

LLM für den Hausgebrauch

Karsten Keßler

17. Januar 2026

Inhaltsverzeichnis

1 LLM für den Hausgebrauch	1
1.1 Motivation	4
1.2 Transformer	6
1.3 Mathematische Grundlagen	7
1.4 Ein Rechenbeispiel	16
1.5 Lernprozess und Backpropagation	25
1.6 Grenzen des klassischen Language Modells	46
1.7 Fazit und Diskussion	47
1.8 Glossar	48

Stand: 17. Januar 2026

1 LLM für den Hausgebrauch

Inhaltlich führt dieses Dokument Schritt durch die wichtigsten Bausteine moderner Large Language Models (LLMs): von der historischen Entwicklung der KI (Symbolic AI → Machine Learning → Deep Learning → LLMs) über das Trainingsprinzip **Self-Supervised Learning** (Next-Token-Prediction) bis zur grundlegenden Verarbeitungskette im Modell (**Tokenisierung → Embeddings → Transformer/Self-Attention → Logits/Softmax → Loss → Backpropagation**). Ein Schwerpunkt liegt auf der Intuition hinter **Self-Attention** (Query/Key/Value, Attention-Gewichte, Kontextvektoren) und darauf, wie daraus kontextabhängige Bedeutungen entstehen.

Darüber hinaus behandelt die Präsentation wichtige Systemaspekte jenseits des reinen Modells: **Vektor-Datenbanken** als semantischer Wissensindex, **Retrieval Augmented Generation (RAG)** als Kombination aus Retrieval und Generierung sowie (als Ausblick) **Memory-Konzepte** wie bei Titans. Abschließend werden typische Grenzen und Risiken von LLMs eingeordnet, u.a. **Halluzinationen**, begrenzte Kontextfenster, Kosten/Compute und Aspekte wie Robustheit und Sicherheit.

Zielgruppe: Das Material richtet sich an Studierende sowie technisch interessierte Praktiker:innen, die LLMs nicht nur „benutzen“, sondern die zentralen Konzepte und den Rechenweg dahinter verstehen möchten (Produkt- oder Toolkenntnisse sind nicht erforderlich).

Voraussetzungen: Erwartet werden grundlegende Programmiererfahrung (z.B. in Python) und ein Basisverständnis von Mathematik für Machine Learning (Vektoren/Matrizen, einfache Funktionen/Gradienten). Details zu Ableitungen,

Softmax/Cross-Entropy oder Rechenbeispielen werden im Dokument schrittweise aufgebaut; tiefe Vorkenntnisse in Deep Learning oder Transformer-Architekturen sind nicht notwendig.

Wo es passt, sind zusätzlich **Links zu Python-Notebooks** angegeben, mit denen sich zentrale Konzepte praktisch nachvollziehen lassen (z.B. mathematische Grundlagen, Tokenisierung, Attention-Rechenbeispiele, Next-Token-Prediction oder semantische Suche). Dadurch eignet sich das Dokument sowohl als Begleitmaterial zur Präsentation als auch als Nachschlagewerk zum Wiederholen und Vertiefen.

Agenda

1. Motivation
2. Transformer-Architektur & Self-Attention
3. Mathematische Grundlagen
4. Ein Rechenbeispiel: Wie arbeitet das Modell?
5. Wie lernt das Modell?
6. Wie entscheidet das Modell?
7. Modellarchitekturen im Überblick
8. Titans: Ein Ausblick auf neue Architekturen
9. Grenzen von LLMs
10. Fazit & Diskussion
11. Glossar



Einordnung

- Überblick über den roten Faden: Von Motivation und Begriffen bis zu Grenzen und Ausblick.
- Erwartungsmanagement: Fokus auf Grundprinzipien (Transformer, Training, Tokenisierung), nicht auf Produktmarketing.
- Orientierung für die folgenden Kapitel und Demos: Welche Folien sind konzeptionell, welche rechnerisch.

1.1 Motivation

Motivation: Vom Token zur Antwort

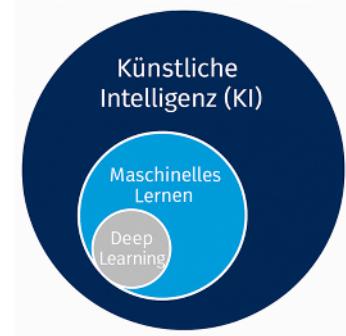
**Wie Large Language Models wirklich funktionieren am Beispiel:
„Paris ist die Hauptstadt von?“**

ML ist ein Teilgebiet der KI, bei dem Computer aus Beispielen lernen, ohne explizit programmiert zu sein.

Ziel: Muster erkennen, Vorhersagen treffen (z. B. Spamfilter, Bilderkennung).

DL ist eine spezielle Form des maschinellen Lernens, die auf tiefen, künstlichen neuronalen Netzen mit vielen Schichten basiert.

Besonders gut geeignet für komplexe Aufgaben wie Spracherkennung, Bildanalyse oder Textgenerierung (z. B. GPT).



Generative Pretrained Transformer (GPT): ein Deep-Learning-Modell

Generative: Das Modell erzeugt neue Texte (nicht nur klassifiziert)

Pretrained: Vortrainiert auf riesigen Textmengen, bevor es feingetunet wird

Transformer: Architektur mit Self-Attention, Kontext über viele Tokens hinweg

Wie kann ein Modell ohne Weltwissen korrekte Antworten erzeugen?

Wie entsteht „Verstehen“ allein durch Statistik?

Mensch

Kennt Bedeutung & Kontext

Kann abstrahieren & hinterfragen

Lernt durch Erfahrung & Feedback

GPT

Kennt nur Wahrscheinlichkeiten

Errechnet nächstwahrscheinliches Token

Lernt durch Milliarden Token + Loss

Live-Demo mit LM Studio

```
You
Rom ist die Hauptstadt von

google/gemma-3-12b
Rom ist die Hauptstadt von Italien.

You
Paris ist die Hauptstadt von

google/gemma-3-12b
Paris ist die Hauptstadt von Frankreich.

⌚ 3.79 tok/sec • 8 tokens • 2.40s to first token • Stop reason: EOS Token Found
```



Einordnung

- Historische Entwicklung der KI: Symbolic AI → Machine Learning → Deep Learning → LLMs.
- Verschiebung von handcodierten Regeln zu datengetriebenen Repräsentationen (Features/Embeddings).
- Einordnung von GPT in die DL-/NLP-Landschaft: Transformer als Architektur, Decoder-only als Modellfamilie.

Einordnung von GPT & Co.

Das Modell sieht Texteingaben: „Paris ist die Hauptstadt von“
Es soll das nächste Token korrekt vorhersagen (z.B. „Frankreich“)
Das Modell sieht: Tokens, keine Bedeutungen

„von“ → schaut stark auf „Hauptstadt“ und etwas auf „Paris“ → daraus entstehen die Gewichtungen

Lernart	Kurzbeschreibung	Typisches Beispiel
Supervised Learning	Lernen mit menschlichen Labels	Katze vs. Hund, Bildklassifikation
Self-Supervised Learning	Labels entstehen aus den Eingabedaten selbst	Next Token Prediction in GPT, Masked LM in BERT
Unsupervised Learning	Keine Labels, Modell erkennt Strukturen selbst	Clustering (z. B. K-Means), Word2Vec-Embeddings
Reinforcement Learning	Lernen über Belohnung/Bestrafung nach Aktionen	Schach, Go, RLHF-Finetuning von ChatGPT

Warum hier Self-Supervised?

Das Modell bekommt Milliarden Textbeispiele.

Es „lernt aus sich selbst“:

Bsp: Aus dem Satz „Paris ist die Hauptstadt von Frankreich“ schneidet man z. B. „Frankreich“ ab → Aufgabe: Vorhersage dieses Wortes.

Kein Mensch muss manuell annotieren, das macht Self-Supervised Lernen mächtig für LLMs.

Technisch gesehen ist diese Beispiel Supervised Learning (weil Eingabe und Ziel vorliegen), aber die Labels erzeugt der Text selbst → darum nennt man es Self-Supervised Learning.



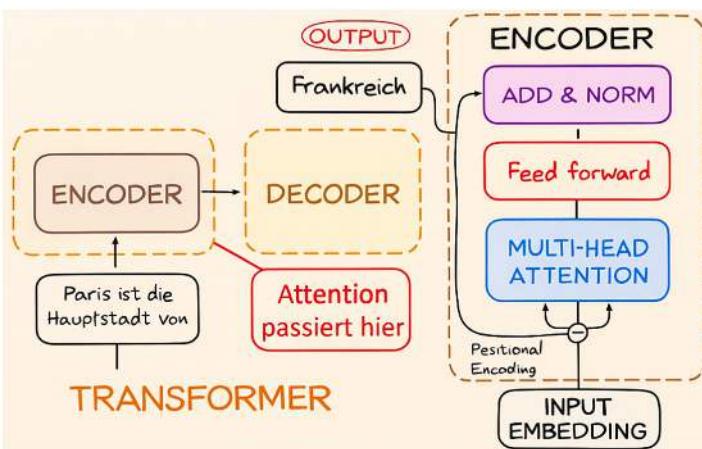
Einordnung

- Self-Supervised Learning: Next-Token-Prediction als Trainingsziel (Labels entstehen aus dem Text selbst).
- Abgrenzung zu Supervised, Unsupervised (Clustering/Embeddings) und RL/RLHF als Feintuning-Schritt.
- Intuition: Modelle lernen statistische Regularitäten und Weltwissen indirekt über Textzusammenhänge.
- Beispiel: Paris → Frankreich als Illustration für Kontextnutzung und semantische Assoziation.

Transformer

LLM Pipeline Überblick:

Text → Tokenisierung → Embedding → Attention → Output → Softmax → Loss → Backpropagation



→ „Paris ist die Hauptstadt von“

→ **Tokenisierung:** Zerlegung in kleinere Token, z. B. „Paris“, „ist“, „die“ ...

→ **Embedding:** Jedem Token einen Vektor zuweisen

→ **Attention:** Kontext wird einberechnet: „Auf wen soll ich achten?“

→ **Output** Es wird ein Vektor für nächsten Token erzeugt

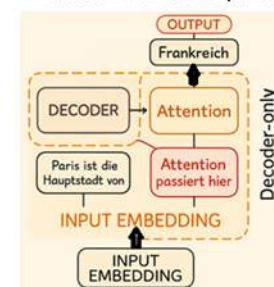
→ **Softmax** Umwandlung Wahrscheinlichkeiten für alle möglichen Wörter

→ **Loss:** Vergleich mit dem Zielwort, um den Fehler zu berechnen

→ **Backpropagation:** Fehler wird zurückgeleitet und Gewichte angepasst damit das Modell lernt

Decoder-Encoder-Architektur (links, z.B. T5)
(vereinfacht): In dieser Präsentation liegt der Fokus aber auf dem Decoder-only Modell GPT, insbesondere auf dem Self-Attention-Mechanismus. Einem Modell, das Token für Token vorhersagt, gesteuert durch Self-Attention.

Decoder-Architektur (z.B. GPT)



Umgangssprachlich wird die Decoder-Architektur als Language Modell bezeichnet.

Architekturtyp	Beispiel	Anwendung	Merkmal
Encoder-only	BERT	Klassifikation, NER (Erkennung benannter Entitäten)	Tokens „sehen sich gegenseitig“, ganze Eingabe sichtbar (bidirektional)
Decoder-only	GPT	Textgenerierung	Tokens sehen nur vorherige Tokens (autoregressiv)
Encoder–Decoder	T5, BART, Marian	Übersetzung, Zusammenfassung	Encoder verarbeitet Eingabe, Decoder generiert Ausgabe (mit Cross-Attention)



Einordnung

- Gesamtpipeline: Text → Tokenisierung → Embedding → (mehrfach) Self-Attention/FFN → Output-Logits → Softmax → Loss → Backpropagation.
- Trennung von Vorwärtsrechnung (Inference) und Training (zusätzlich Loss & Gradienten).
- Decoder-only (GPT) für Generierung, Encoder-only (BERT) für Verständnis/Maskierung, Encoder-Decoder (T5/BART) für Sequenz-zu-Sequenz.

Mathematische Grundlagen

Gradientenberechnung

Ziel: Verlustfunktion L minimieren

Gradient = Richtungsvektor des stärksten Anstiegs:

$$\nabla L = [\partial L / \partial w_1, \partial L / \partial w_2, \dots, \partial L / \partial w_n]$$

Kettenregel: $\partial L / \partial W = \partial L / \partial z \cdot \partial z / \partial W$

Beispiel für lineare Schicht:

$$z = W \cdot x \rightarrow \partial z / \partial W = x^T$$

$$\rightarrow \partial L / \partial W = (p - y) \cdot x^T$$

Update-Regel (Gradient Descent):

$$W_{\text{neu}} := W_{\text{alt}} - \eta \cdot \partial L / \partial W$$

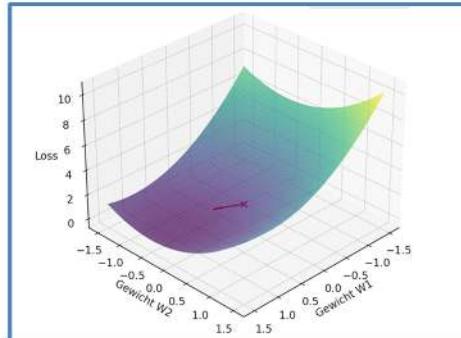
Was ist z ?

Voraktivierung eines Neurons:

$$z = W \cdot x + b$$

Lineare Kombination aus Gewichten und Eingabe

Die Größe, über die sich der Fehler an die Gewichte weitergibt



Warum Kettenregel?

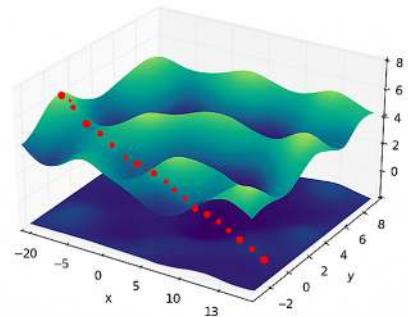
Die Lossfunktion hängt nicht direkt von den Gewichten ab.

$$W \rightarrow z \rightarrow L$$

Darum:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W}$$

Bergbild mit 2 Gewichten (im Rechenbeispiel 4, tatsächlich Millionen)



Loss-Landschaft mit dem Ziel der Minimierung

Die Wörter erzeugen Vektoren. Diese Vektoren fließen in den Attention-Mechanismus, wo diese Wörter gegenseitig aufeinander „schauen“.

Am Ende werden die Gewichte so optimiert, dass „von“ z. B. besser erkennt, dass „Hauptstadt“ wichtig ist.

Dazu verwendet man die Kettenregel, weil der Fehler über mehrere Zwischenschritte von den Gewichtsmatrizen abhängt.

Die Kettenregel erlaubt es, diese Zwischenschritte sauber zu verrechnen.

Der Gradient sagt dem Modell, in welcher Richtung und wie stark die Gewichte geändert werden sollen, damit den Fehler zu verringert wird.



