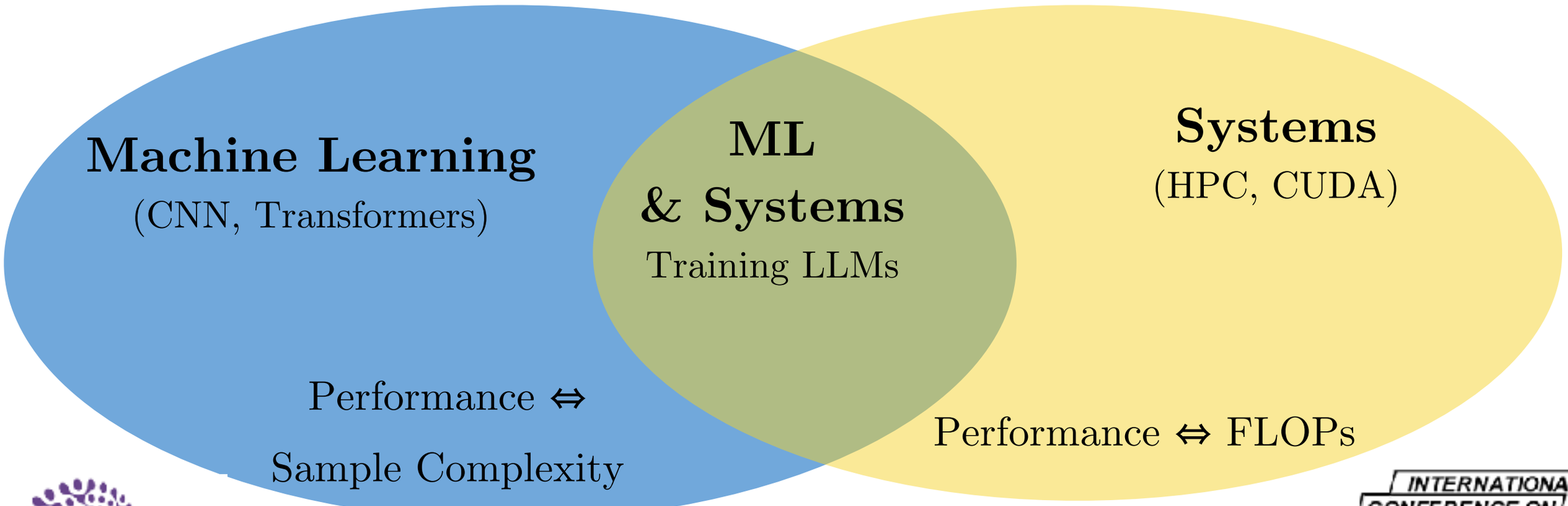
Lecture 1: Getting Started on Distributed LLM Training

Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

What this class is about and what it's not



NEURAL INFORMATION PROCESSING SYSTEMS

MLSys

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

Before we start...

- **Prerequisites:** I will assume you already know Python and PyTorch.

- If you already have solid experience with distributed training, you may find parts of the class too introductory.

- **Course goal:** Give you practical tools to understand training workflows, optimization, and the link between High-Performance Computing (HPC) and Machine Learning.

- What this class is not: Not a math-heavy class, and not a pure programming class.

Covered Topics

- **Lecture 1:** Transformers overview and the ML behind their training
- **Lecture 2:** Hardware fundamentals for training large models
- **Lecture 3:** Parallelization techniques for LLM training
- **Lecture 4:** Communication bottlenecks and Decentralized training
- **Lecture 5:** Post-training
- **Lecture 6:** Inference and serving
- **Lecture 7** (tentative): Agentic AI
- **Lecture 8**: Grading

Typical Lectures

- A typical 4h class will correspond typically to about 1h55 of lectures, followed by a 10 minute break and 1h55 of lab.

- Resources: Unfortunately, we won't have access to GPU resources for this class ...

- I assume everyone is a ChatGPT/LeChat/Claude/... user. **(so please don't hide yourselves or pretend you don't use it)**

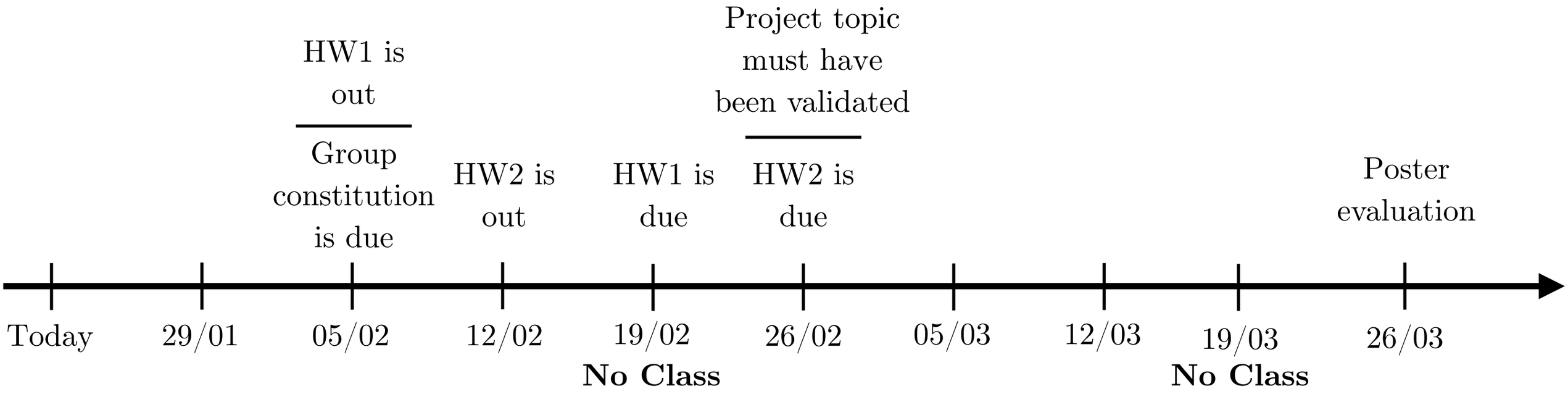
- On my side, I like honesty on this topic.

Grading

- You'll work in **pairs (groups of 2)** — please coordinate and communicate with your teammate.

- **Grading:** HW1 **25%**, HW2 **25%**, HW3/Project **50%**.

- **Project:** choose **one paper** (NeurIPS, MLSys, or similar) on those lectures topic. **Get my approval** that the topic is acceptable before you start.



What you will learn today

- Why analysing neural networks training is hard
- Data: what's used and why it matters
- Transformer architecture: the essentials
- An introduction to training recipes: how transformers are trained in practice

What is an LLM?

- An LLM (Large Language Model) is an **neural network trained on huge amounts of text** to *predict the next word* in a sentence and then specialised to a given task.

- LLMs rely on Transformers, and since 2017, their capacity and performance have kept increasing.

- They can generate, summarize, translate and answer questions in human-like language.

Why LLMs Are So Useful in ML?

- Part of the popularity of LLMs in ML is their ability to be used as
  - **Foundation Models:**
    - Trained once on massive, general-purpose data,
    - Then reused and adapted for many downstream tasks.
- **In context learner:**
  - No retraining: can work in **zero-shot** or **few-shot** mode.
  - No gradient updates, no optimizer – yet they behave as if they "learn" the new task from the prompt (i.e., the input).
  - LLMs act as universal task interfaces: we change the *prompt*, not the *model.*

The two phases of LLM training

- The process of training an LLM is particularly costly and can be split in two phases:

  - **Pre-training** (generic learning):

    - The model is trained once on a massive corpus of raw text

    - It aims to learn general language patterns, world knowledge, and basic reasoning skills.

  - **Post-training** (specialisation & alignment):

    - The pre-trained model is typically further trained on curated data and human feedback

    - It aims to specialise it for particular tasks (e.g. instruction following, coding, chat) and align it with safety & style constraints.

Distributed Optimization in a Nutshell

Setting

- Optimization aims at solving problems such that

$$\inf_{\theta \in \mathbb{R}^d} f(\theta)$$

where f has some regularity assumptions, such as L-smoothness:

$$\|\nabla f(\theta) - \nabla f(\theta')\| \leq L\|\theta - \theta'\|$$

- Typically f is an empirical risk, which has a finite sum structure with a regularizer:

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} F(\theta, \xi_i) + \frac{\lambda}{2}\|\theta\|^2$$

and the simplest is to perform a gradient descent leading to algorithms of the type, for $\eta > 0$:

$$\theta_{t+1} = \theta_t - \frac{\eta}{n}\sum_{i=1}^{n} \nabla F(\theta_t, \xi_i) \quad \text{until} \quad f(\theta_t) - f(\theta^*) \leq \epsilon$$

- In this case, two relevant quantities are :

$T(\epsilon)$: the number of steps to get to an ϵ-solution.

$G(\epsilon)$: the number of gradients oracle calls.

