Profiling the real world potential of neural network compression

Joe Lorentz*, Assaad Moawad†, Thomas Hartmann‡ and Djamila Aouada§
DataThings S.A.
Luxembourg
Email: *joe.lorentz@datathings.com, †assaad.moawad@datathings.com, ‡thomas.hartmann@datathings.com,
SnT, University of Luxembourg
Luxembourg
Email: *joe.lorentz@ext.uni.lu, §djamila.aouada@uni.lu,

***Abstract*—Many real world computer vision applications are required to run on hardware with limited computing power, often referred to as ”edge devices”. The state of the art in computer vision continues towards ever bigger and deeper neural networks with equally rising computational requirements. Model compression methods promise to substantially reduce the computation time and memory demands with little to no impact on the model robustness. However, evaluation of the compression is mostly based on theoretic speedups in terms of required floating-point operations. This work offers a tool to profile the actual speedup offered by several compression algorithms. Our results show a significant discrepancy between the theoretical and actual speedup on various hardware setups. Furthermore, we show the potential of model compressions and highlight the importance of selecting the right compression algorithm for a target task and hardware. The code to reproduce our experiments is available at *https://hub.datathings.com/papers/2022-coins*.**

***Keywords*—machine learning, computer vision, model compression, edge computation**

I. Introduction

Driven by recent developments like the internet of things [1], many processes are becoming increasingly automated and data-driven. Machine learning and computer vision are used for an ever rising number of domains and applications, e.g. autonomous driving [2], industrial quality assurance [3], space systems [4], or surveillance systems [5]. Data transmission is problematic for such applications, due to the inferred latency and can even be impossible at certain circumstances, for example when a device in space is disconnected. Therefore, data often needs to be processed close to its source, on so-called edge devices, which usually have limited computational power [6].

Meanwhile, the state-of-the-art in computer vision and machine learning is moving towards ever bigger and deeper neural networks [7]. Since the groundwork of Krizhevsky et al.[8], convolutional neural networks (CNNs) have provided state-of-the-art results on multiple computer vision challenges [9, 10, 11]. Now, it is common for CNNs to have more than 100 layers and millions of parameters [12, 13, 14]. The downside of the high model robustness are equally high memory and time requirements for training and inference. The research field of DNN compression aims at lowering the resource demands while keeping the model accuracy high [7]. The main compression methodologies are pruning [15], tensor decomposition [16], quantization [17] and knowledge distillation [18]. State-of-the-art methods report significant compression potential with negligible drop in accuracy [19, 20, 21, 22]. The achieved compression is, however, usually based on theoretic values. Acceleration is commonly estimated based on the number of floating point operations (FLOPs) needed to execute a given model and its compressed counterpart [15]. This FLOP estimation can be very misleading, diverging significantly from the observed acceleration on a given hardware setup. The main goal of this paper is to quantify this discrepancy and provide a better understanding of the potential of model compression. Another issue is that the literature focuses on the inference stage, i.e. forward computation of models and rarely considers the impact on the backward computation. However, for some applications on the edge, the backward stage is equally important [4, 6]. Online learning scenarios, for example, require parts or the whole training to be executed directly on the target device [6]. Other applications might require real-time decision explanations, while state-of-the-art explainable AI methods use partial or even full backward computations [23, 24]. Therefore, this work considers both forward and backward computations.

The contribution of this work can be summarized as follows:

1) implementation of a simple model profiling tool
2) codebase to experiment with structured pruning and tensor decomposition
3) investigation of theoretical vs observed speedup
4) investigation of the potential of compression for applications on the edge

The remainder of this work is structured as follows. Section II provides an overview of model compression methods

The present project is supported by the National Research Fund, Luxembourg; Project Reference 14297122
The final authenticated version is available online at https://doi.ieeecomputersociety.org/10.1109/COINS54846.2022.9854973

as well as existing profiling alternatives. Our profiling tool is presented in Section III. Experiments and results are given in Section IV and concluding thoughts are given in Section V.

II. Related work

Deep neural network compression is a well established field [7]. Existing methods can be roughly put into 4 categories: pruning, tensor decomposition, quantization and knowledge distillation.

