EE 508 Final Project
Efficient Processing of Large Language Models

Instructor: Arash Saifhashemi
TA: Lei Gao, Kevin Yang, Chaoyi Jiang

1 Introduction

Large language models (LLMs) based on transformer architecture, such as OpenAI's GPT-4 and ChatGPT, have greatly improved the ability to generate and understand text. These models are trained on large-scale datasets from the internet and can perform well on a wide range of language tasks. They are also able to follow instructions and handle multi-step problems, which makes them useful for more complex applications and brings us closer to building general-purpose AI systems.

In this project, we will use Meta AI's open-source LLaMA3.2 1B model to explore both efficient inference and fine-tuning techniques. On the inference side, we will evaluate methods such as key-value (KV) caching to reduce latency and computational overhead. For fine-tuning, we will investigate approaches that improve training efficiency and reduce resource requirements, including gradient accumulation, gradient checkpointing, mixed-precision training, and low-rank adaptation (LoRA).

2 Phase 1: Background Knowledge

In this phase, you will review the shortened version of the LLM Foundations paper provided in the project repository. After reading through the paper, please answer the questions listed below. You are welcome to use external resources like Google or ChatGPT to help with your understanding, but you are responsible for ensuring the accuracy and completeness of your responses.

1. What is language modeling?
2. What is self-supervised pertaining?
3. Why is pretraining more hardware-efficient for Transformer- or attention-based models compared to RNN-based models?
4. What is the difference between encoder-only and decoder-only models, and why are decoder-only models more popular?
5. Suppose the vocabulary consists of only three words: *Apple, Banana, and Cherry.* During decoder-only pretraining, the model outputs the probability distribution $\Pr_\theta(\cdot \mid x_0, \ldots, x_i) = (0.1, 0.7, 0.2)$. If the correct next word is *Cherry*, represented by the one-hot vector $(0, 0, 1)$, what is the value of the log cross-entropy loss? What is the loss value if the correct next word is *Banana* instead?
6. What are zero-shot learning, few-shot learning, and in-context learning?
7. Why is fine-tuning necessary? What is instruction fine-tuning?
8. What is tokenization? What is a word embedding layer?
9. What is position embedding? What kind of position embedding method is used in Llama models?
10. What is the difference between multi-head attention and grouped-query attention? Which type of attention mechanism is used in Llama models?

11. What kind of activation function is used in Llama models?
12. What is layer normalization? What is the difference between layer normalization and batch normalization?
13. What is the auto-regressive generation process, and how is a decoding strategy used during text generation?

Deliverable: Submit a PDF file named `phase_1.pdf` to the project repository with your answers to the questions above.

3 Phase 2: LLM Inference

We first describe the LLM inference process and then provide guidance for this phase.

3.1 LLM Inference Process

The inference process of decoder-only LLMs employs an auto-regressive approach, generating tokens sequentially. It consists of two stages: the **prefilling stage** and the **decoding stage**. In the prefilling stage, the input to the i-th decoder layer is denoted as $X^i \in \mathbb{R}^{b \times s \times h}$, where $i \in \{1, \ldots, n\}$, b is the batch size, s is the prompt length, and h is the input embedding dimension. The Attention block computes a set of queries (Q), keys (K), and values (V) through linear projections of X^i:

$$Q^i = X^i \cdot W_Q^i, \quad K^i = X^i \cdot W_K^i, \quad V^i = X^i \cdot W_V^i, \tag{1}$$

where $W_Q^i, W_K^i, W_V^i \in \mathbb{R}^{h \times h}$ are the projection matrices. The generated K^i and V^i are stored in the KV cache.
The self-attention score in Attention block is computed as:

$$Z^i = \text{softmax}\left(\frac{Q^i (K^i)^T}{\sqrt{d_{\text{head}}}}\right) \cdot V^i, \tag{2}$$

where d_{head} represents the dimension of each attention head. Finally, the attention score is applied with a linear projection to produce the output of the Attention block:

$$O^i = Z^i \cdot W_O^i, \tag{3}$$

where $W_O^i \in \mathbb{R}^{h \times h}$ is the projection matrix.

