The Math Behind Transformer

Training Transformer LLMs

Learning goals

- Learn to calculate Transformer number of parameters
- Understand Transformer computation and memory load
- Learn about Flash Attentions
- Understand Scaling Laws and Chinchilla

LLM PARAMETERS: MAIN COMPONENTS

- Model parameters are half-precision (bfloat16) numbers of 2 bytes
- One block (decoder unit) consists of:
  - W_q, W_k, W_v matrices which are each $d_{model} \cdot n_{heads} \cdot d_{head}$ and project the input into the query, key, and value used in self-attention.
  - A W_0 matrix which is also $d_{model} \cdot n_{heads} \cdot d_{head}$ used on the output of self-attention, before the MLP (feedforward) layer
  - MLP weights, which are two matrices each of ${d_{model}}^2 \cdot 4$. Here the 4 is based on calculations and means that the MLP is 4 times the size of the model embedding dimension.
- In most architectures, $d_{model} = n_{heads} \cdot d_{head}$

LLM PARAMETERS: FORMULA

Combining the above, for one layer/block we get this formula:

$$P_{layer} = 3 \cdot d_{model} \cdot n_{heads} \cdot d_{head} + \cdot d_{model} \cdot n_{heads} \cdot d_{head} + 2 \cdot 4 \cdot {d_{model}}^2$$

$$P_{layer} = 4 \cdot d_{model} \cdot n_{heads} \cdot d_{head} + 8 \cdot {d_{model}}^2$$

$$P_{layer} = 4 \cdot d_{model} \cdot d_{model} + 8 \cdot {d_{model}}^2$$

$$P_{layer} = 12 \cdot {d_{model}}^2$$

For a LLM of n layers, we get:

$$P = 12 \cdot n_{layers} \cdot {d_{model}}^2$$

LLM PARAMETERS: EXAMPLE

GPT-3 Small has:

$$n_{params} = 125\,M\ ;\ n_{layers} = 12\ ;\ d_{model} = 768\ ;\ n_{heads} = 12\ ;\ d_{head} = 64$$

GPT-3 Medium has:

$$n_{params} = 350\,M\ ;\ n_{layers} = 24\ ;\ d_{model} = 1024\ ;\ n_{heads} = 16\ ;\ d_{head} = 64$$

Applying the above formula we get $\sim$85 M parameters for GPT-3 Small and $\sim$302 M parameters for GPT-3 Medium.

What are we missing...?!

LLM PARAMETERS: OTHER COMPONENTS

- Word Embedding parameters
- Position Embedding parameters
- Token Type Embedding parameters
- Embedding Layer Normalization, weight and Bias
- Other model-specific parameters

They do not scale with model size.

COMPUTE REQUIREMENTS

$C \approx \tau T = 6PD$

where:

- C is compute to train the model, in total floating point operations
- $C = C_{forward} + C_{backward}$
- $C_{forward} \approx 2PD$
- $C_{backward} \approx 4PD$
- τ is throughput of hardware: (No. GPUs) x (FLOPs/GPU)
- T is the time spent training the model, in seconds
- P is the number of parameters in the model
- D is the dataset size, in tokens

COMPUTE UNITS

C can be measured in different units:

- FLOP-seconds which is [Floating Point Ops / Second] x [Seconds]
  - We also use multiples GFLOP-seconds, TFLOP-seconds etc.
  - Other multiples like PFLOP-days are used in papers
  - 1 PFLOP-day = $10^{15} \cdot 24 \cdot 3600$ FLOP-seconds
- GPU-hours which is [No. GPUs] x [Hours]
  - GPU model is also required, since they have different compute capacities
  - For any GPU model, its Actual FLOPs are always lower than the advertised theoretical FLOPs

PARAMETER VS DATASET

- Model performance depends on number of parameters P, but also on number of training tokens D
- We need to decide about P and D, so that we get the best performance withing the compute budget
- The optimal tradeoff between P and D is: $D = 20P$
  - This is usually true for Chinchilla models, but not for all LLMs
- Training a LLM for less than 200 billion tokens is not recommended
- Rule of thumb: First determine the upmost inference cost, and then train the biggest model within that boundary.

MEMORY REQUIREMENTS

Common questions:

- How big is this model in bytes?
- Will it fit/train in my GPUs?

