The Maximum-Memory Rule:
Reducing Memory in Neural Network Training

Ross Tooley
Department of Computer Science and Technology
University of Cambridge
rjt80@cam.ac.uk

Abstract

In this project I implement multiple techniques to reduce memory when training neural networks, and use them both individually and together on `WideResNet-50-2` and `Transformer`. I introduce the Maximum-Memory Rule, a novel rule which accurately models memory consumption, and I show how this places a theoretical limit the best-possible memory reduction that techniques like checkpointing and microbatching can achieve.

The source code for this project is available at `https://github.com/rosstooley/pytorch-low-mem-training`.

1 Introduction

Training deep neural networks is memory intensive, but past research has been able to reduce this memory consumption by exchanging it for increased computation time or reduced accuracy. This project builds directly on the work of Sohini et al. [6] who study the benefit of using multiple such techniques together on the same neural network, namely sparsity, quantisation, micro-batching and checkpointing. Sohoni et al. only analyse some of these techniques theoretically due to lack of implementation support in the major frameworks at their time of writing. But since then, the major frameworks have added implementation support for these techniques, so in this project I have reproduced their experiments with real implementations in PyTorch [2] to determine whether their theoretical results are true in practise.

In these experiments have led to the discovery that that the model used by Sohoni et al. to compute the total memory required during training does not equal the actual maximum memory measured during my experiments. In order to explain this difference, I have created a new theoretical model called the Maximum-Memory Rule. This rule imposes a limit on the best-possible memory reduction that can be achieved by some techniques such as checkpointing and microbatching. In the next section, I review the model used by Sohini et al., demonstrate how it differs from the real-world maximum memory, and then introduce the Maximum-Memory Rule.

I then implement memory-saving techniques in PyTorch and run a series of experiments which demonstrate how the techniques can be used individually and together to reduce the memory compared to a baseline model. I am able to explain the results of each technique, and I am able to demonstrate empirically that checkpointing and microbatching can hit the limits imposed by the Maximum-Memory Model. Finally, I conclude with a comparison of my results to those of Sohoni et al.

Preprint. Under review.

2 A precise model of maximum memory consumption

2.1 A review of the memory model by Sohini et al.

In the experiments by Sohini et al. [6] they do not actually implement their techniques because of lack of implementation support in the major ML frameworks. Instead they compute memory consumption using a static analysis tool. This tool models the memory consumption of training by splitting it into three types: *weights* includes all tensors used to specify the weights and biases of the neural network; *gradients* includes the tensors for accumulating the gradients of the weights until the weights are updated; *activations* includes the tensors which store the activations and their gradients. To compute the total memory, Sohini et al. compute the max of each type of memory and add them together.

Breaking the memory model down into these types is useful because some of the memory-saving techniques have a different effect on each type of memory, as I enumerate in Figure 2. Furthermore, many projects must resort to distributed training because they consume too much memory for a single machine. A popular solution is data-parallel training, which splits the data across multiple machines, thereby only reducing the size of *activations* on each machine, not the size of *weights* or *gradients*.

Lastly, it is important to confirm what is meant by 'total memory consumption'. I take it to mean the maximum memory allocated on the GPU at any point during training, because if the maximum memory were to exceed the memory available on the GPU then the project would have to be distributed to more machines.

2.2 Empirical evidence for a new memory model

To motivate the creation of a new model for predicting maximum memory, I first show an empirical example where the true maximum memory consumption during training is different to what Sohini et al. predict. In this project I use real implementations of these techniques and measure the real memory consumption using a GPU-memory profiler from Sicara [5]. This takes readings of the total memory allocated on the GPU and produces a memory profile for a single training batch which shows how the total memory allocated increases and decreases throughout the batch. I take the maximum point on this profile to be the maximum memory.

Figure 1 shows the profile of my baseline. This baseline is the `WideResNet-50-2` convolutional neural network [9], trained on a single batch of 256 `32x32` colour images. The graph shows that the maximum memory required to train this baseline is 1222MB, and the total time for one batch is 630ms. The horizontal lines show the predicted memory consumption according to the model by Sohoni et al. This model predicts that maximum memory is the sum of *weights*, *gradients* and *activations*, which is the top horizontal line on the graph, but this is clearly disproven by my profile which shows the true peak to be much less. So, a new model is required to account for this difference.

