CSE-234: Data Systems for Machine Learning, Winter 2025

17: LLM - 2: Scaling law, MoE

Lecturer: Hao Zhang *Scribe: Alon Lahav, Runpeng Jian, Yen Ting, Yutian Shi, Sidney Pan, Yujia Wang, Luning Yang, Shaolong Li, Shreyansh Joshi, Zhihang Li, Salma Wafa, Daniel Shi, Yesheng Liang, Luyu Han*

1 Recap

1.1 Compute – Where is the Potential Bottleneck?

We have discussed how to calculate the number of parameters and flops needed to train an LLM. In this lecture, we will focus on calculating the memory. Figure 1 illustrates the flops required for Transformer decoder layers. Before we begin our memory calculation discussion, let's address two important questions about this figure:

1. Why $\times 3$?
2. What happens when $b = 1$ and $s = 1$?

Answer:

1. We multiply the FLOPs by 3 because in training LLMs, the FLOPs in the backward pass are twice that of the forward pass in one complete iteration.
2. When $s = 1$, it means the LLM is decoding (inferencing), as it is computing on only 1 token. When $b = 1$, it means only one user is making a request. We notice that when b and s are very small numbers, the computational complexity is also small, which can lead to GPU under-utilization. This indicates that optimization for LLM inferencing is challenging due to these inherent constraints.

2 Calculate the memory needed to train an LLM

2.1 Composition of Memory Usage (Training)

The composition of memory usage is shown in Table 1

2.2 Holistic View of Transformers

1. If we checkpoint at the transformer boundary, what's the activation memory (except embeddings)?

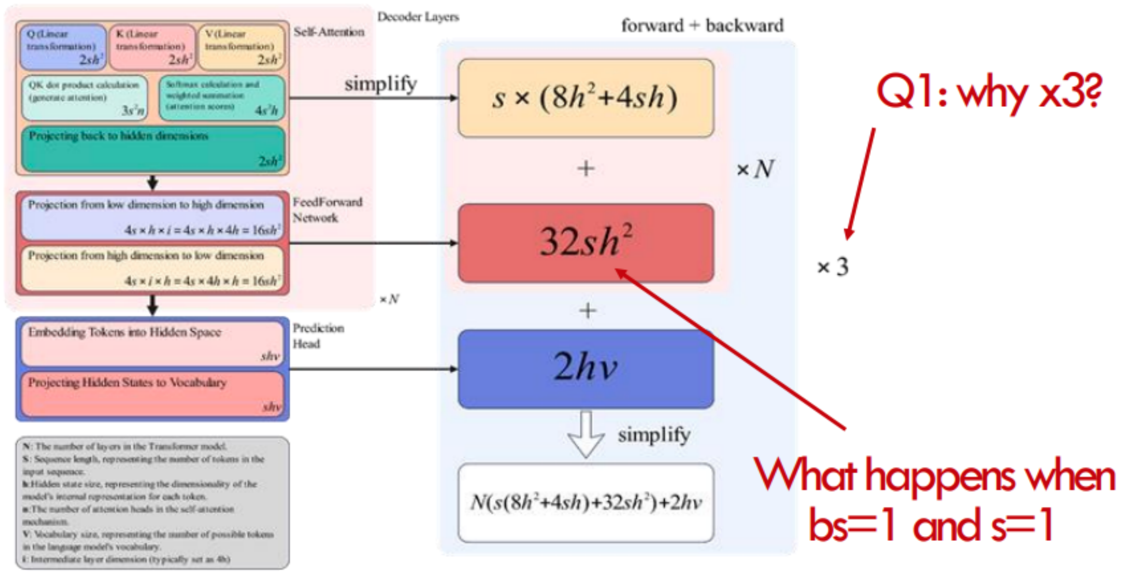
Decoder Layers
Self-Attention
Q (Linear transformation) $2sh^2$
K (Linear transformation) $2sh^2$
V (Linear transformation) $2sh^2$
QK dot product calculation (generate attention) $3s^2n$
Softmax calculation and weighted summation (attention scores) $4s^2h$
Projecting back to hidden dimensions $2sh^2$
FeedForward Network
Projection from low dimension to high dimension $4s \times h \times i = 4s \times h \times 4h = 16sh^2$
Projection from high dimension to low dimension $4s \times i \times h = 4s \times 4h \times h = 16sh^2$
$\times N$
Prediction Head
Embedding Tokens into Hidden Space shv
Projecting Hidden States to Vocabulary shv
N: The number of layers in the Transformer model.
S: Sequence length, representing the number of tokens in the input sequence.
h: Hidden state size, representing the dimensionality of the model's internal representation for each token.
n: The number of attention heads in the self-attention mechanism.
V: Vocabulary size, representing the number of possible tokens in the language model's vocabulary.
i: Intermediate layer dimension (typically set as 4h)
simplify
forward + backward
$s \times (8h^2+4sh)$
+
$32sh^2$
$\times N$
+
$2hv$
simplify
$N(s(8h^2+4sh)+32sh^2)+2hv$
$\times 3$
Q1: why x3?
What happens when bs=1 and s=1

