EE 599 ML Systems Final Project
Efficient Fine-tuning of LLM on a Single GPU

Instructor: Arash Saifhashemi
TA: Lei Gao, Kevin Yang

1 Introduction

Transformer-based large language models (LLMs) like OpenAI's GPT-4 and ChatGPT have significantly expanded language generation and understanding capabilities. Trained on extensive web-scale datasets, these models excel not only in natural language processing but also in solving complex tasks through adept instruction handling and multi-step reasoning. This makes LLMs pivotal in advancing towards artificial general intelligence (AGI).

In our project, we focus on instruction tuning as a key method for adapting pretrained LLMs, using Meta AI's open-sourced LLaMA2 7B model. We'll investigate efficient fine-tuning techniques like gradient accumulation, checkpointing, mixed precision training, and low-rank adaptation, which are crucial for optimizing LLMs in resource-constrained environments, particularly for mitigating concerns regarding memory usage.

2 Background

In this section, your task is to review the LLM survey paper [1] and the LLaMA papers [2, 3] selectively, and provide comprehensive answers to the questions below. These questions will serve as a guide for reading and understanding the content of the papers. You may use Google or ChatGPT to assist in your research, but ensure that your answers are accurate and demonstrate a clear understanding of the concepts discussed in the papers.

1. What are the four major aspects of LLMs covered in the LLM survey paper?
2. What are the three major differences between LLMs and PLMs? What are the three typical emergent abilities for LLMs?
3. Where are pre-training data from?
4. What are the three main types of instruction tuning datasets? What is the Alpaca dataset [4]? Pick a training sample and describe it.
5. What are the pertaining data used by LLaMA? How many tokens are in the entire training set for training LLaMA?
6. Before pre-training, what is the procedure for data preprocessing?
7. What is tokenization? Name a few tokenization algorithms. What is the tokenization method used by LLaMA?
8. What are the main three types of transformer architecture? Name a few models for each type.
9. Why are LLMs mainly decoder-only architecture?
10. What is position embedding? Name a few position embedding algorithms.
11. What is language modeling? What is the difference between causal language modeling and masked language modeling?

12. How to generate text from LLMs [5]? Explain different decoding strategies.
13. What is instruction tuning, and why is it important?
14. What is alignment tuning, and why is it important?
15. What is parameter-efficient fine-tuning, and why is it important? What is the difference between prefix tuning and prompt tuning?
16. What is prompt engineering, and why is it important? What is zero-shot and few-shot demonstrations?
17. What is the difference between in-context learning and chain-of-thought prompting?
18. What are the three basic types of ability evaluation for LLMs? What is perplexity [6]? What is hallucination?
19. What is human alignment, and why is it important?
20. Name a few comprehensive evaluation benchmarks for the evaluation of LLMs.

Your answer:

3 Optimization Methods

3.1 Gradient Accumulation

In standard neural network training, we typically divide our data into smaller chunks called mini-batches and process them sequentially. The network generates predictions for each batch, and we compute the loss by comparing these predictions to the actual targets. Afterward, we perform a backward pass to calculate gradients, which are used to adjust the model's weights in the direction of these gradients.

Gradient accumulation alters this last step of the training process. Instead of updating the model's weights after processing each individual batch, we save the gradient values and continue to the next batch, accumulating the new gradients. Weight updates are deferred until the model has processed several batches.

The primary purpose of gradient accumulation is to simulate a larger effective batch size. For instance, imagine you intend to use a batch size of 32 images, but your hardware can handle only up to 8 images without running out of memory. In this scenario, you can work with smaller batches of 8 images and update the weights after every 4 batches by accumulating gradients from each batch in between. To ensure the accumulated gradient remains equivalent to the non-accumulated gradient on a batch of 32 images, you must divide your loss value on each mini-batch of 8 images by 4. Consider a linear model with MSE loss, and mathematically explain why this division step can yield identical results.

Your answer:

However, it's important to note that gradient accumulation may not always yield identical results, especially for deep neural network architectures that employ batch-wise regularization methods like batch normalization. Batch normalization relies on statistics calculated within each batch, and when you accumulate gradients, these statistics may not be consistent. Mathematically explain the batch normalization layer behavior and why gradient accumulation yields non-identical results in this case.