Here: $G(\epsilon) = nT(\epsilon)$

An embarrassingly parallel setting

- How can we exploit the structure of $\frac{1}{n}\sum_{i=1}^{n}\nabla F(\theta,\xi_i)$?
- Assume bounded variance, i.e. $\frac{1}{n}\sum_{i=1}^{n}\|\nabla F(\theta,\xi_i)-\nabla f(\theta)\|^2\leq\sigma^2$
- Let's introduce: $g_k(\theta)=\frac{1}{k}\sum_{j=1}^{k}\nabla F(\theta,\xi_{i_j})$ with $i_1,...,i_k\sim\text{Unif}\big(\{1,\dots,n\}\big)$
- In this case, we have $\mathbb{E}[g_k(\theta)]=\nabla f(\theta)$ and $\text{Var}(g_k(\theta))\leq\frac{\sigma^2}{k}$

$$\text{also:}\quad G(\epsilon,k)=kT(\epsilon,k)$$

- We are in an embarrassingly parallel regime: can we choose the batch size k to minimise the complexity $G(\epsilon,k)$ or the number of steps $T(\epsilon,k)$?
- Let's study briefly this simplified setting to get some intuitions.

Example: convex setting

- For instance, assuming that $F(.,\xi)$ is convex and L-smooth then, with the right step size using gradient descent:

$$\mathbb{E}[f(\theta_t) - f(\theta^\star)] \leq \frac{LR^2}{t} + \frac{\sigma R}{\sqrt{t}}, \quad R = \|\theta_0 - \theta^\star\|.$$

- This leads to: $T(\epsilon, k) = \Theta\left\{\frac{LR^2}{\varepsilon} + \frac{\sigma^2 R^2}{k\,\varepsilon^2}\right\}$ with $G(\epsilon, k) = kT(\epsilon, k)$

- Two cases can be distinguished:
  - Noise σ^2 dominates (right): then we can reduce the number of steps (time) while maintaining the total complexity, better pick $k = n$.
  - Optimization term dominates (left): increasing k only increases the complexity, better pick $k = 1$.

- Does it bring any insights on Deep Learning training and the required batch-size k?

Deep networks and theory: limitations

- For deep networks, many standard assumptions *fail*: regularity is unclear, and the gradient noise can be effectively unbounded for common nonlinearities.

- Even under very strong assumptions, we typically do not get usable guarantees for *optimal* hyperparameters (e.g. step size η, batch size) or the *optimal* value.

- Convexity allows sharp characterisations, but neural networks are highly non-convex: what works in convex settings need not work in non-convex ones, and vice versa.

- In practice, both batch size and learning rate *must* be tuned (e.g. by cross-validation)

- However, even when bounds are essentially vacuous, theory still plays a role as a **sanity check** and conceptual guide.

Proofs as a sanity check

- The typical assumption one will encounter in the literature are L smoothness (L-lipschitz gradients) and bounded variance.

- To prove convergence, one typically (implicitly or explicitly) relies on a Lyapunov function, $V : \mathbb{R}^d \to \mathbb{R}_+$ which has to satisfy for a gradient path $(\theta_t)_{t\geq 0}$:

$$d\frac{V(\theta_t)}{dt} = \langle \nabla V(\theta_t), \theta_t \rangle \leq 0$$

- Typically, one shows that additional conditions hold for those functions, e.g.:

$$\|\nabla V(\theta)\|^2 \geq \mu(V(\theta) - V^*) \qquad \mu\text{-PL conditions}$$

$$d\frac{V(\theta_t)}{dt} \leq -\alpha V(\theta_t) \qquad \text{strong convexity}$$

The difficulty is to find both the dynamic θ_t and Lyapunov function V.

The training loop

- The usual training loop of a Transformer is rather simple, assuming data are spliced across n workers via $\Xi = \cup_i \Xi_i$:

initialise layers

until completion, on worker i:

$$\text{sample } \xi_t^i \sim \boxed{\Xi^i} \quad \text{// sample from the chunk of the data}$$

$$\texttt{loss}_t^i = f(\theta_t, \xi_t^i) \quad \text{// forward pass}$$

$$\texttt{loss}_t = \frac{1}{n}\sum_{i=1}^{n} \texttt{loss}_t^i \quad \text{// average losses}$$

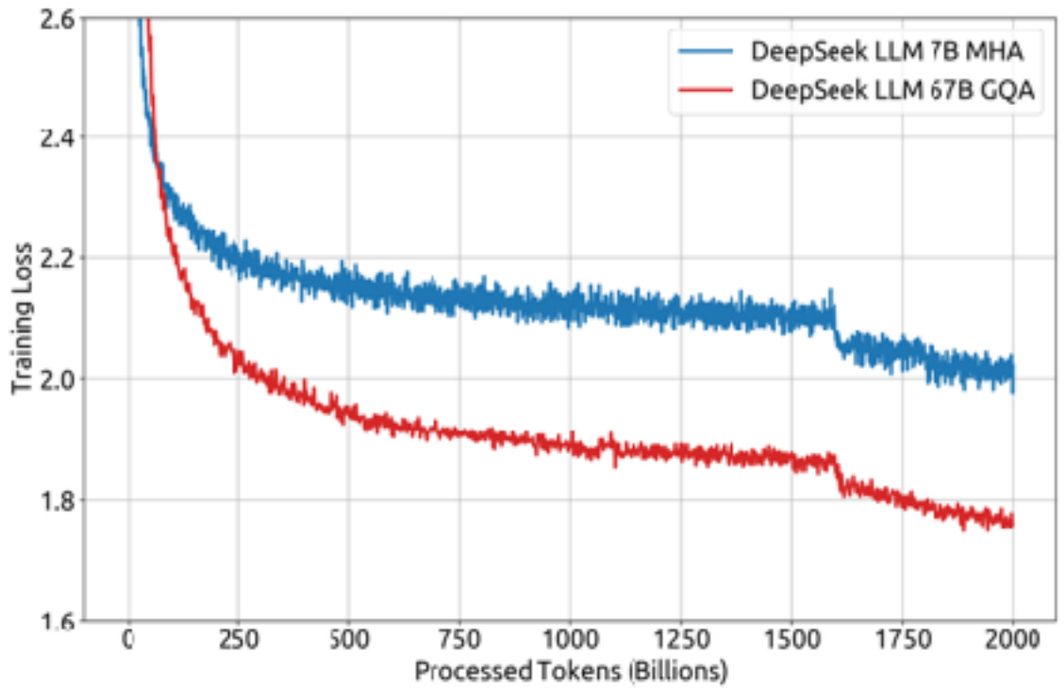
$$g_t^i = \nabla f(\theta_t, \xi_t^i) \quad \text{// backward pass}$$

$$g_t = \frac{1}{n}\sum_{i=1}^{n} g_t^i \quad \text{// average gradients}$$

$$\theta_{t+1}, \boxed{s_{t+1}} = \texttt{Optimizer}(\theta_t, g_t, \boxed{s_t}) \quad \text{// update parameters}$$

Example: Convergence of the previous training loop

In practice, we see such loss curves, with drops due to the learning-rate scheduler. Can we obtain basic guarantees in a simplified setting?



- Assume $\texttt{Optimizer} = \texttt{SGD}$
- Let $f^* = \inf_\theta f(\theta)$ and write $V(\theta) = f(\theta) - f^*$.
- Assume f is L-smooth and the gradient has variance σ^2, then for any learning rate η, we have:

$$\inf_{0 \le t < T} \mathbb{E}\|\nabla f(\theta_t)\|^2 \;\le\; \frac{2\big(f(\theta_0) - f^*\big)}{\eta T} \;+\; L\eta\sigma^2.$$

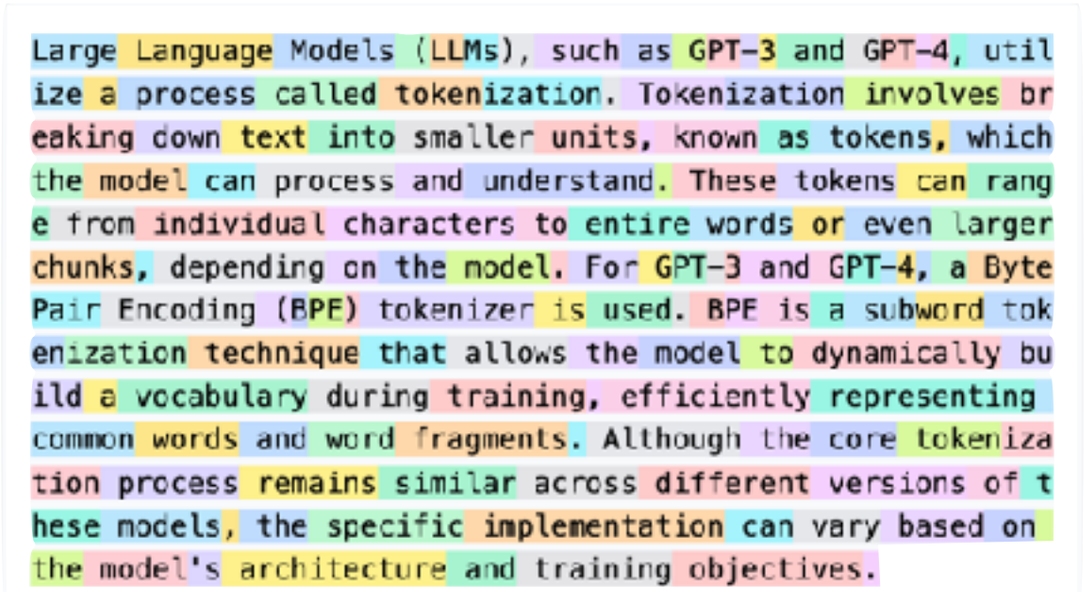
Thus, it converges to a stationary point, but the limit point and rate are unclear, and it fails to explain much of the observed training behavior.