Figure 1: Visualization of FLOPs needed for Transformer decoder layers

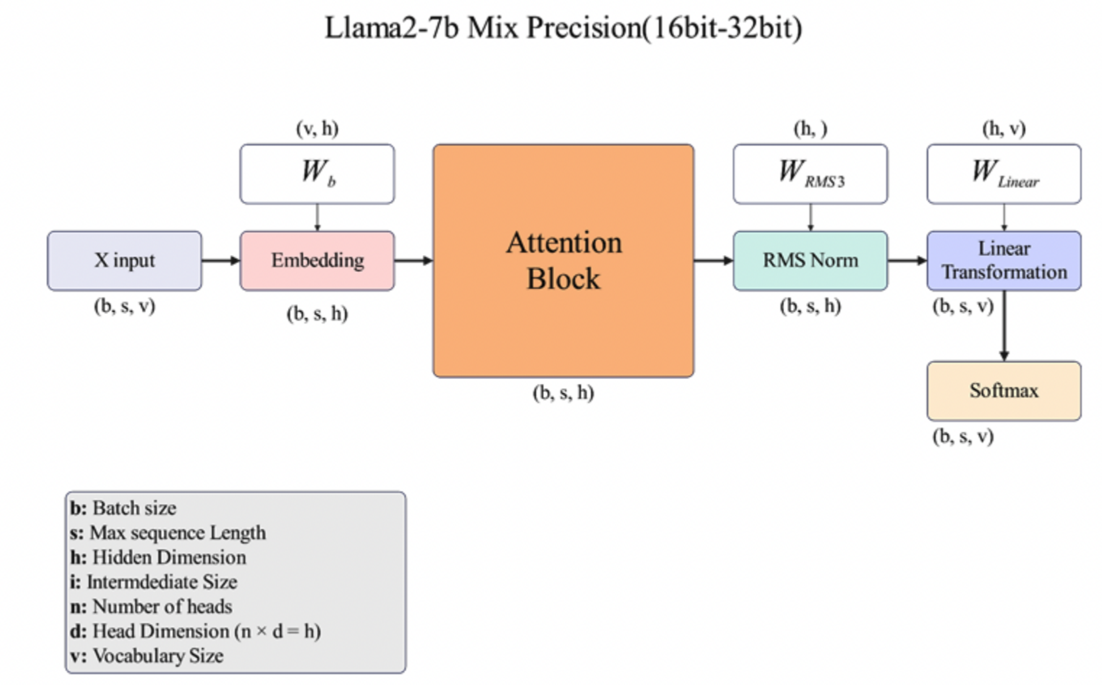

Figure 2: Llama2-7b Mix Precision(16bit-32bit)

Component	**Memory Requirement**
Model Weights	$2M$ (fp16)
Intermediate Activation Values	*Yet to be discussed*
Optimizer States	$12M$ (fp16-32 mixed precision)
	– Master copy (fp32)
	– Adam mean and variance (fp32)
Weight Gradients	$2M$ (fp16)
Activation Gradients	*Yet to be discussed*

Table 1: Memory requirements breakdown for LLM training, where M is the number of parameters.

2. If we equally assign layers to devices, what's the P2P communication overhead?

Answer:

1. At the end of each MLP, the output shape is (b, s, h), so the activation memory consumption is *number of layers* $\times$ bsh.
2. The P2P communication overhead corresponds to the activations exchanged between transformer blocks, which is bsh.

2.3 Attention Block

The attention block is both memory- and compute-bound due to the quadratic complexity of the sequence length when calculating softmax; however, this issue has been mitigated by FlashAttention.

3 Partition Along Sequence Length

3.1 Motivation

As we scale LLMs to handle increasingly longer sequences, the maximum sequence length (s) will grow significantly, making partitioning along s inevitable, despite its high cost. In tensor parallelism, scaling beyond eight GPUs is impractical due to the excessive number of all-reduce operations, which are computationally expensive. Without NVLink, these operations become even slower. As we push our model to scale beyond eight GPUs, partitioning along s will be necessary.

3.2 Attention VS MLP

Figure 4 illustrates the architecture of a sub-layer within a Transformer-type block. In this figure, the only dimension undergoing expansion and subsequent reduction is the hidden dimension $(h \to i \to h)$. The sequence dimension (s), along with the batch dimension (b), is carried through the matrix multiplications

