

Llama - Architecture

SOTA2 - Group C

Anton Kesy, Étienne Muser, Katharina Schindler, Lukas Fehrenbacher, Nico Ruschmann

Offenburg University of Applied Sciences

WS 2024/2025

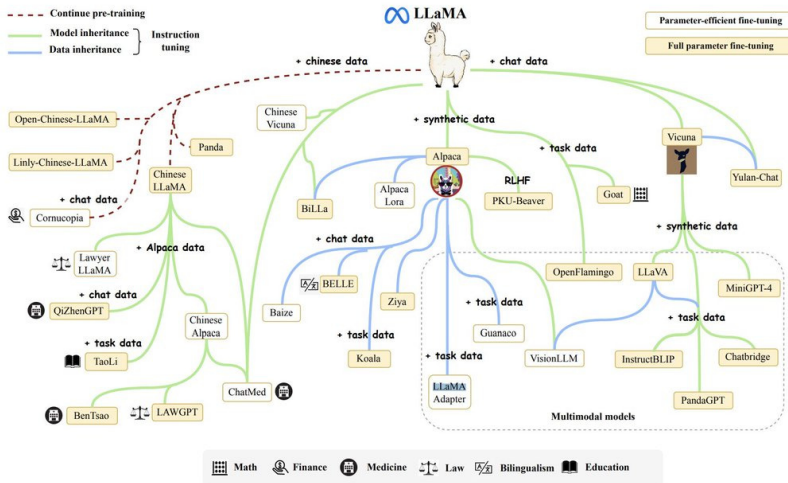


Was und wo ist Llama?

- Llama steht für **Large Language Model Meta AI**
 - früher *LLaMA*
- von Meta entwickelt
- Ziel: Forschung in KI und LLMs voranzutreiben
- Sourcecode unter GPL 3 auf GitHub
- Open-Source-Modelle zum Runterladen
- Gewerbliche Nutzung erfordert eine Lizenz

Llama Arten

10



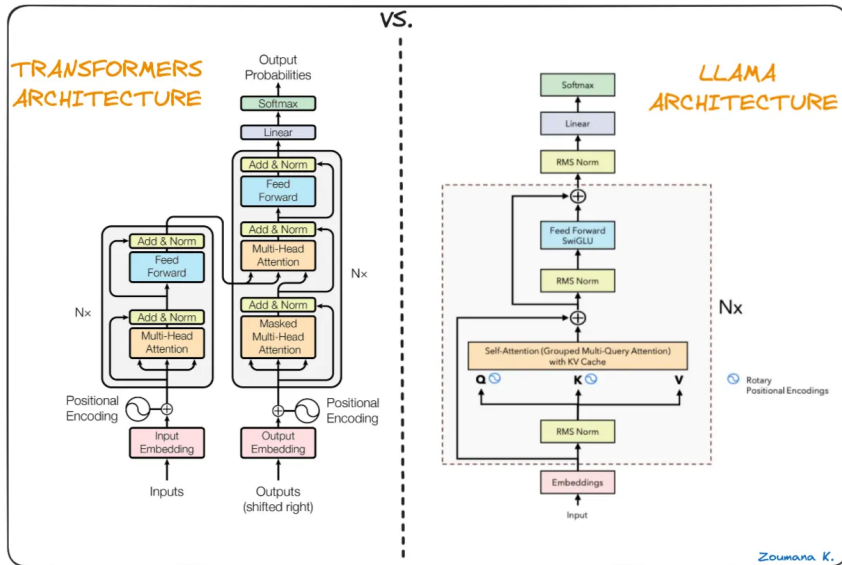
Wann wurde Llama veröffentlicht?

- Llama 1 - Februar 2023
- Llama 2 - Juli 2023
- Llama 3 - April 2024

Trainingsdaten:

- Common Crawl
- Public GitHub
- Wikipedia (20 Sprachen)
- Gemeinfreie Bücher (Project Gutenberg)
- LaTeX von wissenschaftliche Arbeiten (ArXiv)
- Stack-Exchange-Webseiten

Basis Architecture: Overview



Basis Architecture: RoPE

- **Rotary Positional Embeddings** are used in LLaMA for positional encoding.
- They enable dynamic representations of sequence positions.
- Rotation: Instead of fixed vectors like in classical transformers, RoPE performs rotations in the space of query and key vectors.
- RoPE flexibly represents the relative positions of input elements.
- The inner product between query and key vectors depends on their relative distance, making learning more efficient.

$$f_{q,k}(x_m, m) = \begin{pmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{pmatrix} \begin{pmatrix} W_{q,k}^{(11)} & W_{q,k}^{(12)} \\ W_{q,k}^{(21)} & W_{q,k}^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix}$$

Basis Architecture: RMS normalization

- Root Mean Square normalization
- RMS normalization is used in LLaMA instead of traditional layer normalization.
 - Layer normalization centers around the mean and scales by the standard deviation.
 - RMS normalization focuses only on scaling.
- RMS normalization scales inputs based on the root mean square (RMS) value.
- **Benefit:** Improves training stability by avoiding unnecessary computations.

$$\overline{a_i} = \frac{a_i}{RMS(a)} g_i \text{ where } RMS(a) = \sqrt{\frac{1}{N} \sum_{i=1}^N a_i^2}$$

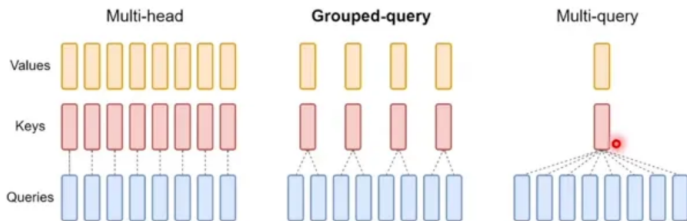
- Der **Key-Value-Cache** speichert Key- und Value-Vektoren, wodurch nur der Query-Vektor bei jedem Schritt aktualisiert werden muss, was die Inferenz beschleunigt.