Einordnung

- Optimierung neuronaler Netze: Kettenregel, Gradient Descent, Interpretation der Loss-Landschaft.
- Rolle der Lernrate: Stabilität vs. Geschwindigkeit (Overshooting vs. langsame Konvergenz).
- Lokale Minima, Sattelpunkte und warum große Netze oft trotzdem gut trainierbar sind (Intuition).



Notebooks

- Open in Colab: Mathematische Grundlagen (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/kapitel_3_mathematische_grundlagen.ipynb)

Matrizenrechnung

- Matrixprodukt (Matrix-Vektor-Produkt, lineare Abb.):

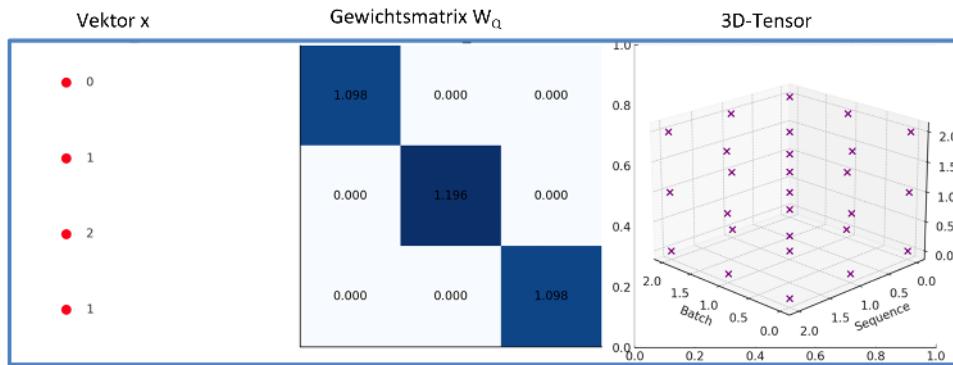
$z = W \cdot x$ (W Gewichtsmatrix,
x Eingabevektor von „von“ z.B. : $[1, 2, 1, 0]$,
z Vektor mit anders codiertem Inhalt)

- $\partial(Wx)/\partial W = x^T$
- $\partial L/\partial W = (p - y) \cdot x^T$ (äußeres Produkt)
- Interpretation: Gradientenmatrix
- Dimensionsregeln:

$$W \in \mathbb{R}^{n \times d}, x \in \mathbb{R}^d \rightarrow z \in \mathbb{R}^d$$

→ Matrizenrechnung in neuronalen Netzen bedeutet:
Man mischt und gewichtet die alten Informationen neu, um passendere Bedeutungsvektoren x zu erzeugen.

Vektoren bewegen sich im Vektorraum \mathbb{R}^d . Die Eingabevektoren (z. B. für „Paris“, „Hauptstadt“, „von“ ...) sind bereits in diesem Vektorraum \mathbb{R}^4 (tatsächlich z. B. im \mathbb{R}^{768}). Die Matrizen W_Q, W_K, W_V transformieren diesen Vektorraum von x nach z (Dimension bleibt erhalten) durch Projektion, Drehung usw. Attention operiert auf den geometrischen Beziehungen in diesem Raum (Ähnlichkeiten, Richtungen). Dadurch entstehen neue Bedeutungsvektoren, die neue Informationen enthalten.



Einordnung

- Matrixrechnung als Grundbaustein: Lineare Projektionen transformieren Vektoren in neue Repräsentationsräume.
- $z = W \cdot x$: Dimensionalitäten erklären (Eingabevektor, Gewichtsmatrix, Ausgabedimension).
- Verbindung zu Transformer: W_Q, W_K, W_V und weitere Projektionsmatrizen sind genau solche linearen Abbildungen.
- Interpretation: Gewichte kodieren gelernte Muster; Multiplikation ist „Feature-Mischung“/Aggregation.

Wahrscheinlichkeitsrechnung

LLMs sind letztendlich Wahrscheinlichkeitsmodelle: Sie lernen, für jede mögliche Fortsetzung eines Textes die bedingte Wahrscheinlichkeit korrekt vorherzusagen.

$P(E)$: Wahrscheinlichkeit des Ereignis E

$P(A|B)$: Bedingte Wahrscheinlichkeit von B unter der Bedingung A

Satz von Bayes:

$$P(A|B) = P(B|A) \cdot P(A) / P(B)$$

Erwartungswert: $E[X] = \sum x_i \cdot P(x_i)$

Klassifikation in LLMs:

Likelihood $P(y|x)$ maximieren → rechnen mit Cross-Entropy-Loss

$P(y|x)$ = Wahrscheinlichkeit, dass das Modell das korrekte nächste Wort y vorhersagt, wenn der Kontext x gegeben ist.

→ Eingabe-Sequenz:

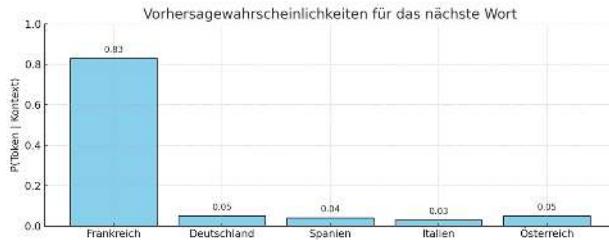
"Paris ist die Hauptstadt von"

→ LLM berechnet:

$P(\text{"Frankreich"} | \text{"Paris ist die Hauptstadt von"})$

→ Cross-Entropy vergleicht diese Vorhersage mit dem tatsächlichen Zielwort. Erwartungswert (durchschnittlicher Fehler über alle Trainingsbeispiele), dieser wird minimiert.

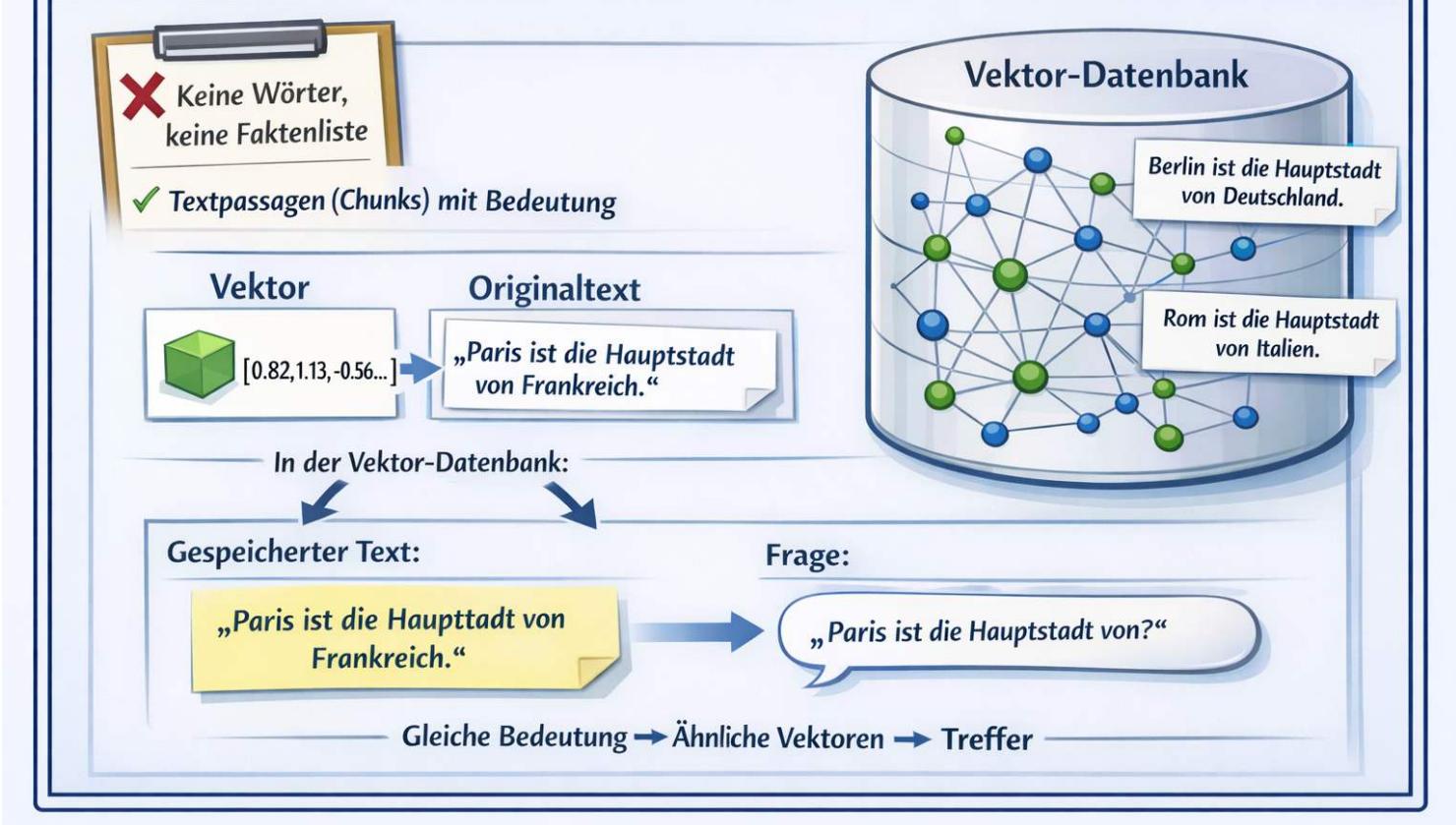
→ Likelihood-Wahrscheinlichkeit maximieren: Modell soll dem korrekten nächsten Wort (z. B. „Frankreich“) möglichst hohe Wahrscheinlichkeit geben.



Einordnung

- Von Logits zu Softmax: Das Modell liefert zunächst unnormierte Scores (Logits) pro Token.
- Softmax macht daraus eine Wahrscheinlichkeitsverteilung über das Vokabular: $p(\text{next token} | \text{Kontext})$.
- Numerische Stabilität: Softmax wird praktisch stabil gerechnet (z.B. LogSumExp-Trick).
- Vorbereitung für Training: Cross-Entropy vergleicht die Verteilung mit dem Ziel-Token.

Was muss in der Vektor-Datenbank stehen, damit „Paris“ gefunden wird?



Einordnung

- Vektor-Datenbanken speichern keine Wörter, sondern Embeddings von Textpassagen (Chunks).
- Suche erfolgt semantisch: Ähnliche Bedeutung ⇒ ähnliche Vektoren (Nearest Neighbor).
- Ergebnis ist Kontextmaterial, das ein LLM zur Antwortgenerierung nutzen kann (statt reines Par-
amterwissen).

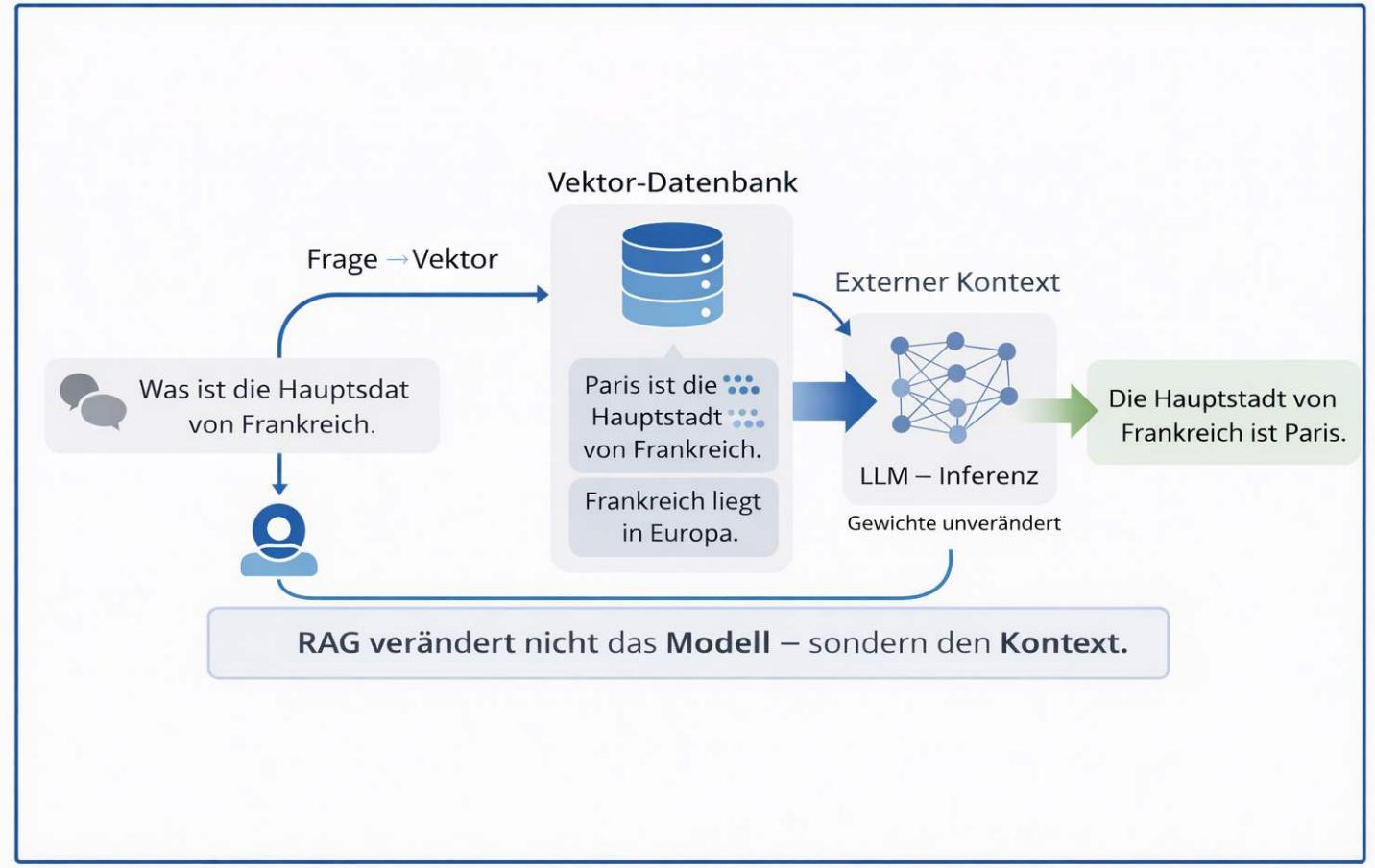
Beispiel:

„Paris ist die Hauptstadt von Frankreich.“ Die KI sucht **nicht nach dem Wort „Paris“**, sondern nach **Texten mit gleicher Bedeutung**.



Hinweis

- Die Vektor-Datenbank ist ein semantisch durchsuchbarer Wissensindex.



