Efficient training, fine-tuning, and inference of (mostly) Large Language Models

Constantine Dovrolis

Director of Computational Science & Technology Research Center, CYI

Adjunct Professor of Computer Science, Georgia Tech

XM Chair in Artificial Intelligence at Univ. of Cyprus (starting in Sept'26)

Georgia Tech

THE CYPRUS INSTITUTE

RESEARCH•TECHNOLOGY•INNOVATION

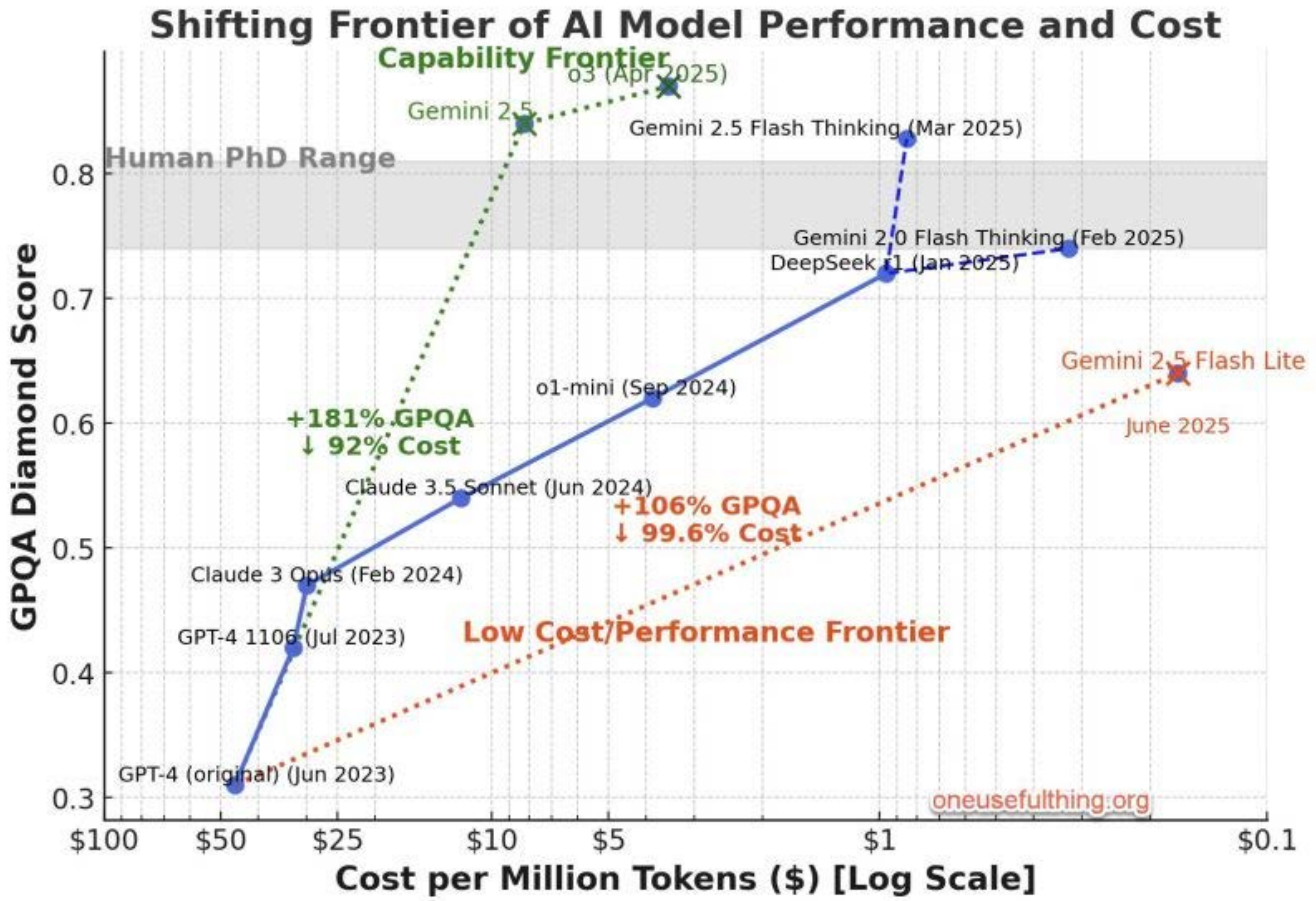
University of Cyprus

Outline

1. Efficiency objectives and token economics
2. Designing and training the base model
3. Post-training and adaptation
4. Efficient attention & KV-caching
5. Quantization for deployment
6. Serving systems and application efficiency
7. Co-design and practitioner choices

Section 1 — Efficiency as Token Economics

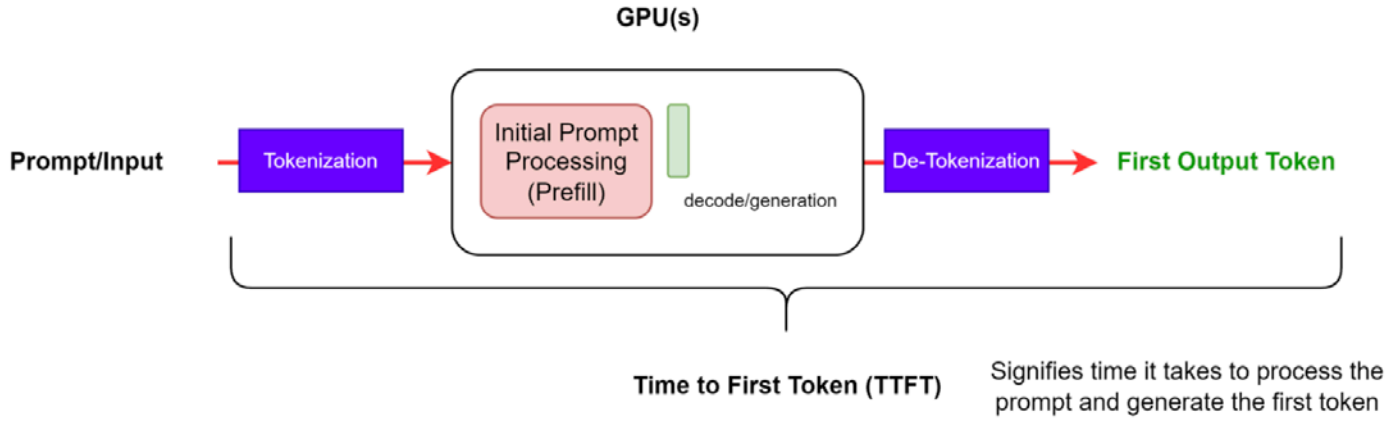
It is not only about performance – cost is increasingly a dominant factor!

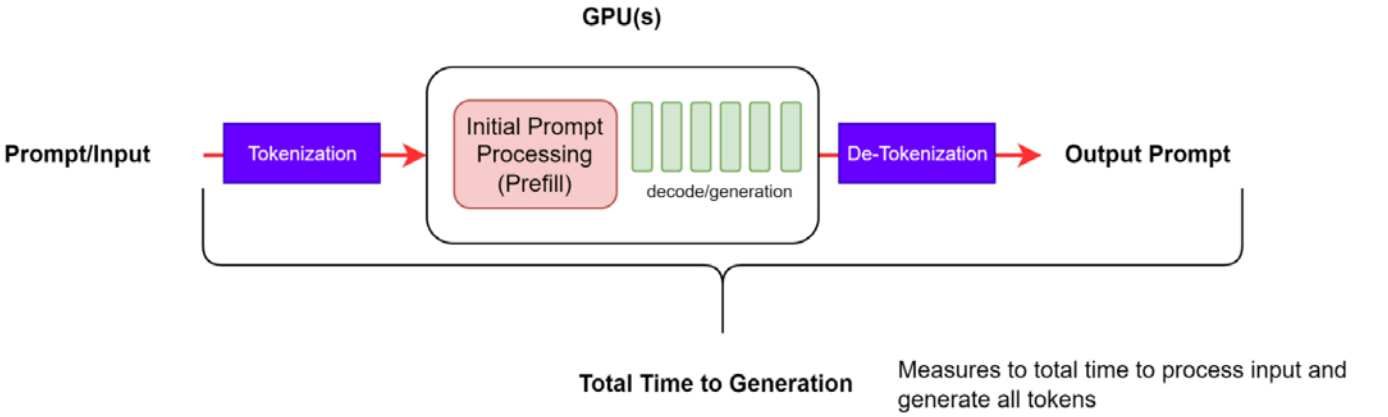


Source: https://www.linkedin.com/posts/emollick_no-signs-of-an-end-to-rapid-gains-in-ai-ability-activity-7340929711537086464-avc8/

Metrics in practice: What Are We Optimizing?

Latency, throughput, memory, utilization, and cost are related but not interchangeable





- Interactive applications care about Time To First Token (TTFT), Inter-Token Latency (ITL), and p95/p99 tail latency.
- Batch and data-generation workloads care more about throughput: tokens per second, requests per second, and cost per output token.
- Training efficiency uses different metrics: FLOPs, model FLOPs utilization (MFU), memory footprint, and communication overhead.

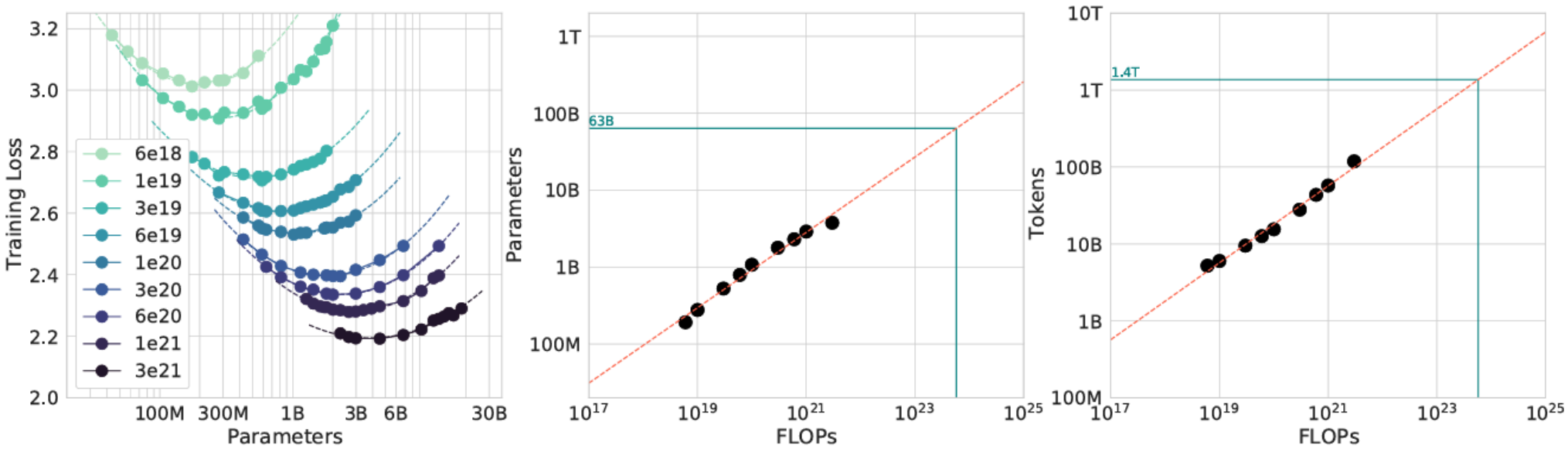
Source URL: https://developer.nvidia.com/blog/llm-benchmarking-fundamental-concepts/

Section 2 — Designing and training the base model efficiently

- Compute-optimal scaling: right model, enough tokens
- Modern pre-training data pipelines
- Mixture-of-Experts: sparse compute, dense complexity
- Where training memory goes
- Distributed training: different forms of parallelism
- Activation checkpointing and recomputation
- Low-precision training: BF16, FP8, emerging FP4

Compute-optimal scaling: right model, enough tokens

For a fixed FLOP budget, parameters and training tokens are jointly optimized.

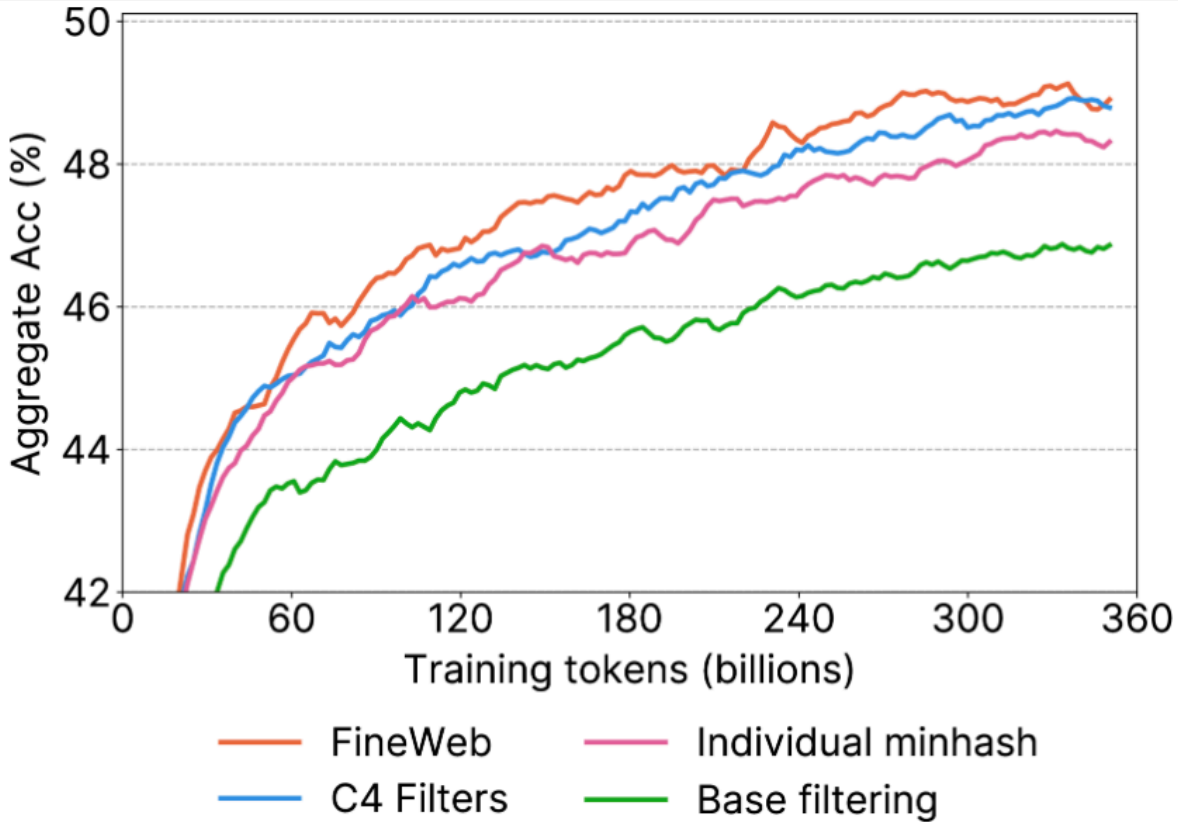


- Scaling model size alone can waste compute.
- The data budget matters just as much.
- The Chinchilla result: compute-optimal training scales model size and token count together.
- In practice: many open models favor smaller dense models trained longer on more curated data.