Your answer:

3.2 Gradient Checkpointing

Gradient checkpointing is another memory-saving technique that strikes a balance between memory and computation during neural network training. While it's well understood that storing all activations and intermediate results from the forward pass is essential for performing backward propagation, this process can lead to a significant increase in memory usage, often referred to as the "memory blow-up" problem. When monitoring memory consumption during model training, you'll notice that it gradually accumulates and reaches its peak after completing the forward pass. Explain why model inference does not have such "memory blow-up" problem.

Your answer:

Instead of retaining all activations and intermediate results in memory throughout the forward pass, gradient checkpointing offers an alternative approach. It strategically selects checkpoints within the network where memory efficiency takes precedence over computation time. When the algorithm encounters these checkpoints during the subsequent backward pass, it does not rely on stored activations because they were not stored in memory in the first place. Instead, it recalculates them by re-executing the forward pass up to the specific checkpoint.

To optimize memory usage and forward computation steps using gradient checkpointing in a feed-forward neural network composed of n layers, let's consider two strategies: Imagine a simple feed-forward neural network with n layers, each generating activations of the same size. Upon completing the forward pass, the algorithm retains all activations, resulting in an $O(n)$ memory requirement for this network. In other words, it consumes n units of memory. In terms of computation, the forward pass requires $O(n)$ steps, assuming that each layer requires 1 unit of computation.

Now, the question revolves around determining the optimal placement of checkpoints to strike a balance between memory and computation. A poor strategy would be discarding all activations during the forward pass and recomputing them later when needed. Conversely, a more general and optimal strategy suggests discarding activations every $\sqrt{n}$ steps. What are the memory requirement and forward computation steps for the two strategies in big O notation? Check this medium post [7] for more illustrations.

Your answer:

In practical scenarios, neural networks often feature layers with varying activation sizes, and the computational complexity of each layer's forward pass can differ significantly. Furthermore, the forward graph of these networks may not adhere to a strictly sequential pattern. Consequently, the task of strategically placing checkpoints to optimize memory usage and computation steps remains an active area of research and attention.

3.3 Low-Rank Adaptation (LoRA)

Parameter-efficient Fine-tuning (PEFT) is a technique in NLP that enhances the performance of pre-trained language models on specific tasks. It freezes most of the pre-trained model's parameters and only fine-tunes a subset of layers, reducing computational and memory demands and training time, while retaining a good performance as possible. The idea of reducing the number of trainable parameters has its roots in transfer learning within computer vision tasks, where traditionally, only the final fully connected layer was updated.

Before we dive into the technical aspect of Low-Rank Adaptation (LoRA), one must revisit the transformer architecture and understand the mechanism of self-attention. In addition, LoRA uses the concept of truncated Singular Value Decomposition (SVD). Please study and understand the concepts of SVD and truncated SVD. Answer the following questions:

1. What is matrix rank?
2. What are three decomposed matrices by SVD?
3. U and V are orthogonal matrices. Why does it imply $UU^T = I, VV^T = I$?
4. If a matrix $W \in R^{n\times n}$ is full rank, what is its rank?
5. Suppose a full rank matrix $W \in R^{n\times n}$ represents an image. After we apply SVD to this matrix, we modify the singular matrix by only keeping its top-k singular values and discarding the rest (i.e., set the rest of the singular values to zero). Then, we reconstruct the image by multiplying U, modified S, and V. What would the reconstructed image look like? What if you increase the values of k (i.e., keep more singular values)?
6. If a matrix $W \in R^{n\times n}$ is low rank, what does its singular matrix look like?
7. If the top-k singular values of a matrix $W \in R^{n\times n}$ are large, and the rest are near zero, this matrix W exhibits low-rank or near-low-rank behavior. Can you represent W by two low-rank matrices, A and B? If so, what are those two matrices' expressions in terms of U, S, and V? Do you think those two matrices are a good approximation of W (i.e., $W \approx AB$)?

8. The above operation is called truncated SVD. Under what situation do you think truncated SVD fails to make a good approximation? Think about the singular matrix.

Your answer:

Now, you are equipped with the knowledge to understand LoRA. As one of the most effective PEFT techniques, proposed by Edward Hu et al. in 2021 [8], the success of LoRA is based on an important observation that both pre-trained weights and change of weights during fine-tuning for LLMs are low-rank matrices.

