Exploring Memory and Compute in nanoGPT

Owen Fahey

Introduction and Experimental Setup

Several experiments were undertaken to explore how various hyperparameters and optimizations influence peak memory usage, runtime, and performance when training nanoGPT. Observing the effects of making these modifications at a small scale shines a light on the trade-offs involved in training large language models and provides useful insights for deciding how to train much larger models.

The runs were conducted on an A100 GPU on Google Colab using the works of Shakespeare as training data. The default model configuration was taken from the following nanoGPT repo:

`https://github.com/karpathy/nanoGPT/blob/master/config/train_shakespeare_char.py`. This configuration specifies a context size of 256 characters, 6 layers, 6 attention heads, an embedding dimension of 384, a batch size of 64, and a training duration of 5000 iterations.

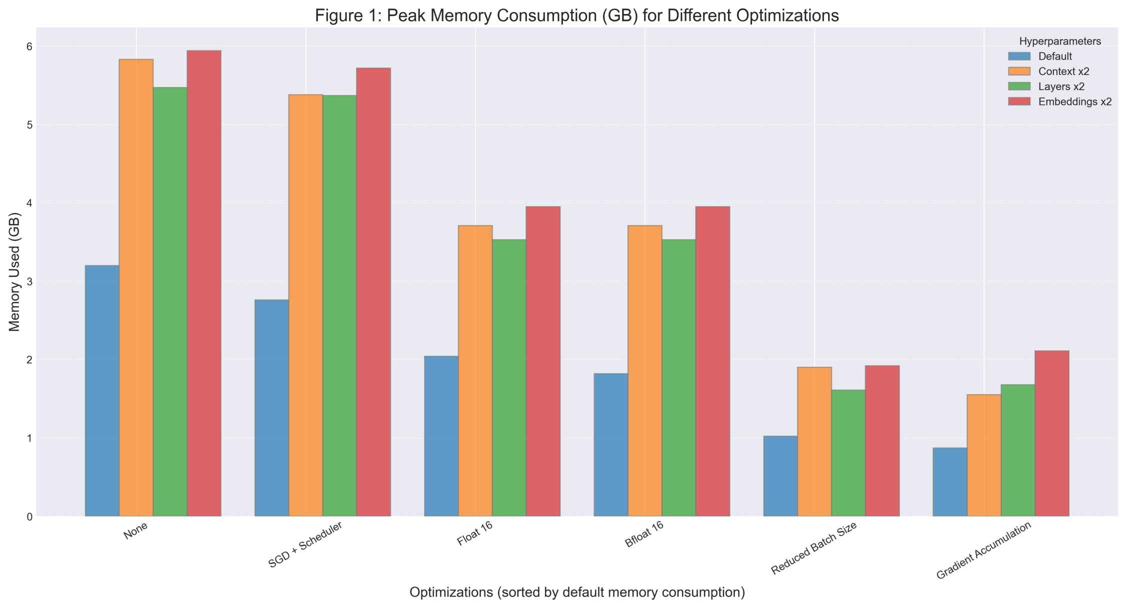
Experiments were also run exploring several modifications to these hyperparameters. These changes involved **(1)** doubling the number of layers to 12, **(2)** doubling the context size to 512 characters, and **(3)** doubling the embedding dimensions to 768. The applied optimization techniques included **(1)** gradient accumulation, **(2)** reducing the batch size from 64 to 16, **(3)** using the float16 format, **(4)** using the bfloat16 format, and **(5)** utilizing a stateless optimizer (Stochastic Gradient Descent) with a cosine decay learning rate scheduler. Training runs without any one of these optimizations were run using the float32 format.

Thus, there were 24 training runs, one for each of the 4 model configurations and 6 possible optimizations. For each run, training duration in seconds, peak memory utilization, as reported by the `torch.cuda.max_memory_reserved()` command and final loss were recorded.

Results

More tables and figures can be found at the end of the document.

