THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

LLM Pretraining

COMP6211J

Binhang Yuan

Overview

- What is a language model?
- Tokenization:
  - How do we represent language to machines?
- Model categorization:
  - Encoder-only, decoder-only, encoder-decoder.
- Training objectives:
  - How are large language models (LLM) trained?
- Transformer architecture:
  - The main innovation that enabled large language models.
- Scaling law：
  - The relationship between computation load, data volume, and model scale.

Language Model

What Is a Language Model?

- The classic definition of a ***language model (LM)*** is a probability distribution over sequences of tokens.
- Suppose we have a vocabulary $\mathcal{V}$ of a set of tokens.
- A language model P assigns each sequence of tokens $x_1, x_2, \ldots, x_L \in \mathcal{V}$ to a probability (a number between 0 and 1): $p(x_1, x_2, \ldots, x_L) \in [0,1]$.
- The probability intuitively tells us how "good" a sequence of tokens is.
  - For example, if the vocabulary is $\mathcal{V} = \{\text{ate}, \text{ball}, \text{cheese}, \text{mouse}, \text{the}\}$, the language model might assign:

$$p(\text{the}, \text{mouse}, \text{ate}, \text{the}, \text{cheese}) = 0.02$$
$$p(\text{the}, \text{cheese}, \text{ate}, \text{the}, \text{mouse}) = 0.01$$
$$p(\text{mouse}, \text{the}, \text{the}, \text{chesse}, \text{ate}\ ) = 0.0001$$

Language Model Generation

- A language model P takes a sequence and returns a probability to assess its goodness.
- We can also generate a sequence given a language model.
- The purest way to do this is to sample a sequence $x_{1:L}$ from the language model P with probability equal to $p(x_{1:L})$ denoted:

$$x_{1:L} \sim P$$

Autoregressive Language Models

- A common way to write the joint distribution $p(x_{1:L})$ of a sequence to $x_{1:L}$ is using the *chain rule of probablity*:

$$p(x_{1:L}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \dots p(x_L|x_{1:L-1}) = \prod_{i=1}^{L} p(x_i|x_{1:i-1})$$

- In particular, $p(x_i|x_{1:i-1})$ is a conditional probability distribution of the next token x_i given the previous tokens $x_{1:i-1}$.
- An autoregressive language model is one where each conditional distribution $p(x_i|x_{1:i-1})$ can be computed efficiently (e.g., using a feedforward neural network).

Tokenization

Tokenization

- Recall: language model P is a probability distribution over a sequence of tokens where each token comes from some vocabulary $\mathcal{V}$, e.g.,:

  [I, love, cats, and, dogs]

- Natural language doesn't come as a sequence of tokens, but as just a string (concretely, sequence of Unicode characters):

  I love cats and dogs

- A ***tokenizer*** converts any string into a sequence of tokens:

  I love cats and dogs ⟹ [I, love, cats, and, dogs]

Split by Space

- The simplest solution is to do: text.split(' ')
- This doesn't work for languages such as Chinese, where sentences are written without spaces between words:
  - 我今天去了商店: [I went to the store today.]
- Then there are languages like German that have long compound words:
  - Abwasserbehandlungsanlange: [Wastewater treatment plant]
- Even in English, there are hyphenated words (e.g., father-in-law) and contractions (e.g., don't), which should get split up.

What Makes a Good Tokenization?

- We don’t want too many tokens:
  - The extreme case characters or bytes;
  - The sequence becomes difficult to model.
- We don’t want too few tokens:
  - There won’t be parameter sharing between words (e.g., should mother-in-law and father-in-law be completely different)?
  - This is especially problematic for morphologically rich languages (e.g., Arabic, Turkish, etc.).
- Each token should be a linguistically or statistically meaningful unit.

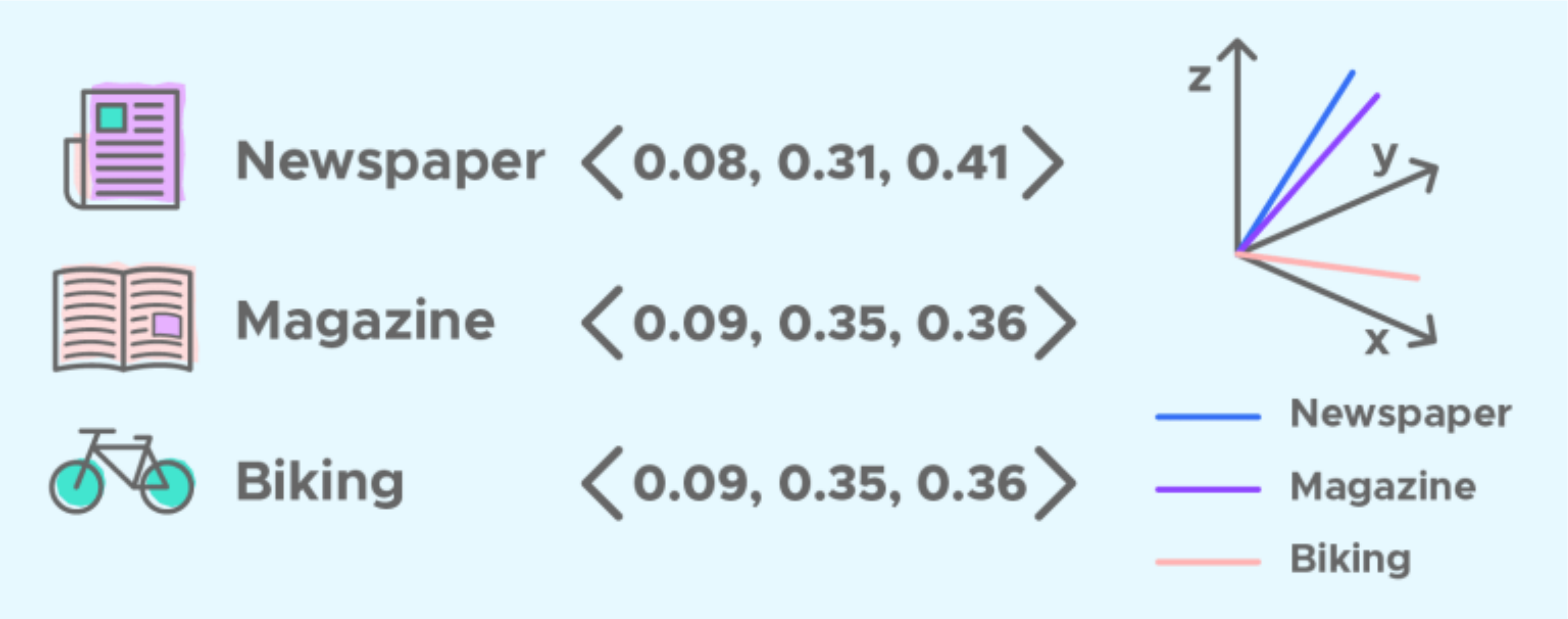
Some Encoding Methods

- Byte pair encoding (BPE)
  - Start with each character as its own token and combine tokens that co-occur a lot.
  - https://arxiv.org/pdf/1508.07909.pdf
- Unigram model (SentencePiece):
  - Rather than just splitting by frequency, a more "principled" approach is to define an objective function that captures what a good tokenization looks like.
  - https://arxiv.org/pdf/1804.10959.pdf

Representation: Word as Vectors

- Tokens can be represented as number index:

  [I, love, cats, and, dogs] ⟹ [328, 793, 3989, 537, 3255, 269]
- But indices are also meaningless.
- Represent words in a vector space
  - Vector distance ⟹ similarity。


Newspaper ⟨0.08, 0.31, 0.41⟩
Magazine ⟨0.09, 0.35, 0.36⟩
Biking ⟨0.09, 0.35, 0.36⟩
z
y
x
Newspaper
Magazine
Biking

RELAXED
SYSTEM LAB

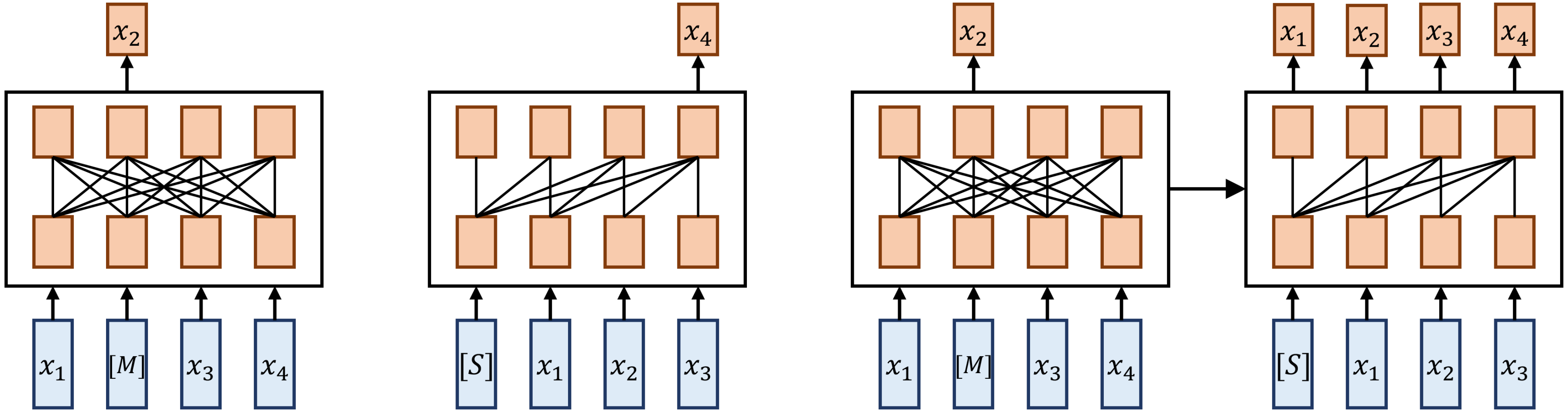
LLM Categorization

Contextual Embeddings

- Language model:
  - Associate a sequence of tokens with a corresponding sequence of contextual embeddings.
- Embedding function (analogous to a feature map for sequences):
  - $\emptyset\colon \mathcal{V}^L \rightarrow \mathbb{R}^{L\times D}$
  - A token sequence $x_{1:L}[x_1, x_2, \dots, x_L] \in \mathcal{V}^L$
  - Map to $\emptyset(x_{1:L}) \in \mathbb{R}^{L\times D}$
- For example, if $D = 2$:
  - [I, love, cats, and, dogs] ⟹ [328, 793, 3989, 537, 3255, 269] $\Longrightarrow \begin{bmatrix} (0.2, 0.3) \\ (0.8, 0.7) \\ (0.2\ 0.1) \\ (0.0,0.7) \\ (0.1, 0.0) \\ (0.1, 0.4) \end{bmatrix}$

Types of language models

- Encoder-only models (BERT, RoBERTa, etc.)
- Encoder-decoder models (BART, T5, etc.)
- **Decoder-only models** (GPT-3, Llama-3, Deepseek-V3, etc.)