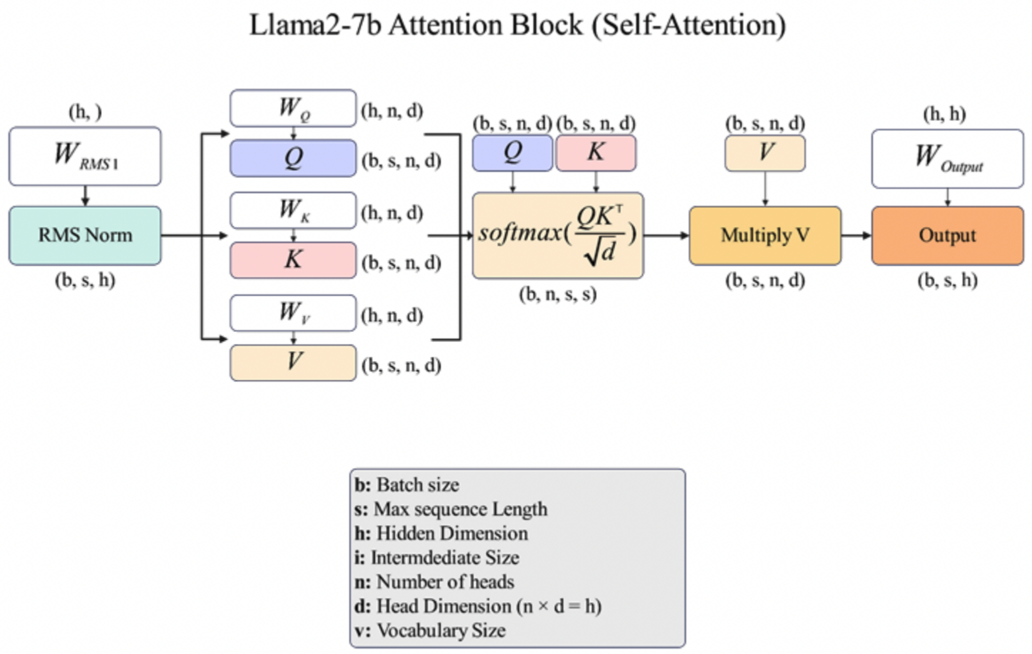

Figure 3: Llama2-7b Attention Block (Self-Attention)

without being summed out or contracted. Therefore, s is not involved in the reduction step. Here, “reduction” specifically refers to reducing the intermediate dimensionality (i) back to the hidden dimensionality (h). As a result, partitioning along s will not be prohibitively expensive.

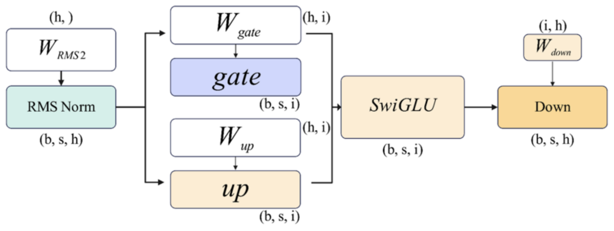

Figure 4: "feed-forward” (MLP) sub-layer of a Transformer-type block

Figure 5 illustrates the architecture of an attention block. In this figure, unlike the previous case, the sequence dimension (s) is directly involved in the reduction loop. A simple way to see this is that each query slice (a single token’s query vector) must interact with all tokens’ value vectors along the sequence dimension (s). In attention mechanisms, the output at each query position is a weighted sum over all positions, meaning a summation occurs along s.

This explains why s appears in the reduction loop: when multiplying the attention matrix softmax$(QK^\top/\sqrt{d})$ (shape (b, n, s, s)) by the value tensor V (shape (b, s, n, d)), a matrix multiplication is performed that reduces over the s dimension. Consequently, partitioning along s in this case would be computationally expensive.

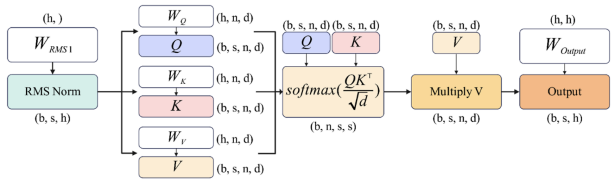

Figure 5: "feed-forward" (MLP) sub-layer of a Transformer-type block

3.3 Deepspeed Ulysses Sequence Parallelism

A recent method, DeepSpeed Ulysses Sequence Parallelism, focuses on partitioning the sequence dimension (s) in the MLP component while avoiding partitioning along s in the attention mechanism. Instead, they partition along the number of heads (n), assigning different attention heads to different devices and computing in parallel. Once the computation reaches the MLP stage, they switch the partitioning axis to s.

This partitioning strategy involves all-to-all communication because the first stage partitions along n, while the second stage partitions along s.

3.4 Extreme Case Analysis

What if the number of heads is much smaller than the number of GPUs? In that case, partitioning along the number of heads (n) becomes inefficient. As a result, we still need to partition along the sequence dimension (s). This means partitioning s in both the attention and MLP components—a strategy known as Ring Attention, designed for training ultra-long-context language models.

4 Scaling Law

4.1 Computing Requirements

Compute is a function of h, i, b, where h is the hidden dimension, i is the upper projection and down projection. b is the batch size. We also spent time on understanding that the number of parameters is a function of h and i. So h and i are essentially the model specifications, and b is essentially the number of data. Hence we can draw the conclusion that compute correlates with the number of parameters. The more parameters, the more compute we need to spent on training the model. Compute also correlates with the size of data that more training data needs more compute.