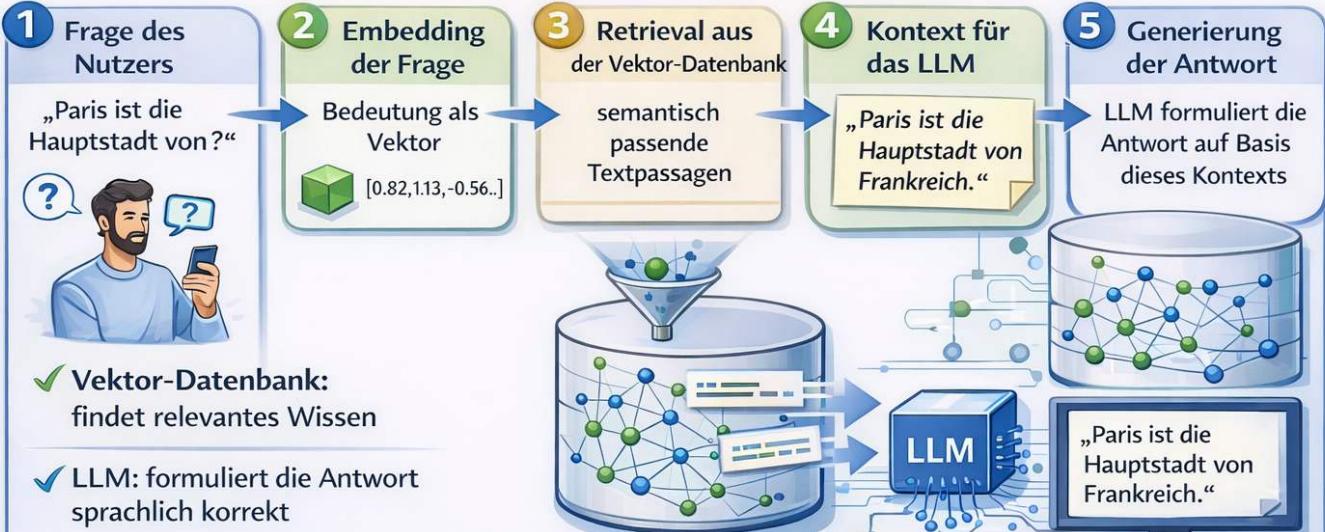
Einordnung

- Brücke zwischen Vektor-Datenbank und RAG-Pipeline: Das Modell bleibt gleich, nur der Kontext wird ergänzt.
- Retrieval liefert passende Textpassagen; das LLM nutzt sie zur Antwort – ohne Retraining.
- Merksatz: RAG erweitert das Wissen „zur Laufzeit“ (über Prompt-Kontext), nicht über neue Gewichte.

Retrieval Augmented Generation (RAG)

Antworten mit externem Wissen

RAG kombiniert Nachschlagen und Generieren.

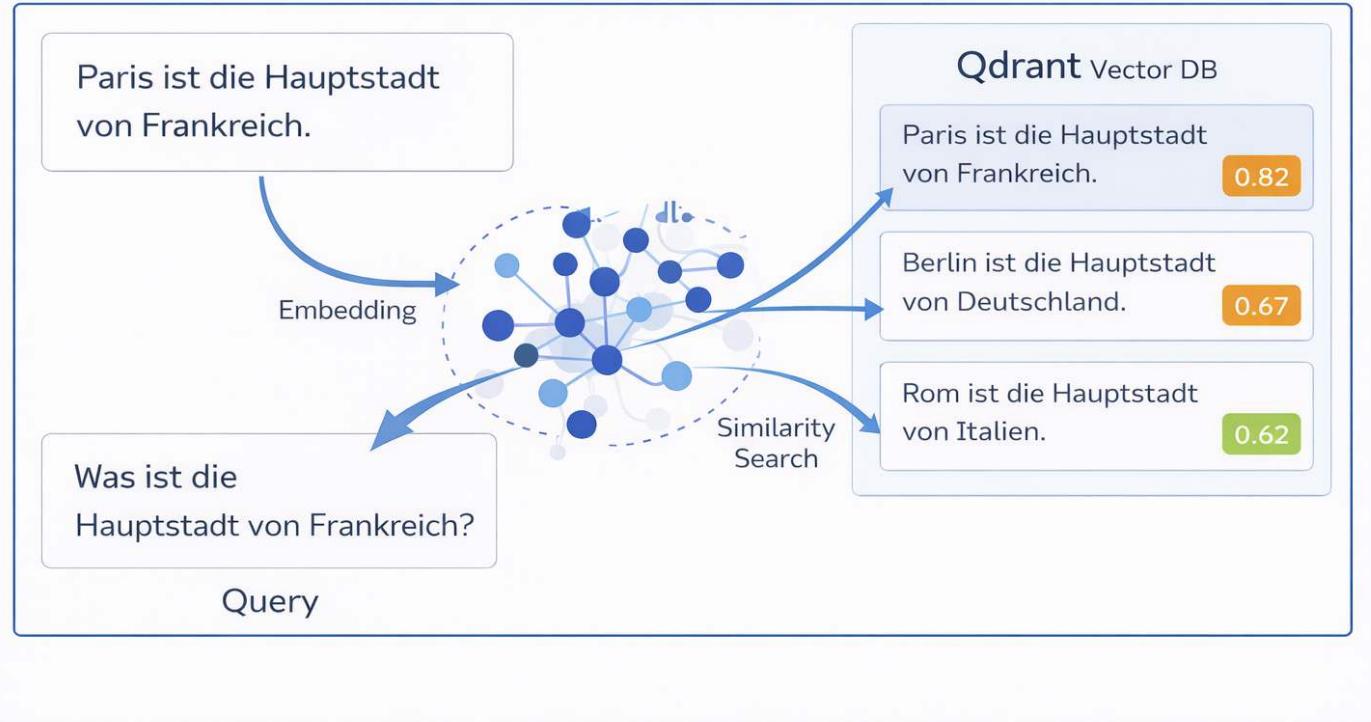


Einordnung

- RAG kombiniert Retrieval (Kontextbeschaffung) und Generation (Antwortformulierung).
- Ablauf: Frage → Embedding → semantische Suche → relevante Passagen → Prompt mit Kontext → LLM-Antwort.
- Vorteil: Aktualisierbares Wissen ohne Modell-Retraining; Quellen können zitiert/überprüft werden.
- Rollen: Vektor-DB als Wissensindex, LLM als Sprach- und Schlussfolgerungsmodul (auf Basis des Kontexts).

Live-Demo: Semantische Suche mit einer Vektor-Datenbank

Qdrant (Local Mode, ohne Docker)



Einordnung

- Demonstriert den Retrieval-Schritt von RAG in einer echten Vektor-DB (Qdrant, Local Mode).
- Pipeline: Dokumente chunking → Embeddings → Index; Anfrage → Query-Embedding → Similarity Search.
- Wichtig: Gespeichert werden Textpassagen (mit Metadaten), nicht Stichwortlisten.
- Ergebnisinterpretation: Trefferqualität hängt von Chunking-Strategie und Embedding-Modell ab.



Notebooks

- Open in Colab: Qdrant Demo (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/qdrant_demo.ipynb)

Tokenisierung

Beispiel-Wortfolge:

["Paris", "ist", "die", "Hauptstadt", "von"]

Tokenfolge: ['Paris', 'is', 'the', 'capital', 'of']

Hinweis: Ein Wort besteht aus mehreren Tokens.

```
You  
Paris ist die Hauptstadt von  
  
google/gemma-3-12b  
Paris ist die Hauptstadt von Frankreich.  
4.63 tok/sec 8 tokens 1.56s to first token • Stop reason: EOS Token Found
```

Eingabetext: „Paris ist die Hauptstadt von“ → 8 Tokens

⌚ Eingabetext: Paris ist die Hauptstadt von
🕒 Token-IDs: [40313, 318, 83, 4656, 49696, 457, 38863, 18042]
🕒 Token-Strings: ['Paris', 'is', 'the', 'capital', 'of']
🕒 Anzahl der Tokens: 8

Die Token-ID entspricht dem Index des Tokens im Vokabular.
Das Vokabular ist groß (Zehntausende Tokens), aber dadurch kann das Modell praktisch jeden Text darstellen. Und gegenüber der Gesamtzahl aller möglichen Wörter ist es sogar eher klein.

⌚ Eingabetext: Berlin ist die Hauptstadt von Deutschland. Paris ist die Hauptstadt von
🕒 Token-IDs: [24814, 2815, 318, 83, 4656, 49696, 457, 38863, 18042, 1024, 40768, 1044, 13, 6342, 318, 83, 4656, 49696, 457, 38863, 18042]
🕒 Token-Strings: ['Berlin', 'is', 'the', 'capital', 'of', 'Germany', 'Paris', 'is', 'the', 'capital', 'of']
🕒 Anzahl der Tokens: 21

Embeddings als Matrix $X \in R^{21 \times 768}$ (GPT2). Jeder Token wird in einen Vektor der Länge 768 abgebildet.



Einordnung

- Sprache wird als Token-Sequenz modelliert (diskrete Einheiten statt „ganze Sätze“).
- Token können Wörter, Subwords oder Zeichen/Sonderzeichen sein (abhängig vom Tokenizer).
- Tokenisierung legt fest, welche Muster das Modell überhaupt unterscheiden/lernen kann (OOV, Kompression, Mehrdeutigkeit).
- Praktische Konsequenz: Prompt-Design und Kosten hängen stark an der Tokenanzahl.



Notebooks

- Open in Colab: Tokenisierung (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/tokenisierung_beispiel.ipynb)

Embeddings

Ein Embedding ist ein Zahlenvektor, der die Bedeutung eines Tokens im Modell darstellt.

Jeder Token wird in einen Vektor im Raum R^{768} umgewandelt.

Der Token Paris hat in „Paris ist die Hauptstadt von“ zunächst den gleichen Vektor wie „Paris hat gestern gewonnen“ (Token-Embedding (roh), trägt nur Grundinformationen).

👉 Vektor für das erste Token:

$[-0.229 \quad 0.065 \quad -0.211 \quad -0.089 \quad -0.02 \quad -0.31 \quad 0.286 \quad 0.213 \quad -0.179 \quad 0.119]$

Kontextuelles Embedding:

Dasselbe Token hat je nach Kontext unterschiedliche kontextualisierte Vektoren, weil der Vektor durch Attention verändert wird:

● Vergleich für Token 'paris' an Index 1

◆ Rohes Token-Embedding (erste 10 Werte): $[-0.0218334 \quad -0.06728435 \quad -0.02705017 \quad 0.01860892 \quad -0.05082323 \quad -0.03754737$

$-0.06119448 \quad -0.03154157 \quad 0.03267446 \quad -0.0200461]$

◆ Kontextualisiertes Embedding (erste 10 Werte): $[-0.28785715 \quad 0.14921233 \quad 0.12685123 \quad -0.08659903 \quad -0.26973185 \quad -0.03011261$

$0.28518507 \quad 0.3154861 \quad 0.1849614 \quad -0.6070591]$

```
[ 0.018 -0.125 -0.127 -0.067 0.101 -0.185 0.084 0.126 -0.221 0.012]
[-0.523 0.11 -0.191 0.297 0.576 0.862 2.452 1.073 -0.22 0.249]
[-0.79 0.071 -0.882 -0.131 0.367 -0.162 0.886 0.638 -0.205 -0.244]
[ 0.152 -0.573 -0.493 0.5 0.181 -0.015 1.245 0.096 -0.517 -0.223]
[ 0.44 -0.301 0.778 0.406 -0.102 -0.254 1.907 -0.232 0.053 0.081]
[ 0.801 -0.738 -0.411 0.203 -0.479 -0.599 0.587 -1.193 0.208 -0.259]
[-0.79 -0.751 -0.283 1.425 0.281 -0.073 0.93 -0.154 -0.893 0.624]
[-0.166 -0.767 -0.597 0.617 0.334 -0.077 4.264 -0.175 -0.811 0.743]
[-0.119 -0.335 -0.33 0.374 0.266 0.074 4.47 -0.332 -0.103 0.267]
[ 0.693 -0.28 0.482 0.239 0.346 -0.698 -1.758 -1.165 -0.191 -0.191]
[ 0.288 -0.156 -0.157 1.448 0.504 -0.231 6.008 -0.06 -1.086 0.983]
[ 0.377 -0.246 -0.752 0.55 0.34 0.245 3.727 -0.102 -0.392 0.539]
[ 0.908 0.219 0.555 0.746 0.101 0.689 0.405 0.117 0.117 0.541]
```

Die hier gezeigten Embedding-Vektoren entstehen nicht zufällig. Sie wurden beim Training des Sprachmodells (z. B. GPT, Gemma) gelernt und sind jetzt fest im Modell gespeichert.

Bei der Verwendung des Modells (wie in LM Studio) wird jeder Token einfach in der gelernten Embedding-Matrix nachgeschlagen. Es findet kein Training und keine Veränderung mehr statt.

Verändert wird nur der Vektor x , der das Token repräsentiert. Die Gewichtsmatrizen bleiben gleich.

→ Mathematisches Arbeiten mit Bedeutungsräumen:

Wörter werden in Vektoren übersetzt, um mit ihnen zu rechnen, als wären sie Punkte in einem Koordinatensystem (z.B. Abstand von Vektoren). Diese Vektoren können miteinander verglichen werden, z.B. per Kosinus-Ähnlichkeit.



Einordnung

- Tokenisierung bestimmt die „Auflösung“ der Texteingabe: Granularität von Einheiten im Embedding-Raum.
- Auswirkungen: OOV-Handling, Robustheit bei seltenen Wörtern, Umgang mit Komposita (Deutsch) und Tippfehlern.
- Modellqualität hängt auch vom Tokenizer ab (z.B. Vokabulargröße vs. Sequenzlänge).
- Verweis auf Zusatzmaterial für Details und Beispiele.

1.4 Ein Rechenbeispiel

Ein Rechenbeispiel: Wie arbeitet das Model?

Betrachtet wird ein konkretes Mini-Beispiel, in dem das Token „von“ im Fokus steht.

Ziel: Das Modell soll lernen, dass das Wort „von“ in diesem Satz besonders eng mit „Hauptstadt“ und „Frankreich“ verknüpft ist.

Beispiel-Satz: „Paris ist die Hauptstadt von Frankreich von“

Fokus: Das Wort „von“ steht im Satz zwischen „Hauptstadt“ und „Frankreich“. Das Modell soll lernen, dass es diese beiden Wörter miteinander verbindet.

Dafür berechnet das Modell:

1. Einen Query-Vektor Q für „von“
2. Key- und Value-Vektoren K/V für die anderen Wörter im Satz
3. Attention-Scores: Wer ist wichtig für „von“?

Jeder Token betrachtet **alle anderen Tokens**, inklusive sich selbst, das nennt man **Self-Attention**

4. Einen Output-Vektor, also den kontextualisierten Vektor für „von“

Was passiert im Hintergrund? Im Training passt das Modell seine Gewichtsmatrizen an, damit „von“ nächstes Mal noch besser versteht, worauf es achten soll.

- Q/K/V entstehen aus Gewichtsmatrizen
- „von“ vergleicht sich mit allen anderen
- Bei Fehlern werden die Gewichte mit Backpropagation angepasst



Einordnung

- Ausgangspunkt: Tokens sind bereits in Embeddings überführt; das Modell verarbeitet alle Tokens parallel.
- Self-Attention berechnet für jedes Token eine gewichtete Mischung der anderen Tokens (Kontextaggregation).
- Ergebnis: Kontextabhängige Bedeutung (z.B. „von“ als Relation, nicht isoliertes Wort).
- Intuition: Statt fester Regeln werden Relevanzen aus Daten gelernt (Gewichtungen als „weiche Regeln“).