2.3 The Maximum-Memory Rule

The reason why the baseline in Figure 1 is less than the prediction by Sohoni et al. [6] can be explained by examining the periods during which each type of memory are used during a training batch. Unused tensors are quickly evicted from the GPU so the maximum memory allocated will not be the naive sum of all tensors used during a training batch. Before the forward pass, only the weights are stored; during the forward pass the activations are created and stored so that the current memory grows to *weights* + *activations* by the end of the forward pass; during the backward pass the activations are used to compute the gradients so that the total memory after the backward pass is *weights* + *gradients*.

The maximum memory is therefore found somewhere on the backward pass. Either this is at the beginning of the backward pass, when total memory equals *weights* + *activations*; or it is at the end of the backward pass when total memory equals *weights* + *gradients*; or it is somewhere in the middle of the backward pass when the gradients computed so far plus the activations yet to be computed are maximal. I call this the Maximum-Memory Rule, and it can be stated more mathematically as follows:

$$\texttt{maximum memory} = weights + max_{l \in L}(activations[:l] + gradients[l:]) \quad (1)$$

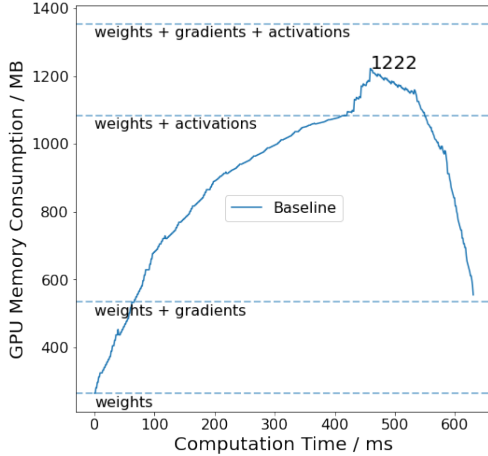

Figure 1: Memory profile of one training batch. Max memory is labelled. Horizontal lines show sums of each type of memory. Baseline is WideResNet-50-2 on 256 CIFAR-10 images.

Figure 2: How each technique changes each type of memory.

	Δ Memory		
Technique	Weight	Gradients	Activations
Checkpointing	—	—	↓
Mixed Precision	↑	↓	↓
FP16	↓	↓	↓
Microbatching	—	—	↓

where L is the set $\{0, \ldots, |L|-1\}$ of indexes of the layers of the network, $activations[:l]$ is the memory required for the activations of the first l layers, and $gradients[:l]$ is the memory required for the gradients of the last $|L|-l$ layers.

The Maximum-Memory Rule provides two bounds on the effectiveness of a memory-saving technique:

1. When maximum memory is at $l=0$ then the maximum memory is equal to $weights + gradients$ and there is no benefit in reducing *activations* any further.
2. When maximum memory is at $l=|L|-1$ then the maximum memory is equal to $weights + activations$ and there is no benefit in reducing *gradients* any further.

The first of these limits impacts memory-saving techniques such as checkpointing and microbatching which only reduce *activations*.

3 Implementing memory-saving techniques

In this section I evaluate the impact of the Maximum-Memory Rule on each memory-saving technique. I do this by analysing the theoretical reduction that each technique should provide (summarised in Figure 2) and whether it is impacted by the rule. I have also implemented these techniques in PyTorch [2] using its existing library support, and I have run experiments testing these techniques on benchmark models. In this section I evaluate the data from these experiments to learn how each technique changes memory consumption and computation speed, and how each one is affected by the Maximum-Memory Rule. Each of these experiments is conducted an NVIDIA P4 GPU.

The techniques I consider are gradient checkpointing, FP16, mixed precision and microbatching, each of which are summarised in their section. I do not investigate sparsity because this technique only saves memory on certain types of specialised hardware (not including a GPU) which are able to operate on compressed, sparse matrices in memory. Lastly, FP16 and mixed precision exchange memory consumption for training accuracy, but I have not been able to measure accuracy due to limited availability of GPUs, so this is out of scope for this project.