People always want to scale their model because of the common belief that larger model has better performance. But the problem here is that we don't have infinite compute and we are constrained to a compute budget (how much money we have). For example, if we have $5 million to deliver the best language model, how will we do that? We are facing these questions:

1. **Train models longer vs train bigger models**? (i.e. bigger model with fewer data or smaller model with more data?)
2. **Collect more data vs get more GPUs**? (Collecting money is non-trivial, should we spend money on it or we just spend money on GPUs keep scaling up?)

3. **How to choose the exact** h, i, etc.?

4.2 Motivation of Scaling Law

In the previous guest lecture, we already talked about a so called planning phase, which is a time consuming part might even be longer than the training. The decisions we faced that mentioned above are essentially the planning.

These questions are answered by the Scaling Law.

Given a fixed budget, we want to know:
1. **How large a model (detailed specs) should we train**?
2. **How many data should we use**?
3. Can we **predict the performance** if we choose a size of model and a size of data, subjecting to a compute budget($)?

4.3 In Traditional ML: Data Scaling Law

We consider a simple Gaussian setting:

$$x_1, x_2, \ldots, x_n \sim \mathcal{N}(\mu, \sigma^2),$$

and our task is to estimate the average

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

By standard results,

$$\mathbb{E}\big[(\hat{\mu} - \mu)^2\big] \;=\; \frac{\sigma^2}{n},$$

so the estimation error decreases with more data. In other words,

$$\log(\text{Error}) \;=\; -\log(n) + 2\log(\sigma),$$

and this serves as a **data scaling law** (e.g. an error rate of $1/n^{\alpha}$).

However, when we ask, "Can we do the same for large Transformer-based language models?" the answer is: **unfortunately, no.** The mathematical derivation above does not directly extend to machine learning models, which have non-convex neural architectures with many layers and parameters.

4.4 Mathematics vs. Physics

For simple Gaussian models, our mathematical analysis is straightforward. However, when it comes to large language models, our current mathematical toolkit cannot offer precise error bounds that scale with data size or model parameters in a closed-form manner.

A Physics-Like Approach. Instead of relying solely on formal proofs:

- ML scientists collect empirical evidence by running experiments and systematically scaling up model size, data, or compute.

- Then, they fit practical laws to link increased resources with improved performance.

This perspective underscores that modern LLMs, due to their complexity, are **not** amenable to the neat, closed-form guarantees found in simpler statistical models.

4.5 Scaling Law Example 1: Transformers vs LSTMs

Around five years ago, there was a strong debate in the machine learning community about whether transformers are superior to LSTMs. To answer this question, one brute-force approach would be to spend tens of millions to train an LSTM-based GPT-3 model with the same number of parameters and computational resources. However, this is infeasible due to the high cost. An alternative approach relies on empirical scaling laws: we formulate a hypothesis, train many small LSTMs and transformers, and compare their performance. Our hypothesis is that once we observe a trend, we can extrapolate it to larger models. To test this, researchers invested a substantial but feasible amount of computation to train multiple models with varying parameter sizes.

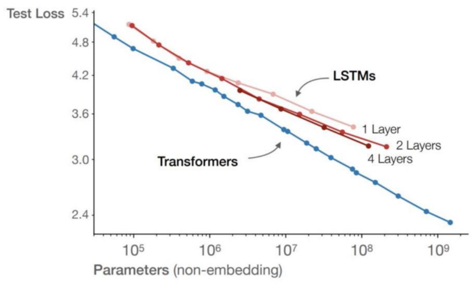

Figure 6: Transformers vs LSTMs Plot

Figure 6 illustrates an example of a scaling law, where, in the absence of an analytical understanding from a mathematical perspective, an empirical approach is employed with the hypothesis that the observed trend can be extrapolated to larger models. The research team trained models with up to 10^9 parameters, which, while not considered large today, was sufficiently large at the time of the experiment. Each point in the figure represents a model, and the researchers aimed to study the correlation between the number of parameters and the test loss. In a logarithmic scale, we observe that as the number of parameters increases, the test loss decreases, suggesting that the original relationship follows an exponential trend. They also trained the same number of transformers, visualized as a blue line in the plot. The blue curve (representing transformers) is consistently below the red curve (representing LSTMs), confirming that at this scale, transformers outperform LSTMs.

4.6 Scaling Law Example 2: Number of Layers

Similarly, we can analyze the correlation between test loss and various hyperparameters. For example, in Figure 7, the research team examined the relationship between test loss and the depth of neural networks.

