FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

NI-NLM – Lecture 8

Efficiency: quantization, distillation, low-rank adaptation, MoE.

Zdeněk Kasner

14 Apr 2026

Why efficiency matters

The cost of running LLMs

Modern LLMs are expensive to **train**, **store**, and **run**:

- **Training**: GPT-4 used an estimated $78 million worth of compute to train.
- **Memory**: a 70B model in `float32` (=4 bits per parameter) requires $\approx$ 280 GB of GPU memory / disk space.
- **Inference**: OpenAI spent $8.67 billion on inference in the first nine months of 2025 (nearly double their revenue for the same period).

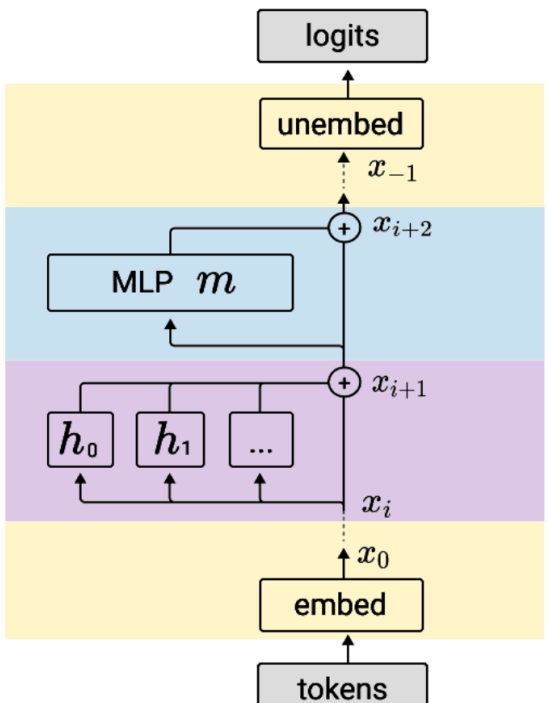
> **Question**
>
> Can you think of ways how to make LLMs smaller, faster, and cheaper (without sacrificing too much performance?)

Transformer: algorithmic complexity

Variables: hidden state dimension (D), sequence length (N), vocabulary size (V).

	Time	Space
final projection (unembedding)	$N \cdot V \cdot D$ each state projected to logits over the vocabulary	$V \cdot D + V \cdot N$ (un)embedding matrix 📦 output logits 🧮
MLP	$N \cdot D^2$ ↘ projection matrix 📦	$D^2 + N \cdot D$ activations 🧮
multi-head attention	$N \cdot D^2 + N^2 \cdot D$ queries, keys, values ↘ attention matrix 📦	$N^2 + N \cdot D + D^2$ projected values 🧮 Q, K, V matrices 📦
embedding	$N \cdot D$ each token is embedded	$V \cdot D + N \cdot D$ embedding matrix 📦 embeddings 🧮

📦 = model parameters, 🧮 = computed values



source: Transformer circuits

LLM bottlenecks: computational throughput

Time complexity

MLP & projections *attention matrix* *unembedding*

$$N \cdot D^2 + \boldsymbol{N^2} \cdot \boldsymbol{D} + N \cdot V \cdot D$$

long context → $N^2 \gg D^2$

happens only once

Computing **multi-head attention** becomes the main bottleneck when $N^2 \gg D^2$ (long context).

Solution: efficient attention, MoE, distillation.

Typical hyperparameter values:

Param	Min	Max
N	512	128,000+
D	768	4,096+
V	30,000	100,000+
blocks	8	128+

LLM bottlenecks: disk space

Space complexity: model parameters 📦

embedding matrix $V \cdot D$ + *MLP & projections* D^2

stored only once

MLPs typically take up the majority of model parameters – they are repeated in each layer and the multiplicative factor is larger than for attention.

Solution: parameter quantization, (Q)LoRA, distillation.

Typical hyperparameter values:

Param	Min	Max
N	512	128,000+
D	768	4,096+
V	30,000	100,000+
blocks	8	128+

LLM bottlenecks: memory

Space complexity: activations 🧮

$$\underbrace{N^2}_{\text{attention matrix}} + \underbrace{N \cdot D}_{\text{hidden states}} + \underbrace{N \cdot V}_{\text{logits}}$$

gets more important
with growing input context size

Activations take extra space in GPU memory (alongside the model parameters) when the model is performing a computation.

Solution: parameter quantization, (Q)LoRA, distillation

Typical hyperparameter values:

Param	Min	Max
N	512	128,000+
D	768	4,096+
V	30,000	100,000+
blocks	8	128+

Building an intuition about sizes

Question

How much memory does it approximately take to store a **7B model** (=7 billion parameters) in **fp16 precision**, meaning each parameter is stored as a 16-bit floating point number?

Each parameter takes 2 bytes (16 bits), so:

7 billion parameters · 2 bytes per parameter = 14B bytes = **14 GB**.

LLM bottlenecks: training vs. inference

In terms of resource requirements: **pre-training** $\gg$ **finetuning** $\gg$ **inference**.

However, unlike training, we typically want to run inference on consumer hardware.

Dataset size

- Pre-training: 10T tokens $\approx$ 50 TB.
- Finetuning: up to 100s GB.

Memory requirements

- Inference: 2 bytes per parameter → 14 GB for a 7B model.
- Training / finetuning: 2 bytes per parameter + 2 bytes per gradient + 12 bytes per optimizer state (Adam) → **112 GB** for a 7B model.

Efficient algorithms: overview

Techniques we will cover:

Algorithm	What it targets	Train. memory	Train. speed	Inf. memory	Inf. speed
Quantization	Reducing param. size	–	–	✓✓	✓
Distillation	Reducing # of params	–	–	✓✓	✓✓
(Q)LoRA	Efficient finetuning	✓✓	✓	–	–
MoE	Reducing active params	–	✓✓	–	✓
Linear attention	Faster attention using math tricks	✓	✓	✓	✓
FlashAttention	Faster attention using HW optimizations	✓✓	✓	✓	✓✓