Feeding Largue Language Models

Tokens: the fuel of LLMs

- LLMs consume sequences of **tokens**: discrete units (subwords, bytes, or image patches) produced by a tokenizer and fed directly to the model.

- Tokenization comes from NLP / text compression: it defines an (almost) invertible encoding of the raw signal (text, bytes, sometimes images).



"abc... " $\xrightarrow{\texttt{encode}}$ 10201... $\xrightarrow{\texttt{decode}}$ "abc... "

- Each pretrained model is tied to a **fixed tokenizer**; changing it later would require retraining (at least) the input/output embeddings.

What matters in Tokenisation for LLMs?

- Fidelity: does $\texttt{decode}(\texttt{encode}(s)) = s$? (e.g., handle white space, code syntax, emojis, URLs, . . .)

- Compression: how many tokens per byte (TPB) we get on typical data.
  Higher TPB = more tokens for the same text $\rightarrow$ *worse* compression.

- Predictability: a sequence is predictable if the model (human, LLM) assigns high probability to the correct next token, as reflected by low bits per token (bpt) from the cross-entropy loss.

- Efficiency is thus measured as bits per byte $\text{bpt} \times \text{TPB}$. There is a trade off, as
  - very fine tokenization: a character level tokenisation has a TPB high and btp is low (easy predictions)
  - very coarse tokenisation: TPB is low, but predicting each token is very hard

- Some tokens have special roles: end-of-sequence (EOS), beginning-of-sequence (BOS), padding, and task markers such as `<think>`, `<code>`, etc.

Generally, TPB (French) $>$ TPB (English)

PREDICTION AND ENTROPY OF PRINTED ENGLISH, Shanon proves this with N-grams instead of LLMs tokens dictionaries

From Tokens to Embeddings

- Tokens are mapped to **continuous vectors from** $\mathbb{R}^d$ via an embedding layer, e.g., a look-up table of size $\mathbb{R}^{d \times n_{\text{vocab}}}$.

- Tokenizer leads to Compute trade-offs:
  - Fewer tokens for the same bytes $\rightarrow$ **shorter sequences**, so more context fits in a fixed window.
  - But achieving this often requires a **larger vocabulary**, which increases embedding memory and latency.

- This raises the key question: How should we design our tokenizer to balance these costs?

- One of the solution is *Byte per Encoding*: simple, fast to train, with good compression and predictability.

Training BPE

- BPE is a simple greedy way to move toward a more efficient code (fewer bits per byte) by collapsing frequent patterns into single symbols.

- It greedily minimizes the number of tokens needed to encode a training corpus (under a fix budget). The algorithm is as follow:

    - Initialise with the characters of the alphabet:
      D {'a', 'b', 'c', ... }
        - **Count** all adjacent symbol pairs $w.w', (w, w') \in \mathcal{D}$
        - Find the most frequent $f_{\bar{w}}$ pair with $\bar{w} = w.w'$
        - Aggregate $\mathcal{D} \leftarrow \mathcal{D} \cup \{\bar{w}\}$

Remark 1: at every merge, the total number of tokens decrease by $f_{\bar{w}}$ (and it is the largest drop possible)

Remark 2: we can't really prove generalisation to unseen data, no optimality guarantee on the number of token. In practice: reasonable

Fast BPE inference

- After training, we have an **ordered list of merges** (a ranked dictionary of token pairs), namely `rank`.
- To tokenize new text:
  - First, **pre-tokenize** into base symbols (characters or bytes).
  - Then, **greedily apply merges in rank order**: for each merge rule, replace all matching pairs in the sequence. This is computationally extensive.
- More efficient variant:
  - Scan the sequence once, record all adjacent token pairs.
  - Maintain a priority queue (heap) keyed by merge rank.
  - Repeatedly take the best pair from the heap, merge it, and update only neighboring pairs.

$$\texttt{rank}[\text{"he"}] = 1$$
$$\texttt{rank}[\text{"lo"}] = 2$$
$$\texttt{rank}[\text{"llo"}] = 3$$

$$\text{h} \mid \text{e} \mid \text{l} \mid \text{l} \mid \text{o} \longrightarrow \text{he} \mid \text{l} \mid \text{lo} \longrightarrow \text{he} \mid \text{llo}$$

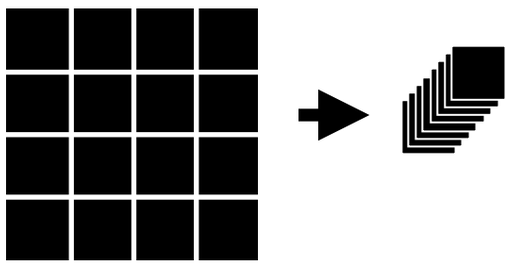
Remark: quasi-linear complexity: every successful merge replaces two tokens by one

/!\ this is not equivalent to scanning from left to right

Tokenisation in image (ViTs)

- For images, the simplest way in the spirit of a LLM is to directly embed images:

  - Split the image into a grid of fixed-size **patches** (e.g. 16×16 pixels).



  - Flatten each patch and apply a linear projection → **patch embedding** (a vector) so that they belong to $\mathbb{R}^d$.

  - Add **positional information** (which patch is where).

- From now, treat those as text-tokens.

Multi-Modal tokenisation

- A key property of transformers is that they can process **any sequence of tokens**. To handle **multimodal data**, we need *everything* (text, images, etc.) in token form.

- Example (vision–language prompt):

  "I find this image nice `<img1>` and this one too `<img2>`. What's in them?"
  Here, `<img1>` and `<img2>` are special tokens that stand for an image representation.

- Pipeline idea:
  Text: tokenized into text tokens.
  Images: encoded into **image tokens** (patch embeddings or discrete codebook IDs).

- The transformer then processes **one joint sequence**:

  [text tokens ... <img1 tokens> ... <img2 tokens> ...].

Pretraining data in practice

- Modern LMs are trained on huge, mixed corpora rather than a single dataset.

- Many open datasets are built on top of Common Crawl or similar sources, which you can think of as an *"arXiv of the web"* that we then filter, clean, and remix.

- Scale matters (we often train on trillions of tokens), but quality matters more: deduplication, filtering, and smart mixtures usually beat "just add more noisy web".

Dataset	**Size**	**# Tokens**	**Type of data**
FineWeb	44 TB	15 T	Web (English, filtered Common Crawl)
RefinedWeb	2.8 TB	0.6 T	Web (mostly English, filtered Common Crawl)
Dolma	9.6 TB	3.0 T	Mixed (web, academic, code, books, social)
SlimPajama	0.90 TB	630 B	Mixed (filtered from Common Crawl)
C4	0.81 TB	160 B	Web (English, filtered Common Crawl)
The Pile	0.83 TB	380 B	Mixed (22 English sources)
ROOTS	1.6 TB	340 B	Multilingual text (59 languages)
The Stack v2	68 TB	900 B	Source code

Largue Language Models Architectures

From RNNs to Transformers

- **Pre-2017:** NLP baselines were mostly RNN/LSTM/GRU encoder–decoders (seq2seq), i.e. recurrent architectures:

$$x_{n+1} = \phi(x_n, ..., x_{n-L})$$

- **RNN bottlenecks:** inherently sequential (poor GPU/TPU utilization), vanishing/exploding gradients, and limited long-range context even with attention on top.

- Transformer (2017): replaces recurrence with self-attention, by processing all tokens in parallel.

- What this enables:

  - **Parallelism (beats recurrence):** full-sequence parallel compute leads to much faster training.

  - **Long-range dependencies:** attention connects any token pair directly.

  - **Simple, scalable block:** repeating generic **transformers blocks** scales cleanly to billions of params

