Strategies for memory management improvement in deep learning algorithms for contrast enhancement of high-resolution radiological images

Daniel A. Rodriguez[1,2], Daniel Sanderson[1,2], Javier Garcia Blas[2,3], Manuel Desco[1,2,4,5], and Monica Abella[1,2,5]

[1] Departamento de Bioingeniería, Universidad Carlos III de Madrid, Leganés, España, mabella@ing.uc3m.es
[2] Laboratorio de Imagen Medica, Instituto de Investigación Sanitaria Gregorio Marañón, Madrid, España
[3] Departamento de Informática, Universidad Carlos III de Madrid, Leganés, España
[4] Centro de Investigación Biomédica en Red de Salud Mental, Madrid, España
[5] Centro Nacional de Investigaciones Cardiovasculares Carlos III, Madrid, España

Abstract. This paper presents a detailed analysis of different techniques to reduce memory consumption during the training stages in a Deep Learning algorithm for contrast enhancement in radiography. This makes it possible to work with large images, such as radiological images, avoiding the problems derived from working with patches or the loss of spatial resolution caused by subsampling. For this purpose, different approaches have been studied experimentally, both in single-GPU executions and in multi-GPU systems based on data and model parallelism. Experimental evaluation shows that it is possible to achieve up to 20% reduction for a single node and up to 70% for the distributed model without loss of accuracy.

Keywords: Model parallelism · Distributed learning · Medical Image · U-Net · Deep learning.

1 Introduction

Radiology is one of the most widely used imaging methods due to its ease of use and low cost [4]. In recent years, the use of Deep Learning techniques for image processing and analysis in the medical field has become equally popular [2]. In order to visualize the images generated, it is necessary to employ very high resolutions. This can be an impediment for most neural networks, since they usually work with reduced images, as is the case of U-Net[22] with 512×512 pixels, being more common to use lower resolutions, as occurs in networks such as ResNet[10] or VGG[26], both with an input size of 224×224 pixels. This reduction in resolution can mean the loss of pixels needed to identify health problems prematurely. Thus, the closer we can work to the original resolution of the image, the more reliable the results can be.

Nowadays, there are different approaches to reduce memory consumption in distributed environments. However, most of those techniques are focused on large language models, where the number of the network's parameters can occupy tens of GBs. Examples are GPT-2[19] with 1.5 billion parameters or T5[20], with up to 11 billion parameters. These reductions come through more efficient use of parameters and communication [25]. In the medical imaging field, they often solve these problems by developing new training methods [11]. Nevertheless, as reported in [6], radiology offers little contrast in soft tissue. Conventional processing algorithms are available to perform contrast enhancement on them and partially overcome these drawbacks (see Figure 1). Additionally, those conventional algorithms rely on a high number of parameters and require manual adjustments depending on the use case [3]. Therefore, alternatives based on deep learning have started to be developed to eliminate this parameter selection and allow automatic processing [15].

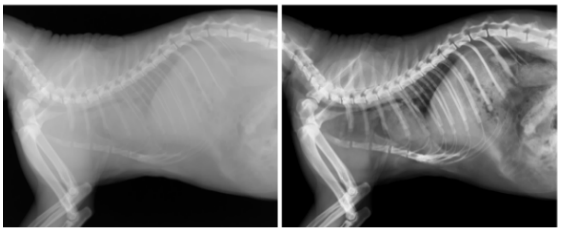

Fig. 1. Cat chest radiography in raw data (left) and processed by conventional algorithms (right).

The main goal of this paper is the evaluation of different techniques available to reduce GPU memory consumption in order to increase the size of the training images without compromising the quality of the results. For that tasks, we part from the model previously developed in [6], which uses a U-Net network [22] based on ResNet-34, with the Multiscale Structure Similarity Index (MS-SSIM) [28] as a cost function. U-Net is a network divided into two distinct parts: *Encoder*, which reduces and encodes the image, and *Decoder*, which is responsible for recovering the original size. The original resolution of the images used can be up to 4188×4188 pixels. However, due to the memory limitations of these algorithms, the current maximum size that can be trained on an 8GB GPU is 1440×1440 pixels. Even if new GPUs (e.i., NVIDIA L40S) provided far more memory, the memory wall is a predominant barrier to use larger images or to increase the batch size used during training.