Model size components:

- Model parameters
- Optimizer states
- Gradients
- Activations

MODEL PARAMETERS

Parameter size depends on chosen representation:

- Pure fp32: $Mem_{model} = 4\ bytes/param \cdot N_{params}$
- fp16 or bf16: $Mem_{model} = 2\ bytes/param \cdot N_{params}$
- int8: $Mem_{model} = 1\ byte/param \cdot N_{params}$

It is practically common to use mixed representations:

- fp32 + fp16
- fp32 + bf16

OPTIMIZER STATES

AdamW: $Mem_{AdamW} = 12\ bytes/param \cdot N_{params}$

- fp32 copy of parameters: 4 bytes/param
- Momentum: 4 bytes/param
- Variance: 4 bytes/param

bitsandbytes (8-bit optimizer): $Mem_{AdamW} = 6\ bytes/param \cdot N_{params}$

- fp32 copy of parameters: 4 bytes/param
- Momentum: 1 byte/param
- Variance: 1 byte/param

SGD: $Mem_{AdamW} = 8\ bytes/param \cdot N_{params}$

- fp32 copy of parameters: 4 bytes/param
- Momentum: 4 bytes/param

GRADIENTS

They are usually stored in the same datatype as the model parameters.

Their memory overhead contribution is:

- fp32: $Mem_{grad} = 4\ bytes/param \cdot N_{params}$
- fp16 or bf16: $Mem_{grad} = 2\ bytes/param \cdot N_{params}$
- int8: $Mem_{grad} = 1\ byte/param \cdot N_{params}$

ACTIVATIONS

- GPUs are bottlenecked by memory, not FLOPs
- Save GPU memory by recomputing activations of certain layers
- Various schemes for selecting which layers to clear
- They take some extra memory, but save even more

Total memory when training **without** activations:
$Mem_{training} = Mem_{params} + Mem_{opt} + Mem_{grad}$

Total memory when training **using** activations:
$Mem_{training} = Mem_{params} + Mem_{opt} + Mem_{grad} + Mem_{activ}$

In the latter case, Mem_{params}, Mem_{opt} and Mem_{grad} are significantly smaller than in the former.

DISTRIBUTED TRAINING

- Training LLMs faster on many GPUs
- Avoiding OOM issues
- **Data parallelism:** split the data on different model replicas
- **Tensor parallellism:** split model parameters accross GPUs
- **Sharded optimizers:** reduce optimizer overhead by No. GPUs
    - ZeRO (Zero Redundancy Optimizer)
    - Requires low extra communication between GPUs
    - Decreases optimizer memory requirement
    - Improves training speed

THE $O(n^2)$ PROBLEM OF TRANSFORMER LLMs

Quadratic time & memory complexity of Self-Attention

- Inductive bias of Transformer models: Connect all tokens in a sequence to each other
- **Pro:** Can (theoretically) learn contexts of arbitrary length
- **Con:** Bad (quadratic)scalability limiting context size

Resulting Problems:

- Several tasks require models to consume longer sequences
  - Text summarization of documents
  - Sentiment analysis of documents
  - Classification of EEG trace of thousand time steps
  - Classification of coding or non-coding genes
  - ... many more from biology/medicine
- *Efficiency:* Are there more efficient (e.g., linear) implementations which achieve similar or even better performance?

EFFICIENT TRANSFORMERS

Broad overview on so-called "X-formers":

- Efficient & fast Transformer-based models
  $\rightarrow$ Reduce complexity from $\mathcal{O}(n^2)$ to (up to) $\mathcal{O}(n)$
- Claim on-par (or even) superior performance
- Different techniques used:
  - Fixed/Factorized/Random Patterns
  - Learnable Patterns (extension of the above)
  - Low-Rank approximations or Kernels
  - Recurrence (see e.g. ▸ Transformer-XL (Dai et al., 2019))
  - Memory modules

Side note:

- Most Benchmark data sets not explicitly designed for evaluating long-range abilities of the models.
- Recently proposed: ▸ *Longe Range Arena: A benchmark for efficient transformers* (Tay et al., 2020)

INTRODUCING PATTERNS

Reasoning:

- Making every token attend to every other token might be unnecessary
- Introduce sparsity in the commonly dense attention matrix

Example:

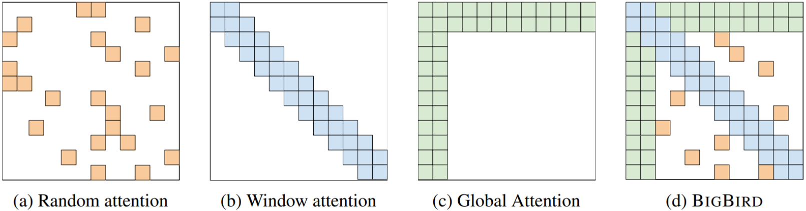


Figure 1: Building blocks of the attention mechanism used in BIGBIRD. White color indicates absence of attention. (a) random attention with $r = 2$, (b) sliding window attention with $w = 3$ (c) global attention with $g = 2$. (d) the combined BIGBIRD model.