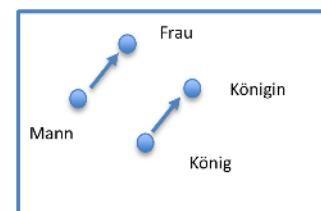
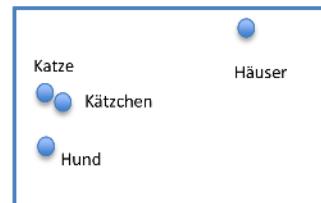
Wort-Embeddings visualisieren

7 Semantische Dimensionen, tatsächlich hunderte oder tausende

Wort	Lebewesen	Katzenartig	Menschlich	Geschlecht	Königlich	Verb	Plural
Katze	0.6	0.9	0.1	0.4	-0.7	-0.3	-0.2
Kätzchen	0.5	0.8	-0.1	0.2	-0.6	-0.5	-0.1
Hund	0.7	-0.1	0.4	0.3	-0.4	-0.1	-0.3
Häuser	-0.8	-0.4	-0.5	0.1	-0.9	0.3	0.8
Mann	0.6	-0.2	0.8	0.9	-0.1	-0.9	-0.7
Frau	0.7	0.3	0.9	-0.7	0.1	-0.5	-0.4
König	0.5	-0.4	0.7	0.8	0.9	-0.7	-0.6
Königin	0.8	-0.1	0.8	-0.9	0.8	-0.5	-0.9

Wort-Embeddings kodieren semantische Bedeutung in Vektoren. Ähnliche Bedeutungen → ähnliche Vektoren.

Reduzierter 2-dimensionaler Raum:



Mann verhält sich zu Frau wie König zu Königin, Vektorpfeil repräsentiert die semantische Richtung männlich → weiblich



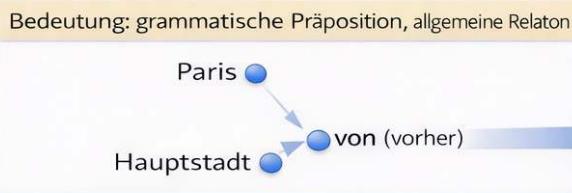
Einordnung

- Wort-Embeddings kodieren Bedeutung in kontinuierlichen Vektorräumen (Distributional Semantics).
- Nähe im Raum korreliert mit semantischer Ähnlichkeit; Richtungen können Relationen ausdrücken (Analogien).
- Embeddings sind die Schnittstelle zwischen diskreten Tokens und differentiellen neuronalen Netzen.

Kontextualisierte Embeddings – vor und nach Self-Attention

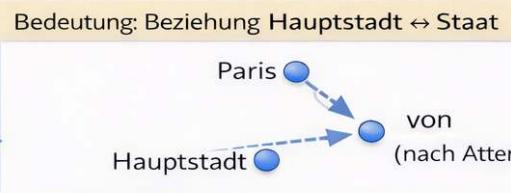
Vor Self-Attention (statisches Embedding)

- Jedes Token besitzt zunächst ein kontext-unabhängiges Embedding
- Der Vektor repräsentiert die „allgemeine Bedeutung von „von““
- Alle Vorkommen von „von“ starten mit ähnlicher Repräsentation



Nach Self-Attention (kontextualisiertes Embedding)

- Das Embedding von „von“ wird unter Berücksichtigung aller anderen Tokens neu berechnet
- Tokens wie „Hauptstadt“ und „Frankreich“ erhalten hohe Attention-Gewichte
- Der neue Vektor enthält konkrete Kontextinformation



Einordnung:

Wort-Embeddings sind im Transformer nicht statisch. Durch Self-Attention wird jedes Token-Embedding kontextabhängig angepasst und trägt anschließend Informationen über relevante andere Tokens im Satz.

Merksatz: Self-Attention verwandelt statische Wort-Embeddings in kontextualisierte Bedeutungen.



Einordnung

- Self-Attention macht Embeddings kontextabhängig: gleiche Token-ID kann je nach Satz verschiedene Vektoren erhalten.
- Das reduziert Mehrdeutigkeit (Polysemie) und kodiert syntaktische/semantische Rollen.
- Beispiel: „von“ wird zur Relation zwischen „Hauptstadt“ und „Frankreich“ statt eigenständigem Inhalt.

Self-Attention

Jeder Token wird durch Self-Attention abhängig vom Kontext neu berechnet. Vor Self-Attention wird Positions-Information codiert (**Positional Encoding**), sodass die Reihenfolge der Tokens erhalten bleibt.

Die Attention-Gewichte sagen, welche Vektoren (Wörter) wichtig sind.

Es entsteht ein neuer kontextualisierter Vektor pro Token

Beispiel:

→ Der Vektor von „von“ wird neu berechnet, indem er stark auf „Paris“ und „Hauptstadt“ schaut

→ So kann das Modell lernen: „von“ schaut auf „Hauptstadt“



2 Self-Attention-Schritt Q, K, V & Gewichtung

- Berechnung von Q, K, V für jedes Token
- Aufmerksamkeit: Welche Tokens sind wichtig für welches?
- z. B. „von“ beachtet „Paris“ und „Hauptstadt“ besonders stark

„von“ bildet Q, die anderen bilden K & V

„von“ berechnet:

$$Q_{\text{von}} = W_Q \cdot \vec{x}_{\text{von}}$$

Alle Wörter (auch „von“) berechnen:

$$K_j = W_K \cdot \vec{x}_j$$

$$V_j = W_V \cdot \vec{x}_j$$

→ Dabei sind W_Q , W_K , W_V gelernte Gewichtsmatrizen.

$W_Q \in R^{d \times d}$, $W_K \in R^{d \times d}$, $W_V \in R^{d \times d}$ (je nach Modellgröße $d=64, 128, 768$, etc.).

In echten Modellen werden diese Berechnungen mehrfach parallel ausgeführt:

Multi-Head Attention berechnet mehrere Sets von Q, K, V gleichzeitig und kombiniert sie, um unterschiedliche Beziehungen gleichzeitig zu lernen.

1 Token-Eingabevektor

Token	Vektor
Paris	\vec{x}_{Paris}
ist	\vec{x}_{ist}
die	\vec{x}_{die}
Hauptstadt	$\vec{x}_{\text{Hauptstadt}}$
von	\vec{x}_{von}

3

Vergleich: Wie ähnlich ist Q_{von} mit jedem K_j ?

Für jedes Token j berechnet das Modell:

$$\text{Score}_j = \frac{Q_{\text{von}} \cdot K_j^T}{\sqrt{d}}$$

→ Das ist ein Maß für „Wie sehr passt j zu „von“?“



Einordnung

- Q, K, V sind lineare Projektionen der Token-Embeddings und definieren die Attention-Berechnung.
- Query: „Worauf soll ich achten?“ Key: „Was biete ich an?“ Value: „Welche Information liefere ich?“
- Attention-Gewichte entstehen aus Query-Key-Ähnlichkeiten; Values werden entsprechend gemischt.
- Multi-Head Attention: mehrere parallele Sichtweisen auf Kontext (verschiedene Projektionen).

Schritt 1: Eingabevektoren \vec{x}

Paris	[2, 0, 1, 1]
ist	[0, 2, 0, 1]
die	[1, 1, 0, 0]
Hauptstadt	[0, 1, 3, 1]
von (Fokus)	[1, 2, 1, 0]

^{x)} Hinweis: Diese Vektoren wurden zur Veranschaulichung frei gewählt. In einem echten Modell stammen sie aus einer trainierten Embedding-Matrix, die jedem Token einen Vektor zuordnet.

Schritt 2: Gewichtsmatrizen

$$W_Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$W_K = W_V = 0.5 I = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}$$

Schritt 3: Berechne von Q_{von}

$$Q_{\text{von}} = W_Q \cdot \vec{x}_{\text{von}} = [1, 2, 1, 0]$$

Die Query Q_{von} entsteht nicht „aus dem Nichts“, sondern durch die Gewichtsmatrix W_Q , die das Modell beim Training lernt (kommende Folien).

Das Skalarprodukt misst Ähnlichkeit zwischen dem aktuellen Token („von“) und den anderen im Vektorraum. In diesem Beispiel ergibt sich zufällig ein hoher Wert für „ist“ – aber das Modell lernt im Training, solchen grammatischen Wörtern mehr **weniger Bedeutung** zu geben, indem es die Gewichtsmatrizen W_i entsprechend anpasst.

Schritt 4: Berechne $K_j = W_K \cdot \vec{x}_j$

Token	K_j
Paris	[1.0, 0.0, 0.5, 0.5]
ist	[0.0, 1.0, 0.0, 0.5]
die	[0.5, 0.5, 0.0, 0.0]
Hauptstadt	[0.0, 0.5, 1.5, 0.5]

Die **Keys K_j** entstehen durch die Matrix W_K , die wie W_Q beim Training gelernt wird.

Diese Matrix bestimmt, wie sichtbar ein Token für andere ist (ob es Aufmerksamkeit bekommt).

Schritt 5: Berechnung der Scores

$$\text{Score}_j = \frac{Q_{\text{von}} \cdot K_j^T}{\sqrt{d}}, \text{ mit } d = 4 \Rightarrow \sqrt{4} = 2$$

Token	Skalarprodukt	Score
Paris	1.5	0.75
ist	2.0	1.00
die	1.5	0.75
Hauptstadt	2.5	1.25

Schritt 6: Softmax-Anwendung

Token	e Score
Paris	2.12
ist	2.72
die	2.12
Hauptstadt	3.49

$$\sum e^{\text{Score}} = 10.45$$

$$\text{Softmax-Gewichte } \alpha_j = \frac{e^{\text{Score}_j}}{\sum_k e^{\text{Score}_k}}$$

Token	Gewicht α_j
Paris	0.203
ist	0.260
die	0.203
Hauptstadt	0.334

Schritt 7: Berechnung $V_j = W_V \cdot \vec{x}_j$

Token	V_j
Paris	[1.0, 0.0, 0.5, 0.5]
ist	[0.0, 1.0, 0.0, 0.5]
die	[0.5, 0.5, 0.0, 0.0]
Hauptstadt	[0.0, 0.5, 1.5, 0.5]

Schritt 8: Finaler Output-Vektor von „von“

$$\begin{aligned} \text{Output} &= \sum \alpha_j \cdot V_j \\ &= 0.203 \cdot [1.0, 0.0, 0.5, 0.5] \\ &+ 0.260 \cdot [0.0, 1.0, 0.0, 0.5] \\ &+ 0.203 \cdot [0.5, 0.5, 0.0, 0.0] \\ &+ 0.334 \cdot [0.0, 0.5, 1.5, 0.5] \end{aligned}$$

Gewichtete Summe der Value-Vektoren

Auch die **Values V_j** entstehen durch eine lernbare Matrix W_V . Sie bestimmen, welche Information ein Token weitergibt, wenn es von einem anderen Token beachtet wird (Attention). Auch W_K und W_V werden durch denselben Fehler (Loss) angepasst wie W_Q .

Wenn das Modell z. B. eine falsche Attention-Verteilung lernt, fließt der Fehler auch zurück durch W_K und W_V . → So werden alle drei Matrizen gleichzeitig trainiert, um bessere Vorhersagen zu ermöglichen.



Einordnung

- Score-Berechnung als skaliertes Skalarprodukt: $\text{Score}_{i,j} = \frac{Q_i \cdot K_j}{\sqrt{d_k}}$.
- Skalierung mit $\sqrt{d_k}$ stabilisiert Wertebereiche und Gradienten bei größeren Dimensionen.
- Softmax über Scores liefert Attention-Gewichte (Summe = 1), interpretierbar als Relevanzen.
- Kontextvektor entsteht als gewichtete Summe der Value-Vektoren.

Ergebnis

[0.304, 0.529, 0.603, 0.339]

Dieser Vektor ist der neue, kontextualisierte Bedeutungsvektor für das Token „von“. Er setzt sich aus den gewichteten Beiträgen der vorherigen Tokens zusammen. Dabei hat „Hauptstadt“ den stärksten Einfluss. In einem echten Sprachmodell würde dieser Vektor weiterverwendet, um z. B. den nächsten Token vorherzusagen. Das Modell erkennt so schließlich Strukturen wie: „Hauptstadt von Paris“. Der neue Vektor fließt nun in die nächste Vorhersage ein: Was folgt auf „Hauptstadt von?“?

Ursprünglich \vec{x}_{von} :
[1, 2, 1, 0]

Nach Attention:
[0.304, 0.529, 0.603, 0.339]

Token	Gewicht α_j	Score
-----	-----	-----
Hauptstadt	(0.334)	1.25
ist	(0.260)	1.00
Paris	(0.203)	0.75
die	(0.203)	0.75

In großen Modellen wie GPT oder BERT passiert genau dieselbe Berechnung aber mit typischerweise 768-dimensionalen Vektoren, mehreren Attention-Heads gleichzeitig und mehrschichtigen Attention-Blöcken. Diese einfache Version ist also ein verständlicher Ausschnitt dessen, was große Sprachmodelle intern millionenfach anwenden.

Architektur-Vergleich: Unser Beispiel vs. GPT/BERT

Aspekt	Rechenbeispiel	GPT/BERT-Modell
Vektorgröße (x, Q, K, V)	4	768 (oder mehr)
Anzahl Attention-Heads	1	12 (BERT Base), 12–96 (GPT-3/4)
Anzahl Schichten	1	12 (BERT Base), 96+ (GPT-4)
Gewichtsmatrizen	$1 \times W_Q, W_K, W_V$	Pro Head eigene Matrizen
Berechnung	einmalig	mehrschichtig & mit Residual + LN Residual = Skip-Verbindungen (Information wird direkt weitergegeben); LN = Layer Normalization zur Stabilisierung



Einordnung

- Ergebnis der Attention: neuer Vektor pro Token, der Bedeutung im Satzkontext kodiert.
- Damit kann das Modell Relationen (Subjekt/Objekt, „von“-Beziehungen, Referenzen) rechnerisch abbilden.
- Wichtig: „Verstehen“ im Transformer ist Ergebnis von Matrixoperationen, nicht symbolischer Regeln.



Notebooks

- Open in Colab: Attention Rechnung (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/attention_rechenbeispiel.ipynb)

Next Token Vorhersage

Wie entsteht der nächste Token?

Kontextvektor nach Attention: $\mathbf{h} = [0.304, 0.529, 0.603, 0.339]$ (s. Folie 13, Ergebnisse)

Schritt 1: Lineare Projektion auf den Vokabelraum

Man verwendet eine Gewichtsmatrix \mathbf{W}_{out} , um den Vektor in einen Logit-Vektor (Scores für alle möglichen nächsten Tokens) zu transformieren:

$$\mathbf{z} = \mathbf{W}_{\text{out}} \cdot \mathbf{h} + \mathbf{b}$$

(\mathbf{b} = Bias-Vektor,
Grundwahrscheinlichkeit
für jedes Wort)

Der Vektor \mathbf{h} stammt aus der Attention-Berechnung für das aktuelle Token (z. B. „von“).

\mathbf{W}_{out} hat die Dimension:
Vokabulgroße \times Hidden-Size
also z. B. 30.000×768 in GPT.

In einem Mini-Beispiel könnten man z. B. ein hypothetisches kleines Vokabular annehmen:
Vokabular = {Frankreich, Deutschland, Spanien, ...}

Schritt 2: Berechnung der Logits (Rohwerte)

Nach der Multiplikation entstehen die sog. Logits:

$$\mathbf{Z} = [z_{\text{Frankreich}}, z_{\text{Deutschland}}, z_{\text{Spanien}}, \dots]$$

Das sind reelle Zahlen, z. B. $z_{\text{Frankreich}} = 3.2$, $z_{\text{Deutschland}} = 1.7$ etc.)

Schritt 3: Softmax über die Logits

Damit aus den Logits Wahrscheinlichkeiten werden, wird wieder Softmax angewendet:

$$p_{\text{Token}} = \frac{e^{z_{\text{Token}}}}{\sum_k e^{z_k}}$$

Jetzt entstehen Wahrscheinlichkeiten für jedes Wort im Vokabular.

- Im Training vergleicht das Modell diese Wahrscheinlichkeiten mit dem tatsächlichen nächsten Wort.
- Daraus wird ein Fehler berechnet (z. B. Cross-Entropy-Loss).
- Dieser Fehler wird rückwärts durch das Netz propagiert, um z. B. die Matrizen \mathbf{W}_{out} , \mathbf{W}_Q , \mathbf{W}_K , \mathbf{W}_V zu verbessern.



Einordnung

- Next-Token-Prediction: Aus dem Kontextvektor wird über eine lineare Projektion ein Logit pro Vokabel berechnet.
- Softmax macht daraus Wahrscheinlichkeiten; Sampling/Decoding bestimmt das tatsächlich generierte Token.
- Training treibt die richtigen Tokens hoch, indem die Loss (Cross-Entropy) minimiert wird.

Schritt 4: Beispielhafte Illustration (vereinfacht):

Nehmen wir an, das Modell würde folgende Logits berechnen:

Token	Logit z	e^z	Wahrscheinlichkeit
Frankreich	3.2	24.5	0.70
Deutschland	1.7	5.47	0.16
Spanien	1.2	3.32	0.095
...	

Ergebnis:

Das Modell wählt mit höchster Wahrscheinlichkeit den nächsten Token **Frankreich**.



Schritt 5: konkrete Berechnung

Voraussetzungen:

$$h = [0.304, 0.529, 0.603, 0.399]$$

Vokabular={Frankreich, Deutschland, Spanien}

Token	Gewichtungsvektor (Zeile in W_{out})
Frankreich	[1.0, 0.5, 1.0, 0.5]
Deutschland	[0.0, 1.0, 0.0, 1.0]
Spanien	[1.0, 0.0, 1.0, 0.0]

Rechnung, Schritt 1: Logits berechnen

$$z_{\text{Frankreich}} = 1.0 \cdot 0.304 + 0.5 \cdot 0.529 + 1.0 \cdot 0.603 + 0.5 \cdot 0.399 = 1.372$$

$$z_{\text{Deutschland}} = 0.0 + 1.0 \cdot 0.529 + 0.0 + 1.0 \cdot 0.399 = 0.928$$

$$z_{\text{Spanien}} = 1.0 \cdot 0.304 + 0.0 + 1.0 \cdot 0.603 + 0.0 = 0.907$$

Zuerst Exponentialwerte:

$$e^{1.372} \approx 3.94, e^{0.928} \approx 2.53, e^{0.907} \approx 2.48$$

Summe:

$$\sum e^z \approx 3.94 + 2.53 + 2.48 = 8.95$$

Wahrscheinlichkeiten:

$$p_{\text{Frankreich}} = 3.94/8.95 = 0.44$$

$$p_{\text{Deutschland}} = 2.53/8.95 = 0.28$$

$$p_{\text{Spanien}} = 2.48/8.95 = 0.28$$

Token	Logit	Softmax-Wahrscheinlichkeit
Frankreich	1.372	44% (höchste)
Deutschland	0.928	28%
Spanien	0.907	28%



Einordnung

- Mini-Vokabular macht Softmax konkret: Logits werden exponentiiert und normiert.
- Kleine Logit-Änderungen können Wahrscheinlichkeiten stark verändern (Sensitivity).
- Decoding-Strategien (Greedy, Temperature, Top-k/Top-p) steuern Vielfalt und Determinismus.



Notebooks

- Open in Colab: Next Token Vorhersage 1 (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/next_token.ipynb)
- Open in Colab: Next Token Vorhersage 2 (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/next_token_prediction_toy.ipynb)



Hinweis

- LLMs wählen das wahrscheinlichste nächste Token, nicht „die Wahrheit“.

Self-Attention: Beispiel von „von“ als aktives Token

Fokus auf Token: „von“ → erzeugt Query (Q)

Token	Rolle im Satz	Q / K / V Funktion
von	aktives Token	Q: „Ich möchte wissen, worauf ich mich beziehe.“
Hauptstadt	möglicher Bezugspunkt	K: „Ich bin ein Ort“ V: „Kontext: Stadt, Hauptstadtfunktion.“
Paris	konkreter geografischer Begriff	K: „Ich bin ein Eigenname.“ V: „Kontext: Stadtname, Geografie.“
ist / die	Hilfswörter, grammatisch	K/V: gering relevant → kaum Bezug zu „von“



Ziel:

„von“ vergleicht seine Query mit allen Keys, gewichtet die Values und erhält dadurch kontextuelles Verständnis:

→ „Ich gehöre zur Phrase „Hauptstadt von ...““

Modell lernt:

„von“ bekommt hohe Attention auf „Hauptstadt“, da „Hauptstadt von ...“ eine typische Struktur ist.

Metapher:

Jeder Mensch in einem Raum ist ein Token.
Jeder trägt ein T-Shirt mit seiner Bedeutung (z. B. „Paris“, „ist“, „die“, „Hauptstadt“, „von“).
Ein Mensch (z. B. „von“) schaut sich die anderen an und denkt nach:

- „Mit wem habe ich am meisten zu tun?“
→ Das ist die Query Q von „von“
- Die anderen tragen Keys K und Values V

Element	Metapher im Raum
Token „von“	Eine Person mit „von“-Shirt, die ihre Aufmerksamkeit verteilt
Q von „von“	„Worauf beziehe ich mich?“
K von anderen	„Was bietet ich an Kontext an?“ (z. B. „Hauptstadt“)
V von anderen	„Welchen Kontext bringe ich mit?“ (z. B. „Ich bin eine Stadt“)
Score(Q,K)	Wie gut passt das Angebot zum Bedürfnis von „von“?
Attention-Output	Mischung der passenden Values → neuer Kontext für „von“



Einordnung

- Attention-Gewichte sind eine Verteilung darüber, wie stark ein Token andere Tokens berücksichtigt.
- Visualisierung hilft, syntaktische/semantische Bezüge zu sehen (z.B. Subjekt-Verb, Referenzen).
- Wichtig: Hohe Gewichte bedeuten Relevanz im Modell, aber nicht zwingend „Erklärbarkeit“ im menschlichen Sinn.

1.5 Lernprozess und Backpropagation

Lernprozess & Backpropagation: Wie lernt das Modell im Training?

Wie werden die Gewichtsmatrizen gelernt?

Beispiel: Das Modell soll lernen, dass „Frankreich“ der richtige nächste Token ist und diesen vorhersagen.

Die Gewichtsmatrizen W_Q , W_K , W_V werden nicht manuell vorgegeben sondern sie werden vom Modell trainiert.

Die Matrix W_{out} , die den Kontextvektor in Logits für die Token-Vorhersage überführt, wird ebenfalls über Backpropagation angepasst.

Trainingsprozess mit Gradient Descent:

Der Text „Paris ist die Hauptstadt von Frankreich“ wird eingegeben

Das Modell berechnet z. B. für das Token „von“ einen Vektor \vec{z}_{von}

Dieser Vektor geht in die Vorhersage: Was kommt als nächstes?

Das Modell gibt z. B. „Deutschland“ aus → falsch. Das Modell berechnet die Wahrscheinlichkeit für alle möglichen nächsten Tokens.

Das Ziel ist, die Wahrscheinlichkeit für den korrekten nächsten Token (hier: "Frankreich") zu maximieren.

Der Fehler (Loss) wird berechnet.

Durch Backpropagation wird der Gradientenfluss berechnet.

Die Matrizen werden mit Gradient-Descent angepasst:

$$W_Q^{\text{new}} \leftarrow W_Q - \eta \cdot \frac{\partial \text{Loss}}{\partial W_Q}$$

(analog für W_K und W_V)

So lernt das Modell, welche Gewichtungen zu guten Vorhersagen führen. Beim späteren Einsatz werden diese Gewichtungen nur noch angewendet.

Alle vier Matrizen sind lernbare Parameter und werden durch **Backpropagation** angepasst.

→ Während W_Q , W_K , W_V den Kontext berechnen, wandelt W_{out} diesen in eine **Token-Vorhersage** um.

Matrix	Symbol	Lernfunktion	Beispielhafte Frage, die das Modell stellt
Query-Matrix	W_Q	Lernt, wie gefragt wird	„Worauf will ich achten?“
Key-Matrix	W_K	Lernt, wem zugehört wird	„Biete ich relevante Information an?“
Value-Matrix	W_V	Lernt, was weitergegeben wird	„Was gebe ich weiter, wenn man auf mich schaut?“
Output-Matrix	W_{out}	Lernt, wie der Vektor in Token-Scores übersetzt wird (→ Logits)	„Wie wahrscheinlich ist jedes mögliche Wort?“



Einordnung

- Kontextvektor ist die gewichtete Summe der Values: Information wird „eingesammelt“ und verdichtet.
- Ergebnis pro Token kann als „kontextualisiertes Embedding“ verstanden werden.
- Dieser Schritt passiert pro Layer mehrfach und baut schrittweise abstraktere Repräsentationen.

Schritt 1: Loss bestimmen (Ausgangswerte stark vereinfacht)

Eingabevektor $\vec{x}_{\text{von}} = [1, 2, 1, 0]$

Wie entsteht W_Q ?

$$W_Q = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} \text{Initial, zur} \\ \text{Vereinfachung:} \\ \text{Einheitsmatrix I} \end{matrix}$$

$$\text{Loss} = \frac{1}{2}(z - \tilde{z})^2$$

Wie werden die Gewichtsmatrizen gelernt?
→ Gradientenabstieg

Hierbei:

z = tatsächlicher Zielwert (z. B. perfekte Bedeutungskomponente)

\tilde{z} = aktuelle Modell-Ausgabe (z. B. erste Komponente unseres Output-Vektors)

Beispiel:

Gewünschter Zielwert: $z = 0.5$

Aktueller Output (erste Komponente):

$$\tilde{z} = W_Q \cdot x_{\text{von},1} = 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 1 + 1 \cdot 0 = 4.0$$

Damit:

$$\text{Loss} = \frac{1}{2}(4.0 - 0.5)^2 = 6.125$$

Schritt 2: Gradientenberechnung

Ziel: Durch Gradientenabstieg die Gewichte so anpassen, dass \tilde{z} näher an den Zielwert 0.5 kommt.

$$\tilde{z} = W_Q \cdot x_{\text{von},1}$$

$$W_Q = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

$$\frac{\partial \text{Loss}}{\partial w_{11}} = (z - \tilde{z}) \cdot x_{\text{von},1}$$

Einsetzen:

$$\tilde{z} - z = 4 - 0.5 = 3.5$$

$$x_{\text{von},1} = 1$$

Ergibt:

$$\frac{\partial \text{Loss}}{\partial w_{11}} = 3.5 \cdot 1 = 3.5$$

$$\text{Loss} = \frac{1}{2}(4 - 0.5)^2 = 6.125 \text{ (Fehler)}$$

Gradientenabstieg

Lernrate: $\eta = 0.1$

Update-Regel:

$$w_{11}^{\text{neu}} = w_{11} - \eta \cdot \frac{\partial \text{Loss}}{\partial w_{11}} = 1 - 0.1 \cdot (3.5) = 0.65$$

$$\begin{matrix} w_{11} & \text{beeinflusst } x_{\text{von},1} \\ w_{21} & \text{beeinflusst } x_{\text{von},2} \\ w_{31} & \text{beeinflusst } x_{\text{von},3} \\ w_{41} & \text{beeinflusst } x_{\text{von},4} \end{matrix}$$

$$0.65 \quad 0 \quad 0 \quad 0$$

$$\text{Neues } W_Q \text{ (nur erstes Element verändert)} = \begin{matrix} 0.65 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$



Einordnung

- Vollständiges Rechenbeispiel verbindet: Projektionen $Q, K, V \rightarrow$ Scores \rightarrow Softmax \rightarrow Kontextvektor.
- Zeigt konkret, wo Multiplikationen, Normalisierung und Summen im Transformer auftreten.
- Gute Stelle, um Dimensionsprüfung und Intuition („woher kommt welche Information?“) zu üben.

Gradientenmatrix im Detail:

Kettenregel: $\frac{\partial \text{Loss}}{\partial W_Q} = \frac{\partial \text{Loss}}{\partial \tilde{z}} \cdot \frac{\partial \tilde{z}}{\partial W_Q}$ (Loss hängt von \tilde{z} und \tilde{z} von W_Q → Kettenregel)

$\sum \alpha_j \cdot \vec{v}_j$ (Die Value-Vektoren hängen am Ende über $Q_{\text{von}} = W_Q \cdot x^*$ indirekt von W_Q ab)

Schritt 1:

$$\frac{\partial \text{Loss}}{\partial \tilde{z}}$$

Wir nehmen wieder:

$$\text{Loss} = \frac{1}{2}(z - \tilde{z})^2, \quad \frac{\partial \text{Loss}}{\partial \tilde{z}} = \tilde{z} - z$$

Angenommen:

$$\tilde{z} = 4.0, z = 0.5$$

Dann:

$$\frac{\partial \text{Loss}}{\partial \tilde{z}} = 4.0 - 0.5 = 3.5$$

(Wie verändert sich der Fehler, wenn ich an W_Q drehe?)

→ Wenn \tilde{z} größer, wäre der Fehler kleiner

Schritt 2:

$$\frac{\partial \tilde{z}}{\partial W_Q}$$

(Wie stark ändert sich der Output z^* , wenn man W_Q verändert?)

$$Q_{\text{von}} = W_Q \cdot x^*$$

Somit hängt \tilde{z} von W_Q ab über x^*

von direkt ab. Und der Gradient der Matrix-Multiplikation ist:

$$\frac{\partial Q_{\text{von},i}}{\partial W_Q} = \text{jeweils die } i\text{-te Zeile: } x_{\text{von},i}$$

Da W_Q eine Diagonalmatrix ist, können wir sagen:

$$\frac{\partial \tilde{z}}{\partial w_{kk}} = \alpha_k \cdot V_{k,1} \cdot x_{\text{von},k}$$

Da wir aber — um es für deine Folie vereinfacht darzustellen — am Anfang nur den direkten Einfluss nehmen, setzen wir:

$$\frac{\partial \tilde{z}}{\partial w_{kk}} \approx x_{\text{von},k}$$

Das heißt (direkter Einfluss auf jede Diagonalkomponente von W_Q):

Komponente $x_{\text{von},k}$

$$w_{11} \quad 1$$

$$w_{22} \quad 2$$

$$w_{33} \quad 1$$

$$w_{44} \quad 0$$



Einordnung

- Zusammenfassung der Attention-Schritte als wiederverwendbares „Rezept“.
- Betonung der Kernidee: Relevanzgewichtung statt fester Regeln.
- Brücke zu Multi-Head/mehrere Layer: derselbe Mechanismus wird wiederholt und kombiniert.

Schritt 3:

$$\frac{\partial \text{Loss}}{\partial W_Q}$$

Nach Kettenregel: $\frac{\partial \text{Loss}}{\partial W_Q} = \frac{\partial \text{Loss}}{\partial z} \cdot \frac{\partial z}{\partial W_Q}$

Die Ableitung des Loss nach einem Gewicht hängt davon ab, wie sich z durch dieses Gewicht verändert.

Einsetzen:

Komponente	Berechnung	Ergebnis
w_{11}	$3.5 \cdot 1$	3.5
w_{22}	$3.5 \cdot 2$	7.0
w_{33}	$3.5 \cdot 1$	3.5
w_{44}	$3.5 \cdot 0$	0

→ Interpretation:

Je größer x_i desto stärker der Gradient (mehr Einfluss auf Ergebnis)

Wenn man w_{22} leicht erhöht, steigt z und der Fehler verkleinert sich, da man näher an $z=0.5$ herankommt.

Schritt 4: Update mit Lernrate

Lernrate $\eta=0.1$

Update-Regel:

$$w_{11}^{neu} = w_{11} - \eta \cdot \frac{\partial \text{Loss}}{\partial w_{11}}$$

Lernrate $\eta=0.1$. Sie bestimmt, wie stark das Modell reagiert. Höhere Werte = schneller, aber instabiler.

Komponente	Berechnung	Ergebnis
w_{11}	$1 - 0.1 \cdot 3.5$	0.65 (verringert)
w_{22}	$1 - 0.1 \cdot 7.0$	0.3 (noch stärker verringert)
w_{33}	$1 - 0.1 \cdot 3.5$	0.65 (verringert)
w_{44}	$1 - 0.1 \cdot 0$	1.0 (unverändert)

Neue W_Q^*

$$W_Q = \begin{matrix} 0.65 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0.65 & 0 \\ 0 & 0 & 0 & 1.0 \end{matrix}$$

Das Modell passt sich in Richtung des Zielwerts $z=0.5$ an.

Man sieht, wie der Fehler auf alle Gewichtsmatrix-Einträge wirkt.

Je stärker die Komponente im Eingabevektor war, desto größer der Gradient.

*) Zur Vereinfachung Diagonalmatrix und Gradient nur für Diagonalelemente berechnet. In der Realität eine volle Matrix.

Wie entsteht das Model?



Einordnung

- Softmax erzeugt Wahrscheinlichkeiten; Cross-Entropy misst, wie gut die Verteilung zum Ziel-Token passt.
- Backpropagation liefert Gradienten für alle Gewichte (inkl. Projektionen in Attention und Output-Layer).
- Training ist systematisches Anpassen der Gewichte, um die Loss über viele Beispiele zu senken.

Schritt 5: Neuer Iterationsschritt*)

Aus letzten Schritt W_Q

$$W_Q = \begin{matrix} 0.65 & 0 & 0 & 0 \\ 0 & 0.30 & 0 & 0 \\ 0 & 0 & 0.65 & 0 \\ 0 & 0 & 0 & 1.00 \end{matrix}$$

Denselben Eingabevektor:

$$\vec{x} \text{ von } [1, 2, 1, 0]$$

Schritt 1: Neuen Output \tilde{z} berechnen

Da $\tilde{z} = W_Q \cdot \vec{x}$ von (nur diagonal!), rechnet man:

(Es wird nur eine Komponente \tilde{z} betrachtet.)

$$\tilde{z} = 0.65 \cdot 1 + 0.3 \cdot 2 + 0.65 \cdot 1 + 1.0 \cdot 0 = 0.65 + 0.6 + 0.65 + 0 = 1.90$$

\tilde{z} ist jetzt 1.90 (vor dem Update 4.0). Das Netz scheint in die richtige Richtung zu lernen.

Schritt 2: Loss berechnen

Ziel $z=0.5$

$$\text{Loss} = 1/2(z - \tilde{z})^2 = 1/2(0.5 - 1.90)^2 = 1/2(-1.40)^2 = 0.98$$

Der Fehler ist kleiner geworden (vor dem Update 6.125).

Schritt 3: Gradienten berechnen

$$\partial \text{Loss} / \partial \tilde{z} = \tilde{z} - z = 1.90 - 0.5 = 1.4$$

Komponente	Rechnung	Ergebnis
w11	1.4 · 1	1.4
w22	1.4 · 2	2.8
w33	1.4 · 1	1.4
w44	1.4 · 0	0

Schritt 6: Neue Gewichtsmatrix W_Q berechnen mit Lernrate $\eta=0.1$

$$W^{\text{neu}} = W - \eta \cdot \text{Gradient}$$

Komponente	Rechnung	Ergebnis
w11	0.65 - 0.1 · 1.4	0.51
w22	0.30 - 0.1 · 2.8	0.02
w33	0.65 - 0.1 · 1.4	0.51
w44	1.00 - 0.1 · 0.0	1.00

$$W_Q = \begin{matrix} 0.51 & 0 & 0 & 0 \\ 0 & 0.02 & 0 & 0 \\ 0 & 0 & 0.51 & 0 \\ 0 & 0 & 0 & 1.00 \end{matrix}$$

Interpretation:

Das Modell hat im 1. Update gelernt, dass der Output zu hoch war ($\tilde{z} = 1.9$). Es reduziert die Gewichte leicht, besonders dort, wo der Einfluss am stärksten war (z. B. w_{22}).

Dadurch nähert sich der Output dem Ziel $z=0.5$ und der Fehler wird kleiner.

→ So funktioniert Gradient Descent: **größerer Einfluss = stärkeres Update.**

*) Hinweis: Zur Vereinfachung wird hier nur der direkte Einfluss der geänderten W_Q -Matrix auf den Query-Vektor gezeigt, um den Effekt der Lernrate zu demonstrieren. In einem echten Durchlauf müsste der gesamte Attention-Output (Schritte 3-8, Folie 12) neu berechnet werden, was zu einem anderen, aber konzeptionell ähnlichen Lerneffekt führen würde.



Einordnung

- Lernprozess als Klassifikation: Nächstes Token ist eine Klasse im Vokabular.
- Ziel ist nicht „Wahrheit“, sondern möglichst hohe Wahrscheinlichkeit für das korrekte Fortsetzungstoken im Trainingskorpus.
- Verbindet probabilistische Sicht (Verteilungen) mit Optimierung (Loss-Minimierung).

Weiter Schritte 5: Das Netz lernt durch Fehlerkorrektur

Iteration	Output \bar{z}	Fehler	w11	w22	w33	w44
0	4.00	3.50	1.000	1.000	1.000	1.00
1	1.90	1.40	0.650	0.300	0.650	1.00
2	1.060	0.56	0.510	0.020	0.510	1.00
3	0.724	0.224	0.454	-0.092	0.454	1.00
4	0.590	0.90	0.432	-0.137	0.432	1.00
5	0.54	0.04	0.42	0.15	0.42	1.00

Beobachtungen:

1. Stabile Konvergenz: Das System nähert sich stetig dem Ziel 0.5
2. w_{22} wird negativ: Das System lernt, die starke Eingabe "2" zu kompensieren
3. w_{44} bleibt unverändert: Da die Eingabe dort 0 ist, gibt es keinen Gradienten
4. Symmetrie: w_{11} und w_{33} entwickeln sich identisch (gleiche Eingabe "1")

Die Einheitsmatrix ist ein "neutralerer" Startwert als die zufälligen Gewichte aus dem ersten Beispiel

Ergebnis: Das System hat erkannt, dass bei diesem Kontext der Token „Frankreich“ wahrscheinlicher ist. Es soll also auf „Frankreich“ achten.



Einordnung

- Vertiefung: Iteratives Update der Gewichte über viele Schritte und Mini-Batches.
- Unterschied Train/Inference: During training wird das korrekte Token „vorgegeben“ (Teacher Forcing).
- Generalisierung: Modell lernt Muster, die auf neue Texte übertragen werden sollen.



Notebooks

- Open in Colab: Gradientenabstieg Iteration (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/gradient_descent_wq_diagonal.ipynb)

Kontextvektor & Weiterverarbeitung

Nach dem Lernprozess (z. B. Schritt 3–5) wird der gelernte Kontextvektor weiterverarbeitet, z. B. durch Feedforward-Netze und LayerNorm. Diese Schritte wurden im Lernbeispiel zur Vereinfachung weggelassen.

Die Zwischenschritte LayerNorm, Feedforward etc. dienen dazu, das Modell robuster und leistungsfähiger zu machen, sind aber für das einfache Lernbeispiel nicht entscheidend.

→ In diesem Beispiel betrachten man nur den Self-Attention-Schritt. In GPT/BERT folgen danach noch weitere Verarbeitungsschritte wie Feedforward-Blöcke (MLP), Residual Connections und Layer Normalization, die hier zur Vereinfachung ausgelassen wurden.



Einordnung

- Weitere Aspekte des Lernprozesses: Regularisierung, Overfitting, Trainingsdatenqualität.
- Praktische Stellschrauben: Batchgröße, Learning-Rate-Schedule, Weight Decay.
- Intuition: Gute Performance entsteht aus Daten, Architektur und Training zusammen (nicht nur „mehr Parameter“).

Wie lernt das Modell?

$$\partial L / \partial W = (p - y) \cdot x^T$$

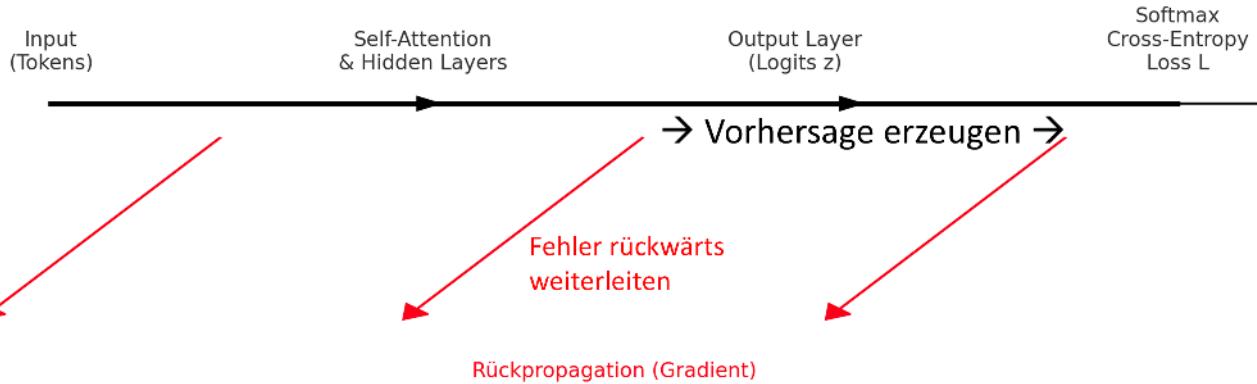
$$W_{\text{neu}} = W_{\text{alt}} - \eta \cdot \partial L / \partial W$$

Backpropagation: Der berechnete Gradient wird durch das Netzwerk zurückgerechnet und aktualisiert die Gewichte entlang aller Schichten.

(Rückpropagation: Der Fehler wird durch das Netz zurückgeleitet)

→ Wiederholung über Millionen Datensätze

Wie lernt das Modell? - Vorwärts- und Rückwärtsdurchlauf



Einordnung

- Zusammenfassung: Optimierung minimiert Loss; Gradienten sagen, wie Parameter angepasst werden.
- Blick auf die Praxis: Pretraining vs. Finetuning und warum Datenmengen entscheidend sind.
- Übergang zu Architekturfragen: Was genau wird in welchen Schichten gelernt?

Wie entscheidet das Modell: Decoder-Vorhersage

Beim vorherigen Lernen von W_Q lernte das Modell, wie es aus dem Eingabekontext eine sinnvolle Repräsentation erzeugt (\tilde{z}). Jetzt lernt es, wie es diese Repräsentation verwendet, um den richtigen nächsten Token (z. B. „Frankreich“) vorherzusagen, mithilfe von Softmax, Cross-Entropy und einem weiteren Gewichtsmatrix-Update.

Vektor → Logits (Scores) über alle Wörter im Vokabular (z. B. 50.000)

Softmax → Wahrscheinlichkeiten über das Vokabular (die rohen Scores selbst sind noch keine Wahrscheinlichkeiten)

$$\text{Formel } p_{\text{Token}} = \frac{e^{\text{Score}_{\text{Token}}}}{\sum_k e^{\text{Score}_k}}$$

Höchste Wahrscheinlichkeit → „Frankreich“

Optional: Sampling, Top-k für kreative Textgeneratoren (ChatGPT etc.)

Vektor → Logits (Scores) → Softmax → Wahrscheinlichkeiten → höchstwahrscheinlicher Token --ID→
Tokenizer → Wort

Rückübersetzung des Tokens:

Höchste Wahrscheinlichkeit → z. B. Token-ID 12856

Tokenizer-Umkehrung:

12856 → "Frankreich"



Einordnung

- Gesamtüberblick verbindet: Tokenisierung → Repräsentationen → Attention → Output → Loss → Update.
- Nützlich als mentale Landkarte: Wo greifen Demos/Rechenbeispiele in die Pipeline ein?
- Vorbereitung für Modellvergleich und Bausteine des Transformers (Residuals, LayerNorm, FFN).

Beispiel: Softmax & Cross-Entropy (Update für W_{out})

Das Beispiel zeigt, wie das Modell durch Training an W_{out} lernt, welche Tokens es mit welcher Wahrscheinlichkeit vorhersagen soll.
Ziel: das Modell soll lernen, welches Wort mit welchem Vektor verknüpft ist.

$p = \text{Softmax}(z)$ ($p \rightarrow$ Wahrscheinlichkeitsverteilung von Vokabular)

Cross-Entropy Loss: $-\log(p_y)$ ($y \rightarrow$ One-Hot-Zielvektor: richtiges Wort 1, sonst 0)

Cross-Entropy → misst Abstand zwischen Ziel y und Vorhersage p

Negativer Log-Wert der Wahrscheinlichkeiten für das Ziel-Token k .

Gradient bzgl. z : $\frac{\partial L}{\partial z} = p - y$

Ableitung des Loss nach den Logits

Gradienten bzgl. W : $\frac{\partial L}{\partial W} = (p - y) \cdot x^T$

Ableitung der Gewichtsmatrix W

Je höher p_{richtig} desto kleiner der Loss

→ das ist die zentrale Lernidee.

Fehler (Cross-Entropy):

$$L = -\log(p_{\text{zielklasse}}) = -\log(0.6590) \approx 0.417$$

→ Je wahrscheinlicher das Zielwort ist, desto kleiner ist der Fehler (Logarithmus)

Nächste Iteration:

Neuer Wert: $p_{\text{Frankreich}} = 0.85 \rightarrow L \approx 0.163 \rightarrow$ Fehler wurde kleiner

Klasse 1 (Tokens): „Frankreich“, Klasse 2: „Deutschland“, Klasse 3: „Spanien“ (tatsächlich z.B. 50.000 Wörter)

Zielwort „Frankreich“ $y = [1, 0, 0]$

$$z = [2.0, 1.0, 0.1]$$

Logits: Ausgaben nach der letzten linearen Schicht des Modells vor der Softmax

Bsp. 1

$$z = [2.0, 1.0, 0.1]$$

$$\text{Softmax}(p) \approx [0.6590, 0.2424, 0.0986]$$

$$y = [1, 0, 0]$$

$$p - y = [-0.3410, 0.2424, 0.0986]$$

$$\frac{\partial L}{\partial z} \approx [-0.3410, 0.2424, 0.0986]$$

$$x = [1.5, 0.5] \text{ (Ergebnis von Attention)}$$

$$\frac{\partial L}{\partial W} = (p - y) \cdot x^T = \begin{pmatrix} -0.5115 & -0.1705 \\ 0.3639 & 0.1212 \\ 0.1479 & 0.0493 \end{pmatrix}$$

Klasse	$p - y$	Gradientzeile $(p - y) \cdot x$	Bedeutung
Klasse 1	-0.3410	[-0.5115, -0.1705]	Modell war zu sicher für Klasse 1 (war ja das Ziel), daher negatives Update
Klasse 2	+0.2424	[+0.3636, +0.1212]	Modell war hier zu unsicher, daher positive Korrektur
Klasse 3	+0.0986	[+0.1479, +0.0493]	Modell hat hier auch leicht zu wenig Wahrscheinlichkeit gegeben

Bsp. 2

$$z = [3.52, 2.37, 1.185, 0.07]$$

$$p = [0.692, 0.219, 0.066, 0.022]$$

$$y = [1, 0, 0, 0]$$

$$p - y = [-0.308, 0.219, 0.066, 0.022]$$

$$\frac{\partial L}{\partial z} \approx [-0.308, 0.219, 0.066, 0.022]$$

$$x = [1.2, 0.8, 0.5]$$

$$\frac{\partial L}{\partial W} = \begin{pmatrix} -0.3696 & -0.2464 & -0.154 \\ 0.2628 & 0.1752 & 0.1095 \\ 0.792 & 0.0528 & 0.033 \\ 0.0264 & 0.0176 & 0.011 \end{pmatrix}$$

Ziel: Der Fehler $L = -\log(p_{\text{Frankreich}}) \approx 0.417$ soll durch Anpassung von W_{out} minimiert werden.

Klasse 1 (Tokens): „Frankreich“, Klasse 2: „Deutschland“, Klasse 3: „Spanien“, Klasse 4: „Italien“

Zielwort „Frankreich“ $y = [1, 0, 0, 0]$

Klasse	$p - y$	Gradientzeile $(p - y) \cdot x$	Bedeutung
Klasse 1	-0.308	[-0.3696, -0.2464, -0.154]	Zielklasse: Modell war gut, aber noch nicht negative Korrektur
Klasse 2	+0.219	[+0.2628, +0.1752, +0.1095]	Wahrscheinlichkeit für Klasse 2 erhöhen
Klasse 3	+0.066	[+0.792, +0.0528, +0.033]	Wahrscheinlichkeit für Klasse 3 leicht erhöhen
Klasse 4	+0.022	[+0.0264, +0.0176, +0.0110]	Wahrscheinlichkeit für Klasse 4 minimal erhöhen



Einordnung

- Softmax & Cross-Entropy als zentrales Duo: Wahrscheinlichkeiten erzeugen und Fehler messen.
- Interpretation: Cross-Entropy bestraft „selbstsichere“ falsche Vorhersagen besonders stark.
- Verbindet mathematische Formel mit praktischem Training: Loss dient als Optimierungsziel.



Notebooks

- Open in Colab: Softmax & Cross-Entropy (https://colab.research.google.com/github/karkessler/llm/blob/main/notebooks/decoder_softmax_crossentropy_wout.ipynb)

Training vs. Inferenz – Was passiert wann?

- **Training (offline, teuer)**
 - Millionen Texte
 - Next-Token-Prediction
 - Loss & Backpropagation
 - Gewichte werden verändert
 - **Inferenz (online, schnell)**
 - Nur Vorwärtsrechnung
 - Kein Lernen
 - Wahrscheinlichkeitsbasierte Auswahl
- Beim Antworten lernt das Modell nicht, es rechnet nur.
→ [TRAINING] —————► [FERTIGES MODELL] —————► [INFERENZ]

Bis hierhin wissen wir: Beim Antworten lernt das Modell nicht.
Wenn wir wollen, dass es trotzdem aktuelles oder spezifisches
Wissen nutzt, müssen wir dieses Wissen vor der Modellrechnung einspeisen.
Genau das macht RAG.



Einordnung

- Vergleich GPT vs. BERT: Decoder-only generiert autoregressiv, Encoder-only lernt bidirektionale Repräsentationen.
- Unterschiedliche Pretraining-Objectives (Causal LM vs. Masked LM) führen zu unterschiedlichen Stärken.
- Praktische Konsequenz: Wahl der Architektur hängt von Aufgabe ab (Generierung vs. Klassifikation/Extraktion).

Retrieval Augmented Generation (RAG)

Kernaussagen:

- Teil der Inferenz (kein Training)
- LLM-Gewichte bleiben unverändert
- Externes Wissen wird als Kontext ergänzt
- Antwort = Sprachmodell + Kontext

→ Merksatz:

RAG verändert nicht das Modell – sondern den Kontext.

Man hat essehen: Beim Antworten lernt das Modell nicht.

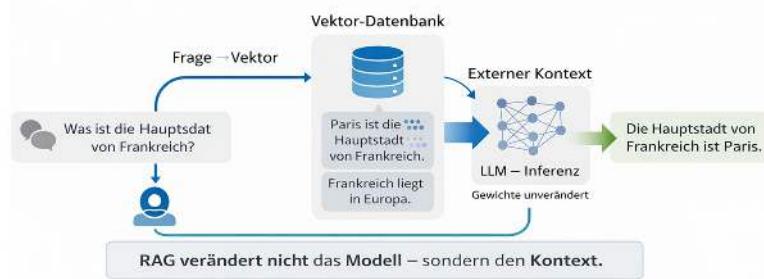
RAG ist deshalb kein Training, sondern ein vorgeschalteter Schritt.

Relevante Texte werden gesucht und als Kontext übergeben –

das Modell formuliert daraus die Antwort.

Die Gewichte bleiben unverändert.

Thema	Erklärung
Training	Gewichte ändern sich
Inferenz	Gewichte fix
RAG	Kontext wird ergänzt
Vektor-DB	Wissensquelle



Einordnung

- Feed-Forward-Netz (FFN) erweitert die Kapazität pro Token nach der Attention (nichtlineare Transformation).
- Residuals stabilisieren Training tiefer Netze und erleichtern Gradientenfluss.
- LayerNorm normalisiert Aktivierungen und verbessert Trainingsstabilität (insb. bei großen Modellen).

Modellarchitekturen im Überblick

Modellarchitektur

Einbettung der Rechenschritte:

Token → Embedding (Vektorraum) → Attention (Self-Attention & Transformer-Blöcke) → Lineare Projektion
 $(z = W_{\text{out}} \cdot h + b)$ → Softmax (p) → Loss (Cross-Entropy)

Wichtige Bestandteile:

Embedding-Schicht: Token werden in

Vektoren (z. B. 768 Dimensionen) umgewandelt.

Multi-Head Self-Attention: Tokens schauen aufeinander → Berechnung der Kontexte.

Feedforward-Schichten (MLP): Weiterverarbeitung nach Attention.

Lineare Projektion (Output Layer): Berechnung der Logits (z).

Softmax & Cross-Entropy: Umwandlung in Wahrscheinlichkeiten und Berechnung des Loss.

→ Softmax berechnet Wahrscheinlichkeiten p :

→ Cross-Entropy misst Abstand zur Zielwahrscheinlichkeit.

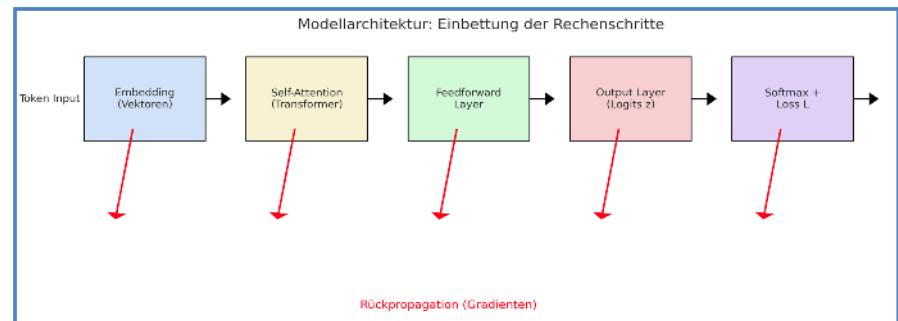
Backpropagation & Gradientenabstieg: Optimierung der Gewichtsmatrizen ($W_Q, W_K, W_V, W_{\text{out}}$).

→ Wiederholung über viele Layer

(z. B. GPT-3: 96 Transformer Layer)

→ Training auf Milliarden Beispielen

Architekturtyp	Beispiel	Einsatz	Attention
Encoder-only	BERT	Klassifikation, NER	bidirektional
Decoder-only	GPT	Textgenerierung	Autoregressiv
Encoder–Decoder	T5, BART	Übersetzung, Summarization	Encoder: bidi Decoder: autoregressiv



Modelarchitektur



Einordnung

- Vertiefung Modellarchitekturen: Bausteine wiederholen sich pro Layer (Attention + FFN + Norm + Residual).
- Skalierungsgesetze (grobe Intuition): Mehr Daten/Parameter/Compute führen oft zu besseren Modellen.
- Architekturentscheidungen beeinflussen Latenz, Speicherbedarf und Kontextlänge.

Interpretation & Kontrolle

Softmax-Summe = 1, es werden folglich Wahrscheinlichkeitsverteilungen erzeugt.

Der Gradient zeigt, wie die Gewichte geändert werden müssen, um den Fehler zu verringern.

Die Jacobi-Matrix $J_{ij} = \partial z_j / \partial p_i$ beschreibt, wie stark kleine Änderungen der Eingabe z jeden einzelnen Softmax-Ausgabewert p_i beeinflussen (Sensitivität aller Klassen auf Logit-Änderungen).

LLMs sind Wahrscheinlichkeits-Vervollständiger. Sie schätzen Wahrscheinlichkeiten für sinnvolle nächste Tokens (Vervollständigung von Texten).

Begriff	Was ist es?	Wofür verwenden wir es?	Beispiel
Gradient	Vektor der partiellen Ableitungen einer skalaren Funktion	Für Funktionen $L(z)$, also z. B. den Loss	Nabla L
Jacobi-Matrix	Matrix aller partiellen Ableitungen einer Vektorwertigen Funktion	Wenn eine Funktion mehrere Ausgaben hat, z. B. Softmax $p = f(z)$	J

Gradient = Richtung der Fehlerkorrektur
Jacobian = Sensitivität der Ausgaben auf Eingaben



Einordnung

- Zusammenfassung der Architektur-Bausteine und ihrer Rollen (Aufmerksamkeit, Nichtlinearität, Stabilisierung).
- Mentales Modell: Attention mischt Information über Positionen, FFN transformiert pro Position.
- Brücke zu neueren Varianten: effiziente Attention, längere Kontexte, Speichermechanismen.

Biologisches Neuron vs. Künstliches Neuronales Netz

Biologisches Gehirn (Neuron)	Künstliches neuronales Netz (LLM)
Neuronen (Zellen)	Vektoren
Synapsen (Verbindungen, Verstärkung/Abschwächung)	Gewichtsmatrizen (z. B. W_Q , W_K , W_V)
Signalweitergabe per Neurotransmitter	Matrix-Multiplikation & Aktivierungsfunktionen (Attention), GeLU (Feedforward)
Lernen durch Plastizität (z. B. synaptische Verstärkung)	Gradientenabstieg (Backpropagation)
Sehr energieeffizient, biologisch flexibel	Hoher Rechenaufwand, hochparallel auf GPUs

Was ist die Mathematik dahinter?

Vektoren, Matrizen, Wahrscheinlichkeiten = Grundbausteine
Matrixmultiplikation + Aktivierungsfunktionen erzeugen „Verstehen“
Softmax erzeugt Wahrscheinlichkeiten
Cross-Entropy misst, wie falsch das Modell liegt
→ Das macht Lernen durch Backpropagation möglich



Einordnung

- Gesamtüberblick der Modellarchitekturen: Einordnung verschiedener Familien und Trade-offs.
 - Wichtige Achsen: Kontextlänge, Compute pro Token, Trainingsobjective, Einsatzgebiet.
 - Vorbereitung für den Ausblick auf Titans/Memory-Konzepte.
-

Was fehlt in dem Rechenbeispiel?

Baustein	Unser Beispiel	GPT / BERT
Tokenization	✓	✓
Embedding	✓	✓
Self-Attention	✓	✓
Feedforward-Block (MLP mit GeLU)	✗ (weggelassen für Einfachheit)	✓
Residual Connections	✗	✓
LayerNorm	✗	✓



Einordnung

- Titans als Ausblick: Erweiterung klassischer Transformer um explizite Memory-Mechanismen.
- Motivation: Grenzen durch Kontextfenster, Langzeitkonsistenz und effiziente Speicherung von Wissen.
- „Memory“ als zusätzliche Komponente neben Parametern und Kontext (Prompt).

Was im Rechenbeispiel vereinfacht wurde

Vergleich: Rechenbeispiel vs. GPT/BERT

Baustein	Beispiel	GPT/BERT
Tokenisierung	✓	✓
Embedding	✓	✓
Self-Attention	✓	✓
Feedforward (MLP)	✗ (weggelassen, z. B. GeLU-Aktivierung)	✓
Residuals	✗	✓
LayerNorm	✗	✓

Warum diese Vereinfachung?

- Fokus auf Kernprinzip Attention + Gradienten
- Komplexität reduziert für besseres Verständnis



Einordnung

- Short-Term vs. Long-Term Memory: kurzfristige Kontextverarbeitung vs. längerfristige Speicherung/Abfrage.
- Ziel: relevante Informationen selektiv behalten (komprimieren) statt alles im Kontext zu halten.
- Parallele zu RAG: Auch dort wird Wissen extern gehalten, aber mit anderen Mechanismen/Trade-offs.

Wo arbeiten LLMs mit Tensoren?

Objekt	Tensor-Rang	Beispiel
Eingabevektor (Token)	1	[1, 2, 1, 0]
Gewichtsmatrizen (W_a , W_k , W_v)	2	[4 x 4]
Attention-Scores	1	Score-Liste
Attention-Output-Vektor	1	[0.304, 0.529, 0.603, 0.399]
Multi-Head Attention (echte LLMs)	4	{Batch, Heads, Seq, Seq}
Komplette Modell-Batches	3	(Batch, Sequence, Hidden) → z.B. (32, 128, 768)



Einordnung

- Persistent Memory: Informationen werden über Sessions/Anfragen hinweg gespeichert und wieder verwendet.
- Nutzen: Personalisierung, langfristige Aufgabenverfolgung, konsistente Wissensbasis.
- Herausforderung: Aktualisierung, Vergessen, Datenschutz und Kontrolle über gespeicherte Inhalte.

Ausblick: Titans-Architektur

Ein Ausblick auf neue Architekturen

Titans (Modellarchitektur)

Weiterentwicklung klassischer Transformer-Architekturen
Alternative Blockdesigns: Attention + MLP + Gating
Verbesserte Positionsemmbeddings (Rotary, State-Space Embeddings)
Bessere Parallelisierung, längere Kontexte
Genutzt in Gemini 1.5, Gemini Ultra (Google DeepMind)

Titans (Hardware / TPU v5p)

Spezialhardware für LLM-Training
Optimiert für massive Matrixoperationen (MatMul, Attention)
Tausende TPUs pro Cluster (TPU v5p Pod)
Ermöglicht Training von Multi-Billionen-Parameter-Modellen

Grundprinzip bleibt:

Tensoroperationen, Softmax, Cross-Entropy, Gradientenabstieg
Alle Prinzipien bleiben gleich nur die Skala wächst auf Milliarden Rechenoperationen und Billionen Parameter

Surprise sagt dem System: "Das war neu oder unerwartet":
„Ich habe meine Rechnung verloren, können Sie mir helfen?“
→ Das Modell erkennt eine neue Bedeutungskombination „Rechnung verloren“ = hoch überraschend, da selten gesehen.
→ Überraschung → wird gespeichert (Surprise Memory)

Long-Term Memory speichert diesen Überraschungszustand länger, weil es häufig auftritt. Modell liest oft und speichert es:
– „Bitte schicken Sie mir eine Kopie der letzten Rechnung.“
– „Ich habe meine letzte Rechnung nicht erhalten.“
– „Könnte ich eine Rechnung erneut bekommen?“

Adaptive Forgetting entfernt alte oder irrelevante Informationen automatisch.
Eine alte Regel, dass „Rechnung“ häufig im Vertriebskontext vorkommt, wird weniger wichtig, weil die neue Kundensupport-Bedeutung häufiger genutzt wird.
→ Diese alte Info wird aus dem aktiven Gedächtnis verdrängt.

Momentum sorgt dafür, dass das Memory weich und stabil angepasst wird.

Persistent Memory speichert dauerhaft aufgabenübergreifendes Wissen.
„Rechnung verloren“ gehört oft zu Anfragen wie Kopie zusenden, Kundenservice, PDF bereitstellen:
→ Diese Assoziation wird dauerhaft im Aufgabenwissen gespeichert

Ausblick: Titans



Einordnung

- Zusammenfassung Titans: Memory als dritte Säule neben Parametern und Prompt-Kontext.
- Einordnung: unterschiedliche Speicherarten lösen unterschiedliche Probleme (Kontextlänge vs. Langzeitkonsistenz).
- Ausblick: Kombinationen aus RAG, Caching und Memory-Architekturen in modernen Systemen.

Titans – Architekturvergleich

Komponente	Beispiel	GPT/BERT	Titans
Tokenization	✓	✓	
Embedding	✓	✓	✓ (SSM State Space Position Encoding)
Self-Attention	✓	✓	✓ (verbessert: : multi-query, Gating, SSM)
Feedforward (MLP)	✗	✓ (GeLU)	✓ (Gating)
Residuals/LayerNorm	✗	✓	✓
Gating	✗	✗	✓
PositionEmbedding	implizit	✓ (rotary)	✓ (Rotary, SSM)

Gating: funktioniert wie ein neuronaler Schalter, der entscheidet, ob Informationen im Netzwerk weitergegeben oder unterdrückt werden.

MLP (Multilayer Perceptron): vollständig verbundene neuronale Netze, die nach der Attention-Schicht folgen und die Informationen weiterverarbeiten. In GPT kommt z. B. eine Aktivierungsfunktion wie GeLU zum Einsatz. Nach der Attention folgt bei GPT ein sogenannter Feedforward-Block, auch MLP genannt, ein mehrschichtiges neuronales Netz, das die berechneten Token-Vektoren weiter transformiert. Titans verwenden hier zusätzlich ein Gating, also eine Art Filtermechanismus.



Einordnung

- Grenzen von LLMs: Halluzinationen, begrenztes Kontextfenster, Kosten (Compute/Token).
- Sicherheits- und Qualitätsaspekte: Prompt Injection, Datenleaks, Bias und fehlende Garantien.
- Konsequenz für Praxis: Evaluierung, Quellen, Guardrails und human-in-the-loop bei kritischen Anwendungen.

Titans Memory Architektur

Titans: Erweiterung klassischer Transformer mit Memory-Komponenten

Neue Gedächtnisarchitektur in Titans

Kombiniert klassische Attention mit zusätzlichen Speichersystemen:

→ Short-Term Memory (klassische Attention)

Berechnet wie im Transformer Q, K, V und Softmax

Verarbeitet lokalen Kontext innerhalb der Sequenz

→ Neural Long-Term Memory (Surprise Memory)

Speichert nur "überraschende" Informationen

Überraschung wird aus dem Gradienten berechnet
(höhere Überraschung → Speicherung)

Vergleichbar mit menschlichem selektivem Lernen

→ Persistent Memory (Aufgabenwissen)

Langfristig trainierte Zusatzrepräsentation

Wird am Anfang jeder Eingabe bereitgestellt

Dient als konstanter Aufgaben-Kontext

→ Adaptive Forgetting

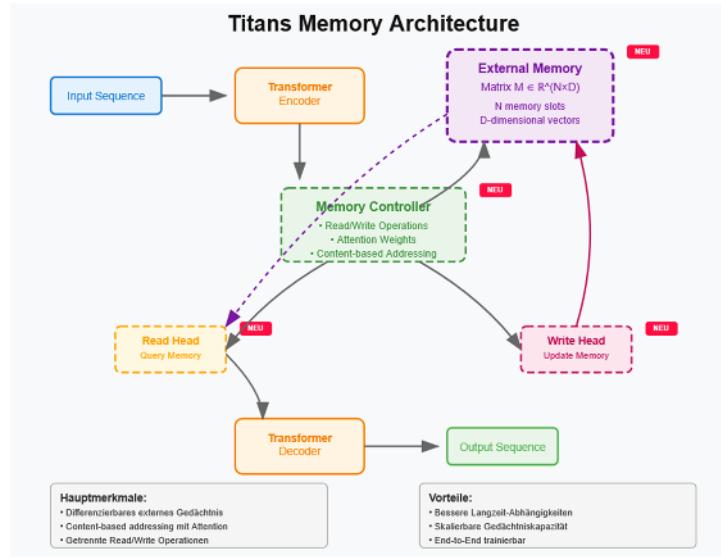
Gating-Mechanismus steuert gezieltes Vergessen von unwichtigen Informationen

Warum Titans?

Skalierbar auf extrem lange Kontexte (> 2 Mio Tokens)

Stabileres und selektiveres Lernen

Besser für komplexe Aufgaben mit vielen Zwischenzuständen (Planung, Reasoning, Retrieval)



Grenzen von klassischen Language Modells

Diese bekannten Schwächen gelten auch für GPT-Modelle wie ChatGPT:

- **Halluzinationen:**
Generierung von falschen Inhalten mit hoher Überzeugungskraft: Modell bestätigt von sich überzeugt die eigenen Fehler.
- **Wissensgrenze:**
Modelle kennen keine Ereignisse nach dem Trainingszeitpunkt.
- **Fehlende Quellenangaben:**
Keine direkten oder überprüfbaren Zitate verfügbar.
- **Datenschutz & Zugriff:**
Kein Zugang zu privaten Daten oder proprietären Informationen.
- **Begrenzte Kontextlänge:**
Nur eine begrenzte Anzahl an Tokens kann gleichzeitig verarbeitet werden.
- **Vergessen von Informationen:**
- Modell hat ein Problem, sich über lange Zeiträume Informationen zu merken.
- **Kosten:** Hohe Kosten für das Generieren langer Texte: Quadratische Abhängigkeit der Rechenzeit und des Speichers von der Sequenzlänge ist der Hauptgrund für die Begrenzung der Kontextlänge.
- GPT versteht überhaupt nichts, es kann nur sehr gut Wahrscheinlichkeiten modellieren.
- GPT hat keine Gefühle oder Intentionen, es berechnet nur die wahrscheinlichste Wortfolge.

GPT scheitert bei:

- Logikrätsel
- Rechenaufgaben mit langen Ketten
- Fakten nach dem Trainingszeitpunkt

Fazit & Diskussion

Kein echtes Weltwissen, aber sehr gutes Musterlernen
„Was, wenn das Modell meine Stimme lernt?“

Glossar wichtiger Begriffe

Begriff	Kurzerklärung	Verwendet in
Adaptive Forgetting	Zielgerichtetes Vergessen irrelevanter Informationen	Titans Memory
Äußeres Produkt	Produkt aus Spalten- und Zeilenvektor; ergibt eine Matrix, wichtig für Gradienten wie $\partial L / \partial W$. In 3D: Vektorprodukt (Kreuzprodukt)	Gradientberechnung, Matrixformeln
Backpropagation	Rückrechnung des Fehlers zur Gewichtsoptimierung	Gradient-Updates
Cross-Entropy Loss	Berechnet den Fehler der Vorhersage	Training
Decoder-only	Modelltyp, der nur Vorwärtsrichtung nutzt (z. B. GPT)	Textgenerierung, GPT-Architektur, Vorhersage des nächsten Tokens
Embedding	Umwandlung von Tokens in Vektoren	Eingabeschicht
Encoder-Decoder	Zwei-Komponenten-Modell für z. B. Übersetzungen (z. B. T5, BART)	Klassifikation, Masked Language Modeling (z. B. BERT), Eingabeanalyse
Encoder-only	Modelltyp mit vollem Kontexteinblick (z. B. BERT)	Übersetzung, Zusammenfassung, Aufgaben mit getrennter Eingabe und Ausgabe
Feedforward (MLP)	Weiterverarbeitung pro Token nach Attention	GPT/BERT Architektur
Gating	Mechanismus zur selektiven Signalweitergabe	Titans Feedforward
Gradientenabstieg	Algorithmus zur schrittweisen Fehlerreduktion	Training, Update-Regel
Head	Ein einzelner Aufmerksamkeitsmechanismus, der Query, Key und Value verarbeitet	Bestandteil von Multi-Head Attention
Jacobi-Matrix	Sensitivität der Ausgaben auf die Eingaben	(kurz bei Softmax-Ableitung)
LayerNorm	Normierung der Zwischenausgaben	GPT/BERT Architektur
Likelihood	Wahrscheinlichkeit für die Vorhersage des korrekten nächsten Tokens gegeben den bisherigen Kontext. Das Modell maximiert diese Wahrscheinlichkeit im Training	Cross-Entropy, Training, Softmax-Ausgabe
Logits (z)	Rohwerte vor der Softmax-Normalisierung	Vor Softmax
Matrix	2-dimensionales Zahlenarray (z. B. Gewichts- oder Embedding-Matrizen)	Gewichtsmatrizen, Attention, Embedding
Mixture of Experts (MoE)	Dynamische Auswahl von Modellkomponenten	(nicht im Vortrag, nur am Rande in Titans möglich)
Multi-Head Attention	Parallele Anwendung mehrerer Heads zur besseren Erfassung unterschiedlicher Beziehungsmuster	Attention-Formeln, Erweiterung in LLMs: Titans Architektur
Neural Long-Term Memory	Speicherung überraschender Informationen	Titans Memory
Persistent Memory	Langfristige Aufgabenrepräsentation	Titans Memory
Positional Encoding	Positionsinformation für Reihenfolge der Tokens	Eingabeschicht Attention
p-y (Gradient)	Fehlermaß zwischen Vorhersage und Ziel	Gradient, Backpropagation
Q, K, V (Query, Key, Value)	Mathematische Projektionen, die Attention ermöglichen	Attention-Formeln
Query (Q)	Repräsentation des aktuellen Tokens, das Informationen aus anderen Tokens anfordert ("Wer schaue ich an?")	Attention-Formeln, Frage
Key (K)	Repräsentation aller Tokens, die betrachtet werden können ("Wer bietet sich an?")	Attention-Formeln, Angebot
Value (V)	Inhaltliche Informationen der Tokens, die (gewichtete) Antworten liefern ("Welche Information bekomme ich?")	Attention-Formeln, Information
RAG (Retrieval-Augmented Generation)	Kombination aus Sprachmodell (LLM) und externem Datenabruft. Vor der Antwort werden passende Dokumente oder Fakten aus z. B. Datenbanken oder Wissensquellen geholt und als Kontext in das Modell eingebettet	Kontextverarbeitung, Anwendung in Agentforce, Verbindung von LLM und externem Wissen
Residual Connection	Hinzufügen der Eingangswerte zur Stabilisierung	GPT/BERT Architektur
Schwellenwert	Wert, ab dem eine Entscheidung getroffen wird (z. B. Aktivierungsfunktion, binäre Klassifikation)	Neuronen, Aktivierung, Klassifikation
Self-Attention	Tokens beziehen gegenseitig Informationen aufeinander	Hauptblock Attention
Skalarprodukt	Multiplikation zweier Vektoren mit Ausgabe eines Skalars (z. B. Ähnlichkeitsmaß)	Attention-Berechnung Ähnlichkeit
Softmax	Umwandlung von Scores in Wahrscheinlichkeiten	Attention-Output Layer
SSM (State Space Model)	Alternative für Positional Encoding in Gemini	Titans Position-Encoding
Surprise Memory	Memory-Komponente, die seltene oder überraschende Muster speichert	Titans-Architektur
Tensor	Mehrdimensionale Verallgemeinerung von Vektoren und Matrizen	Eingabedaten, Modellparameter, Attention-Berechnungen
Titans Architektur	Erweiterte Transformer-Architektur mit Gating & Memory	Ausblick
Token	Kleinste Einheit des Inputs (z. B. Wort, Subwort, Zeichen)	Tokenisierung, Eingabe
Tokenization	Zerlegung des Textes in Tokens	Vorverarbeitung
TPU (Tensor Processing Unit)	Spezialisierte Hardware für LLM-Training	Titans Hardware
Vektor	Zahlenliste zur Darstellung von Tokens	Embedding, Attention