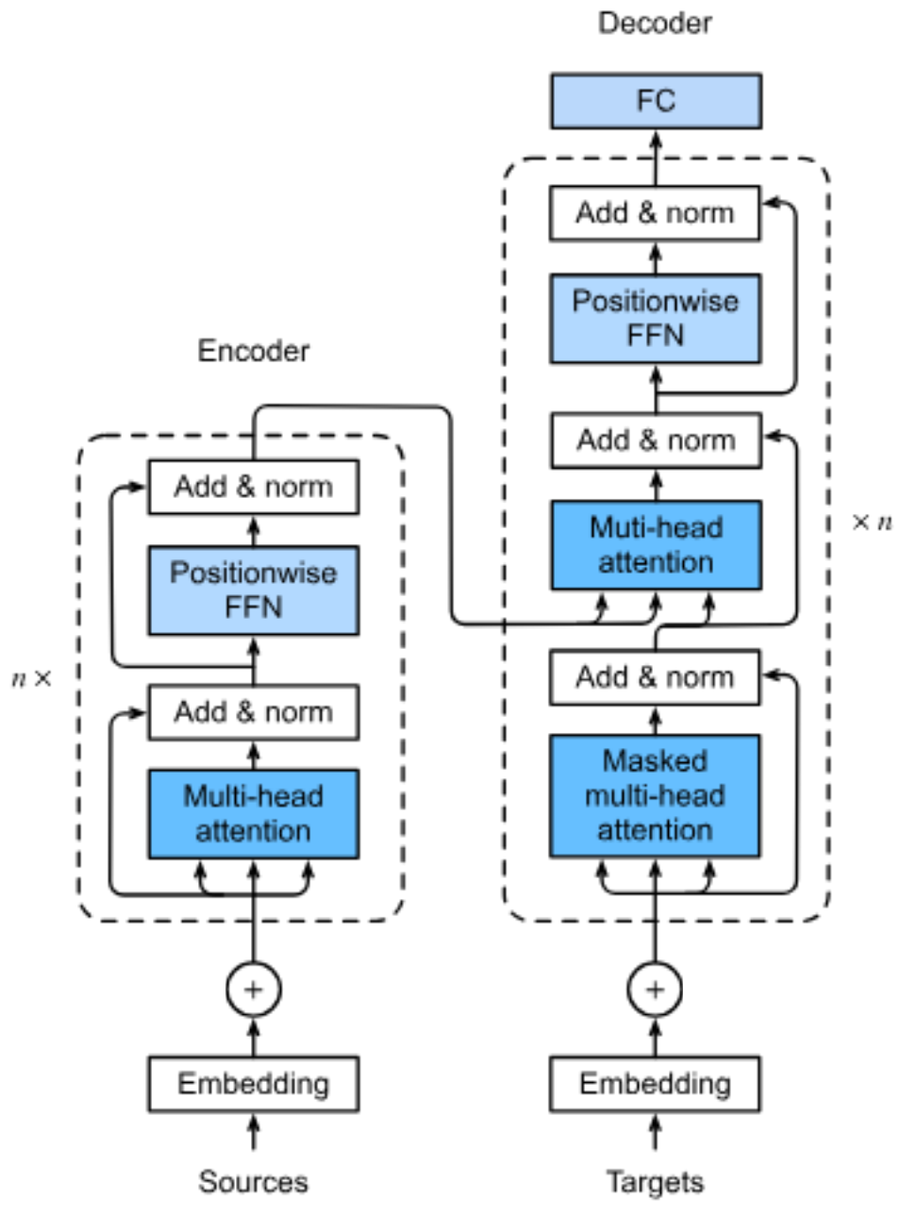
Transformers!

Main Transformer Families

- **Encoder-only:**
  Bidirectional attention over *all* tokens
  Great for understanding tasks (classification, retrieval, etc.)
  Not natively generative (usually trained with masked LM)
  Examples: BERT, RoBERTa

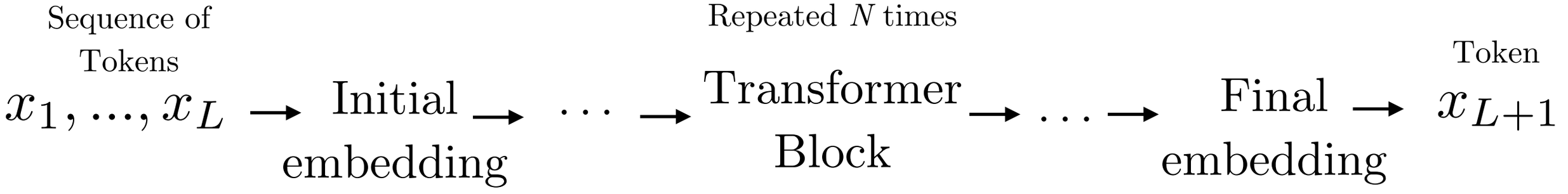
- **Decoder-only:**
  Causal (left-to-right) attention with a mask
  Naturally generative: next-token prediction
  Can still do comprehension via prompting
  Examples: GPT, LLaMA, Mistral

- **Encoder–decoder (seq2seq):**
  Encoder reads full input; decoder generates output with:
  - self-attention (causal)
  - cross-attention to encoder outputs at every layer
  Strong for tasks needing full context *before* generating, e.g. translation, summarisation, instruction tuning
  Examples: T5, BART



Decoder-only Transformers

- A decoder-only Transformer is a stack of identical blocks, each combining **self-attention** and a **two-layer feed-forward network** (with residuals layer norm).



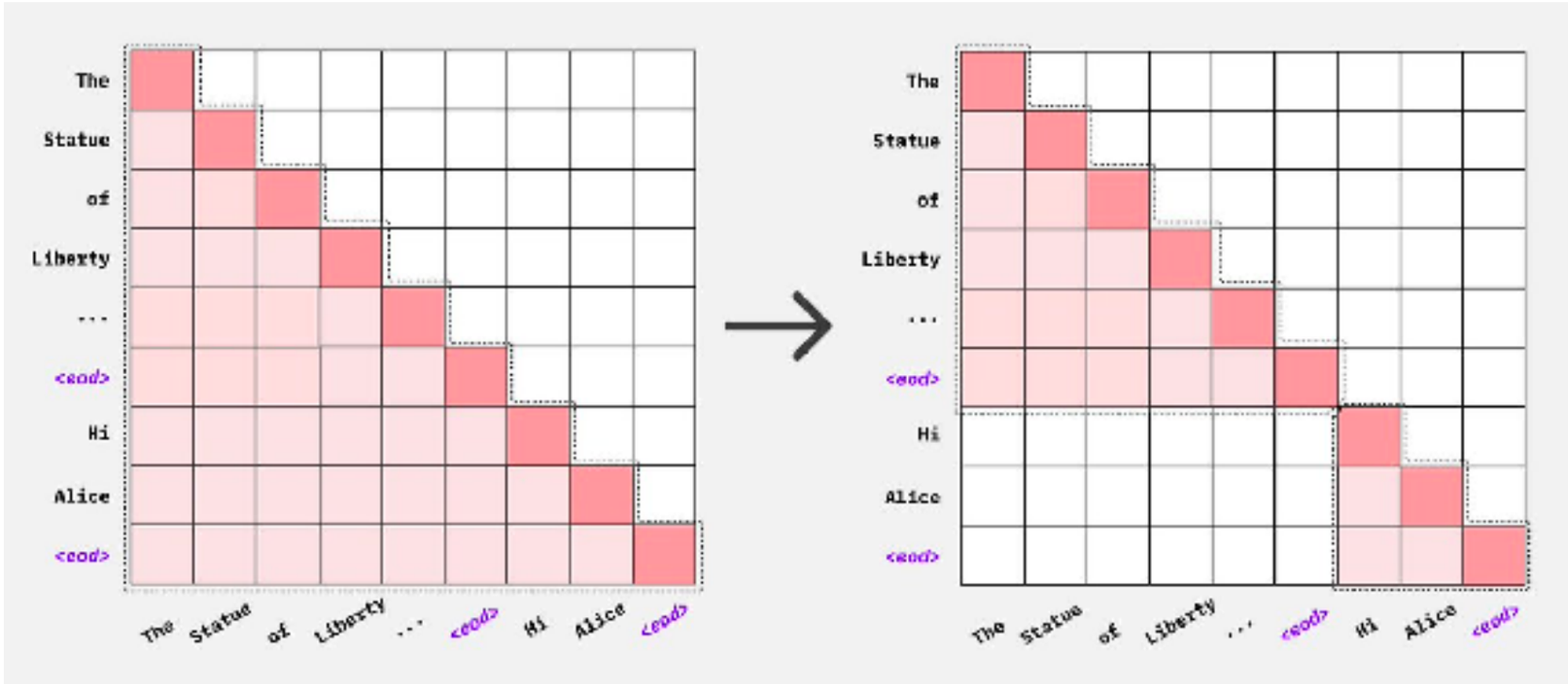
- **Transformer Blocks:**
  - **Attention layer**: performs dot-product between every tokens. Memory consuming but fast to compute.
  - **MLP layer**: performs at least 2 matrix multiplications, computationally intensive

Remark 1: a transformer layer preserves dimension across depth.

In the following, I'll focus on Llama-like models

Pre-training objective and packing

- Inputs are document packed, i.e., documents are concatenated and form a giant sequence (masking or sequence packing are possible)



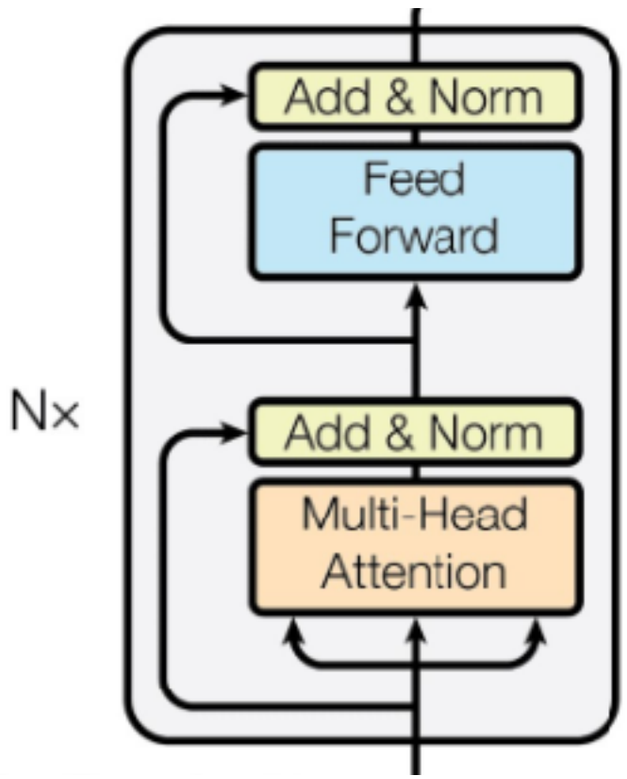
- Two possible ways to perform a prediction:
  - Next-token prediction:
    It's a causal LM objective: predict token x_t from $x_{<t}$.
  - Fill In the Middle (FIM): remove a fraction 10% of the token to predict

    `[prefix] <FIM_MIDDLE> [suffix] <FIM_END> [middle]`

Inside the Transformer

Description of a Transformer block

- **Goal:** a simple block we can stack hundreds of times that
  - uses only matrix multiplies (GPU-friendly)
  - can model long-range interactions
  - stays trainable at scale



- 3 components are composed:

  - Multi-head Attention:
    - *What:* each token attends to other tokens while keeping position (with a causal mask in LMs).
    - *Why:* lets the model route information flexibly and capture long-range dependencies in one step.