Pruning methods try to identify the least important weights of a model [15]. The general assumption is that individual weights or groups of unimportant weights can be removed with negligible impact on the model output. The idea of pruning was pioneered by LeCun et al. [25] and has gained popularity as deep learning started to dominate the AI research. The main features of different pruning methods are the algorithm to evaluate weight importance and the removal granularity. Unstructured methods [19, 26] remove singular weights and in general achieve the best theoretic compression-accuracy trade-offs. However, this leads to sparse weight matrices, which require specialized software or hardware to grant any actual acceleration [15]. In structured pruning [20, 27, 28], groups of weights (e.g. whole filters) are removed to enable a measurable speedup with commodity hardware and software. The downside of filter-level pruning are a lower compression potential as well as inter-layer-dependencies [20] that need to be taken care off.

Tensor decomposition is based on the assumption that the layers of uncompressed models are over-parameterized and their weight tensors can be approximated by lower rank tensors [29]. Essentially the goal is to decompose individual layers into several smaller layers, which combined require fewer operations as the original. Fully connected layers can be decomposed into two steps by applying singular value decomposition, with the rank of the approximated matrix, as a hyperparameter for how much the layer is compressed. Similarly, convolutional layers, can be decomposed using either CP [30] or Tucker [16] decomposition. Decomposition of convolutional layers leads to point-wise and depth-wise convolutions, which are common patterns in efficient network design [31]. Tensor decomposition requires careful tuning of the target rank parameter for each layer. In [16], variational Bayesian matrix factorization (VBMF) [32] is used to automate the hyperparameter tuning.

The goal of quantization is to map the weight tensors, usually stored in float32, to a lower precision, e.g. float16 or int8 [17]. Lower precision values require less memory to store and computations on them are also faster. Compressing forward computations essentially revolves around finding optimal buckets to map multiple weights to, such that the change in the output remains minimal. However, training with quantized networks is challenging as weight updates based on low precision are prone to being "rounded off" [33]. Hence, quantization is either used for inference only or rely on specialized training techniques [33]. With knowledge distillation, the goal is not to compress the original model but rather to train a smaller model directly to mimic the behavior of the original large model [18].

Aside from compression, other techniques to reduce the memory footprint [34, 35] or to optimize CNNs for inference [36] exist but are considered out of scope of this work. Such techniques optimize the execution of a given model rather than the model architecture itself and are therefore, orthogonal to model compression. This work focuses on structured pruning and tensor decomposition as both methods compress the original model directly and the literature suggests that significant reduction in computation time and memory consumption of both forward and backward computations should be expected [7]. Furthermore, pruned or decomposed models, can be fine-tuned with classical gradient-descent to regain robustness [15, 29].

PyTorch [37] is a widely used framework for CNNs with a constantly rising number of users, especially from the research community [38]. PyTorch features an in-build profiling utility to track function executions during model training and inference[1]. NvProf[2] and Nsight systems[3] are tools provided by Nvidia to profile program executions on Nvidia GPUs. PyProf[4] can be used to analyze these profiles when running models defined in PyTorch. More recently, Nvidia has moved to a new utility called DlProf[5]. The Nvidia tools are, however, restricted to profiling on GPU and cannot provide insights for practitioners who are interested in running models on a CPU. In this work, we implement a utility to profile the forward and backward executions of PyTorch models on any device. Other benchmarks on neural network compression exist [7, 15], however, to the best of our knowledge, this is the first work that systematically investigates the discrepancy between theoretic and observed compression.

III. Profiling PyTorch models

For this work, we implemented a new tool to profile PyTorch models. Our profiler provides information on the estimated memory consumption and computational complexity, as well as measured execution time for forward and backward computation. For measuring the memory consumption, we rely purely on estimations. More specifically, we reuse an existing estimator[6], based on counting the bytes which are necessary to store all model tensors (weights, inputs, outputs, intermediate). The same tool is also used to estimate the computational complexity of models. To the best of our knowledge, it is not possible to determine the exact allocated memory (both CPU and GPU) while running a PyTorch model directly via an API. Tracking exact memory consumption is only possible via external tools, such as htop[7] or nvidia-smi[8]. Initial experiments

[1]https://pytorch.org/docs/stable/profiler.html
[2]https://developer.nvidia.com/nvidia-visual-profiler
[3]https://developer.nvidia.com/nsight-systems
[4]https://github.com/NVIDIA/PyProf
[5]https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/
[6]https://github.com/TylerYep/torchinfo
[7]https://htop.dev/
[8]https://developer.nvidia.com/nvidia-system-management-interface