Hyperparameters	Optimization	Memory Used (GB)	Training Time (s)	Final Loss
Default	None	3.20	239.97	0.8247
	SGD + Scheduler	2.76	228.39	2.5818
	Float 16	2.04	157.77	0.8079
	Bfloat 16	1.82	150.99	0.8209
	Reduced Batch Size	1.02	104.26	1.1896
	Gradient Accumulation	0.87	229.22	0.8288
Context x2	None	5.83	459.59	0.5426
	SGD + Scheduler	5.38	449.02	2.5865
	Float 16	3.71	260.81	0.5450
	Bfloat 16	3.71	262.45	0.5431
	Reduced Batch Size	1.90	157.54	1.0484
	Gradient Accumulation	1.55	375.62	0.5513
Layers x2	None	5.47	424.55	0.3751
	SGD + Scheduler	5.37	423.20	2.5683
	Float 16	3.53	265.74	0.3733
	Bfloat 16	3.53	265.06	0.3750
	Reduced Batch Size	1.61	166.13	1.1355
	Gradient Accumulation	1.68	396.46	0.3914
Embeddings x2	None	5.94	529.70	0.3195
	SGD + Scheduler	5.72	519.50	2.5349
	Float 16	3.95	312.25	0.3112
	Bfloat 16	3.95	304.39	0.3002
	Reduced Batch Size	1.92	179.93	1.3644
	Gradient Accumulation	2.11	417.97	0.3222



Changing Hyperparameters

As illustrated in Figure 4, doubling the number of layers, context, and embedding dimensions generally elevated memory consumption and training duration while concurrently reducing the loss. These observed results align with the anticipated relationship between these hyperparameters and their corresponding metrics. These enhancing parameters increase model performance while demand more computational resources.

In the cases where no optimizations were applied, the most pronounced reduction in loss (-61%) was observed when the embeddings were doubled; however, this also led to the most significant increases in memory usage and training duration, with respective increments of +85% and +151%. The act of doubling the number of layers emerged as the hyperparameter adjustment that offered the most favorable balance between the percentage loss reduction and the percentage escalation in both memory or training duration. Notably, this trend was not consistent in runs where optimizations were applied. Further insights into this will appear in an ensuing subsection.

Representation Formats

As expected, utilizing the float16 and bfloat16 formats decreased peak memory usage and reduced runtime without affecting performance. The anticipated advantages of these formats stem from their inherent design: they allocate half the memory space compared to the float32 format, trading precision for range.

Contrary to initial expectations, the float16 and bfloat16 representations demonstrated nearly identical metrics. This observation diverges from the results presented in the blog post discussed in class where bfloat16 showed a notable improvement in peak memory usage over float16. A potential distinction in our methodology includes the utilization of a gradscaler with the float16 format, a technique not employed in the aforementioned post. Furthermore, it's conceivable that the increased range of bfloat16 over float16 might become more pronounced and advantageous in the context of training larger models.

SGD Optimizer

Transitioning from the Adam optimizer to SGD yielded a marginal improvement in performance, while concurrently yielding a significant escalation in loss. Quantitatively, with the standard model configuration, the shifts recorded were a decrease in memory usage by 14%, a reduction in training time by 5%, and an increase in loss by 213%. The observed alterations, albeit in anticipated directions, presented magnitudes that were somewhat unexpected. In particular, the changes in memory and training time were relatively modest, marking this optimization as having the least influence on memory efficiency and training duration. Regarding the elevated loss, it is known that standard SGD usually requires more iterations to converge compared to the Adam optimizer. This likely accounts for the higher loss observed at the conclusion of the 5000 iterations. Nevertheless, it was anticipated that the learning rate scheduler would play a more pronounced role in offsetting this effect. I suspect that changing the learning rate scheduler's parameters could significantly improve the effects of this optimization. My specific implementation can be found here:

`https://github.com/frowenz/nanoGPT/blob/master/train_sgd.py`.