  - MLP (feed-forward network)
    - *What:* 2-layer MLP (often with expansion + non-linearity like SwiGLU).
    - *Why:* gives non-linear feature transforms and per-token capacity

  - RMSNorm + residual connections
    - *What:* normalize activations and add skip-connections around each sub-layer.
    - *Why:* keep activations **well-scaled**, stabilize gradients, and make it possible to train **very deep stacks** of identical blocks.

FLOPs count

- A FLoating Point Operation (FLOP) is one of the elementary operations performed by a computation unit. We use it to measure training/inference compute and to size jobs.

- FLOPs guide LLM architecture design; the goal is high *effective* FLOPS to reduce training time and avoid wasting GPU hours

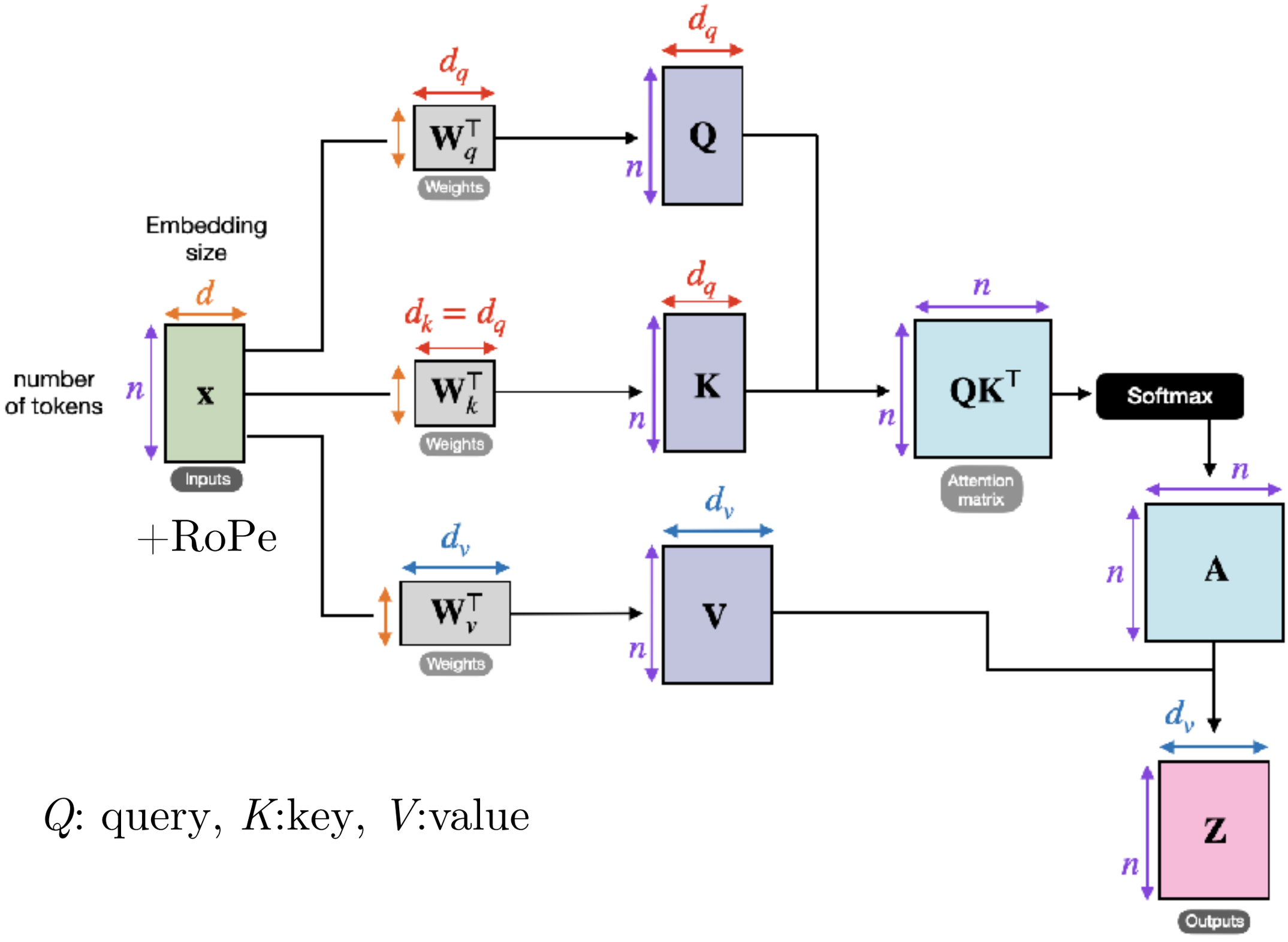
- We'll see in the next lecture that wall-clock time depends also on other considerations (e.g., IO access)

$A \in \mathbb{R}^{m \times n}, x \in \mathbb{R}^n$ what are the FLOPS to obtain $Ax \in \mathbb{R}^m$?

$$A_{11}x_1 + A_{12}x_2 + ... + A_{1n}x_n \longrightarrow m(2n-1) \text{ FLOPS}$$

Remark: "**FLOP vs FLOPs vs FLOPS:** FLOP=one operation; FLOPs=operation **count**; FLOPS=operations **per second**.

The attention layer



Q: query, K:key, V:value

RoPe (Rotary Positional Embedding)

How transformers learn about order

- The **goal of RoPe is to** give self-attention a sense of **token order** without fixed positional vectors. It relies on a maximum context length N.

- The idea of a RoPe is to encode the position "l" of a token $x_l \in \mathbb{R}^d$via

$x_l = [x_l^r, x_l^i], x_l^r, x_l^i \in \mathbb{R}^{d/2}$ and to define $\hat{x}_l = (x_r + \mathbf{i}x_i)e^{2\mathbf{i}\pi\frac{l}{N}}$

- Why is the positional encoder like this? Fourier flavour and covariance to translation!

Let $(Lx)_l \triangleq x_{l+1}$ then $\widehat{Lx}_l = e^{\frac{2\pi\mathbf{i}}{N}}.\hat{x}_l$ (translating is a multiplication)

Also, relative positions are well encoded: $\hat{x}_l^* \hat{x}_k = e^{i\frac{2\pi(k-l)}{N}} x_l^\top x_k$

Attention layers

Let the "stable implementation" of the softmax:

$$\sigma : \mathbb{R}^{L'} \to \mathbb{R}^{L'}, \quad \sigma(x)_i = \frac{\exp(x_i - \max_k x_k)}{\sum_{j=1}^{L'} \exp(x_j - \max_k x_k)}$$

The attention layer processes:

Query: $q_1, ..., q_L \in \mathbb{R}^d$

Key: $k_1, ..., k_{L'} \in \mathbb{R}^d$

Value: $v_1, ..., v_{L'} \in \mathbb{R}^{d'}$

Attention →

Output:

$y_1, ..., y_L \in \mathbb{R}^{d'}$

Then, perform the attention step:

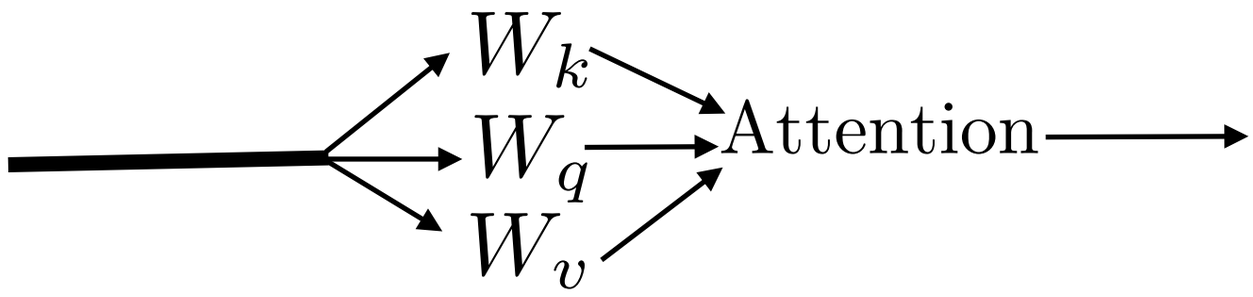
$$y_l = \sum_{l'=1}^{L'} \sigma(q_l^\top k_{l'})_{l'} v_{l'} \qquad \text{or more simply} \qquad y = \sigma(qk^\top)v$$

FLOPs: $LL'(2d + 5 + 2d')$

- Note: Including the previously introduced RoPE, these operations are not permutation-invariant, they preserve positional information.
- Note: During training, $L = L'$.

