Lowering PyTorch's Memory Consumption for Selective Differentiation

Samarth Bhatia
IIT Delhi
samarth.bhatia23@alumni.iitd.ac.in

Felix Dangel
Vector Institute
fdangel@vectorinstitute.ai

Abstract

Memory is a limiting resource for many deep learning tasks. Beside the neural network weights, one main memory consumer is the computation graph built up by automatic differentiation (AD) for backpropagation. We observe that PyTorch's current AD implementation neglects information about parameter differentiability when storing the computation graph. This information is useful though to reduce memory whenever gradients are requested for a parameter subset, as is the case in many modern fine-tuning tasks. Specifically, inputs to layers that act linearly in their parameters (dense, convolution, or normalization layers) can be discarded whenever the parameters are marked as non-differentiable. We provide a drop-in, differentiability-agnostic implementation of such layers and demonstrate its ability to reduce memory without affecting run time[1].

1. Introduction & Motivation

The success of many deep learning applications is driven by scaling computational resources [32]. One important resource is GPU memory, specifically on low- and mid-end GPUs which usually offer between 6 to 16 GiB. Therefore, down-scaling the computational demands of deep learning is an important objective to widen its accessibility to researchers and practitioners with fewer hardware resources.

Two major memory consumers are the neural network weights, and the computation graph stored by the automatic differentiation (AD) engine. There exist various approaches to reduce them; e.g., the parameters can be compressed with low-precision data types (quantization [16, 19, 21]) or sparsified [7, 12], and the computation graph can be off-loaded to CPU [26], compressed [3], randomized [22], or partially recorded and re-computed (gradient checkpointing [5, 10]).

Here, we consider a special AD scenario we call *selective differentiation* where gradients are requested only for a subset of the computation graph leafs (the neural network inputs and parameters). This approach has gained a lot of practical relevance in the era of large foundation models. For instance, fine-tuning techniques for pre-trained models rely on training only a subset of layers [18, 36], or extend some with parameter-efficient adapters [e.g. LoRA 15] which are then trained, keeping the original weights fixed. But it is also common in other applications such as generating adversarial examples [9] and neural style transfer [8] which optimize the input to a network with frozen weights.

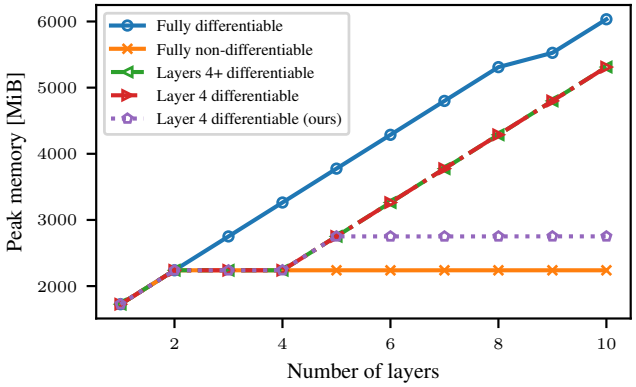

Figure 1. *PyTorch's AD is not agnostic to parameter differentiability.* We consider a deep CNN that consists entirely of size-preserving convolutions and measure the forward pass's peak memory when processing a mini-batch of size (256, 8, 256, 256), requiring 512 MiB storage. Memory increases linearly in the number of layers when all parameters are marked differentiable and remains constant when all parameters are marked non-differentiable. Surprisingly, *when only one layer's parameters are marked as differentiable the memory increases as if all subsequent parameters were marked differentiable*. Our drop-in solution stores layer inputs depending on the layer parameter's differentiability and reduces memory compared to the current PyTorch implementation.

We find that PyTorch's [23] current AD allows for additional, simple, optimizations to further reduce memory consumption in the context of selective differentiation:

1. We observe that PyTorch's AD neglects the differentiability of layer parameters when storing the computation graph (Figure 1). This information is useful though as it allows discarding inputs to linear layers whose parameters are marked as non-differentiable.
2. We provide a drop-in implementation of various lay-

[1] Code: github.com/plutonium-239/memsave_torch.

ers that is agnostic to the parameter differentiability and demonstrate on various convolutional neural networks (CNNs) that it lowers on the default implementation's memory footprint without increasing run time.