Source: https://arxiv.org/pdf/2203.15556

Modern pre-training data pipelines

Raw web text becomes training data through many lossy decisions.

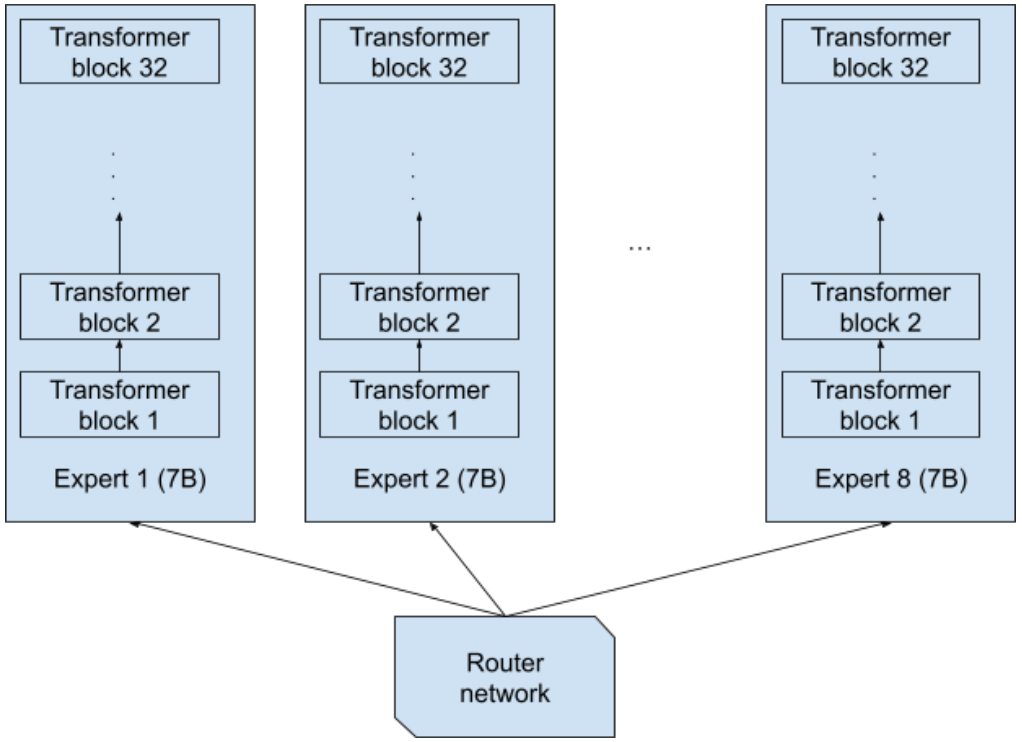


Source: https://arxiv.org/html/2406.17557v1

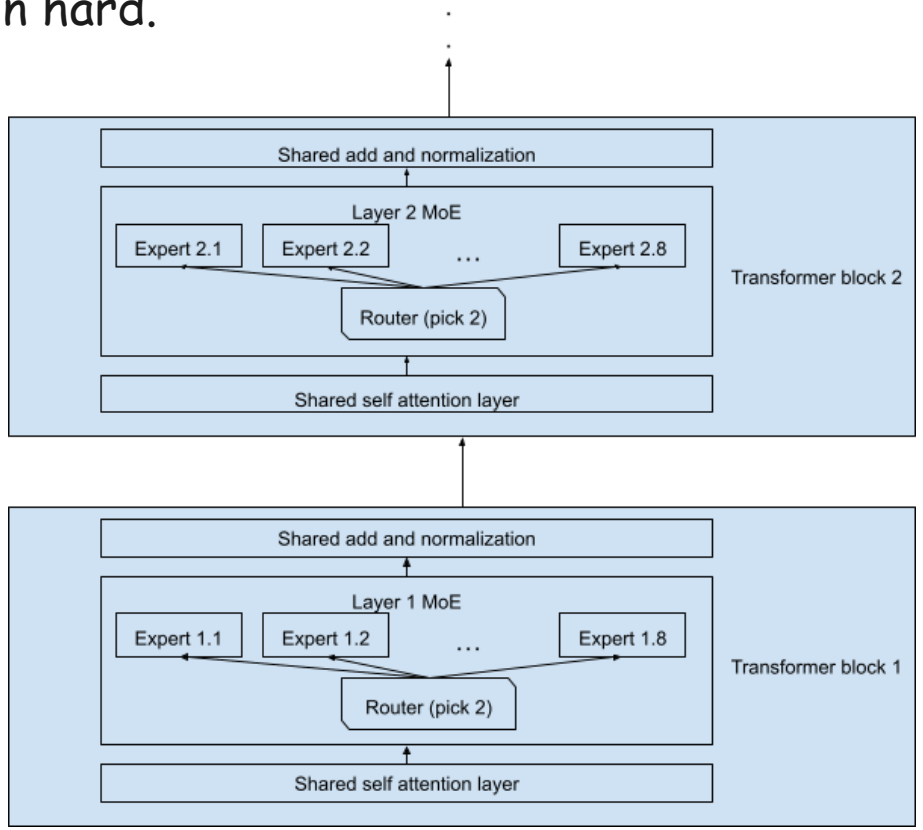
- Pipeline stages:
  crawl → extract → language identification → filter → deduplicate → decontaminate → mix → pack.
- The hard part is removing junk without removing useful diversity.
- In practice: Hugging Face DataTrove/FineWeb and NVIDIA NeMo Curator are production-style examples.

Mixture-of-Experts: sparse compute, dense complexity

Activate only part of the model per token (but pay new costs in routing and communication)



- Mixture-of-Experts (MoE) replaces some Feed-Forward Network (FFN) layers with multiple experts and a router.
- Sparse MoE activates only the top-k experts per token; compute is sparse, but memory footprint and expert communication remain hard.



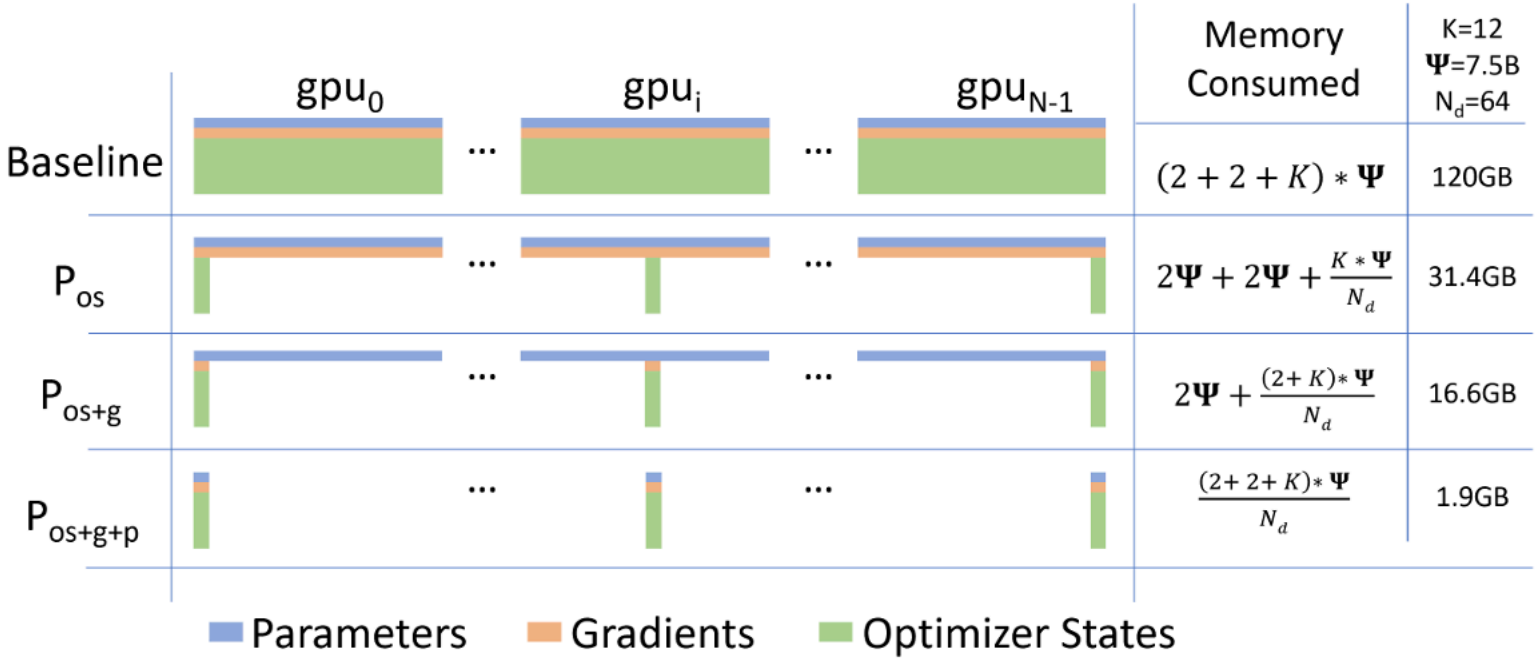
- In practice, Mixtral activates two of eight experts per token; DeepSeek-V3 uses finer-grained MoE with 37B active parameters out of 671B total.

https://developer.nvidia.com/blog/applying-mixture-of-experts-in-llm-architectures/

Where training memory goes

Weights are only part of the memory problem.

- Training stores parameters, gradients, optimizer states, activations, temporary buffers, and communication buffers.
- Zero Redundancy Optimizer (ZeRO) and Fully Sharded Data Parallel (FSDP) reduce redundant model-state memory.
- In practice: sharding, offloading, and checkpointing determine the largest trainable model per GPU

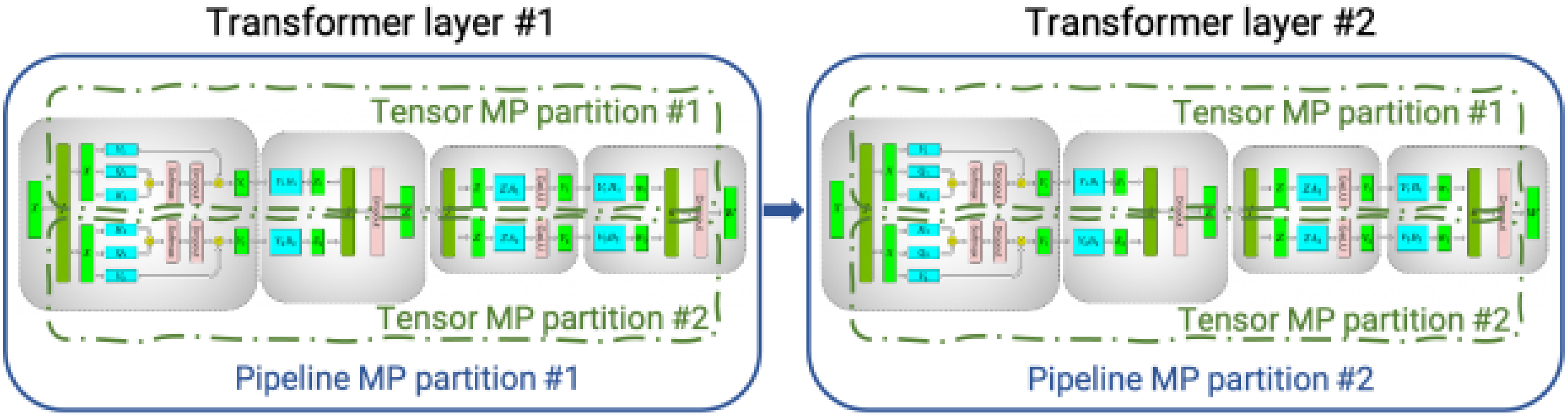


Source: https://arxiv.org/pdf/1910.02054

Distributed training: different forms of parallelism

Large training runs combine multiple kinds of parallelism, not one trick.

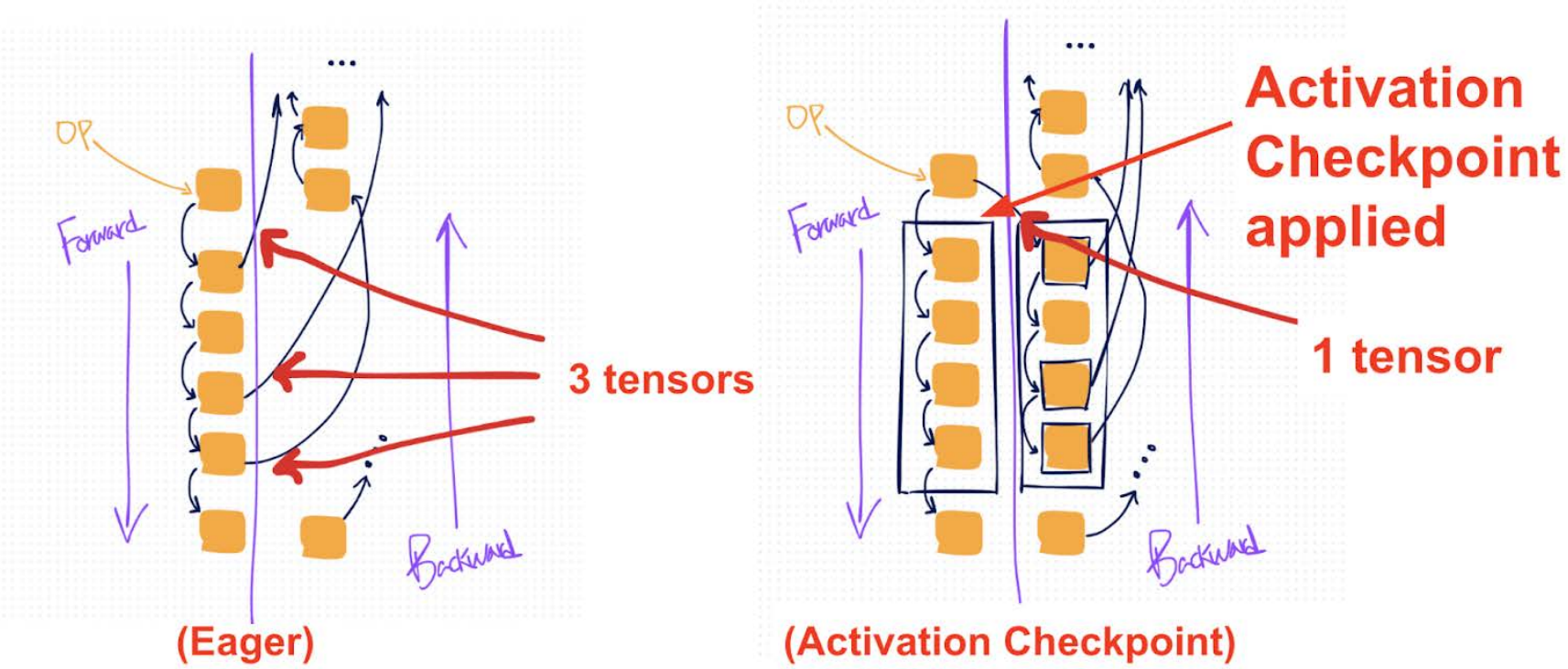
- Data Parallelism (DP) replicates the model; Tensor Parallelism (TP) splits layers; Pipeline Parallelism (PP) splits depth.
- Sequence/Context Parallelism splits long sequences; Expert Parallelism (EP) splits mixture-of-experts (MoE) layers.
- In practice: Megatron-Core, DeepSpeed, and PyTorch FSDP combine these with topology-aware communication.