Multi-Head Attention

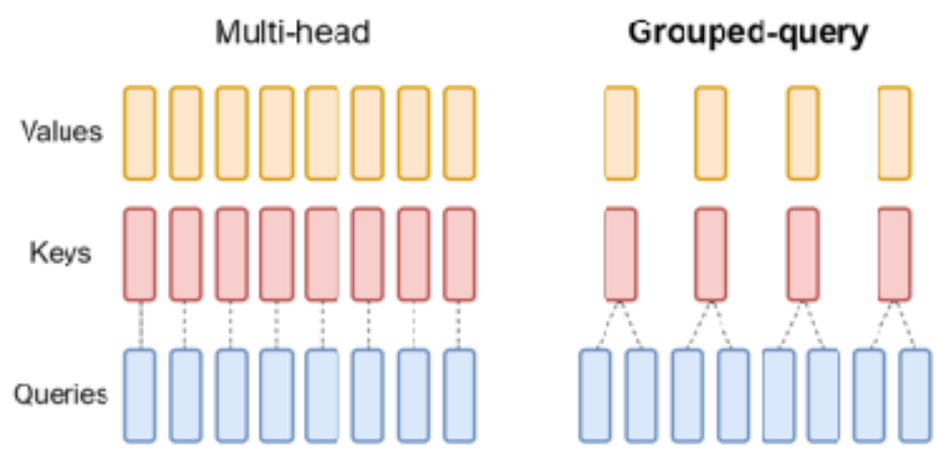
- Attention is always combined with linear layers



- Some structural constraints and redundancies (to save FLOPs) can be introduced into these layers while still maintaining good expressivity.
- A typical design uses H_q independent attention modules ("heads") for the queries, while only H_{kv} key-value heads are instantiated and shared across several query matrices.

$$W_q^{h'} \in \mathbb{R}^{d \times m}, h \leq H_q$$

$$W_k^h, W_v^h \in \mathbb{R}^{d \times m}, h \leq H_{kv},$$



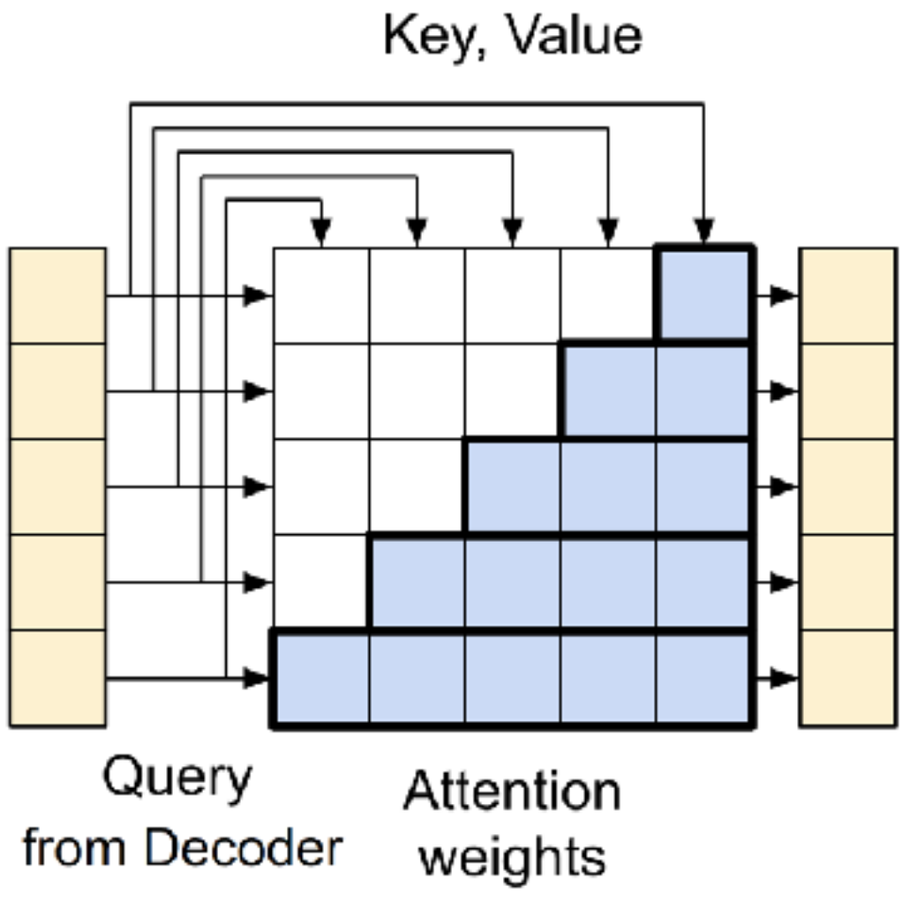
- Self-attention: typically $H_q = H_{kv} = 1, m = d$
- Multi-head Attention $H_{kv} = H_q > 1, H_q m = d$
- Group Query Attention: $H_q > H_{kv} > 1, H_q m = d$

Causal masks in Decoder-only

- The goal of a causality mask is to make training match inference: the model should not use future tokens to predict the next token ("auto-regressive" behaviour).

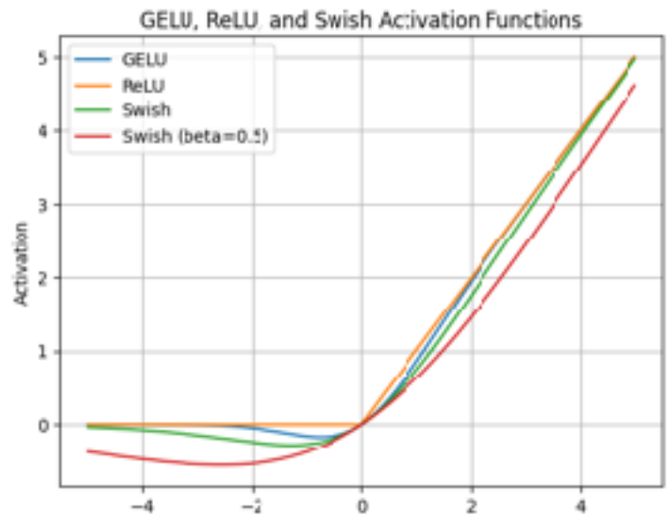
- A binary mask is applied to attention scores

- Position i can only attend to positions $\leq i$.



MLP layer

- Fix a pointwise non linearity: $\sigma : \mathbb{R} \to \mathbb{R}$ so that $\sigma(0) = 0$



- A successful strategy, used in Llama family models, is to "gate" the activations and to slightly increase the dimensionality.

$$W_b, W_q \in \mathbb{R}^{d \times m} \quad \text{and} \quad W_o \in \mathbb{R}^{m \times d}$$

so that $$y = W_o(\sigma(W_q x) \odot (W_b x))$$

- Typically, m is an expansion factor that increases dimensionality (expressiveness)

RMS norm

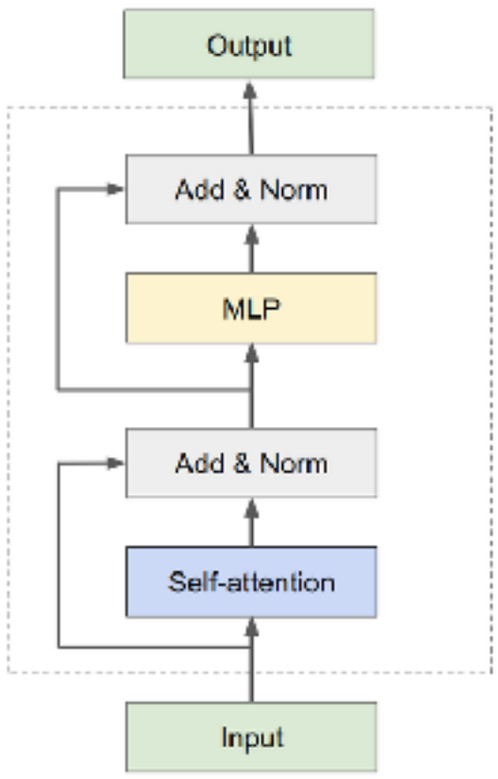
- Root Mean Square norm is defined as a normalisation used between layers, with a learnable parameter $\gamma \in \mathbb{R}^d$ and $\epsilon > 0$.

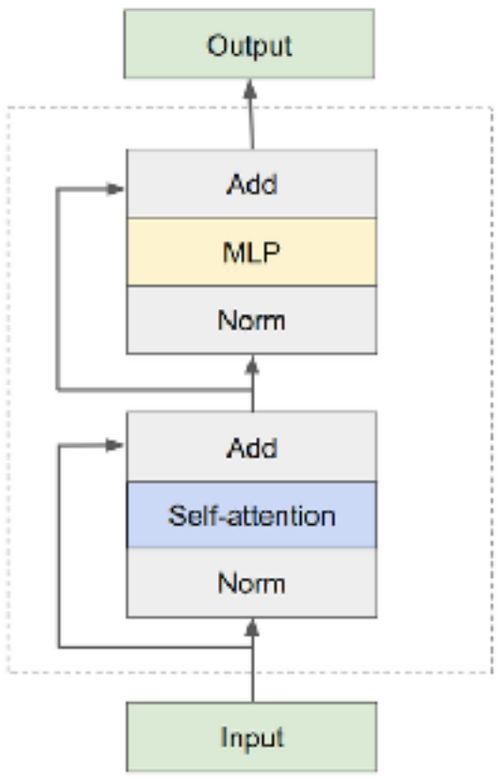
$$\text{RMSNorm}(x) = x \odot \frac{\gamma}{\sqrt{\varepsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}},$$

- RMS norm is much more amenable to parallelisation than a batch norm!! (no cross-communications between samples)

- Two strategies (post-norm seems to take the lead)

Post-norm





Pre-norm

Forward: the "$2P$" rules

- Let's provide a simple heuristic to quickly know the number of FLOPs of a model with P parameters, per token.