Reduced Batch Size

In line with expectations, reducing the batch size led to a marginal increase in loss while substantially diminishing both memory usage and training duration. Specifically, with the standard model configuration, memory consumption was reduced by 68%, training time decreased by 57%, and there was a 44% rise in loss. A smaller batch size inherently implies fewer activations being retained and a reduction in computational tasks per iteration. Additionally, the computational overhead per iteration is lighter with smaller batches. Consequently, these trends align well with the anticipated outcomes of batch size reduction.

Gradient Accumulation

Employing gradient accumulation led to a notable improvement in memory efficiency without impacting the model's loss or training duration. This outcome is consistent with the foundational principle of gradient accumulation, where gradients from multiple mini-batches are aggregated before a single update, thereby reducing the memory requirement for each individual gradient computation. Specifically, in the context of our standard model configuration, gradient accumulation presented the most pronounced memory savings, accounting for a substantial 73% reduction in memory usage.

Interaction between Hyperparameters and Optimizations

There were instances of non-linearity when combining non-default model configurations with optimizations. Specifically, the change in a given performance metric, when both a model modification and an optimization were applied, was not merely the sum of the changes observed when each was applied independently. See Figure 5 for full details.

Using bfloat16 or float16 in conjunction with doubled hyperparameter configurations enhanced memory usage and training beyond a mere linear combination of the two. This can be attributed, in part, to GPU optimizations tailored for lower-precision calculations.

The observed synergistic effects between gradient accumulation and the doubling of hyperparameter configurations on both memory and training were unexpected, especially since gradient accumulation primarily enhanced memory in isolation. One hypothesis for this phenomenon is that gradient accumulation stabilizes the learning process, especially when introducing more complexity through hyperparameter adjustments. By moderating and smoothing the learning dynamics, gradient accumulation might help harness the potential of increased model complexity, leading to more efficient and robust training outcomes.

Sources of Error Discussion

Measuring the peak memory consumption during training can be tricky. I opted to use `torch.cuda.max_memory_reserved()` over alternatives like `nvidia-smi`. My decision was driven by the aim to secure a metric more tightly aligned with PyTorch's internal memory management, ensuring better comparisons between different training runs.

The function `torch.cuda.max_memory_reserved()` tracks the maximum GPU memory that PyTorch's caching allocator reserves. It offers an understanding of the extent of memory PyTorch commits, even if a portion of this is dormant. On the other hand, `nvidia-smi` provides a comprehensive snapshot of GPU memory, encompassing both the memory occupied by PyTorch and additional potential GPU usages not necessarily linked to our training.

Because `torch.cuda.max_memory_reserved()` is PyTorch-centric, it is unaware of GPU memory being used by other processes. Elements such as the CUDA context, GPU driver overheads, and allocations not related to PyTorch add to the total GPU memory consumption. As a result, `torch.cuda.max_memory_reserved()` might lean towards underestimating the true peak memory consumption of the GPU. If the goal were to discern precise memory requirements, relying solely on this metric would be insufficient.

Because of constraints on both time and computational resources, each experimental run was conducted just once. It is important to acknowledge the inherent variability present in runtime, memory usage, and loss metrics, even for runs with identical configurations. Such variability introduces a degree of uncertainty into our results. With that said, in the process of preparing the experiments, I observed identical runs with the same settings but different seeds and found that the evaluation metrics obtained were mostly consistent.

As a final source of error, there was a bug in my gradient accumulation training script. This required rerunning these experiments on a different Google Colab runtime. Although the computation still employed an A100 GPU, differences in instance configurations or background processes inherent to different runtimes could have injected minor inconsistencies.

Improvement Suggestions

If I were to redo or refine these experiments, I would consider the following:

- **Granular Hyperparameter Variation:** Adopt a more granular approach to hyperparameter adjustments rather than solely doubling specific values. This might yield a comprehensive perspective on how each hyperparameter affects the training phase and its interaction with the optimizations.