To illustrate this, consider representing the final optimal weights after fine-tuning as $W^* = W_0 + \Delta W$, where $W_0 \in R^{n\times n}$ represents the pre-trained weights, and $\Delta W \in R^{n\times n}$ captures the weight changes during fine-tuning. Notably, ΔW can be approximated as the product of two matrices, AB, with $A \in R^{n\times r}$ and $B \in R^{r\times n}$. This implies that during fine-tuning, we don't have to update the whole matrix W_0. Instead, we modify the forward pass in the layer, changing it from $y = xW_0$ to $y = xW_0 + xAB = x(W_0 + AB)$, as depicted in Figure 1. We freeze W_0 and only update A and B. Notice that since $r << n$, the total number of parameters of A and B is much smaller than that of W_0. To make the modified forward pass identical to the original one at the beginning of fine-tuning, A is randomly initialized in Gaussian distribution, and B is initially set to 0, so $AB = 0$.

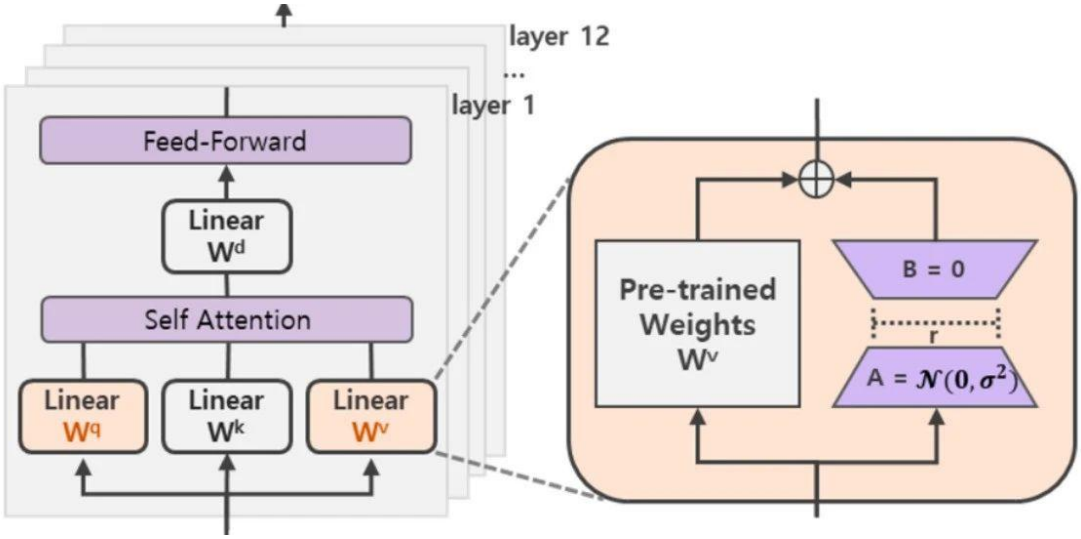

Figure 1: LoRA

Technically, we can insert A and B into any layers, such as the query, key, value, output projection layers, or the FFN layers in each attention block. However, the paper suggests that applying LoRA only to query and value projection layers can yield performance comparable to full parameter fine-tuning. The hyperparameter r typically takes on small values within the range of 4 to 32. By inserting A and B into specific layers and keeping the rest of the pre-trained weights frozen, LoRA achieves a substantial reduction in the total number of trainable parameters. For $r = 8$, the percentage of parameters that are subject to training accounts for less than 1% of the overall parameters in a typical LM. This remarkable reduction in the number of trainable parameters is a key factor in the efficiency of LoRA as a PEFT technique.

It seems like applying LoRA would add more parameters to the model, thus increasing forward pass costs and inference latency. How does LoRA paper overcome this drawback?

Your answer:

3.4 Mixed Precision Training

Mixed precision training [9] is a technique used in large neural network training that combines the use of both 16-bit and 32-bit floating-point representations for different parts of the training process. This approach

leverages the advantages of lower precision (16-bit) for some computations while still using higher precision (32-bit) where necessary.

As illustrated in Figure 2, during the forward pass, the model converts the FP32 weights to FP16 and computes FP16 activations, and in the backward pass, both activation gradients and weight gradients are calculated in FP16. Subsequently, the FP32 master copy of weights is updated with the FP16 weight gradients. The authors also claim that some operations should be read and written in FP16 but perform arithmetic in FP32 to maintain model accuracy. Read the paper and answer why we need an FP32 master copy of weights and why we need to scale up the loss.

