 2017. [Online]. Available: http://arxiv.org/abs/1706.04983 (visited on 06/15/2023) (cit. on p. 2).

[4] X. Xiao, T. Bamunu Mudiyanselage, C. Ji, J. Hu, and Y. Pan, "Fast Deep Learning Training through Intelligently Freezing Layers," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Atlanta, GA, USA: IEEE, Jul. 2019, pp. 1225–1232, ISBN: 978-1-72812-980-8. DOI: 10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00205. [Online]. Available: https://ieeexplore.ieee.org/document/8875312/ (visited on 06/15/2023) (cit. on p. 2).

[5] S. Li, Y. Zhao, R. Varma, *et al.*, *PyTorch Distributed: Experiences on Accelerating Data Parallel Training*, arXiv:2006.15704 [cs], Jun. 2020. [Online]. Available: http://arxiv.org/abs/2006.15704 (visited on 06/14/2023) (cit. on p. 2).

[6] Google, *Vertex AI documentation*, en, May 2021. [Online]. Available: https://cloud.google.com/vertex-ai/docs (visited on 06/17/2023) (cit. on p. 2).

[7] Y. Carmon, A. Raghunathan, L. Schmidt, J. C. Duchi, and P. S. Liang, "Unlabeled Data Improves Adversarial Robustness," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/hash/32e0bd1497aa43e02a42f47d9d6515ad-Abstract.html (visited on 06/15/2023) (cit. on p. 3).