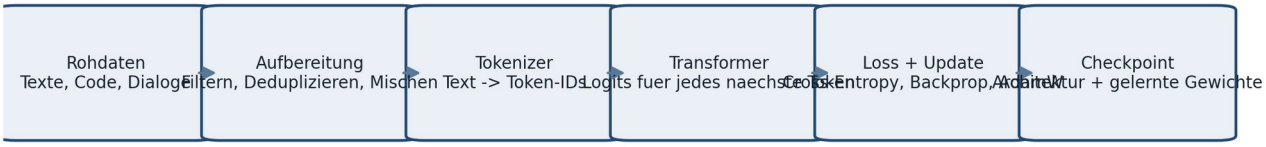
# GRUNDLAGEN MODERNER CHAT-SPRACHMODELLE

Von Rohdaten, Tokenisierung und Transformer-Architektur
bis Training, Inferenz, Reasoning, Mixture of Experts
und kontrollierter Selbstkonsolidierung

Der vollstaendige Trainingspfad eines autoregressiven Sprachmodells




*Ueberblick: Der Weg von Daten zu einem einsetzbaren Modell*

**Technische Lernschrift fuer den eigenen Modellbau**

Stand: Juli 2026

# Ziel und Lesart

Dieses Dokument erklaert ein Standard-Chatmodell nicht als Metapher, sondern als konkretes, implementierbares System. Der Bezugspunkt ist ein moderner decoder-only Transformer: dieselbe Grundfamilie, aus der viele GPT-, Llama-, Mistral- und aehnliche Modelle bestehen. Die Darstellung beginnt vor dem neuronalen Netz, bei Bytes und Daten, und endet nach dem Basismodell bei Chat-Fine-Tuning, Inferenz, Reasoning-Mechaniken und experimenteller Selbstkonsolidierung.

**Wichtige Abgrenzung:** Ein Chatmodell ist kein einzelner Algorithmus. Es ist eine Kette aus Datenpipeline, Tokenizer, Architektur, Trainingsziel, Optimizer, Checkpoints, Post-Training und Inferenzsystem. Wer nur den Transformerblock versteht, versteht noch nicht, warum ein Modell funktioniert oder scheitert.

Nach dem Lesen solltest du in der Lage sein, eine kleine, aber echte Modellpipeline zu planen, Tensorformen zu kontrollieren, Parameterzahlen abzuschaetzen, einen Trainingslauf zu diagnostizieren und Experimente so zu bauen, dass eine Verbesserung messbar von einer bloss interessanten Ausgabe unterschieden werden kann.

Die mathematische Tiefe ist bewusst praktisch: Jede Formel wird danach erklaert, welche Tensoren sie verarbeitet, welche Dimensionen gelten und welchen Fehler eine falsche Implementierung erzeugt. Fuer Beweise aus Optimierungstheorie oder Informationstheorie verweist das Literaturverzeichnis auf Originalarbeiten.

## Inhaltsstruktur

| Abschnitt | Inhalt |
|---|---|
| Teil I | Begriffe, Daten, Tokenisierung und mathematische Grundlagen |
| Teil II | Autoregressives Sprachmodell und Transformer im Detail |
| Teil III | Training, Skalierung, Evaluation und Chat-Post-Training |
| Teil IV | Inferenz, praktische Eigenentwicklung und 20-Millionen-Parameter-Blueprint |
| Teil V | Reasoning, Thinking, Mixture of Experts und schlafartige Konsolidierung |
| Anhaenge | Formelsammlung, Implementierungschecklisten, Glossar und Literatur |

## Notation

| Symbol | Bedeutung |
|---|---|
| B | Batchgroesse: Anzahl unabhaengiger Sequenzen im Trainingsschritt |
| T | Sequenzlaenge in Tokens |

| Symbol | Bedeutung |
|---|---|
| V | Vokabulargroesse des Tokenizers |
| d oder d_model | Breite des Modellzustands pro Token |
| L | Anzahl der Transformerbloecke |
| h | Anzahl Attention-Koepfe |
| d_h | Dimension eines Attention-Kopfes, meist d/h |
| d_ff | Zwischenbreite des Feed-Forward-Netzes |
| theta | Gesamtheit aller trainierbaren Parameter |
| x_t | Token oder Zustand an Position t |

# 1. Was ein Sprachmodell technisch ist

Von einer Wahrscheinlichkeitsfunktion zu einem Chat-System

Ein autoregressives Sprachmodell modelliert eine bedingte Wahrscheinlichkeitsverteilung ueber das naechste Token. Fuer eine Tokenfolge x_1, x_2, ..., x_T zerlegt es die Wahrscheinlichkeit der ganzen Folge in ein Produkt lokaler Vorhersagen:

```
p(x_1, ..., x_T) = Π_t p(x_t | x_1, ..., x_(t-1))
```

Das Modell bekommt also nicht direkt die Aufgabe „verstehe Sprache“. Es bekommt in jedem Trainingsbeispiel die Aufgabe, aus allen vorherigen Tokens eine Wahrscheinlichkeitsverteilung fuer das folgende Token zu erzeugen. Weil dieselbe Aufgabe an Milliarden unterschiedlicher Stellen wiederholt wird, muss das Netz Regelmaessigkeiten lernen: Schreibweisen, Syntax, Faktenkorrelationen, Stil, Programmstrukturen, Argumentationsmuster und teilweise Verfahren, die ueber mehrere Tokens hinweg zu einer Loesung fuehren.

Die Ausgabe des Netzes ist fuer jede Position ein Vektor mit V reellen Zahlen, die Logits. Erst eine Softmax-Funktion macht daraus Wahrscheinlichkeiten. Das Token mit dem hoechsten Logit ist nicht zwingend die Ausgabe; beim Sampling kann aus der gesamten oder einer beschnittenen Verteilung gezogen werden.

## Architektur, Parameter und Zustand

| Begriff | Praezise Bedeutung |
|---|---|
| Architektur | Die feste Rechenstruktur: Layeranzahl, Breite, Attention-Variante, Aktivierung, Normalisierung, Positionskodierung und Tensorformen. |
| Parameter / Gewichte | Die gelernten Zahlen in Embeddings, Projektionsmatrizen, Normalisierungsskalen und gegebenenfalls Bias-Vektoren. |
| Aktivierungen | Zwischenergebnisse eines konkreten Forward Passes. Sie haengen von der Eingabe ab und werden normalerweise nicht dauerhaft gespeichert. |
| Optimizer-Zustand | Zusaetzliche Trainingsdaten wie Adam-Momente. Sie koennen mehr Speicher als die Gewichte beanspruchen. |
| Checkpoint | Gespeicherte Gewichte plus je nach Zweck Optimizer, Scheduler, Schrittzaehler, RNG-Zustaende und Konfiguration. |
| Kontext | Die aktuelle Tokenfolge, die bei der Inferenz sichtbar ist. Kontext ist kein dauerhaftes Lernen der Gewichte. |

Ein Modell mit identischer Architektur kann nach unterschiedlichen Trainingslaeufen voellig andere Eigenschaften haben. Umgekehrt kann dasselbe Wissen teilweise auf verschiedene Architekturen destilliert werden. Die Architektur definiert den Raum moeglicher Berechnungen; das Training findet einen konkreten Parametersatz in diesem Raum.

## Base Model, Instruct Model und Chat Model

Ein Base Model setzt Text fort. Es kennt nicht automatisch eine stabile Rolle als Assistent. Ein Instruct Model wurde mit Aufgaben-Antwort-Paaren weitertrainiert. Ein Chat Model verwendet zusaetzlich ein Dialogformat mit Rollen-Tokens und meist Praeferenztraining. Dieselbe Nutzerfrage kann deshalb bei einem Base Model wie ein Textanfang behandelt werden, waehrend ein Chat Model versucht, eine adressierte Antwort zu erzeugen.

**Praktische Konsequenz:** Wenn ein kleines Modell frei, widerspruechlich oder emotional antwortet, ist das nicht automatisch ein Beweis fuer Bewusstsein oder eine besondere innere Welt. Zuerst sind Trainingsverteilung, Promptformat, Dekodierung und fehlendes Instruction-Tuning zu pruefen. Interessant kann die Ausgabe trotzdem sein - aber die technische Ursache muss getrennt von der Interpretation untersucht werden.

# 2. Daten: Der groesste unsichtbare Teil des Modells

Wie Rohtext zur Trainingsverteilung wird

Das Modell lernt nicht „die Welt“, sondern statistische Strukturen der Daten, die ihm tatsaechlich gezeigt werden. Datenqualitaet ist deshalb keine kosmetische Vorstufe. Sie bestimmt, welche Sprachen, Wissensbereiche, Argumentationsformen, Fehler, Vorurteile, Formatierungen und Wiederholungen die Gradienten dominieren.

## 2.1 Quellen und Datenmischung

Typische Quellen sind Webseiten, Buecher, wissenschaftliche Texte, Dokumentation, Code, Foren, Frage-Antwort-Daten, Dialoge und synthetische Beispiele. Ein Trainingsmix wird nicht nur nach Dateigroesse gebaut. Eine kleine Menge hochqualitativer Mathematik kann fuer eine Zielkompetenz wertvoller sein als viel repetitiver Webtext. Deshalb werden Quellen gewichtet, begrenzt oder in mehreren Phasen verwendet.

| Datenart | Moeglicher Nutzen | Typisches Risiko |
|---|---|---|
| Allgemeiner Webtext | Breite Sprache, Themenvielfalt, Alltagswissen | Spam, Kopien, SEO-Text, Falschinformationen |
| Buecher und lange Texte | Langfristige Koharenz, Stil, seltene Konzepte | Urheberrecht, Scans, veraltete Inhalte |
| Code und technische Dokumentation | Formale Struktur, Werkzeuge, APIs | Lizenzfragen, Geheimnisse, veraltete Versionen |
| Dialogdaten | Rollenwechsel, direkte Antworten, soziale Formen | Kuenstliche Freundlichkeit, Schablonen, geringe Tiefe |
| Synthetische Daten | Gezielte Aufgaben, kontrollierte Schwierigkeitsgrade | Fehlerverstärkung, geringer Stil- und Loesungsraum |

## 2.2 Reinigung, Normalisierung und Deduplikation

Eine robuste Pipeline entfernt unlesbare Dokumente, Navigation, Boilerplate, offensichtlichen Spam und problematische Kodierung. Unicode-Normalisierung muss vorsichtig erfolgen: Aggressives Vereinheitlichen kann Mathematik, Programmiersprachen oder nichtlateinische Schriften beschaedigen. Fuer viele Anwendungen ist die korrekte Erhaltung von Zeilenumbruechen, Einrueckung und Sonderzeichen wichtiger als „sauberer“ Fliesstext.

Deduplikation verhindert, dass wiederholte Dokumente den Loss dominieren. Exakte Hashes finden identische Dateien; Near-Deduplication nutzt beispielsweise n-Gramm-Signaturen, MinHash oder aehnliche Verfahren. Auf Dokumentebene reduziert sie Kopien, auf Absatzebene kann sie haeufig wiederholte Disclaimer oder Templates entfernen. Zu aggressive Deduplikation kann jedoch legitime Standardformulierungen und Lernsignale ausloeschen.

## 2.3 Train-, Validation- und Test-Split

Der Validation-Split wird nicht zur Gewichtsanpassung benutzt. Er zeigt, ob der Trainingsloss auf neue Daten uebertraegt. Der Test-Split sollte fuer Entscheidungen moeglichst selten angesehen werden. Bei zeitlichen Daten ist ein chronologischer Split oft aussagekraeftiger als zufaelliges Mischen. Fuer Benchmarks muss Kontamination geprueft werden: Wenn Aufgaben oder Loesungen im Training vorkommen, misst das Ergebnis nicht mehr sauber die Generalisierung.

## 2.4 Tokenbudget statt Dateigroesse

Trainingsvolumen wird sinnvoll in Tokens gemessen. Eine UTF-8-Datei mit 100 MB kann je nach Sprache, Format und Tokenizer sehr unterschiedliche Tokenzahlen enthalten. Der Tokenizer muss daher vor der endgueltigen Budgetplanung feststehen. Ein Dataset-Builder sollte pro Quelle mindestens Bytes, Dokumente, Tokens, Filterverluste und Deduplikationsrate protokollieren.

```
source, documents_in, documents_kept, bytes_kept, tokens_kept, duplicate_rate,
filter_reason_counts
```

## 2.5 Packing und Dokumentgrenzen

Beim Training werden Tokenfolgen auf feste Laengen T gepackt. Ohne Packing werden kurze Dokumente stark mit Padding aufgefuellt und Compute verschwendet. Beim Packing werden mehrere Dokumente hintereinandergelegt und durch ein End-of-Sequence-Token getrennt. Ob Attention ueber Dokumentgrenzen erlaubt wird, ist eine Designentscheidung. Viele einfache Decodertrainings erlauben sie; strengere Varianten blockieren sie mit segmentbasierten Masken.