The remainder of the document is divided as follows. In Section 2, similar previous works are presented. The evaluated and applicable techniques for a single node and distributed training of the model are introduced in Section 3. The methodology used to carry out the experiments of this work is detailed in Section 4. Next, Section 5 presents our discussion of the results obtained for the

different techniques. Finally, Section 6 summarizes the conclusions drawn based on the work performed and the results obtained.

2 Related work

There are numerous works aimed at reducing GPU memory consumption for Deep Learning algorithms. However, the vast majority of these works are oriented to the development of new techniques and methodologies, some of which are listed later in Section 3. In the analysis and comparison of techniques, it is common to find compilation works such as [18] and [31], which although they compile and describe in detail a multitude of techniques for different situations, they lack of comparable results.

In [24], a framework for automatic selection of some network parameters and automatic application of checkpointing [5, 8] is developed. However, the authors missed to evaluate the results of the network, only the memory consumption. In [32], authors evaluated the impact of *quantization* on different models targeting low-resource devices. This aforementioned work performs an evaluation on both memory consumption and accuracy on a very specific set of models and techniques. Finally, [16] performed an evaluation on different models of some of the most known techniques and combinations between them. This work lacks of completeness as not all models have been evaluated under the same conditions. Memory consumption has been evaluated through batch size, and no loss or accuracy values resulting from training are compared.

3 Techniques

In this section, we present and briefly describe the techniques evaluated to optimize memory consumption during the training of neural networks.

3.1 Sequential techniques

Sequential techniques seek to reduce memory consumption on a single GPU, either by delaying computations (discarding intermediate values) or by changing the algorithm of the network itself. Below, we introduce the most popular ones:

- *Gradient checkpointing* [5, 8] reduces memory consumption in exchange for computation time. For this purpose, intermediate computations that are saved during the *forward* stage are discarded and computed from scratch during *back-propagation*.
- *Automatic Mixed Precision (AMP)* [17] is based on determining the calculations during training that do not require operating with single precision floating point numbers. In those cases, the precision can be reduced to 16 bits without loss of quality. There are three possible approaches:
  - *float16*: 1 sign bit, 10 bits for precision, and 5 bits for range.

  - *bfloat16*: Developed by Google [1]. 1 sign bit, 7 bits for precision, and 8 bits for range.
  - *TensorFloat32*: Developed by NVIDIA[14] and available on their graphics cards from the 3000 series upwards. 1-bit sign, 10 bits for precision, and 8 bits for range.
- *Quantization*[9] corresponds to a set of techniques that seek to reduce memory and network resource consumption by reducing overall precision (similar to AMP) and combining operations to reduce intermediate memory consumption. This reduction of precision has a negative impact on accuracy.
- *Invertible ResNet* [13, 7] is a specific technique for residual networks. It is based on being able to reconstruct the activations of a layer only with information from the previous layer. As a result, it is only necessary to store the last activations.

3.2 Distributed techniques

Distributed techniques seek to distribute the workload among different computational nodes. In the case of neural networks, training is distributed among several GPUs that may be on the same node or distributed among several interconnected machines. The most popular techniques include:

- *Data parallelism (DP)*: the model is fully replicated on different GPUs and the training data is distributed. In this type of parallelism, network parameter updates must be synchronized to avoid inconsistencies.
- *Model parallelism (MP)*: data are fed into the network sequentially, but the model is distributed among different GPUs. Network outputs must be transmitted between devices and network parameter updates must be synchronized. It is not automatic (automatic approximations are being developed [27, 30]) and often requires network modifications and changes to the training function.
- *Sharding parameters*[21, 29]: This optimization algorithm seeks to reduce the duplicity of parameters on data parallelism. To this end, when performing DP, the parameters are not duplicated but distributed among the different devices.
- *Pipelines*[12]: improvement applicable to MP. Since data ingestion is sequential in MP, GPUs utilization declines compared to DP or no parallelism. *Pipelines* seek to increase the utilization rate by subdividing the input data into smaller chunks, so multiple inputs can be processed at the same time. It increases GPU occupancy and thus reduces training time.