Quantization

Bits per parameter

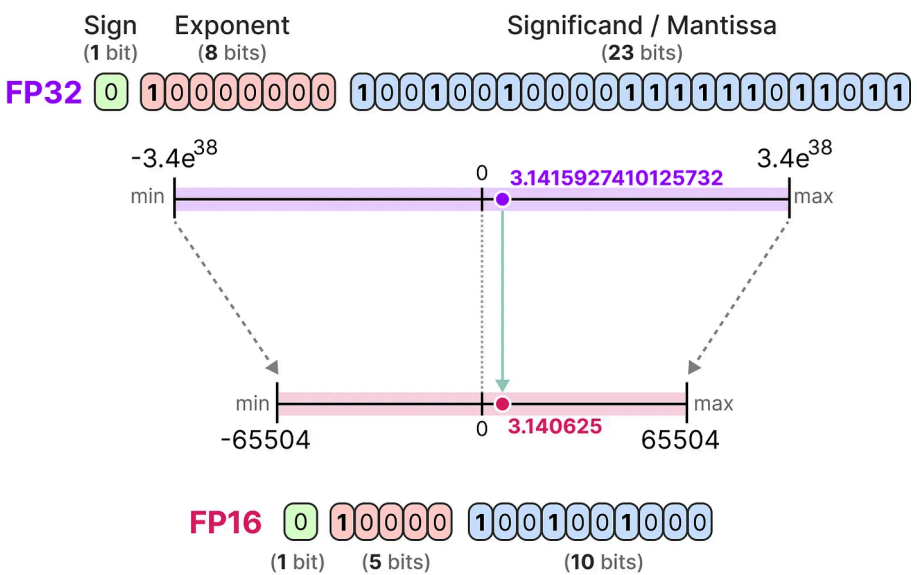
source: Maarten Grootendorst's blog

Model parameters are floating point numbers. How do we store them? And can we store them more efficiently?

Floating point numbers

- `float64` – native Python (→64 bits).
- `float32` – baseline precision (→32 bits).
- `float16` – half precision (→16 bits).

- Inference: **little performance degradation** with `float16` → used with LLMs.
- Training: `float16` can lead to vanishing gradients → **mixed-precision training**.

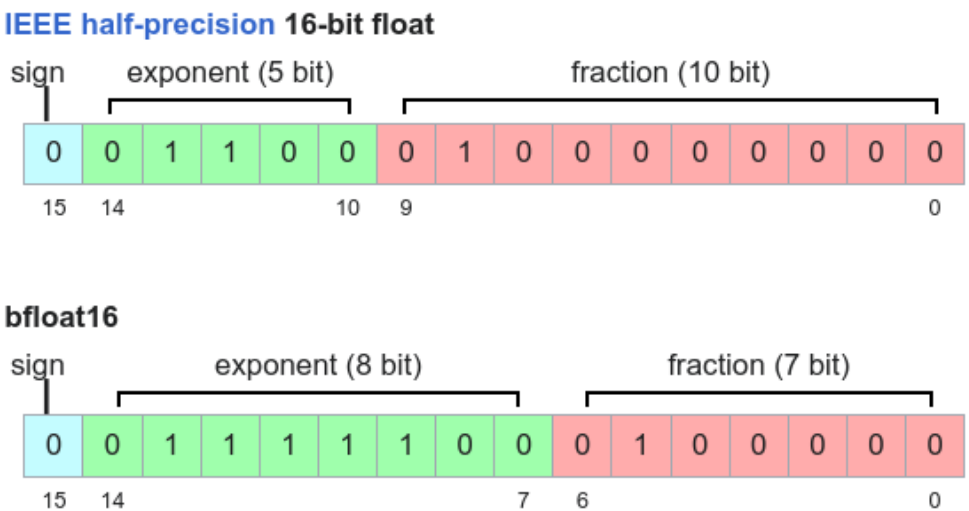


bfloat16

source: Wikipedia

`bfloat16`: Developed by Google Brain (→ “brain float”).

- A 16-bit float **optimized for ML models**.
- Greater **dynamic range** than float16 (supports outlier weights) at the cost of lower precision.
- Now the **standard format** for training and inference in most LLM frameworks.



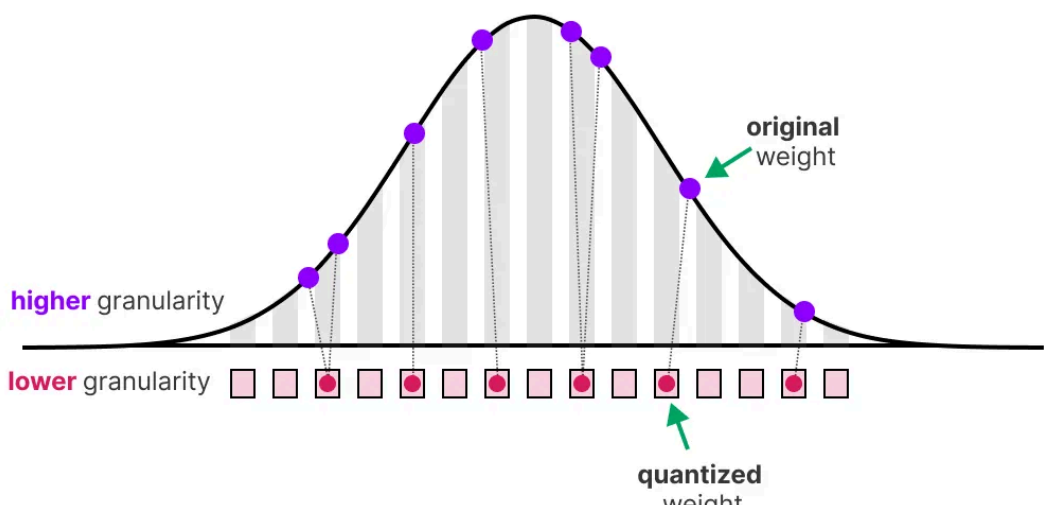
Parameter quantization

source: Maarten Grootendorst's blog

Can we go beyond a 16-bit float?

Idea

Let's use `int8` having **8-bit per parameter** → we'll save twice as much space.



Parameter quantization

source: Maarten Grootendorst's blog

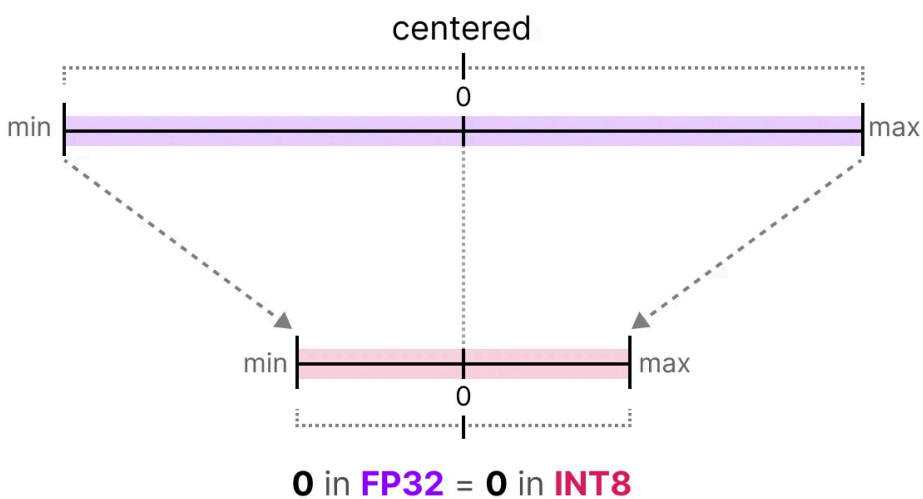
Question

The range of `int8` is $(-127, 127)$, while floats have a huge dynamic range ($\approx 10^{-38}$ to 10^{38}). How do we squeeze floating-point weights into the int range?