Source: https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

Activation checkpointing and recomputation

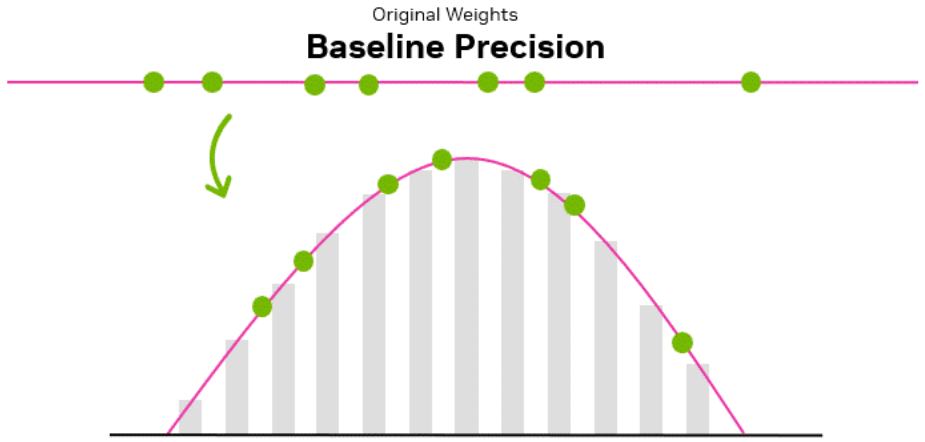
Trade extra compute for lower activation memory.

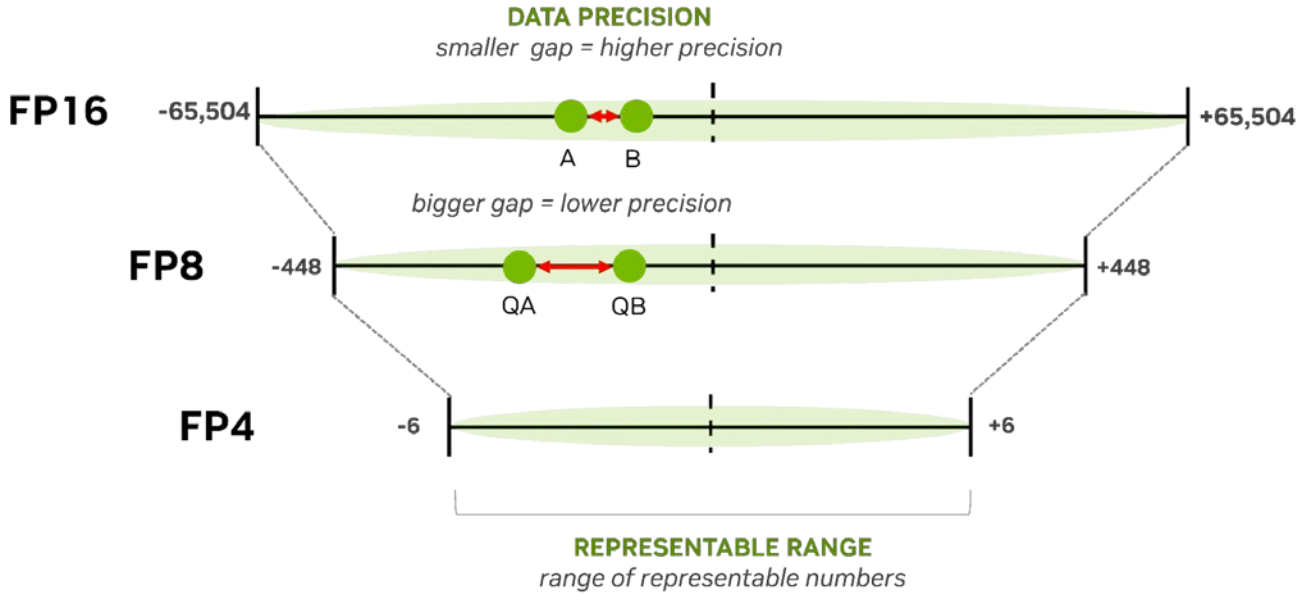


Source: https://pytorch.org/blog/activation-checkpointing-techniques/

- Activation Checkpointing (AC) saves only selected tensors during the forward pass.
- Missing activations are recomputed during the backward pass, reducing peak memory.
- In practice: enabled when sequence length or batch size is memory-bound.

Numerical formats: range, precision, and hardware support





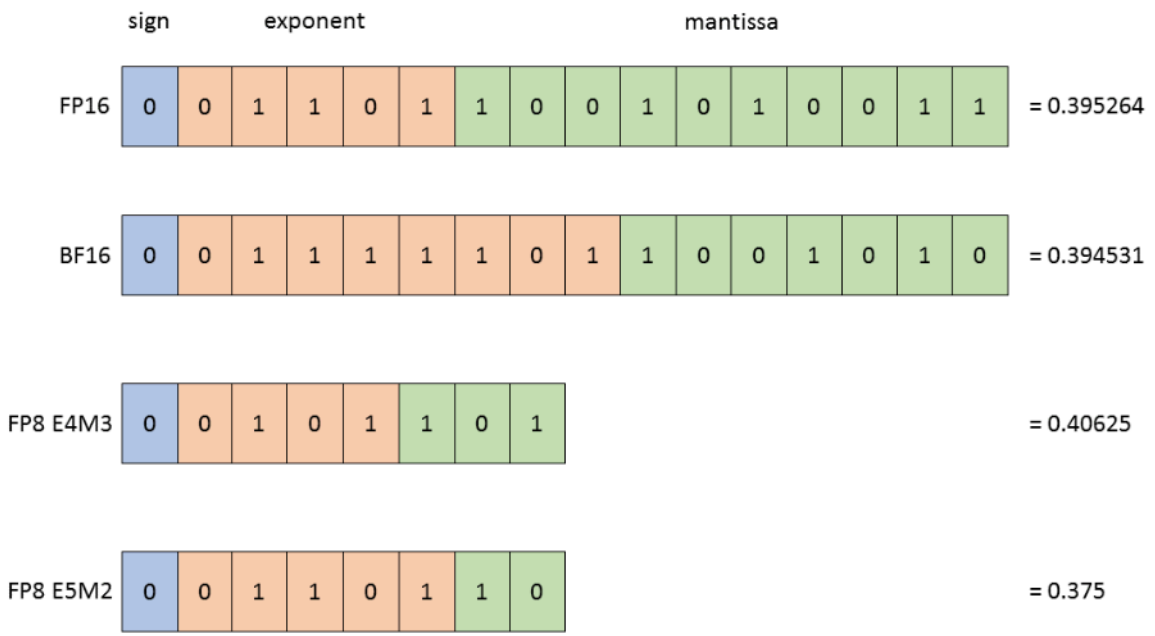
Source: https://developer.nvidia.com/blog/optimizing-llms-for-performance-and-accuracy-with-post-training-quantization/

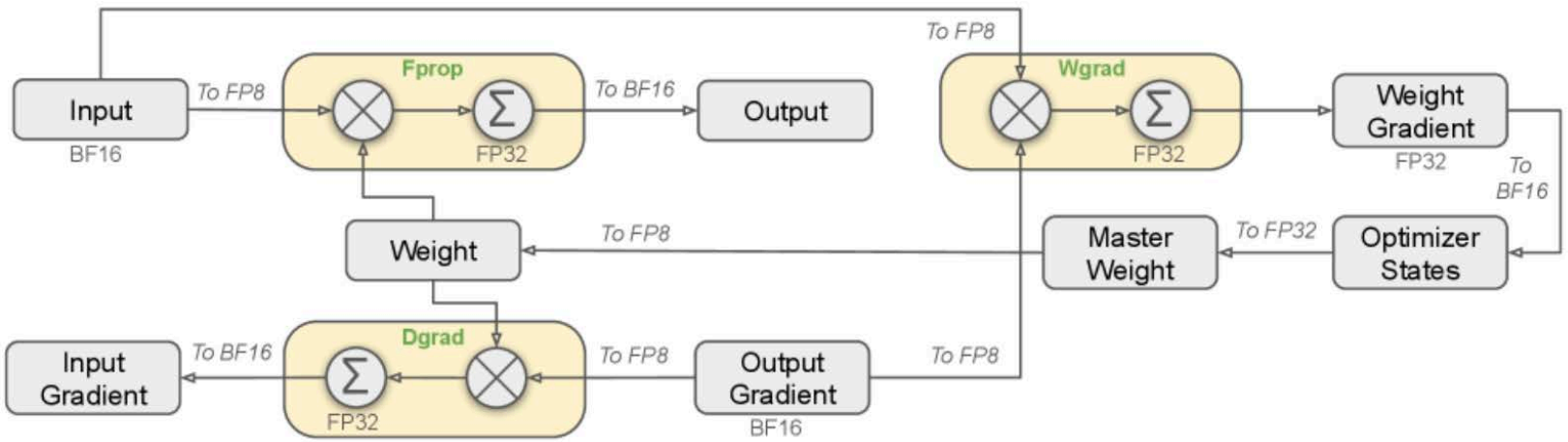
Low-precision training: BF16, FP8, emerging FP4

Precision is selected per tensor and per operation, not globally.

- Brain Floating Point 16-bit (BF16) is a common baseline; 8-bit floating point (FP8) improves throughput on newer accelerators.
- Stable low-precision training needs scaling, accumulation, and selective higher-precision operations.
- In practice: NVIDIA Transformer Engine and DeepSeek-V3 does FP8 mixed-precision training; 4-bit floating point (FP4) is emerging.

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide





https://arxiv.org/html/2412.19437v1

Section 3 Post-training and adaptation efficiency

- Post-training approaches
- Synthetic and teacher-generated data
- Low-Rank Adaptation (LoRA)
- Quantized Low-Rank Adaptation (QLoRA)
- Distillation: move behavior into cheaper models

Post-training approaches

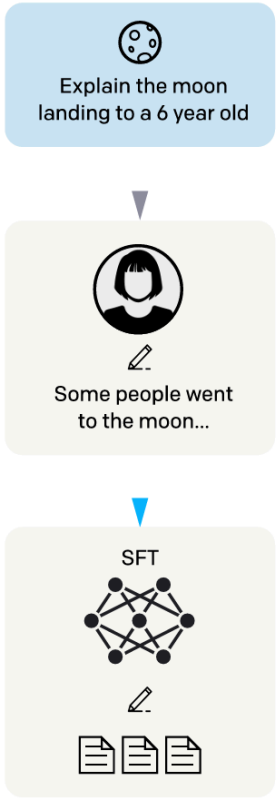
Step 1

Collect demonstration data, and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



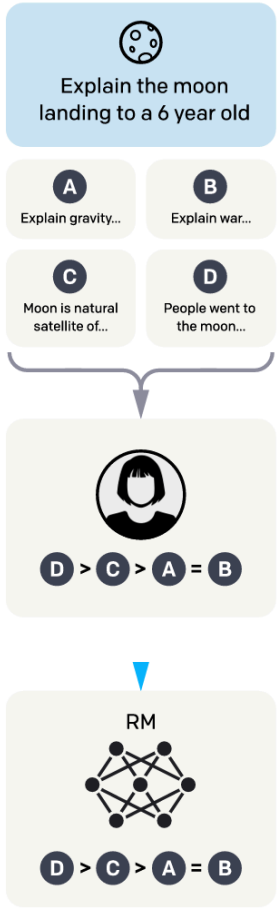
Step 2

Collect comparison data, and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

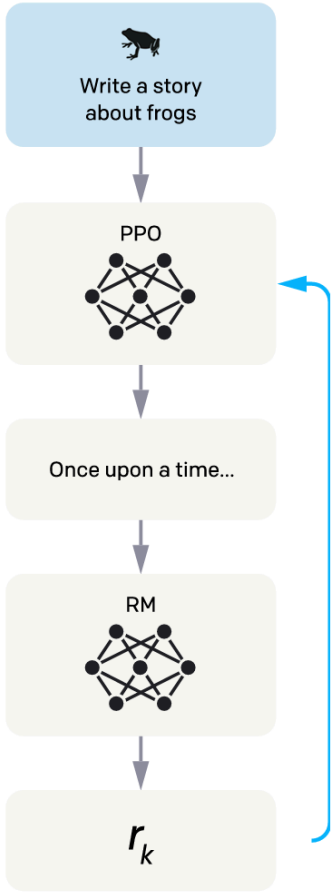
Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

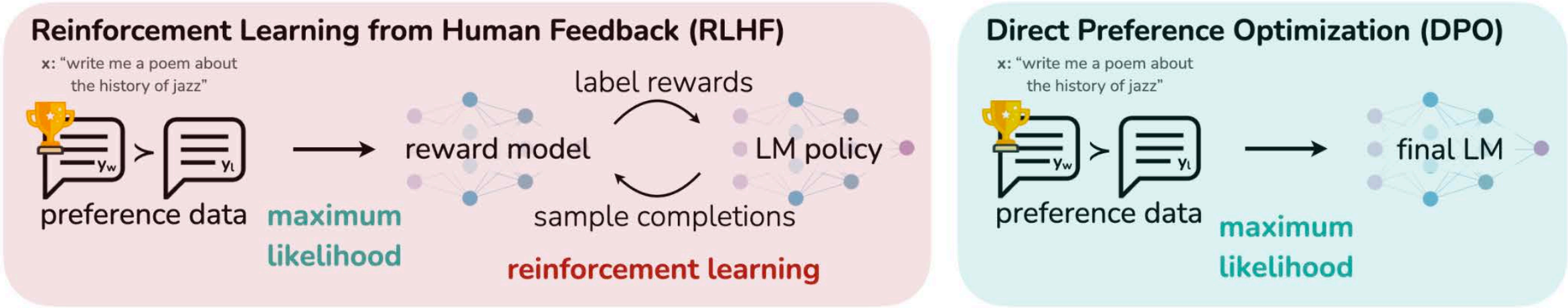
The reward is used to update the policy using PPO.