4 Methodology

In order to carry out the experimental evaluation of the techniques introduced in Section 3, we firstly describe the software employed. We analyze a U-NET architecture based on *Resnet-34* pre-trained with the *ImageNet* database. In

order to apply the network to radiological images in gray levels, the weights of the three color channels of the first convolutional layer have been summed. To normalize the output of the network and to avoid the appearance of intensity peaks in the resulting image, a sigmoid with a dynamic range of [0,1] has been used.

The network was trained with 24 cat images, 34 dog images, and 2 turtle images. The resolution of these radiography ranges from 1288×950 to 4188×4188 pixels. The images have been normalized to a range of intensities between 0 and 1. The training set consists of 48 randomly chosen images and 12 images for validation, which are 5 dog images, 6 cat images, and 1 turtle image. A test set was not used. The reference images were obtained using conventional algorithms and validated by an expert radiologist.

Python 3.10 has been used to generate the results, together with the PyTorch library version 2.2. For the implementation of the invertible ResNet, the *MomentumNet* library [23] was used. Results obtained with one GPU were computed with an NVIDIA RTX 2060 GPU with 8GB of RAM and driver 545.23.08. For the multi-GPU tests, we used three nodes with one NVIDIA 3070 GPU with 8GB RAM each, driver 535.161.07, and interconnected via 10 Gbit/s Ethernet.

Memory consumption profiles have been recorded using the *pynvml* package, which provides access to the *NVIDIA Management Library* to check the GPU status. In every scenario, the same consumption has been measured: the *forward* and *backward* phases during the training of the neural network. For this purpose, three image sizes have been used: 512×512, 1024×1024 (which were already trainable in the original version), and 2048×2048 pixels, which would be the next ideal image size with which to train the model and is currently not possible. Different combinations of the techniques presented above have been applied to each image size. For the calculation of loss functions, the network has been trained for 50 epochs.

In the single-GPU run, all the techniques listed in the subsection 3.1 were evaluated. For the multi-GPU experimentation, we evaluated data and model parallelism in combination with single GPU techniques. To evaluate model parallelism, each U-Net part, *Encoder* and *Decoder*, have been sent to a different GPU. The model division, parameter, and node synchronization have been done entirely with PyTorch. As each GPU is going to process *forward* and *backward* stages, the indicated memory consumption are the average.

5 Experimental results

This section shows and discusses the results obtained by applying the different techniques in the two scenarios proposed: single GPU and multiple GPUs.

5.1 Single GPU

Figure 2 summarizes memory consumption for image size 512×512 pixels and 1024×1024 pixels. There are no results for 2048×2048 pixels since only the

AMP and *Checkpointing* methods were able to execute one iteration of *forward*, but none of *backward*.

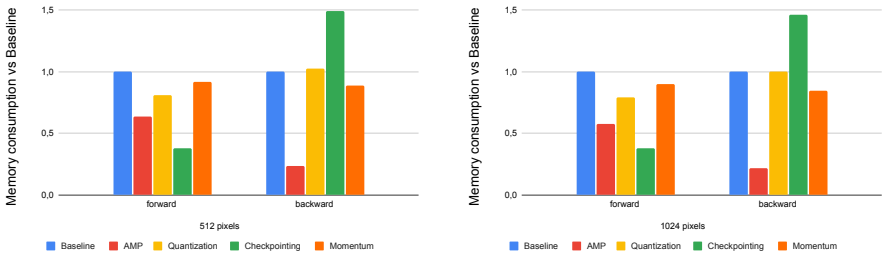

Fig. 2. Memory consumption ratio analyzed for the different techniques executed on a single GPU and image size of 512 and 1024 pixels. Smaller values are better.