using these tools confirmed that existing estimations are close to reality and only diverge by a base amount, necessary for running PyTorch, and are not directly related to the model. Counting the floating point operations (FLOPs) needed for a forward pass on a model is commonly used in the literature to estimate computational complexity [15]. FLOP estimations can, however, not take the underlying hardware into account. The actual latency of two models with similar FLOP count may diverge drastically when executed on the same hardware. For this reason, we create an additional time profile, measuring the actual latency. To get reliable numbers, it is important to repeat the measurements many times and report the median. This ensures that random influence of unrelated processes does not distort the profile. Additionally, untracked warm up iterations help to further increase the reliability of the measurements as the first few iterations usually take longer due to necessary background tasks of PyTorch. The number of warm up and profiling iterations can be changed by the user. We determined empirically that 5 iterations for warm up and 50 for profiling are usually enough and are therefore, used by default. Time-profiles are measured individually for two scenarios. For the forward-only (inference) use-case, gradient computation is disabled and the model is set to ”eval-mode”, which disables dropout layers and the moving average computation of batch normalization layers. In a loop, the model is queried with a random input tensor and a timestamp before each iteration is noted with Python’s time module. The execution is synchronized at the start of each iteration, which is particularly important when profiling on GPU due to the extensive use of parallelization.

For the training scenario a similar process is repeated. The model is set to ”train-mode” and queried again with the same random input tensor. The output tensor is then used to compute the cross-validation loss against a random target tensor and the gradients are computed for this loss in a backward pass before synchronizing and taking a timestamp. After all iterations have been processed, the execution times are computed based on the tracked timestamps and a median value is calculated. The input batch size defaults to 1 but can be changed to investigate realistic training scenarios.

In summary, the tool provides: the number of parameters used, the FLOP estimation of the forward pass, the total estimated memory as well as individual values required for storing parameters and intermediate compute tensors (layer inputs and outputs). The time-profile indicates latencies for each forward-only and training iteration, as well as the median value for both.

IV. Experiments

All experiments were conducted with Python 3.8.5 and PyTorch 1.9.1. Four different hardware scenarios were investigated, two on GPU and two on CPU. The commodity Nvidia Quadro T2000 offered the highest computational power with 1024 CUDA cores. Nvidia Jetson Nano, with 128 CUDA cores, was used as a potential GPU for edge computation. The CPU experiments were run on an Intel i7-985H, with and without multi-threading enabled. Two model-dataset combinations were investigated. We used ResNet20 [12] on Cifar10 as a benchmark that is commonly used in the literature [15]. In addition, we trained ResNet18 [12] on a Steelpatch defect detection dataset [39] to simulate a potential industrial use-case. All hyperparameters for training and fine-tuning have been determined empirically. First, baseline models were trained for both scenarios. The Cifar10 model was trained from scratch, with randomly initialized weights, for 200 epochs with a batch size of 128 and start learning rate of 0.1. The Steelpatch model was started form a pre-trained ImageNet [9] state and fine-tuned for 100 epochs with a batch size of 32 and start learning rate of 0.001. Both models were trained with SGD as optimizer with weight decay of 0.0001 and momentum of 0.9. The learning rate was decayed by a factor of 10 after 50% and 75% of epochs. The baseline models were then compressed and fine-tuned for 100 (Cifar10) and 30 (Steelpatch) epochs respectively, with a start learning rate of 0.01 and 0.001 respectively, and a decay of factor 10 at the midpoint. For each model we report the validation accuracy and f1 score as average over the final 5 epochs in Table I.

TABLE I: Model training results

Architecture	Compression	Parameters	Accuracy	f1
ResNet20	Baseline	272K	92.02%	91.98%
	Prune 12.5%	208K	91.3%	91.27%
	Prune 25%	153K	90.59%	90.25%
	Tucker VBMF	25K	79.35%	79.36%
ResNet18	Baseline	11.18M	89.78%	86.95%
	Prune 12.5%	8.56M	89.28%	86.12%
	Prune 25%	6.29M	88.91%	85.38%
	Tucker VBMF	2.17M	89.47%	87.15%

Basic methods of pruning and tensor decomposition were used to compress the baseline models. The main goal of this work is to investigate the real-world potential of model compression and analyze the discrepancy between theoretic and observed speedup. Optimizing the accuracy-compression-trade-off is considered out of scope. Structural filter-level pruning as in [20] was performed based on the l2-norm of weights. All convolutional layers were pruned proportionally by removing $x\%$ of each layer’s filters at the same time. Tensor decomposition was performed using the Tucker decomposition [16] on every convolutional layer. The target ranks of the intermediate layers were automatically determined with VBMF [32]. Table I shows that even with these simple methods, it is possible to reduce the model parameter count considerably with drops in accuracy and f1 below 1%. Using the Tucker-VBMF method on the small scale ResNet20 does, however, lead to a significant loss in accuracy and f1-score of more than 10%.