Source: https://cdn.openai.com/papers/Training_language_models_to_follow_instructions_with_human_feedback.pdf

Direct Preference Optimization (DPO)

Use preference pairs directly instead of training a separate reward model and running reinforcement learning.

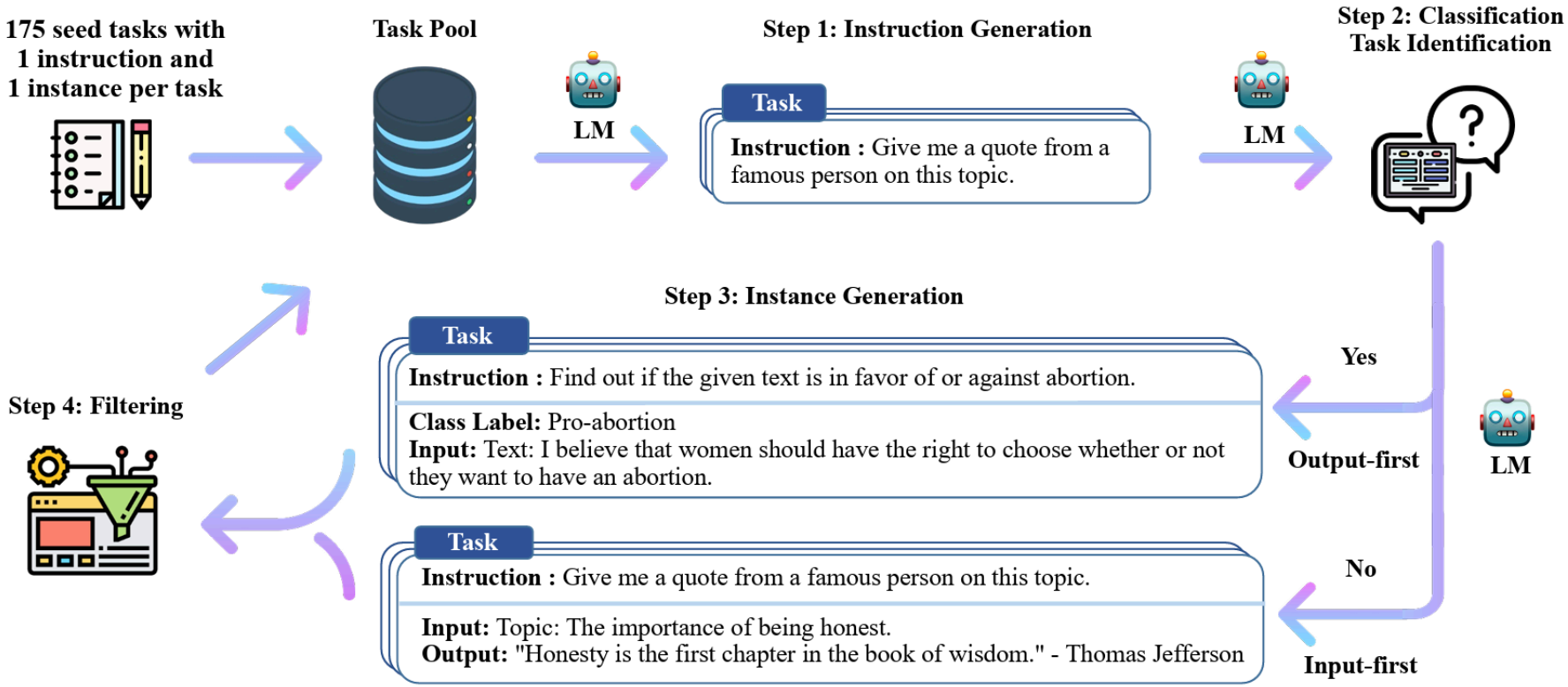


Source: https://ar5iv.labs.arxiv.org/html/2305.18290v3

- DPO trains on pairs: for the same prompt, increase likelihood of the chosen response relative to the rejected response.
- It removes the explicit Reward Model (RM) and reinforcement learning loop used in classic RLHF pipelines.
- In practice, DPO and related offline preference methods are popular because they fit normal fine-tuning workflows and are easier to stabilize.

Synthetic and teacher-generated data

Use expensive language models to create data that cheaper models can learn from.



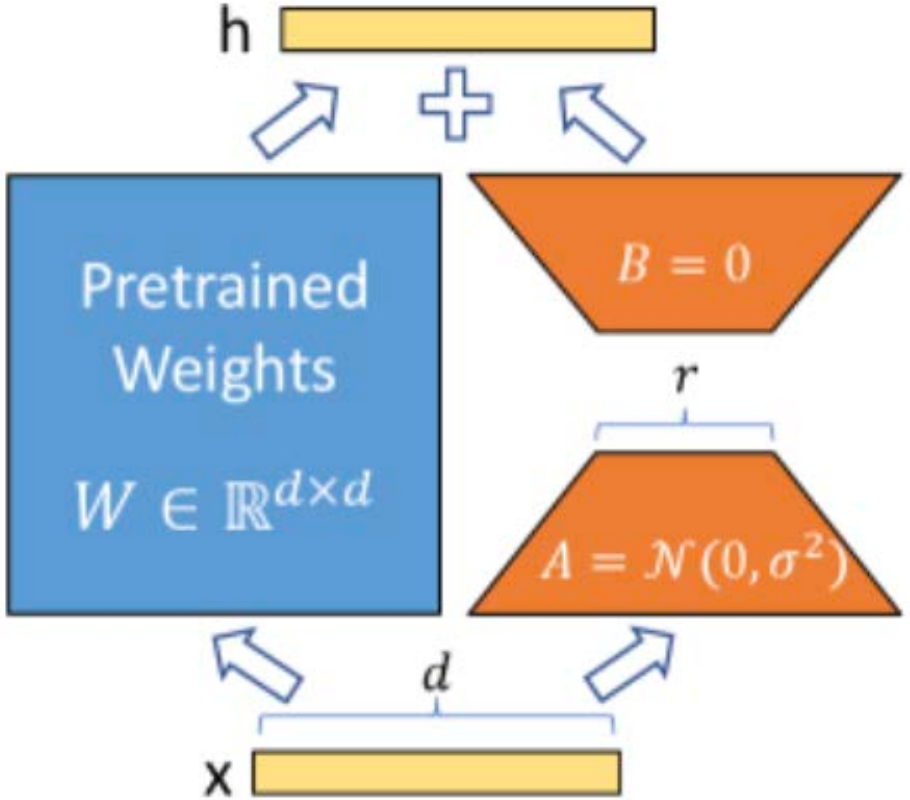
https://aclanthology.org/2023.acl-long.754.pdf

- Synthetic instructions, solutions, explanations, and labels can reduce human labeling cost.
- Filtering and diversity checks are as important as generation.
- In practice: used for instruction tuning, domain adaptation, evaluators, and data-quality classifiers.

Low-Rank Adaptation (LoRA)

Freeze the base model; train small low-rank updates that can be swapped, served, or merged.

• Instead of updating all weights, LoRA learns a low-rank update $\Delta W = BA$ while the original weight matrix W stays frozen.



Source: https://ar5iv.labs.arxiv.org/html/2106.09685v2

• Training memory drops because optimizer states and gradients are needed only for adapter parameters

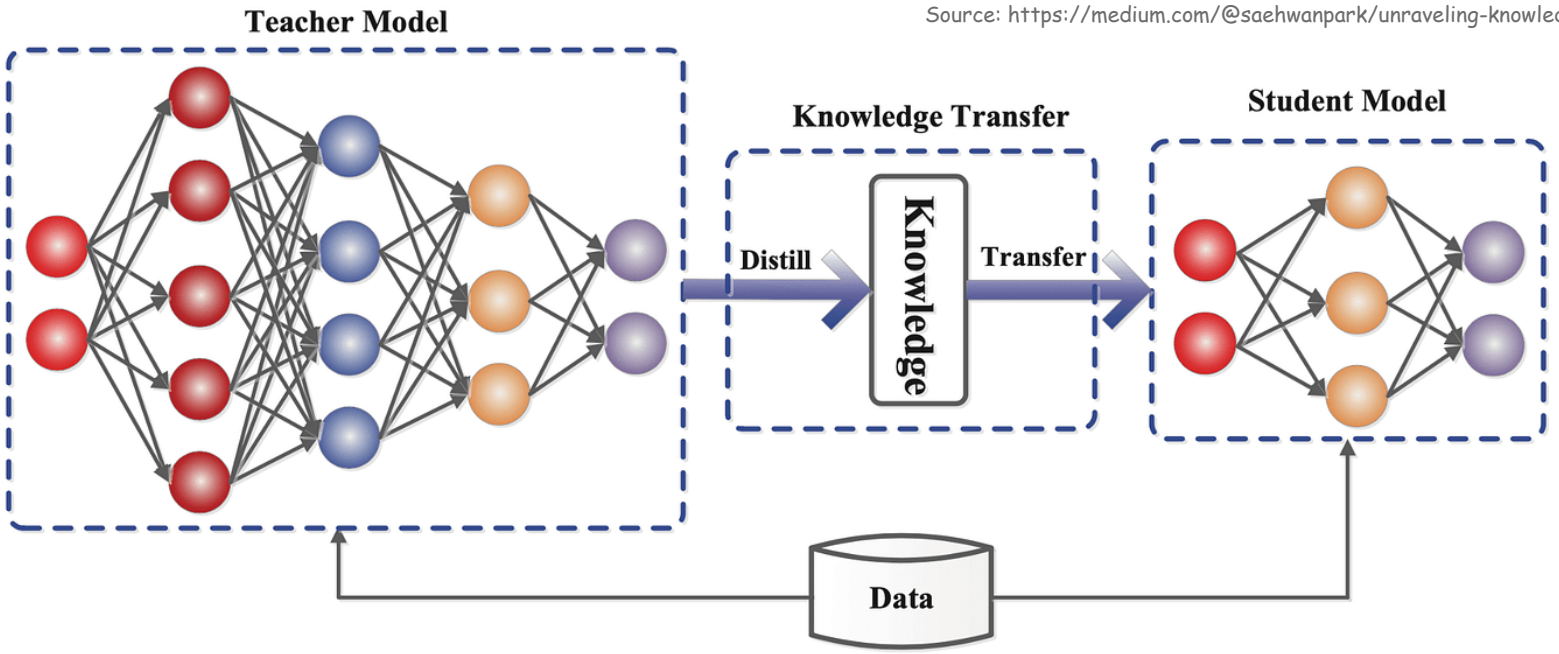
.

• In practice, LoRA is the default Parameter-Efficient Fine-Tuning (PEFT) method for domain adapters and customer/task customization.

Distillation: move behavior into cheaper models

Use an expensive teacher once; train smaller students that are cheaper to serve repeatedly.

Source: https://medium.com/@saehwanpark/unraveling-knowledge-distillation-in-ai-and-ml-d2695b1b214a



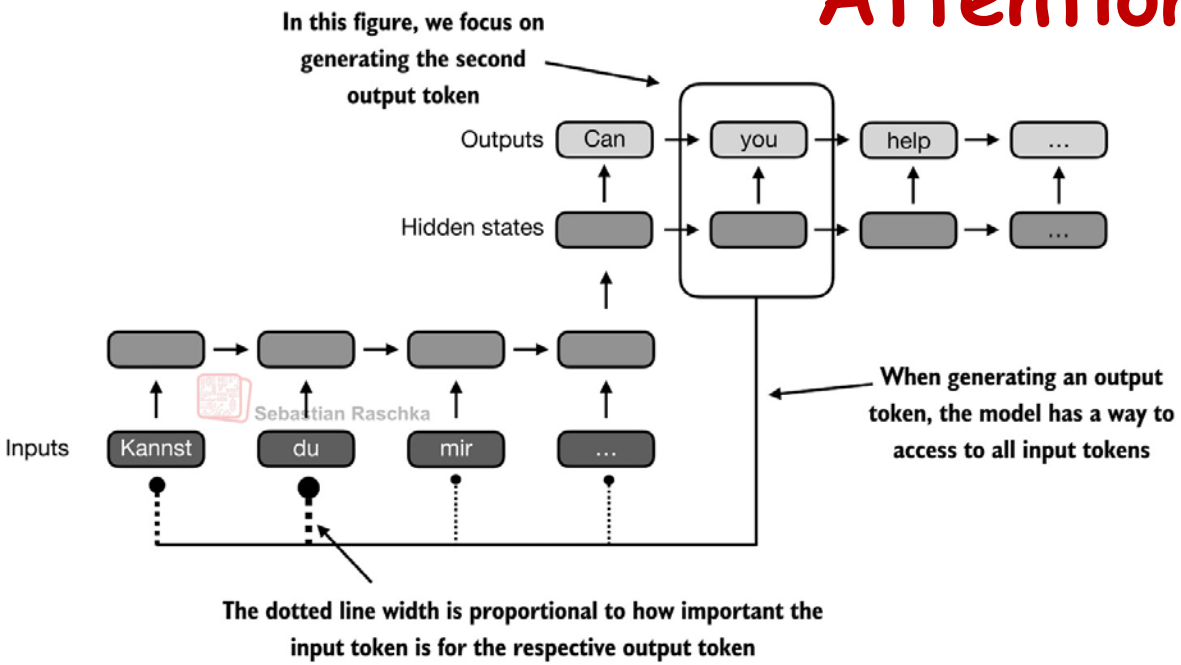
- A teacher model generates outputs, explanations, or reasoning traces; a smaller student learns from those examples by fine-tuning.
- Distillation can transfer instruction style, domain behavior, safety policy, tool-use format, or reasoning patterns.
- In practice, companies use distillation to reduce serving cost, improve latency, and deploy specialist models on cheaper hardware.