We can find the **maximum absolute value** and scale all weights proportionally:

$$x_{\text{quant}} = \text{round}\left(\frac{127}{\max(|x|)} \cdot x\right)$$

We only use the range of our actual model weights → the range is not that wide.

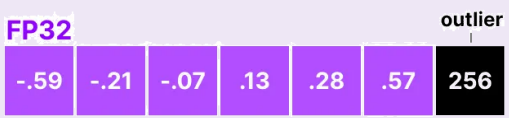


Parameter quantization

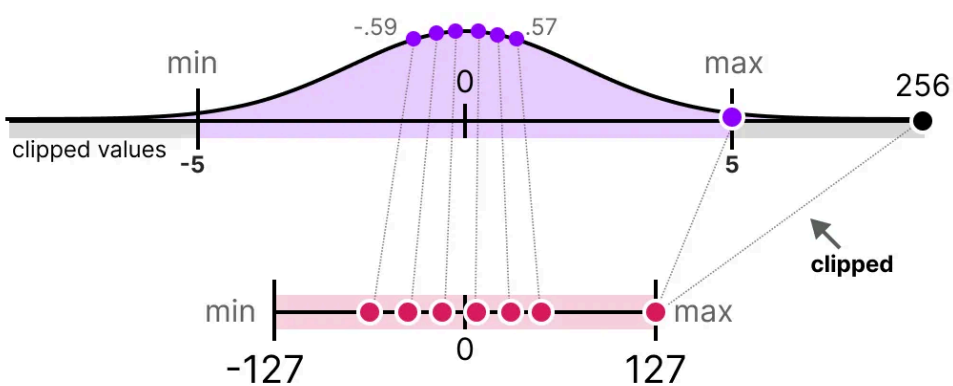
source: Maarten Grootendorst's blog

Question

What if some of the weights are **outliers**? This would map most of the weights to zero (or near zero):



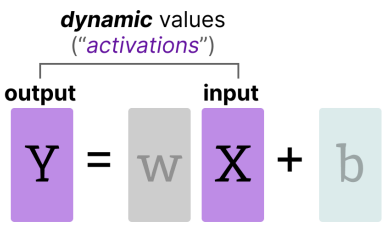
Simple solution: **clip the outliers**:



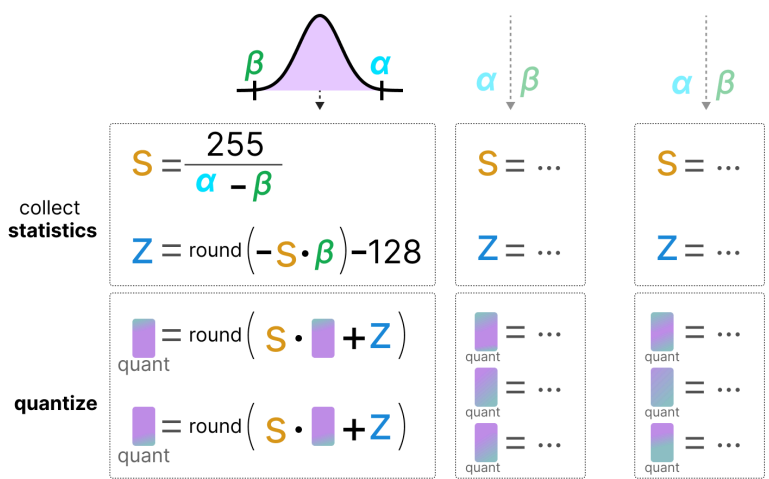
Parameter quantization

source: Maarten Grootendorst's blog

Model parameters can be quantized offline, but activations change with every input.



Solution: compute the quantization range **dynamically** at inference time:



Extreme quantization: 1-bit models

source: Ma et al. (2024)

Can we go all the way down to **1-bit**?

BitNet: Scaling 1-bit Transformers for Large Language Models

Hongyu Wang*†‡ Shuming Ma*† Li Dong† Shaohan Huang†
Huaijie Wang§ Lingxiao Ma† Fan Yang† Ruiping Wang‡ Yi Wu§ Furu Wei†◇
† Microsoft Research ‡ University of Chinese Academy of Sciences § Tsinghua University
https://aka.ms/GeneralAI

BitNet (Ma et al., 2024): Each weight is either −1 or +1.

Dot product reduced to addition → massive speedup on specialized hardware.

Post-training quantization in practice

GPTQ (Frantar et al., 2022)

- Algorithm for layer-wise quantization.
- Optimized for GPU inference.
- More robust for extreme quantization (→ down to 2-bit).
- Used with HF transformers.

GGUF

- File format based on block-wise quantization.
- Optimized for CPU inference.
- Originally from llama.cpp, used by Ollama.

> **Note**
>
> Models in Ollama are quantized by default to 4-bit.

Quantization: summary

Format	Bits	7B model size	Notes
`float32`	32	≈ 28 GB	Rarely used with LLMs
`float16`	16	≈ 14 GB	Inference with older GPUs (before ≈2020, like V100s)
`bfloat16`	16	≈ 14 GB	**Standard for training & inference**
`int8`	8	≈ 7 GB	Good accuracy, 2× savings
`int4`	4	≈ 3.5 GB	Popular for local deployment
`1-bit`	1	≈ 0.9 GB	Experimental, requires special training

Rule of thumb

Going from `float16` to `int4` reduces model size by ≈ 4× with negligible performance degradation for larger models.