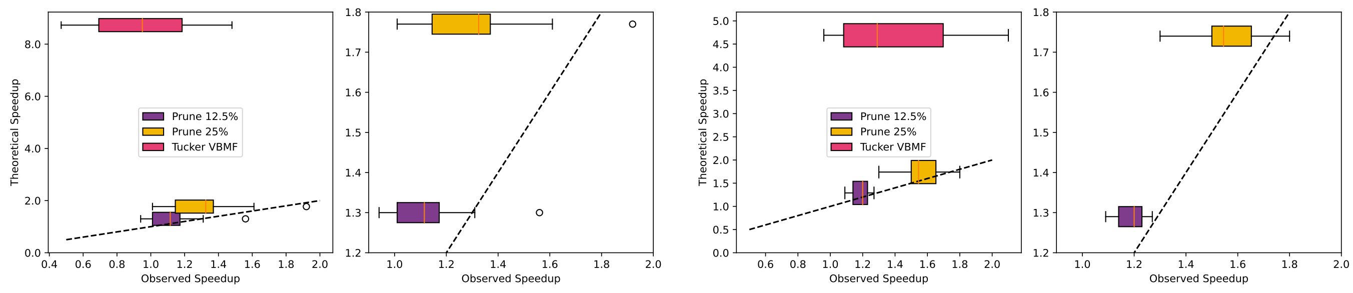
The Profiler, introduced in Section III, was used to profile each baseline and compressed model on the various hardware setups mentioned earlier in this Section. Profiles were created for two batch sizes each, 1 and the respective training batch size. The results of profiling are summarized in Table II for ResNet20 on Cifar10 and in Table III for ResNet18 on Steelpatch. Both tables indicate compression ratios and speedup

TABLE II: Compression ratios and speedup for ResNet20 [12] on Cifar10 [40]. Higher is better, below 1.0 indicates bigger/slower model.

Batch Size	Method	Memory Compression	Theoretical Speedup	Observed Speedup: Inference / Train			
				Quadro T2000	Jetson Nano	Intel i7	Intel i7 Single Thread
1	Prune 12.5%	1.18	1.30	1.01 / 1.17	1.01 / 0.96	1.24 / 1.56	1.11 / 1.31
	Prune 25%	1.42	1.77	1.01 / 1.19	1.06 / 1.01	1.56 / 1.92	1.30 / 1.61
	Tucker VBMF	1.06	8.73	0.51 / 0.52	0.51 / 0.47	0.91 / 1.01	1.18 / 1.18
128	Prune 12.5%	1.14	1.30	1.13 / 1.18	1.06 / 1.17	1.01 / 1.12	0.94 / 1.08
	Prune 25%	1.33	1.77	1.36 / 1.37	1.13 / 1.26	1.36 / 1.37	1.15 / 1.35
	Tucker VBMF	0.82	8.73	1.40 / 1.27	0.75 / 0.96	0.94 / 0.91	1.48 / 1.20

TABLE III: Compression ratios and speedup for ResNet18 [12] on Steelpatch [39]. Higher is better, below 1.0 indicates bigger/slower model.

Batch Size	Method	Memory Compression	Theoretical Speedup	Observed Speedup: Inference / Train			
				Quadro T2000	Jetson Nano	Intel i7	Intel i7 Single Thread
1	Prune 12.5%	1.22	1.29	1.23 / 1.19	1.14 / 1.19	1.09 / 1.21	1.23 / 1.22
	Prune 25%	1.53	1.74	1.54 / 1.50	1.54 / 1.56	1.65 / 1.80	1.60 / 1.66
	Tucker VBMF	1.37	4.69	1.06 / 0.90	1.66 / 1.03	1.19 / 1.28	2.10 / 1.83
32	Prune 12.5%	1.15	1.29	1.23 / 1.14	1.14 / 1.12	1.15 / 1.27	1.26 / 1.23
	Prune 25%	1.34	1.74	1.50 / 1.34	1.31 / 1.31	1.54 / 1.70	1.60 / 1.67
	Tucker VBMF	0.78	4.69	1.30 / 1.00	1.19 / 1.09	1.41 / 1.33	2.05 / 1.81



(a) ResNet20 on Cifar10

(b) ResNet18 on Steelpatch

Fig. 1: Distribution of observed speedup compared to theoretical speedup. Dashed line indicates equal compression.