As can be seen, all the techniques reflect memory reduction at the *forward* stage, with *AMP* standing out, since data itself is represented 16 bits instead of 32 bits in memory, and *Checkpointing* since all the calculations that should be saved are discarded to be subsequently recalculated. In the *backward* phase, *AMP* considerably reduces memory usage to almost 20% of the original memory. On the other hand, since *Checkpointing* must recalculate all previously discarded values, its memory consumption is multiplied making the total consumption equal to or higher than the original consumption. *Quantization* and *Momentum-Net* maintain similar values in the *backward* phase, reducing memory by about 15% in the case of *Momentum-Net*. Another parameter to consider, given that we are modifying how the model works and the values that are stored in the model, are the values of the loss function in training and validation for each of the techniques.

Table 1. Loss values for 512×512 pixels for each of the different techniques (a) and combinations (b). The best results are highlighted.

(a)

	Train Loss	Validation Loss
Original	4.0504	6.0083
AMP	4.8612	6.2515
Quantization (Quant)	4.7272	**5.9758**
Checkpointing (Chkp)	**4.0482**	6.0592
Momentum-Net (Mom)	4.1456	5.9941

(b)

	Train Loss	Validation Loss
Original	**4.0504**	**6.0083**
AMP+Chkp	4.8612	6.2761
AMP+Quant	-	-
Chkp+Mom	4.1456	6.0115
AMP+Chkp+Mom	15.1742	17.2537

As can be seen in Table 1A, the worst results are obtained by *AMP*. This is due to the reduction of accuracy of the tensors, since, as mentioned at the beginning of this work, the lower the resolution, the worse the results. Contrary to this reasoning, *Quantization* returns the best validation loss, however, among the four techniques, *Quantization* was the most unstable with a high error rate and low successful executions compared with the others. The rest of the techniques are always successfully executed and some of them maintain results close, or better, to the original values.

Since they are not mutually exclusive techniques, different combinations have been evaluated in order to study their impact. The results can be seen in Table 1B. The best combination is provided by *Checkpointing* and *Momentum-Net*, with a lower memory consumption than individually, as can be seen in Table 2, although it does not achieve a lower memory consumption than using *AMP*. This is because it is possible to maintain the gain of *Checkpointing* in the *forward* phase without such a large penalty in the *backward* phase thanks to the invertible *ResNet* introduced with *Momentum-Net*. This combination makes it possible to achieve reductions of up to 20%. This memory reduction is not as large as with *AMP*, which is up to 50%. However, if our use case does not allow precision reduction, the combination of these techniques is a good alternative. There is no *AMP + Quant* results as due to the loss of precision it never successfully finished. Combining *Quantization* or *AMP* with other techniques increases the final loss due to precision changes. On the other hand, it also increases memory consumption compared to their individual versions. As for the execution time, it remained stable without increasing or decreasing considerably for each of the techniques.

Table 2. Memory consumption in MB for 512×512 pixels and average execution time per epoch. Smaller values are better.

	Forward	Backward	Time (s)		Forward	Backward	Time (s)
Original	757.76	440.32	35.63	AMP + Chkp	266.24	194.56	34.30
AMP	481.28	102.40	34.02	AMP + Quant	-	-	-
Quant	614.40	450.56	35.28	Chkp + Mom	337.92	563.20	37.52
Chkp	286.72	655.36	35.15	AMP + Chkp + Mom	317.44	286.72	32.36
Mom	696.32	389.12	37.91				

These memory improvements using AMP and its combinations already allow to train the model with images of 2048×2048 pixels. However, it has to be taken into account the train and validation loss increase in Table 1 to consider the quality of the results.

5.2 Multi GPU

Here, an analysis of the memory usage and results using multiple GPUs is presented. Firstly, results from data parallelism (DP) are discussed. Lastly, results

obtained with model parallelism (MP) are presented and compared with previous results.

Data parallelism In this scenario, when using multiple GPUs, the model is usually completely replicated through the different devices. However, there are few techniques that can be applied to reduce redundancies and therefore, reduce memory usage. It is the case of *Sharding parameters*. This technique does not fully replicate the model through the different GPUs but rather splits the parameters so each GPU has a partial view of the whole set of parameters. The selected technique is the Zero Redundancy Optimizer (ZeRO) [21], already implemented in PyTorch. Table 3, shows the loss values obtained when doing DP and combining it with different techniques, Table 4 shows the memory usage for these combinations.

Table 3. Loss values for 512 × 512 pixels combining different techniques using one node with two GPUs. The best results are highlighted.

	Train Loss	Validation Loss		Train Loss	Validation Loss
Original	4,0504	6,0083	DP + ZeRO + AMP	2,4507	6,2196
DP	2,1602	6,1418	DP + ZeRO + Chkp	2,1602	**5,8895**
DP + AMP	2,4507	6,2196	DP + ZeRO + Mom	**1,5953**	6,8307
DP + ZeRO	2,2267	6,1418	DP+ZeRO+AMP+Chkp	2,1586	6,2355

As can be seen, training loss has been reduced significantly compared with the original model. This can be misleading because this loss is calculated based on the input image. As each GPU is training a subset of the dataset, the models on each GPU "overfit" more easily to their subset than before. However, when validating the model with the validation set, loss remains similar to the original values, which means that the final model is not overfitted to the training data. It is also worth noticing that ZeRO does not change the validation loss, as the other techniques may do.

Memory consumption behaves similarly to using only one GPU, *AMP* reduces memory in both stages, *Checkpointing* reduces forward and increases backward, and *Momentum-Net* mainly reduces backward. The ZeRO optimizer slightly reduces memory consumption, as only a few GPUs are being used and when combined with other techniques, does not increase memory usage. The biggest reduction possible is combining all the techniques to get almost a 65% reduction. However, validation loss increases in this case. The best memory reductions also allow training using images of $2048x2048$ pixels, but only if the data structure used to split the data between the GPUs is allocated in another computer. This is because its size increases with the dataset size.

Model parallelism In this scenario, as each node has a single GPU, only single-node improvements were evaluated along model parallelism. As it was

Table 4. Memory consumption in MB for 512 × 512 pixels using one node with two GPUs. The best results are highlighted.

	Forward	Backward	Total	Time (s)
Original	757,76	440,32	1.198,09	35,63
DP	757,76	532,48	1290,24	14,44
DP + AMP	491,52	**122,88**	614,4	**10,01**
DP + ZeRO	768,00	337,92	1105,92	12,97
DP + ZeRO + AMP	491,52	**122,88**	614,4	13,46
DP + ZeRO + Chkp	296,96	501,76	798,72	15,24
DP + ZeRO + Mom	686,08	276,48	962,56	14,44
DP + ZeRO + AMP + Chkp	**276,48**	153,60	**430,08**	12,98

not possible to combine model and data parallelism. Table 5 depicts the loss values obtained in this distributed environment combining different techniques and Table 6 shows the memory usage for these combinations.

Table 5. Loss values for 512×512 pixels combining different techniques in a distributed environment. The best results are highlighted.

	Train Loss	Validation Loss		Train Loss	Validation Loss
Original	**4,0504**	6,0083	MP + Mom	4,6009	5,9407
MP	4,5295	5,9778	MP + AMP + Chkp	4,6431	5,7533
MP + AMP	4,6431	5,7533	MP + AMP + Mom	4,5370	5,7758
MP + Chkp	4,4820	**5,6649**	MP+AMP+Chkp+Mom	4,5370	5,7758

Parallelizing the model on two GPUs has reduced the total memory consumption to 39% for 512 × 512 pixel images. If we apply in this distributed implementation the techniques seen for the execution on a single GPU, we can observe that the results follow a similar trend as before. As it can be seen in Table 6, when mixing MP + AMP with other techniques, memory consumption increases considerably depending on the technique during the *backward* stage for all techniques. Only the combination of *AMP* and *Checkpointing* provides an average memory consumption equal to the original distributed execution (see Table 6). This is because the memory consumption during the *forward* stage decreases as the increase in the *backward* stage. Unlike sequential and DP execution, the average execution time for each epoch increases by 14% compared with sequential execution and by 306% with DP with input size 512 × 512 pixels. This is due to the overhead introduced by having to communicate and synchronize the nodes during training.