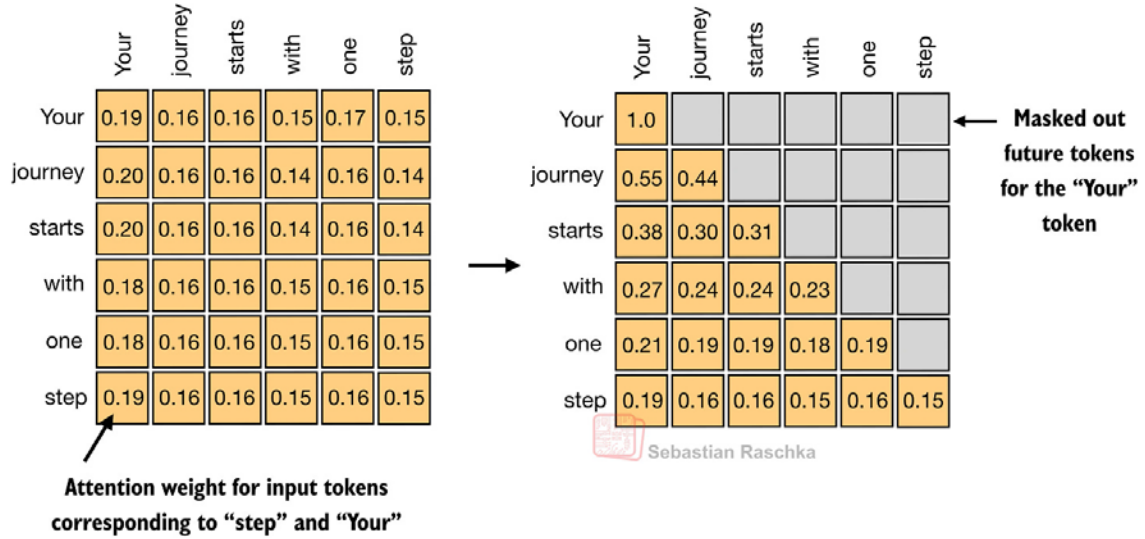
Section 4 — Efficient attention architectures

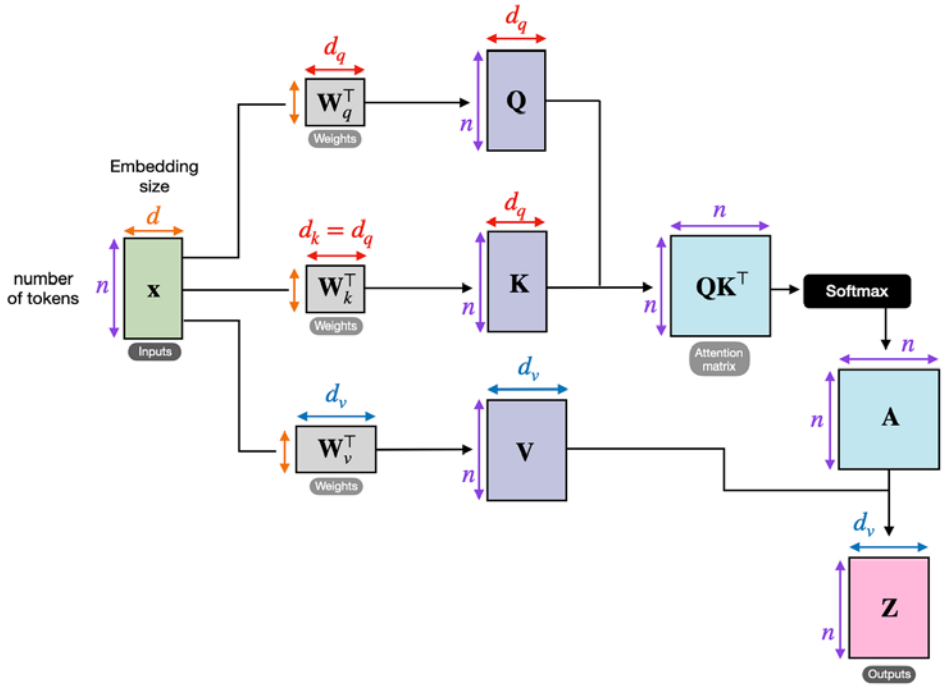
Architectural choices that reduce memory, bandwidth, and active computation

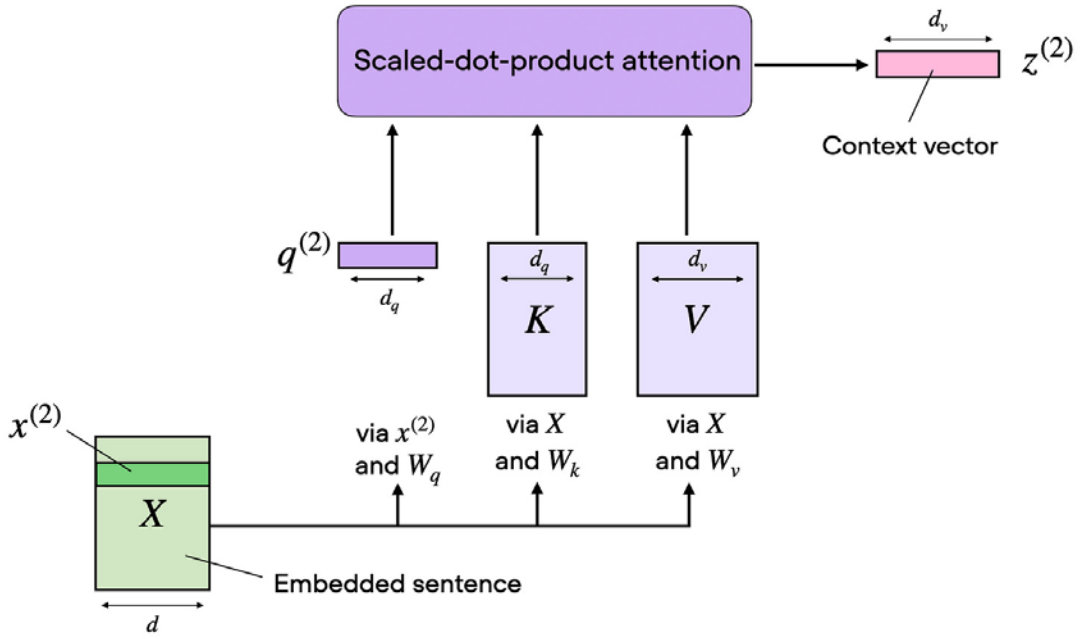
- Attention refresher: causal self-attention and Queries, Keys, and Values
- Input/output-aware kernels: FlashAttention
- Key-Value (KV) cache: the central inference data structure
- Multi-Head Attention (MHA): full per-head KV state
- Grouped-Query Attention (GQA): share KV heads
- Multi-Head Latent Attention (MLA): compress the cached state

Attention refresher



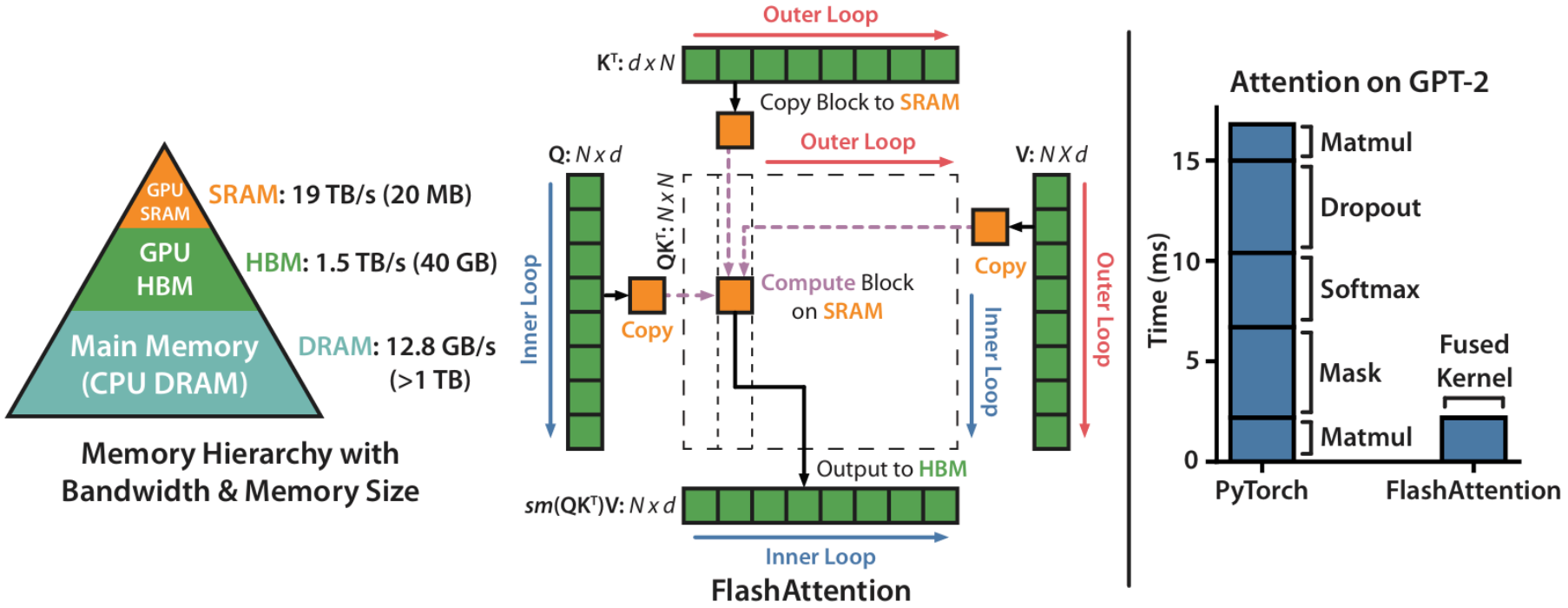






Input/output-aware kernels: FlashAttention

The bottleneck is often data movement, not just arithmetic.



Source: https://openreview.net/pdf?id=H4DqfPSibmx

- Standard attention materializes the N×N attention matrix in High Bandwidth Memory (HBM).
- FlashAttention tiles attention into SRAM and avoids writing large intermediates.
- In practice: FlashAttention-style kernels are standard in modern training and inference stacks.

Key-Value (KV) cache: the central inference data structure

Autoregressive generation saves past attention states instead of recomputing them

- The cache stores Key (K) and Value (V) tensors for previous tokens at every transformer layer.
- It trades compute for memory: faster decoding, but memory grows with batch size and context length.
- Many serving systems are effectively KV-cache managers with schedulers around them.

without caching	with caching
for each step, recompute all previous `K` and `V`	for each step, only compute current `K` and `V`
attention cost per step is **quadratic** with sequence length	attention cost per step is **linear** with sequence length (memory grows linearly, but compute/token remains low)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_{\text{head}}}} \times \text{mask}\right) V$$

$$\text{Attention}(q_t, [\underbrace{k_1, k_2, \ldots, k_{t-1}}, k_t], [\underbrace{v_1, v_2, \ldots, v_{t-1}}, v_t])$$

$$K_{\text{cache}} \leftarrow \text{concat}(K_{\text{past}}, k_t), \quad V_{\text{cache}} \leftarrow \text{concat}(V_{\text{past}}, v_t)$$

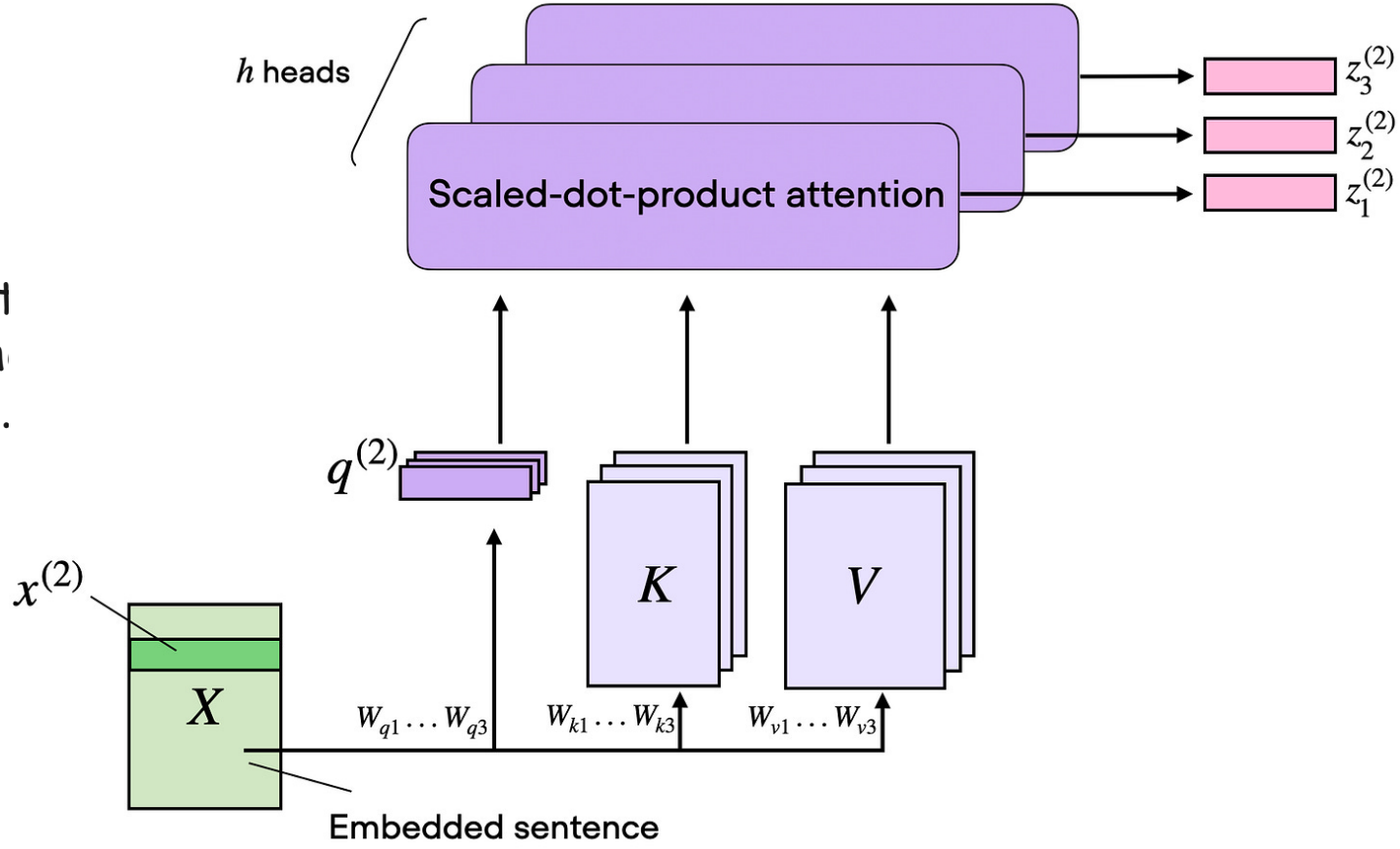
KV cache size ∝ Batch_size * Context_len * Layers * KV_heads * Head_dim * Bytes/Value