3.1 Gradient Checkpointing

Gradient Checkpointing is a memory-saving technique which reduces activation memory in exchange for increased computation time. Rather than storing all activations during the forward pass, this technique only stores the activations for some layers, known as checkpoints. Then, the backward pass

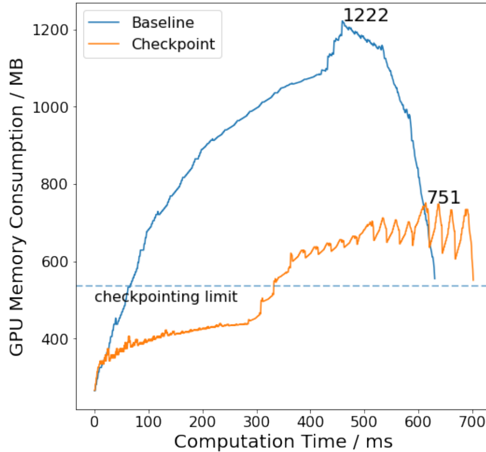

Figure 3: WideResNet-50-2 profile comparison with checkpointing. Checkpointing scheme is res-1 and reduces memory by 38.5%. Horizontal line shows greatest reduction theoretically achievable by checkpointing alone, 56.2%, according to the Maximum-Memory Rule.

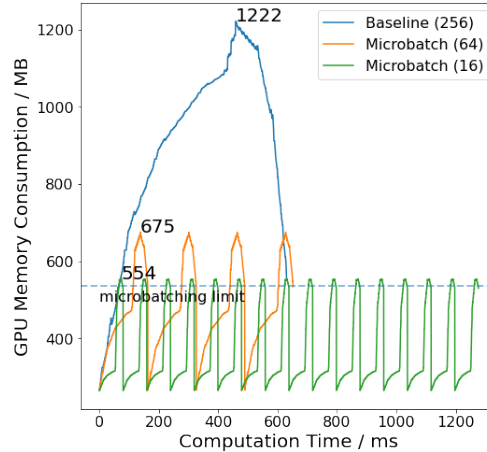

Figure 4: WideResNet-50-2 profile comparison with microbatching. Microbatches of size 64 and 16 save 44.8% and 54.7% memory respectively. Horizontal line shows greatest reduction theoretically achievable by microbatching alone, 56.2% according to the Maximum-Memory Rule

is replaced by multiple smaller forward-backward passes, each one on a different segment between two of the checkpoints. This reduces *activations*, because fewer activations are stored at once, but does not reduce *weights* or *gradients*. The Maximum-Memory Rule means that checkpointing has a limit which is reached when the maximum memory never exceeds $weights + gradients$. The ideal checkpointing scheme reaches this limit with the least impact on computation time.

I implement a checkpointing scheme for `WideResNet-50-2` and show its profile in Figure 3. This scheme is known as `res-1` and it checkpoints the output of each residual block in the network. The maximum memory consumption of the checkpointed network is 38.5% less than the baseline, whereas the theoretical limit (shown on the graph by the horizontal line) is a reduction of 56.2% over this baseline. This shows that `res-1` is a good, but not ideal, checkpointing scheme for this baseline. A stronger checkpointing scheme could save more memory, but may reach the limit and start wasting computation time without reducing memory. Beyond the limit, stronger checkpointing schemes are only useful when combined with techniques that also reduce *gradients*.

Figure 3 also shows that checkpointing with `res-1` takes 700ms, 11% longer than the baseline, which makes it clear that checkpointing exchanges memory for computation time. The PyTorch implementation of gradient checkpointing which I use is the `checkpoint` library [3]. To add this to the pre-built `WideResNet-50-2` model I simply override the implementation of the PyTorch `forward` function.

3.2 Quantisation

The tensors used to store weights, gradients and activations are ordinarily stored in single precision, meaning they use 32 bits per element. However, quantisation techniques exchange memory for model accuracy by using half-precision tensors instead, which use 16 bits per element. PyTorch enables two types of quantisation; FP16 sets all tensors to half precision, whereas mixed precision is a more nuanced technique created by NVIDIA and Baidu Research [1] which uses a mixture of 16-bit and 32-bit tensors to achieve much higher accuracy than FP16. Mixed precision casts all tensors to 16-bit but retains a 32-bit copy of the weights. It automatically scales the gradients to ensure they avoid underflow and overflow. Lastly, it updates the weights by casting the gradients to FP32 and summing the two.