- Typically, the context length L is much smaller than the ambient dimension d.

- Thus the FLOPs are mostly due to the linear layers rather than the attention.

- If we neglect also the FLOPs due to the RMS, we end up with

$$\text{FLOPs/token} = 2P$$

What about the backward?

Complexity of the backward pass of a linear layer

- Let: $x \in \mathbb{R}^d,\ A \in \mathbb{R}^{n \times d}$ and $\ell : \mathbb{R}^n \to \mathbb{R}$

Fix $f(x, A) = \ell(Ax)$ so that $\nabla \ell(Ax) \in \mathbb{R}^n$

- What is the fully complexity of a GEMM (General Matrix Multiply)?

Forward: $f(x, A) = \ell(Ax) \longrightarrow 2nd$

Backward of the inputs: $\nabla_x f(x, A) = A^\top \big(\nabla \ell(Ax)\big) \in \mathbb{R}^d.$

$\longrightarrow 2nd$

Backward of the weights: $\nabla_A f(x, A) = \big(\nabla \ell(Ax)\big)\, x^\top \in \mathbb{R}^{n \times d},$

$\longrightarrow 2nd$ (it's aggregated over a mini batch)

The total complexity if then $6nd$

Backward in Transformers

- Consequently, as for the forward pass, the methodology to have an approximation of the number of flops lead to 6P FLOPs per token for the backward pass.

- Note that some methodology do not take in account the embedding layer, or use the exact number of FLOPs including the context length.

Ref.: DeepSeek LLM Scaling Open-Source Language Models with Longtermism

Non embedding parameter

$$6N_1 = 72\, n_{\text{layer}}\, d^2_{\text{model}}$$

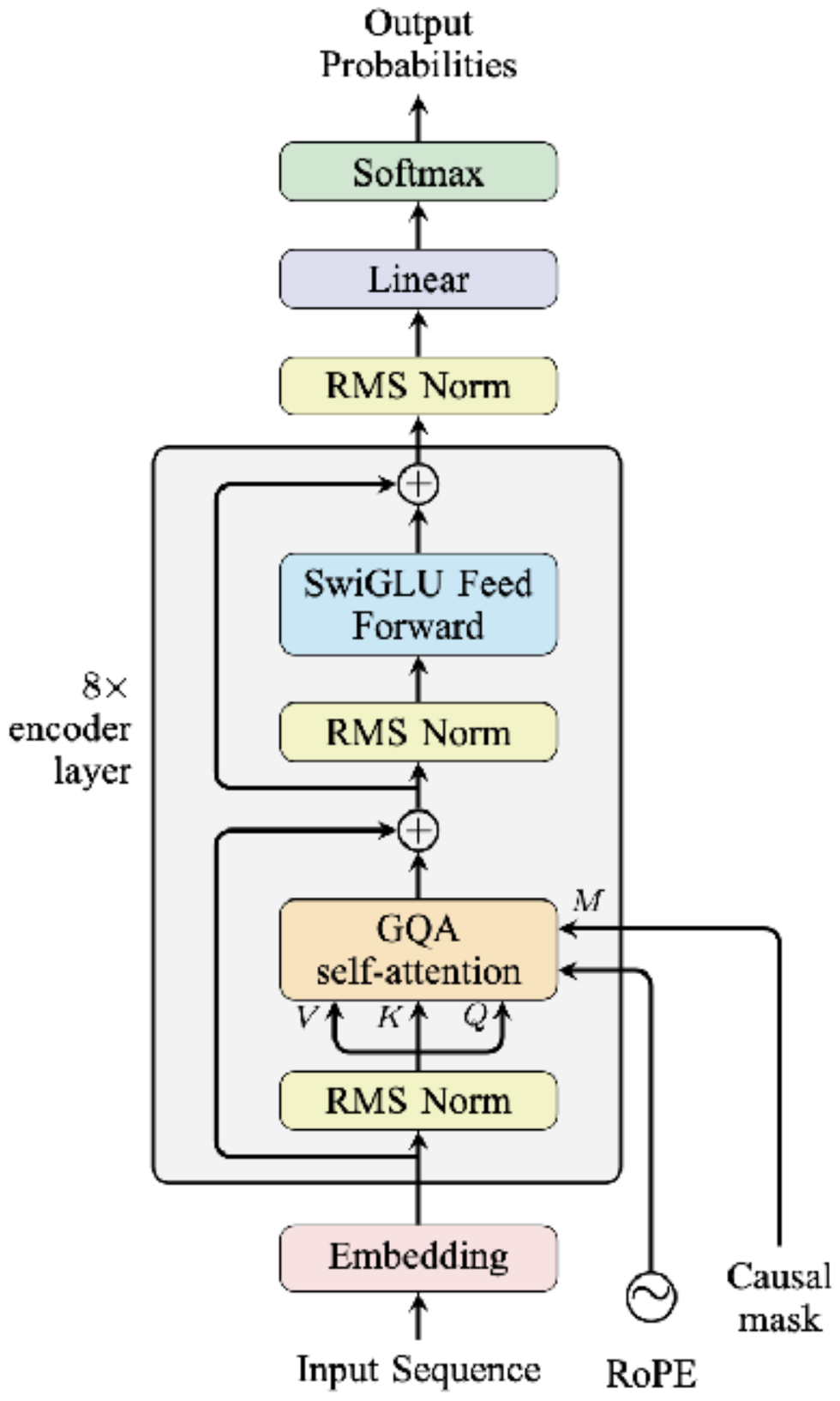
Total parameters

$$6N_2 = 72\, n_{\text{layer}}\, d^2_{\text{model}} + 6\, n_{\text{vocab}}\, d_{\text{model}}$$

$$M = 72\, n_{\text{layer}}\, d^2_{\text{model}} + 12\, n_{\text{layer}}\, d_{\text{model}}\, l_{\text{seq}}$$

"non-embedding FLOPs per token"

Full Anatomy of a Decoder-Only Transformer



Training Recipes

Cross-validation is costly

- On small-scale tasks like **CIFAR-10**, we can cross-validate most hyper-parameters and run many experiments on a short time windows

- For training **LLMs**, the situation is very different: full cross-validation is computationally impossible and each run is extremely expensive (many GPUs, long training time)

- We must rely on **heuristics** to *predict* good hyperparameters within a limited and affordable compute budget.

Core ingredients to train a LLM

- We've chosen the **architecture type** and describe the **data,** what's le
  - Which optimizer?
  - Which learning-rate schedule?
    - For how many tokens/steps?
    - How should we initialise the weights?
- Let's come back to the initial training loop.

The training Loop

- Let's unroll one optimizer step to make each operation explicit:

initialise layers $\theta_0 \sim \Gamma_0$

for t=1… T

sample $\xi_t^i \sim \Xi^i$, on worker i:

$$g_t = \texttt{Clip}_m\left(\frac{1}{n}\sum_{i=1}^{n} \nabla f(\theta_t, \xi_t^i)\right)$$

$$\theta_{t+1}, s_{t+1} = \texttt{AdamW}_{\eta_t,\lambda,\epsilon,\beta_1,\beta_2}(\theta_t, g_t, s_t)$$

List of the remaining hyper-parameters:

Γ_0 : initialisation rules

mostly fixed:

Explicit regularisation:

m: the magnitude of clipping

λ: weight decay (ℓ^2 regularisation)

Optimizer hyper parameters:

$\epsilon, \beta_1, \beta_2$: AdamW parameters

extrapolation/cross-validation:

Implicit regularisation:

η_t : learning rate

n : batch size

T : maximal step

AdamW vs Adam

$$g_t = \nabla f(\theta_{t-1})$$

$$G_t = g_t \qquad G_t = g_t + \lambda\theta_{t-1}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) G_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) G_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_t = \theta_{t-1} - \eta_t \cdot \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \cdot \theta_{t-1} \right) \qquad \theta_t = \theta_{t-1} - \eta_t \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- Contrary to Adam, AdamW uses a decoupled weight decay, an this makes AdamW invariant to rescaling.

- To remove instability (exploding gradients), clipping is often use so that:

$$\texttt{Clip}_\alpha(g) = \begin{cases} g, & \text{if } \|g\|_2 \leq \alpha, \\ \dfrac{\alpha}{\|g\|_2}\, g, & \text{if } \|g\|_2 > \alpha. \end{cases}$$

- The typical parameters choices are (and they are probably the last things to cross validate if you haven't identified divergence)

$$\lambda = 10^{-1} \qquad \epsilon = 10^{-8} \qquad \beta_1 = 0.9 \qquad \alpha = 1 \qquad \beta_2 = 0.95$$

Choosing the Learning Rate Schedule

- Assume we fix the optimizer hyperparameters, the batch size, and the total number of steps. What remains is to choose the learning rate schedule.