Knowledge distillation

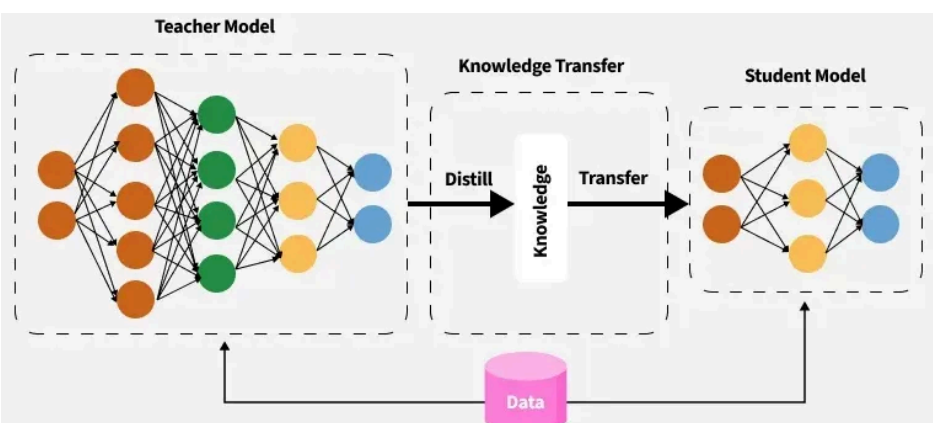
What is knowledge distillation?

source: https://www.geeksforgeeks.org/nlp/what-is-llm-distillation/

We have a **large model** and we want to make it more **efficient**.

Idea: knowledge distillation

We can train a smaller model (**"student"**) on the knowledge of the large model (**"teacher"**).

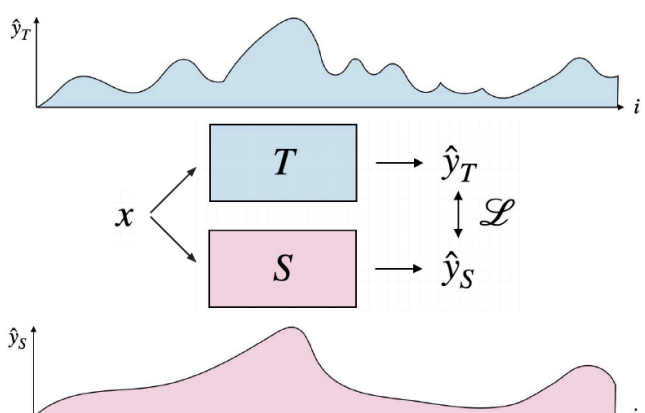


What is knowledge distillation?

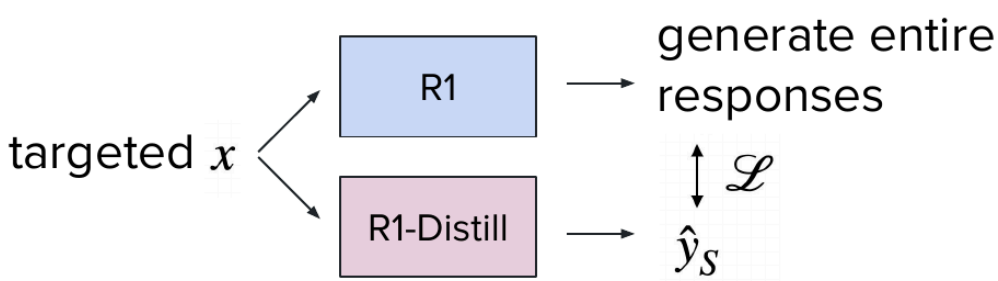
source: CME295

> ⚠️ **Warning**
>
> "Knowledge distillation" has two distinct meanings in the context of LLMs.

1. Learning from teacher's distribution:



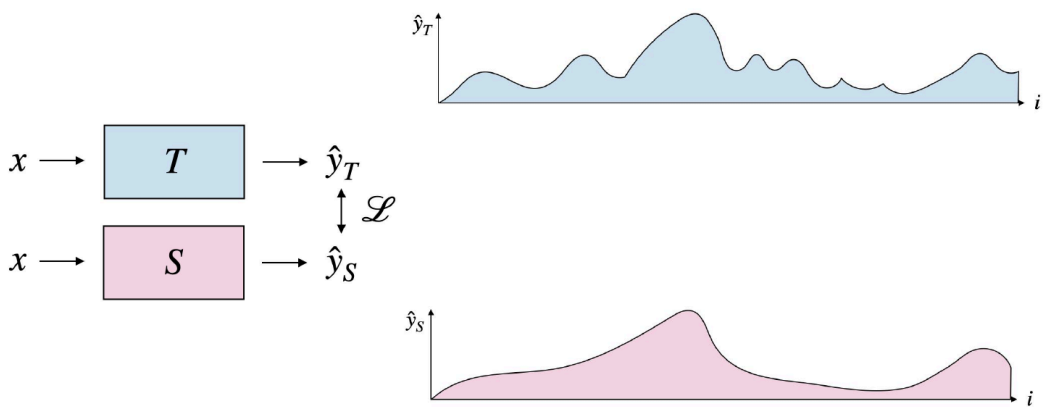
2. Learning from teacher's outputs:



Learning from teacher's distribution

source: CME295

Learning from teacher's distribution: The "original" knowledge distillation introduced by Hinton et al. (2015) for classification models.



The student learns from the teacher's output probability distribution (**soft outputs**) that carry **richer information than hard labels**.

Learning from teacher's distribution

The student is trained using a **KL-divergence loss** between their distribution $\hat{y}_S$ and the teacher's distribution $\hat{y}_T$:

$$\mathrm{KL}(\hat{y}_T \parallel \hat{y}_S) = \sum_i \hat{y}_T^{(i)} \log \left(\frac{\hat{y}_T^{(i)}}{\hat{y}_S^{(i)}} \right)$$

Example

- Ground truth (**hard label**): [cat: 1.0, dog: 0.0, car: 0.0]
- Teacher's distribution (**soft label**): [cat: 0.7, dog: 0.2, car: 0.1]

The soft label carries more signal. It tells the student that "a cat is somewhat similar to a dog, but not to a car".

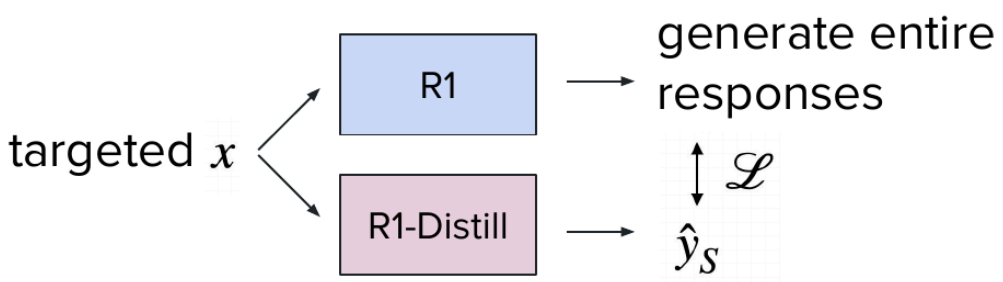
Learning from teacher's outputs

source: CME295

Sometimes we only have access to the teacher's generated text.