The feedforward network (FFN) is followed after the Attention block, which consists of two fully connected layers with a non-linear activation function applied between them. It processes the attention output O^i to generate the input for the next decoder layer as follows:

$$X^{i+1} = \sigma(O^i \cdot W_1^i) \cdot W_2^i, \tag{4}$$

where $W_1^i \in \mathbb{R}^{h \times d_{\text{FFN}}}$ and $W_2^i \in \mathbb{R}^{d_{\text{FFN}} \times h}$ are the weight matrices of the two linear layers, and $\sigma(\cdot)$ denotes the activation function.

In the decoding stage, the i-th decoder layer receives a single token $x^i \in \mathbb{R}^{b \times 1 \times h}$. The KV cache is updated by concatenating the newly computed key and value pairs with the existing ones:

$$\begin{aligned} K^i &= \text{concat}(K^i, \quad x^i \cdot W_K^i), \\ V^i &= \text{concat}(V^i, \quad x^i \cdot W_V^i). \end{aligned} \tag{5}$$

The remaining attention and feedforward computations in the decoding stage are identical to those in the prefilling stage.

3.2 Guidance

In this phase, you will study the LLaMA 3.2-1B codebase for text generation. Please follow the instructions in the project repository README file to download the model weights and run the Python script `inference.py`. You are expected to read and understand the code in `llama/model.py` and `llama/generation.py`. A helpful video tutorial provides an overview of the LLaMA model architecture and inference process.

After understanding the codebase, you are required to modify the code to disable the KV cache feature. Since KV caching is only used during inference and we will perform fine-tuning in the next phase, it is necessary to remove KV caching from the current implementation. This will ensure compatibility with training workflows where caching is not applicable. In summary, you will need to

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.

Deliverable: Submit a PDF file named `phase_2.pdf` that includes the following: a summary of the changes made to the code, and a comparison of model outputs before and after fine-tuning using sample prompts.

We also provide a benchmark script, `benchmark_inference.py`, in the repository. This script is designed to evaluate the inference performance of the LLaMA3.2-1B model under controlled conditions. It measures the runtime and peak GPU memory usage while varying the batch size, input prompt length, and output generation length. Benchmark the peak memory usage (in MB) and the runtime performance (in seconds) with and without the KV cache under the configurations listed below. Include your measurements in the same PDF file.

input len=256, output len=32		batch size=1	batch size=8	batch size=16
with KV cache	Peak Mem			
	Runtime			
without KV cache	Peak Mem			
	Runtime			

Table 1: Inference System Performance Benchmark

4 Phase 3: LLM Instruction Fine-Tuning

We first describe four important LLM fine-tuning techniques and then provide guidance for this phase.

4.1 Gradient Accumulation

Gradient accumulation is a method used to train neural networks when memory constraints prevent the use of large batch sizes. In traditional training, the model computes the loss and its corresponding gradients for each mini-batch, and immediately updates its weights using an optimizer. However, this approach can be problematic when the desired batch size is too large to fit into memory, especially when training deep models or working with high-resolution inputs.

Gradient accumulation changes the way updates are performed. Instead of updating the model after every mini-batch, the gradients are accumulated over multiple forward and backward passes. The model processes several smaller mini-batches, and the gradients from each are added together. After a predefined number of steps (referred to as the accumulation steps), the optimizer updates the weights using the sum of the accumulated gradients. This process effectively simulates training with a larger batch size, while still operating within the memory limits of the hardware.

To maintain consistency with what would be computed using a large batch, it is essential to normalize the loss. Specifically, the loss value from each mini-batch should be divided by the number of accumulation steps. This ensures that the accumulated gradients have the same scale as those obtained from a single large batch. Failing to normalize the loss would result in larger gradient magnitudes and potentially destabilize training.

While gradient accumulation is a practical solution for memory limitations, it comes with certain trade-offs. In particular, layers such as batch normalization rely on statistics computed within each mini-batch. When using small batches, the estimated statistics may not be as stable or representative as those computed from larger batches, potentially affecting model performance.

4.2 Mixed Precision Training

Mixed precision training is a technique used to accelerate the training of large neural networks by combining 16-bit and 32-bit floating-point representations in different parts of the training pipeline. This approach exploits the performance and memory advantages of lower precision (FP16) computations while retaining higher precision (FP32) where necessary to maintain numerical stability and accuracy.