4.7 Development of Scaling Law

This approach may resemble hyperparameter tuning, but it generalizes remarkably well. Researchers, particularly those from Google and OpenAI, have been studying this field since 2018. Initially, they conducted

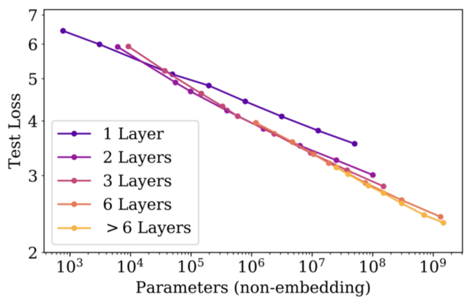

Figure 7: Number of Layers vs Testing Loss Plot

experiments on hyperparameters, but later, they began predicting the performance of larger models by running numerous experiments and analyzing the results. This allowed them to gather evidence to determine whether investing in training a model of a specific size would be worthwhile.

The most critical thing in the scaling law are the following:

1. **Number of parameters** - There is a hypothesis in machine learning and more parameters results in better performance, without loss of generality. However, it is important to note that there is no precise formulae or understanding of how many parameters would result in the best performance. In fact, the number of FLOPs is a very complex function of the number of parameters as seen previously - it's not possible to arbitrarily scale the number of parameters just to get the maximum FLOPs. This is something that needs to be finetuned very carefully.
2. **Amount of training data** - Again, as in the above case, it's not very trivial to decide how much training data should be there to train a model, as having too much data of the same type, containing the same information would be very expensive and would result in diminishing returns.

Essentially, the most important question at hand is, for a given budget, whether we want to train a larger model with lesser data or a smaller model with more data. To understand this better, lot of emperical experiments have been performed and it eventually boils down to solving this optimization problem.

$$N_{\text{opt}}(C), D_{\text{opt}}(C) = \underset{N,D \text{ s.t. } \text{FLOPs}(N,D)=C}{\arg\min} L(N, D).$$

In the above equation, L denotes the next-token prediction loss for our language model, N is the number of parameters in our language model, D is the amount of training data and C is the budget under which we have to operate and that's a function of N and D.

The goal here is to minimize the loss which would result in the best next-token prediction capability for our model. And for such minimized loss under the compute constraint C, what's the best possible N and D, that's what the above optimization problem is solving.

Through such empirical experiments, it has been observed that as the models become larger and larger and as the amount of data fed to the model increases, the next-token prediction losses decreases, thereby improving the model performance. As a result if we take the language model and evalute it on some academic benchmarks, the benchmark performance will improve a lot. This essentially, is the planning phase which is very critical for any scalable ML system.

Note: Next-token prediction refers to the ability of a language model to predict the next token, given all the previous tokens it has predicted along with the context it has (from previous answers or input prompts). This is fundamentally how LLMs operate.

Today's SoTA scaling law, also known as "Chinchilla scaling law" was released by DeepMind in 2022. Google invested a lot of computing power in this project to study the correlation between N, D and L.

5 Chinchilla Scaling Law

$$L(N, D) = \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.29}} + 1.69$$

Figure 8: Chinchilla equation

5.1 Equation

The Chinchilla Scaling Law can be summarized by the equation above, where N represents the number of parameters used to train the model (model size), D represents the amount of data used to train the model (data size), and L(N, D) represents the next-token prediction loss. Through thousands of experiments, Google DeepMind has determined specific values for these parameters. Given a transformer architecture, the relationship between next-token prediction loss, model size, and data size follows the equation above. This equation helps answer a key question in machine learning: given a fixed FLOP budget, how should one trade off model size and the number of training tokens?

5.2 Summary

Scaling laws serve as the "physics" of machine learning, marking a new era in ML research. Prior to scaling laws, ML researchers emphasized rigorous theoretical analysis and sought mathematical proofs. However, with the advent of scaling laws, researchers now rely more on empirical observations to develop practical laws, such as the Chinchilla Law. These laws are not limited to transformers but can also be applied to other models. To establish a scaling law for a given model, one must systematically vary the model size and data size to observe their effects on prediction loss. A model's scaling law provides insight into its efficiency. If a model achieves a lower loss with fewer parameters and less data, it has a better scaling law and is considered computationally efficient. Additionally, researchers have found that as model size N and data size D increase, model performance follows a predictable trajectory. For example, GPT-4.5 is approximately 10 times larger than GPT-4o.

6 MoE-based LLM

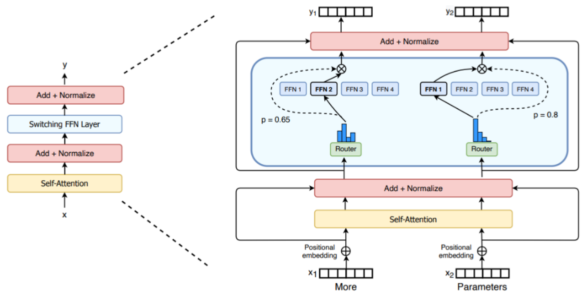

Figure 9: MoE model

6.1 MoE model

Unlike typical transformers, a Mixture of Experts (MoE) model contains multiple MLPs, each referred to as an "expert." The key idea behind MoE is that each expert specializes in predicting the correct answer for a subset of cases. In a typical transformer, after the normalization layer, the model proceeds to the MLP layers, consisting of up-projection and down-projection operations. In an MoE model, these projections are replicated multiple times, corresponding to the number of experts (e.g., FFN1-FFN4 in the figure above). Instead of forwarding the attention output directly to the MLP layers, MoE models first pass it through a router. The router analyzes the attention output and determines which expert should process it. If there is only one expert, the MoE model functions like a typical transformer.