In that case, we still can:

- **Generate a large dataset** using the teacher model.
- **Finetune the student** on this synthetic dataset.



Use in practice

This technique was used e.g. to make smaller DeepSeek-R1 models. Also used for training smaller open models on the outputs from larger commercial models.

Finetuning & low-rank adaptation

Finetuning and when it helps

Question

In which cases we may want to finetune a model on our own?

→ To get outputs in a **consistent format** without extensive prompting.

→ To make the model learn our own **domain or a private knowledge base**.

→ To get a **smaller, more efficient** model (→ knowledge distillation).

⚠ **But finetuning is computationally expensive**

We saw that finetuning a 7B model requires ≈ **112 GB** of GPU memory. For reference, a single H100 (96GB GPU) costs around $25,000.

Low-rank adaptation (LoRA)

source: Hu et al. (2021)

The issue is alleviated by **LoRA** (Hu et al., 2021) : low-rank adaptation method.

Key ideas of LoRA

- Instead of updating model weights W directly, we keep the ΔW (the "diff") in a **separate matrix**.
- We ensure that the ΔW matrix has a **low rank** → it can be decomposed into two smaller matrices.

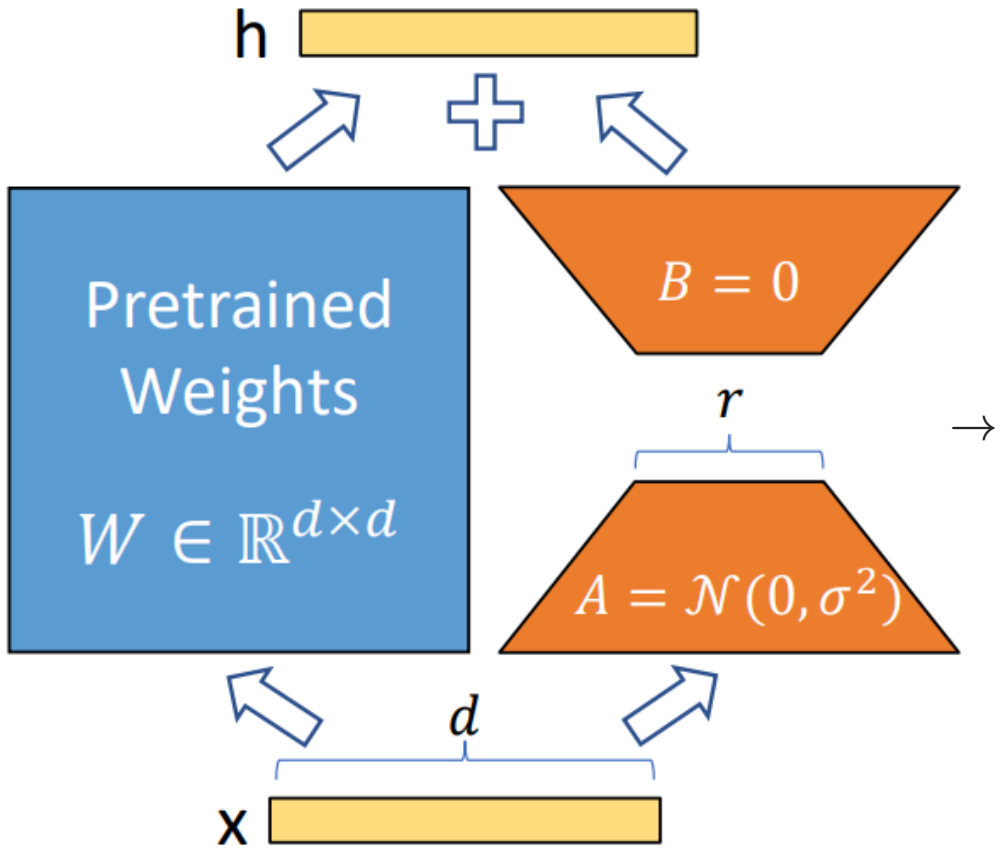
$$W' = W + \Delta W = W + BA$$

where $W \in \mathbb{R}^{d \times k}$, $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and $r \ll \min(d, k)$.

Low-rank adaptation (LoRA)

original LLM (very large)

←



→

LoRA weights: (small, can be stored as a separate "adapter")

Low-rank adaptation (LoRA)

source: Hu et al. (2021)

- The original model weights W are **frozen** (no gradient updates), only the small matrices A and B are trained.
- For inference, LoRA weights can be **merged** into the base model:

$$W' = W + BA$$

  → no additional latency at inference time.
- The LoRA weights can be stored as a separate **adapter** (typically a few MB).

> **Info**
>
> For a rank $r = 8$ and $d = 4096$: LoRA adds only $2 \times 4096 \times 8 = 65,536$ parameters per layer instead of $4096^2 \approx 16.7M$.

QLoRA: finetuning quantized models

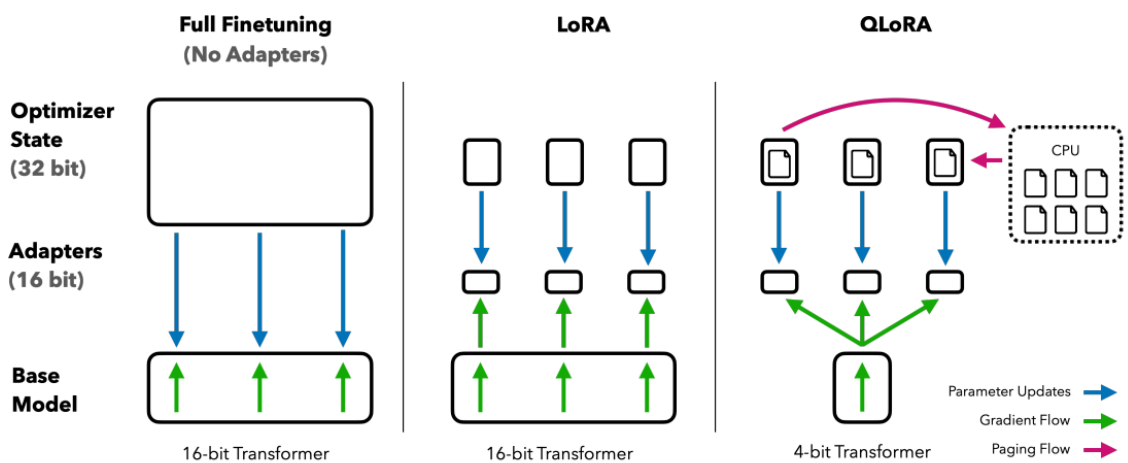
source: Dettmers et al. (2023)

Idea

Can we combine **quantization** and **LoRA** for even higher efficiency?