Encoder-only Decoder-only Encoder-decoder

Encoder-only Models

- Encoder-only models produce contextual embeddings but cannot be used directly to generate text:

$$x_{1:L} \Rightarrow \phi(x_{1:L})$$

- These contextual embeddings are generally used for classification tasks (sometimes boldly called natural language understanding tasks).
  - Example: sentiment classification: [[CLS],the,movie,was,great] ⇒ positive.
- Pros:
  - Contextual embedding for x_i can depend bidirectionally on both the left context ($x_{1:i-1}$) and the right context ($x_{i+1:L}$).
- Cons:
  - Cannot naturally generate completions.
  - Requires more ad-hoc training objectives (masked language modeling).

Decoder-only Models

- Decoder-only models are our standard autoregressive language models.
- Given a prompt $x_{1:i}$ produces both contextual embeddings and a distribution over next tokens x_{i+1}, and recursively, over the entire completion $x_{i+1:L}$:

$$x_{1:i} \Rightarrow \emptyset(x_{1:i}), p(x_{i+1} | x_{1:i})$$

- Example: text autocomplete
  - [[CLS],the,movie,was]⇒great
- Pro:
  - Can naturally generate completions.
  - Simple training objective (maximum likelihood).
- Con:
  - Contextual embedding for x_i can only depend **unidirectionally** on both the left context ($x_{1:i-1}$).

Encoder-decoder Models

- Encoder-decoder models can be the best of both worlds: they can use bidirectional contextual embeddings for the input $x_{1:L}$ and can generate the output $y_{1:L}$:

$$x_{1:L} \Rightarrow \emptyset(x_{1:L}), p(y_{1:L}|\emptyset(x_{1:L}))$$

- Example: table-to-text generation
  - [name,:,Clowns,|,eatType,:,coffee,shop]⇒[Clowns,is,a,coffee,shop].
- Pro:
  - Can naturally generate outputs.
- Con:
  - Requires more ad-hoc training objectives.

LLM Training Objectives

Decoder-only Model Training Objectives

- Recall that an autoregressive language model defines a conditional distribution: $p(x_i|x_{1:i-1})$
- Define it as follows:
    - Map $x_{1:i-1}$ to contextual embedding $\emptyset(x_{1:i-1}) \in \mathbb{R}^{(i-1)\times D}$;
    - Apply an embedding matrix $E \in \mathbb{R}^{D\times|\mathcal{V}|}$ to obtain scores for each token $\emptyset(x_{1:i-1})_{i-1}E \in \mathbb{R}^{|\mathcal{V}|}$ (where $\emptyset(x_{1:i-1})_{i-1} \in \mathbb{R}^{D}$);
    - Exponentiate and normalize it to produce the distribution over x_i.
- Put them together:

$$p(x_{i+1}|x_{1:i}) = \mathrm{softmax}(\emptyset(x_{1:i})_i E)$$

Decoder-only Model Training Objectives

- Maximum likelihood. Let θ be all the parameters of large language models.
- Let $\mathcal{D}$ be the training data consisting of a set of sequences. We can then follow the maximum likelihood principle and define the following negative log-likelihood objective function:

$$\mathcal{O}(\theta) = \sum_{x_{1:L \in \mathcal{D}}} -\log p_\theta(x_{1:L}) = \sum_{x_{1:L \in \mathcal{D}}} \sum_{i=1}^{L} -\log p_\theta(x_i | x_{1:i-1})$$

- Then we can use the SGD optimizers we have talked to optimize this loss function.

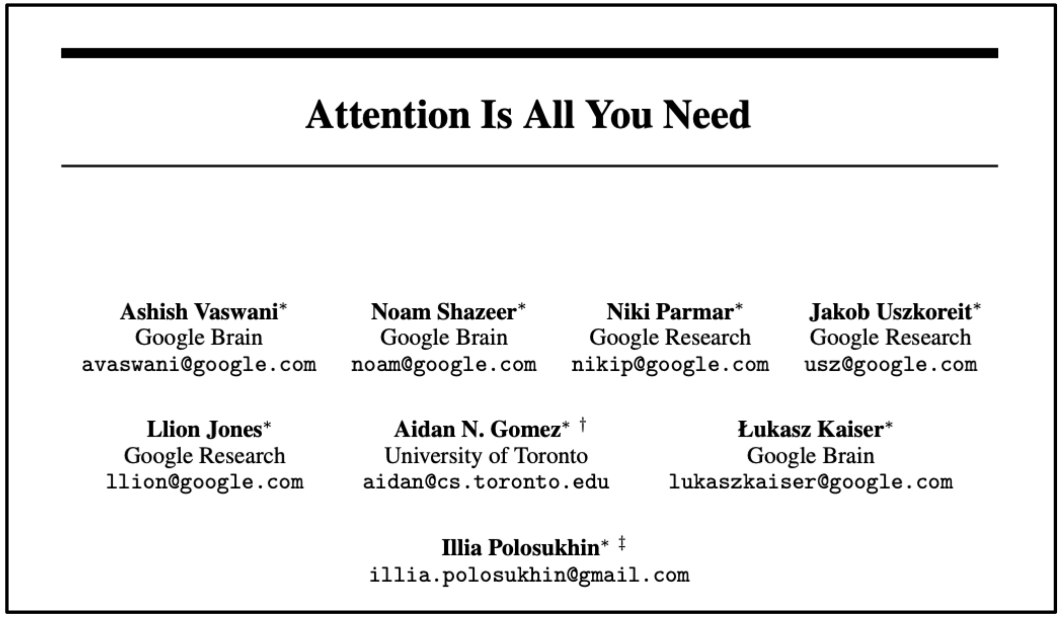
LLM Architecture Details

EmbedToken

- Convert sequences of tokens into sequences of vectors.
- EmbedToken does exactly this by looking up each token in an embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times D}$, a parameter that will be learned from data.
- $\text{EmbedToken}(x_{1:L}: \mathcal{V}^L) \rightarrow \mathbb{R}^{L \times D}$:
  - Turns each token x_i in the sequence $x_{1:L}$ into a vector $E_{x_i} \in \mathbb{R}^D$;
  - Return $\left[E_{x_1}, E_{x_2}, \dots, E_{x_L}\right]$.
- These are *context-independent* word embeddings.
- Next the TransformerBlock(s) takes these context-independent embeddings and maps them into contextual embeddings.

TransformerBlock

- TransformerBlock(s) takes these context-independent embeddings and maps them into contextual embeddings.
- TransformerBlocks$(X_{1:L}: R^{L\times D}) \rightarrow \mathbb{R}^{L\times D}$:
  - Process each element $X_i \in \mathbb{R}^D$ in the sequence $X_{1:L} \in R^{L\times D}$ with respect to other elements.
- TransformerBlock(s) are the building blocks of decoder-only (GPT-2, GPT-3), encoder-only (BERT, RoBERTa), and decoder-encoder (BART, T5) models.



Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

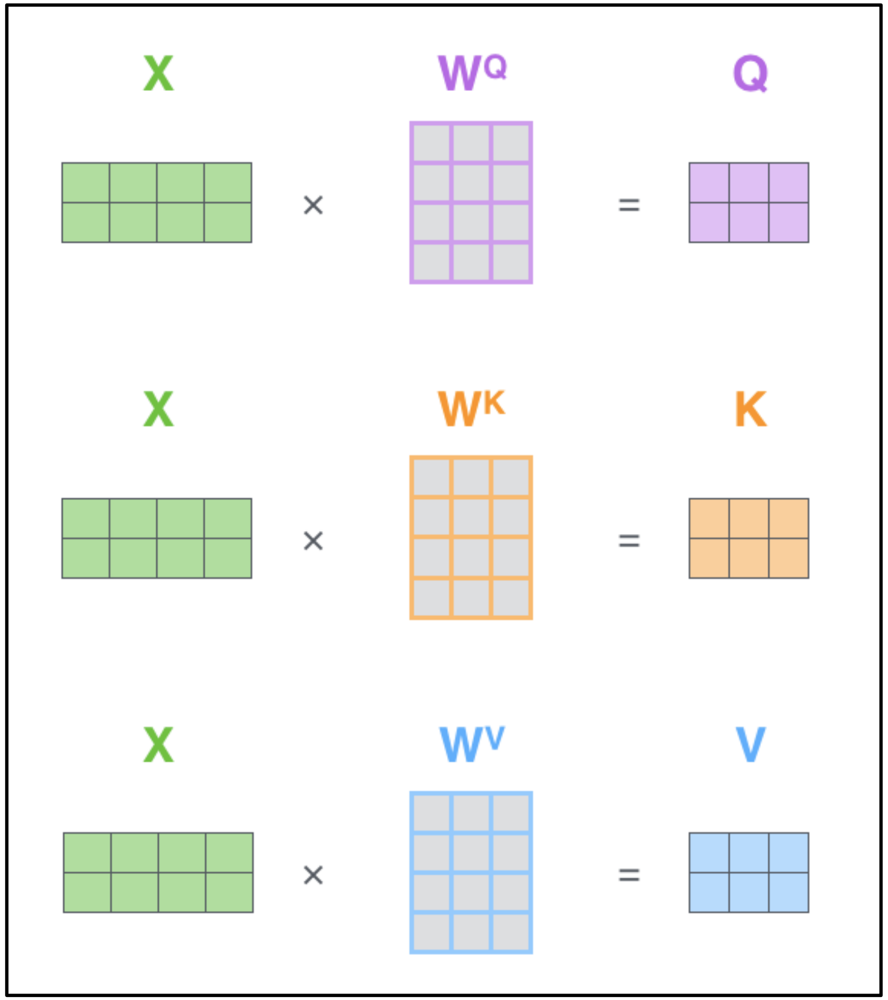
Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

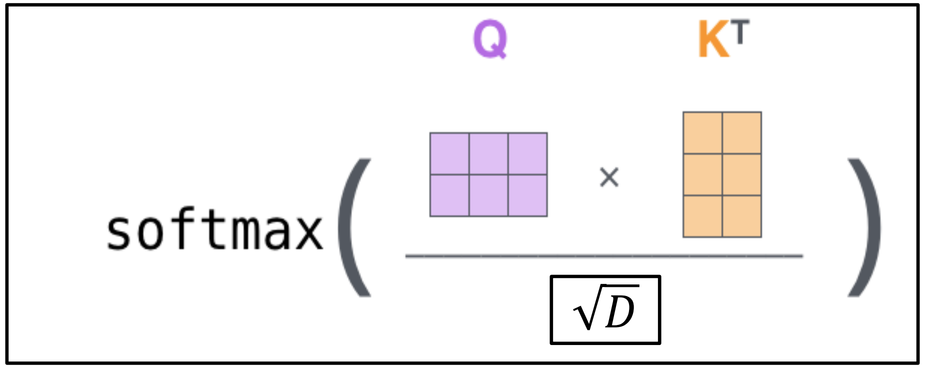
Attention Mechanism-1

- **<u>First step</u>**: in each transformer block, for each token, we create a query vector, a key vector, and a value vector by multiplying the embedding by three weight matrices.
- Formally, for each token $X_i \in \mathbb{R}^D$:
  - Query: $Q_i = X_i \times W^Q$
  - key: $K_i = X_i \times W^K$
  - Value: $V_i = X_i \times W^V$
- In the tensor representation for sequence $X_{1:L} \in \mathbb{R}^{L \times D}$:
  - Query: $Q = Q_{1:L} = X_{1:L} \times W^Q$
  - key: $K = K_{1:L} = X_{1:L} \times W^K$
  - Value: $V = V_{1:L} = X_{1:L} \times W^V$



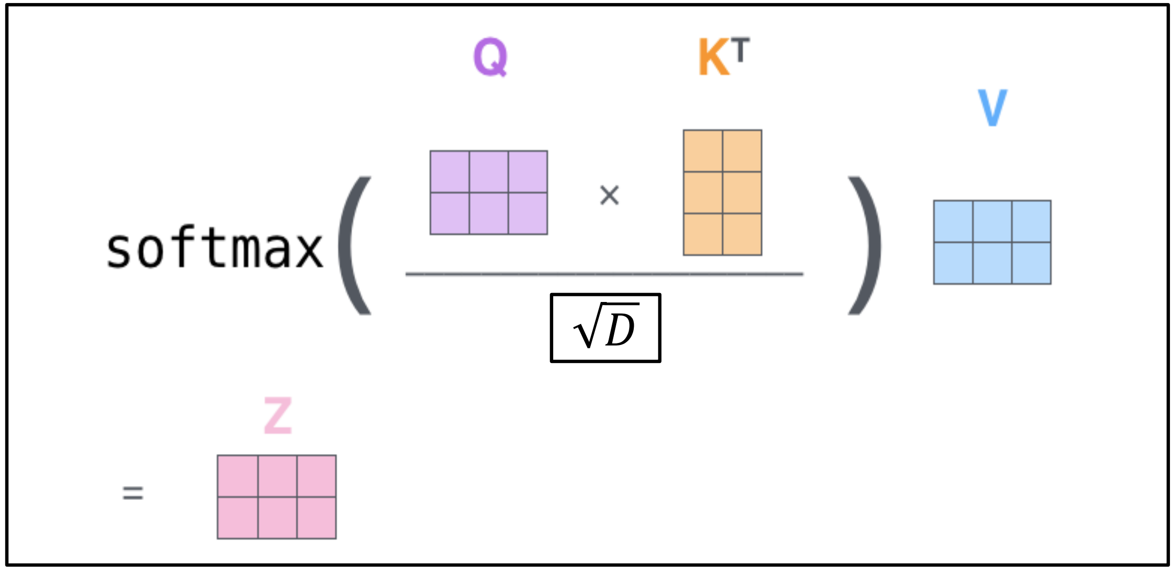
Attention Mechanism-2

- **<u>Second step</u>**: Calculate a score determining how much focus to place on other parts of the input sentence as we encode a token at a certain position.
- Calculated by:
  - Taking the dot product of the query vector with the key vector of the respective word we're scoring;
  - Divide the scores by the square root of the dimension of the key vectors;
  - Conduct a softmax operation.
- score = softmax($\frac{QK^T}{\sqrt{D}}$)



Attention Mechanism-3

- **Third step**: combine the value and the score.
  - $Z = \text{att} = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V$
- **Multi-head Attention**: there can be multiple aspects (e.g., syntax, semantics) we would want to match on.
- To accommodate this, we can simultaneously have **multiple attention heads (e.g. n_H heads)** and simply combine their outputs, e.g:
  - $Z = [\text{att}^1, \text{att}^2, \dots, \text{att}^{n_H}]$
- The attention output will be:
  - $\text{Out} = ZW^O$



Feedforward Layer

- After the attention layer, the output is put to a feed-forward neural network, then sends out the output upwards to the next encoder.
  - $\mathrm{X}'_{1:L} = \mathrm{relu}(\mathrm{Out}W^1)W^2$
  - W^1, W^2 are two weight matrices;
  - $X'_{1:L}$ is the output embedding for the current layer and the input of the next layer.
- Summarize a common weight dimension in one TransformerBlock:
  - Attention layer: $W^Q, W^K, W^V, W^O \in \mathbb{R}^{D\times D}$
  - Feedforward layer: $W^1 \in \mathbb{R}^{D\times 4D}$, $W^2 \in \mathbb{R}^{4D\times D}$

TransformerBlocks$(X \in R^{L\times D}) \to X' \in \mathbb{R}^{L\times D}$

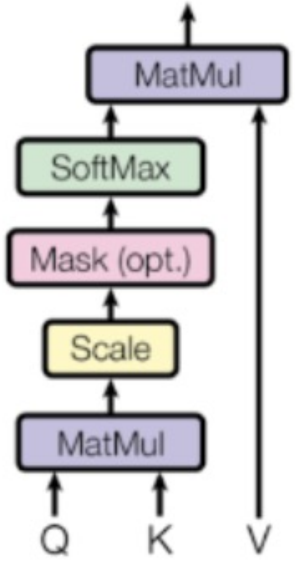
- L is the sequence length;
- D is the model dimension;
- Multi-head attention: $D = n_H \times H$
- H is the head dimension;
- n_h is the number of heads.

Computation	Input	Output
$Q = XW^Q$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^Q \in \mathbb{R}^{D\times D}$	$Q \in \mathbb{R}^{L\times D}$
$K = XW^K$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^K \in \mathbb{R}^{D\times D}$	$K \in \mathbb{R}^{L\times D}$
$V = XW^V$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^V \in \mathbb{R}^{D\times D}$	$V \in \mathbb{R}^{L\times D}$
$[Q^1, Q^2 \dots, Q^{n_H}] = \mathrm{Partion}_{-1}(Q)$	$Q \in \mathbb{R}^{L\times D}$	$Q^h \in \mathbb{R}^{L\times H}, h = 1, \dots n_H$
$[K^1, K^2 \dots, K^{n_H}] = \mathrm{Partion}_{-1}(K)$	$K \in \mathbb{R}^{L\times D}$	$K^h \in \mathbb{R}^{L\times H}, h = 1, \dots n_H$
$[V^1, V^2 \dots, V^{n_H}] = \mathrm{Partion}_{-1}(V)$	$V \in \mathbb{R}^{L\times D}$	$V^h \in \mathbb{R}^{L\times H}, h = 1, \dots n_H$
$\mathrm{score}^{\mathrm{h}} = \mathrm{softmax}(\frac{Q^h {K^h}^T}{\sqrt{H}}), i = 1, \dots n_H$	$Q^h, K^h \in \mathbb{R}^{L\times H}$	$\mathrm{score}^{\mathrm{h}} \in \mathbb{R}^{L\times L}$
$Z^h = \mathrm{score}^{\mathrm{h}} V^h, h = 1, \dots n_H$	$\mathrm{score}^{\mathrm{h}} \in \mathbb{R}^{L\times L}, V^h \in \mathbb{R}^{L\times H}$	$Z^h \in \mathbb{R}^{L\times H}$
$Z = \mathrm{Merge}_{-1}\ ([Z^1, Z^2 \dots, Z^{n_H}])$	$Z^h \in \mathbb{R}^{L\times H}, h = 1, \dots n_H$	$Z \in \mathbb{R}^{L\times D}$
$\mathrm{Out} = ZW^O$	$Z \in \mathbb{R}^{L\times D}$, $\mathrm{W}^O \in \mathbb{R}^{D\times D}$	$\mathrm{Out} \in \mathbb{R}^{L\times D}$
$A = \mathrm{Out}\ W^1$	$\mathrm{Out} \in \mathbb{R}^{L\times D}$, $\mathrm{W}^1 \in \mathbb{R}^{D\times 4D}$	$A \in \mathbb{R}^{L\times 4D}$
$A' = \mathrm{relu}(A)$	$A \in \mathbb{R}^{L\times 4D}$	$A' \in \mathbb{R}^{L\times 4D}$
$X' = A'W^2$	$A' \in \mathbb{R}^{L\times 4D}$, $\mathrm{W}^2 \in \mathbb{R}^{4D\times D}$	$X' \in \mathbb{R}^{L\times D}$

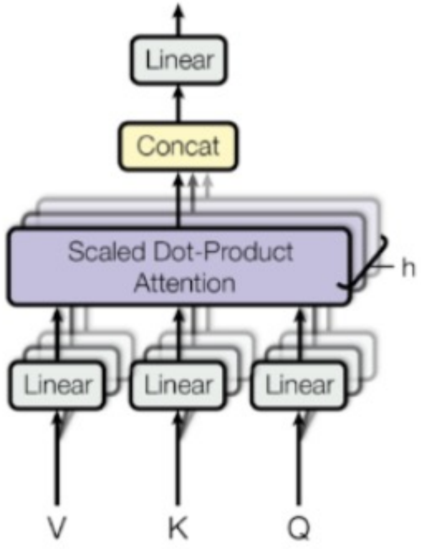
Other Components

- Residual connections:
  - Instead of simply return $\text{TransformerBlock}(X_{1:L})$
  - Return: $X_{1:L} + \text{TransformerBlock}(X_{1:L})$
- Layer normalization:
  - $\text{LayerNorm}(X_{1:L}) = \alpha \frac{X_{1:L} - \mu}{\sigma} + \beta$
  - μ is the mean; σ is the standard deviation.
  - α and β are learnable parameters.
- Positional embeddings:
  - So far, the embedding of a token doesn't depend on where it occurs in the sequence, which is not sensible. ($\text{PosEmb} \in \mathbb{R}^{L\times D}$)
  - $\begin{cases} \text{PosEmb}(i, 2j) = \sin(\frac{\text{i}}{10000^{2j/D}}) \\ \text{PosEmb}(i, 2j+1) = \cos(\frac{\text{i}}{10000^{2j/D}}) \end{cases}$
  - Where $i = 1, \ldots, L, j = 1, \ldots, \frac{D}{2}$
  - $X_{1:L} = X_{1:L} + \text{PosEmb}$ before computing $Q_{1:L}, K_{1:L}, V_{1:L}$.

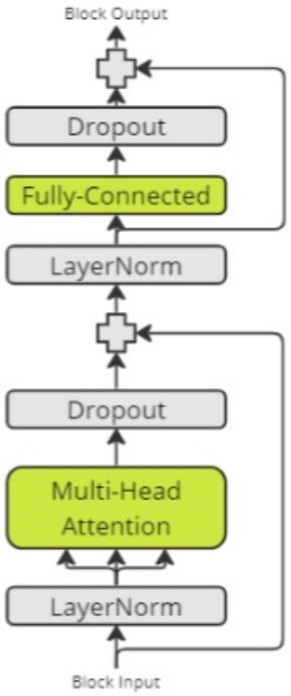
Put Them Together



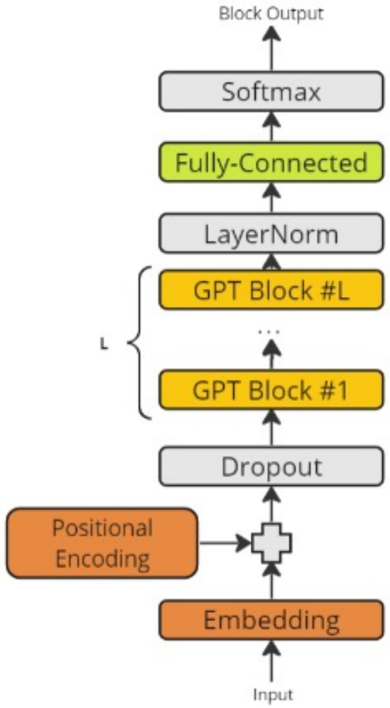
Scale Causal Attention



Multi-Head Attention



Transformer Block



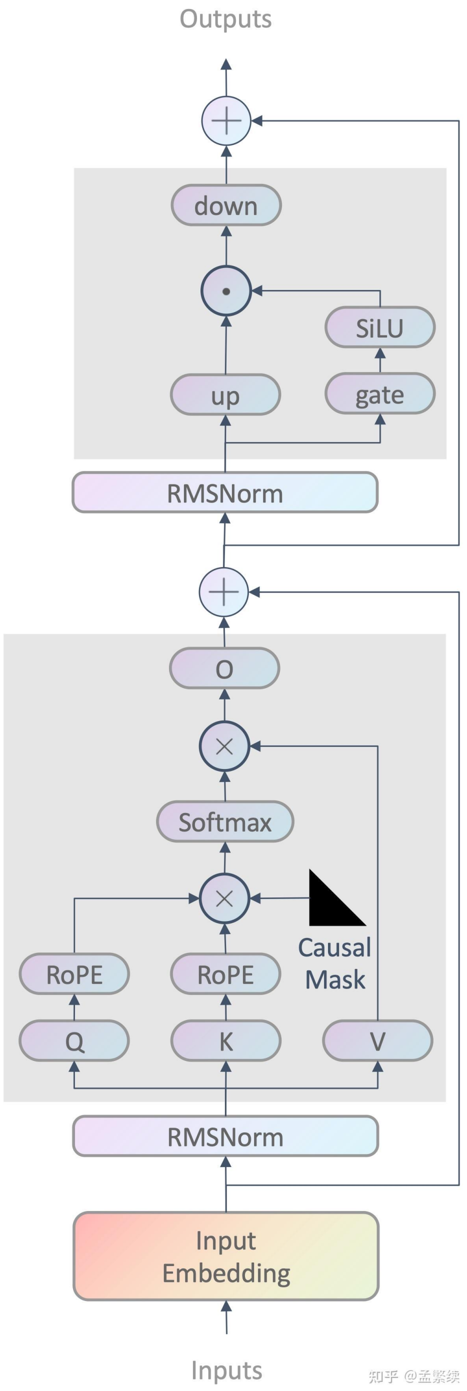
GPT Model

LLM Architecture
Case Study

LLaMa 3

Llama-3 Block Overview

- RMSNorm (Root Mean Square Norm);
- RoPE (Rotary Position Embedding);
- GQA (Group Query Attention);
- SiLU activation function in MLP.

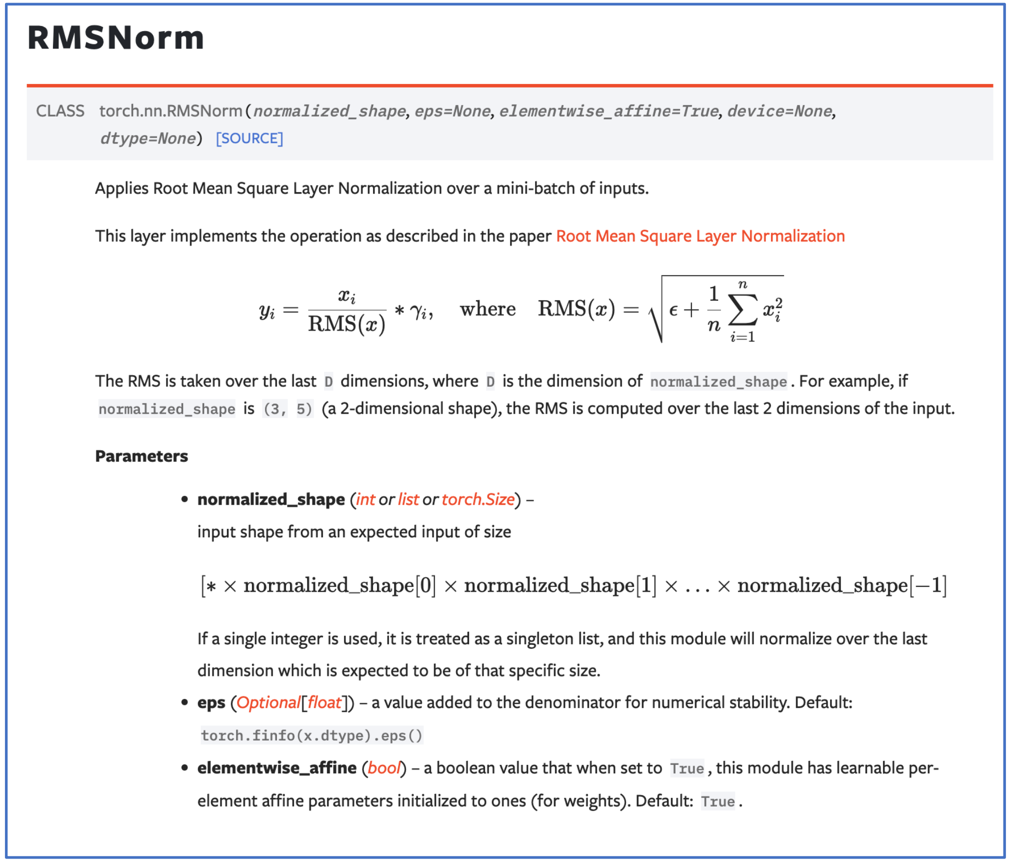


知乎 @孟繁续

RELAXED SYSTEM LAB

Llama-3 RMSNorm

- RMSNorm: $\mathrm{RMSNorm}(X_{1:L}) = \alpha \frac{X_{1:L}}{\sqrt{\mu^2+\epsilon}}$
  - μ is the mean.
  - 40% Speed-up compared with LayerNorm.



RMSNorm

CLASS torch.nn.RMSNorm(*normalized_shape*, *eps=None*, *elementwise_affine=True*, *device=None*, *dtype=None*) [SOURCE]

Applies Root Mean Square Layer Normalization over a mini-batch of inputs.

This layer implements the operation as described in the paper Root Mean Square Layer Normalization

$$y_i = \frac{x_i}{\mathrm{RMS}(x)} * \gamma_i, \quad \text{where} \quad \mathrm{RMS}(x) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

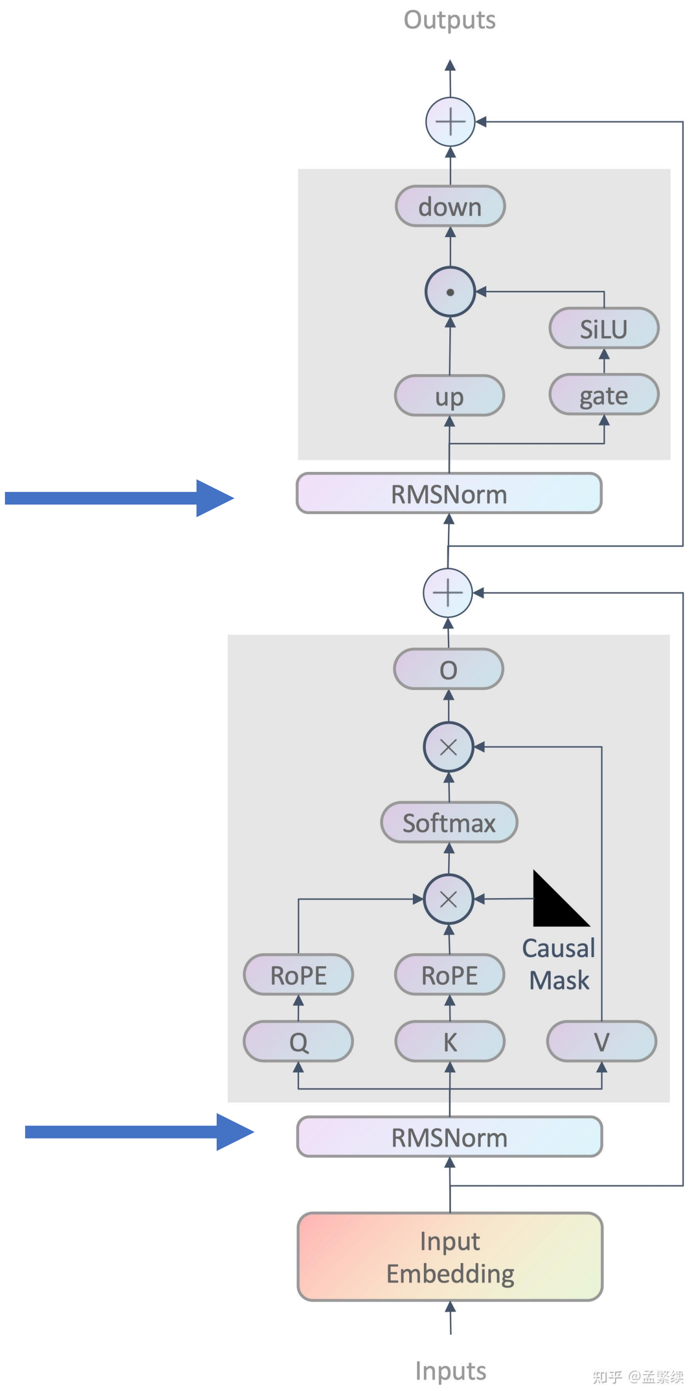
The RMS is taken over the last `D` dimensions, where `D` is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the RMS is computed over the last 2 dimensions of the input.

Parameters

- **normalized_shape** (*int* or *list* or *torch.Size*) – input shape from an expected input of size

  $$[* \times \text{normalized\_shape}[0] \times \text{normalized\_shape}[1] \times \ldots \times \text{normalized\_shape}[-1]]$$

  If a single integer is used, it is treated as a singleton list, and this module will normalize over the last dimension which is expected to be of that specific size.
- **eps** (*Optional*[*float*]) – a value added to the denominator for numerical stability. Default: `torch.finfo(x.dtype).eps()`
- **elementwise_affine** (*bool*) – a boolean value that when set to `True`, this module has learnable per-element affine parameters initialized to ones (for weights). Default: `True`.

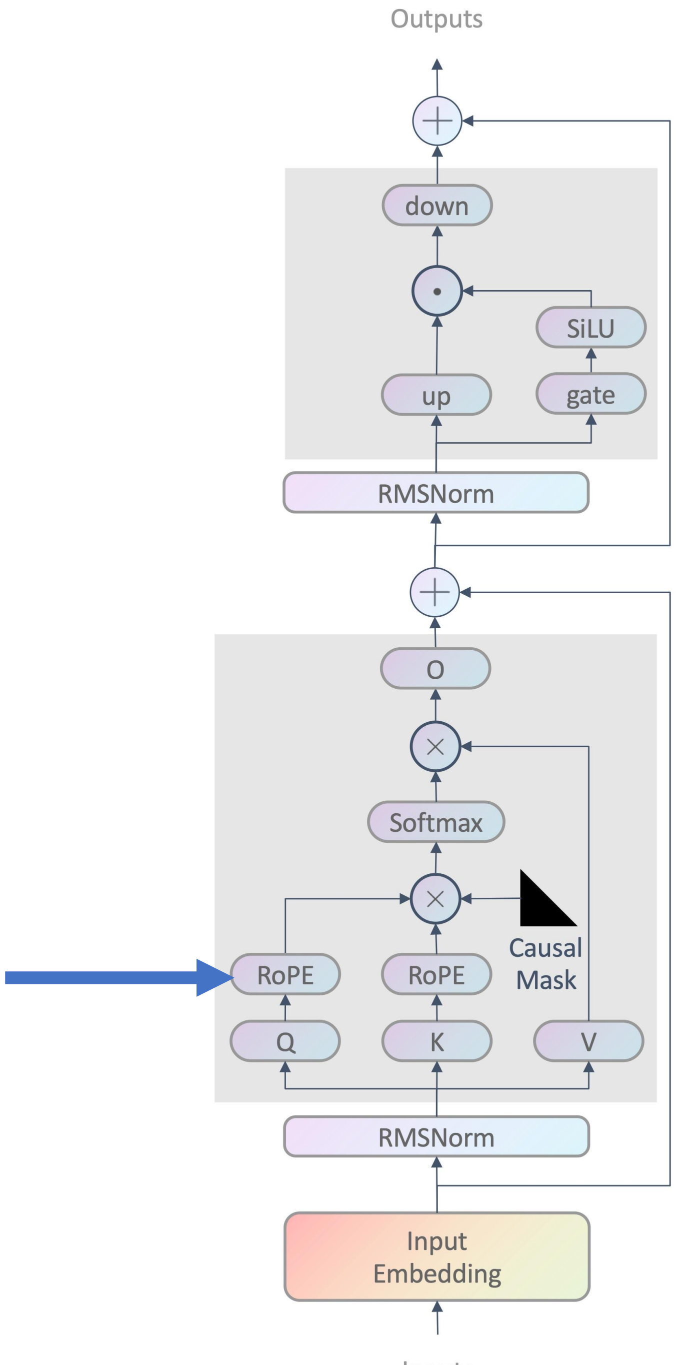


Llama-3 RoPE

- RoPE incorporates both absolute and relative positional information.
- Computation efficient implementation transform a position i:

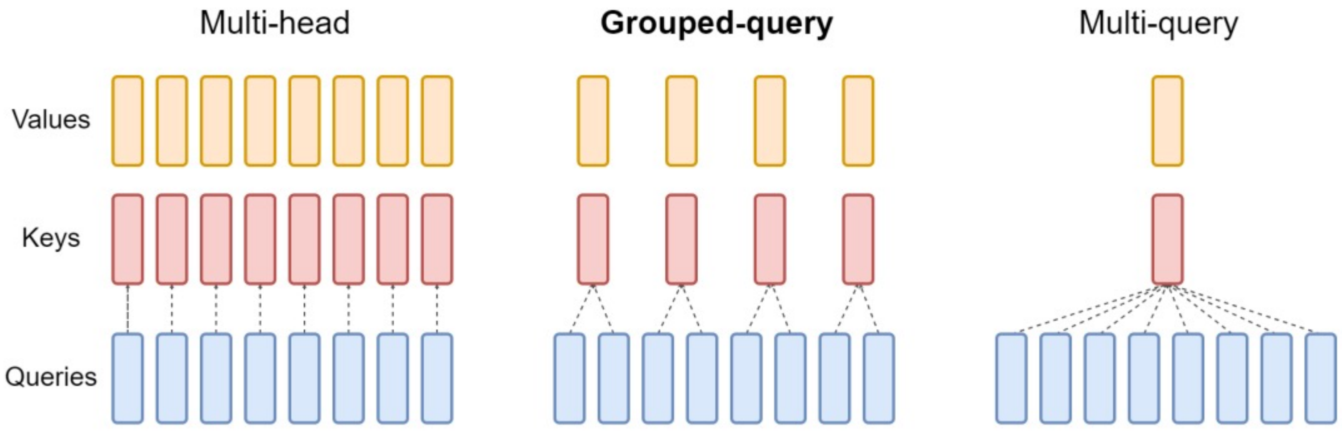
$$\begin{pmatrix} X_{i,1} \\ X_{i,2} \\ X_{i,1} \\ X_{i,1} \\ \vdots \\ X_{i,D-1} \\ X_{i,D} \end{pmatrix} \otimes \begin{pmatrix} \cos i\theta_1 \\ \cos i\theta_1 \\ \cos i\theta_2 \\ \cos i\theta_2 \\ \vdots \\ \cos i\theta_{D/2} \\ \cos i\theta_{D/2} \end{pmatrix} + \begin{pmatrix} -X_{i,2} \\ X_{i,1} \\ -X_{i,4} \\ X_{i,3} \\ \vdots \\ -X_{i,D} \\ X_{i,D-1} \end{pmatrix} \otimes \begin{pmatrix} \sin i\theta_1 \\ \sin i\theta_1 \\ \sin i\theta_2 \\ \sin i\theta_2 \\ \vdots \\ \sin i\theta_{D/2} \\ \sin i\theta_{D/2} \end{pmatrix}$$

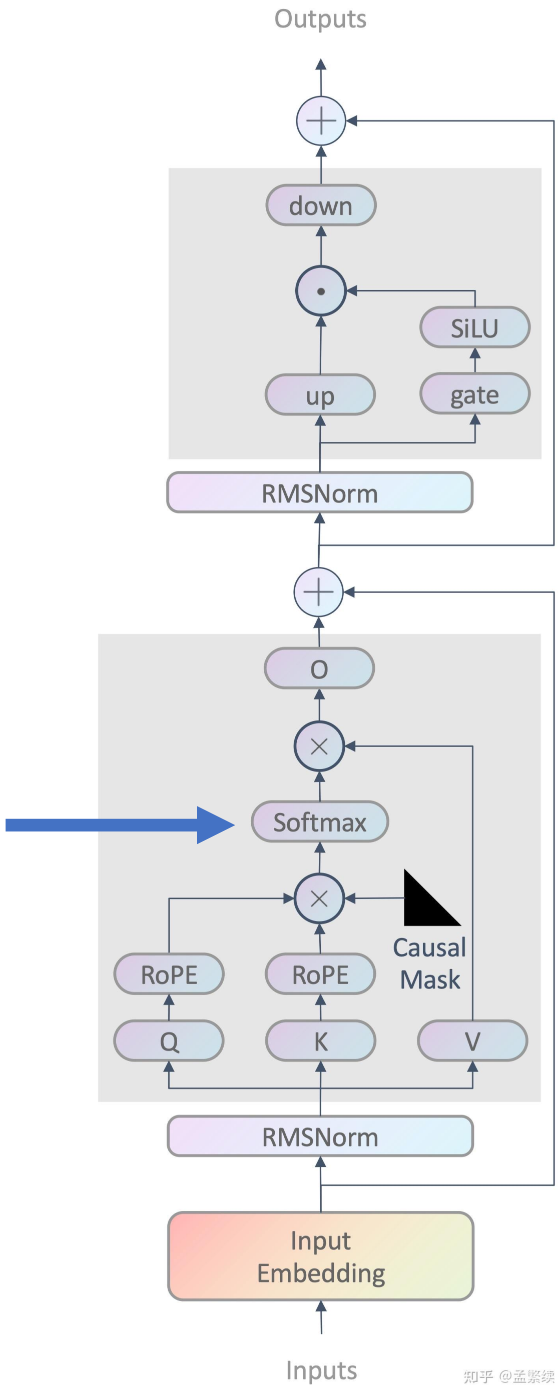
  - $\otimes$ indicates element-wise multiplication;
  - $\theta_j = 10000^{-\frac{2j}{D}}$
- https://arxiv.org/pdf/2104.09864



Llama-3 GQA

- Replace multi-head attention with grouped-query attention.

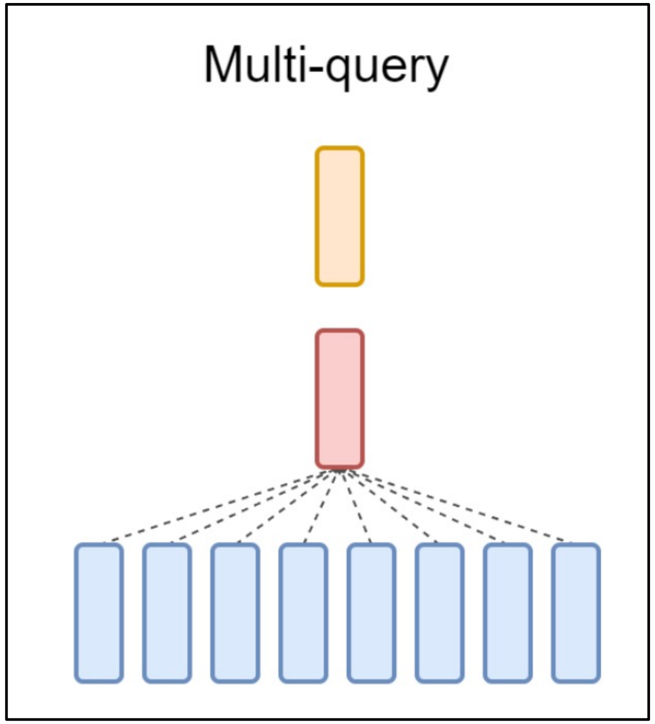




Multi-Query Attention (MQA)

- The idea is simple yet effective:
  - Use multiple query heads but only a single key and value head.

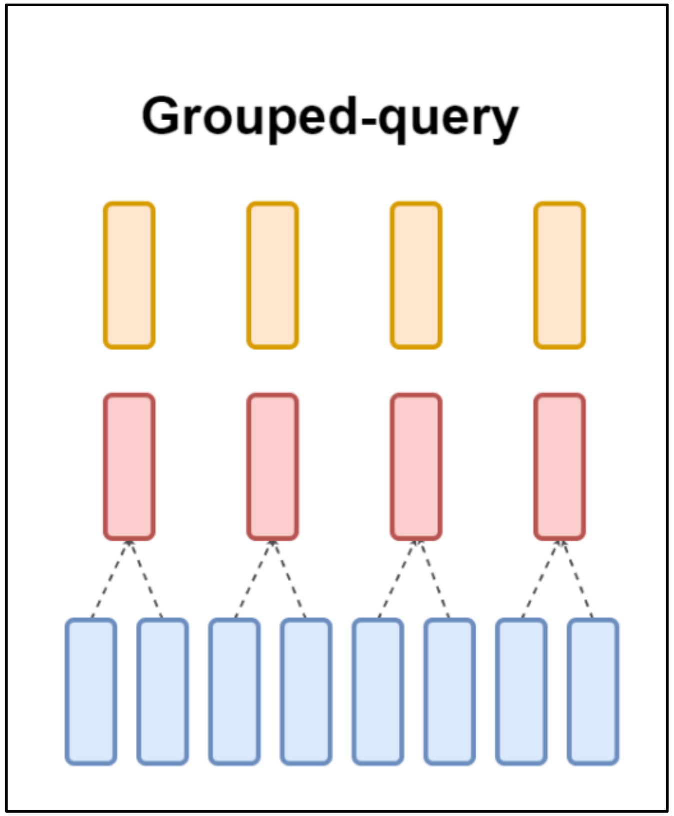
Computation	Input	Output
$Q = XW^Q$	$X \in \mathbb{R}^{L\times D}$, $W^Q \in \mathbb{R}^{D\times D}$	$Q \in \mathbb{R}^{L\times D}$
$K = XW^K$	$X \in \mathbb{R}^{L\times D}$, $W^K \in \mathbb{R}^{D\times H}$	$K \in \mathbb{R}^{L\times H}$
$V = XW^V$	$X \in \mathbb{R}^{L\times D}$, $W^V \in \mathbb{R}^{D\times H}$	$V \in \mathbb{R}^{L\times H}$
$[Q^1, Q^2 \dots, Q^{n_H}] = \text{Partition}_{-1}(Q)$	$Q \in \mathbb{R}^{L\times D}$	$Q^h \in \mathbb{R}^{L\times H}, h = 1, \dots n_H$
$\text{score}^h = \text{softmax}(\frac{Q^h K^T}{\sqrt{D}}), h = 1, \dots n_H$	$Q^h, K \in \mathbb{R}^{L\times H}$	$\text{score}^h \in \mathbb{R}^{L\times L}$
$Z^h = \text{score}^h V, h = 1, \dots n_H$	$\text{score}^h \in \mathbb{R}^{L\times L}, V \in \mathbb{R}^{L\times H}$	$Z^h \in \mathbb{R}^{L\times H}$



Group-Query Attention (GQA)

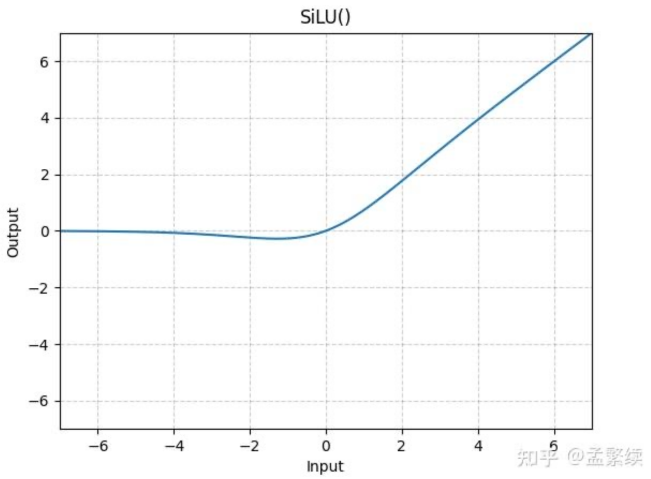
- The trade-off between MHA and MQA:
  - Divide query heads into g groups, each sharing a single key head and value head;
  - MHA: $g = n_H$; MQG: $g = 1$.

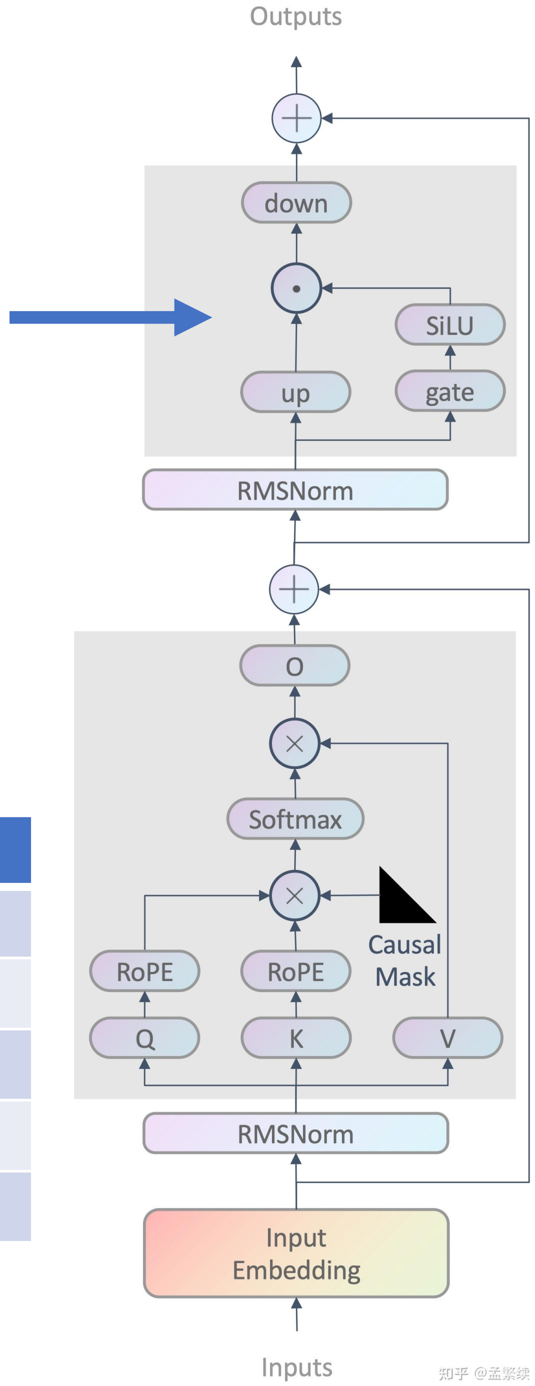
Computation	Input	Output
$Q = XW^Q$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^Q \in \mathbb{R}^{D\times D}$	$Q \in \mathbb{R}^{L\times D}$
$K = XW^K$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^K \in \mathbb{R}^{D\times gH}$	$K \in \mathbb{R}^{L\times gH}$
$V = XW^V$	$X \in \mathbb{R}^{L\times D}$, $\mathrm{W}^V \in \mathbb{R}^{D\times gH}$	$V \in \mathbb{R}^{L\times gH}$
$[Q^1, Q^2 \ldots, Q^{n_H}] = \text{Partition}_{-1}(Q)$	$Q \in \mathbb{R}^{L\times D}$	$Q^h \in \mathbb{R}^{L\times H}, h = 1, \ldots n_\mathrm{H}$
$[K^1, K^2 \ \ldots, K^g] = \text{Partition}_{-1}(K)$	$K \in \mathbb{R}^{L\times gH}$	$K^h \in \mathbb{R}^{L\times H}, h = 1, \ldots g$
$[V^1, V^2 \ \ldots, V^g] = \text{Partition}_{-1}(V)$	$V \in \mathbb{R}^{L\times gH}$	$V^h \in \mathbb{R}^{L\times H}, h = 1, \ldots g$
$\text{score}^{\text{h}} = \text{softmax}(\frac{Q^h K^{\lfloor \text{h}/g \rfloor^T}}{\sqrt{D}})$, $\text{h} = 1, \ldots n_H$	$Q^h, K^{\lfloor h/g \rfloor} \in \mathbb{R}^{L\times H}$	$\text{score}^{\text{h}} \in \mathbb{R}^{L\times L}$
$Z^h = \text{score}^{\text{h}} V^{\lfloor \text{h}/g \rfloor}$, $h = 1, \ldots n_H$	$\text{score}^{\text{h}} \in \mathbb{R}^{L\times L}$, $V^{\lfloor \text{h}/g \rfloor} \in \mathbb{R}^{L\times H}$	$Z^h \in \mathbb{R}^{L\times H}$



Llama-3 SiLU MLP

- Empirically shown to enhance model quality in various tasks.





Computation	Input	Output
$A = \mathrm{Out}\, W^1$	$\mathrm{Out} \in \mathbb{R}^{L\times D}$, $\mathrm{W}^1 \in \mathbb{R}^{D\times 4D}$	$A \in \mathbb{R}^{L\times 4D}$
$B = \mathrm{Out} W^2$	$\mathrm{Out} \in \mathbb{R}^{L\times D}$, $\mathrm{W}^2 \in \mathbb{R}^{D\times 4D}$	$B \in \mathbb{R}^{L\times 4D}$
$B' = SiLU(B)$	$B \in \mathbb{R}^{L\times 4D}$	$B' \in \mathbb{R}^{L\times 4D}$
$B'' = A \otimes B'$	$A \in \mathbb{R}^{L\times 4D}$, $B' \in \mathbb{R}^{L\times 4D}$	$B'' \in \mathbb{R}^{L\times 4D}$
$X' = B'' W^2$	$B'' \in \mathbb{R}^{L\times 4D}$, $\mathrm{W}^3 \in \mathbb{R}^{4D\times D}$	$X' \in \mathbb{R}^{L\times D}$

Scaling Law

Recall the Linear Layer Computation

- Forward computation of a linear layer: $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}$
  - Given input: $\boldsymbol{X} \in \mathbb{R}^{B \times D_1}$
  - Given weight matrix: $\boldsymbol{W} \in \mathbb{R}^{D_1 \times D_2}$
  - Compute output: $\boldsymbol{Y} \in \mathbb{R}^{B \times D_2}$
- Backward computation of a linear layer:
  - Given gradients w.r.t output: $\frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$
  - Compute gradients w.r.t weight matrix: $\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{X}^{\boldsymbol{T}} \frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$
  - Compute gradients w.r.t input: $\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{W}^{T} \in \mathbb{R}^{B \times H_2}$

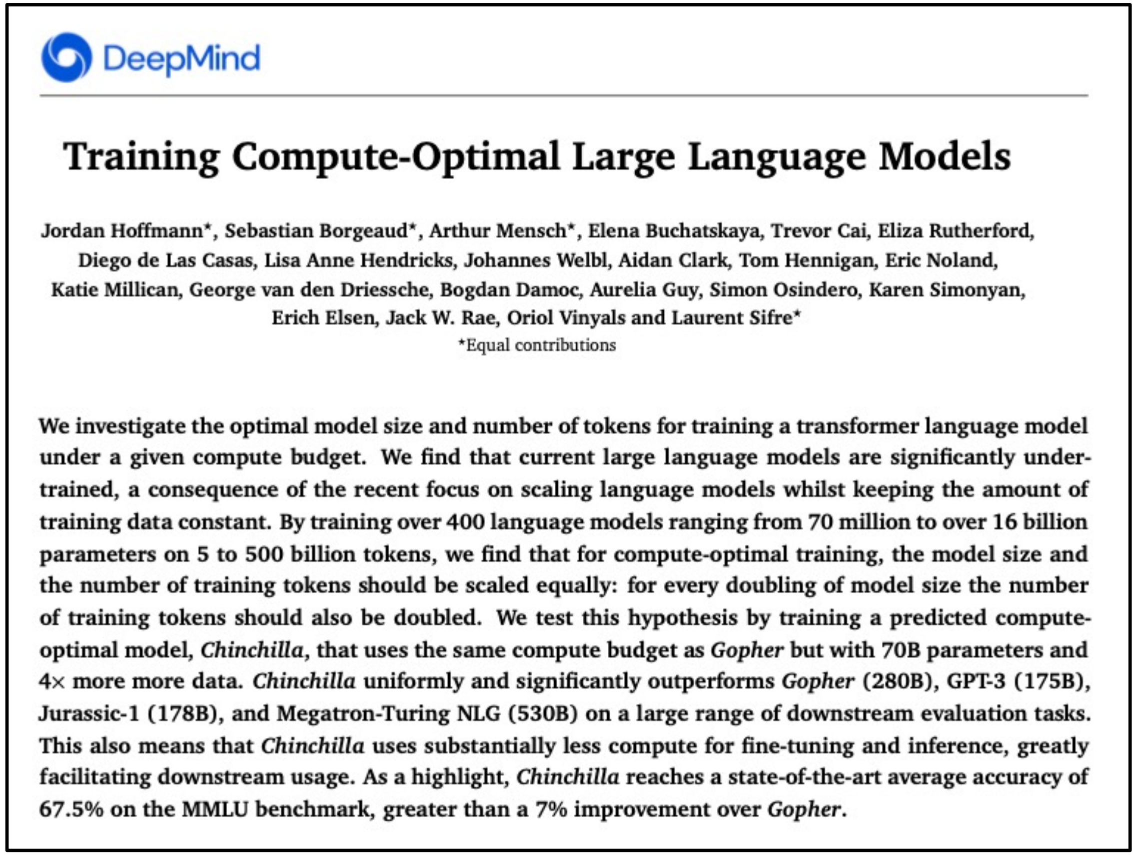
Estimate the Total Computation

- Suppose:
  - C is the total number of FLOPs, representing the computation load;
  - N is the number of model parameters;
  - D is the total amount of training data counted by tokens.
- The total computation:

$$C \approx 6ND$$

Key Question

- Intuitively:
  - Increase parameter # N → better performance
  - Increase dataset scale D → better performance
- But we have a fixed computational budget on $C \approx 6ND$
- ***<u>To maximize model performance, how should we allocate C to N and D?</u>***



DeepMind

Training Compute-Optimal Large Language Models

Jordan Hoffmann*, Sebastian Borgeaud*, Arthur Mensch*, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals and Laurent Sifre*

*Equal contributions

We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly undertrained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, *Chinchilla*, that uses the same compute budget as *Gopher* but with 70B parameters and 4× more more data. *Chinchilla* uniformly and significantly outperforms *Gopher* (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that *Chinchilla* uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, *Chinchilla* reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over *Gopher*.

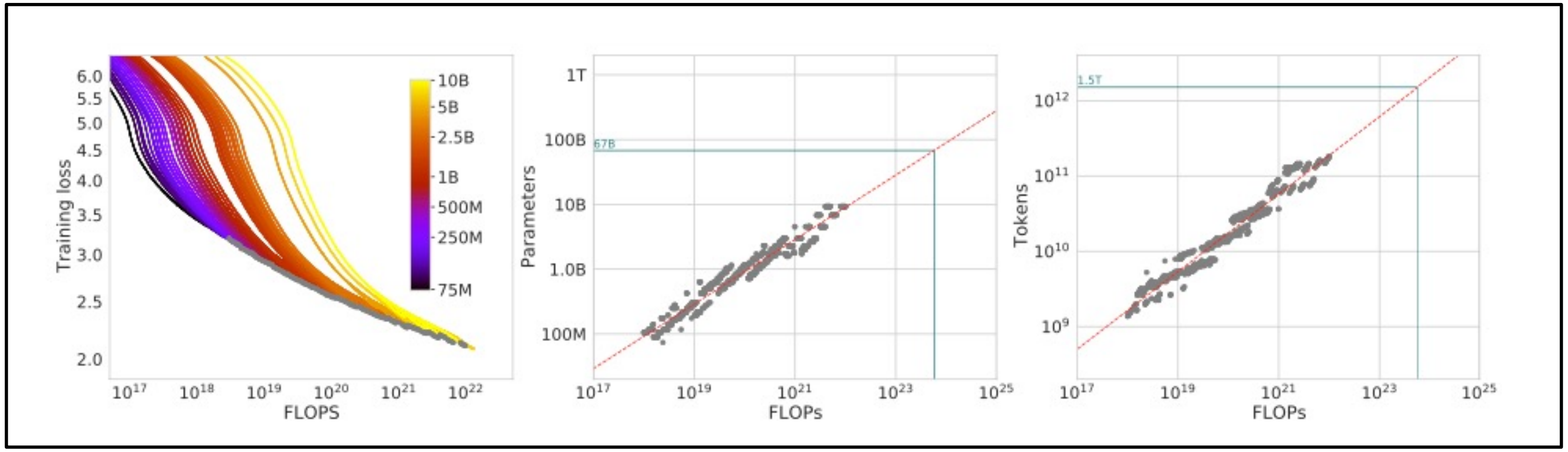
Chinchilla Scaling Law

- Given a fixed FLOPs budget, how should one trade off model size and the number of training tokens?
- For a large language model (LLM) autoregressively trained for one epoch, with a cosine learning rate schedule, we have:

$$\hat{L}(N,D) \triangleq E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}$$

  - $\hat{L}(N,D)$ is the average negative log-likelihood loss per token achieved by the trained LLM on the test dataset;
  - E represents the loss of an ideal generative process on the test data;
  - $\frac{A}{N^{\alpha}}$ captures the fact that a Transformer language model with N parameters underperforms the ideal generative process;
  - $\frac{B}{D^{\beta}}$ captures the fact that the model trained on D tokens underperforms the ideal generative process.

Chinchilla Scaling Law



- Conduct a series of benchmarks and optimizations to fit the function;
- Results:
  - $\alpha = 0.34,\ \beta = 0.28, A = 406.4, B = 410.7, E = 1.69$
- Conclusion:
  - $\begin{cases} N_{opt}(C) = 0.1C^{0.5} \\ D_{opt}(C) = 1.7C^{0.5} \end{cases}$

Chinchilla Scaling Law

- According to ***Chinchilla scaling law***, to achieve compute-optimal, the number of model parameters (N) and the number of tokens for training the model (D) should scale in approximately equal proportions.

$\boldsymbol{C}$	$\boldsymbol{N_{opt}(C)}$	$\boldsymbol{D_{opt}(C)}$
1.92E+19	400 Million	8.0 Billion
1.21E+20	1 Billion	20.2 Billion
1.23E+22	10 Billion	205.1 Billion
5.76E+23	67 Billion	1.5 Trillion
3.85E+24	175 Billion	3.7 Trillion
9.90E+24	280 Billion	5.9 Trillion
3.43E+25	520 Billion	11.0 Trillion
1.27E+26	1 Trillion	21.2 Trillion
1.30E+28	10 Trillion	216.2 Trillion

Evaluating Distributed Training System

Evaluating Distributed Computation

- Scaling law tells us given a fixed computation budget, how should we decide the model scale and data corpus.
- The computation budget is formulated by the total FLOPs demanded during the computation.
- But the GPU cannot usually work at its peak FLOPs.
- *How can we evaluate the performance of a distributed training workflow?*
  - Training throughput (token per second);
  - Scalability;
  - Model FLOPs Utilization.

Training Throughput

- **Training throughput** is a simple measurement:
  - How many tokens can be processed in a time unit (e.g., one second) for the whole cluster?
  - Processed means to finish forward computation, backward computation, and SGD updates in the training iteration.
  - E.g., **given a cluster of 256 A100 GPUs to train a 7B model, conducting one SGD iteration with a batch size of 2048, each sample with a sequence length of 4096 takes 12.7 seconds, what is the training throughput?**
    - $\frac{2048 \times 4096}{12.7} \approx 0.66$ million tokens per second
- Some people also like to use the term of **training throughput per GPU**:
  - In the above example, it becomes:
    - $\frac{2048 \times 4096}{12.7 \times 256} \approx 2580$ tokens per second per GPU

Scalability

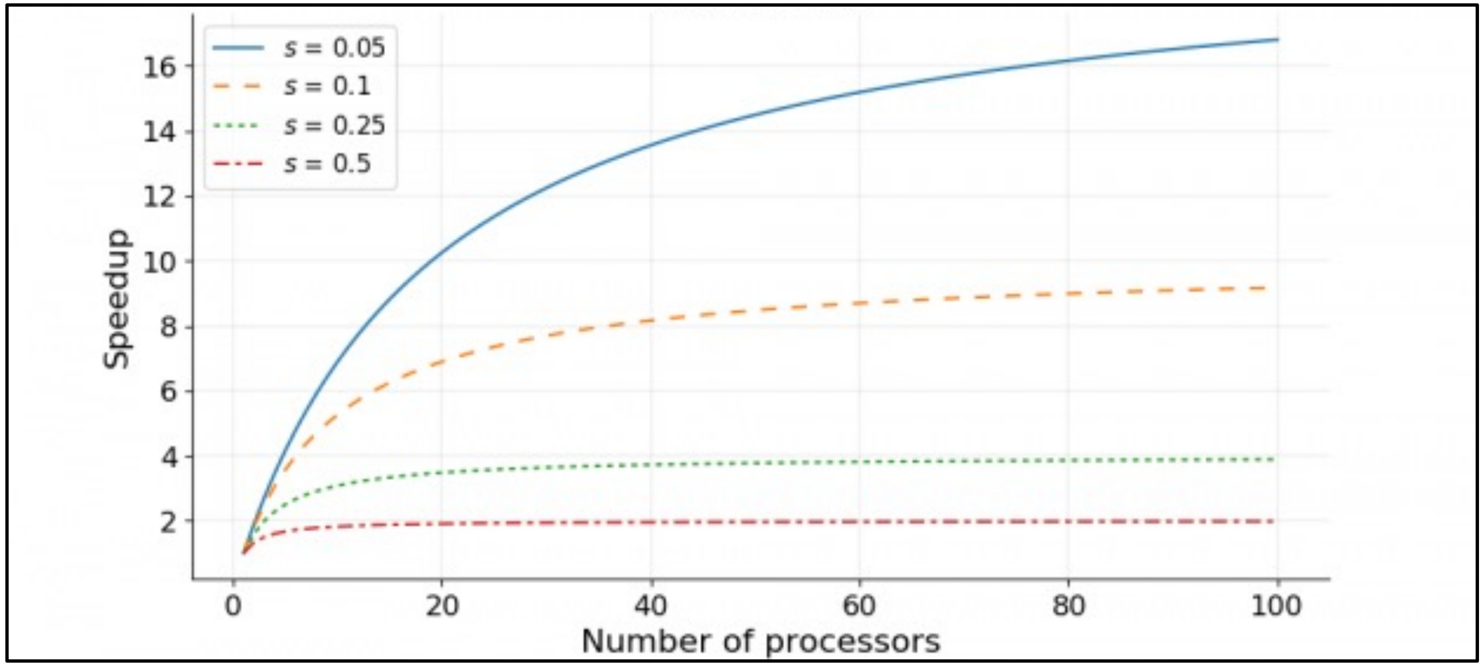
- Distributed systems can solve big problems (like our training computation) using a large number of processors.
- Scalability or scaling is widely used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased.
- The speedup in parallel computing can be straightforwardly defined as
  - $\text{speedup} = \frac{t_1}{t_N}$
  - t_1 is the computational time for running the software using one processor;
  - t_N is the computational time running the same software with N processors.
- Ideally, we want systems to have a linear speedup that is equal to the number of processors ($\text{speedup} = N$), which means that every processor would be contributing 100% of its computational power.
- Unfortunately, this is a very challenging goal for real (ML) applications to attain.

Strong Scalability

- ***Strong scalability*** suggests that the speedup is limited by the fraction of the serial part of the computation that is not amenable to parallelization:
  - $\text{speedup} = \frac{1}{s+\frac{p}{N}}$
  - S is the proportion of execution time spent on the serial part;
  - p is the proportion of execution time spent on the part that can be parallelized;
  - N is the number of processors.
- Strong scalability indicates that for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code.

Strong Scalability

- Strong scalability gives the upper limit of speedup for a problem of fixed size.
    - If one would like to gain a 500 times speedup on 1000 processors, strong scalability requires that the proportion of serial parts cannot exceed 0.1%.
- In practice, the sizes of problems scale with the amount of available resources.
    - We can use a larger batch size with more GPUs.



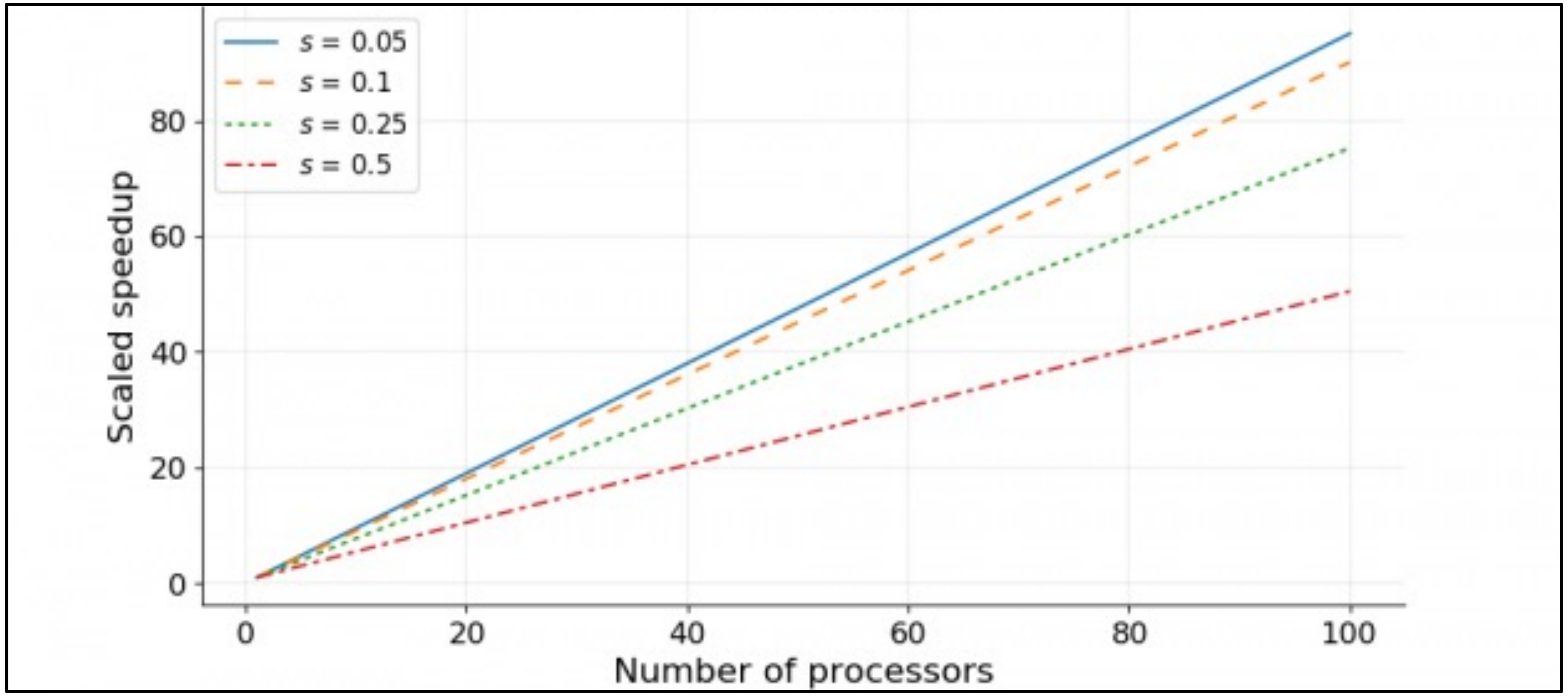
Weak Scalability

- ***Weak scalability*** is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem.
  - $\text{speedup} = s + p \times N$
  - S is the proportion of execution time spent on the serial part;
  - p is the proportion of execution time spent on the part that can be parallelized;
  - N is the number of processors.

Weak Scalability

RELAXED SYSTEM LAB

- Weak scalability indicates that speedup is calculated based on the amount of work done for a *scaled* problem size (in contrast to strong scalability that focuses on fixed problem size).



Scalability in Distributed ML Example

RELAXED SYSTEM LAB

Given a cluster of up to 256 A100 GPUs to train a 7B model, conducting one SGD iteration where each sample has a sequence length of 4096.

- Strong scalability:

# of GPU	1	4	16	64	256
Batch size on each GPU	2048	512	128	32	8
Global batch size	2048	2048	2048	2048	2048

- Weak scalability:

# of GPU	1	4	16	64	256
Batch size on each GPU	32	32	32	32	32
Global batch size	32	128	512	2048	8192

Model FLOPs Utilization

- Model FLOPs Utilization (MFU) measures how efficient/busy the hardware is during the execution of the training job:
  - $\epsilon = \frac{\text{Actual Compute FLOPS}}{\text{Cluster Peak FLOPS}} = \frac{6 \times N \times \frac{D}{t}}{F \times K} = \frac{6 \times N \times T}{F \times K}$
  - N the number of model parameters;
  - D the number of tokens in the training dataset;
  - t is the end-to-end training time of going through the whole training dataset;
  - T is the training throughput of the cluster, we assume: $T = \frac{D}{t}$ (i.e., no interruption or system failure).
  - F is the peak FLOPS of the GPU (e.g. $F_{A100} = 312$ TFLOPs)
  - K is the number of GPUs in this cluster.

Model FLOPs Utilization

- **Given a cluster of 256 A100 GPUs ($F = 312$ TFLOPs) to train a 7B model ($N = 7\times10^9$), conducting one SGD iteration with a batch size of 2048 (each sample with a sequence length of 4096) takes 12.7 seconds.**
  - What is the MFU for this cluster?
    - $\epsilon = \frac{6\times N\times T}{F\times K} = \frac{6\times7\times10^9\times\frac{2048\times4096}{12.7}}{312\times10^{12}\times256} = 35\%$
  - According to the Chinchilla Scaling Law, the desired training data should be around $D = 150$ Billion tokens, how long will the training finish?
    - $t = \frac{D}{T} = \frac{150\times10^9}{\frac{2048\times4096}{12.7}} \approx 63$ hours

References

- https://scholar.harvard.edu/sites/scholar.harvard.edu/files/binxuw/files/mlfs_tutorial_nlp_transformer_ssl_updated.pdf
- https://jalammar.github.io/illustrated-transformer/
- https://stanford-cs324.github.io/winter2022/lectures/introduction/
- https://stanford-cs324.github.io/winter2022/lectures/modeling/
- https://stanford-cs324.github.io/winter2022/lectures/training/
- https://zhuanlan.zhihu.com/p/636784644
- https://arxiv.org/pdf/2104.09864
- https://medium.com/@parulsharmmaa/understanding-rotary-positional-embedding-and-implementation-9f4ad8b03e32
- https://scholar.harvard.edu/sites/scholar.harvard.edu/files/binxuw/files/mlfs_tutorial_nlp_transformer_ssl_updated.pdf
- https://jalammar.github.io/illustrated-transformer/
- https://stanford-cs324.github.io/winter2022/lectures/introduction/
- https://stanford-cs324.github.io/winter2022/lectures/modeling/
- https://stanford-cs324.github.io/winter2022/lectures/training/
- https://en.wikipedia.org/wiki/Neural_scaling_law#cite_note-10
- https://stanford-cs324.github.io/winter2022/assets/pdfs/Scaling%20laws%20pdf.pdf
- https://www.cs.princeton.edu/courses/archive/fall22/cos597G/lectures/lec12.pdf
- https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/