As illustrated in Figure 1, during the forward pass, model weights stored in FP32 are cast to FP16, and the activations are computed in FP16. Similarly, in the backward pass, both activation gradients and weight gradients are computed in FP16. However, the FP32 master copy of the weights is updated using these FP16 gradients. The authors note that certain operations, such as Softmax, should read and write in FP16 but perform arithmetic in FP32 to avoid accuracy degradation.

Maintaining a master copy of the weights in FP32 is crucial for stable optimization due to two inherent limitations of FP16. First, FP16 cannot represent very small values—any number with magnitude less than 2^{-24} is rounded to zero. When gradients are scaled by a learning rate, they can easily fall below this threshold and be lost entirely. Second, even when an update is representable in FP16, it may still vanish during the weight update step if the weight's magnitude is much larger than the update. FP16 has only 10 bits of mantissa, and when the weight is more than 2048 times larger than the update, the update may be right-shifted out during addition, effectively becoming zero.

The authors also claim that although storing an additional FP32 master copy increases the memory footprint of the weights by 50% compared to pure FP32 training, the overall impact on memory usage is small. This is because training memory is typically dominated by activations, especially with large batch sizes, and these are stored in FP16. As a result, the total memory usage is roughly halved. However, this assumption does not always hold in the context of LLMs, where model weights take more memory usage than activations. In such cases, the memory overhead introduced by storing FP32 master weights becomes more significant, and the runtime reduction of mixed precision training may come at a higher memory cost.

4.3 Gradient Checkpointing

Gradient checkpointing is a memory-saving technique that balances memory usage and computational cost during neural network training. In standard training, all activations and intermediate results from the forward pass must be stored in memory to compute gradients during backpropagation. However, this approach leads to significant memory consumption, often referred to as the memory blow-up problem. If you monitor memory usage during training, you will notice that it increases progressively and typically peaks at the end of the forward pass. In contrast, model inference does not suffer from this memory blow-up problem because it does not require gradient computation. During inference, once the output of a layer is used in the subsequent layer, its activation can be immediately discarded. Since there is no need to retain intermediate results for backpropagation, the memory footprint remains much smaller throughout the forward pass.

Gradient checkpointing addresses the memory limitations during training by trading memory for additional computation. Instead of storing all activations, it selectively discards activations at specific checkpoints. During the backward pass, activations that were not stored are recomputed by re-executing parts of the forward pass up to the relevant checkpoints. This technique reduces memory usage at the expense of increased computation.

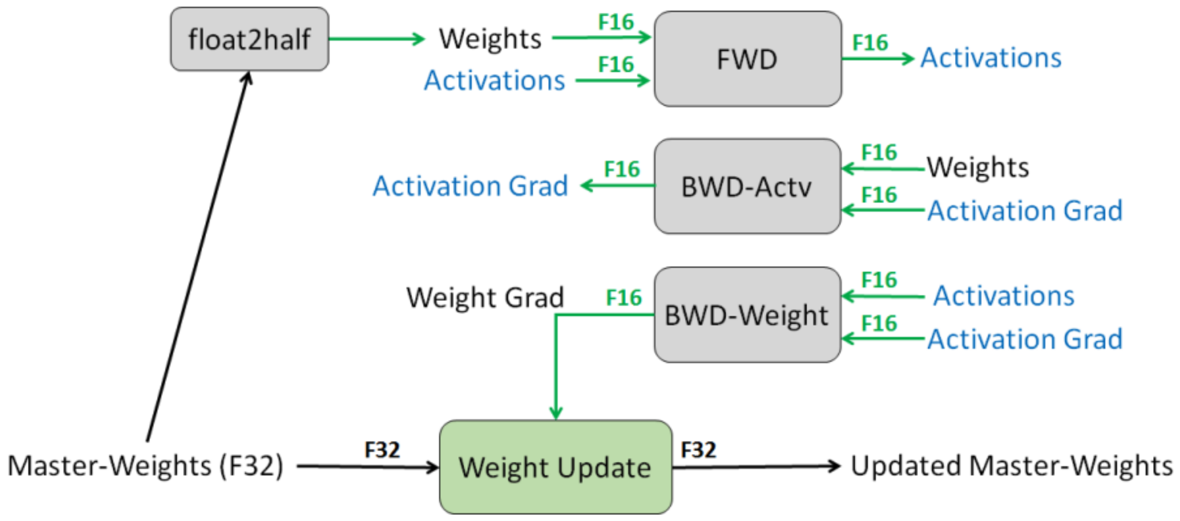

Figure 1: Mixed precision training

To understand the impact of checkpoint placement, consider a simple feed-forward neural network with n layers, where each layer produces activations of the same size and has uniform computation cost. If all activations are stored during the forward pass, the memory requirement is $O(n)$, and the forward computation takes $O(n)$ steps. A naive strategy would be to discard all activations and recompute them as needed during the backward pass. While this minimizes memory usage to $O(1)$, it leads to a computational cost of $O(n^2)$ because each layer's activation may need to be recomputed multiple times. An improved strategy is to store activations every $\sqrt{n}$ layers. With this approach, the memory requirement becomes $O(\sqrt{n})$, and the total forward computation, including recomputation during the backward pass, remains $O(n)$.

However, in practice, deep learning models are more complex than simple feed-forward architectures. They often include various types of layers—such as convolutional, attention, or recurrent layers, with differing computational costs and activation sizes. Moreover, the network topology may include branches, skip connections, and other non-linear structures. These factors make the optimal placement of gradient checkpoints significantly more challenging. As a result, selecting which activations to checkpoint in order to achieve the optimal trade-off between memory usage and computational overhead remains an open research question.

4.4 Low-Rank Adaptation (LoRA)

Adapting pre-trained LLMs to specific downstream tasks through fine-tuning is essential for achieving strong task performance. However, the large size and high computational demands of these models make fine-tuning challenging in resource-constrained settings. Parameter Efficient Fine-Tuning (PEFT) mitigates this issue by significantly reducing the number of trainable parameters and memory usage, while still achieving performance comparable to full-parameter fine-tuning.

LoRA, as one of the most effective PEFT techniques, is based on an important observation that both pre-trained weights and change of weights during fine-tuning for LLMs are low-rank matrices. For a linear layer, LoRA freezes the pre-trained weights $W_0 \in \mathbb{R}^{h\times h}$ and injects trainable low-rank matrices $A \in \mathbb{R}^{r\times h}$ and $B \in \mathbb{R}^{h\times r}$, constraining the weight updates in a low-rank space. This design significantly reduces the number of trainable parameters, as the rank r is much smaller than h. In practice, the hyperparameter rank r is usually chosen in the range of 4 to 32. The forward pass is then modified as

$$\boldsymbol{y} = W_0 \cdot \boldsymbol{x} + BA \cdot \boldsymbol{x}, \tag{6}$$

where input $\boldsymbol{x} \in \mathbb{R}^h$ and output $\boldsymbol{y} \in \mathbb{R}^h$. The matrix A is initialized from a random Gaussian distribution, and B is initialized to zero. Therefore, the output $\boldsymbol{y}$ remains the same as the original layer at the beginning of training.

Technically, A and B can be inserted into any layers within the transformer architecture, such as the query, key, and value projection layers or the feedforward network (FFN) layers. However, the paper shows that applying LoRA only to the query and value projections is sufficient to match the performance of full fine-tuning. By updating only a small subset of the model, LoRA significantly reduces memory usage associated with gradients and optimizer states. While it does not reduce activation memory, and introduces a very minimal overhead for storing A and B. Although LoRA introduces additional parameters and may seem to increase inference cost, this can be avoided: after fine-tuning, the low-rank component BA can be merged back into W_0, restoring the original model structure and maintaining inference efficiency.