6.2 A closer look at Mixture-of-Experts

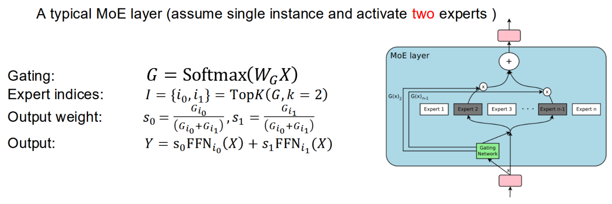
A typical MoE layer (assume single instance and activate two experts)
Gating: $G = \text{Softmax}(W_G X)$
Expert indices: $I = \{i_0, i_1\} = \text{Top}K(G, k = 2)$
Output weight: $s_0 = \frac{G_{i_0}}{(G_{i_0}+G_{i_1})}, s_1 = \frac{G_{i_1}}{(G_{i_0}+G_{i_1})}$
Output: $Y = s_0 \text{FFN}_{i_0}(X) + s_1 \text{FFN}_{i_1}(X)$
MoE layer
Expert 1
Expert 2
Expert 3
Expert n-1
Expert n
Gating Network

Figure 10: MoE layer

Gating score calculation is the first step. The gating score G is computed as the probability distribution over the product of input X and the learnable gating weight Wg. Applying a softmax function ensures that the gating scores form a valid probability distribution, summing to one and representing the importance of each expert. Next is expert selection. In the figure above, k = 2, meaning the top two experts with the highest gating scores are selected. Selecting only the top k experts ensures that the most relevant ones are used while significantly reducing computation. After selecting the top k experts, their gating scores are normalized again to ensure that the final output is a weighted average of these experts. The outputs of the selected experts are combined using their normalized gating scores s0 and s1, and only these outputs are forwarded to the next layer. By leveraging MoE, models can efficiently allocate computational resources, improving scalability and performance while keeping inference costs manageable.

7 Analysis of MOE

7.1 Details of MOE

We want to analyze the MOE just like what we have done before for the transformer. To do so, let's first take a closer look into the implementation of MOE. The figure below is a illustration of MOE architecture.

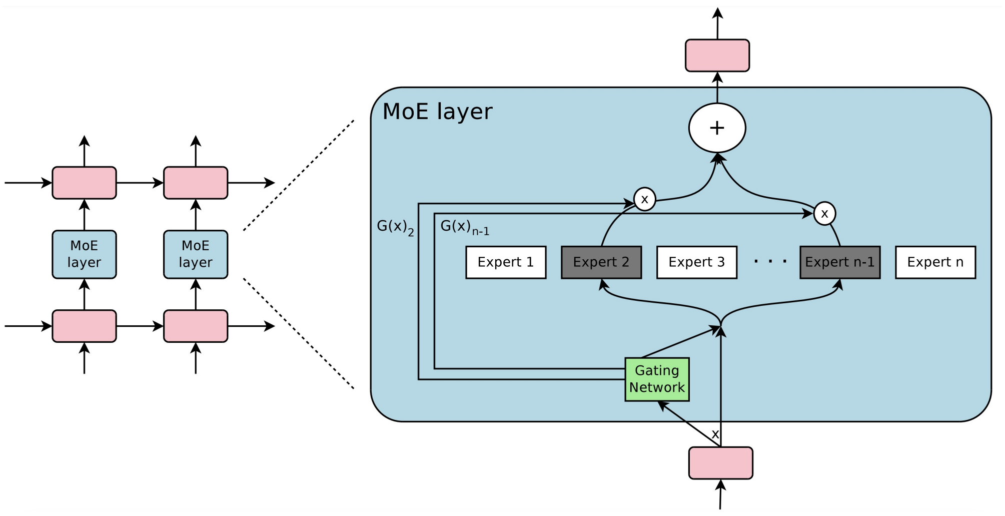

Figure 11: The illustration of MOE (mixture of experts) block in LLMs

It consists of a gating network and several experts. The gating network, or the router, is used to determine which expert to use. It is implemented by a softmax function. Say X is the input, W_G the parameter of the gating network, then the gate can be expressed as $G = \text{softmax}(W_G X)$, where G_i is the "score" of X_i. It is also the distribution over all the experts. Given G, we can implement numerous ways to select which expert to use. The method from the lecture is to select the indices of the top-k values of G, where k is a hyper-parameter. If $k = 1$, the we are basically selecting $i = \arg\max(G)$. Given the k indices $\{i_1, i_2, \ldots, i_k\}$, the output of the MOE block will then be computed by the weighted sum over the chosen experts:

$$Y = \sum_{j=1}^{k} s_j \text{FFN}_{i_j}(X), \quad s_j = \frac{G_{i_j}}{\sum_{p=1}^{k} G_{i_p}}$$