QLoRA (Dettmers et al., 2023):

- Careful parameter quantization + optimizing CPU-GPU memory transfers.
- Enables finetuning a 65B model on a single 48 GB GPU.



LoRA and QLoRA in practice

All things being equal: **full finetuning > LoRA > QLoRA** in terms of model performance.

→ The goal is to make the most out of the memory you have.

Method	Bits	Memory (7B model)
Full finetuning	32	≈ 120 GB
Full finetuning	16	≈ 60 GB
LoRA	16	≈ 16 GB
QLoRA	8	≈ 10 GB
QLoRA	4	≈ 6 GB

source: LLaMA-Factory

Frameworks: LLaMA-Factory, HuggingFace PEFT, Unsloth.

Mixture of experts

Mixture of experts (MoE)

We need to re-run the full Transformer stack for every decoded token

→ Inference with large models can get very **slow**.

> **Info**
>
> For Llama 3.1 405B, Oracle got around 27 tokens/sec on a specialized HW.

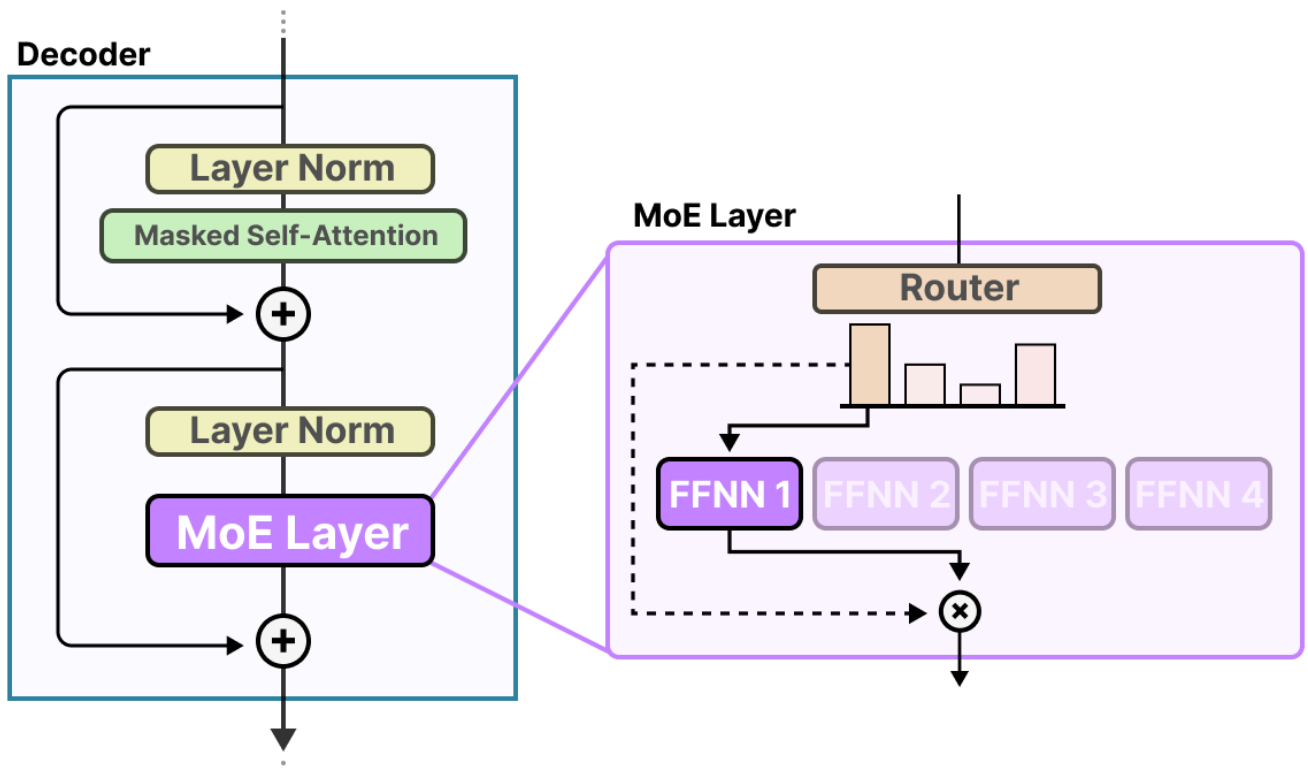
But perhaps the model does not need *all* of its parameters for *each* token?

> **Idea**
>
> Let's (1) force the model to **specialize** subsets of its parameters for different tasks and (2) only **activate** the specific subset → Mixture of experts.

Mixture of experts (MoE)

source: Maarten Grootendorst's blog

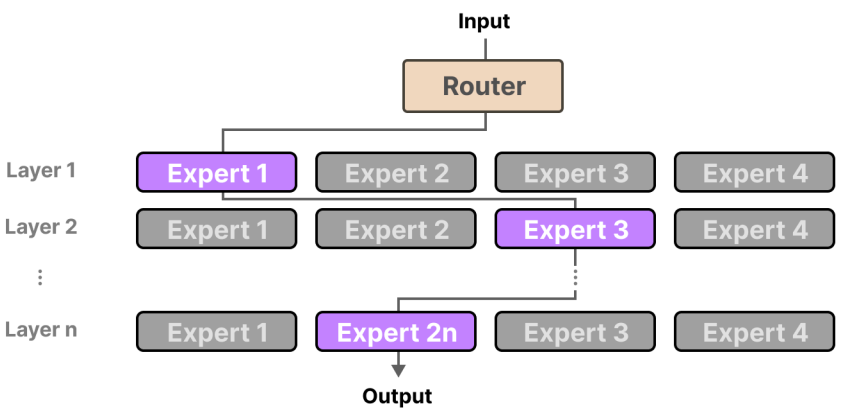
“Experts” = multiple feed-forward networks in each MLP layer of the Transformer.