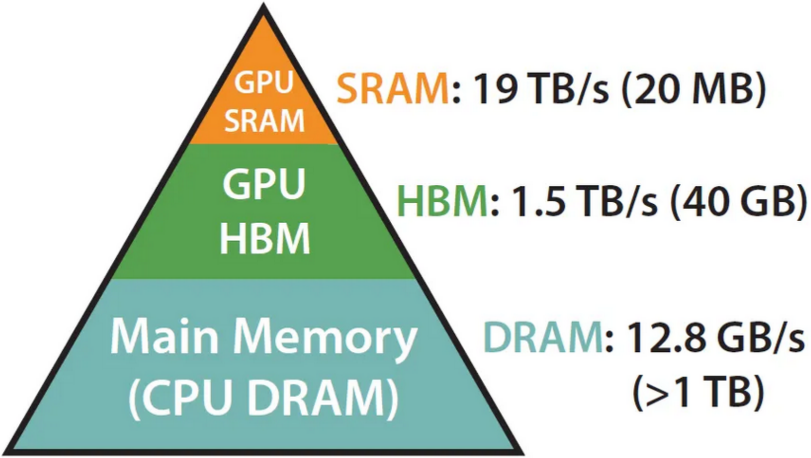
Source: Zaheer et al. (2020)

FlashAttention

Fast and Memory-Efficient Exact Attention with IO-Awareness

- Fast
  - 15 % faster than BERT
  - 3x faster than GPT-2
  - 2.4x faster than Megatron-LM
- Memory-efficient
  - Reducing from $O(n^2)$ to $O(n)$
- Exact
  - Same as “vanilla attention”, not an approximation
- IO aware
  - Reducing memory load/store operations

GPU MEMORY HIERARCHY

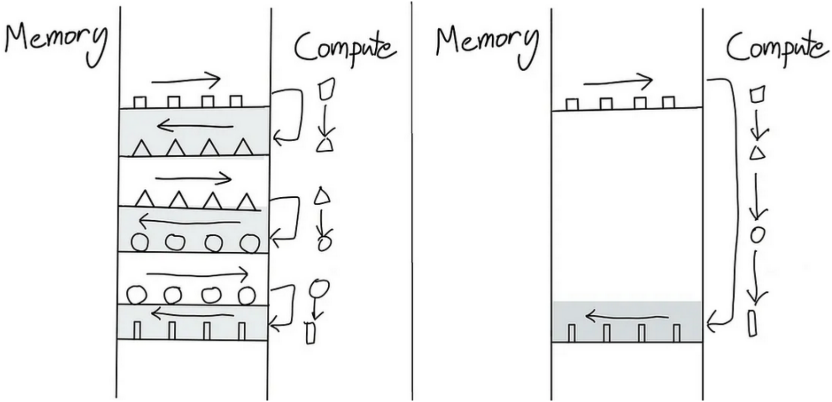


Source: Dao et al. (2022)

COMPUTING CONSIDERATIONS

- GPU compute has been growing faster than memory bandwidth
  - GPU has to wait for data
- Transformer operations are memory-bound
  - Elementwise operations with high memory access
- IO aware means reducing memory load/store operations
- FlashAttention implements the following:
  - Operation fusion to reduce memory access
  - Tiling or chunking the softmax matrix into blocks
  - Recomputation for better memory utilization

OPERATION FUSION



Source: `https://horace.io/brrr_intro.html`

LIMITATIONS AND PROSPECTS

- FlashAttention requires writing attention to CUDA language
  - A new CUDA kernel for each new attention implementation
  - CUDA is lower-level than PyTorch
  - Implementation may not be transferable accross GPUs
- Towards IO-Aware Deep Learning
  - Extending beyonde attention
- Multi-GPU IO-Aware Methods
  - FlashAttention computation may be parallelizable accross multiple GPUs

SCALING LAWS

Kaplan et al. (2020)

- Performance depends strongly on scale, weakly on model shape
  - Scale means: parameters N, data D, and compute C
  - Shape means: depth and width
- Smooth power laws
  - Performance has power-law relation with each factor N, D, C
  - When not bottlenecked by the other two
  - Trend spanning more than six orders of magnitude
- Universality of overfitting
  - Performance enters regime of diminishing returns if N or D held fixed while the other increases
  - Performance penalty depends on $N^{0.74}/D$