Despite this slower execution, the combination of techniques in the same way as in the single node execution makes it possible to achieve an average reduction in memory consumption per GPU of up to 70%. This makes it possible to train the model with 2048 × 2048 pixel images, something that was not possible before.

Table 6. Memory consumption in MB for 512×512 pixels in distributed environment and average execution time per epoch. The best results are highlighted.

	Forward	Backward	Total	Avg. execution time (s)
Original	757,76	440,32	1.198,09	**35,63**
MP	424,96	307,20	732,16	53,29
MP + AMP	281,60	**117,76**	399,36	40,69
MP + Chkp	215,04	399,36	614,40	57,32
MP + Mom	445,44	225,28	670,72	57,13
MP + AMP + Chkp	**163,84**	230,40	**394,24**	57,26
MP + AMP + Mom	312,32	158,72	471,04	58,36
MP + AMP + Chkp + Mom	204,80	291,84	496,64	50,02

Theoretically, MP and one GPU execution should return the same train and validation loss, however, the loss obtained in the MP implementation differs from the single node due to the communication protocol used in PyTorch for MP using multiple nodes. Due to this, it is not possible to use a scheduler for the learning rate, which is fixed during the whole training. Unlike using only one node (see Table 1B), combining *AMP* with other techniques does not worsen the results as before. This is because NVIDIA 3000 series cards are used in the distributed nodes, which include *TensorCores*, allowing a slight increase in accuracy over *float16* when performing *AMP*. For the use case presented, this small difference is very important, as it can mean the difference between soft tissue or bone.

6 Conclusions

This work demonstrates the usefulness of current techniques for memory reduction, both in single GPU and distributed environments. By combining different techniques, memory consumption can be reduced by 20% in single node environments and 70% in distributed environments without significant loss of accuracy. This does not mean that the techniques discussed are always compatible with any use case or neural network, where reducing the accuracy of the operations worsened the results considerably in some cases. Additionally, the capabilities of the library being used must be taken into account, since not all of them include implementations for the discussed techniques. For example, PyTorch does not include tools to easily build an invertible network, and therefore the MomentumNet library was used to create an invertible ResNet.

Model parallelism through interconnected nodes (one GPU per node) currently limits the techniques that can be used, such as schedulers for the learning rate or *pipelines*, since these are not compatible formats with the communication and synchronization system implemented by Pytorch for model parallelism. Still, the results are promising and even with these limitations the final results are very close to the original run, proving to be a viable alternative for memory reduction.

As future work, we are considering the use of elastic solutions that allow to adapt the training loads to the available resources. This would allow better load balancing on existing resources between CPU and GPU. Additionally, we are also working on addressing memory consumption reduction techniques under data parallelization systems. Finally, we are also working on the implementation of swarms of neuron networks to improve the accuracy of the model. This work will also require a higher degree of parallelism, so performance enhancement techniques will need to be addressed.