https://huggingface.co/docs/transformers/en/cache_explanation

Multi-Head Attention: full per-head KV state

The deployment question is how much state each token leaves behind

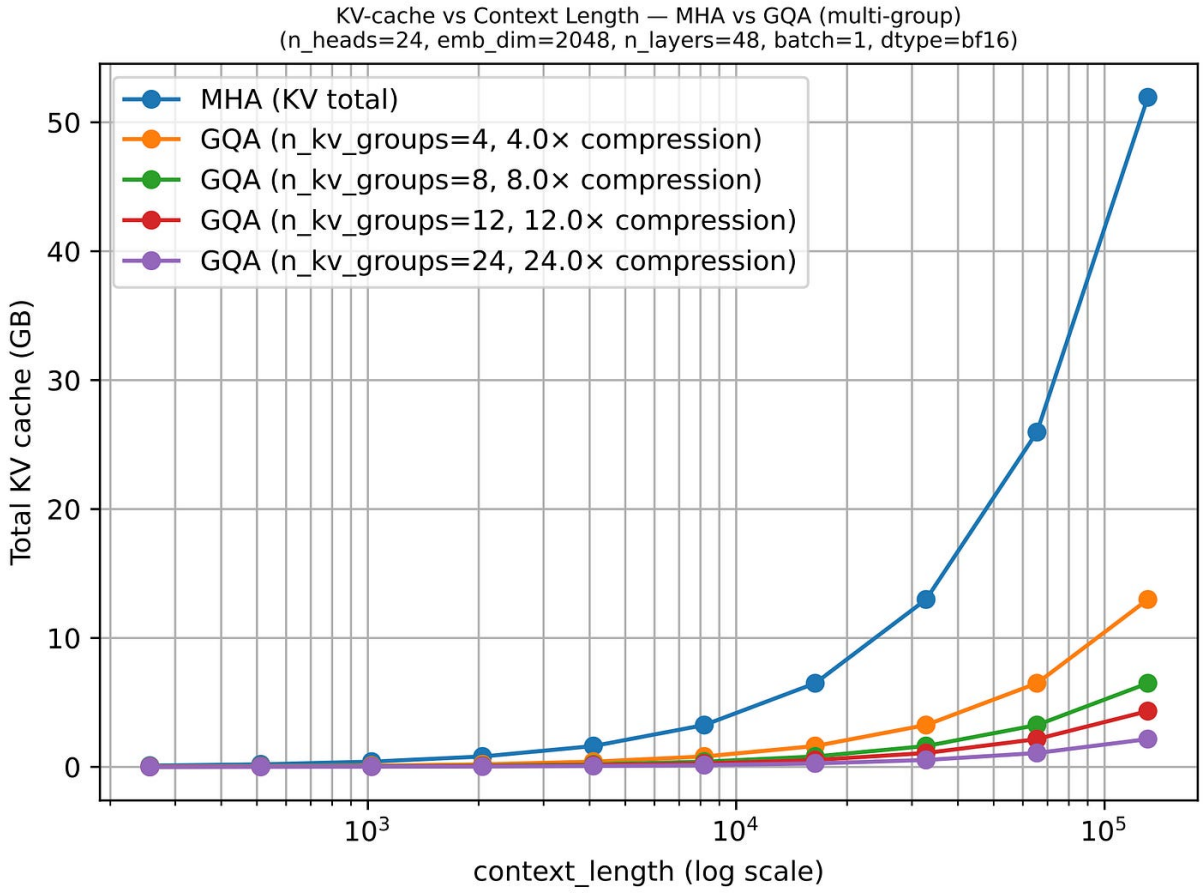
- Multi-Head Attention (MHA) stores separat
  Key and Value vectors for every attention h
  it is accurate but expensive at long context.

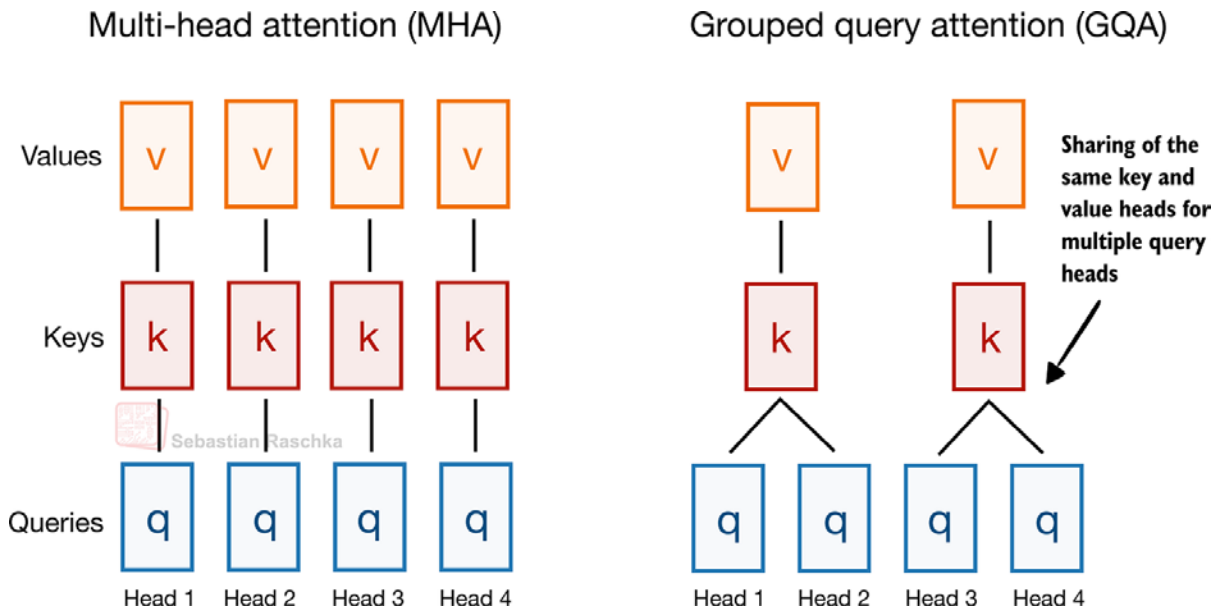


https://magazine.sebastianraschka.com/p/visual-attention-variants

Grouped Query Attention (GQA): share KV heads

The deployment question is how much state each token leaves behind



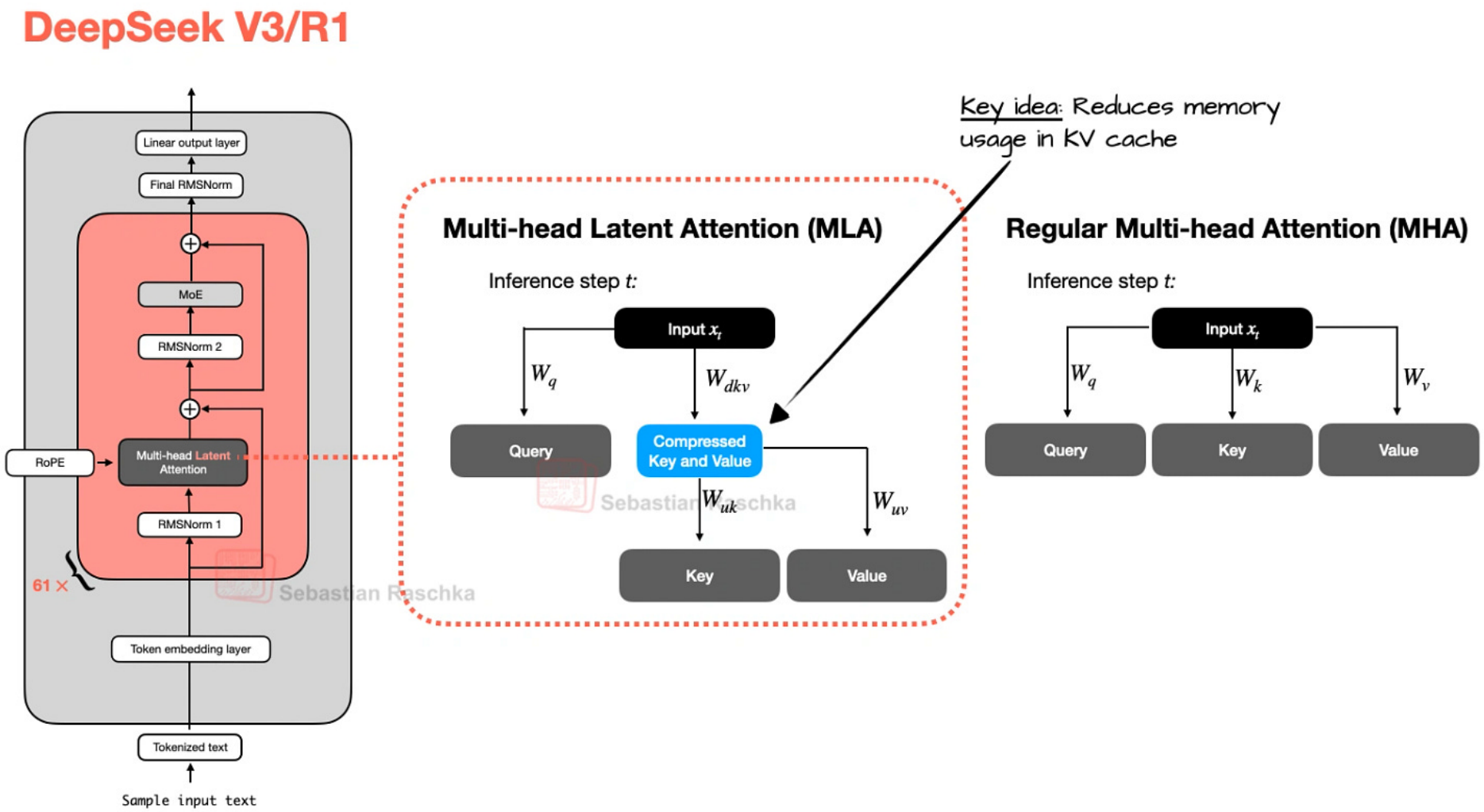


- Multi-Query Attention (MQA) and Grouped-Query Attention (GQA) reduce Key-Value (KV) cache traffic by sharing K/V heads across query heads.

https://magazine.sebastianraschka.com/p/visual-attention-variants

Multi-Head Latent Attention (MLA): compress cached state

The deployment question is how much state each token leaves behind



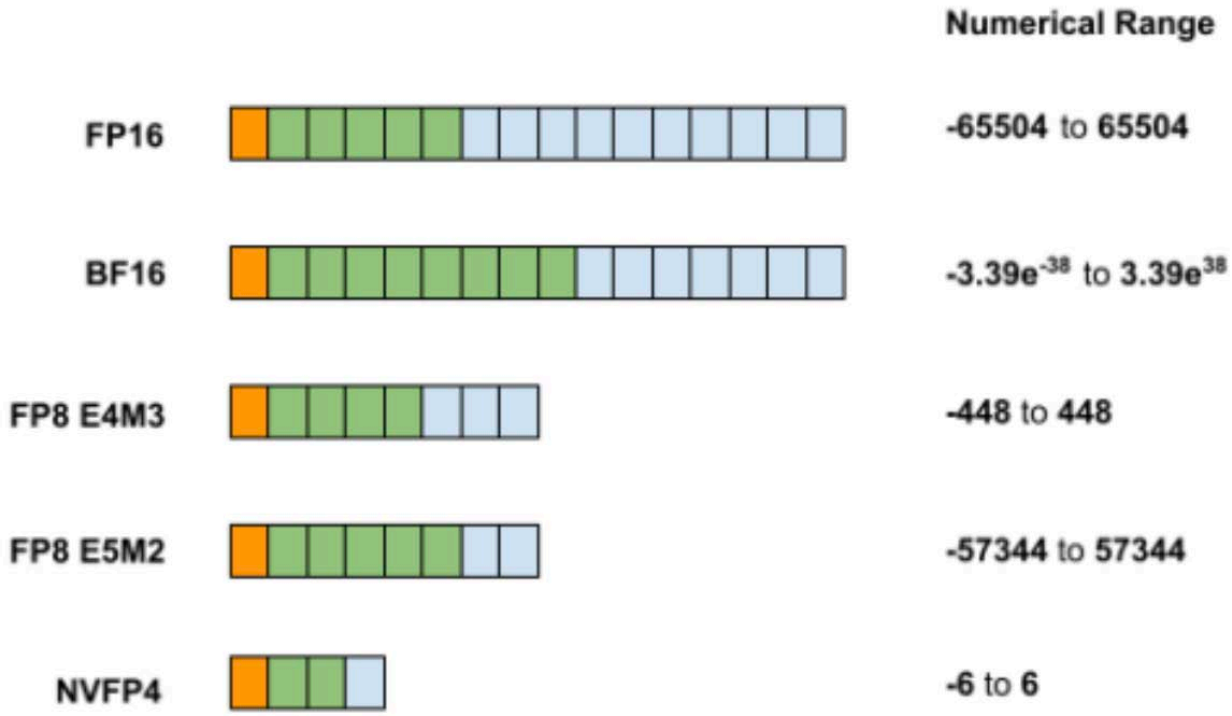
https://magazine.sebastianraschka.com/p/visual-attention-variants

Section 5 — Compression and Quantization for Deployment

- Quantization map: what is being quantized?
- Weight-only Post-Training Quantization (PTQ): GPTQ and AWQ
- Weight-activation quantization: SmoothQuant
- Key-Value (KV) cache quantization

Quantization map: what is being quantized?

Weights, activations, and the Key-Value (KV) cache affect different deployment bottlenecks.



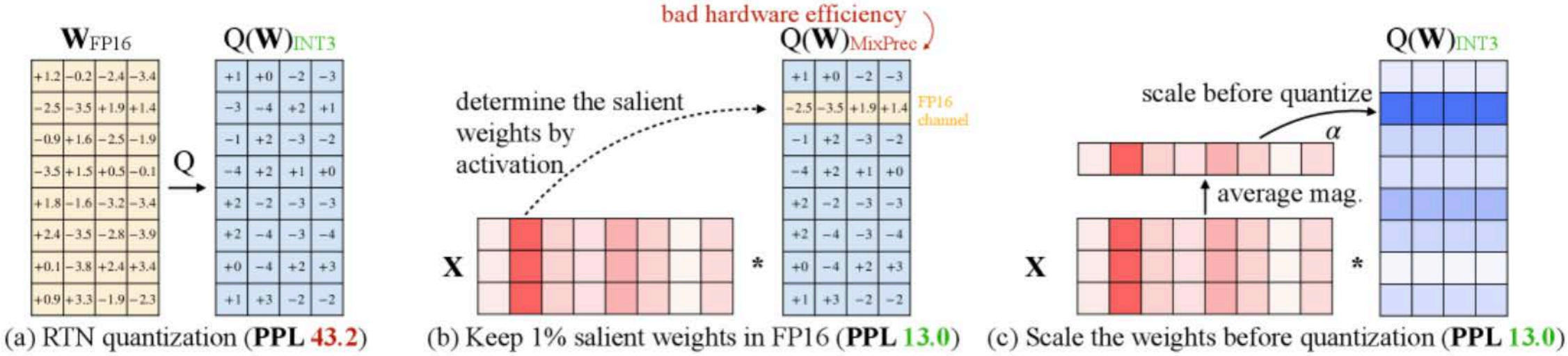
- Weight quantization mainly reduces model memory and memory bandwidth; it does not automatically shrink activations or the KV cache.
- Activation quantization can accelerate matrix multiplications when supported by kernels and hardware, but calibration and outliers matter.
- KV cache quantization targets long context and high concurrency; it is a separate lever from weight quantization.

https://developer.nvidia.com/blog/model-quantization-concepts-methods-and-why-it-matters/

Weight-only Post-Training Quantization (PTQ): GPTQ and AWQ

The common deployment move: store weights in 3–4 bits while keeping activations in higher precision.