SCALING LAWS

- Universality of training
  - Training curves follow predictable power-laws
  - Their parameters are roughly independent of model size
  - It is possible to predict by extrapolating the early part of the training curve
- Transfer improves with test performance
  - When evaluating on text with different distribution from training text, results are strongly correlated to those on the validation set
  - Transfer to different distribution incurs a constant penalty but improves in line with performance on training set
- Sample efficiency
  - Large models are more sample-efficient than small models
  - They reach same performance with fewer optimization steps

SCALING LAWS

- Convergence is inefficient
  - When C is fixed but N and D are not, optimal performance is achived by training very large models and stopping significantly short of convergence
- Optimal batch size
  - Ideal size is a power of the loss only
  - It is $\sim$1-2 million tokens for the largest models we can train

Larger language models will perform better and be more sample efficient than current models.

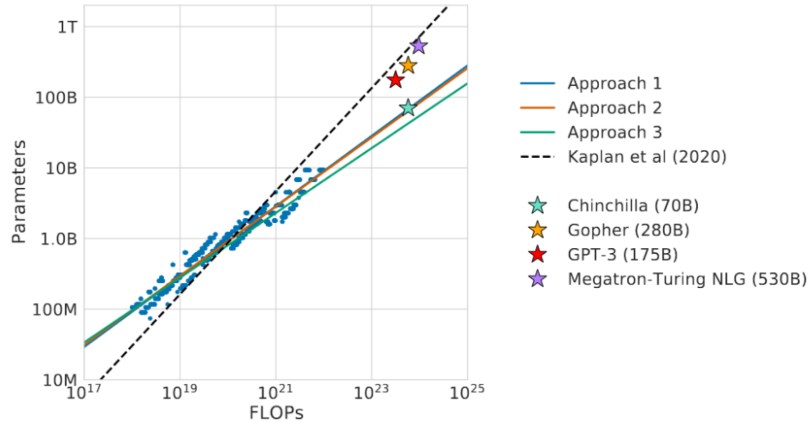
COMPUTE-OPTIMAL LLMs

Hoffmann et al. (2022)

Given a fixed FLOPs budget, how should we trade-off model size and text size to optimize performance?

- Find N and D so that $FLOPs(N, D) = C$ and $L(N, D)$ is minimal
- Empirically estimated N and D based on 400 models.
    - Ranging from 70 M to 16 B parameters
    - Trained on 5 B to 400 B tokens
- Different results from those of Kaplan et al. (2020)
- Results verified using Chinchilla
    - Chinchilla has 70 B parameters and is trained on 1.4 T tokens
    - 4x less parameters and 4x more tokens than Gopher
    - Chinchilla outruns Gopher and has reduced memory footprint and inference cost

COMPUTE-OPTIMAL LLMs



Source: Hoffmann et al. (2022)

CHINCHILLA AND THE OTHER LLMs

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

Source: Hoffmann et al. (2022)

Model	**Layers**	**Number Heads**	**Key/Value Size**	$\mathbf{d_{model}}$	**Max LR**	**Batch Size**
Gopher 280B	80	128	128	16,384	4×10^{-5}	$3M \rightarrow 6M$
Chinchilla 70B	80	64	128	8,192	1×10^{-4}	$1.5M \rightarrow 3M$

Source: Hoffmann et al. (2022)

CHINCHILLA ON MMLU

Random	25.0%
Average human rater	34.5%
GPT-3 5-shot	43.9%
Gopher 5-shot	60.0%
***Chinchilla* 5-shot**	**67.6%**
Average human expert performance	*89.8%*
June 2022 Forecast	57.1%
June 2023 Forecast	63.4%

Source: Hoffmann et al. (2022)

CHINCHILLA ON QA

	Method	*Chinchilla*	*Gopher*	GPT-3	SOTA (open book)
Natural Questions (dev)	0-shot	16.6%	10.1%	14.6%	54.4%
	5-shot	31.5%	24.5%	-	
	64-shot	35.5%	28.2%	29.9%	
TriviaQA (unfiltered, test)	0-shot	67.0%	52.8%	64.3 %	-
	5-shot	73.2%	63.6%	-	
	64-shot	72.3%	61.3%	71.2%	
TriviaQA (filtered, dev)	0-shot	55.4%	43.5%	-	72.5%
	5-shot	64.1%	57.0%	-	
	64-shot	64.6%	57.2%	-	

Source: Hoffmann et al. (2022)