This easy-to-use insight benefits many tasks with selective differentiation and enables them to scale further. We hope it will stimulate further research into AD optimizations.

2. Selective Differentiation in PyTorch

PyTorch users can specify whether they want to compute gradients w.r.t. a tensor through its `requires_grad` attribute, which is dominantly inherited by child tensors: If any input (parent) to an operation is marked differentiable, the output (child) will also be differentiable and require the computation graph to be stored for backpropagation. As we will see, PyTorch does not check the differentiability of all parents, but rather starts storing the computation graph as if all parents were differentiable once it encounters a differentiable input. In the following example, we demonstrate this behavior, and how it misses out on possible optimizations for linear operations with non-differentiable parameters.

Experimental procedure: We use PyTorch 2.2.1 and measure the forward pass's peak memory of different neural networks as a proxy for the computation graph size stored by the AD engine. We chose peak memory as it is a reasonable proxy for the computation graph size and the relevant metric for causing out-of-memory errors in practise. On CPU, we use the `memory_profiler` library [24]. On GPU, we rely on `torch.cuda.max_memory_allocated()`, the maximum memory allocated by CUDA. Each measurement is performed in a separate Python session to avoid memory (de)allocation leaks between consecutive runs.

2.1. A Simple Example

We start with a synthetic example to probe the internals of PyTorch's AD that are responsible for identifying and storing the computation graph during a forward pass (summarized in Figure 2). We consider a deep CNN consisting entirely of convolutions without bias. Each convolution preserves its input size (kernel size 3, unit padding and stride) and number of channels and we vary its depth as well as the parameter differentiability. As input, we choose a mini-batch of size (256, 8, 256, 256). Storing this tensor, and each intermediate output generated by a layer, requires 512 MiB in single precision. Memory consumed by the convolution kernels of shape (8,8,3,3) is negligible compared to the hidden features. Figure 1 summarizes our findings.

First, we investigate the computation graph size when marking all or no parameters as differentiable. When all parameters are differentiable, all layer inputs must be stored to compute gradients. Consequently, we observe a linear relation with a slope corresponding to the 512 MiB consumed by each intermediate input (Figure 1, *fully differentiable*). When no parameter is differentiable, the memory consumption flattens after more than two layers (Figure 1, *fully non-differentiable*). This is because during a forward pass both input and output tensors of a layer are allocated in memory, in addition to the network's input. Hence, at most two tensors are in memory at a time for a single layer, while at most three are allocated for two or more layers. Next, we observe the memory consumption in the context of selective differentiation. We consider two scenarios: In the first, all parameters after and including the fourth layer are differentiable, hence all layer inputs after the third layer must be stored in memory. In the second, only the fourth layer is differentiable and therefore only that layer's input is necessary to compute gradients. However, we observe that both scenarios exhibit the *same* memory footprint (Figure 1, *layers 4+ differentiable & layer 4 differentiable*)!

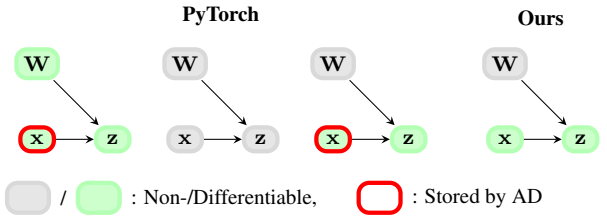

Figure 2. *PyTorch's current behaviour of storing the computation graph.* We illustrate this on a linear layer $\mathbf{z} = \mathbf{W}\mathbf{x}$. PyTorch stores the layer input whenever it is differentiable, although this is not necessary if the weight does not require gradients. Our approach uses this information to discard the layer input if possible.