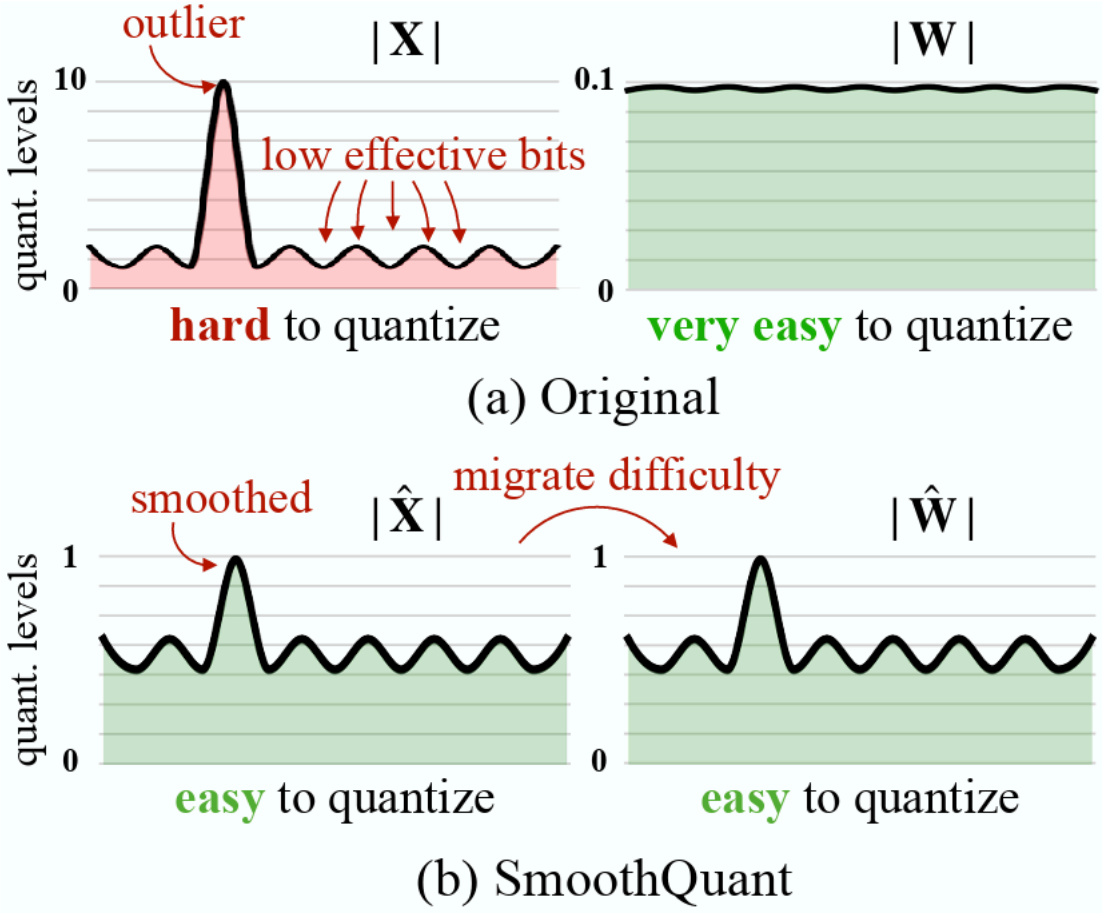
URL: https://arxiv.org/html/2306.00978v6



- GPTQ is a one-shot PTQ method for Generative Pre-trained Transformer weights; it uses approximate second-order information for error compensation.
- Activation-aware Weight Quantization (AWQ) uses activation statistics to protect salient weight channels before low-bit quantization.
- In practice: W4A16 models are served with optimized Graphics Processing Unit (GPU) kernels in runtimes such as vLLM and TensorRT-LLM.

Weight-activation quantization: SmoothQuant

Move quantization difficulty away from activation outliers so both weights and activations can use low precision.

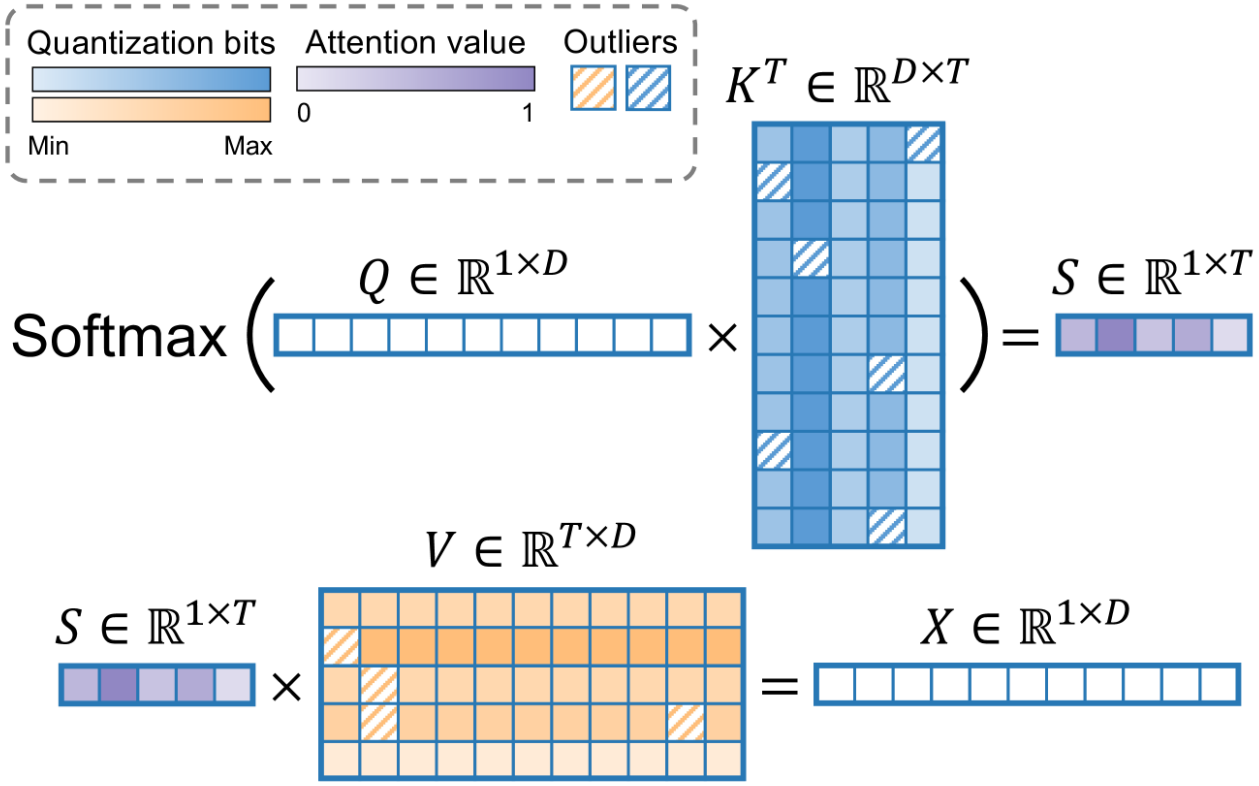


- Naïve W8A8 quantization can fail because Large Language Model activations often contain large outlier channels.
- SmoothQuant applies an offline, mathematically equivalent scaling: smooth activations, adjust weights, preserve the layer computation.
- In practice: used for INT8 W8A8-style inference when the target stack has efficient low-precision matrix multiplication kernels.

URL: https://arxiv.org/pdf/2211.10438

Key-Value (KV) cache quantization

For long-context and high-concurrency serving, the cache can become the dominant memory object.



URL: https://arxiv.org/abs/2403.04643

- During autoregressive decoding, KV cache memory grows with batch size, context length, number of layers, and KV heads.
- QAQ (Quality Adaptive Quantization): achieves up to 10x the compression ratio of the KV cache size with a neglectable impact on model performance quantizes keys per-channel and values per-token, based on their different outlier/error structure.
- In practice: Hugging Face Transformers exposes quantized KV cache; TensorRT-LLM supports FP8 and NVFP4 KV cache modes.

Section 6 — Serving systems and application efficiency

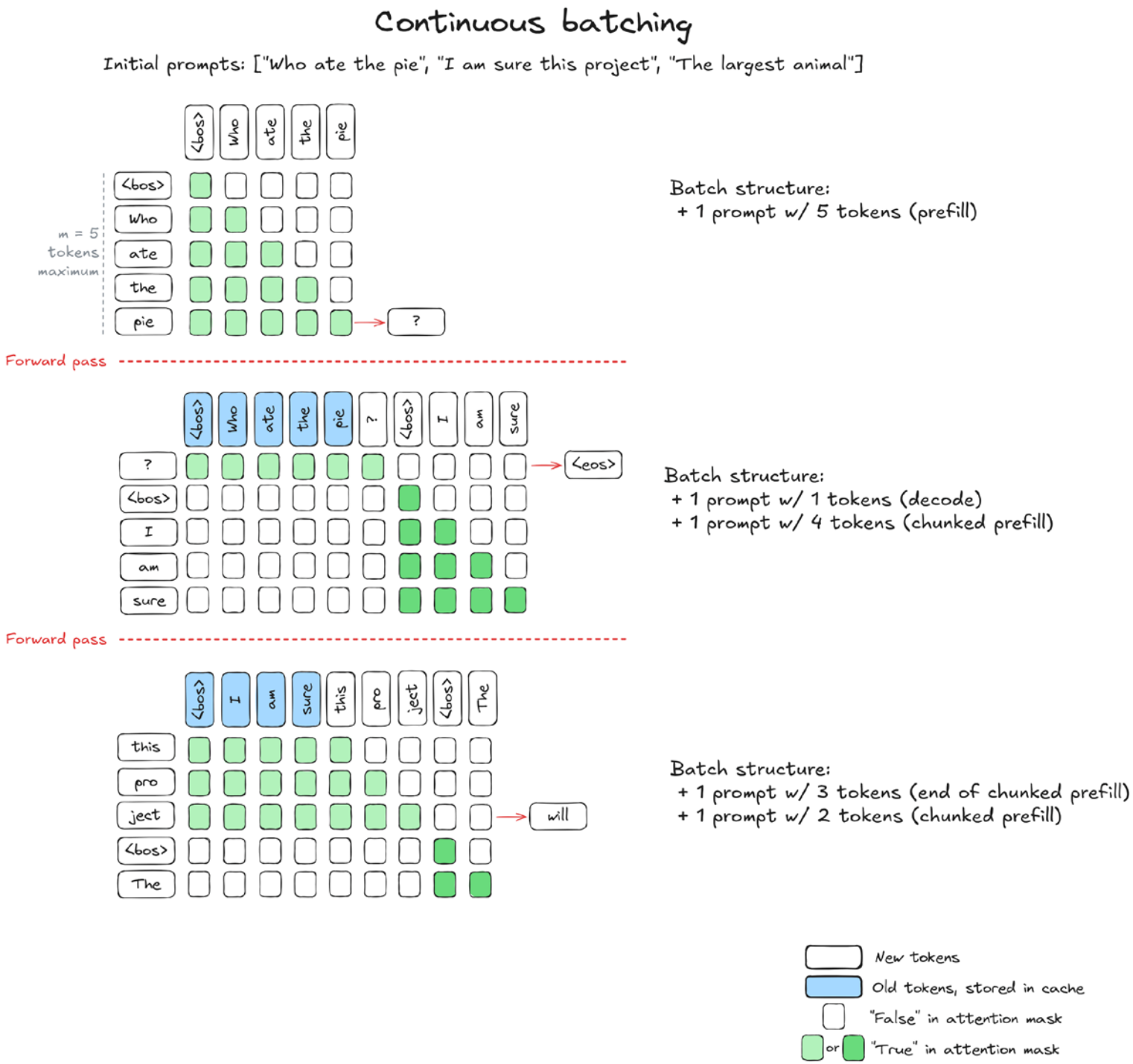
Outline

- Continuous batching
- Prefix, prompt, and context caching
- Speculative decoding and multi-token prediction
- Retrieval-Augmented Generation (RAG) and context selection
- Model routing and cascades

Continuous batching

Keep the accelerator full by admitting and removing requests at token boundaries

- Static batches waste work when requests have different input and output lengths.
- Continuous batching dynamically replaces completed requests with waiting ones.
- Production systems combine dynamic scheduling with ragged batching and chunked prefill.

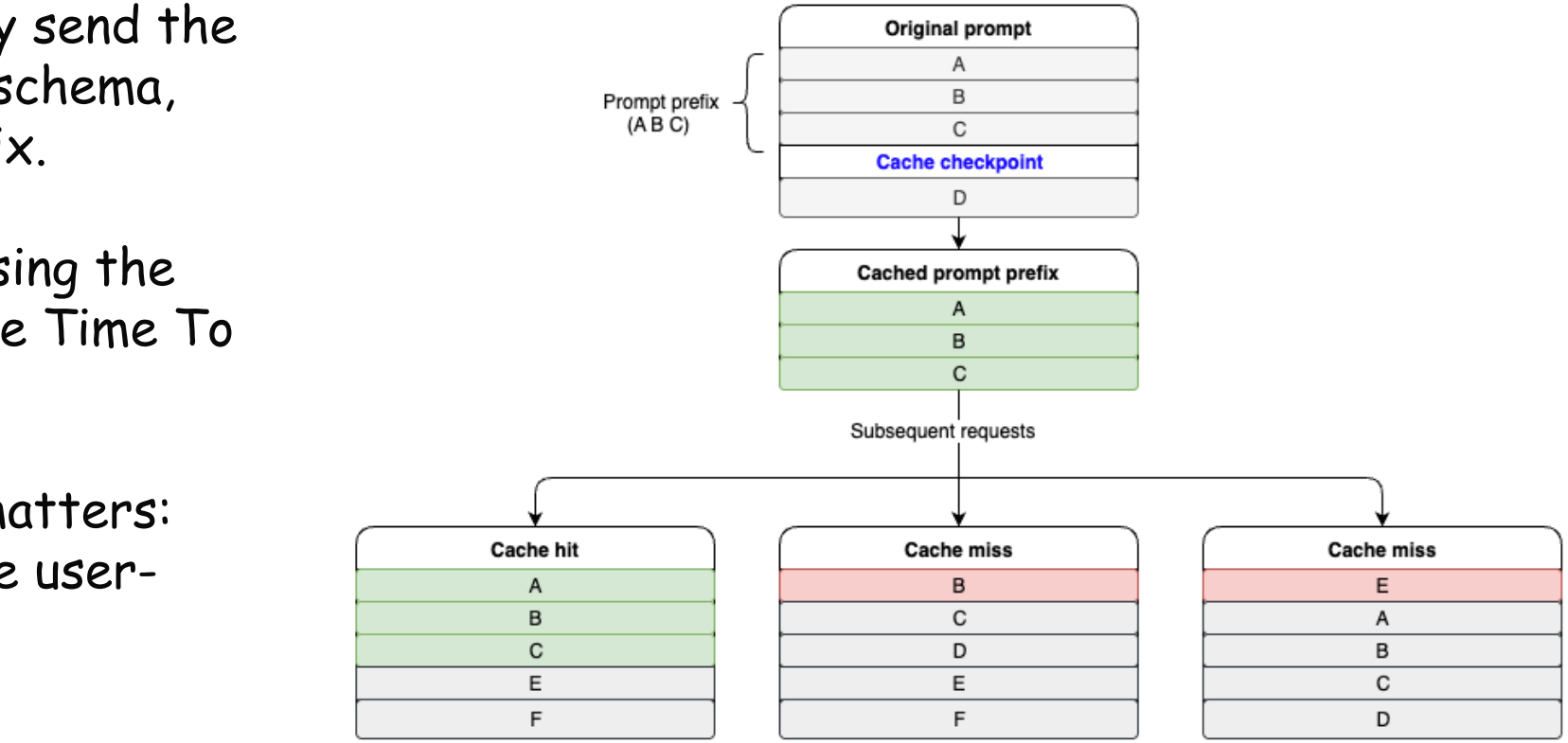


https://huggingface.co/blog/continuous_batching

Prefix, prompt, and context caching

Reuse repeated prompt prefixes to reduce prefill latency and input-token cost

- Many applications repeatedly send the same system prompt, tools, schema, examples, or document prefix.
- A cache hit avoids reprocessing the shared prefix and can reduce Time To First Token
- In practice, prompt layout matters: stable content first, variable user-specific content last.