- **Multiple Runs on Varied Hardware:** Conduct experiments multiple times with varied seeds on different hardware to offset potential errors due to hardware variability and attain more dependable outcomes.

- **Loss Thresholding:** Gauge the duration required for the model to hit a designated loss threshold, as opposed to maintaining a fixed 5000 training iterations. Considering the slower convergence rate associated with the SGD optimizer, this method could present a clearer view of the time required to achieve a model of a specific quality, arguably a more pertinent metric. This would also be useful in illuminating the benefits of reducing the batch size.

- **Interactions Between Optimizations:** Investigate the interplay between simultaneously applying optimizations to unearth nuanced interactions that could offer a deeper understanding of the model's behavior under different configurations.

- **Interactions Between Hyperparameters:** Investigate the interplay between simultaneously changed hyperparameters for the same aforementioned reasons.

More Tables and Charts

Figure 2: Training Time (s) for Different Optimizations

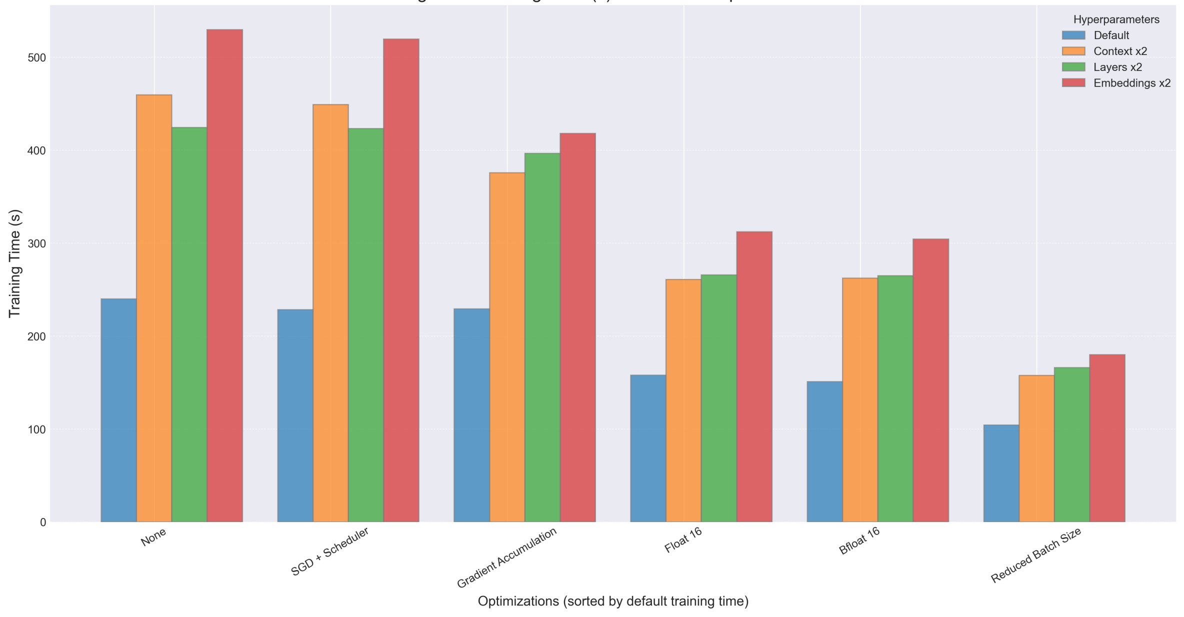


Figure 3: Final Loss for Different Optimizations

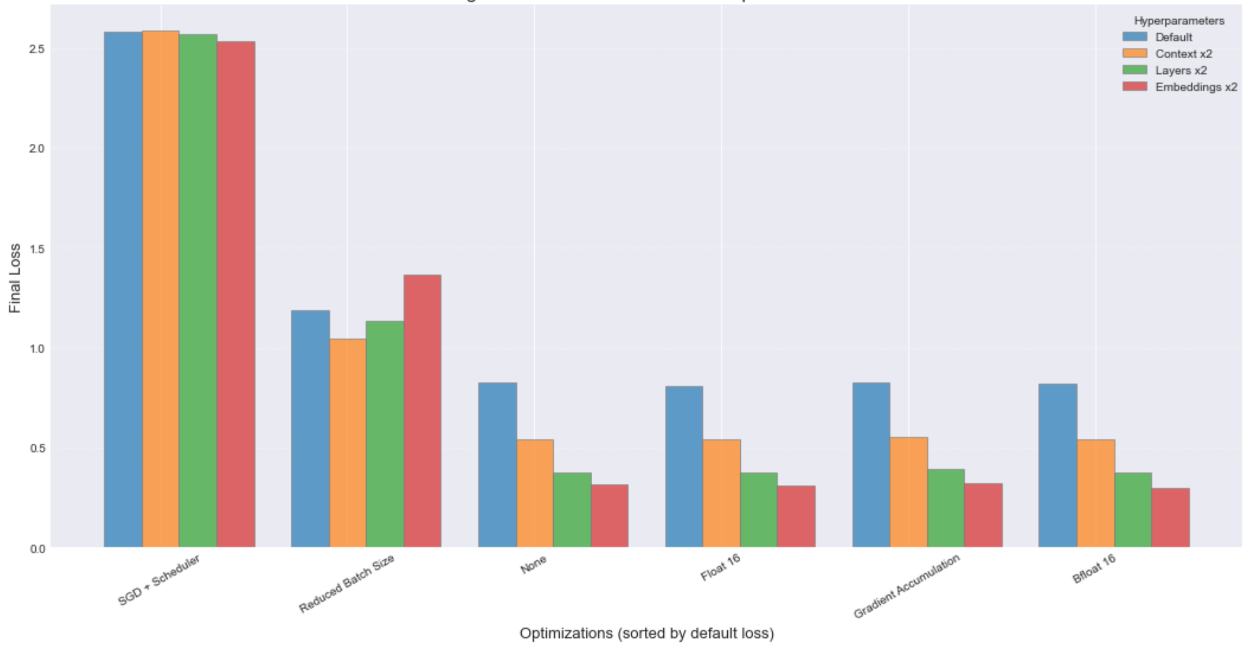


Figure 4: Percentage Change in Metrics for Each Configuration Compared to the Default

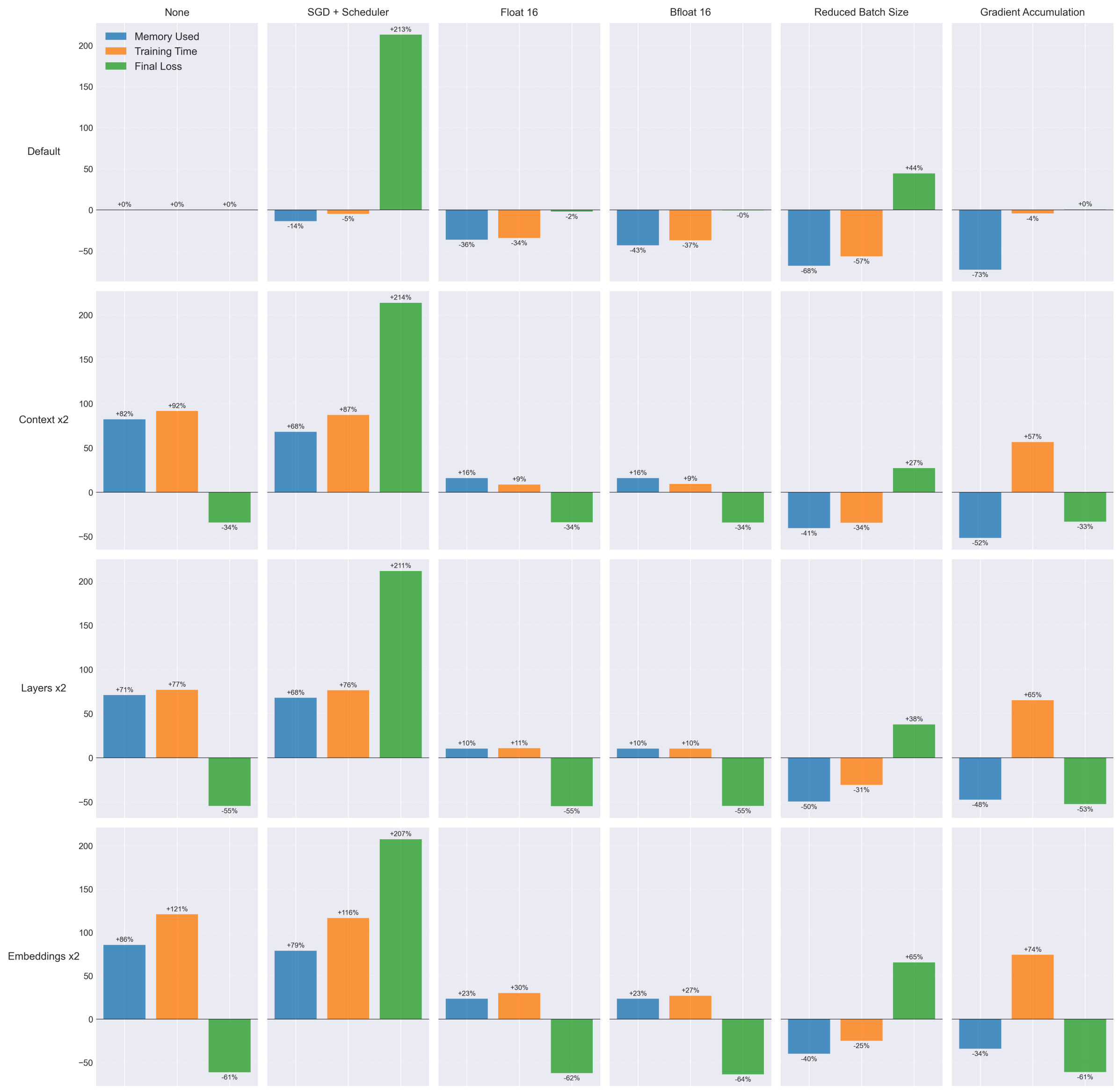
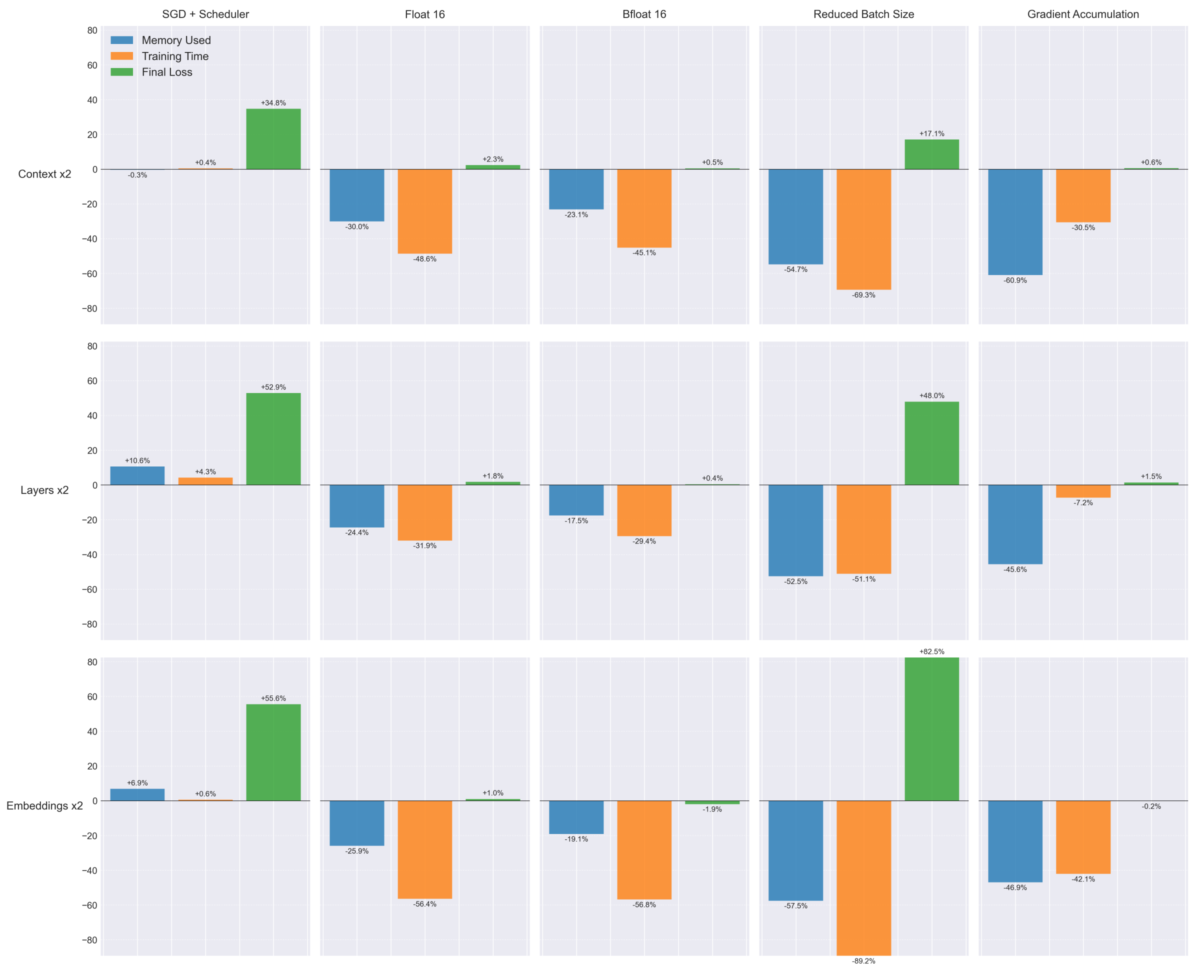