We conclude that PyTorch stores a layer input with `requires_grad = True` although the layer's parameters might be non-differentiable, and therefore not require it to be stored. This information can be useful to reduce the information stored for backpropagation, as we show with our implementation (Figure 1, *layer 4 differentiable (ours)*).

2.2. Implementation Details

Our implementation of memory-saving layers is straightforward and does not require low-level code. For each layer we create a new `torch.autograd.Function` AD primitive, and its associated `torch.nn.Module` layer. The primitive uses the same forward and backward routines as the original operation (from `torch.nn.functional` and `torch.ops.aten`), but has additional logic for the information that is saved to the AD tape, which we describe below. Hence, our implementation shares the performance of PyTorch's. We also provide a converter that replaces supported layers of a net with our memory-saving equivalents.

2.2.1 Linear Layers

Consider a dense layer $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$ with vector input $\mathbf{x}$ and output $\mathbf{z}$, weight matrix $\mathbf{W}$, and bias vector $\mathbf{b}$. Its input Jacobian depends on $\mathbf{W}$, the weight Jacobian on $\mathbf{x}$,

Case	Memory [GiB] All	Input	Conv	Norm
(**GPU**) Default ResNet-101	**7.82 (1.00)**	7.82 (1.00)	7.82 (1.00)	7.82 (1.00)
+ swap Convolution	**7.82 (1.00)**	7.78 (0.99)	7.82 (1.00)	7.78 (0.99)
+ swap BatchNorm, ReLU	8.75 (1.12)	**1.61 (0.21)**	**4.84 (0.62)**	**5.44 (0.70)**
(**CPU**) Default ResNet-101	**8.24 (1.00)**	8.24 (1.00)	8.24 (1.00)	8.26 (1.00)
+ swap Convolution	8.35 (1.01)	8.30 (1.01)	8.33 (1.01)	8.26 (1.00)
+ swap BatchNorm, ReLU	9.37 (1.14)	**2.52 (0.31)**	**5.53 (0.67)**	**6.37 (0.77)**
	Time [s]			
(**GPU**) Default ResNet-101	1.55 (1.00)	1.39 (0.90)	1.57 (1.01)	1.40 (0.90)
+ swap Convolution	1.50 (0.97)	1.37 (0.88)	1.50 (0.97)	1.36 (0.88)
+ swap BatchNorm, ReLU	1.46 (0.94)	1.32 (0.85)	1.45 (0.94)	1.30 (0.84)
(**CPU**) Default ResNet-101	22.78 (1.00)	16.81 (0.78)	22.84 (1.00)	16.66 (0.73)
+ swap Convolution	23.05 (1.01)	17.08 (0.75)	23.14 (1.02)	16.90 (0.74)
+ swap BatchNorm, ReLU	25.34 (1.11)	16.79 (0.74)	23.68 (1.04)	17.57 (0.77)

Table 1. *Peak memory and run time comparison between PyTorch and our memory-saving layers* on ResNet-101. Normalized values are relative to 'All' with PyTorch's default layers.

`Bottleneck`	(64, 56, 56)
— `Conv2d`	(64, 56, 56)
— `BatchNorm2d`	(64, 56, 56)
— `ReLU`	(64, 56, 56)
— `Conv2d`	(64, 56, 56)
— `BatchNorm2d`	(64, 56, 56)
— `ReLU`	(64, 56, 56)
— `Conv2d`	(64, 56, 56)
— `BatchNorm2d`	(256, 56, 56)
— `Sequential`	(64, 56, 56)
—— `Conv2d`	(64, 56, 56)
—— `BatchNorm2d`	(256, 56, 56)
— `ReLU`	(256, 56, 56)

Table 2. *Details of one residual block of ResNet101.* Shapes of layer inputs (excluding the batch sizes) are shown on the right. The output of a ReLU layer is stored by both the activation layer itself, and the convolution layer it feeds into.

and the bias Jacobian has no dependency [e.g. 6, Chapter 2.3]. During the forward pass, we check the differentiability of $\mathbf{W}, \mathbf{x}$, and only store tensors required by the Jacobians that will be applied during backprogation. The same dependency pattern holds for other layers that process their inputs linearly w.r.t. their weight and add a bias term, such as convolution [see 2] and normalization [1, 17, 33, 34] layers. We implement them in exactly the same fashion.

2.2.2 Interaction with Other Layers

Real neural nets contain additional layers that are interleaved with linear layers: activations, dropout, pooling, etc. The information stored by such layers may overlap with the input to a linear layer, e.g. if an activation layer feeds into a convolution. Depending on the implementation details, our described optimizations might not apply in such cases because the linear layer's input tensor could still be stored by the preceding activation. We encountered this effect for ReLU layers in real-world CNNs (Section 3), but were able to overcome it using a customized ReLU implementation that saves a boolean mask of the output instead. As PyTorch currently only supports 1-byte booleans, this leads to a 4x reduction of the stored tensor's size, which could be further reduced to 32x with a 1-bit boolean implementation (which is in the works).

This underlines an important challenge for saving memory in selective differentiation whenever multiple layers use the same tensor for backpropagation. Inputs to linear layers, even if their parameters are non-differentiable, might still be stored by neighboring layers. We believe it should often be possible to resolve this scenario—albeit through careful implementation—e.g. whenever the Jacobian can be implemented by either storing the layer input or output, as is the case for various activation functions, and pooling layers.

3. Real-World Examples

Here we measure the effect of our memory-saving layers on real-world CNNs. We consider four different scenarios:

All: All network parameters are differentiable. This serves as reference to establish similar performance of our layers in the absence of selective differentiation.

Input: Only the input to the neural net is differentiable. This situation resembles constructing adversarial examples [9] or style-transfers [8] by optimizing noisy inputs.

Conv: Only convolution layers are differentiable. This situation is similar to surgical fine-tuning [18], which splits a network into different blocks, each containing a subset of layers, which are then trained one at a time.

Norm: Only normalization layers are differentiable, resembling layer norm fine-tuning in LLMs [36].

We report GPU results on an NVIDIA RTX A6000 with 48 GiB of VRAM, and CPU results for an Apple M2 with 16 GiB of RAM. All networks are fed inputs of size (64, 3, 224, 224) and we measure according to the procedure described in Section 2. For object detection models, the batch size is 4 and 2 boxes are predicted per input image.

3.1. ResNet-101

For ResNet-101, we take a detailed look at our memory-saving layer's effects. It is a decently powerful, modern model and frequently used as a backbone for other architectures such as CLIP [25] and LAVA [11]. It contains dense, convolution, batch normalization, ReLU, and max/average pooling layers. Table 1 summarizes our findings.

PyTorch is unaware of selective differentiation: In the upper part of Table 1, we see that the default PyTorch im-

Case	Memory [GiB] All	Input	Conv	Norm	Time [s] All	Input	Conv	Norm
DeepLabv3 (RN101) [4]	**22.78 (1.00)**	22.78 (1.00)	22.78 (1.00)	22.78 (1.00)	2.00 (1.00)	1.76 (0.88)	2.00 (1.00)	1.76 (0.88)
+ MemSave	24.90 (1.09)	**4.28 (0.17)**	**13.72 (0.55)**	**15.17 (0.61)**	1.97 (0.99)	1.76 (0.88)	1.99 (1.00)	1.77 (0.89)
EfficientNetv2-L [30, 31]	**26.83 (1.00)**	26.83 (1.00)	26.83 (1.00)	26.83 (1.00)	1.79 (1.00)	1.59 (0.89)	1.79 (1.00)	1.57 (0.88)
+ MemSave	**26.83 (1.00)**	**10.45 (0.39)**	**18.62 (0.69)**	**18.64 (0.69)**	1.66 (0.93)	1.45 (0.87)	1.63 (0.91)	1.45 (0.87)
FCN (RN101) [20]	**22.19 (1.00)**	22.19 (1.00)	22.19 (1.00)	22.19 (1.00)	1.94 (1.00)	1.72 (0.89)	1.94 (1.00)	1.71 (0.88)
+ MemSave	24.41 (1.10)	**4.26 (0.17)**	**13.42 (0.55)**	**15.14 (0.62)**	1.90 (0.98)	1.70 (0.88)	1.91 (0.98)	1.68 (0.87)
MobileNetv3-L [14, 28]	**2.80 (1.00)**	2.80 (1.00)	2.80 (1.00)	2.80 (1.00)	1.46 (1.00)	1.35 (0.92)	1.48 (1.01)	1.35 (0.92)
+ MemSave	2.94 (1.05)	**0.87 (0.30)**	**1.90 (0.65)**	**1.91 (0.65)**	1.44 (0.99)	1.30 (0.90)	1.44 (0.99)	1.31 (0.90)
ResNeXt101-64x4d [35]	**15.18 (1.00)**	15.18 (1.00)	15.18 (1.00)	15.18 (1.00)	1.71 (1.00)	1.53 (0.89)	1.68 (0.98)	1.53 (0.89)
+ MemSave	16.75 (1.10)	**2.46 (0.15)**	**9.28 (0.55)**	**9.89 (0.59)**	1.59 (0.93)	1.44 (0.84)	1.61 (0.94)	1.43 (0.84)
VGG-16 [29]	4.93 (1.00)	4.93 (1.00)	4.93 (1.00)	N/A	1.50 (1.00)	1.33 (0.89)	1.49 (0.99)	N/A
+ MemSave	**4.30 (0.87)**	**2.60 (0.60)**	**4.30 (0.87)**	N/A	1.45 (0.97)	1.35 (0.90)	1.45 (0.97)	N/A
Faster-RCNN (RN101) [27]	**6.62 (1.00)**	6.56 (0.99)	6.57 (0.99)	6.53 (0.99)	1.58 (1.00)	1.47 (0.93)	1.61 (1.02)	1.46 (0.92)
+ MemSave	7.21 (1.09)	**1.90 (0.26)**	**4.55 (0.63)**	**4.43 (0.61)**	1.51 (0.96)	1.39 (0.88)	1.54 (0.97)	1.38 (0.87)
SSDLite (MobileNetv3-L) [28]	**0.53 (1.00)**	0.53 (1.00)	0.53 (1.00)	0.53 (1.00)	1.54 (1.00)	1.39 (0.90)	1.53 (0.99)	1.40 (0.91)
+ MemSave	0.55 (1.04)	**0.26 (0.48)**	**0.40 (0.73)**	**0.41 (0.74)**	1.48 (0.96)	1.35 (0.88)	1.50 (0.97)	1.34 (0.88)

Table 3. *Peak memory and run time comparison between PyTorch and our memory-saving layers* on GPU on different scenarios and CNNs.

plementation uses the same amount of memory for any scenario. In fact, marking only the input to a neural net as differentiable consumes as much memory as marking all parameters, although the former does not require storing the inputs to linear layers. This confirms the findings on the toy model from Section 2.

Layer interactions diminish memory savings: To gradually investigate the effect of our layers, we only swap out the convolutions; *without* observing any effects. At first, this seems counter-intuitive. However, a closer look at the architecture (Table 2) reveals that the convolutions are preceded by ReLU activations which unconditionally store their outputs and render our convolution layer's optimizations ineffective as described in Section 2.2.2.

After swapping out ReLU with our mask-based custom implementation, we observe substantial memory savings. E.g., memory consumption for 'Input' dropped by a factor of almost 5x on GPU. Only when all parameters are differentiable, we see a slight increase in memory after swapping out ReLUs. This is to be expected as the mask stored by our custom implementation requires additional storage.

Run time remains unaffected: The bottom half of Table 1 shows that run time for a fixed case remains equal up to measurement noise. This confirms that our memory-saving layers share the default implementation's performance.