7.2 Analyzing the resource consumption of MOE

In the popular MOE based LLMs, the MLP is replicated by $N/2$. We know that in transformers MLP is the layer with the most number of parameters. Now, after replicating the MLP layers by $N/2$ times, the parameters in such LLMs are drastically increased (in DeepSeek, $N = 256$). So the memory used to store these parameters is drastically increased. But the memory needed for storing activations doesn't change a lot. This is because each token will only be routed to a few experts (usually 2 or 1 experts). For the same reason, the compute also doesn't increase much. So by using MOE blocks, we will get a model that has the size several times bigger than traditional LLMs, but has roughly the same compute.

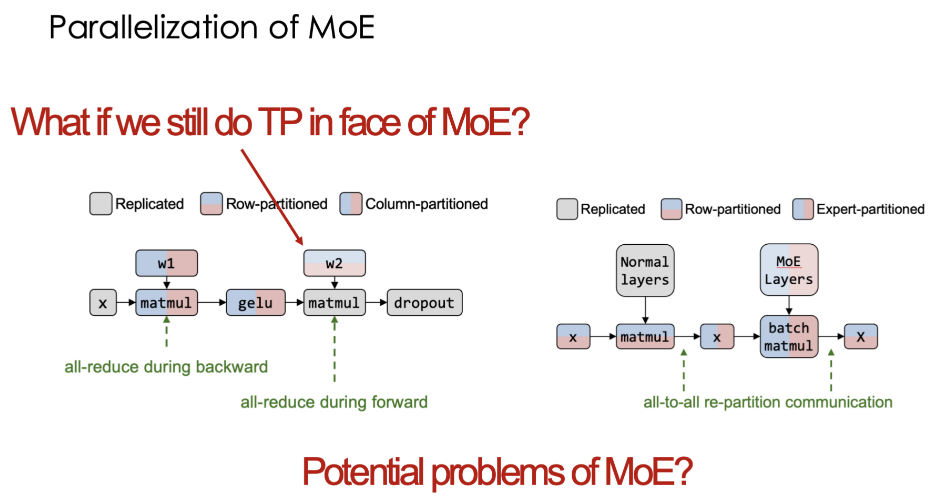

Figure 12: Parallelization of MoE

7.3 MOE from the scaling lay perspective

From the analysis above we can see that with MOE block, we can build a much larger model. Researchers has found that the increasing of parameters in MOE based LLMs leads to the increase in the performance. From this perspective, MOE based LLMs has a better scaling law than dense transformers, and are more compute-efficient (it is speculated that GPT-4 is MOE).

8 MoE-based LLMs

8.1 Why is Dense Transformer Important?

While increasing the number of experts in Mixture of Experts (MoE) models can improve performance, it introduces several challenges:

1. Memory Limitation: Increasing the number of experts drastically increases the number of parameters, leading to a significant memory requirement. Each parameter takes 2 bytes to store, causing memory usage to potentially explode.
2. Communication Overhead: In dense Transformers, tensor parallelism is used to partition weights (e.g., w_1, w_2 in the MLP). Switching from a dense Transformer to MoE requires replicating these weights multiple times across experts. This increases the cost of all-reduce operations, making them expensive and slow (Fig. 12).

8.2 Potential Problems of MoE

1. Partitioning: MoE models partition along the expert dimension, as each expert computes independently. Before entering the MoE module, partitions are along dimensions $S/H/B$, not the expert dimension.

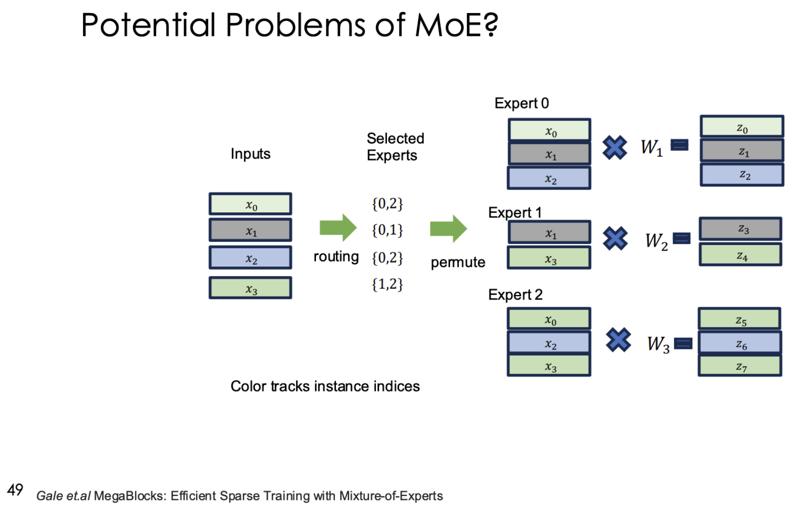
Potential Problems of MoE?
Inputs
Selected Experts
routing
permute
Expert 0
Expert 1
Expert 2
{0,2}
{0,1}
{0,2}
{1,2}
x_0 x_1 x_2 x_3
W_1 W_2 W_3
z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7
Color tracks instance indices
49 Gale et.al MegaBlocks: Efficient Sparse Training with Mixture-of-Experts

Figure 13: Potential Problems of MoE

Switching between partitions requires performing an all-to-all operation, which is costly.