Your answer:

FP16 operations offer significant speed advantages over FP32 operations, especially on modern GPUs equipped with NVIDIA's Volta and Ampere architectures. These architectures include dedicated hardware, known as Tensor Cores, designed to accelerate FP16 multiplication and accumulation into either FP16 or FP32 outputs. Thus, it is clear that mixed precision training can reduce computation time by utilizing FP16 computation units. In terms of memory consumption, according to the paper:

"Even though maintaining an additional copy of weights increases the memory requirements for the weights by 50% compared with single precision training, impact on overall memory usage is much smaller. For training memory consumption is dominated by activations, due to larger batch sizes and activations of each layer being saved for reuse in the back-propagation pass. Since activations are also stored in half-precision format, the overall memory consumption for training deep neural networks is roughly halved. "

Do you think this statement still holds for LLM fine-tuning?

Your answer:

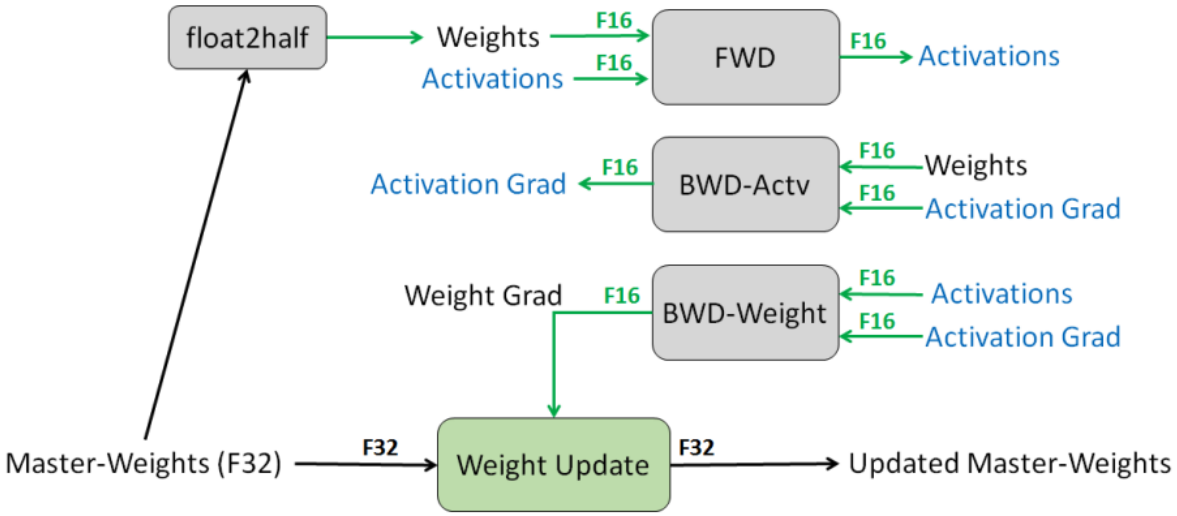

Figure 2: Mixed precision training

4 Phase 1: Preliminary Study

Complete all questions in Sections 2 and 3, and upload your answers as a PDF document named `phase_1` to GitHub. It's strongly advised to complete this phase as soon as possible to allow enough time for the subsequent phases.

5 Phase 2: LLaMA2 Model Inference

In this phase, we focus on understanding how the LLaMA2 7B model generates text and making specific changes to its code. Our goal is to remove the parts of the code that deal with KV-caching. The LLaMA2 7B model and its tokenizer are located at the directory `project/saifhash_1190/llama2-7b` on the CARC system.

Guidelines:

1. Understanding the Code Base: Familiarize yourself with the model's architecture, the tokenization process, batch generation, and the decoding mechanism. Utilize debugging tools to inspect values and flow as necessary.

2. Identifying KV-Caching Features: Look through the code to locate all components and functions related to KV-caching.

3. Modifying the Code: Carefully remove the KV-caching features from the code. Make sure the model can still generate text without these features.

4. Testing: After making changes, test the model with sample prompts to ensure it still works correctly.

Deliverables: Submit a PDF document named `phase_2` containing the details of the changes you've made to the code, along with the prompts and the outputs from testing the generation process.