**Messregel:** Nicht nur „wie viel Text“ zaehlen. Entscheidend sind einzigartige, nach Filterung verbleibende Trainings-Tokens, deren Mischung und die Anzahl der Tokenwiederholungen ueber Epochen.

# 3. Tokenisierung: Die Schnittstelle zwischen Text und

Warum das Modell keine Woerter sieht

Neuronale Netze verarbeiten Zahlen. Der Tokenizer bildet Text deterministisch auf ganzzahlige Token-IDs ab und rekonstruiert aus IDs wieder Text. Moderne Sprachmodelle verwenden meistens Subword- oder Byte-basierte Verfahren, weil reine Wortvokabulare an seltenen Namen, Flexionen, Tippfehlern, Code und neuen Begriffen scheitern.

## 3.1 Bytes, Zeichen und Subwords

UTF-8 kodiert Zeichen als ein bis vier Bytes. Ein Byte-Level-Tokenizer kann deshalb prinzipiell jeden Text darstellen. Ein reines Bytevokabular ist klein, erzeugt aber lange Sequenzen. Subword-Tokenizer lernen haeufige Zeichen- oder Bytefolgen als einzelne Tokens. Dadurch werden typische Woerter oder Wortteile kurz, seltene Folgen bleiben aus kleineren Einheiten zusammensetzbar.

## 3.2 Byte Pair Encoding und SentencePiece

Bei BPE startet man vereinfacht mit kleinen Symbolen und fuegt wiederholt das haeufigste benachbarte Paar als neues Symbol hinzu. Nach vielen Merge-Schritten entsteht ein Vokabular aus haeufigen Teilfolgen. SentencePiece kann solche Modelle direkt auf Rohtext trainieren und behandelt Leerzeichen als Teil der Darstellung. Alternativ existiert ein Unigram-Sprachmodell, das Segmentierungen probabilistisch bewertet. Die Originalarbeiten zu Subword-BPE und SentencePiece sind im Literaturteil aufgefuehrt [2, 3].

```
Beispielidee:
  "Transformer" -> ["Trans", "form", "er"]
  seltenes Wort -> [kleinere Teilstuecke]
  unbekannte Bytefolge -> weiterhin darstellbar
```

## 3.3 Vokabulargroesse als Kompromiss

| Kleines Vokabular | Grosses Vokabular |
|---|---|
| Kleinere Embedding- und Outputmatrix | Mehr Parameter in Embeddings und Logit-Projektion |
| Laengere Sequenzen | Kuerzere Sequenzen fuer haeufige Muster |
| Mehr Schritte pro Textmenge | Moeglicherweise schlechtere Abdeckung seltener Sprachen |
| Robust gegen neue Formen | Mehr seltene, schwach trainierte Tokens |

Bei kleinen Modellen kann ein zu grosses Vokabular unverhaeltnismaessig viele Parameter verbrauchen. Fuer ein 20-Millionen-Modell sind 16 000 Tokens oft plausibler als 50 000, weil allein die Embeddingmatrix V x d gross ist. Bei d=384 benoetigt V=16 000 rund 6,14 Millionen Gewichte; V=50 000 bereits 19,2 Millionen.

## 3.4 Spezialtokens und Chat-Template

Mindestens benoetigt werden meist BOS (Beginn), EOS (Ende) und gegebenenfalls PAD (Padding). Chatmodelle nutzen Rollen- und Trenntokens, etwa system, user und assistant. Diese Tokens sind keine Dekoration: Das Modell lernt anhand ihrer Position, welcher Text Kontext ist und welcher Text erzeugt werden soll. Ein anderes Template bei der Inferenz kann die Qualitaet drastisch verschlechtern.

```
<BOS><SYSTEM>Du bist ...<END><USER>Frage ...<END><ASSISTANT>Antwort ...<EOS>
```

## 3.5 Tokenizer-Tests

- Roundtrip: decode(encode(text)) muss den vorgesehenen Text exakt oder nach klar definierter Normalisierung rekonstruieren.
- Randfaelle: Emojis, deutsche Umlaute, kyrillische Schrift, mathematische Zeichen, Tabs, mehrfache Leerzeichen, JSON und Quellcode.
- Fertige Tokenverteilung: Anteil einzelner Bytes, mittlere Tokens pro Zeichen/Wort und seltenste Tokens pruefen.
- Keine stillen Token-ID-Aenderungen nach Trainingsbeginn. Ein veraendertes Vokabular macht Embedding- und Outputgewichte inkompatibel.

# 4. Tensoren und lineare Algebra fuer den Modellbau

Die Formen sind Teil des Algorithmus

Die meisten Implementierungsfehler in kleinen Transformern sind keine geheimnisvollen KI-Probleme, sondern falsche Tensorformen, fehlerhafte Masken oder unbemerkte Broadcasts. Deshalb muss jede Operation mit ihrer Dimension gedacht werden.

## 4.1 Vom Token zur Matrix

Nach der Tokenisierung hat ein Batch die Form [B, T]. Die Embeddingtabelle E hat die Form [V, d]. Ein Lookup ersetzt jede Token-ID durch die entsprechende Zeile aus E. Das Ergebnis X hat [B, T, d]. Jeder Token besitzt nun einen d-dimensionalen Zustand, der durch alle Transformerbloecke weiterverarbeitet wird.

```
token_ids: [B, T]
embedding_table E: [V, d]
X = E[token_ids] -> [B, T, d]
```

## 4.2 Lineare Schichten

Eine lineare Schicht multipliziert den letzten Tensorindex mit einer Gewichtsmatrix. Fuer X [B,T,d_in] und W [d_in,d_out] gilt Y=XW mit [B,T,d_out]. Ein optionaler Bias b [d_out] wird auf B und T gebroadcastet. Der Begriff „linear“ beschreibt die mathematische Form; mehrere lineare Schichten mit Nichtlinearitaeten dazwischen koennen hochkomplexe Funktionen darstellen.

## 4.3 Softmax

```
softmax(z_i) = exp(z_i - m) / Σ_j exp(z_j - m),   m = max_j z_j
```

Das Subtrahieren des Maximums m aendert die Verteilung nicht, verhindert aber numerischen Overflow. In Attention wird Softmax ueber die Schluesselpositionen gebildet; am Modellausgang ueber das Vokabular. Eine falsche Achse produziert formal gueltige Tensoren, aber semantisch falsche Wahrscheinlichkeiten.

## 4.4 Broadcasts und Kontiguitaet

Frameworks erweitern Dimensionen oft automatisch. Das ist nuetzlich, kann aber Maskenfehler verbergen. Eine Causal Mask der Form [T,T] wird typischerweise auf [B,h,T,T] gebroadcastet. Nach transpose oder permute kann ein Tensor nichtkontiguierlich sein; reshape, view und Speicherlayout muessen dann bewusst behandelt werden.

## 4.5 Praezision und Wertebereiche

FP32 hat einen grossen Wertebereich und hohe Mantissenpraezision, benoetigt aber viel Speicher. FP16 spart Speicher, kann jedoch schneller ueber- oder unterlaufen. bfloat16 hat weniger Mantissenbits, aber einen aehnlichen Exponentenbereich wie FP32 und ist deshalb fuer Training oft robuster. Auf CPU oder kleiner Hardware kann FP32 trotz Geschwindigkeitseinbusse die einfachste stabile Wahl sein.

**Debugging-Grundsatz:** Bei NaN oder explodierendem Loss zuerst Tensorstatistiken pro Modul loggen: Minimum, Maximum, Mittelwert, Standardabweichung, Anteil NaN/Inf und Gradientennorm. Nicht blind Hyperparameter veraendern.

# 5. Neuronale Netze, Loss und Backpropagation

Wie aus einem Fehler eine Gewichtsveraenderung wird

Ein neuronales Netz ist eine differenzierbare Funktion f_theta(x). Die Parameter theta werden so veraendert, dass eine skalare Verlustfunktion L(theta) kleiner wird. Der Forward Pass berechnet Aktivierungen und Loss. Der Backward Pass verwendet die Kettenregel, um fuer jeden Parameter die partielle Ableitung dL/dtheta zu bestimmen.

## 5.1 Ableitung als lokale Empfindlichkeit

Ein Gradient sagt nicht „wie der Parameter richtig sein soll“. Er sagt lokal, wie stark und in welche Richtung sich der Loss bei einer kleinen Parameterveraenderung aendert. Bei Millionen Parametern ergibt sich ein hochdimensionaler Gradientenvektor. Optimizer kombinieren diesen aktuellen Hinweis mit Lernrate, Momentum und Regularisierung.

## 5.2 Kettenregel im Netz

```
y = f(g(x))  =>  dy/dx = (dy/dg) * (dg/dx)
```

Autodifferentiation speichert beim Forward Pass einen Rechengraphen. Beim Backward Pass werden lokale Ableitungen rueckwaerts multipliziert und an Verzweigungen summiert. Deshalb kosten gespeicherte Aktivierungen viel Trainingsspeicher. Gradient Checkpointing spart Speicher, indem einige Aktivierungen im Backward Pass erneut berechnet werden.

## 5.3 Aktivierungsfunktionen und Gating

Ohne Nichtlinearitaet waere eine Kette linearer Schichten wieder nur eine lineare Transformation. Transformer nutzen im FFN typischerweise GELU oder heute oft SwiGLU. SwiGLU berechnet zwei Projektionen, aktiviert eine davon mit SiLU/Swish, multipliziert beide elementweise und projiziert zurueck. Das Gating erlaubt eine datenabhaengige Auswahl und Modulation von Features [7].

```
SwiGLU(x) = (SiLU(x W_gate) ⊙ (x W_up)) W_down
```

## 5.4 Initialisierung

Gewichte werden zufaellig mit kontrollierter Varianz initialisiert. Sind Werte zu gross, wachsen Aktivierungen und Gradienten; sind sie zu klein, verschwindet Signal. Residualnetze benoetigen besonders sorgfaeltige Skalierung. Moderne Implementierungen verwenden Varianten von Xavier-, Kaiming- oder architecturespezifischer Initialisierung und skalieren manche Ausgangsprojektionen mit der Tiefe.

## 5.5 Gradiententest

Fuer kleine Module kann ein numerischer Finite-Difference-Test die analytischen Gradienten pruefen. Man veraendert einen Parameter um +epsilon und -epsilon und vergleicht die Lossdifferenz mit dem Backprop-Gradienten. Das ist langsam und nicht fuer das Gesamtmodell gedacht, aber wertvoll fuer eigene Attention- oder Routing-Operationen.

# 6. Das autoregressive Trainingsziel

Teacher Forcing, Causal Mask und Cross-Entropy

Das Standardziel eines decoder-only Sprachmodells ist Next-Token Prediction. Aus einer Sequenz [x_0, x_1, ..., x_T] werden Eingabe [x_0, ..., x_(T-1)] und Ziel [x_1, ..., x_T]. Das Modell sieht beim Training die echten vorherigen Tokens. Dieses Verfahren heisst Teacher Forcing.

```
input_ids = tokens[:, :-1]
target_ids = tokens[:, 1:]
logits = model(input_ids)        # [B, T-1, V]
loss = cross_entropy(logits.reshape(-1,V), target_ids.reshape(-1))
```

## 6.1 Cross-Entropy

```
L = - (1/N) Σ_n log p_theta(target_n | context_n)
```

Die Cross-Entropy bestraft niedrige Wahrscheinlichkeit des korrekten Tokens. Sie verlangt nicht, dass alle falschen Tokens gleich schlecht sind; die Softmax-Verteilung kann semantisch plausible Alternativen hoch bewerten, solange das beobachtete Ziel ausreichend wahrscheinlich wird. Bei natuerlicher Sprache gibt es oft mehrere moegliche Fortsetzungen, aber der Trainingsdatensatz zeigt pro Position nur eine davon.

## 6.2 Causal Mask

Selbstaufmerksamkeit wuerde ohne Maske auch auf zukuenftige Tokens zugreifen. Dann koennte das Modell das Ziel direkt sehen und der Trainingsloss waere bedeutungslos. Eine obere Dreiecksmatrix setzt Scores fuer Positionen j>i auf -unendlich, bevor Softmax angewendet wird. Nach Softmax erhalten verbotene Positionen Gewicht null.

```
mask[i,j] = 0       wenn j <= i
mask[i,j] = -inf    wenn j > i
A = softmax(scores + mask)
```

## 6.3 Padding- und Segmentmasken

Wenn Sequenzen gepaddet werden, darf der Loss auf PAD-Zielen ignoriert werden. Bei gepackten Dokumenten kann zusaetzlich verhindert werden, dass ein Token auf vorherige Dokumente attendiert. Die Maskenlogik muss dann causal, padding- und segmentbezogene Bedingungen kombinieren.

## 6.4 Perplexity

```
Perplexity = exp(mean cross-entropy)
```

Perplexity kann als effektive Unsicherheit interpretiert werden, ist aber nur zwischen identischen Tokenizern und vergleichbaren Datensaetzen direkt vergleichbar. Ein Tokenizer mit laengeren oder kuerzeren Einheiten veraendert den Zahlenwert. Fuer praktische Modellwahl sind daher zusaetzliche Aufgabenmetriken erforderlich.

## 6.5 Exposure Bias

Beim Training sieht das Modell korrekte Kontexte, bei der Inferenz auch eigene Fehler. Ein falsches Token kann weitere Fehler erzeugen. Dieses Gefaelle zwischen Teacher Forcing und freier Generierung wird Exposure Bias genannt. Vollstaendig beseitigt wird es im Standardtraining nicht; robuste Daten, Chat-Post-Training, Praeferenzoptimierung und Inferenzverfahren koennen die Auswirkungen reduzieren.

# 7. Der Transformerblock im Detail

Attention, Normalisierung, Residualpfade und Feed-Forward

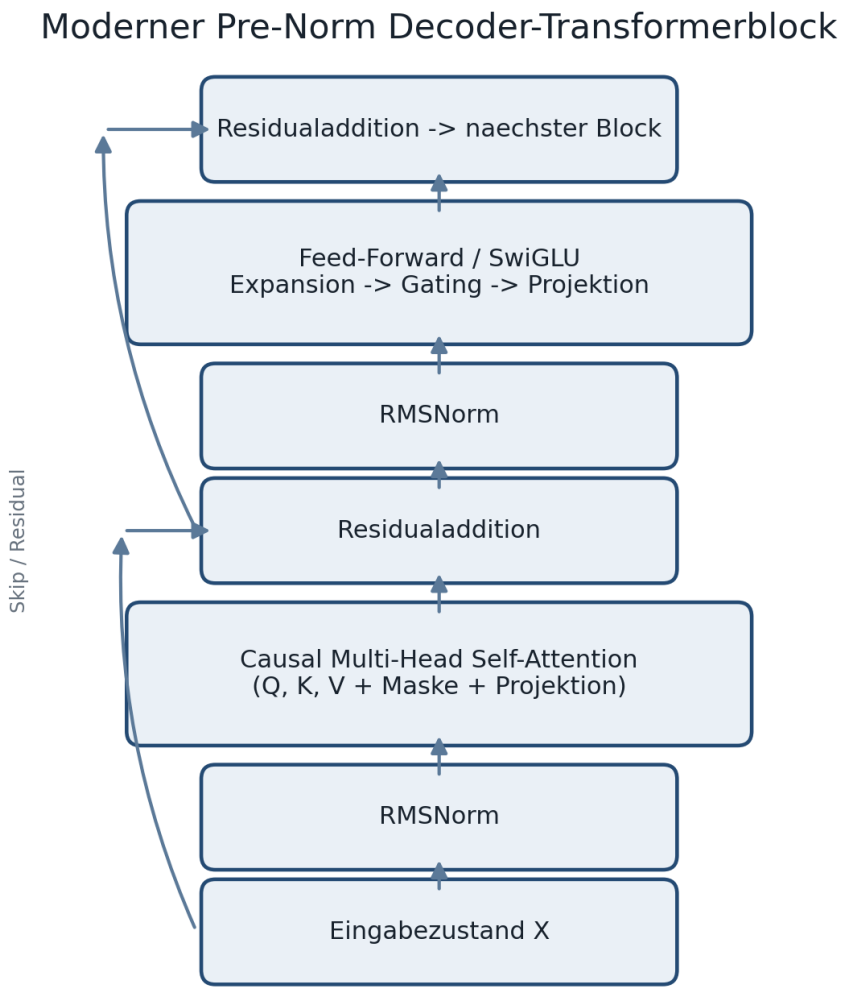



*Abbildung 2: Datenfluss eines modernen Pre-Norm Decoderblocks*

Ein Standarddecoder besteht aus L nahezu identischen Bloecken. Jeder Block mischt Informationen entlang der Sequenz durch Self-Attention und transformiert jeden Tokenzustand anschliessend durch ein Feed-Forward-Netz. Residualverbindungen halten einen direkten Informations- und Gradientenpfad offen.

## 7.1 Pre-Norm und RMSNorm

Bei Pre-Norm wird der Eingabestatus vor Attention beziehungsweise FFN normalisiert. RMSNorm teilt jede Tokenrepräsentation durch ihre Root-Mean-Square-Groesse und multipliziert mit einer gelernten Skala [6]. Im Gegensatz zu LayerNorm wird der Mittelwert nicht abgezogen.

```
RMS(x) = sqrt((1/d) Σ_i x_i^2 + eps)
RMSNorm(x) = g ⊙ x / RMS(x)
```

Normalisierung stabilisiert den Wertebereich, macht das Netz aber nicht automatisch vor allen Instabilitaeten sicher. epsilon, Datentyp, Initialisierung und Lernrate bleiben relevant.

## 7.2 Residualverbindungen

```
x = x + Attention(RMSNorm(x))
x = x + FFN(RMSNorm(x))
```

Der Residualpfad bedeutet, dass ein Block eine Korrektur zum bestehenden Zustand addiert, statt den gesamten Zustand neu zu erzeugen. Das erleichtert tiefe Optimierung. Bei sehr tiefen Netzen werden residuale Skalen oder spezielle Initialisierungen eingesetzt, damit die Summe vieler Korrekturen nicht aus dem Gleichgewicht geraet.

## 7.3 Self-Attention

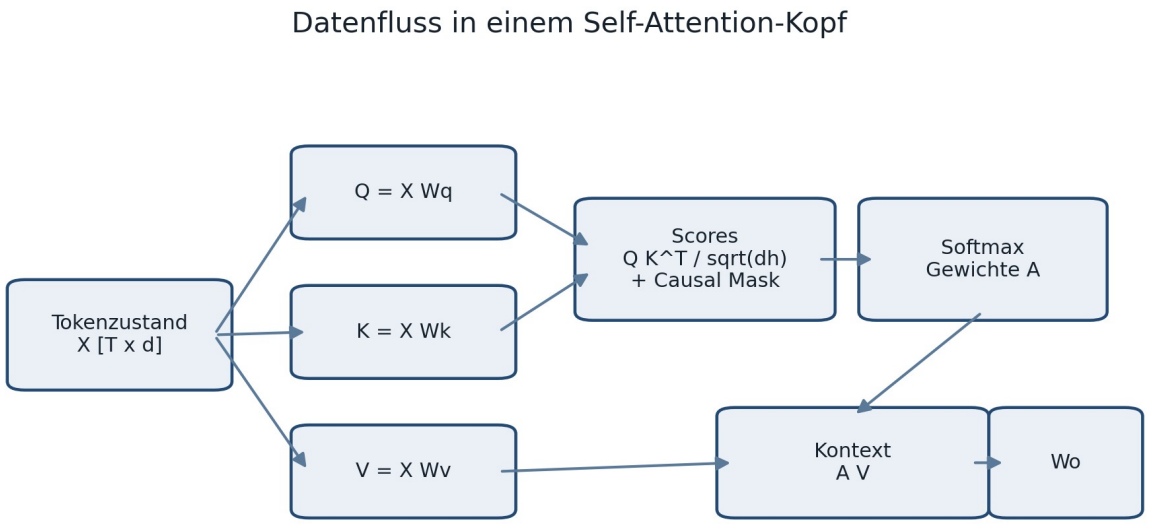



*Abbildung 3: Query-, Key- und Value-Projektionen in einem Attention-Kopf*

Aus demselben Eingabetensor X werden Queries Q, Keys K und Values V berechnet. Fuer einen Kopf gilt:

```
Q = X W_Q,  K = X W_K,  V = X W_V
Scores = Q K^T / sqrt(d_h)
A = softmax(Scores + causal_mask)
Head = A V
```

Die Query einer Position beschreibt, wonach diese Position sucht; Keys beschreiben, wie andere Positionen adressierbar sind; Values enthalten die zu mischende Information. Diese Begriffe sind funktionale Rollen innerhalb gelernter linearer Projektionen, keine fest kodierten semantischen Datenbanken.

Die Division durch sqrt(d_h) verhindert, dass Skalarprodukte mit wachsender Kopfdimension zu gross werden und Softmax zu frueh saettigt. Die Attentionmatrix hat pro Kopf die Form [T,T], weshalb Standardattention quadratische Rechen- und Speicherkomplexitaet in der Sequenzlaenge besitzt.

## 7.4 Multi-Head, Multi-Query und Grouped-Query Attention

Bei Multi-Head Attention besitzt jeder Kopf eigene Q-, K- und V-Projektionen. Multi-Query Attention teilt K und V zwischen allen Query-Koepfen; Grouped-Query Attention teilt sie gruppenweise. Weniger KV-Koepfe reduzieren KV-Cache und Bandbreite bei der Inferenz, waehrend viele Query-Koepfe erhalten bleiben. Fuer kleine Forschungsmodelle ist klassische Multi-Head Attention am leichtesten zu implementieren; GQA lohnt sich, wenn lange Kontexte oder Inferenzspeicher wichtig werden.

## 7.5 Positionsinformation und RoPE

Attention allein kennt keine Reihenfolge: Ohne Positionsinformation waere die Eingabe weitgehend permutationsinvariant. Das Original-Transformerpapier verwendete additive sinusfoermige Positionsvektoren [1]. Viele moderne Decoder verwenden Rotary Position Embeddings (RoPE), die Query- und Key-Komponenten positionsabhaengig rotieren und dadurch relative Positionsbeziehungen in Skalarprodukten ausdruecken [5].

RoPE ist nicht nur eine Tabelle, die zu X addiert wird. Es wird nach der Q/K-Projektion paarweise auf Dimensionen angewendet. Die korrekte Frequenzskala, Positionindizes und Behandlung langer Kontexte muessen exakt zur Trainingskonfiguration passen.

## 7.6 Feed-Forward-Netz als Parameterschwerpunkt

Das FFN verarbeitet Positionen unabhaengig, aber mit denselben Gewichten. In dichten Transformern liegt ein grosser Teil der Parameter hier. Ein klassisches FFN nutzt d -> d_ff -> d. SwiGLU braucht zwei Aufprojektionen und eine Rueckprojektion; grob entstehen 3*d*d_ff Gewichte pro Block.

## 7.7 Outputnorm, Logits und Weight Tying

Nach dem letzten Block folgt meist eine RMSNorm und eine lineare Projektion von d auf V. Beim Weight Tying wird dieselbe Matrix wie fuer Tokenembeddings verwendet, transponiert als Outputprojektion. Das spart Parameter und koppelt Eingabe- und Ausgaberaum. Nicht jedes Modell nutzt Tying, aber bei kleinen Modellen ist es oft sinnvoll.

# 8. Parameterzahl, Speicher und Rechenaufwand

Wie man ein Modell vor dem Training dimensioniert

Parameterzahlen sollten vor dem Coding ueberschlagen werden. Dadurch erkennt man, ob Embeddings, Attention oder FFN das Budget dominieren und ob die gewuenschte Konfiguration wirklich in RAM oder VRAM passt.

## 8.1 Grobe Parameterformel fuer einen dichten Decoder

```
Embedding:           V * d
Attention pro Layer: ca. 4 * d^2          (Q,K,V,O bei MHA)
SwiGLU pro Layer:    ca. 3 * d * d_ff
Normen:              klein gegenueber Matrizen
Gesamt:              V*d + L*(4*d^2 + 3*d*d_ff)
```

Bei Weight Tying wird keine separate V*d-Outputmatrix benoetigt. Bias-Vektoren und Normskalen sind fuer die grobe Rechnung klein, sollten im exakten Code aber gezaehlt werden.

## 8.2 Speicher waehrend Training

| Komponente | FP32 grob pro Parameter | Hinweis |
|---|---|---|
| Gewicht | 4 Byte | Modellparameter |
| Gradient | 4 Byte | nach Backward vorhanden |
| Adam erster Moment | 4 Byte | m |
| Adam zweiter Moment | 4 Byte | v |
| Summe ohne Aktivierungen | ca. 16 Byte | bei reinem FP32-AdamW |

Mixed-Precision-Training kann FP16/bfloat16-Gewichte und Aktivierungen verwenden, behaelt aber oft FP32-Mastergewichte oder Optimizerzustaende. Aktivierungen haengen stark von B, T, L, d und Attentionimplementierung ab. Sie koennen den Parameterspeicher uebertreffen.

## 8.3 FLOPs-Naeherung

Fuer dichte Transformer wird der Trainingscompute oft grob mit etwa 6*N*D FLOPs abgeschaetzt, wobei N die Parameterzahl und D die Zahl verarbeiteter Trainingstokens ist. Der Faktor ist eine Naeherung fuer Forward plus Backward und ignoriert Details wie Attentionquadratik, Embeddinglookup und Hardwareineffizienzen. Fuer kleine Modelle und lange T kann die T^2-Komponente relativ wichtiger werden.

## 8.4 Breite gegen Tiefe

Mehr Breite vergroessert Matrizen quadratisch; mehr Tiefe linear in L. Tiefe schafft mehr aufeinanderfolgende Verarbeitungsschritte, Breite mehr parallele Merkmalskapazitaet. Es gibt keine universelle optimale Form. Fuer eigene Experimente sollte eine Basisfamilie festgelegt und nur eine Dimension gleichzeitig veraendert werden, sonst laesst sich die Ursache einer Aenderung nicht bestimmen.

## 8.5 Skalierungsgesetze

Empirische Skalierungsgesetze zeigen, dass Loss ueber grosse Bereiche regelmaessig mit Modellgroesse, Daten und Compute sinkt [8]. Die Chinchilla-Arbeit zeigte zudem, dass unter einem festen Compute-Budget zu grosse Modelle mit zu wenigen Tokens untertrainiert sein koennen [9]. Fuer kleine Modelle folgt daraus keine magische exakte Tokenzahl, aber eine klare Regel: Nur Parameter zu erhoehen reicht nicht; Tokenbudget und Datenqualitaet muessen mitwachsen.

# 9. Der Trainingsprozess als Engineering-System

Optimizer, Lernrate, Stabilitaet und Checkpoints

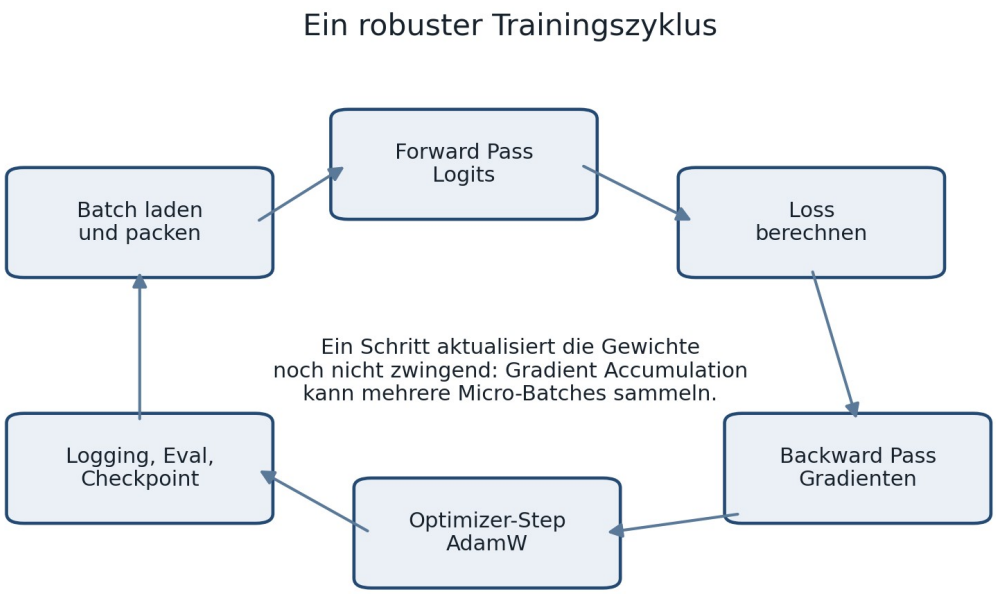



*Abbildung 4: Wiederholter Trainingszyklus mit Evaluation und Sicherung*

## 9.1 Batches, Micro-Batches und Gradient Accumulation

Ein globaler Batch kann aus mehreren Micro-Batches bestehen. Bei Gradient Accumulation wird der Loss jedes Micro-Batches durch die Anzahl Accumulationsschritte geteilt, Backward aufgerufen und erst danach ein Optimizer-Step ausgefuehrt. Das simuliert groessere Batches bei begrenztem Speicher. Wichtig ist, Metriken in Tokens pro Optimizer-Step und nicht nur in Iterationen zu dokumentieren.

## 9.2 AdamW

AdamW fuehrt gleitende Mittelwerte des Gradienten und seines Quadrats. Dadurch erhaelt jeder Parameter eine adaptiv skalierte Schrittweite. Weight Decay wird getrennt vom Gradientenupdate angewendet, was sich von einer blossen L2-Strafe in Adam unterscheidet [4].

```
m_t = beta1*m_(t-1) + (1-beta1)*g_t
v_t = beta2*v_(t-1) + (1-beta2)*g_t^2
theta <- theta - lr * m_hat/(sqrt(v_hat)+eps)
theta <- theta - lr * weight_decay * theta
```

Norm- und Biasparameter werden haeufig vom Weight Decay ausgenommen. Ob das fuer ein kleines Modell optimal ist, sollte experimentell, aber konsistent getestet werden.

## 9.3 Lernratenplan

Ein verbreiteter Plan beginnt mit Warmup: Die Lernrate steigt ueber die ersten Schritte von nahe null auf den Spitzenwert. Danach sinkt sie per Cosine Decay oder linear. Warmup schuetzt die fruehe Phase, in der Adam-Momente und Aktivierungsstatistiken noch nicht stabil sind. Eine zu hohe Lernrate kann den Loss explosionsartig machen; eine zu niedrige trainiert scheinbar stabil, lernt aber kaum.

## 9.4 Gradient Clipping

Global Norm Clipping skaliert alle Gradienten gemeinsam, wenn ihre Gesamtnorm einen Grenzwert ueberschreitet. Es ist eine Sicherung gegen seltene extreme Schritte, kein Ersatz fuer korrekte Lernrate oder stabile Architektur. Der Anteil geclippter Schritte sollte geloggt werden: Dauerhaftes Clipping zeigt ein tieferes Problem.

## 9.5 Dropout und Regularisierung

Dropout deaktiviert zufaellig Aktivierungen im Training. Sehr grosse LLM-Pretrainings nutzen teilweise wenig oder keinen Dropout, weil Datenmenge und Optimierung anders wirken; kleine Datensaetze und kleine Modelle koennen davon profitieren. Zu viel Dropout verhindert jedoch, dass ein 5M- oder 20M-Modell seine ohnehin begrenzte Kapazitaet nutzt.

## 9.6 Mixed Precision und Loss Scaling

Bei FP16 koennen kleine Gradienten unterlaufen. Dynamic Loss Scaling multipliziert den Loss vor Backward, teilt die Gradienten spaeter zurueck und verwirft Schritte bei Overflow. bfloat16 benoetigt wegen seines Exponentenbereichs oft kein Loss Scaling. Auf Hardware ohne native Unterstuetzung kann Mixed Precision langsamer statt schneller sein.

## 9.7 Checkpoints und Reproduzierbarkeit

- Modellgewichte und Architekturkonfiguration gemeinsam speichern.
- Fuer Fortsetzung: Optimizer, Scheduler, globaler Schritt, verarbeitete Tokens und RNG-Zustaende sichern.
- Tokenizerdateien versionieren und Hash notieren.
- Dataset-Mix, Filterversion und Git-Commit protokollieren.
- Atomar speichern: zuerst temporaere Datei, dann umbenennen. Regelmaessig einen bekannten guten Checkpoint behalten.

**Fehlerquelle:** Ein Checkpoint, der nur Gewichte enthaelt, reicht fuer Inferenz. Fuer exakte Trainingsfortsetzung fehlen jedoch Optimizer- und Zufallszustaende. Der Loss kann nach einem Neustart deshalb sichtbar anders verlaufen.

## 9.8 Verteiltes Training

Data Parallelism repliziert das Modell und teilt Batches. Tensor Parallelism teilt Matrizen innerhalb eines Layers. Pipeline Parallelism verteilt Layergruppen. ZeRO/FSDP shardet Parameter, Gradienten und Optimizerzustand. Fuer ein 20M-Modell ist diese Komplexitaet meist unnoetig; saubere Ein-GPU- oder CPU-Implementierung ist fuer Architekturexperimente wertvoller.

# 10. Evaluation: Woran man erkennt, ob das Modell bes wird

Loss, Aufgaben, Robustheit und Ablationen

Ein sinkender Trainingsloss beweist nur, dass das Modell seine Trainingsdaten besser vorhersagt. Ein Forschungsmodell braucht mehrere Ebenen der Evaluation, weil Verbesserungen auf einer Kennzahl andere Faehigkeiten verschlechtern koennen.

## 10.1 Validation Loss

Validation Loss ist die frueheste Warnung fuer Overfitting. Wenn Trainingsloss weiter sinkt und Validation Loss steigt, passt sich das Modell zu stark an Trainingsdetails an. Bei sehr kleinen Korpora kann ein Modell ganze Passagen memorieren, waehrend freie Dialogqualitaet kaum steigt.

## 10.2 Aufgabenbasierte Evaluation

| Kategorie | Beispielmessung | Warum |
|---|---|---|
| Sprachmodellierung | Loss/Perplexity je Datenquelle | Zeigt Verteilungsluecken und Mixprobleme |
| Fakten | geschlossene Fragen mit automatisch pruefbaren Antworten | Trennt Faktentreffer von fluessigem Stil |
| Reasoning | Mathe, Logik, algorithmische Aufgaben | Misst Mehrschrittverfahren |
| Code | Unit Tests statt Textaehnlichkeit | Ausfuehrbare Korrektheit |
| Dialog | Rubrikbewertung, Rollenstabilitaet, Befolgung | Post-Training-Qualitaet |
| Robustheit | Paraphrasen, Stoerungen, laengere Kontexte | Verhindert Benchmark-Ueberanpassung |

## 10.3 Generationsstichproben

Qualitative Chats sind unverzichtbar, aber leicht zu ueberinterpretieren. Testprompts muessen versioniert werden; Temperatur und Seed werden festgehalten. Einzelne spektakulaere Ausgaben sind keine belastbare Verbesserung. Eine Vergleichsmatrix mit mehreren Seeds und Blindbewertung ist besser.

## 10.4 Ablationsstudien

Bei einer Ablation wird genau eine Komponente entfernt oder geaendert: zum Beispiel RoPE gegen gelernte Positions-Embeddings, SwiGLU gegen GELU, Replay an gegen aus. Alle anderen Faktoren, einschliesslich Trainings-Tokens und Compute, bleiben moeglichst gleich. Ohne Ablation ist bei kuriosen Architekturen oft unklar, ob die Idee oder nur mehr Parameter geholfen haben.

## 10.5 Datenkontamination und Leckage

Ein Modell kann Benchmarkloesungen memoriert haben. Exakte und fuzzy Suche im Trainingskorpus, zeitlich spaetere Tests und selbst generierte Varianten reduzieren das Risiko. Fuer eigene HTI-Versionen sollte ein fester geheimer Testsatz nie in synthetische Trainingsgeneratoren eingespeist werden.

# 11. Vom Base Model zum Chatmodell

Continued Pretraining, SFT und Praeferenzoptimierung

Pretraining und Fine-Tuning sind verschiedene Phasen, aber beide veraendern Gewichte durch Gradienten. Der Begriff Fine-Tuning sagt nichts ueber die Modellgroesse aus. Ein 5-Millionen-Modell kann vortrainiert oder feinabgestimmt sein; entscheidend ist, auf welchem Ausgangscheckpoint und mit welchem Ziel weitertrainiert wurde.

## 11.1 Continued Pretraining / Domain Adaptation

Ein Base Model kann auf zusaetzlichen Fachtexten mit demselben Next-Token-Ziel weitertrainiert werden. Das erweitert oder verschiebt die interne Verteilung. Risiken sind Overfitting und Vergessen allgemeiner Faehigkeiten. Ein gemischter Replay-Anteil allgemeiner Daten und eine kleinere Lernrate sind typische Gegenmittel.

## 11.2 Supervised Fine-Tuning (SFT)

SFT trainiert auf vollstaendigen Zielantworten. Bei Chatdaten wird der Loss oft nur auf Assistant-Tokens berechnet; System- und Nutzertext dienen als Kontext. Sonst lernt das Modell auch, Nutzerfragen nachzuahmen. Gute SFT-Daten zeigen nicht nur Stil, sondern korrekte Aufgabenausfuehrung, angemessene Laenge, Unsicherheit und Formatkonstanz.

```
labels = input_ids.clone()
labels[system_and_user_positions] = IGNORE_INDEX
loss = CE(model(input_ids), labels)
```

## 11.3 Praeferenzdaten und DPO

Praeferenzdaten enthalten zu demselben Prompt eine bevorzugte und eine abgelehnte Antwort. Direct Preference Optimization optimiert direkt, dass das Modell die bevorzugte Antwort relativ zur abgelehnten und relativ zu einem Referenzmodell hoeher bewertet [10]. Das ist einfacher als eine volle RLHF-Pipeline mit separatem Reward Model, ersetzt aber nicht die Notwendigkeit guter und konsistenter Praeferenzdaten.

## 11.4 RLHF und verifizierbare Rewards

Bei RLHF wird typischerweise ein Reward Model aus menschlichen Vergleichen gelernt und die Policy anschliessend gegen diesen Reward unter einer KL-Beschraenkung optimiert. Bei Mathematik, Code oder formalen Aufgaben koennen Rewards direkt aus korrekten Endergebnissen oder Tests kommen. Das reduziert subjektive Labelkosten, kann aber Reward Hacking erzeugen: Das Modell findet Wege, die Messung zu erfuellen, ohne die beabsichtigte Loesung zu lernen.

## 11.5 Chat-Template und Systemverhalten

Das Systemprompt wirkt nur innerhalb dessen, was das Modell im Training gelernt hat. Ein Base Model wird durch eine Systemnachricht nicht automatisch zu einem verlaesslichen Assistenten. Umgekehrt kann ein gut trainiertes Chatmodell durch falsche Rollen-Tokens oder fehlendes Endtoken in Wiederholungsschleifen geraten.

## 11.6 Catastrophic Forgetting

Beim sequenziellen Training auf neuen Daten koennen alte Faehigkeiten sinken. Dieses catastrophic forgetting ist fuer fortlaufend lernende Systeme zentral. Replay alter Daten [22], parameterbasierte Regularisierung wie Elastic Weight Consolidation [23], Adapter, Distillation und begrenzte Lernraten sind moegliche Bausteine. Keine Methode macht unkontrolliertes Online-Lernen automatisch sicher.

# 12. Inferenz: Wie aus Logits eine Antwort entsteht

Prefill, KV-Cache und Sampling

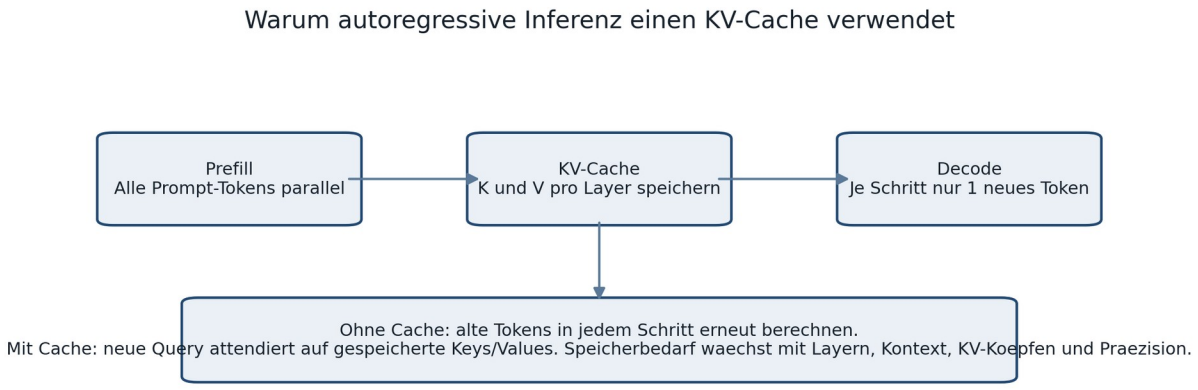



*Abbildung 5: Prefill und tokenweises Decoding mit KV-Cache*

## 12.1 Promptserialisierung

Vor dem Forward Pass wird der Dialog in das exakt trainierte Chatformat serialisiert und tokenisiert. Danach laeuft der gesamte Prompt einmal durch das Modell (Prefill). Das Modell erzeugt Logits fuer die naechste Position. Nach Auswahl eines Tokens wird dieses angehaengt und der Prozess wiederholt.

## 12.2 KV-Cache

Im Prefill werden Keys und Values fuer alle Promptpositionen in jedem Layer berechnet. Beim Decode muessen sie nicht erneut berechnet werden. Nur Query, Key und Value des neuen Tokens entstehen; die neue Query attendiert auf den gespeicherten Cache. Dadurch bleibt die Generierung dennoch sequentiell, aber deutlich schneller.

```
KV_cache_memory ≈ layers * context * kv_heads * head_dim * 2(K,V) * bytes_per_value
```

## 12.3 Temperatur, Top-k und Top-p

| Verfahren | Operation | Effekt |
|---|---|---|
| Greedy | argmax der Logits | Deterministisch, kann frueh in schlechte Pfade geraten |
| Temperatur | Logits durch T teilen | T<1 schaerfer, T>1 flacher |
| Top-k | nur k hoechste Tokens behalten | Begrenzt Ausreisser des langen Verteilungsschwanzes |
| Top-p | kleinste Menge mit kumulierter Wahrscheinlichkeit p | Dynamische Kandidatenzahl |
| Repetition Penalty | bereits verwendete Tokens modifizieren | Kann Schleifen reduzieren, aber Inhalt verzerren |

Samplingparameter veraendern keine Gewichte und machen das Modell nicht intelligenter. Sie waehlen nur anders aus der gelernten Verteilung. Fuer Reasoningaufgaben kann mehr Diversitaet bei mehreren Versuchen helfen; fuer einzelne praezise Antworten ist zu hohe Temperatur oft schaedlich.

## 12.4 Stopbedingungen

Die Generierung endet bei EOS, einem Stop-Token, einer Stopsequenz oder einem maximalen Tokenlimit. Stopsequenzen muessen tokenisierungsrobust behandelt werden. Ein Zeichenstring kann in mehreren Tokenzerlegungen vorkommen.

## 12.5 Halluzination als Verteilungsproblem

Das Modell optimiert Plausibilitaet des naechsten Tokens, nicht Wahrheit. Wenn der Kontext eine Antwortform erwartet, kann eine fluessige, aber falsche Fortsetzung hohe Wahrscheinlichkeit haben. Retrieval, Tools, Quellenpruefung, Kalibrierung und Training mit „unbekannt“ koennen helfen; keine reine Samplingeinstellung beseitigt das Grundproblem.

## 12.6 Quantisierung

Bei Quantisierung werden Gewichte oder Aktivierungen mit weniger Bits dargestellt, zum Beispiel INT8 oder INT4. Das reduziert Speicher und oft Bandbreite, fuehrt aber zu Approximationsfehlern. Post-Training Quantization ist einfach; Quantization-Aware Training kann bessere Qualitaet liefern. Fuer sehr kleine Modelle kann die Runtime-Overheadstruktur wichtiger sein als die theoretische Bitzahl.

# 13. Ein kompletter 20-Millionen-Parameter-Blueprint

Eine Standardbasis fuer HTI-2-artige Experimente

Die folgende Konfiguration ist kein behauptetes Optimum. Sie ist eine saubere, nachvollziehbare Ausgangsbasis fuer ein ungefaehr 20-Millionen-Parameter-Modell, bevor kuriose Architekturen getestet werden. Ziel ist ein Modell, dessen Fehlerquellen verstanden und dessen Varianten fair verglichen werden koennen.

## 13.1 Konfiguration

| Parameter | Vorschlag | Begruendung |
|---|---|---|
| Tokenizer | BPE/SentencePiece, V=16 000 | Begrenzt Embeddingkosten, bleibt multilingual/byte-robust |
| Kontext | T=512 oder 1024 | Auf CPU realistisch; 1024 erhoeht Attentionkosten deutlich |
| d_model | 384 | Ausreichende Breite bei kleinem Budget |
| Layer | 8 | Mehrere sequentielle Verarbeitungsschritte |
| Attention-Koepfe | 6 x 64 | d_head=64, sauber teilbar |
| KV-Koepfe | 6 zunaechst | Klassische MHA als Referenz; spaeter GQA-Ablation |
| d_ff | 1024 | SwiGLU, ungefaehr 2,67*d |
| Norm | Pre-RMSNorm | Moderner stabiler Standard |
| Position | RoPE | Weit verbreiteter Decoderstandard |
| Bias | aus in grossen Matrizen | Spart wenig, vereinfacht moderne Baseline |
| Weight tying | an | Spart separate Outputmatrix |

## 13.2 Parameterrechnung

```
Embedding: 16 000 * 384 = 6 144 000
Pro Layer Attention: 4 * 384^2 = 589 824
Pro Layer SwiGLU: 3 * 384 * 1024 = 1 179 648
8 Layer: 8 * 1 769 472 = 14 155 776
Grob gesamt: 20 299 776 + Normskalen
```

Damit liegt die Basis sehr nah an 20,3 Millionen Parametern. Eine separate Outputmatrix wuerde weitere 6,14 Millionen hinzufuegen. GQA wuerde K/V-Projektionen verkleinern und etwas Budget fuer andere Teile freigeben.

## 13.3 Ordnerstruktur

```
hti2/
  configs/base_20m.yaml
  tokenizer/
  data/raw/
  data/processed/
  src/model.py
  src/attention.py
  src/data.py
  src/train.py
  src/eval.py
  src/generate.py
  checkpoints/
  logs/
  tests/
```

## 13.4 Implementierungsreihenfolge

1. Tokenizer trainieren, versionieren und Roundtrip-Tests bestehen.
2. Dataset in Token-IDs umwandeln, Shards schreiben und Tokenstatistiken dokumentieren.
3. Ein einzelnes Transformerblock-Modul mit Formtests und Causal-Mask-Test bauen.
4. Modell zusammensetzen und Parameterzahl automatisch ausgeben.
5. Overfit-Test auf einem winzigen Batch: Der Loss muss fast auf null fallen koennen.
6. Kurzer Lauf auf kleinem Korpus; NaN-, Gradient- und Checkpoint-Tests.
7. Erst danach langer Pretraining-Lauf.
8. Base-Checkpoint einfrieren und separat Chat-SFT entwickeln.

## 13.5 Minimaler Forward-Pseudocode

```
def forward(ids, targets=None):
    x = token_embedding(ids)              # [B,T,d]
    cos, sin = rope_cache(T)
    for block in blocks:
        x = block(x, cos, sin, causal=True)
    x = final_norm(x)
    logits = x @ token_embedding.weight.T
    if targets is None:
        return logits
    return cross_entropy(logits, targets, ignore_index=PAD_OR_IGNORE)
```

## 13.6 Trainingsstufen

| Stufe | Ziel | Abbruchkriterium |
|---|---|---|
| Smoke Test | Codepfad funktioniert | 100-1000 Schritte ohne Fehler; Checkpoint laedt |
| Tiny Overfit | Lernfaehigkeit beweisen | kleiner Batch nahezu memorisiert |
| Pilot | Hyperparameter und Daten pruefen | Validation Loss sinkt; Generation nicht kollabiert |
| Pretraining | allgemeines Base Model | Tokenbudget erreicht; beste Validation sichern |
| SFT | Chatformat und Aufgabenbefolgung | Rubrik + Held-out-Dialoge |
| Optional Preference/Reasoning | gezielte Verhaltensverbesserung | separate Benchmarkverbesserung ohne Regression |

## 13.7 Was fuer faire HTI-Versionen konstant bleiben sollte

- Gleicher Tokenizer oder klar dokumentierter Tokenizerwechsel.
- Gleiche Datenmischung und moeglichst gleiches Trainingstokenbudget.
- Gleiche Evaluation, Seeds und Samplingparameter.
- Compute beziehungsweise FLOPs dokumentieren, nicht nur Parameterzahl.
- Bei Architekturtests mindestens eine Baseline mit gleicher oder aehnlicher aktiver Parameterzahl.

# 14. Fehlerdiagnose bei kleinen Sprachmodellen

Was typische Symptome wirklich bedeuten koennen

| Symptom | Wahrscheinliche Ursachen | Pruefung |
|---|---|---|
| Loss wird NaN | Lernrate, FP16-Overflow, falsche Maske, instabile Norm | Aktivierungen/Gradienten, FP32-Test, kleineren LR |
| Loss sinkt kaum | Daten-/Labelshift falsch, LR zu klein, Parameter eingefroren | Tiny-Overfit, Gradienten ungleich null |
| Modell wiederholt Text | fehlendes EOS, schlechte Dialogdaten, Sampling, Datenkopien | EOS-Rate, greedy vs sampling, Deduplikation |
| Gute Trainingsfortsetzung, schlechte Chats | Base Model ohne SFT oder falsches Template | SFT-Lossmaske und Rollen-Tokens |
| Starke Halluzination | kleines Wissen, unsichere Verteilung, kein Grounding | geschlossene Faktenbenchmarks, Retrievaltest |
| Antworten werden monoton | zu enge SFT-Daten, niedrige Temperatur, Mode Collapse | Datenvielfalt und Baselinevergleich |
| Neue Faehigkeit, alte weg | catastrophic forgetting | Vorher-/Nachher-Eval, Replay |

## 14.1 Der Tiny-Overfit-Test

Ein Modell sollte einen winzigen festen Batch memorieren koennen. Scheitert das, ist ein langer Lauf sinnlos. Der Test entdeckt fehlerhafte Labels, deaktivierte Gradienten, falsche Masken, Optimizerprobleme und unpassende Lernraten. Dropout sollte dafuer ausgeschaltet oder stark reduziert werden.

## 14.2 Attention-Maskentest

Veraendere ein zukuenftiges Token und pruefe, ob sich Logits frueherer Positionen aendern. Bei korrekter Causal Mask duerfen sie sich nicht aendern. Dieser Test ist staerker als nur eine visuelle Maskenmatrix, weil er den kompletten Rechenpfad prueft.

## 14.3 Checkpoint-Roundtrip

Mit eval-Modus und identischer Eingabe muessen Logits vor dem Speichern und nach dem Laden bis auf numerische Toleranz identisch sein. Fuer Trainingsfortsetzung sollte zusaetzlich ein Referenzschritt vor und nach dem Resume verglichen werden.

## 14.4 Datenpipeline testen

Zufaellige Samples nach Tokenisierung dekodieren. Dokumentgrenzen, Sondertokens, Codeeinrueckung und Sprachen manuell ansehen. Statistiken allein erkennen keine semantisch zerstoerte Aufbereitung.

# 15. Was bei Sprachmodellen als "Thinking" bezeichnet wird

Nicht eine Technik, sondern mehrere Ebenen von zusaetzlicher Berechnung

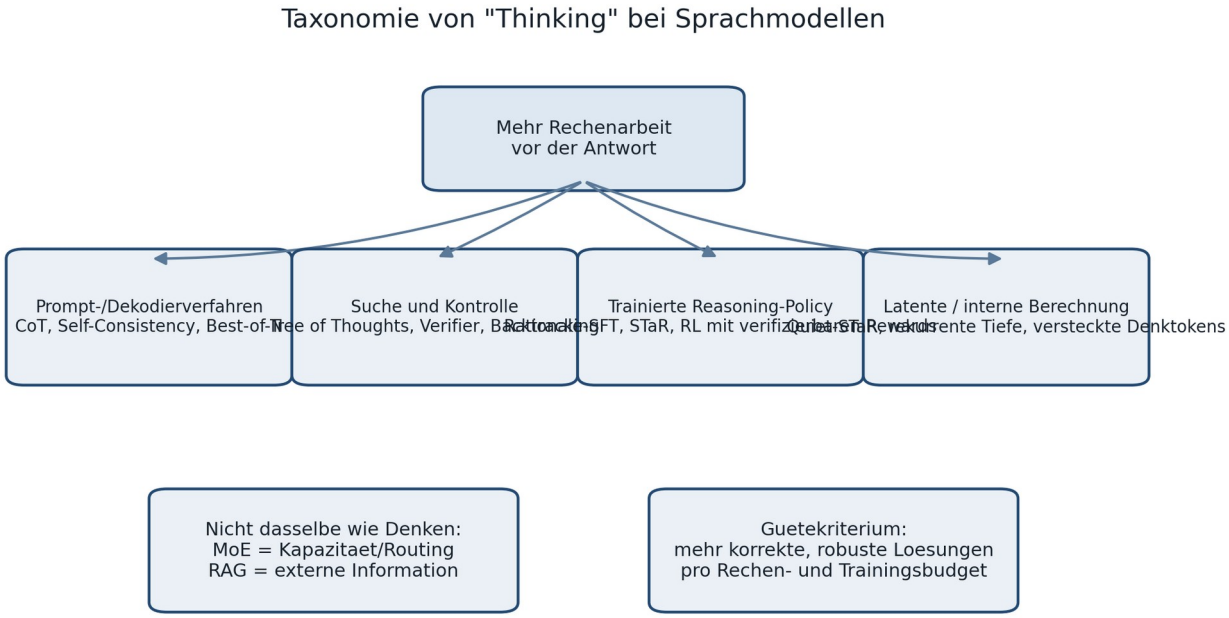



*Abbildung 6: Mehrere technisch verschiedene Bedeutungen von Thinking*

Der Begriff Thinking wird in der Praxis fuer unterschiedliche Mechanismen verwendet. Manche veraendern nur den Prompt oder die Dekodierung, manche fuegen Suche oder Verifier hinzu, manche trainieren das Modell auf Reasoning-Spuren, und manche versuchen, interne zusaetzliche Rechenschritte zu lernen. Diese Ebenen muessen getrennt evaluiert werden.

## 15.1 Sichtbare Chain of Thought (CoT)

Bei Chain-of-Thought-Prompting erzeugt das Modell Zwischenschritte vor der Endantwort. Das kann komplexe Aufgaben in lokal einfachere Tokenentscheidungen zerlegen und wurde bei ausreichend grossen Modellen auf Mathematik, Symbolik und Commonsense verbessert beobachtet [11]. Die sichtbare Begruendung ist jedoch nicht garantiert eine treue Beschreibung der internen Ursache. Ein korrekt klingender Gedankengang kann nachtraeglich rationalisiert oder trotzdem falsch sein.

## 15.2 Scratchpad als Arbeitsflaeche

Ein Scratchpad ist ein Bereich von Tokens, in dem Zwischenwerte, Teilziele oder Plaene abgelegt werden. Weil Transformer nur ueber den Kontextzustand verfuegen, verlaengert ein Scratchpad die explizite Rechenkette. Bei kleinen Modellen muss das Format sehr klar trainiert werden; sonst verbraucht es Tokens fuer fluessige, aber nutzlose Selbstkommentare.

## 15.3 Self-Consistency und Best-of-N

Self-Consistency sampelt mehrere unterschiedliche Reasoningpfade und waehlt die haeufigste Endantwort [12]. Best-of-N bewertet mehrere Kandidaten mit einem Verifier oder Reward Model. Beide Verfahren nutzen Inferenzcompute statt neuer Parameter. Sie helfen nur, wenn die korrekte Loesung mit nennenswerter Wahrscheinlichkeit im Kandidatenraum vorkommt und die Auswahlregel sie erkennen kann.

## 15.4 Tree of Thoughts und Suche

Tree of Thoughts behandelt Zwischenzustaende als Suchknoten. Das System erzeugt mehrere Fortsetzungen, bewertet sie, expandiert gute Zweige und kann zurueckgehen [13]. Der Gewinn stammt aus expliziter Suche und Selbstbewertung, nicht aus einem einzelnen normalen Forward Pass. Kosten steigen mit Verzweigungsfaktor, Tiefe und Bewertungsaufrufen.

## 15.5 ReAct und Werkzeugnutzung

ReAct verschraenkt Reasoningtexte mit Aktionen und Beobachtungen [14]. Eine Aktion kann eine Suche, einen Rechner oder eine Umgebung aufrufen; die Beobachtung wird wieder Teil des Kontexts. Damit kann das System fehlende Information beschaffen und Plaene aktualisieren. Technisch ist das ein Modell-plus-Umgebung-System. Es ist nicht dasselbe wie internes Wissen oder dauerhaftes Lernen.

## 15.6 Test-Time Compute

Thinking-Modelle investieren vor der Antwort mehr Rechenbudget: laengere Spuren, mehrere Samples, Suche, Verifikation oder iterative Korrektur. Der relevante Vergleich ist nicht nur Genauigkeit, sondern Genauigkeit pro Token, Latenz und Energie. Ein Verfahren, das zehnmal mehr Tokens fuer ein Prozent Gewinn braucht, kann fuer Forschung interessant, aber als Standard unpraktisch sein.

# 16. Trainierte Reasoningfaehigkeiten

Rationale-SFT, STaR, Verifier und Reinforcement Learning

Prompting kann nur Faehigkeiten abrufen, die im Modell zumindest teilweise vorhanden sind. Trainierte Reasoningverfahren veraendern die Gewichte so, dass nuetzliche Zwischenberechnungen wahrscheinlicher werden.

## 16.1 Rationale Supervised Fine-Tuning

Das Modell wird auf Aufgaben mit Zwischenschritten und Endantworten trainiert. Datenqualitaet ist kritisch: Falsche oder unnoetig lange Rationales lehren schlechte Verfahren. Eine Mischung aus kurzen direkten Antworten und ausfuehrlichen Spuren kann verhindern, dass das Modell jede triviale Frage mit langem Monolog beantwortet.

## 16.2 STaR: Bootstrapping mit eigenen Rationales

STaR erzeugt Rationales fuer ungelabelte oder nur mit Endantwort gelabelte Aufgaben, behaelt erfolgreiche Spuren, trainiert darauf weiter und wiederholt den Zyklus [19]. Bei falschen Antworten kann die korrekte Endantwort als Hinweis dienen, um eine bessere Rationale zu generieren. Das ist ein konkreter Selbstverbesserungsloop, aber nur dort verlaesslich, wo Erfolg objektiv geprueft werden kann.

## 16.3 Process- und Outcome-Verifier

Ein Outcome-Verifier bewertet nur die Endantwort. Ein Process-Verifier bewertet Zwischenschritte. Process-Supervision kann fruehe Fehler erkennen, benoetigt aber detailliertere Labels und kann auf eine bestimmte Darstellungsform ueberfitten. Automatische Verifier sind besonders stark bei Code-Tests, algebraischen Identitaeten oder formalen Beweisen.

## 16.4 Reinforcement Learning mit verifizierbaren Rewards

Bei RLVR werden Samples erzeugt und anhand automatisch pruefbarer Ergebnisse belohnt. DeepSeek-R1 zeigte einen mehrstufigen Ansatz mit RL, Cold-Start-Daten und Distillation; R1-Zero untersuchte Reasoningentwicklung ohne vorheriges SFT, hatte jedoch Lesbarkeits- und Sprachmischungsprobleme [17]. Die Lehre fuer kleine Modelle ist nicht „RL erzeugt aus nichts Intelligenz“, sondern: Ein brauchbares Base Model, explorierbare Aufgaben und robuste Rewards sind Voraussetzung.

## 16.5 Reward Hacking und Laengenbias

Wenn lange Antworten im Datensatz haeufig besser bewertet werden, lernt das Modell moeglicherweise Laenge statt Reasoning. Wenn ein Checker Luecken hat, optimiert das Modell die Luecke. Deshalb braucht jede RL-Reasoningpipeline Gegenmetriken: Korrektheit, Laenge, Konsistenz, Format, Diversitaet, Ausnutzung des Verifiers und Regression auf allgemeinen Aufgaben.

## 16.6 Distillation

Ein staerkeres Modell kann Reasoningspuren oder Loesungen erzeugen, auf denen ein kleineres Modell trainiert wird. Distillation uebertraegt Verhaltensmuster, aber auch Fehler und Stil. Fuer HTI-Modelle ist ein guter Ansatz, synthetische Daten nur dann zu verwenden, wenn Endergebnisse automatisch geprueft oder von mehreren unabhaengigen Verfahren bestaetigt werden.

# 17. Latentes und internes Thinking

Wenn nicht jeder Rechenschritt als normaler Antworttext erscheinen soll

Sichtbare CoT ist teuer und kann unnoetige oder sensible Zwischentexte erzeugen. Daher untersucht Forschung, zusaetzliche Berechnung intern oder in speziellen Tokens auszufuehren.

## 17.1 Quiet-STaR

Quiet-STaR trainiert ein Modell, an vielen Tokenpositionen interne Rationales zu sampeln und zu lernen, welche Gedanken die Vorhersage spaeterer Tokens verbessern [18]. Gedanken werden durch spezielle Start-/Endtokens markiert und ihre Nuetzlichkeit ueber den Vorhersagegewinn trainiert. Das ist konzeptionell naeher an „vor dem Sprechen denken“ als normales CoT-Prompting, aber rechenintensiv und implementierungstechnisch deutlich komplexer.

## 17.2 Latente Tokens

Eine Architektur kann spezielle Denktokens erzeugen, die nicht an den Nutzer ausgegeben werden. Sie koennen dennoch im normalen diskreten Tokenraum liegen. Eine weitergehende Idee ist, kontinuierliche versteckte Zustaende rekurrent zurueckzufuehren, ohne sie in Woerter zu dekodieren. Das erfordert ein klares Trainingssignal; sonst lernt das Netz nur zusaetzliche, nicht informative Rechenschritte.

## 17.3 Rekurrente Tiefe und Pondering

Statt L verschiedene Layer einmal zu durchlaufen, kann ein Block mehrfach mit geteilten Gewichten angewendet werden. Dadurch steigt effektive Rechentiefe bei weniger Parametern. Ein Haltemechanismus kann entscheiden, wie viele Iterationen eine Eingabe benoetigt. Die Herausforderung ist stabile Optimierung und die Frage, ob zusaetzliche Iterationen wirklich neue Information verarbeiten oder nur den Zustand verwischen.

## 17.4 Hidden Reasoning ist schwer zu beaufsichtigen

Wenn Zwischenschritte nicht lesbar sind, koennen sie nicht direkt manuell kontrolliert werden. Man braucht indirekte Tests: Endkorrektheit, kausale Eingriffe, lineare Probes, Ablation der Denkschritte und Verifikation, dass mehr interne Schritte auf schwierigen Aufgaben helfen und auf einfachen nicht nur Zeit verschwenden.

**Forschungsregel:** Ein internes Thinkingmodul gilt nicht als erfolgreich, weil Aktivierungen interessant aussehen. Es muss gegen eine compute-gematchte Baseline gewinnen: gleiche aktive Parameter, aehnliche FLOPs und identische Daten.

# 18. Mixture of Experts (MoE)

Mehr Gesamtparameter bei sparsamer Aktivierung

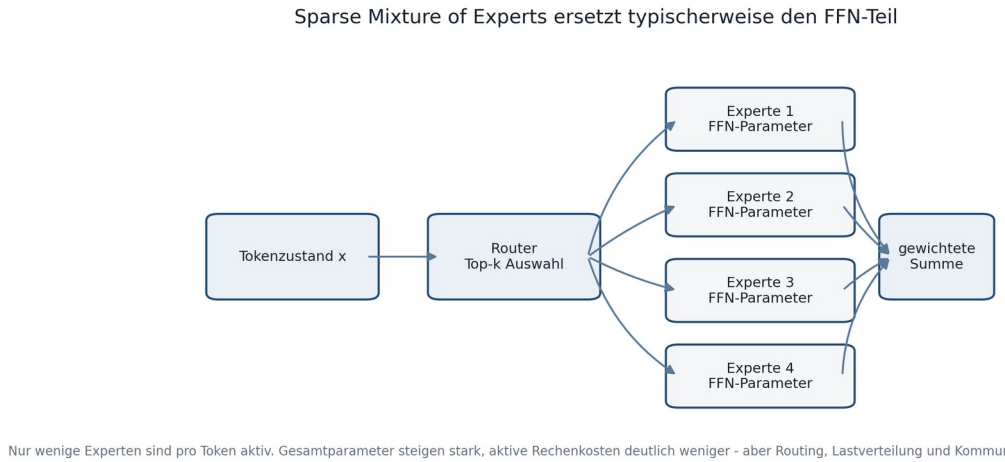



*Abbildung 7: Routing eines Tokens zu wenigen FFN-Experten*

Der vom Nutzer gemeinte Begriff „Model of Experts“ wird in der Fachliteratur ueblicherweise Mixture of Experts genannt. MoE ist keine Thinkingmethode. Es ist eine Architekturtechnik, um die Gesamtparameterzahl zu erhoehen, waehrend pro Token nur ein kleiner Teil aktiv ist.

## 18.1 Router und Experten

Ein Router berechnet fuer jeden Token Scores ueber E Experten. Top-k waehlt beispielsweise zwei Experten. Jeder Experte ist meist ein eigenes FFN. Die Ausgaben werden mit Routergewichten kombiniert. Mixtral 8x7B verwendet acht FFN-Experten und aktiviert pro Token zwei [16]. Switch Transformer vereinfachte Routing auf einen aktiven Experten und untersuchte Training sehr grosser sparse Modelle [15].

```
router_logits = x W_router
selected = top_k(router_logits, k)
y = Σ_{e in selected} softmax_weight_e * Expert_e(x)
```

## 18.2 Warum Experten sich spezialisieren koennen

Unterschiedliche Tokens und Kontexte erzeugen unterschiedliche Routerentscheidungen. Dadurch erhalten Experten unterschiedliche Gradientenverteilungen und koennen sich funktional spezialisieren. Die Spezialisierung ist jedoch nicht garantiert sauber semantisch; Experten koennen nach Position, Sprache, Format oder statistischen Eigenheiten aufgeteilt werden.

## 18.3 Load Balancing

Ohne Zusatzloss kann der Router fast alle Tokens zu wenigen Experten schicken. Das ueberlastet diese und laesst andere untrainiert. Ein Load-Balancing-Loss foerdert gleichmaessigere Nutzung. Capacity Limits bestimmen, wie viele Tokens ein Experte pro Batch annimmt; ueberschuessige Tokens werden gedroppt oder umgeleitet.

## 18.4 Kommunikationskosten

Auf mehreren Geraeten muessen Tokenaktivierungen zu den Geraeten der gewaehlten Experten gesendet und zurueckgebracht werden. All-to-all-Kommunikation kann den theoretischen Computevorteil dominieren. Auf einem einzelnen kleinen Rechner passt ein MoE eventuell als Experiment, ist aber oft langsamer als ein dichter compute-gematchter Transformer.

## 18.5 MoE-Experiment fuer kleine HTI-Modelle

Eine faire kleine Studie ersetzt in einigen Layern das SwiGLU-FFN durch vier Experten und aktiviert top-1 oder top-2. Vergleiche: gleiche aktive FFN-Parameter pro Token, gleiches Tokenbudget, Router-Entropie, Expertenauslastung, Loss, Durchsatz und Aufgabenqualitaet. Nur die Gesamtparameterzahl zu nennen waere irrefuehrend, weil inaktive Parameter keinen direkten Compute im aktuellen Token erhalten.

# 19. "Schlafartiges" Selbstlernen als kontrolliertes Forschungsprogramm

Replay, Verifikation, Konsolidierung und Rueckrollbarkeit

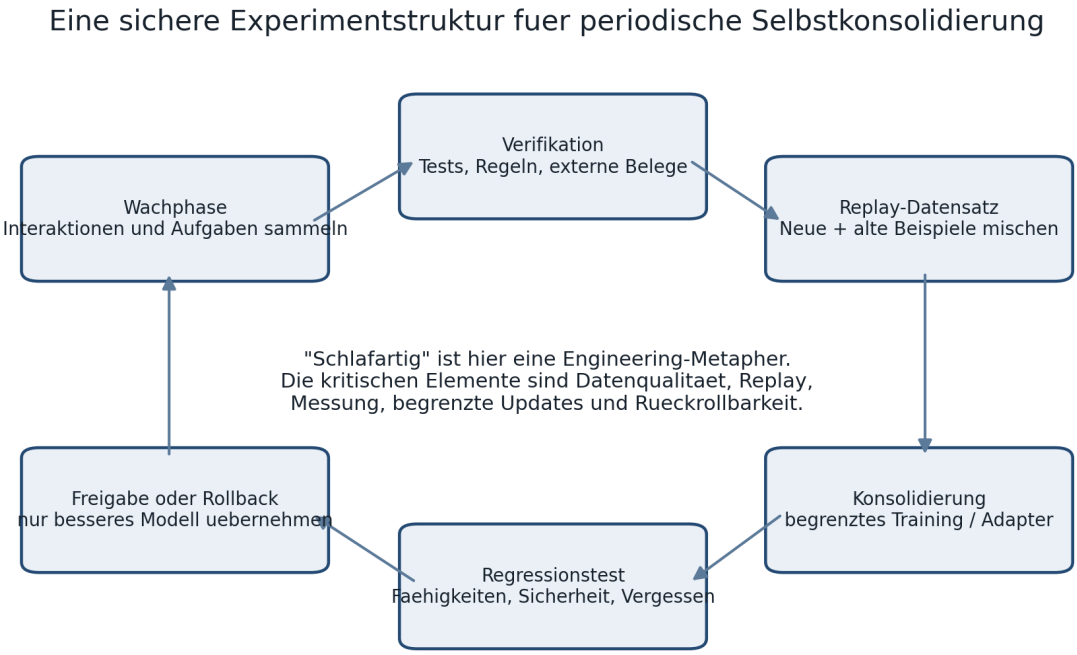



*Abbildung 8: Periodischer Lernzyklus mit Verifikation und Regressionstests*

Die Idee einer Wach- und Schlafphase hat historische Vorlaeufer, etwa den Wake-Sleep-Algorithmus fuer generative Modelle [24]. Fuer moderne Chatmodelle sollte „Schlaf“ jedoch nicht als biologische Gleichsetzung verstanden werden, sondern als getrennte Offlinephase: Erfahrungen sammeln, filtern, wiederholen, trainieren und pruefen.

## 19.1 Warum direktes Online-Gewichtlernen riskant ist

Wenn ein Modell jede Unterhaltung sofort als Wahrheit trainiert, kann es Tippfehler, Manipulationen, private Daten, eigene Halluzinationen und kurzfristige Stimmungen konsolidieren. Kleine Modelle vergessen dabei schnell alte Faehigkeiten. Ausserdem ist ein schlechter Gewichtsupdate schwerer rueckgaengig zu machen als ein Kontext- oder Datensatzfehler.

## 19.2 Ein praktikabler Zyklus

9. Wachphase: Aufgaben, Antworten, Fehler, Nutzerfeedback und objektive Ergebnisse als Ereignisse protokollieren.
10. Kandidatenbildung: Nur Lernbeispiele erzeugen, die eine klare Zielantwort oder ein pruefbares Verbesserungsziel besitzen.
11. Verifikation: Unit Tests, mathematische Checker, Quellen, Mehrmodell-Konsens oder menschliche Freigabe.
12. Replay: Neue Beispiele mit repräsentativen alten Daten mischen, damit das Netz nicht nur den neuesten Zeitraum sieht.
13. Konsolidierung: Kleine Lernrate, begrenzte Schritte, eventuell Adapter statt Vollmodell.
14. Regressionstest: Alter Testsatz, neue Ziele, Sicherheit, Stil und Kalibrierung vergleichen.
15. Freigabe: Nur bei vorab definiertem Gewinn; sonst Checkpoint verwerfen oder Rollback.

## 19.3 Drei Arten von Gedaechtnis unterscheiden

| Ebene | Beispiel | Eigenschaft |
|---|---|---|
| Kontextgedaechtnis | Tokens der aktuellen Sitzung | sofort, begrenzt, keine Gewichtsveraenderung |
| Externe episodische Speicherung | Logs, Datenbank, Replaybuffer | editierbar, pruefbar, nicht im Modellkern |
| Parametrische Konsolidierung | Fine-Tuning der Gewichte | kompakt und generalisierbar, aber schwer zu kontrollieren |

Auch wenn dein langfristiges Ziel kein klassisches externes Memory sein soll, ist fuer Forschung ein temporaerer Replay- und Auditdatensatz fast unvermeidbar. Ohne beobachtbare Zwischenstufe kannst du nicht wissen, was das Modell im „Schlaf“ gelernt hat oder warum es regressiert.

## 19.4 Replay und Schutz alter Faehigkeiten

Experience Replay mischt alte Erfahrungen mit neuen und reduziert Vergessen [22]. EWC bremst Aenderungen an Gewichten, die fuer alte Aufgaben wichtig waren [23]. Generative Replay laesst ein altes Modell Beispiele erzeugen, birgt aber Selbstverstärkung von Fehlern. Adapter oder LoRA koennen neue Faehigkeiten isolieren; spaeter kann man sie kontrolliert mergen oder verwerfen.

## 19.5 Selbstgenerierte Daten

STaR und verwandte Verfahren zeigen, dass eigene erfolgreiche Rationales Trainingsmaterial werden koennen [19]. Der entscheidende Filter ist Erfolg, nicht Selbstvertrauen. Fuer offene Weltwissen-Aussagen ist automatische Wahrheit schwer zu pruefen; fuer Mathematik, Code, Spiele und Simulationen ist sie wesentlich besser operationalisierbar. Deshalb sollten erste Selbstlernexperimente in verifizierbaren Umgebungen stattfinden.