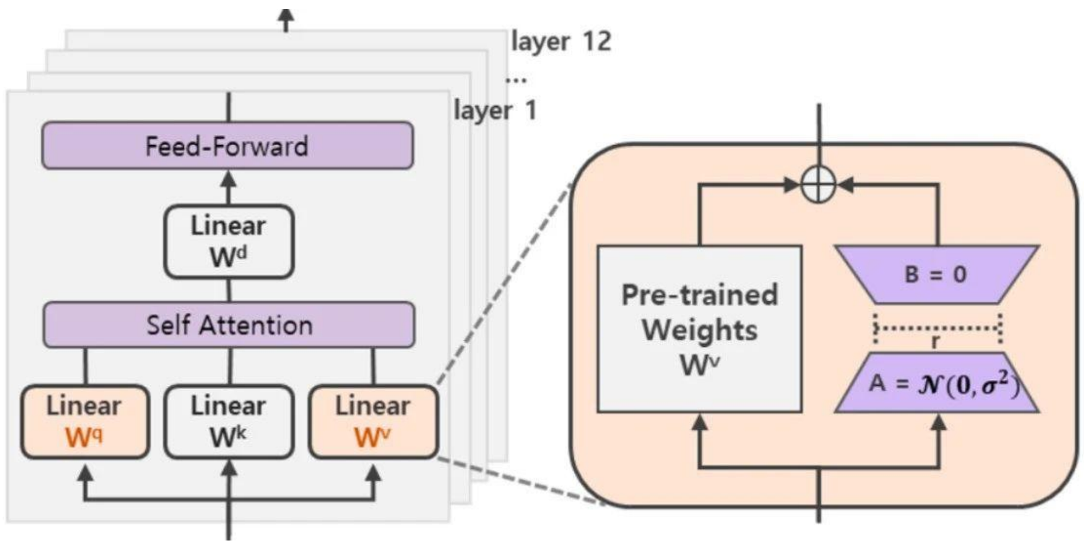

Figure 2: LoRA

4.5 Guidance

In this phase, your task is to perform instruction tuning on the **LLaMA3.2-1B** model using a small subset (200 samples) of the Alpaca dataset. By applying the optimization techniques discussed earlier, it becomes feasible to fine-tune large language models even on a single P100 GPU with 16GB of memory. To ensure a thorough understanding of the fine-tuning process, we will avoid using high-level libraries such as Hugging-Face, which abstract away many important details. Instead, all implementations will rely solely on PyTorch's built-in functions.

1. End-to-End Instruction Tuning Flow: Create a file named `finetuning.py`. Implement the complete instruction tuning workflow using PyTorch. For data preprocessing and construction of supervised data loaders, refer to the official Alpaca repository.

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

3. Gradient Accumulation and Mixed Precision Training: Implement gradient accumulation and mixed precision training using PyTorch's AMP package. Useful references include this blog post and the official AMP documentation.

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

5. Gradient Checkpointing: Use PyTorch's `torch.utils.checkpoint` API to insert gradient checkpoints. Choose which layers to apply checkpointing to and justify your decision in the report. Refer to this blog post for more information.

6. Model Fine-Tuning: Apply all techniques above to fine-tune the LLaMA3.2-1B model on the Alpaca dataset subset. The dataset is intentionally small for educational purposes, so training should only take a few minutes. The goal is not to achieve high model performance but to observe a decreasing loss, which indicates the setup is functioning correctly.

7. Hyperparameters: Use the SGD optimizer without momentum to minimize memory usage. Set `learning_rate = 1e-5`, `batch_size = 1`, and `gradient_accumulation_step = 8`. For the LoRA configuration, use `r = 16`, `alpha = 32`, and `dropout_rate = 0.05`.

Deliverable: Submit a PDF file named `phase_3.pdf` that includes the following: a summary of the changes made to the code, a log of the training loss over time, and a comparison of model outputs before and after fine-tuning using sample prompts.

In the same PDF file, provide the following analysis and measurements. For Table 2, fill in the blanks using the following notation: use ↑ to indicate an increase in resource usage, ↓ for a decrease, and − to denote no significant change. For Table 3, measure and report the peak memory usage (in MB) and the averaged runtime per training step (in seconds) during the fine-tuning process for each combination of techniques applied to the LLaMA3.2-1B model on the P100 GPU. When measuring runtime, exclude the time used for loading the model weights. If an out-of-memory error occurs, mark the corresponding entry with ×. You may disable gradient accumulation for this step and use a batch size of 1.

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

Table 2: Fine-Tuning System Performance Analysis

GC	OFF				ON			
MP	OFF		ON		OFF		ON	
LoRA	OFF	ON	OFF	ON	OFF	ON	OFF	ON
Peak Mem								
Runtime								

Table 3: Fine-Tuning System Performance Benchmark