- The typical strategy to pick a learning rate is to use a schedule that includes a warmup period T_w and a cosine decay rule:

$$\eta(t) = \begin{cases} \eta_{\text{start}} + (\eta_{\text{peak}} - \eta_{\text{start}})\,\dfrac{t}{T_w}, & 0 \le t < T_w, \\ \eta_{\min} + \dfrac{\eta_{\text{peak}} - \eta_{\min}}{2}\left(1 + \cos\left(\pi\,\dfrac{t - T_w}{T - T_w}\right)\right), & T_w \le t \le T. \end{cases}$$

- Typical values are $\eta_{\text{start}} = 0, \eta_{\text{end}} = 10^{-2} \cdot \eta_{\text{peak}}$ and $T_w = 10^{-2} \cdot T$

- The remaining variable to specify are : the batch size, the number of steps and the peak learning rate η_{peak}: they depend on the number of parameters P.

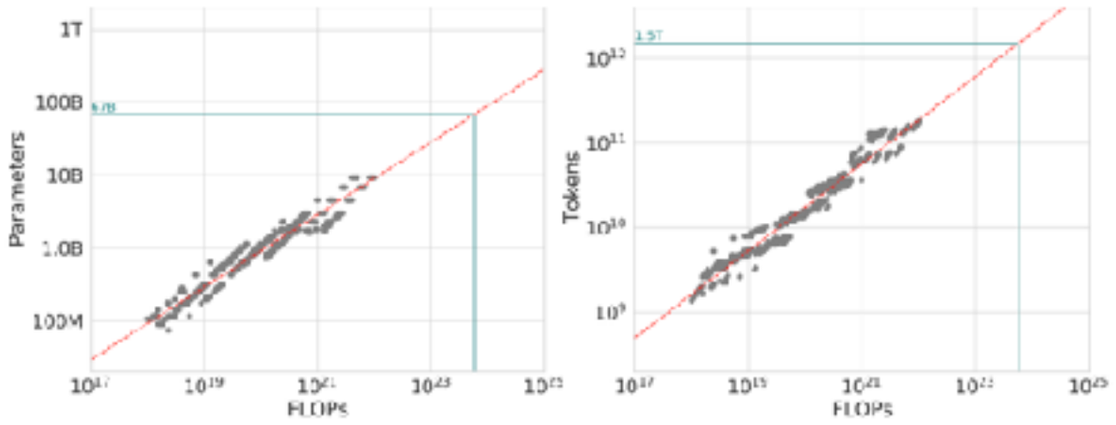
Chinchilla's rule

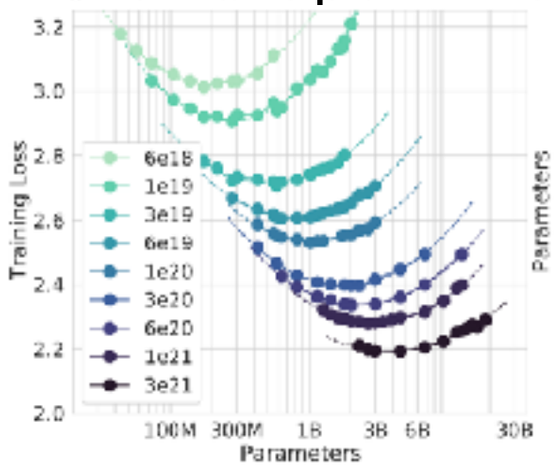
- The "scaling law" describe how performance improve when compute C, total number of tokens T or model size P increase. The dependency are often expressed as power law.

Ref.: Scaling Laws for Neural Language Models

$$P^*, T^* = \arg \min_{P,T,\text{FLOPs}=C} \texttt{loss}(P, T)$$

- Three strategies are proposed:
  - Fix P, compute for multiple T
  - Isoflop, fix C and optimizer over P, T





- Parametrize the loss as $\ell = L + \frac{A}{P\alpha} + \frac{B}{T^\beta}$ $\qquad C = 6PT$
- The conclusion of Chinchilla's rule is that

Ref.: Training Compute-Optimal Large Language Models

$$T^* \approx 20P^*$$

However, this does not guide the design of the learning rate or batch size.

The DeepSeek approach

- Another approach, probably more frugal:
  - Use a small budget to estimate small scale-parameters: fix a model, vary tokens size and derive the optimal learning rate and batch size

$$\eta^*_{\text{peak}} = A \cdot C^{-\alpha}$$

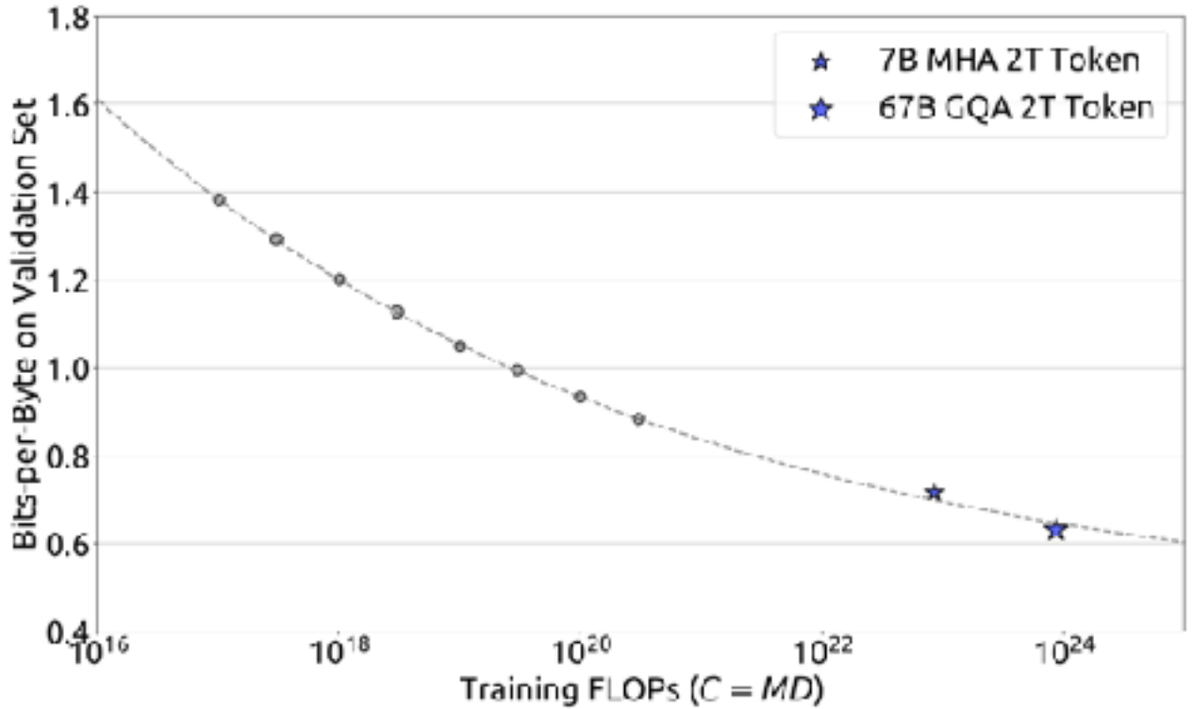
$$n^* = B \cdot C^{\beta}$$

- Using those hyper parameters, estimate varying T and P, under a similar small budget

$$P^* = \Gamma \cdot C^{\gamma}$$

$$T^* = \Delta \cdot C^{\delta}$$

$$\text{with } \gamma + \delta = 1$$



Initialisation via muP

Initialisation in LLMs

- Initialisation has to keep activations and gradients in reasonable range, avoid one layer dominating, behave well with scale and depth

- In many models (e.g., DeepSeek, Llama3) activation initialisation is given by:

$$\mathcal{N}(0, \sigma^2 \mathbf{I}) \quad \text{where} \quad \sigma = 0.02$$

- At the same time, layers often include scalar normalizations (e.g., in the attention layer) and rescalings that influence the effective initialization. In fact, these effects can be understood using **invariance rules**.

Consider: $f(\theta)$ minimised with $\theta_{t+1} = \theta_t - \eta \nabla f(\theta)$ and $\theta_0 \sim \mathcal{N}(0, \sigma^2)$

then any re-parametrization:

$$\{(f^\lambda : \theta \to f(\lambda\theta), \frac{\sigma}{\lambda}, \frac{\eta}{\lambda^2}), \lambda > 0\} \quad \text{leads to the same result}$$

proof:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\lambda^2} \lambda . \nabla f(\lambda . \theta_t) \quad \theta_0 \sim \mathcal{N}(0, \frac{\sigma^2}{\lambda^2}) \quad \text{so:} \quad \theta_t^\lambda = \lambda . \theta_t$$

Mu-P rules

- This means there is an intrinsic notion of rescaling parameters, which depends on the optimizer.

- Interestingly, there exist initialization rules that allow **scaling with depth and width**, such as **Mu-P**

- The idea is to find the initialization and learning rate (with the rescaling constrained) that lead to the largest effective update while keeping training stable.

- The nice thing about Mu-P is that it provides simple rules for how to scale layer size and adjust learning rates as layer width grows.

Mu-P desirata

- When width grows infinitely:
  - Activations vector should have $\Theta(1)$-sized coordinates
  - The output of the neural network should be $\mathcal{O}(1)$.
  - The parameter update $\Delta\theta$ should be maximal.
  - This is the Mu-P initialisation.

Works well in practice!

Conclusion

- Let’s move toward a lab on "tiny scaling laws".