## 19.6 Minimaler Schlaf-Experimentplan

| Phase | Konkrete Umsetzung | Metrik |
|---|---|---|
| Baseline | Checkpoint C0 einfrieren | vollstaendiger Evalbericht |
| Erfahrung | 1000 Aufgaben mit Ergebnissen sammeln | Fehlerarten und Erfolgsrate |
| Filter | nur automatisch verifizierte Beispiele | Precision des Filters |
| Replaymix | z.B. 20% neu, 80% alte Referenzdaten | alte/neue Tokenanteile |
| Update | kleine LR, feste Tokenzahl, neuer C1 | Train/Val Loss und Gradnorm |
| Vergleich | C0 gegen C1 blind testen | neue Ziele minus Regressionen |
| Freigabe | nur bei Schwellenwert | reproduzierbarer Gewinn |

**Unbequeme technische Wahrheit:** Ein System, das selbst Daten erzeugt, selbst bewertet und selbst seine Gewichte aktualisiert, kann sich sehr ueberzeugend in die falsche Richtung optimieren. Der Verifier und die Regressionstests sind deshalb nicht Nebenmodule, sondern der eigentliche Sicherheitskern.

# 20. Forschungsfahrplan fuer ungewoehnliche Architekturen

Von HTI-Baselines zu belastbaren neuen Mechaniken

Kuriose Architekturen sind sinnvoll, sobald eine Standardbaseline stabil trainiert und gemessen werden kann. Sonst ist jeder Effekt mehrdeutig. Ein Forschungsprogramm sollte Hypothesen formulieren, nicht nur Bauteile kombinieren.

## 20.1 Hypothesenformat

```
Hypothese: Mechanismus M verbessert Faehigkeit F, weil ...
Kontrolle: gleiche Daten, aktive Parameter, FLOPs und Trainingstokens
Primaermetrik: ...
Sekundaermetriken: Durchsatz, Speicher, Stabilitaet, Regressionen
Falsifikationskriterium: Wann gilt die Idee als nicht bestaetigt?
```

## 20.2 Sinnvolle Experimentlinien

| Experiment | Aenderung | Saubere Kontrolle |
|---|---|---|
| Rekurrenter Block | denselben Block 2-4 mal anwenden | gegen tieferes dichtes Modell mit aehnlichem Compute |
| Denktokens | spezielle interne Schritte vor Antwort | gegen laengeres normales CoT mit gleicher Tokenzahl |
| Verifier-Schleife | mehrere Kandidaten + Auswahl | gegen Best-of-N mit zufaelliger/heuristischer Wahl |
| Kleines MoE | 4 Experten, top-1/top-2 | gegen dichtes FFN mit gleicher aktiver FLOP-Zahl |
| Schlaf-Replay | periodisches Update aus Erfolgen | gegen gleiches Update ohne Replay und gegen kein Update |
| Spezialisierte Layer | abwechselnde Code/Mathe/Sprachmodule | gegen mehr Datengewichtung ohne Architekturwechsel |

## 20.3 Messungen, die oft vergessen werden

- Tokens pro Sekunde und reale Wandzeit.
- Peak-RAM/VRAM und Checkpointgroesse.
- Aktive gegen Gesamtparameter bei sparsamen Modellen.
- Varianz ueber mindestens mehrere Seeds bei kleinen Modellen.
- Eval pro Schwierigkeitsstufe statt nur Gesamtscore.
- Antwortlaenge und Reasoningtoken als Kostenmetrik.
- Vergessen alter Aufgaben nach jedem neuen Trainingszyklus.

## 20.4 Ein moeglicher HTI-Stufenplan

| Versionsebene | Forschungsfokus |
|---|---|
| HTI-1 bis HTI-3 | saubere dichte Baselines; Daten, Tokenizer und Skalierung verstehen |
| HTI-4 bis HTI-6 | GQA, Kontextlaenge, FFN-Varianten, Norm und Trainingsstabilitaet |
| HTI-7 bis HTI-9 | Verifier, Self-Consistency, Rationale-SFT und kleine rekurrente Tiefe |
| HTI-10+ | MoE, latentes Thinking, periodische Konsolidierung und kombinierte Architekturen |

Das ist keine Vorschrift, sondern eine Abhaengigkeitsordnung: spaetere Experimente sind nur interpretierbar, wenn Daten-, Trainings- und Evaluationsbasis vorher verlaesslich sind.

# 21. Kompakte Implementierungscheckliste

Von null bis zum ersten belastbaren Modell

## Daten und Tokenizer

- Quellen, Lizenzen, Filter und Mix dokumentiert.
- Deduplikation und Testkontamination geprueft.
- Tokenizer-Roundtrip und mehrsprachige Randfaelle bestanden.
- Train/Validation/Test vor dem Training getrennt.
- Zufaellige dekodierte Shards manuell kontrolliert.

## Modell

- Jede Tensorform im Code kommentiert oder getestet.
- Causal Mask durch Zukunftstoken-Test verifiziert.
- Parameterzahl stimmt mit Handrechnung ueberein.
- Weight Tying absichtlich an oder aus.
- RoPE-Positionen und Cache stimmen bei Train und Generate ueberein.

## Training

- Tiny-Overfit bestanden.
- Gradientennormen, Lernrate, Tokens/s und Loss geloggt.
- Validation in festen Tokenabstaenden.
- Checkpoint-Roundtrip und Resume getestet.
- Besten und letzten Checkpoint getrennt gesichert.

## Chat und Reasoning

- Chattemplate identisch in SFT und Inferenz.
- Loss nur auf beabsichtigten Zielbereichen.
- Reasoningdaten automatisch oder manuell verifiziert.
- Antwortlaenge als Kostenmetrik.
- Jede Selbstlernphase hat Rollback und Regressionstest.

# Anhang A. Formelsammlung

Die wichtigsten Gleichungen auf einer Stelle

## Embedding

```
X[b,t,:] = E[token_id[b,t], :]
```

## Scaled Dot-Product Attention

```
Attention(Q,K,V) = softmax(QK^T / sqrt(d_h) + M) V
```

## RMSNorm

```
RMSNorm(x) = g ⊙ x / sqrt(mean(x^2)+eps)
```

## SwiGLU

```
SwiGLU(x) = (SiLU(xW_gate) ⊙ xW_up) W_down
```

## Cross-Entropy

```
L = -mean(log softmax(logits)[target])
```

## Perplexity

```
PPL = exp(L)
```

## AdamW

```
theta_t = weight_decay_step(adaptive_momentum_step(theta_(t-1), grad_t))
```

## Grobe Parameterzahl

```
N ≈ Vd + L(4d^2 + 3d*d_ff)
```

## KV-Cache

```
memory ≈ L*T*n_kv_heads*d_head*2*bytes
```

## Grobe Trainings-FLOPs

```
compute ≈ 6*N*D   (nur Naeherung)
```

# Anhang B. Glossar

Praezise Kurzdefinitionen

| Begriff | Definition |
|---|---|
| Activation / Aktivierung | Zwischentensor, der fuer eine konkrete Eingabe entsteht. |
| Autoregressiv | Vorhersage eines Elements aus vorherigen Elementen, schrittweise. |
| Backpropagation | Rueckwaertsauswertung der Kettenregel zur Gradientenberechnung. |
| Batch | Menge parallel verarbeiteter Sequenzen. |
| Checkpoint | Gespeicherter Modell- und gegebenenfalls Trainingszustand. |
| Context Window | Maximal oder aktuell verarbeitete Tokenhistorie. |
| Cross-Entropy | Loss fuer die Wahrscheinlichkeit des Ziel-Tokens. |
| Embedding | Gelernter Vektor fuer eine diskrete ID. |
| Epoch | Ein ungefaehr vollstaendiger Durchlauf durch einen Datensatz. |
| Fine-Tuning | Weitertraining eines vorhandenen Checkpoints auf neuerem Ziel oder Datenmix. |
| Forward Pass | Berechnung von Eingabe zu Ausgabe und Loss. |
| Gradient | Lokale Ableitung des Loss nach Parametern. |
| Halluzination | Plausible, aber unbelegte oder falsche Generierung. |
| Head | Ein Attention-Unterraum mit eigenen Projektionen. |
| Inference | Nutzung des trainierten Modells ohne normale Gradientenupdates. |
| KV-Cache | Gespeicherte Keys und Values vergangener Tokens fuer schnelleres Decoding. |
| Logit | Unnormalisierter Score vor Softmax. |
| Loss | Skalar, den das Training minimiert. |
| MoE | Sparse Mixture of Experts mit datenabhaengigem Routing. |
| Optimizer | Regel zur Umwandlung von Gradienten in Parameterupdates. |

| Begriff | Definition |
|---|---|
| Parameter | Trainierbare Zahl des Modells. |
| Perplexity | Exponent der mittleren Cross-Entropy. |
| Pretraining | Breites initiales Training, meist mit selbstueberwachtem Sprachziel. |
| Residual | Addition einer Modulkorrektur zum bestehenden Zustand. |
| RoPE | Rotary Position Embedding fuer Query und Key. |
| SFT | Supervised Fine-Tuning auf Zielantworten. |
| Softmax | Normalisierung von Scores zu einer Wahrscheinlichkeitsverteilung. |
| Tokenizer | Abbildung zwischen Text und Token-IDs. |
| Validation | Nicht fuer Updates verwendete Daten zur Modellwahl. |
| Weight Tying | Gemeinsame Parameter fuer Eingabeembedding und Outputprojektion. |

# Anhang C. Literatur und Originalarbeiten

Auswahl der im Text referenzierten Grundlagen

**[1] Vaswani et al. (2017)** Attention Is All You Need. arXiv:1706.03762. Quelle

**[2] Sennrich, Haddow, Birch (2015)** Neural Machine Translation of Rare Words with Subword Units. arXiv:1508.07909. Quelle

**[3] Kudo, Richardson (2018)** SentencePiece: A simple and language independent subword tokenizer and detokenizer. arXiv:1808.06226. Quelle

**[4] Loshchilov, Hutter (2017)** Decoupled Weight Decay Regularization. arXiv:1711.05101. Quelle

**[5] Su et al. (2021)** RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv:2104.09864. Quelle

**[6] Zhang, Sennrich (2019)** Root Mean Square Layer Normalization. arXiv:1910.07467. Quelle

**[7] Shazeer (2020)** GLU Variants Improve Transformer. arXiv:2002.05202. Quelle

**[8] Kaplan et al. (2020)** Scaling Laws for Neural Language Models. arXiv:2001.08361. Quelle

**[9] Hoffmann et al. (2022)** Training Compute-Optimal Large Language Models. arXiv:2203.15556. Quelle

**[10] Rafailov et al. (2023)** Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290. Quelle

**[11] Wei et al. (2022)** Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903. Quelle

**[12] Wang et al. (2022)** Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171. Quelle

**[13] Yao et al. (2023)** Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601. Quelle

**[14] Yao et al. (2022)** ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629. Quelle

**[15] Fedus, Zoph, Shazeer (2021)** Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961. Quelle

**[16] Jiang et al. (2024)** Mixtral of Experts. arXiv:2401.04088. Quelle

**[17] DeepSeek-AI (2025)** DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948. Quelle

**[18] Zelikman et al. (2024)** Quiet-STaR: Language Models Can Teach Themselves to Think Before Speaking. arXiv:2403.09629. Quelle

**[19] Zelikman et al. (2022)** STaR: Bootstrapping Reasoning With Reasoning. arXiv:2203.14465. Quelle

**[20] Touvron et al. (2023)** LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971. Quelle

**[21] Brown et al. (2020)** Language Models are Few-Shot Learners. arXiv:2005.14165. Quelle

**[22] Rolnick et al. (2018)** Experience Replay for Continual Learning. arXiv:1811.11682. Quelle

**[23] Kirkpatrick et al. (2016)** Overcoming catastrophic forgetting in neural networks. arXiv:1612.00796. Quelle

**[24] Hinton, Dayan, Frey, Neal (1995)** The Wake-Sleep Algorithm for Unsupervised Neural Networks. Science 268. Quelle

## Hinweis zur Forschungslage

Reasoning und kontinuierliches Lernen sind aktive Forschungsfelder. Viele Resultate gelten fuer bestimmte Modellgroessen, Daten und Benchmarks und uebertragen sich nicht automatisch auf kleine CPU-Modelle. Deshalb wurden im Dokument Mechanismen und experimentelle Kontrollen getrennt dargestellt: Eine Idee ist erst dann eine Technologie fuer dein Modell, wenn sie unter deiner Hardware, deinem Tokenbudget und deiner Evaluation reproduzierbar gewinnt.