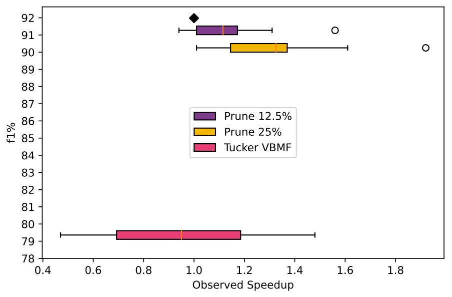
when compared to the respective baseline model. Higher ratios mean a higher reduction of the respective resource (e.g. memory or computation time). Ratios below 1.0 mean that the compressed model actually required more resources than the baseline. Observed speedup is indicated for both inference only and training.

Pruning reduces the memory consumption significantly across all experiments. For tensor decomposition, the memory reduction is lower as the parameter counts in Table I would suggest. With high batch size, the compressed model even requires considerably more memory as the baseline. This is a consequence of splitting each convolutional layer into 3 new layers. The indicated memory value includes weight tensors as well as activation tensors, i.e. layer outputs. By default, PyTorch, stores the activation tensors until the end of the training iteration. With rising batch size, the total memory consumption is dominated by these intermediate tensors. Hence, tripling the number of layers increases the necessary total memory significantly.

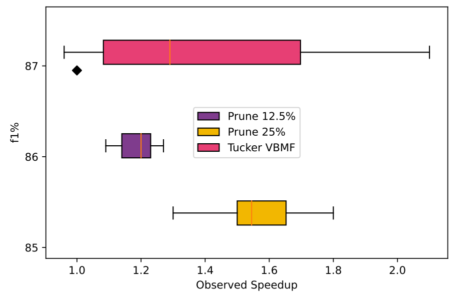
Fig. 1 plots the distribution of observed speedups against the respective theoretical estimation. The boxplots include both inference- and training latencies as well as both batch sizes. A diagonal line indicates equal theoretical and observed speedup. FLOP estimation predicts high compression in computational complexity, especially for Tucker VBMF, with a theoretical speedup of 4.69 on ResNet18 and even 8.73 on ResNet20. However, real world observed speedup is significantly lower in most test cases. This can be well observed on Fig. 1, where most data points are situated above the equal speedup line. Only a few outliers from running pruned models on CPU are faster than estimated by FLOPs. The discrepancy is lower in general for pruning, but still exceeding 0.15 most of the time.

Our results show that FLOP estimation is very unreliable for tucker decomposition with frequent discrepancies well above 3.0. The discrepancies for decomposition are in general higher on GPU devices and for a batch size of 1. This can be again explained by the layer splitting. In theory, the new layers have a lower total computation cost, however, the individual layers can only be executed in sequence. The convolution operation itself, that is needed for an individual layer, is well suited for parallel execution. Hence, the lower computation cost of the new layers is only advantageous if the original layer exceeds the parallel execution capabilities of the underlying hardware and is detrimental if the hardware is partially idling. This is confirmed by the lower discrepancy on CPU, especially in single-thread mode.

Fig. 2 plots the distribution of observed speedups against the respective model f1-score. The boxplots include both inference



(a) ResNet20 on Cifar10



(b) ResNet18 on Steelpatch

Fig. 2: Distribution of observed speedup compared to model f1-score. Black mark indicates baseline model

and training latencies as well as both batch sizes. The plot illustrates that considerable latency reduction can be achieved with negligible loss in robustness, even when using simplistic compression methods. The results also show that the variance among hardware is greater for tensor decomposition and when pruning more filters. This further proofs that practitioners need to consider the target hardware to estimate the potential of compression for a specific use-case.

V. Conclusion

Data-driven applications on the edge are becoming increasingly important in many domains, such as industry, space and surveillance. At the same time, convolutional neural networks (CNNs) are becoming deeper and wider which leads to increased computational complexity and hardware requirements. Compressing CNNs promises to produce efficient models with negligible loss of robustness. However, compression is rarely proven under real world constraints. This work provides the tools to experiment with structured pruning and tensor decomposition and profile the compression potential on target hardware. We show that a significant discrepancy exists between observable speedup and the commonly used FLOP-based estimation. Nonetheless, neural network compression has great potential. Under the right circumstances, latency can be halved with negligible loss of accuracy even when using basic compression methods. In our experiments, pruning provided more consistent compression than tensor decomposition. This work provides practitioners with valuable insights on the effect of compression for real world applications. Future work could extend our research on a broader spectrum of model architectures, datasets and compression methods.

References

[1] H. Li, K. Ota, and M. Dong, “Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing,” vol. 32, no. 1, pp. 96–101.

[2] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, “A survey of deep learning techniques for autonomous driving,” vol. 37, no. 3, pp. 362–386.

[3] E. N. Malamas, E. G. Petrakis, M. Zervakis, L. Petit, and J.-D. Legat, “A survey on industrial vision systems, applications and tools,” vol. 21, no. 2, pp. 171–188.

[4] G. Furano, G. Meoni, A. Dunne, D. Moloney, V. Ferlet-Cavrois, A. Tavoularis, J. Byrne, L. Buckley, M. Psarakis, K.-O. Voss, and L. Fanucci, “Towards the Use of Artificial Intelligence on the Edge in Space Systems: Challenges and Opportunities,” vol. 35, no. 12, pp. 44–56.

[5] M. Valera and S. Velastin, “Intelligent distributed surveillance systems: A review,” vol. 152, no. 2, p. 192.

[6] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, “Moving Deep Learning to the Edge,” vol. 13, no. 5, p. 125.

[7] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “A Survey of Model Compression and Acceleration for Deep Neural Networks.” [Online]. Available: http://arxiv.org/abs/1710.09282

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” vol. 115, no. 3, pp. 211–252.

[10] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: Common Objects in Context,” in *Computer Vision – ECCV 2014*, ser. Lecture Notes in Computer Science, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Springer International Publishing, vol. 8693, pp. 740–755.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” vol. 86, no. 11, pp. 2278–2324, Nov./1998.

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 770–778.

[13] K. Simonyan and A. Zisserman, “Very Deep Convo-

lutional Networks for Large-Scale Image Recognition." [Online]. Available: http://arxiv.org/abs/1409.1556

[14] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 1–9.

[15] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the State of Neural Network Pruning?" [Online]. Available: http://arxiv.org/abs/2003.03033

[16] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications." [Online]. Available: http://arxiv.org/abs/1511.06530

[17] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference." [Online]. Available: http://arxiv.org/abs/2103.13630

[18] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge Distillation: A Survey." [Online]. Available: http://arxiv.org/abs/2006.05525

[19] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Network," vol. 28, pp. 1135–1143.

[20] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets." [Online]. Available: http://arxiv.org/abs/1608.08710

[21] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper." [Online]. Available: http://arxiv.org/abs/1806.08342

[22] M. Yin, Y. Sui, S. Liao, and B. Yuan, "Towards Efficient Tensor Decomposition-Based DNN Model Compression With Optimization Framework," pp. 10 674–10 683.

[23] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning Important Features Through Propagating Activation Differences," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, pp. 3145–3153. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3306006

[24] A. Chattopadhay, A. Sarkar, P. Howlader, and V. N. Balasubramanian, "Grad-CAM++: Generalized Gradient-Based Visual Explanations for Deep Convolutional Networks," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, pp. 839–847.

[25] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," p. 8.

[26] H. Tanaka, D. Kunin, D. L. K. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow." [Online]. Available: http://arxiv.org/abs/2006.05467

[27] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," pp. 1389–1397.

[28] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, pp. 5068–5076.

[29] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," vol. 51, no. 3, pp. 455–500.

[30] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition." [Online]. Available: http://arxiv.org/abs/1412.6553

[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." [Online]. Available: http://arxiv.org/abs/1704.04861

[32] S. Nakajima, M. Sugiyama, S. D. Babacan, and R. Tomioka, "Global Analytic Solution of Fully-observed Variational Bayesian Matrix Factorization," p. 37.

[33] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training Quantized Nets: A Deeper Understanding," p. 11.

[34] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training Deep Nets with Sublinear Memory Cost." [Online]. Available: http://arxiv.org/abs/1604.06174

[35] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks," pp. 41–53.

[36] H. Vanholder, "EFFICIENT INFERENCE WITH TENSORRT," p. 24.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[38] H. He. The State of Machine Learning Frameworks in 2019. The Gradient. [Online]. Available: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

[39] J. Lorentz, T. Hartmann, A. Moawad, F. Fouquet, and D. Aouada, "Explaining Defect Detection with Saliency Maps," in *Advances and Trends in Artificial Intelligence. From Theory to Practice*, ser. Lecture Notes in Computer Science, H. Fujita, A. Selamat, J. C.-W. Lin, and M. Ali, Eds. Springer International Publishing, vol. 12799, pp. 506–518.

[40] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," p. 60.