Mixture of experts (MoE)

source: Maarten Grootendorst's blog

We can choose a different expert at each **layer**...



and a different set of experts for each **token**:

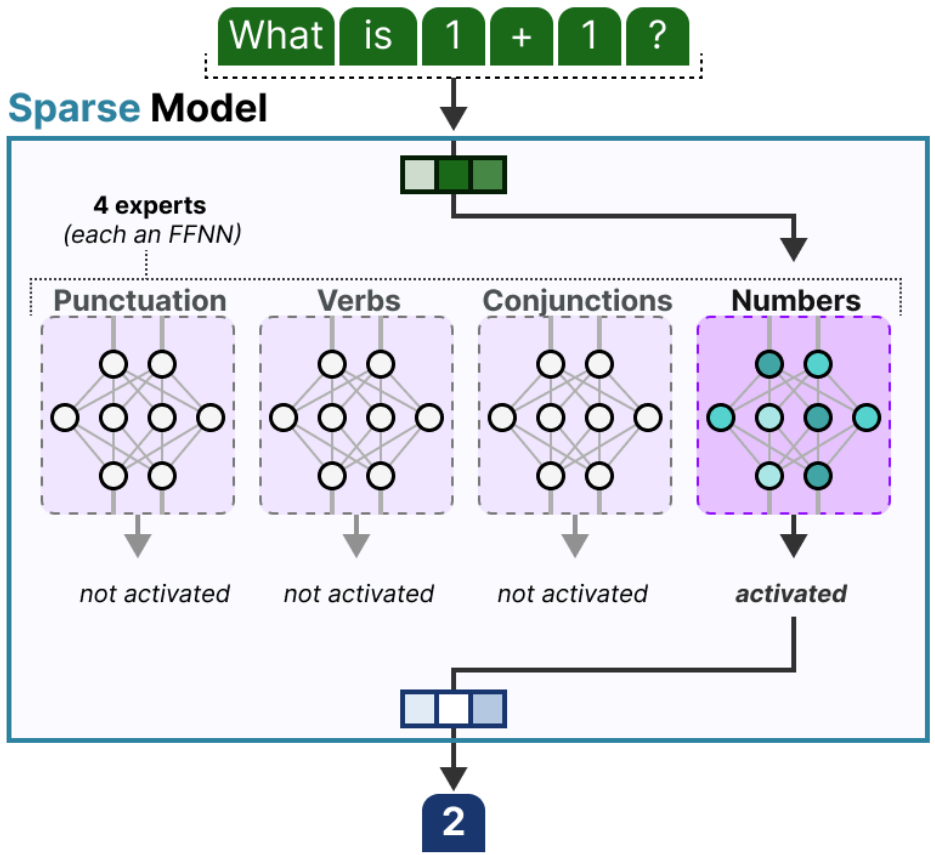
```
class MoeLayer(nn.Module):
    def __init__(self, experts: List[nn.Module],
        super().__init__()
        assert len(experts) > 0
        self.experts = nn.ModuleList(experts)
        self.gate = gate
        self.args = moe_args
```

Mixture of experts (MoE)

source: Maarten Grootendorst's blog

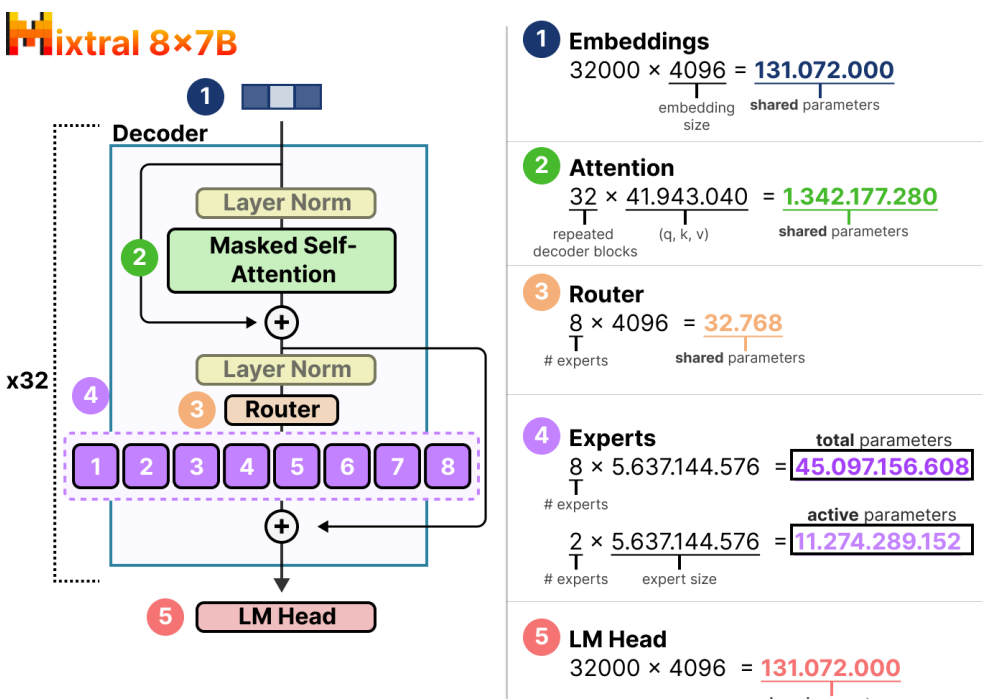
Why is this a good idea?

- Individual experts can **specialize** to solving certain kinds of problems.
- **Faster training** for the same number of total parameters (we only backpropagate through selected experts).
- **Faster inference** (although we still need to load the full model into memory).



MoE: Mixtral

Example: Mixtral 8×7B → 8 expert MLPs, approximately equivalent to a 47B dense model (not 56B, since attention layers and embeddings are shared).