3.2. More results

To further solidify our findings, we now evaluate our layers on other popular and commonly used CNNs, including ResNet-18 [13], VGG-16 [29], ResNeXT101-64x4d [35], EfficientNetv2-L [30, 31], MobileNetv3-L [14, 28], FCN with a ResNet-101 backbone [20], DeepLabv3 with a ResNet-101 backbone [4], Faster-RCNN with a ResNet-50 backbone [27], and SSDLite with a MobileNetv3-L backbone [28]. Table 3 summarizes the comparison.

On this large repertoire of networks, we observe the same effects as on ResNet-101 from the previous section: Due to our customized ReLU implementation, memory is slightly higher in the absence of selective differentiation. Run time is unaffected by swapping in our layers, while the selective differentiation scenarios consistently show lower memory consumption with reduction factors up to 6x, underlining the usefulness of our approach in this context.

4. Conclusion

We have shown how to improve the memory consumption of PyTorch's automatic differentiation in the context of selective differentiation where gradients are only requested for a subset of variables—a common situation in modern fine-tuning tasks. To overcome this, we provide a drop-in implementation which takes into account the differentiability of all tensors when storing the computation graph. Empirically, we demonstrated the effectiveness of our approach to reduce memory in multiple selective differentiation cases without affecting run time.

Our method is easy to use, requiring only a single call to a converter function that replaces all supported layers with our equivalents. Next, we plan to apply it to parameter-efficient LLM fine-tuning or layer norm tuning, and other modern architectures like vision transformers. We expect this to require additional efforts due to the interaction with other transformer layers like softmax and GeLU.

Acknowledgements

Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring Vector Institute.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. 2016. 3

[2] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, 2006. 3

[3] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning (ICLR)*, 2021. 1

[4] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation, 2017. 4

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016. 1

[6] Felix Julius Dangel. Backpropagation beyond the gradient. 2023. 3

[7] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 1

[8] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 1, 3

[9] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015. 1, 3

[10] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008. 1

[11] Sumanth Gurram, David Chan, Andy Fang, and John Canny. LAVA: Language audio vision alignment for data-efficient video pre-training. In *First Workshop on Pre-training: Perspectives, Pitfalls, and Paths Forward at ICML 2022*, 2022. 3

[12] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems (NIPS)*, 1992. 1

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 4

[14] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le. Searching for mobilenetv3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324, Los Alamitos, CA, USA, 2019. IEEE Computer Society. 4

[15] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. 1

[16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research (JMLR)*, 2018. 1

[17] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015. 3

[18] Yoonho Lee, Annie S Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. Surgical fine-tuning improves adaptation to distribution shifts. In *The Eleventh International Conference on Learning Representations*, 2023. 1, 3

[19] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. 1

[20] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, Los Alamitos, CA, USA, 2015. IEEE Computer Society. 4

[21] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021. 1

[22] Deniz Oktay, Nick McGreivy, Joshua Aduol, Alex Beatson, and Ryan P Adams. Randomized automatic differentiation. In *International Conference on Learning Representations (ICLR)*, 2021. 1

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 1

[24] Fabian Pedregosa. memory_profiler: Monitor memory usage of python code. 2

[25] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, pages 8748–8763. PMLR, 2021. 3

[26] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. 1

[27] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural In-*

formation Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada, pages 91–99, 2015. 4

[28] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 4

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 4

[30] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 6105–6114. PMLR, 2019. 4

[31] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, pages 10096–10106. PMLR, 2021. 4

[32] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. The computational limits of deep learning, 2020. 1

[33] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. 2016. 3

[34] Yuxin Wu and Kaiming He. Group normalization. *International Journal of Computer Vision*, 2019. 3

[35] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995. IEEE Computer Society, 2017. 4

[36] Bingchen Zhao, Haoqin Tu, Chen Wei, Jieru Mei, and Cihang Xie. Tuning layernorm in attention: Towards efficient multi-modal LLM finetuning. In *International Conference on Learning Representations (ICLR)*, 2024. 1, 3