Figure 5: Deviation from Expected Additivity in Memory, Training Time, and Loss Across Configurations



Optimization	**Hyperparameters**	**Memory Used (GB)**	**Training Time (s)**	**Final Loss**
None	Default	3.20	239.97	0.8247
	Context x2	5.83	459.59	0.5426
	Layers x2	5.47	424.55	0.3751
	Embeddings x2	5.94	529.70	0.3195
SGD + Scheduler	Default	2.76	228.39	2.5818
	Context x2	5.38	449.02	2.5865
	Layers x2	5.37	423.20	2.5683
	Embeddings x2	5.72	519.50	2.5349
Float 16	Default	2.04	157.77	0.8079
	Context x2	3.71	260.81	0.5450
	Layers x2	3.53	265.74	0.3733
	Embeddings x2	3.95	312.25	0.3112
Bfloat 16	Default	1.82	150.99	0.8209
	Context x2	3.71	262.45	0.5431
	Layers x2	3.53	265.06	0.3750
	Embeddings x2	3.95	304.39	0.3002
Reduced Batch Size	Default	1.02	104.26	1.1896
	Context x2	1.90	157.54	1.0484
	Layers x2	1.61	166.13	1.1355
	Embeddings x2	1.92	179.93	1.3644
Gradient Accumulation	Default	0.87	229.22	0.8288
	Context x2	1.55	375.62	0.5513
	Layers x2	1.68	396.46	0.3914
	Embeddings x2	2.11	417.97	0.3222

Table 1: Training results grouped by optimization