Shared parameters

1 Embeddings	131.072.000	1 Embeddings	131.072.000
2 Attention	1.342.177.280	2 Attention	1.342.177.280
3 Router	32.768	3 Router	32.768
4 Experts	45.097.156.608	4 Experts	11.274.289.152
5 LM Head	131.072.000	5 LM Head	131.072.000

Sparse Parameters *(all parameters)* **46.7B**

Active Parameters *(activated parameters)* **12.8B**

Efficient attention

Linear attention

source: https://haileyschoelkopf.github.io/blog/2024/linear-attn/

Let's revisit the $O(N^2)$ attention. Can we do better?

$$\text{Attention}(Q, K, V) = \text{softmax}\ (QK^\top)V$$

Let's zoom in! 🕵️

$$\text{softmax}\ (QK^\top)V = \frac{\exp(QK^\top)}{\sum_{i=1}^{L} \exp(QK_i^\top)}V$$

very expensive way to measure similarity between queries and keys

Linear attention: rearranging the computation

We can use a cheaper way of computing similarity:

$$\frac{\text{sim}(Q,K)}{\sum_{i=1}^{L}\text{sim}(Q,K_i)}V \qquad \text{sim}(Q,K) = \varphi(Q)\cdot\varphi(K) = \varphi(Q)\varphi(K)^\top$$

φ = optional dimensionality mapping, can be also identity

This also allows us to re-arrange matrix multiplications:

$Q\cdot K \in \mathbb{R}^{N\times N}$

$$\frac{\varphi(Q)\varphi(K)^\top}{\sum_{i=1}^{L}\varphi(Q)\varphi(K_i)^\top}V = \frac{\varphi(Q)\,\varphi(K)^\top V}{\varphi(Q)\sum_{i=1}^{L}\varphi(K_i)^\top}$$

$K\cdot V \in \mathbb{R}^{D\times D}$

Linear attention

The resulting operations are $O(N \cdot D^2)$, which is linear in sequence length.

Wo-hoo! 🎉

r/MachineLearning • před 3 r.
currentscurrents

[D] Why aren't we all using linear transformers?

There's a bunch of them - Linformer, Longformer, Performer, Nystromformer, Big Bird, etc etc. Plus a bunch more that have similar goals but don't necessarily aim for linear complexity, like memory-augmented transformers.

As far as I know, none of them have really seen much use. Even for image problems, which have very long input sizes, people are using regular transformers with tokenization schemes.

- Am I wrong? Are they actually good, or are at least some of them better than regular transformers?
- If not, what's wrong with them? Do they have lower accuracy? Are they slower to train?

26 · 8

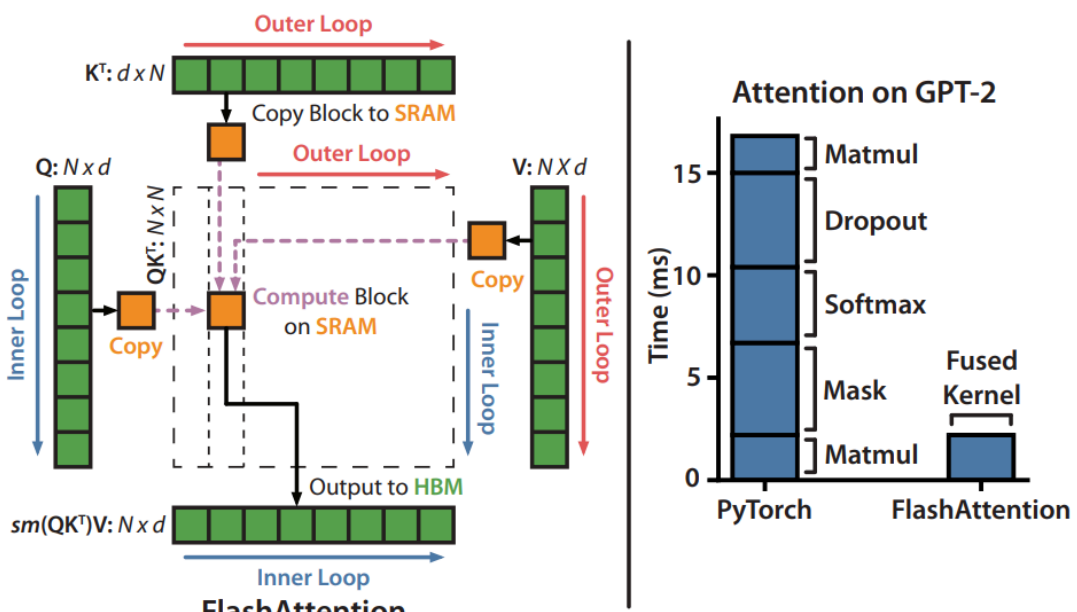
source: Reddit

FlashAttention

source: Dao et al. (2022)

Approximations like linear attention can degrade performance.

Meanwhile, **hardware optimizations** of the full attention mechanism can go a long way.



FlashAttention

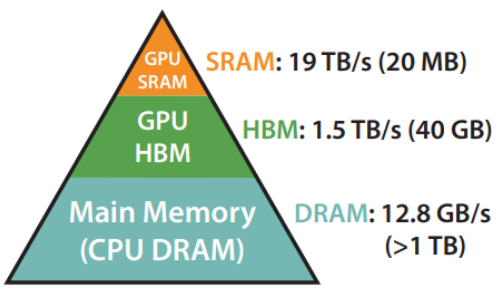
source: Dao et al. (2022)

FlashAttention re-arranges the operations to use the GPU memory more efficiently.

It is:

- **Fast**: 2-3x faster than baselines.
- **Memory-efficient**: linear in sequence length.
- **Exact**: uses no approximations.

It is now implemented in major LLM frameworks (PyTorch, HuggingFace, vLLM, ...)



Memory Hierarchy with Bandwidth & Memory Size

Summary

Summary

- **Quantization**: reducing the precision of model weights to save memory and speed up inference with minimal quality loss.
- **Knowledge distillation**: training a **student** model to mimic a **teacher** model.
- **LoRA**: low-rank adaptation; efficient approach for finetuning LLMs.
- **QLoRA** combines quantization with LoRA for even higher efficiency.
- **Mixture of Experts (MoE)**: only a subset of model parameters (**experts**) is activated per token, enabling faster inference with large total parameter counts.
- **Efficient attention**: linear attention reduces complexity from $O(N^2)$ to $O(N)$.
- **FlashAttention** provides exact attention with hardware-optimized memory access.

Links and resources

- HuggingFace: Quantization overview
- Dettmers et al. (2022): LLM.int8() – 8-bit Matrix Multiplication
- Ma et al. (2024): The Era of 1-bit LLMs (BitNet)
- Hinton et al. (2015): Distilling the Knowledge in a Neural Network
- Hu et al. (2021): LoRA: Low-Rank Adaptation of Large Language Models
- Dettmers et al. (2023): QLoRA: Efficient Finetuning of Quantized Language Models
- Jiang et al. (2024): Mixtral of Experts
- Maarten Grootendorst's blog: A Visual Guide to Mixture of Experts
- Dao et al. (2022): FlashAttention
- Schoelkopf (2024): Linear Attention
- LLaMA-Factory: finetuning framework
- HuggingFace PEFT: Parameter-Efficient Fine-Tuning