FP16 halves all forms of memory, so the Maximum-Memory Rule predicts that maximum memory will be halved. Mixed precision halves *gradients* and *activations*, but it actually increases *weights* by

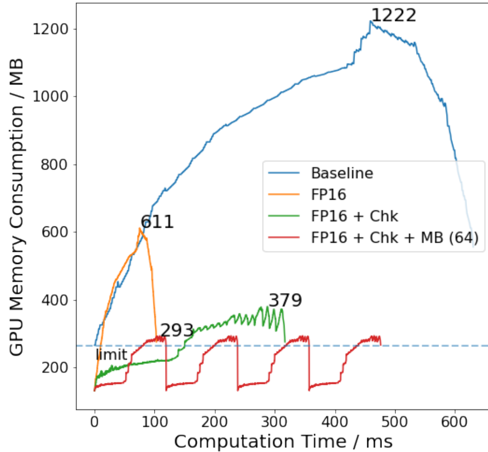

Figure 5: WideResNet-50-2 memory profiles for increasing combinations of different techniques, FP16, then res-1 checkpointing, then microbatches of size 64. Horizontal line shows maximum memory reduction according to the Maximum-Memory Rule.

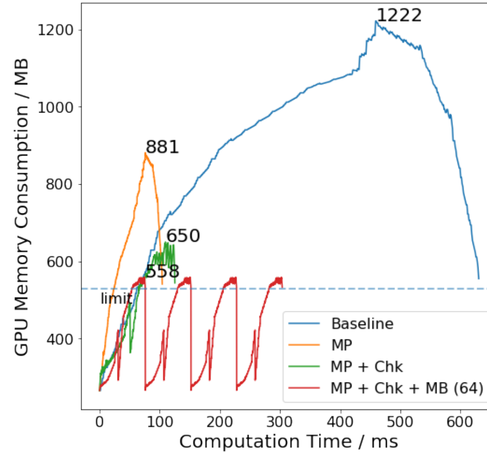

Figure 6: WideResNet-50-2 memory profiles for increasing combinations of techniques, mixed precision, then res-1 checkpointing, then microbatches of size 64. Horizontal line shows maximum memory reduction according to the Maximum-Memory Rule.

half because it stores both an 16-bit and a 32-bit copy of the weights. Therefore my memory model predicts that the profile of mixed precision will bigger than FP16 by the size of a 32-bit copy of the weights.

To demonstrate this empirically, I implement FP16 and mixed precision on `WideResNet-50-2` and show their memory profiles in Figures 5 and 6 respectively. The maximum memory consumption of FP16 is a 50% reduction over the baseline. The mixed precision only reduces it by 27.9%, but this difference is almost entirely accounted for by the addition of a 32-bit copy of the weights which consumes 21.8% of the baseline's memory. The graph also shows that FP16 and mixed precision significantly reduce the computation time (by 74% and 75% respectively), and this is because the NVIDIA P4 GPU has accelerated operations for FP16 and mixed precision computation.

Both of the methods combine nicely with checkpointing. They have the same effect on a model irrespective of whether that model has already had checkpointing applied to it or not. This is because both techniques reduce *gradients* and *activations* so neither of the bounds from the Maximum-Memory Rule are encountered. Although, mixed precision is least effective on use-cases where the weights are large in comparison to the activations (because it makes a 32-bit copy of the weights) such as when using large models on small batches, or when using a checkpointed model. This is also shown empirically in Figures 5 and 6. The combination of FP16 and checkpointing yields a 69.0% memory reduction on the baseline, and a 49.5% improvement versus checkpointing alone. The combination of mixed precision and checkpointing is a 46.8% reduction on the baseline, and a 13.4% improvement versus checkpointing alone.

I implement FP16 in PyTorch by simply casting every tensor to 16-bit using the built-in library function `to(float16)`. I add mixed precision by using the `amp` library [4].

3.3 Microbatching

Microbatching is a technique to reduce memory in exchange for increased computation time. It divides each minibatch into microbatches, does a forward and backward pass on each microbatch while accumulating the gradients but not updating the weights, then updates the weights when all the microbatches within one minibatch are complete. This technique reduces *activations* but keeps *weights* and *gradients* therefore the Maximum-Memory Rule predicts that it will conflict with checkpointing.

In Figure 4 I compare the profile of the baseline (which uses a minibatch of 256 images), to experiments using microbatches of size 64 and 16. These yield memory reductions of 44.8% and

54.7% respectively but incur computation-time overheads of 4% and 103%. The microbatch of size 16 has reached the limit of microbatching according to the Maximum-Memory Rule. The maximum memory of training never exceeds $weights + gradients$ therefore the size of the activations no longer has an effect on the maximum memory of training. This means that using such small microbatches is wasteful because they increase computation time without reducing memory. This insight is only possible after realising the Maximum-Memory Rule.

Furthermore, checkpointing has no effect when applied to the size-16 microbatch experiment. This is because checkpointing also only reduces activation memory and so the Maximum-Memory Rule prevents any further improvements being made. In general, checkpointing and microbatching can be used together effectively only if they do not hit the limit imposed by the Maximum-Memory Rule. Beyond this limit any further checkpointing or microbatching will have no effect. Also, since neither of them reduce the size of *weights* or *gradients* then they will suffer diminishing returns when used together as the size of the weights and the gradients start to dominate.

At time of writing, microbatching has no implementation in PyTorch, so I simulate microbatching using small minibatches which does not change the memory consumption of training. It does change the convergence properties of training, but this doesn't matter for this project because I am not measuring accuracy.

3.4 Summary

In this section I have demonstrated the consequences of the Maximum-Memory Rule on each type of memory-saving technique both theoretically and empirically. I have shown that both checkpointing and microbatching are limited by the rule, and beyond these limits they will only increase computation time without decreasing maximum memory. Conversely, I have showed that FP16 and mixed precision are not limited by the Maximum-Memory Rule because they both reduce both *gradients* and *activations*. FP16 will always halve the memory consumption of a model, whereas mixed precision also needs to store a 32-bit copy of the weights. This means that mixed precision becomes less effective on models with large weights and small activations, such as when using microbatching or checkpointing.

I have also shown how each technique affects computation time. Checkpointing and microbatching both increase computation time, whereas FP16 and mixed precision both reduce computation time because modern GPUs can accelerate 16-bit computation. Instead, FP16 and mixed precision exchange accuracy for memory, but measuring accuracy is beyond the scope of this project. Researchers can use the insights from this project to decide how to balance memory, computation time and accuracy.

4 Empirical comparison with results of Sohoni et al.

In this section I compare my results to the results by Sohoni et al. [6] by using multiple memory-saving techniques together on the same model and comparing the memory reduction achieved. I achieve far smaller memory reductions than claimed by Sohoni et al. and one possible reason for this could be that Sohoni et al. use a static analysis tool to predict their results and they have not factored the Maximum-Memory Rule into their predictions. Another reason is that I use slightly different models to Sohoni et al. My comparison is done using two models, `WideResNet-50-2` [9], and the vanilla `Transformer` [7], whereas Sohoni et al. use `WideResNet-28-2` [9] and the `DC-Transformer` [8]. Ideally they would be the same, but I have struggled to accurately re-create those models in the time available. Since the baselines are different, the percentage reductions are not directly comparable, but I am still able to draw insights. Figure 8 shows the numerical results side-by-side.

In the first experiment, on `WideResNet-50-2`, I apply both `res-1` checkpointing and microbatches of size 64, and one of FP16 and mixed precision. The profiles of these experiments are shown in Figures 5 and 6 and shown that memory consumption is reduced by 76.0% with FP16 and 54.3% with mixed precision. Both of these experiments are at the limit of memory reduction according to the Maximum-Memory Rule, which is 79.4% for FP16 and 56.2% for mixed precision (because mixed precision maintains an additional 32-bit copy of the weights). Both of these methods actually reduce computation time, although they do reduce accuracy.

My second experiment, on `Transformer`, shows that the Maximum-Memory Rule can be applied to different types of model. The `Transformer` baseline has a maximum memory consumption of

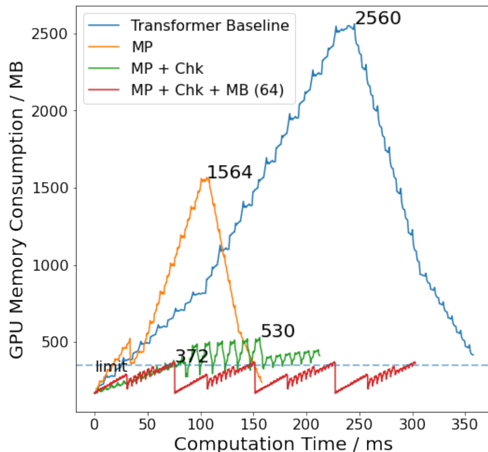

Figure 7: Transformer memory profiles for increasing combinations of techniques: mixed precision, then layer-1 checkpointing, then microbatches of size 64. Horizontal line shows maximum memory reduction according to the Maximum-Memory Rule.

Figure 8: Comparison with results by Sohoni et al.

Author	Techniques	Memory reduction
	WideResNet-28-2	
Sohoni	res-2-* checkpointing FP16 10 Microbatches of 10	97.0%
	WideResNet-50-2	
Tooley	res-1 checkpointing FP16 4 Microbatches of 64	76.0%
Tooley	res-1 checkpointing Mixed precision 4 Microbatches of 64	54.3%
	DC-Transformer	
Sohoni	layer-1 checkpointing FP16 16 Microbatches of 250	88.5%
	Transformer	
Tooley	layer-1 checkpointing Mixed precision 4 Microbatches of 64	85.5%

2560MB on a batch size of 256. The checkpointing scheme applied is `layer-1` which checkpoints the output of layer of the encoder and each layer of the decoder. Figure 7 shows the memory reduction when applying mixed precision, `layer-1` checkpointing and microbatches of size 64.[1] This achieves a memory reduction of 85.5% which is also the lower limit of memory consumption according to the Maximum-Memory Rule. Furthermore, this combination of techniques reduces computation time, in exchange for model accuracy.

Figure 8 compares my results to the results claimed by Sohoni et al., and firstly shows that I achieve far less memory reduction on `WideResNet`, even though the Maximum-Memory Rule states that I can do no better with these techniques. One clear difference between our experiments is that Sohoni et al. use far smaller microbatches and a far stronger checkpointing strategy. Due to the Maximum-Memory Rule, I have shown than even if I were to use as small microbatches and as small checkpoints, I would not be able to reduce memory consumption any further. Two explanations for the huge memory reduction by Sohini et al. are that `WideResNet-28-2` is a smaller model than `WideResNet-50-2` so the Maximum-Memory Rule limit may be lower, or possibly that their results are overstated because their static analysis tool is not factoring in the effect of the Maximum-Memory Rule.

Our results are much closer on the Transformer, primarily because Sohoni et al. use far bigger microbatches and a less aggressive checkpointing strategy. By comparison to my experiment, I predict that these techniques do not exceed the Maximum-Memory Rule limit and I am confident that these results would be shown to be true if they were actually implemented.

5 Conclusion

The objective of this project has been to reproduce the work of Sohoni et al. [6] and investigate how much memory can be saved when training a neural network using gradient checkpointing, microbatching, FP16 and mixed precision. The novel discovery of this project has been the Maximum-Memory Rule which states that the maximum memory required for training is not simply the sum of *weights*, *gradients* and *activations* because the GPU does not store all of these in memory at once. The

[1] FP16 not available for the Transformer because it contains some operations which cannot be casted.

Maximum-Memory Rule also states that if the either *gradients* or *activations* are sufficiently greater than the other, then reducing the smaller one will have zero effect on the total memory consumption. This imposes limits on the effectiveness of memory-saving techniques such as checkpointing and microbatching which only reduce *activations*.

Unlike Sohini et al., I have implemented each technique in PyTorch [2] and I have profiled memory consumption and computation time while training a model under each technique and combination of techniques. This data provides evidence of the limit imposed by the Maximum-Memory Rule; it shows that excessive checkpointing and excessive microbatching eventually stop reducing memory consumption and from then only increase computation time.

Finally, I have compared my results to those by Sohoni et al. despite using slightly different baselines. I was not able to achieve as much memory reduction as Sohini et al., however I am reassured by the Maximum-Memory Rule that I have reached the limit of memory reduction on my baseline with these techniques. I believe that the main contribution of this project to readers is its detailed exposition on how each technique affects memory consumption and computation, and why each technique does so. Readers will be able to use this project to understand what effect each technique will have on their own models, and which technique or set of techniques are best-suited for them.

References

[1] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[3] PyTorch. PyTorch checkpointing library. `https://pytorch.org/docs/stable/checkpoint.html`.

[4] PyTorch and NVIDIA. PyTorch automatic mixed-precision library. `https://pytorch.org/docs/stable/amp.html`.

[5] Sicara. Deep learning memory usage and pytorch optimization tricks. `https://github.com/quentinf00/article-memory-log`.

[6] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2019.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[8] Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*, 2019.

[9] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.