References

1. BFloat16: The secret to high performance on Cloud TPUs — Google Cloud Blog — cloud.google.com. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus (2024)
2. Anaya-Isaza, A., Mera-Jiménez, L., Zequera-Diaz, M.: An overview of deep learning in medical imaging. Informatics in Medicine Unlocked **26**, 100723 (2021). https://doi.org/10.1016/j.imu.2021.100723, https://www.sciencedirect.com/science/article/pii/S2352914821002033
3. Ačkar, H., Almisreb, A., Saleh, M.: A Review on Image Enhancement Techniques. Southeast Europe Journal of Soft Computing **8** (2019)
4. Bhargavan, M., Sunshine, J.H.: Utilization of Radiology Services in the United States: Levels and Trends in Modalities, Regions, and Populations. Radiology **234**(3), 824–832 (2005). https://doi.org/10.1148/radiol.2343031536, https://doi.org/10.1148/radiol.2343031536, pMID: 15681686
5. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training Deep Nets with Sublinear Memory Cost (2016)
6. Daniel, S., Manuel, D., Carlos, Fernández, D.C., M, I, G.R., M, J, R.F., Javier, G.B., Mónica, A.: Corrección automática del contraste en imagen radiográfica mediante aprendizaje profundo, pp. 10–13. CASEIB (November 2022)
7. Etmann, C., Ke, R., Schönlieb, C.B.: iUNets: Fully invertible U-Nets with Learnable Up- and Downsampling (2020)
8. Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., Graves, A.: Memory-Efficient Backpropagation Through Time (2016)
9. Han, S., Mao, H., Dally, W.J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding (2016)
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition (2015)
11. Hou, L., Cheng, Y., Shazeer, N., Parmar, N., Li, Y., Korfiatis, P., Drucker, T.M., Blezek, D.J., Song, X.: High Resolution Medical Image Analysis with Spatial Partitioning (2019)
12. Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M.X., Chen, D., Lee, H., Ngiam, J., Le, Q.V., Wu, Y., Chen, Z.: GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism (2019)
13. Jacobsen, J.H., Smeulders, A., Oyallon, E.: i-RevNet: Deep Invertible Networks (2018)
14. Kharya, P.: NVIDIA Blogs: TensorFloat-32 Accelerates AI Training HPC upto 20x — blogs.nvidia.com. https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/ (2024)

15. Krouma, H., Ferdi, Y., Taleb-Ahmedx, A.: Neural Adaptive Fractional Order Differential based Algorithm for Medical Image Enhancement. In: 2018 International Conference on Signal, Image, Vision and their Applications (SIVA). pp. 1–6 (2018)
16. Liu, X., Jha, S., Cheung, A.: An evaluation of memory optimization methods for training neural networks (2023)
17. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., Wu, H.: Mixed Precision Training (2018)
18. Mittal, S., Vaishay, S.: A survey of techniques for optimizing deep learning on GPUs. Journal of Systems Architecture **99**, 101635 (2019)
19. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019)
20. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research **21**(140), 1–67 (2020), http://jmlr.org/papers/v21/20-074.html
21. Rajbhandari, S., Rasley, J., Ruwase, O., He, Y.: ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (2020)
22. Ronneberger, O., Fischer, P., Brox, T.: U-Net: Convolutional Networks for Biomedical Image Segmentation (2015)
23. Sander, M.E., Ablin, P., Blondel, M., Peyré, G.: Momentum Residual Neural Networks (2021)
24. Shah, A., Wu, C.Y., Mohan, J., Chidambaram, V., Krähenbühl, P.: Memory Optimization for Deep Networks (2021)
25. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism (2020)
26. Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition (2015)
27. Tavarageri, S., Sridharan, S., Kaul, B.: Automatic Model Parallelism for Deep Neural Networks with Compiler and Hardware Support (2019)
28. Wang, Z., Simoncelli, E., Bovik, A.: Multi-Scale Structural Similarity for Image Quality Assessment. Proceedings of the IEEE Asilomar Conference Signals, Systems and Computers (2004)
29. Xu, Y., Lee, H., Chen, D., Choi, H., Hechtman, B., Wang, S.: Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training (2020)
30. Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E.P., Gonzalez, J.E., Stoica, I.: Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning (2022)
31. Zhuang, B., Liu, J., Pan, Z., He, H., Weng, Y., Shen, C.: A Survey on Efficient Training of Transformers (2023)
32. Zhuo, S., Chen, H., Ramakrishnan, R.K., Chen, T., Feng, C., Lin, Y., Zhang, P., Shen, L.: An Empirical Study of Low Precision Quantization for TinyML (2022)