2. Expert Imbalance: Different experts may process different numbers of tokens, leading to imbalanced workloads across GPUs. For example, in Fig. 13, if Expert 1 finishes early, other GPUs (e.g., hosting Experts 0 and 2) must wait, creating idle time or "bubbles". Some experts may become overloaded with tokens ("hot experts"), while others remain underutilized. Techniques, such as those used in DeepSeek, aim to balance the workload across experts.

In summary, there are three main challenges of MoE. (1) Increased Parameters: A significant rise in parameters requires equivalent memory to store the weights. (2) Communication Challenges: Expert parallelism replaces tensor parallelism, increasing communication overhead. (3) Expert Imbalance: Workload imbalance among GPUs introduces inefficiencies and requires additional research and engineering to mitigate.

9 Optimizing Language Models

To optimize language models, we focus on several key components:

1. **MLP Optimization**: MLP layers are basically matrix multiplication. We write good kernels to speedup matmul, and it's well studied.

2. **Attention Mechanism Optimization**: Attention mechanisms, particularly in Transformers, are computationally expensive. While we have good kernels for attention operations, further optimizations are needed, e.g. FlashAttention.

3. **Element-wise Operations**: Element-wise operations (e.g. softmax) are memory-bound and inefficient. To address this, we fuse them in order to increase arithmetic intensity.

10 Inference in Language Models

Language model inference is inherently slow and expensive. Unlike search engines, which provide instant results, language models generate tokens sequentially, leading to noticeable delays. For example, generating a few hundred tokens can take around 20 seconds, even when using powerful hardware like tens of A100 or H100 GPUs.

10.1 Autoregressive Decoding

Language models are token predictors. Given a sequence of tokens, the model predicts the next token based on the conditional probability distribution over the vocabulary. This process is repeated iteratively, with each new token being appended to the sequence and fed back into the model to predict the next token. This iterative process is known as **autoregressive decoding**.

Key points about autoregressive decoding:

- **Sequential Nature**: Tokens are generated one at a time, and each token depends on the previous ones. This sequential dependency makes parallelization difficult.
- **Forward Passes**: For each token generated, the model performs a full forward pass through all layers of the Transformer. This is computationally expensive, especially for large models.
- **Difference from Training**: During training, the entire sequence is known, and the model can compute the loss in parallel. In inference, the model must generate tokens sequentially without knowledge of future tokens.

10.2 Stopping scenarios of model inference

The inference process stops in two scenarios.

- The model reaches its maximum sequence length limit (eg. if the model only supports 2048 tokens, it will stop at the last token)
- Generates the specific token `<|end of sequence|>`, signaling completion.

10.3 Phases of Autoregressive Decoding

- **Pre-filling phase (0-th iteration)**: The model takes the prompt and process all input tokens at once.
- **Decoding phase (all other iteration)**: The model predicts and generates new tokens sequentially, with each token conditioned on all previous tokens.

10.4 Applications in language model inference

There are two applications in language model inference:

- **Serving**: Deploying the model on a server. Serving the model takes in many requests and receives online traffic. We need to minimize cost-per-query and this is reflected as **throughput**. If we map this scenario to computing, this has a large batch size. Maximizing the batch size saves the cost.

- **Inference**: Deploying the language model on your own machine. There are fewer request and low or offline traffic as you are the only user. We want to emphasize the **latency** so that we get the respond as soon as possible. Mapping to computing, this has batch size = 1.

  $$\text{latency} = \text{step latency} * \# \text{ steps}$$

  Speculative Decoding optimizes the Inference Scenario by minimizing the number of steps.

11 Contribution

- Runpeng Jian: Section 1-2.1
- Yen-Ting Lee: Section 2.1-2.3
- Alon Lahav: Organized group and reviewed sections 1-3.4
- Luning Yang: Section 3.1-3.4
- Sidney Pan: Section 4.1-4.2
- Yujia Wang: Section 4.3-4.4
- Yutian Shi: Section 4.5-4.7
- Shreyansh Joshi: Section 4.7-4.8
- Shoalong Li: Section 5-6.2
- Zhihang Li: Section 7-7.3
- Salma Wafa: Overall Revision and Contribution Section 4.7-7.3
- Daniel Shi: Section 10.2 -10.4