6 Phase 3: LLaMA2 Model Training

In this phase, your task is to conduct instruction tuning on the LLaMA2 7B model using the (partial) Alpaca dataset, which enhances and expands the capabilities of the LLaMA2 model. By applying all previously discussed optimization techniques, fine-tuning LLMs will become feasible, even on a single GPU. We will avoid using high-level libraries such as HuggingFace, which abstracts away many important details. Instead, we will solely rely on PyTorch's built-in functions. This approach will ensure you gain a comprehensive understanding of the process involved in the instruction tuning of LLMs.

Guidelines:

1. Initial Setup: Begin by instantiating only 1 decoder layer of the LLaMA2 model. Test all your code on a less powerful yet more accessible GPU, the P100 on CARC. This approach prevents out-of-memory errors even before applying optimization techniques. Once you've ensured an end-to-end training process that is bug-free and incorporates all four optimization techniques, you can scale up to the full-size LLaMA2 model, which includes 32 decoder layers, and utilize a more powerful GPU, the A100, with 40GB of RAM, on CARC system.

2. End-to-End Instruction Tuning Flow: Create a file named `finetuning.py`. Implement the end-to-end instruction tuning workflow in PyTorch. For data preprocessing and the creation of supervised data loaders, refer to the official Alpaca repo.

3. Training Iteration Loop: Replace the HuggingFace `trainer` object used in the Alpaca repo with your own implementation from scratch. To compute loss using PyTorch functions, refer to the HuggingFace's implementation.

4. Gradient Accumulation and Mixed Precision Training: Implement gradient accumulation and mixed precision training using PyTorch's AMP package. Refer to the AMP Recipe and AMP Examples for guidance.

5. LoRA Linear Layer Module: Implement the LoRA linear layer module by referring to the official implementation. Create a file named `lora.py` under the `llama/model` path. Convert your model into a PEFT model by replacing the Q and V projection layers in the LLaMA2 model with your LoRA `Linear` modules. Freeze all model parameters except for `LoRA_A` and `LoRA_B` and report the percentage of trainable parameters.

6. Gradient Checkpointing: Utilize PyTorch's `torch.utils.checkpoint` API for inserting checkpoints. Decide which layer(s) to apply checkpointing to and describe your approach in the report. Refer to this tutorial for more information.

7. Model Fine-Tuning: Apply all the aforementioned techniques to your LLaMA2 7B model and fine-tune it on the Alpaca Dataset using the A100 GPU. To manage time constraints for educational purposes, extract the first 200 samples from the dataset as your training set.

8. Hyperparameters: Set $learning\_rate = 1e-5$, $batch\_size = 1$ and $gradient\_accumulation\_step = 8$. For LoRA configuration, set $r = 16$, $alpha = 32$, and $dropout\_rate = 0.05$.

Deliverables: Submit a PDF file named `phase_3` that includes the implementation details you have completed, along with the prompt and the result used to test the fine-tuned model. In the same PDF file, provide the following analysis and measurements. For Table 1, fill in the blanks with $\uparrow$ to signify an increase in resource usage, $\downarrow$ for a decrease, or $-$ to indicate no change. For Table 2, measure and report the peak memory usage in MB and the runtime in seconds required to process only 200 training samples on the A100 GPU across each combination of techniques applied in the instruction tuning of the LLaMA2 7B model. If an out-of-memory error happens, then simply mark $\times$ in the blanks. You can turn off gradient checkpointing for this step.

		Grad. Accumulation	Grad. Checkpoint	Mixed Precision	LoRA
Memory	parameter				
	activation				
	gradient				
	optimizer state				
Computation					

Table 1: System performance analysis

GA	OFF				ON			
MP	OFF		ON		OFF		ON	
LoRA	OFF	ON	OFF	ON	OFF	ON	OFF	ON
Peak Mem								
Runtime								

Table 2: System performance measurement

References

[1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2023.

[2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[3] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[4] "Huggingface datasets tatsu-lab/alpaca." [Online]. Available: https://huggingface.co/datasets/tatsu-lab/alpaca

[5] "How to generate text: using different decoding methods for language generation with transformers." [Online]. Available: https://huggingface.co/blog/how-to-generate

[6] "Perplexity of fixed-length models." [Online]. Available: https://huggingface.co/docs/transformers/perplexity

[7] "Fitting larger networks into memory." [Online]. Available: https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[9] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," 2018.