Basis Architecture: Grouped multi query attention

- Das Problem mit Multi-Head Attention: Speicherverbrauch!
- KV-Cache-Size: $\text{batch size} * \text{layers} * \text{heads} * \text{embedded dim} * \text{context length} * 2$ (key and value) * 2 bytes (bfloat16)
- Für Llama2 70B:
 - Batch-Size: 64 (fine-tuning)
 - layers: 80?
 - heads: 64?
 - embedded dim: 8192?
 - context length: 4096
 - Speicher: 40960 GiB!

Lösung für das Speicherplatzproblem

- Nicht jede Query hat ihre einen eigenen Key und Value Vektor



- Kaum Performance-Verluste

| | BoolQ | PIQA | SIQA | Hella-Swag | ARC-e | ARC-c | NQ | TQA | MMLU | GSM8K | Human-Eval |
|-----|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|------------|------------|
| MHA | 71.0 | 79.3 | 48.2 | 75.1 | 71.2 | 43.0 | 12.4 | 44.7 | 28.0 | 4.9 | 7.9 |
| MQA | 70.6 | 79.0 | 47.9 | 74.5 | 71.6 | 41.9 | 14.5 | 42.8 | 26.5 | 4.8 | 7.3 |
| GQA | 69.4 | 78.8 | 48.6 | 75.4 | 72.1 | 42.5 | 14.0 | 46.2 | 26.9 | 5.3 | 7.9 |

- Swish-Gated Linear Unit
- Kombination aus Swish-Aktivierung und Lineartransformation
- Effiziente und leistungsstarke Aktivierungsfunktion mit weniger Rechenaufwand als ältere Methoden
- Weniger Rechenlast, trotzdem sehr gute Ergebnisse
- Genutzt in LLaMA für mehr Effizienz ohne die Genauigkeit zu verlieren
- Reduzierte Ressourcenanforderung bei gleicher Modellqualität

Llama v2 im Vergleich zu Llama v1:

Kontextlänge

- erweitert das Kontextfenster von 2048 Token auf 4096 Token
- mehr Informationen verarbeiten

Grouped-Query Attention (GQA)

- Verwendung von GQA
- verbessert die Skalierbarkeit der Inferenz

Weitere Aktualisierungen

- robustere Datenbereinigung
- aktualisierte Datenmischungen
- Training mit 40 % mehr Tokens

Llama v3 im Vergleich zu Llama v2:

Modellgröße

- Modell mit 405 Mrd. trainierbaren Parametern

Kontextlänge

Das Kontextfenster wurde in Llama v3 auf bis zu 128.000 Token erweitert

Grouped-Query Attention (GQA)

- GQA mit 8 Key-Value Heads
- verbessert Inferenzgeschwindigkeit
- Größe des Caches während der Dekodierung wird reduziert

Llama v3 im Vergleich zu Llama v2:

Tokenizer

- Tokenizer mit 128.000 Token
- Kombination vom tiktoken-Tokenizer (100.000) mit 28.000 zusätzlichen Token
- verbesserte Unterstützung für nicht englische Sprachen

Weitere Aktualisierungen

- verbesserte Datenqualität und -vielfalt
- umfangreiche Optimierungen der Trainingsinfrastruktur und -effizienz

Kennt Llama seine eigene Architektur?

```
> ollama run llama3.2
>>> Can you explain me in detail how your LLM architecture works?
I'll provide a high-level overview of the architecture of my large language model (LLM). Please note that I'm a complex system, and this explanation will not be exhaustive. My architecture is based on a transformer-based model, which is widely used in natural language processing tasks.

**Overview**

My LLM is a type of recurrent neural network (RNN) called a transformer. It's designed to process sequential data, such as text, and generate output sequences that are conditioned on the input sequence.

**Components**

The architecture consists of several components:

1. Input Embeddings: The input text is first embedded into a dense vector space using a learned embedding matrix. This step converts the text tokens (e.g., words, characters) into numerical vectors.
2. Positional Encoding: To preserve the sequential structure of the input data, positional encoding is added to the embeddings. This involves adding a fixed-length vector to each token's embedding based on its position in the sequence.
3. Self-Attention Mechanism: The self-attention mechanism is a core component of the transformer architecture. It allows the model to attend to different parts of the input sequence simultaneously and weigh their importance.
4. Encoder Layers: The encoder consists of multiple layers, each containing two sub-layers: self-attention and feed-forward networks (FFNNs). These layers are stacked together to process the input sequence sequentially.
5. Decoder Layers (if applicable): If you're generating text based on a given prompt or context, I have a decoder layer that takes in the output from the encoder and generates additional tokens.
6. Output Layer: The final output is generated by taking the last token from the sequence produced by the model.

Transformer Architecture

Here's a more detailed explanation of the transformer architecture:

1. Self-Attention Mechanism: The self-attention mechanism takes in three inputs:
   * Query (Q): The position of each token in the input sequence.
   * Key (K): The embedding of each token in the input sequence.
   * Value (V): The positional encoding added to each token's embedding.
   The attention weights are calculated as follows:

   
$$\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d}) \cdot V$$


   where 'd' is the dimensionality of the embeddings.

2. Encoder Layers: Each encoder layer consists of two sub-layers: self-attention and FFNNs. The outputs from each sub-layer are passed through a residual connection to preserve the input's original value.
```