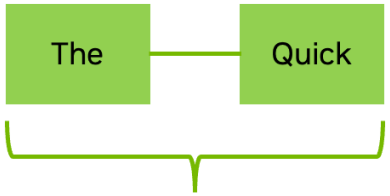


https://aws.amazon.com/blogs/machine-learning/effectively-use-prompt-caching-on-amazon-bedrock/

Speculative decoding and multi-token prediction

Generate candidate tokens cheaply, then verify several at once with the target model

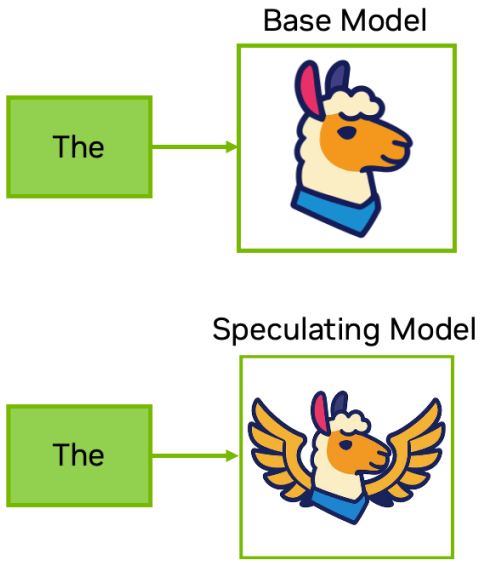
- Classic speculative decoding uses a small draft model and a larger target model.
- The target verifies a block of candidate tokens in parallel and accepts the longest valid prefix.
- Modern variants include Extrapolation Algorithm for Greater Language-Model Efficiency (EAGLE), self-speculation, n-gram speculation, and Multi-Token Prediction (MTP).



https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

Speculative decoding and multi-token prediction

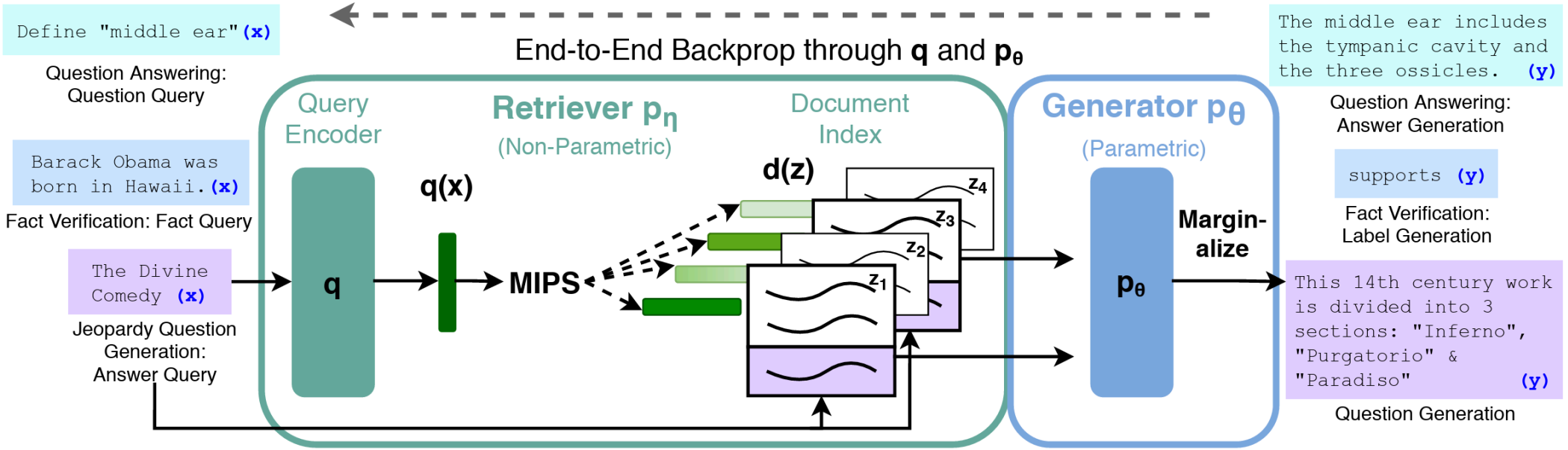
Generate candidate tokens cheaply, then verify several at once with the target model



https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

Retrieval-Augmented Generation (RAG) and context selection

Reduce expensive context by sending the model the right evidence, not all evidence

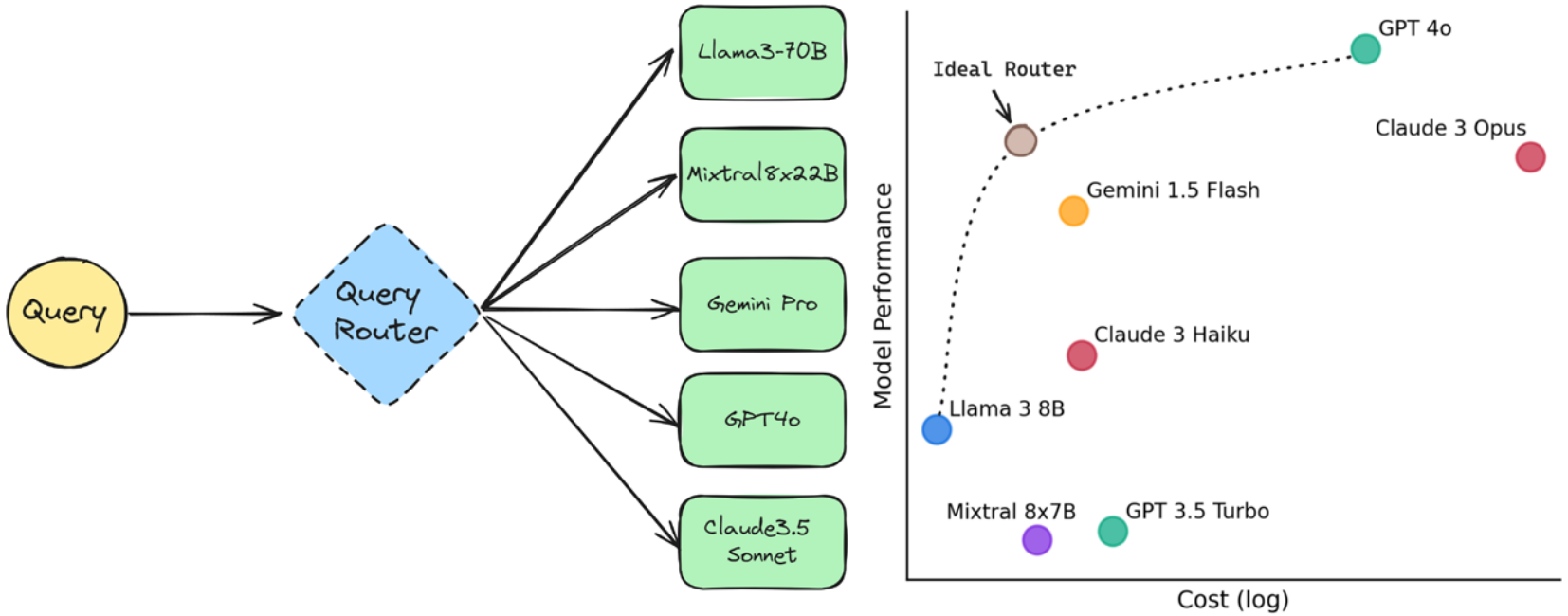


https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

- Retrieval-Augmented Generation (RAG) retrieves external documents and inserts selected passages into the prompt.
- As an efficiency tool, RAG reduces prefill tokens and Key-Value (KV) cache growth compared with sending everything.
- Production systems add chunking, reranking, filtering, summarization, and citation/grounding checks.

Model routing and cascades

Do not send every request to the largest model when a smaller model will do



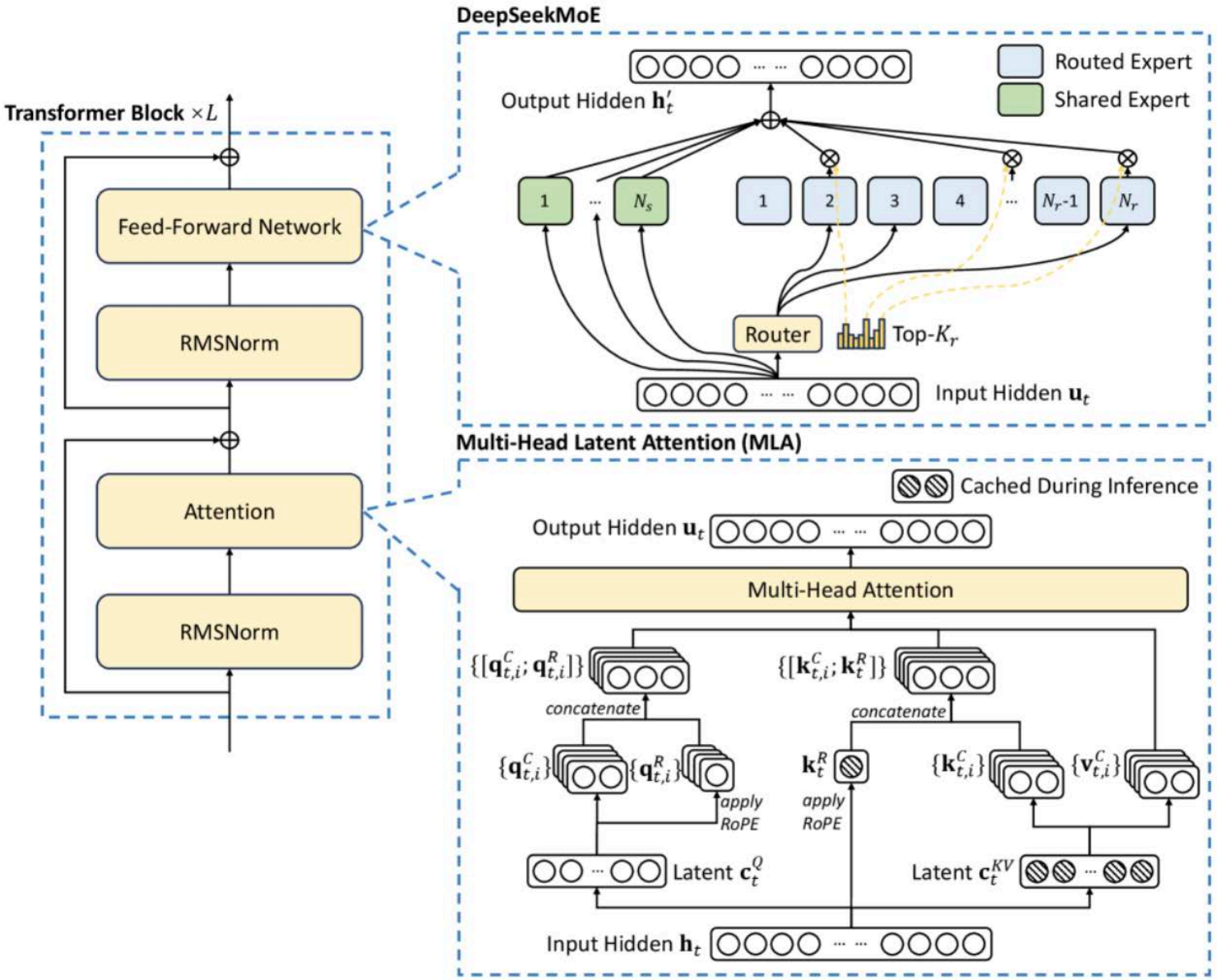
https://www.lmsys.org/blog/2024-07-01-routellm/

- A router predicts whether a request should go to a cheaper model or a stronger model.
- The main design variable is the cost-quality trade-off threshold, not only raw latency.
- In practice, routing can combine small models, specialized models, retrieval, tools, and fallback policies.

Closing remarks

- Case study: DeepSeek-V3 as end-to-end efficiency co-design
- Practitioner decision matrix: what to use when

Case study: DeepSeek-V3 as end-to-end efficiency co-design



- The key bottleneck is now token economics: training tokens, output tokens key-value (KV) cache, and utilization.
- Modern systems combine many levers: sparse activation, low precision, better data, memory-efficient attention, and serving-time scheduling.
- A single model architecture can encode deployment assumptions: active parameters, KV-cache size, communicatio pattern, and supported precision.

Source URL: https://arxiv.org/html/2412.19437v1

Practitioner decision matrix: what to use when

- Memory bottleneck:
  - quantization,
  - Key-Value (KV) cache compression,
  - Grouped-Query Attention,
  - smaller model, or
  - sharding.

- Latency or throughput bottleneck:
  - continuous batching,
  - PagedAttention,
  - speculative decoding,
  - prefix caching,
  - routing, or
  - kernel/runtime tuning.

- Quality or training-cost bottleneck:
  - data curation,
  - post-training,
  - LoRA or QLoRA,
  - distillation, or
  - RAG

Constraint	Primary levers	Typical deployment
memory	quantize, compress, cache	larger batch, longer context
latency	cache, speculate, route	lower TTFT, steady tokens
training cost	better data, precision, sharding	better model per GPU-hour
quality/cost	RAG, distill, LoRA	right model per request

Thanks for your attention

- You can reach me at c.dovrolis@cyi.ac.cy

- Please complete the following questionnaire: