

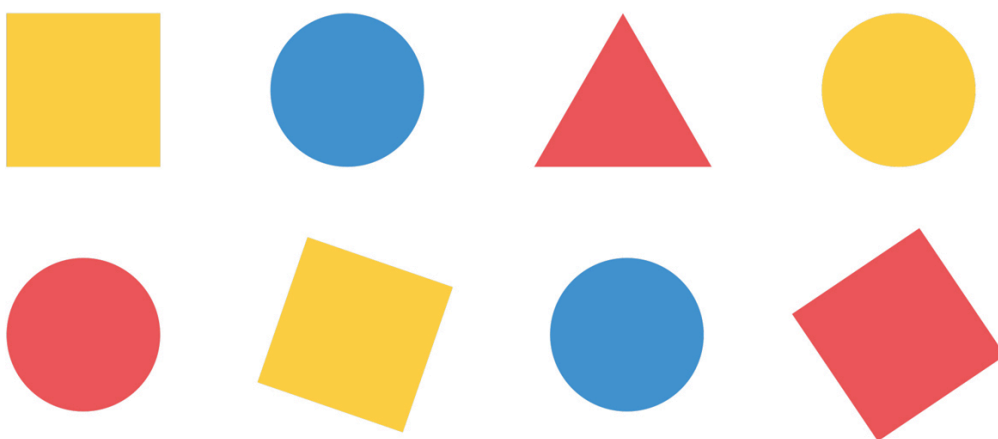
O'REILLY®

Deutsche  
Ausgabe von  
The Hundred-Page  
Language Models Book

Andriy Burkov  
Übersetzung von Frank Langenau

# Language Models kompakt

Praxisorientierte Sprachmodellierung  
mit PyTorch



# Inhalt

**Cover**

**Stimmen zum Buch: Language Models kompakt**

**Hinweise zur Benutzung**

**Titel**

**Impressum**

**Widmung**

**Inhalt**

**Vorwort**

**Einführung**

Für wen dieses Buch gedacht ist

Was dieses Buch nicht ist

Wie dieses Buch aufgebaut ist

Beispiele und Wiki zum Buch

Verwenden von Codebeispielen

Danksagungen

## **1 Grundlagen des Machine Learning**

KI und Machine Learning

Modelle

Machine Learning – Prozess in vier Schritten Vektoren

Neuronale Netze

Matrizen

Gradientenabstieg

Automatisches Differenzieren

## **2 Grundlagen der Sprachmodellierung**

Bag-of-Words

Wort-Embeddings  
Byte-Paar-Codierung  
Sprachmodelle  
Zählbasierte Sprachmodelle  
Sprachmodelle bewerten  
    Perplexität  
    ROUGE  
    Menschliche Bewertung

### **3 Rekurrente neuronale Netze**

Elman-RNN  
Mini-Batch-Gradientenabstieg  
Ein RNN programmieren  
RNN als Sprachmodell  
Embedding-Schicht  
Ein RNN-Sprachmodell trainieren  
Die Klassen Dataset und DataLoader  
Trainingsdaten und Verlustberechnung

### **4 Transformer**

Decoder-Block  
Self-Attention  
    Schritt 1 der Self-Attention  
    Schritt 2 der Self-Attention  
    Schritt 3 der Self-Attention  
    Schritt 4 der Self-Attention  
    Schritt 5 der Self-Attention  
    Schritt 6 der Self-Attention  
Positionsbezogenes Multilayer Perceptron  
Rotary Position Embedding

- Multi-Head-Attention
- Residualverbindungen
- RMS-Normalisierung
- Schlüssel-Wert-Caching
- Transformer in Python

## **5 Große Sprachmodelle (LLMs)**

Warum größer besser ist

- Große Parameteranzahl
- Großer Kontextumfang
- Großer Trainingsdatensatz
- Großer Rechenaufwand

Überwachtes Feintuning

Ein vortrainiertes Modell feintunen

- Baseline für den Klassifizierer von Emotionen
- Emotionen erzeugen
- Feintuning zum Befolgen von Anweisungen

Sampling von Sprachmodellen

- Einfaches Sampling mit Temperatur
- Top-k-Sampling
- Nucleus-(Top-p-)Sampling
- Strafen

Low-rank Adaptation (LoRA)

- Die Kernidee
- Parametereffizientes Feintuning (PEFT)

LLM als Klassifizierer

Prompt Engineering

- Merkmale eines guten Prompts
- Folgeaktionen

Codegenerierung

Synchronisieren der Dokumentation

Halluzinationen

Gründe für Halluzinationen

Halluzinationen verhindern

LLMs, Urheberrecht und Ethik

Trainingsdaten

Generierte Inhalte

Open-Weight-Modelle

Allgemeine ethische Erwägungen

## **6 Fortgeschrittene Themen**

Mixture of Experts

Model Merging

Modellkomprimierung

Präferenzbasierte Ausrichtung

Advanced Reasoning

Sicherheit von Sprachmodellen

Vision Language Models

Überanpassung verhindern

## **Schlussbemerkungen**

Mehr vom Autor

## **Index**

## **Über den Autor**

Transformer-Modelle haben NLP (*Natural Language Processing*) erheblich vorangebracht. Sie überwinden die Einschränkungen von RNNs beim Umgang mit weitreichenden Abhängigkeiten und ermöglichen die parallele Verarbeitung von Eingabesequenzen. Es gibt drei Hauptversionen von Transformer-Architekturen: Encoder-Decoder (ursprünglich formuliert für die maschinelle Übersetzung), nur Encoder (typischerweise für die Klassifizierung verwendet) und nur Decoder (häufig in Chat-LMs zu finden).

In diesem Kapitel untersuchen wir die »Nur-Decoder-Transformer-Architektur« ausführlich, da sie den am weitesten verbreiteten Ansatz für das Training *autoregressiver Sprachmodelle* bildet.

Die Transformer-Architektur führt zwei Schlüsselinnovationen ein: *Self-Attention* (Selbstaufmerksamkeit) und *Positional Encoding* (Positionscodierung). *Self-Attention* ermöglicht dem Modell, während der Vorhersage zu beurteilen, wie jedes Wort mit allen anderen zusammenhängt, während die Positionscodierung die Wortreihenfolge und sequenzielle Muster erfasst. Im Gegensatz zu RNNs verarbeiten Transformer alle Tokens gleichzeitig und verwenden Positionscodierung, um den sequenziellen Kontext trotz paralleler Verarbeitung beizubehalten. Dieses Kapitel untersucht diese fundamentalen Elemente im Detail.

Ein *Nur-Decoder-Transformer* (im Folgenden einfach »Decoder« genannt) besteht aus mehreren identischen<sup>1</sup> Schichten, den sogenannten Decoder-Blöcken, die übereinandergestapelt sind, wie [Abbildung 4.1](#) zeigt.

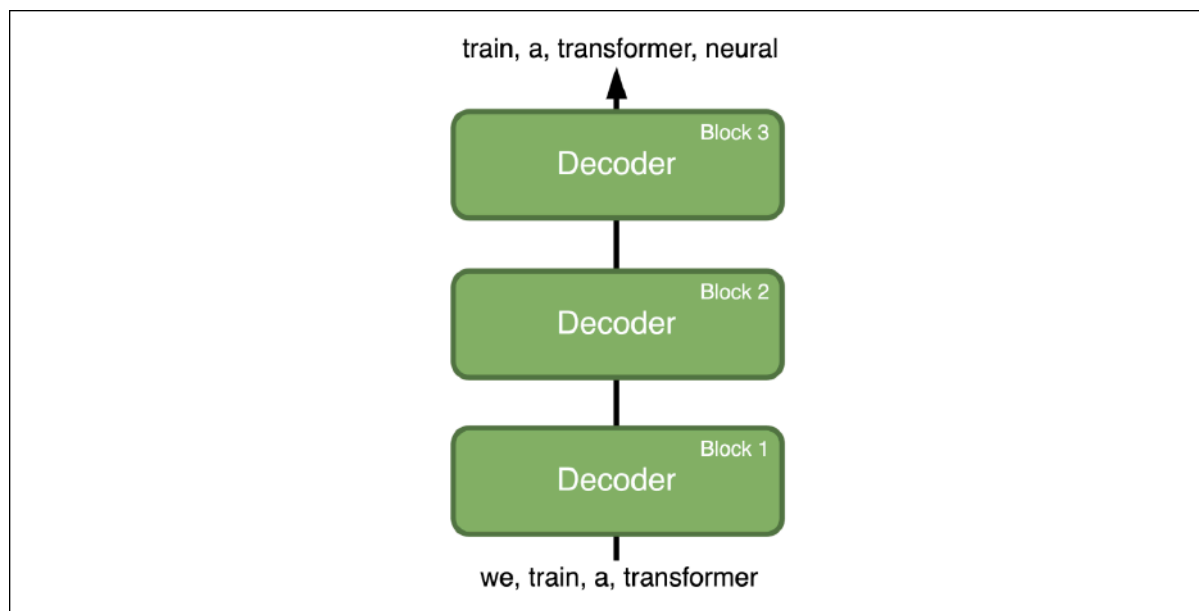


Abbildung 4.1: Transformer-Architektur mit gestapelten Decoder-Blöcken

Wie aus [Abbildung 4.1](#) hervorgeht, wird beim Training eines Decoders jede Eingabesequenz mit einer Zielsequenz, die um ein Token nach vorn verschoben wird, gepaart – die gleiche Methode, wie sie in RNN-basierten Sprachmodellen üblich ist.

## Decoder-Block

Jeder Decoder-Block besteht aus zwei Teilschichten (siehe [Abbildung 4.2](#)): *Self-Attention* und positionsbezogenes mehrschichtiges Perzeptron (*Multilayer Perceptron*, MLP). Die Darstellung vereinfacht bestimmte Aspekte, um nicht zu viele neue Konzepte auf einmal zu präsentieren. Die fehlenden Details führen wir Schritt für Schritt ein.

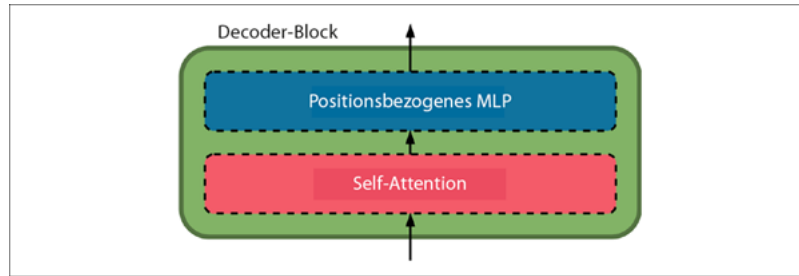


Abbildung 4.2: Vereinfachte Darstellung eines Decoder-Blocks

Schauen wir uns genauer an, was in einem Decoder-Block passiert, und beginnen wir mit dem ersten Block, den [Abbildung 4.3](#) zeigt.

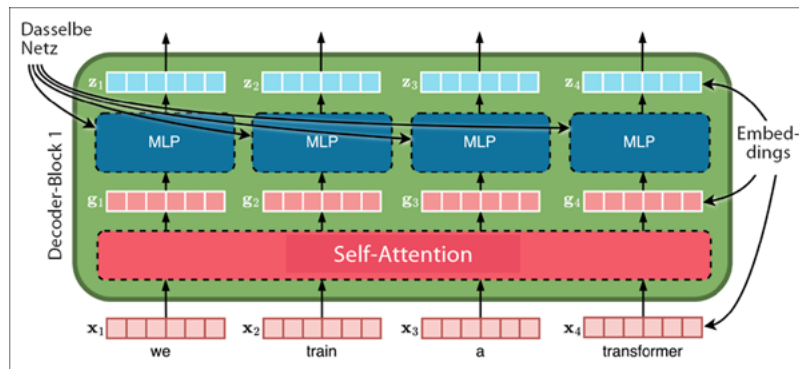


Abbildung 4.3: Schematische Darstellung des ersten Decoder-Blocks

Der erste Decoder-Block verarbeitet Eingabetoken-Embeddings. Für dieses Beispiel verwenden wir 6-dimensionale Eingabe- und Ausgabe-Embeddings, obwohl diese Dimensionen in der Praxis mit der Anzahl der Parameter und dem Token-Vokabular größer werden. Die *Self-Attention-Schicht* transformiert jeden Eingabe-Embedding-Vektor  $\mathbf{x}_t$  in einen neuen Vektor  $\mathbf{g}_t$  für jedes Token  $t$  von 1 bis  $L$ , wobei  $L$  die Eingabelänge angibt.



In [Abbildung 4.3](#) ist jede Einheit vereinfacht als Quadrat dargestellt, genau wie beim Ansatz, den wir für das Netz mit vier Einheiten im Abschnitt »Neuronale Netze« auf [Seite 37](#) verwendet haben. Während in den früheren Kapiteln der Informationsfluss in einem neuronalen Netz von links nach rechts dargestellt wurde, sind wir nun zu einer Ausrichtung von unten nach oben übergegangen – die Standardkonvention für Diagramme von Hochsprachenmodellen in der Literatur. Von nun an bleiben wir bei dieser Ausrichtung.

Nach der Self-Attention verarbeitet das positionsbezogene MLP die einzelnen Vektoren  $\mathbf{g}_t$  unabhängig voneinander. Jeder Decoder-Block verfügt über ein eigenes MLP mit spezifischen Parametern. Innerhalb eines Blocks wird dasselbe MLP unabhängig von den anderen auf den Vektor der jeweiligen Position angewendet, wobei ein  $\mathbf{g}_t$  als Eingabe übernommen und ein  $\mathbf{z}_t$  als Ausgabe erzeugt wird. Wenn das MLP die sequenzielle Verarbeitung jeder Position abgeschlossen hat, entspricht die Anzahl der Ausgabevektoren  $\mathbf{z}_t$  der Anzahl der Eingabetokens  $\mathbf{x}_t$ .

Die Ausgabevektoren  $\mathbf{z}_t$  dienen dann als Eingaben für den nächsten Decoder-Block. Dieser Vorgang wiederholt sich für jeden Decoder-Block, wobei eine Anzahl von Ausgabevektoren erhalten bleibt, die der Anzahl der Eingabetokens  $\mathbf{x}_t$  entspricht.

## Self-Attention

Um zu sehen, wie die *Self-Attention* funktioniert, beginnen wir mit einem intuitiven Vergleich. Die Transformation von  $\mathbf{g}_t$  in  $\mathbf{z}_t$  ist ganz einfach: Ein positionsbezogenes MLP übernimmt einen Eingabevektor und gibt einen neuen Vektor aus, indem eine gelernte Transformation angewendet wird. Genau dafür sind Feedforward-Netze konzipiert. Self-Attention mag jedoch komplexer erscheinen.

Nehmen wir ein Beispiel mit fünf Tokens: [»we« »train« »a« »transformer« »model«]. Außerdem verwenden wir einen Decoder mit einer maximalen Länge der Eingabesequenz von 4.

In jedem Decoder-Block stützt sich die Self-Attention-Funktion auf drei Tensoren trainierbarer Parameter:  $\mathbf{W}^Q$ ,  $\mathbf{W}^K$  und  $\mathbf{W}^V$ . Hier steht  $Q$  für *Query* (Abfrage),  $K$  für *Key* (Schlüssel) und  $V$  für *Value* (Wert).

Die Dimensionen dieser Tensoren nehmen wir mit  $6 \times 6$  an. Dementsprechend wird jeder der vier 6-dimensionalen Eingabevektoren in vier 6-dimensionale Ausgabevektoren transformiert. Zur Veranschaulichung wählen wir als Beispiel das zweite Token,  $\mathbf{x}_2$ , das das Wort »train« darstellt. Die Ausgabe  $\mathbf{g}_2$  für  $\mathbf{x}_2$  berechnet die Self-Attention-Schicht in sechs Schritten.

### Schritt 1 der Self-Attention

Es werden die Matrizen  $\mathbf{Q}$ ,  $\mathbf{K}$  und  $\mathbf{V}$  berechnet. Wie [Abbildung 4.4](#) zeigt, werden die vier Eingabe-Embeddings  $\mathbf{x}_1$ ,  $\mathbf{x}_2$ ,  $\mathbf{x}_3$  und  $\mathbf{x}_4$  zu einer Matrix  $\mathbf{X}$  zusammengefasst.

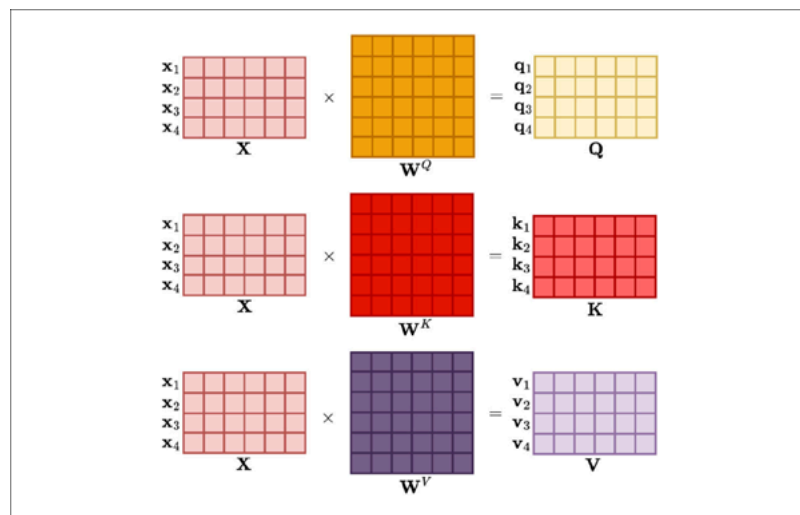


Abbildung 4.4: Matrixmultiplikation in der Self-Attention-Schicht

Dann multiplizieren wir  $\mathbf{X}$  mit den Gewichtsmatrizen  $\mathbf{W}^Q$ ,  $\mathbf{W}^K$  und  $\mathbf{W}^V$ , um die Matrizen  $\mathbf{Q}$ ,  $\mathbf{K}$  und  $\mathbf{V}$  zu erzeugen. Diese Matrizen enthalten jeweils 6-dimensionale Abfrage-, Schlüssel- und Wertvektoren. Da der Prozess die gleiche Anzahl von Abfrage-, Schlüssel- und Wertvektoren wie Eingabe-Embeddings erzeugt, entspricht jedes Eingabe-Embedding  $\mathbf{x}_t$  einem Abfragevektor  $\mathbf{q}_t$ , einem Schlüsselvektor  $\mathbf{k}_t$  und einem Wertvektor  $\mathbf{v}_t$ .

### Schritt 2 der Self-Attention

Für das als Beispiel gewählte zweite Token  $\mathbf{x}_2$  berechnen wir *Attention-Scores*, indem wir das Punktprodukt seines Abfragevektors  $\mathbf{q}_2$  mit jedem Schlüsselvektor  $\mathbf{k}_t$  bilden. Als resultierende Scores nehmen wir folgende an:

$$\mathbf{q}_2 \cdot \mathbf{k}_1 = 4.90, \quad \mathbf{q}_2 \cdot \mathbf{k}_2 = 17.15, \quad \mathbf{q}_2 \cdot \mathbf{k}_3 = 9.80, \quad \mathbf{q}_2 \cdot \mathbf{k}_4 = 12.25$$

Im Vektorformat sieht das so aus:

$$\mathbf{scores}_2 = [4.90, 17.15, 9.80, 12.25]^T$$

### Schritt 3 der Self-Attention

Um die *skalierten Scores* zu erhalten, teilen wir jeden Attention-Score durch die Quadratwurzel aus der Dimensionalität des Schlüsselvektors. Da der Schlüsselvektor in unserem Beispiel eine Dimensionalität von 6 hat, teilen wir alle Scores durch  $\sqrt{6} \approx 2.45$ , was Folgendes ergibt:

$$\mathbf{scaled\_scores}_2 = \left[ \frac{4.9}{2.45}, \frac{17.15}{2.45}, \frac{9.8}{2.45}, \frac{12.25}{2.45} \right] = [2, 7, 4, 5]^T$$

### Schritt 4 der Self-Attention

Auf die skalierten Scores wenden wir dann die *kausale Maske* an. (Auf den Grund für die Verwendung der kausalen Maske gehe ich noch ausführlich ein.) Für die zweite Eingabeposition sieht die kausale Maske wie folgt aus:

$$\text{causal\_mask}_2 \stackrel{\text{def}}{=} [0, 0, -\infty, -\infty]^\top$$

Wir addieren die skalierten Scores zur kausalen Maske hinzu. Das Ergebnis sind die *maskierten Scores*:

$$\text{masked\_scores}_2 = \text{scaled\_scores}_2 + \text{causal\_mask}_2 = [2, 7, -\infty, -\infty]^\top$$

### Schritt 5 der Self-Attention

Auf die maskierten Scores wenden wir die *Softmax-Funktion* an, um die Attention-Gewichte zu erzeugen:

$$\text{attention\_weights}_2 = \text{softmax}([2, 7, -\infty, -\infty]^\top)$$

Da Scores von  $-\infty$  nach Anwendung der Exponentialfunktion zu null werden, sind die Attention-Gewichte für die dritte und vierte Position gleich null. Die verbleibenden zwei Gewichte werden wie folgt berechnet:

$$\text{attention\_weights}_2 = \left[ \frac{e^2}{e^2+e^7}, \frac{e^7}{e^2+e^7}, 0, 0 \right]^\top \approx [0.0067, 0.9933, 0, 0]^\top$$



Indem die Attention-Scores durch die Quadratwurzel aus der Dimensionalität des Schlüssels geteilt werden, verhindert das, dass die Punktprodukte mit zunehmender Dimensionalität zu groß werden, was nach Anwendung der Softmax-Funktion extrem kleine Gradienten ergeben könnte (aufgrund sehr großer negativer oder positiver Werte, die die Softmax-Ausgaben auf 0 oder 1 drücken).

### Schritt 6 der Self-Attention

Wir berechnen den Ausgabevektor  $\mathbf{g}_2$  für das Eingabe-Embedding  $\mathbf{x}_2$  aus der gewichteten Summe der Wertvektoren  $\mathbf{v}_1$ ,  $\mathbf{v}_2$ ,  $\mathbf{v}_3$  und  $\mathbf{v}_4$ , wobei wir die Attention-Gewichte aus dem vorherigen Schritt verwenden:

$$\mathbf{g}_2 \approx 0.0067 \cdot \mathbf{v}_1 + 0.9933 \cdot \mathbf{v}_2 + 0 \cdot \mathbf{v}_3 + 0 \cdot \mathbf{v}_4$$

Wie man sieht, hängt die Ausgabe des Decoders für die Position 2 nur von den Eingaben an den Positionen 1 und 2 ab, wobei Position 2 einen viel stärkeren Einfluss hat. Dieser Effekt ergibt sich aus der kausalen Maske, die das Modell daran hindert, zukünftige Positionen zu berücksichtigen, wenn eine Ausgabe für eine bestimmte Position erzeugt wird. Diese Eigenschaft ist entscheidend, um den *autoregressiven Charakter von Sprachmodellen* beizubehalten. Dabei wird sichergestellt, dass Vorhersagen nur auf vorherigen und aktuellen Eingaben und nicht auf zukünftigen Eingaben beruhen.



Dieses Token bezieht sich in unserem Beispiel zwar hauptsächlich auf sich selbst, doch in verschiedenen Kontexten variieren auch die Attention-Muster. Je nach Satzstruktur kann ein Token andere Tokens, die relevante semantische oder syntaktische Informationen liefern, besonders berücksichtigen.

Die Vektoren  $\mathbf{q}_t$ ,  $\mathbf{k}_t$  und  $\mathbf{v}_t$  können wie folgt interpretiert werden: Jede Eingabeposition (Token oder Embedding) sucht nach Informationen über andere Positionen. Zum Beispiel könnte ein Token wie »I« an einer anderen Position nach einem Namen suchen, sodass das Modell »I« und den Namen in ähnlicher Weise verarbeiten kann. Um dies zu ermöglichen, wird jeder Position  $t$  eine Abfrage  $\mathbf{q}_t$  zugeordnet.

Der Self-Attention-Mechanismus berechnet ein *Punktprodukt* zwischen  $\mathbf{q}_t$  und jedem Schlüssel  $\mathbf{k}_p$  über alle Positionen  $p$ . Ein größeres Punktprodukt deutet auf eine größere Ähnlichkeit zwischen den Vektoren hin. Wenn der Schlüssel  $\mathbf{k}_p$  der Position  $p$  eng mit der Abfrage  $\mathbf{q}_t$  der Position  $t$  zusammenhängt, trägt der Wert  $\mathbf{v}_p$  an der Position  $p$  in größerem Maße zum Endergebnis bei.



Das Konzept der Attention ist bereits vor dem Transformer entstanden. Im Jahr 2014 hat sich Dzmitry Bahdanau während seines Studiums unter Yoshua Bengio mit einer grundlegenden Herausforderung in der maschinellen Übersetzung auseinandergesetzt: Ein RNN sollte sich auf die relevantesten Teile eines Satzes konzentrieren können. Ausgehend von seinen eigenen Erfahrungen beim Erlernen der englischen Sprache – wobei er auf verschiedene Teilen des Texts geachtet hat –, entwickelte Bahdanau einen Mechanismus, damit das RNN bei jedem Übersetzungsschritt »entscheiden« kann, welche

Eingabewörter am wichtigsten sind. Dieser Mechanismus, für den Bengio damals den Begriff »Attention« prägte, wurde zu einem Eckpfeiler neuronaler Netze.

Die Berechnung von  $\mathbf{g}_2$  wird für jede Position in der Eingabesequenz wiederholt, was zu einer Reihe von Ausgabevektoren führt:  $\mathbf{g}_1$ ,  $\mathbf{g}_2$ ,  $\mathbf{g}_3$  und  $\mathbf{g}_4$ . Jede Position hat ihre eigene kausale Maske, sodass  $\mathbf{g}_1$ ,  $\mathbf{g}_2$  und  $\mathbf{g}_3$  für jede Position mit jeweils einer anderen kausalen Maske berechnet werden. Die vollständige kausale Maske für alle Positionen sieht so aus:

$$\mathbf{M} \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Wie Sie sehen können, bezieht sich das erste Token nur auf sich selbst, das zweite Token auf sich selbst und das erste, das dritte auf sich selbst und die ersten beiden sowie das letzte auf sich selbst und alle vorhergehenden Tokens.

Die allgemeine Formel zur Berechnung der Attention für alle Positionen lautet:

$$\mathbf{G} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \stackrel{\text{def}}{=} \text{softmax} \left( \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M} \right) \mathbf{V}$$

Dabei sind  $\mathbf{Q}$  und  $\mathbf{V}$  Abfrage- und Wertmatrizen der Form  $L \times d_k$ .  $\mathbf{K}^T$  ist die mit  $d_k \times L$  transponierte Schlüsselmatrix, wobei  $d_k$  die Dimensionalität der Schlüssel-, Abfrage- und Wertvektoren angibt und  $L$  die Sequenzlänge ist.

Weiter oben haben wir die Attention-Scores für  $\mathbf{x}_2$  explizit berechnet, doch lassen sie sich per Matrixmultiplikation  $\mathbf{Q}\mathbf{K}^T$  für alle Positionen auf einmal berechnen. Diese Methode beschleunigt den Vorgang erheblich.

Damit ist die Definition der Self-Attention abgeschlossen.

## Positionsbezogenes Multilayer Perceptron

Nach der maskierten Self-Attention-Schicht wird jeder Ausgabevektor  $\mathbf{g}_t$  einzeln von einem *mehrschichtigen Perzeptron* (*Multilayer Perceptron*, MLP) verarbeitet. Das MLP wendet eine Reihe zusätzlicher Transformationen an:

$$\mathbf{z}_t = \mathbf{W}_2(\text{ReLU}(\mathbf{W}_1\mathbf{g}_t + \mathbf{b}_1)) + \mathbf{b}_2$$

Dabei sind  $\mathbf{W}_1$ ,  $\mathbf{W}_2$ ,  $\mathbf{b}_1$  und  $\mathbf{b}_2$  gelernte Parameter. Der resultierende Vektor  $\mathbf{z}_t$  wird dann entweder an den nächsten Decoder-Block übergeben oder, wenn es sich um den letzten Decoder-Block handelt, zum Erzeugen des Ausgabevektors verwendet.



Bei dieser Komponente handelt es sich um ein positionsbezogenes mehrschichtiges Perzeptron, weshalb ich diesen Begriff auch verwende. In der Literatur wird sie ebenfalls als Feedforward-Netz, dichte Schicht oder vollständig verbundene Schicht bezeichnet. Diese Namen können aber irreführend sein. Der gesamte Transformer ist ein neuronales Feedforward-Netz. Außerdem enthalten dichte oder vollständig verbundene Schichten in der Regel eine Gewichtsmatrix, einen Bias-Vektor und eine Nichtlinearität in der Ausgabe. Das positionsbezogene MLP in einem Transformer verwendet dagegen zwei Gewichtsmatrizen sowie zwei Bias-Vektoren und verzichtet auf eine Nichtlinearität in der Ausgabe.

## Rotary Position Embedding

Die bisher beschriebene Transformer-Architektur berücksichtigt die Wortreihenfolge nicht von Haus aus. Die kausale Maske stellt sicher, dass jedes Token nicht auf Tokens auf seiner rechten Seite achten kann, aber das Umordnen von Tokens auf der linken Seite hat keinen Einfluss auf die Attention-Gewichte eines bestimmten Tokens. Demgegenüber werden bei RNNs die verborgenen Zustände sequenziell berechnet, wobei jeder Zustand vom vorherigen abhängt. Ändert sich die Wortreihenfolge in RNNs, wirkt sich das auch auf die verborgenen Zustände und folglich die Ausgabe aus. Im Unterschied dazu berechnen Transformer die Attention über alle Tokens auf einmal, und zwar ohne sequenzielle Abhängigkeit.

Um die Wortreihenfolge zu berücksichtigen, müssen Transformer Positionsinformationen einbeziehen. Eine weitverbreitete Methode hierfür ist das *rotierende Positions-Embedding* (*Rotary Position Embedding*, RoPE), das positionsabhängige Rotationen auf die Abfrage- und Schlüsselvektoren im Attention-Mechanismus anwendet. Ein

Hauptvorteil von RoPE ist die Fähigkeit, effektiv auf Sequenzen zu verallgemeinern, die länger sind als die, die beim Training gesehen wurden. Dadurch ist es möglich, Modelle auf kürzeren Sequenzen zu trainieren, was Zeit- und Rechenressourcen spart, während gleichzeitig während der Inferenz viel längere Kontexte unterstützt werden.

RoPE codiert Positionsinformationen, indem die Abfrage- und Schlüsselvektoren gedreht werden. Diese Rotation findet vor der Attention-Berechnung statt. [Abbildung 4.5](#) zeigt, wie dies im zweidimensionalen Fall funktioniert. Der Pfeil mit der Beschriftung »Original« zeigt auf einen positionslosen Schlüssel- oder Abfragevektor in Self-Attention. RoPE bettet Positionsinformationen ein, indem dieser Vektor entsprechend der Position des Tokens gedreht wird.<sup>2</sup>

Die anderen Pfeile zeigen die resultierenden gedrehten Vektoren für die Positionen 1, 3, 5 und 7.

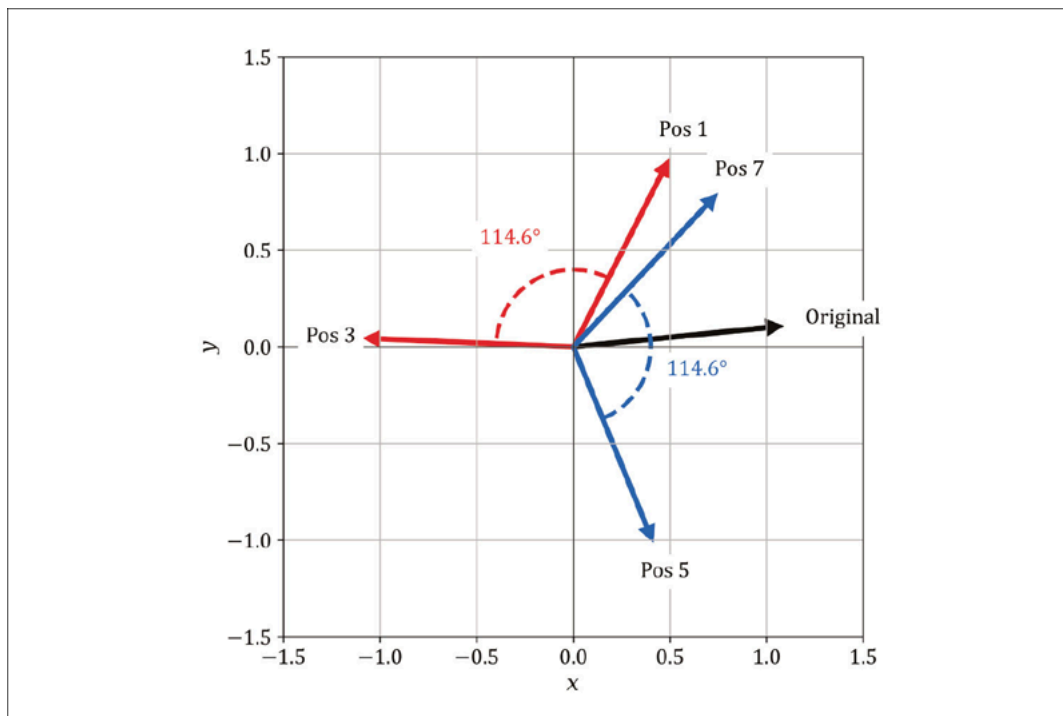


Abbildung 4.5: Beispiele für gedrehte Vektoren

Eine Schlüsseleigenschaft von RoPE ist, dass der Winkel zwischen zwei gedrehten Vektoren den Abstand zwischen ihren Positionen in der Sequenz codiert. Zum Beispiel ist der Winkel zwischen den Positionen 1 und 3 der gleiche wie der Winkel zwischen den Positionen 5 und 7, da beide Paare zwei Positionen voneinander entfernt sind.

Wie also drehen wir Vektoren? Wir verwenden Matrixmultiplikation! *Rotationsmatrizen* sind in vielen Bereichen stark verbreitet, beispielsweise in der Computergrafik, um 3-D-Szenen zu drehen – einer der ursprünglichen Zwecke von GPUs (das »G« in »GPU« steht für »grafisch«), bevor sie auf das Training neuronaler Netz angewendet wurden.

In zwei Dimensionen sieht die Rotationsmatrix für einen Winkel  $\vartheta$  wie folgt aus:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Um beispielsweise den zweidimensionalen Vektor  $\mathbf{q} = [2, 1]^\top$  zu drehen, multiplizieren wir  $\mathbf{q}$  mit der Rotationsmatrix  $\mathbf{R}_\vartheta$ . Das Ergebnis ist ein neuer Vektor, der  $\mathbf{q}$  um einen Winkel  $\vartheta$  entgegen dem Uhrzeigersinn gedreht darstellt.

Für eine 45°-Drehung ( $\theta = \pi/4 \text{ rad}$ ) können wir diese speziellen Werte verwenden:  $\cos(\theta) = \sin(\theta) = \frac{\sqrt{2}}{2}$ . Dies ergibt die Rotationsmatrix:

$$\mathbf{R}_{45^\circ} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

Um den gedrehten Vektor zu ermitteln, multiplizieren wir  $\mathbf{R}_{45^\circ}$  mit  $\mathbf{q}$ :

$$\mathbf{q}_{\text{gedreht}} = \mathbf{R}_{45^\circ} \cdot \mathbf{q} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Diese Multiplikation läuft schrittweise folgendermaßen ab:

$$\mathbf{q}_{\text{gedreht}} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 2 - \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 2 + \frac{\sqrt{2}}{2} \cdot 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2}(2-1) \\ \frac{\sqrt{2}}{2}(2+1) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 3 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{3\sqrt{2}}{2} \end{bmatrix}$$

Abbildung 4.6 veranschaulicht  $\mathbf{q}$  und dessen gedrehte Version für  $\vartheta = 45^\circ$ :

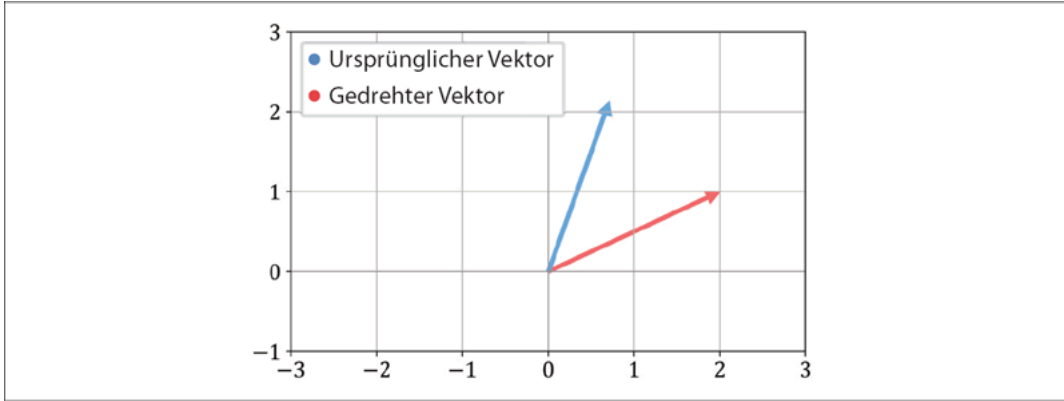


Abbildung 4.6: Beispiel für einen Vektor und den um  $45^\circ$  entgegen dem Uhrzeigersinn gedrehten Vektor

Für eine Position  $t$  dreht RoPE jedes Paar von Dimensionen in den Abfrage- und Schlüsselvektoren, die wie folgt definiert sind:

$$\mathbf{q}_t = \left[ q_t^{(1)}, q_t^{(2)}, \dots, q_t^{(d_q-1)}, q_t^{(d_q)} \right]^\top$$

$$\mathbf{k}_t = \left[ k_t^{(1)}, k_t^{(2)}, \dots, k_t^{(d_k-1)}, k_t^{(d_k)} \right]^\top$$

Hier sind  $d_q$  und  $d_k$  die (gleiche) Dimensionalität der Abfrage- und Schlüsselvektoren. RoPE dreht Paare von Dimensionen, die als  $(2p-1, 2p)$  indiziert sind, wobei sich der Index  $p$  jedes Pairs von 1 bis  $d_q/2$  erstreckt. Um die Dimensionen von  $\mathbf{q}_t$  in  $d_q/2$ -Paare aufzuteilen, gruppieren wir sie wie folgt:

$$\left[ q_t^{(1)}, q_t^{(2)} \right]^\top, \left[ q_t^{(3)}, q_t^{(4)} \right]^\top, \dots, \left[ q_t^{(d_q-1)}, q_t^{(d_q)} \right]^\top$$

Wenn wir  $\mathbf{q}_t(p)$  schreiben, steht dies für das Paar  $\left[ q_t^{(2p-1)}, q_t^{(2p)} \right]$ . Zum Beispiel entspricht  $\mathbf{q}_t(3)$ :

$$\left[ q_t^{(2 \cdot 3 - 1)}, q_t^{(2 \cdot 3)} \right] = \left[ q_t^{(5)}, q_t^{(6)} \right]$$

Jedes Paar erfährt eine Drehung basierend auf der Token-Position  $t$  und einer Rotationsfrequenz  $\vartheta_p$ :

$$\text{RoPE}(\mathbf{q}_t(p)) \stackrel{\text{def}}{=} \begin{bmatrix} \cos(\vartheta_p t) & -\sin(\vartheta_p t) \\ \sin(\vartheta_p t) & \cos(\vartheta_p t) \end{bmatrix} \begin{bmatrix} q_t^{(2p-1)} \\ q_t^{(2p)} \end{bmatrix}$$

Wendet man die Regel der Matrix-Vektor-Multiplikation an, liefert die Drehung den folgenden 2-D-Vektor:

$$\text{RoPE}(\mathbf{q}_t(p)) = \left[ q_t^{(2p-1)} \cos(\vartheta_p t) - q_t^{(2p)} \sin(\vartheta_p t), q_t^{(2p-1)} \sin(\vartheta_p t) + q_t^{(2p)} \cos(\vartheta_p t) \right]^\top$$

Darin ist  $\vartheta_p$  die Rotationsfrequenz für das  $p$ -te Paar. Definiert ist sie wie folgt:

$$\vartheta_p \stackrel{\text{def}}{=} \frac{1}{\vartheta^{2(p-1)/d_q}}$$

Hier ist  $\vartheta$  eine Konstante. Ursprünglich auf 10.000 gesetzt, haben spätere Experimente gezeigt, dass höhere Wert von  $\vartheta$  – wie zum Beispiel 500.000 (in den Modellreihen Llama 2 und 3) oder 1.000.000 (in den Reihen Qwen 2 und 2.5) – höhere Kontextgrößen (Hunderttausende von Tokens) ermöglichen.

Um das vollständig gedrehte Embedding  $\text{RoPE}(\mathbf{q}_t)$  zu konstruieren, werden alle gedrehten Paare verkettet.

$$\text{RoPE}(\mathbf{q}_t) \stackrel{\text{def}}{=} \text{concat} \left[ \text{RoPE}(\mathbf{q}_t(1)), \text{RoPE}(\mathbf{q}_t(2)), \dots, \text{RoPE}(\mathbf{q}_t(d_q/2)) \right]$$

Die Rotationsfrequenz  $\vartheta_p$  nimmt für jedes nachfolgende Paar aufgrund des Exponentialausdrucks im Nenner schnell ab. Dadurch kann RoPE feinkörnige lokale Positionsinformationen in den frühen Dimensionen erfassen, wo Drehungen häufiger vorkommen, und grobkörnige globale Positionsinformationen in den späteren Dimensionen, wo die Drehungen langsamer werden. Durch diese Kombination entsteht eine reichhaltigere Positionscodierung, die es dem Modell ermöglicht, Token-Positionen in einer Sequenz effektiver zu differenzieren als bei Verwendung einer einzigen Rotationsfrequenz in allen Dimensionen.

Um den Prozess zu veranschaulichen, betrachten wir einen 6-dimensionalen Abfragevektor an der Position  $t$  und  $\vartheta = 10.000$ :

$$\mathbf{q}_t = [q_t^{(1)}, q_t^{(2)}, q_t^{(3)}, q_t^{(4)}, q_t^{(5)}, q_t^{(6)}]^\top \stackrel{\text{def}}{=} [0.8, 0.6, 0.7, 0.3, 0.5, 0.4]^\top$$

Zuerst teilen wir den Vektor in drei Paare ( $d_q/2 = 3$ ) auf:

$$\mathbf{q}_t(1) = [q_t^{(1)}, q_t^{(2)}] = [0.8, 0.6]^\top$$

$$\mathbf{q}_t(2) = [q_t^{(3)}, q_t^{(4)}] = [0.7, 0.3]^\top$$

$$\mathbf{q}_t(3) = [q_t^{(5)}, q_t^{(6)}] = [0.5, 0.4]^\top$$

Jedes Paar  $p$  erfährt eine Drehung um den Winkel  $\vartheta_t$ , wobei:

$$\theta_p = \frac{1}{10000^{2(p-1)/d_q}}$$

Nehmen wir als Beispiel eine Position  $t$  von 100 an. Zunächst berechnen wir die Drehungswinkel für jedes Paar (im Bogenmaß):

$$\theta_1 = \frac{1}{10000^{2(1-1)/6}} = \frac{1}{10000^{0/6}} = 1.0000, \text{ daher: } \theta_1 t = 100.00$$

$$\theta_2 = \frac{1}{10000^{2(2-1)/6}} = \frac{1}{10000^{2/6}} \approx 0.0464, \text{ daher: } \theta_2 t = 4.64$$

$$\theta_3 = \frac{1}{10000^{2(3-1)/6}} = \frac{1}{10000^{4/6}} \approx 0.0022, \text{ daher: } \theta_3 t = 0.22$$

Für das gedrehte Paar 1 ergibt sich:

$$\text{RoPE}(\mathbf{q}_{100}^{(1)}) = \begin{bmatrix} \cos(100) & -\sin(100) \\ \sin(100) & \cos(100) \end{bmatrix} \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} \approx \begin{bmatrix} 0.86 & 0.51 \\ -0.51 & 0.86 \end{bmatrix} \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} = [0.99, 0.11]^\top$$

Für das gedrehte Paar 2 ergibt sich:

$$\text{RoPE}(\mathbf{q}_{100}^{(2)}) = \begin{bmatrix} \cos(4.64) & -\sin(4.64) \\ \sin(4.64) & \cos(4.64) \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} \approx \begin{bmatrix} -0.07 & 1.00 \\ -1.00 & -0.07 \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} = [0.25, -0.72]^\top$$

Für das gedrehte Paar 3 ergibt sich:

$$\text{RoPE}(\mathbf{q}_{100}^{(3)}) = \begin{bmatrix} \cos(0.22) & -\sin(0.22) \\ \sin(0.22) & \cos(0.22) \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix} \approx \begin{bmatrix} 0.98 & -0.21 \\ 0.21 & 0.98 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix} = [0.40, 0.50]^\top$$

**Abbildung 4.7** stellt die Paare in ihrer ursprünglichen und ihrer gedrehten Variante grafisch dar.

Der endgültige RoPE-codierte Vektor ist die Verkettung dieser Paare:

$$\text{RoPE}(\mathbf{q}_{100}) \approx [0.99, 0.11, 0.25, -0.72, 0.40, 0.50]^\top$$

Die Berechnungen für  $\text{RoPE}(\mathbf{k}_t)$  sind die gleichen wie für  $\text{RoPE}(\mathbf{q}_t)$ . In jedem Decoder-Block wird RoPE auf jede Zeile der Abfrage- ( $\mathbf{Q}$ ) und Schlüsselmatrizen ( $\mathbf{K}$ ) innerhalb des Self-Attention-Mechanismus angewendet.



Die Wertvektoren liefern nur die Informationen, die nach der Bestimmung der Attention-Gewichte ausgewählt und kombiniert werden. Da die Positionsbeziehungen bereits in der Abfrage-Schlüssel-Ausrichtung erfasst sind, benötigen Wertvektoren keine eigenen rotierenden Embeddings. Mit anderen Worten: Die Wertvektoren »liefern« einfach den Inhalt, sobald die positionsabhängige Attention erkannt hat, wo zu suchen ist.

Wie weiter oben erwähnt, werden  $\mathbf{Q}$  und  $\mathbf{K}$  generiert, indem die Eingaben des Decoder-Blocks mit den Gewichtsmatrizen  $\mathbf{W}^Q$  und  $\mathbf{W}^K$  multipliziert werden, wie **Abbildung 4.4** zeigt. RoPE wird angewendet, und zwar unmittelbar nachdem  $\mathbf{Q}$  und

**K** ermittelt wurden und bevor die Attention-Scores berechnet werden.

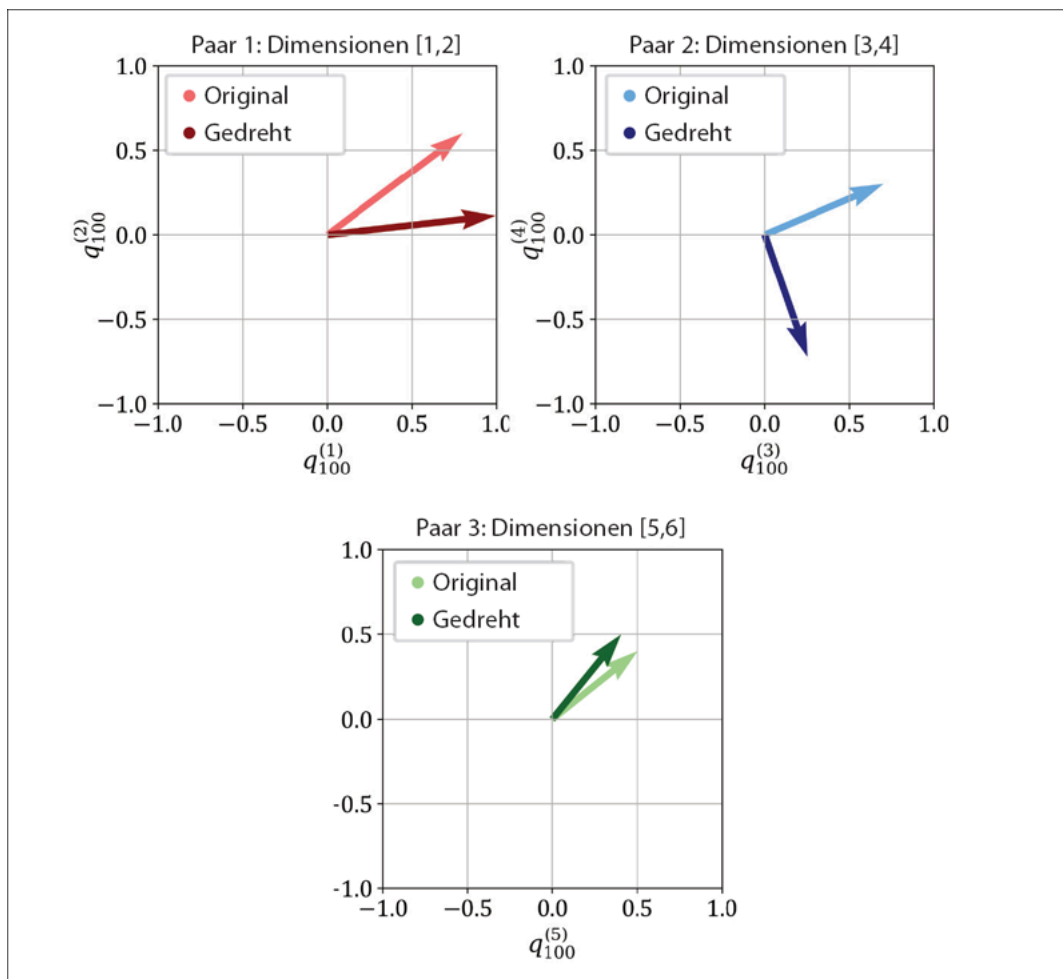


Abbildung 4.7: Die Vektorpaare in der ursprünglichen und der gedrehten Form

Da RoPE über alle Decoder-Blöcke angewendet wird, ist sichergestellt, dass die Positionsinformationen konsequent durch das Netz in seiner gesamten Tiefe fließen. [Abbildung 4.8](#) veranschaulicht die Implementierung in zwei sequenziellen Decoder-Blöcken. Hier dienen die Ausgaben des zweiten Decoder-Blocks dazu, Logits für jede Position zu berechnen. Dazu werden die Ausgaben des letzten Decoder-Blocks mit einer Matrix der Form (Dimensionalität der Embeddings, Vokabulargröße) multipliziert, die für alle Positionen gleich ist. Auf diesen Teil gehen wir ausführlicher ein, wenn wir das Decoder-Modell in Python implementieren.

Der bisher beschriebene Self-Attention-Mechanismus würde ohne Änderungen funktionieren. Transformer nutzen jedoch in der Regel eine erweiterte Version, die sogenannte *Multi-Head-Attention* (Attention mit mehreren Köpfen). Dadurch kann sich das Modell auf mehrere Aspekte von Informationen gleichzeitig konzentrieren. Beispielsweise könnte ein Attention-Kopf syntaktische Beziehungen erfassen, ein anderer vielleicht semantische Ähnlichkeiten betonen, und ein dritter könnte weitreichende Abhängigkeiten zwischen Tokens erkennen.

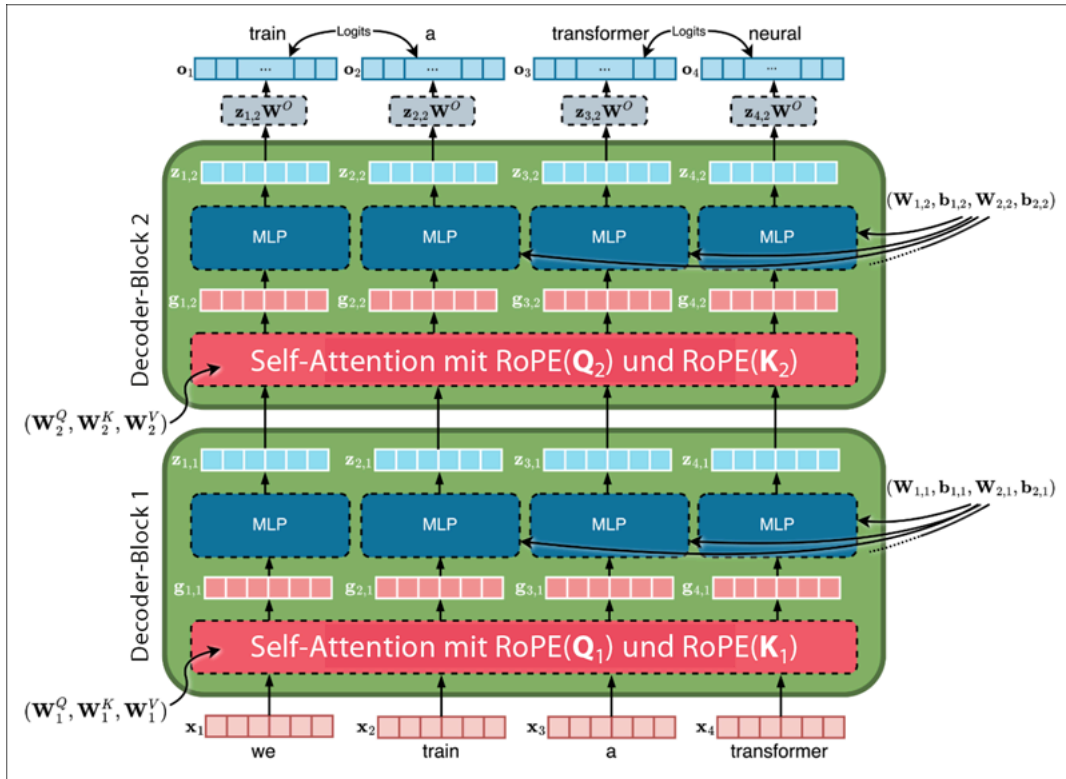


Abbildung 4.8: RoPE-Implementierung mit zwei sequenziellen Decoder-Blöcken

## Multi-Head-Attention

Wenn Sie die Funktionsweise der Self-Attention einmal verstanden haben, ist es nur ein kleiner Schritt bis zur Multi-Head-Attention. Für jeden Kopf (Head)  $h$ , von 1 bis  $H$ , gibt es ein separates Triplet von Attention-Matrizen:

$$\left\{ \left( \mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V \right) \right\}_{h \in 1, \dots, H}$$

Jedes Triplet wird auf die Eingabevektoren  $\mathbf{x}_1, \dots, \mathbf{x}_4$  angewendet, was  $H$  Matrizen  $\mathbf{G}_h$  erzeugt. Für jeden Kopf ergeben sich vier Vektoren  $\mathbf{g}_{h,1}, \dots, \mathbf{g}_{h,4}$ , wie [Abbildung 4.9](#) für drei Köpfe ( $H = 3$ ) zeigt. Wie Sie sehen, verarbeitet der Multi-Head-Self-Attention-Mechanismus eine Eingabesequenz über mehrere Self-Attention-»Köpfe«. Bei drei Köpfen beispielsweise berechnet jeder Kopf unabhängig von den anderen die Self-Attention-Scores für die Eingabetokens. RoPE wird in jedem Kopf separat angewendet.

Alle Eingabetokens  $\mathbf{x}_1, \dots, \mathbf{x}_4$  werden von allen drei Köpfen verarbeitet, wodurch die Ausgabematrizen  $\mathbf{G}_1, \mathbf{G}_2$  und  $\mathbf{G}_3$  entstehen. Jede Matrix  $\mathbf{G}_h$  hat so viele Zeilen, wie es Eingabetokens gibt, d.h., jeder Kopf erzeugt ein Embedding für jedes Token. Die Embedding-Dimensionalität jeder  $\mathbf{G}_h$  wird auf ein Drittel der gesamten Embedding-Dimensionalität reduziert. Infolgedessen gibt jeder Kopf im Vergleich zur ursprünglichen Embedding-Größe Embeddings niedrigerer Dimensionalität aus.

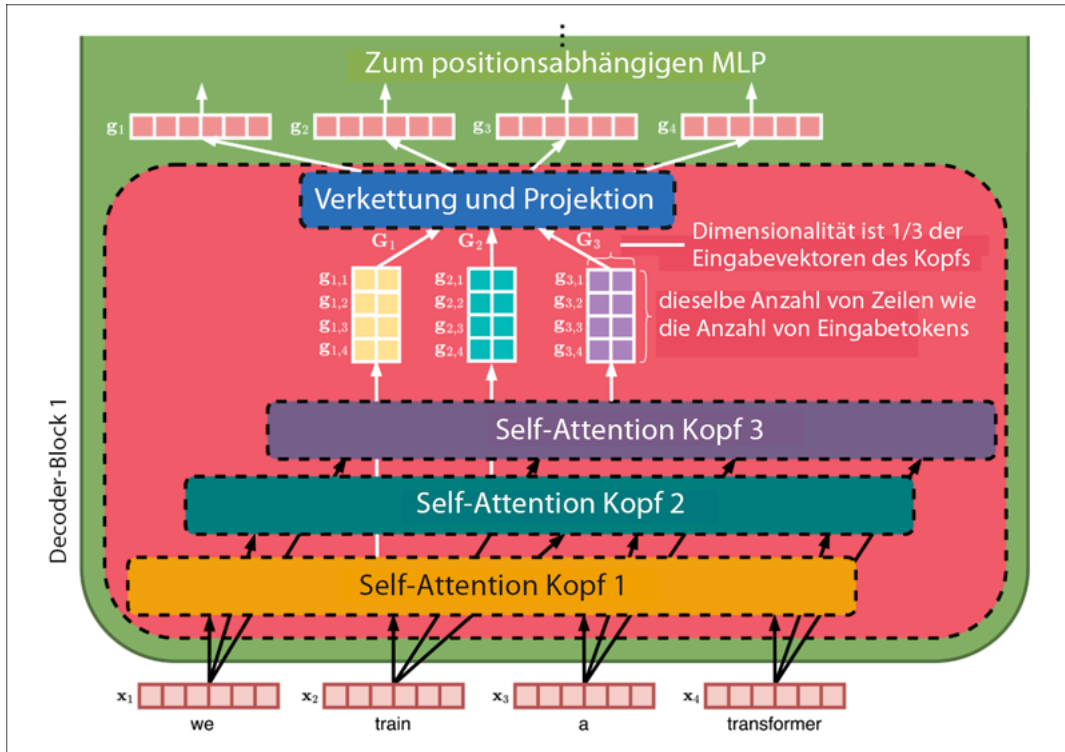


Abbildung 4.9: Self-Attention mit drei Köpfen

Die Ausgaben der drei Köpfe werden entlang der Embedding-Dimension in der *Verketzungs- und Projektionsschicht* verkettet. Dabei entsteht eine einzelne Matrix, die Informationen aus allen Köpfen integriert. Diese Matrix wird dann durch die Projektionsmatrix  $\mathbf{W}^O$  transformiert, wodurch die endgültige Ausgabematrix  $\mathbf{G}$  entsteht. Diese Ausgabe wird an das positionsbezogene MLP übergeben (siehe [Abbildung 4.10](#)).

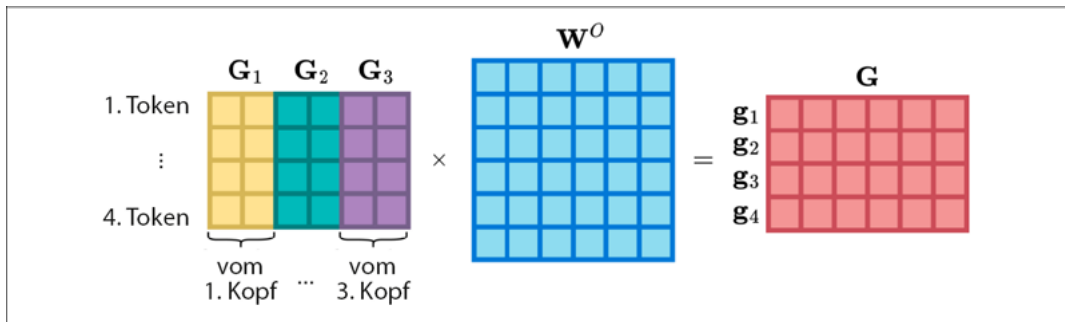


Abbildung 4.10: Die Matrix aus der Verkettungs- und Projektionsschicht wird durch die Projektionsmatrix transformiert, was eine endgültige Ausgabematrix ergibt.

Die Verkettung der Matrizen  $\mathbf{G}_1$ ,  $\mathbf{G}_2$  und  $\mathbf{G}_3$  stellt die ursprüngliche Embeddings-Dimensionalität wieder her (z. B. 6 in diesem Fall). Allerdings ermöglicht die Anwendung der trainierbaren Parametermatrix  $\mathbf{W}^O$  dem Modell, die Informationen der Köpfe effektiver zu kombinieren als durch bloße Verkettung.



Moderne LLMs verwenden oftmals bis zu 128 Köpfe.

Mittlerweile sollten die Leserinnen und Leser die Transformer-Modellarchitektur im Großen und Ganzen verstehen. Zwei wichtige technische Details müssen aber noch untersucht werden: die Normalisierung der Schichten und die Residualverbindungen, beides wesentliche Komponenten, die die Effektivität des Transformers ermöglichen. Beginnen wir mit Residualverbindungen.

## Residualverbindungen

*Residualverbindungen* (oder *Skip-Verbindungen*) sind ein wesentlicher Bestandteil der Transformer-Architektur. Sie lösen das Problem der verschwindenden Gradienten in tiefen neuronalen Netzen und ermöglichen das Training von viel tieferen Modellen.

Ein Netz, das mehr als zwei Schichten umfasst, gilt als *tiefes neuronales Netz* (*Deep Neural Network*). Das Training derartiger Netze wird als *Deep Learning* bezeichnet. Vor der Einführung von ReLU und Residualverbindungen hat das *Problem der verschwindenden Gradienten* die Tiefe der Netze stark eingeschränkt. Wie Sie wissen, sind beim Gradientenabstieg partielle Ableitungen im Spiel, um alle Parameter zu aktualisieren, was durch kleine Schritte in die entgegengesetzte Richtung des Gradienten erfolgt. In tieferen Netzen werden diese Aktualisierungen bereits in frühen Schichten (die näher an der Eingabe liegen) sehr klein, was effektiv einem Stillstand der Parameteraktualisierung gleichkommt. Residualverbindungen verstärken diese Aktualisierungen, indem sie für den Gradienten Wege schaffen, um bestimmte Schichten zu »überspringen« (daher die Bezeichnung »Skip-Verbindungen«).

Um das Problem der verschwindenden Gradienten besser zu verstehen, analysieren wir ein neuronales Netz mit drei Schichten, ausgedrückt als zusammengesetzte Funktion:

$$f(x) = f_3(f_2(f_1(x))),$$

wobei  $f_1$  für die erste Schicht,  $f_2$  für die zweite und  $f_3$  für die dritte Schicht (Ausgabeschicht) steht. Diese Funktionen sollen folgendermaßen definiert sein:

$$z = f_1(x) \stackrel{\text{def}}{=} w_1x + b_1$$

$$r = f_2(z) \stackrel{\text{def}}{=} w_2z + b_2$$

$$y = f_3(r) \stackrel{\text{def}}{=} w_3r + b_3$$

Hier sind  $w_l$  und  $b_l$  skalare Gewichte und Bias-Werte für jede Schicht  $l \in \{1,2,3\}$ .

Die Verlustfunktion  $L$  definieren wir in Bezug auf die Netzausgabe  $f(x)$  und das wahre Label  $y$  als  $L(f(x), y)$ . Der Gradient der Verlustfunktion  $L$  in Bezug auf  $w_1$  in der Form  $\frac{\partial L}{\partial w_1}$  ist wie folgt gegeben:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1}$$

Hierbei sind:

$$\frac{\partial f_3}{\partial f_2} = w_3, \frac{\partial f_2}{\partial f_1} = w_2, \frac{\partial f_1}{\partial w_1} = x$$

Wir können also schreiben:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot w_3 \cdot w_2 \cdot x$$

Das Problem der verschwindenden Gradienten tritt auf, wenn Gewichte wie  $w_2$  und  $w_3$  klein sind (kleiner als 1). Miteinander multipliziert, ergeben sie noch kleinere Werte, sodass der Gradient für frühere Gewichte wie  $w_1$  gegen null geht. Dieses Problem ist bei Netzen mit vielen Schichten besonders gravierend.

Nehmen wir LLMs als Beispiel. Derartige Netze enthalten oft 32 oder mehr Decoder-Blöcke. Der Einfachheit halber nehmen wir an, dass alle Blöcke als vollständig verbundene Schichten realisiert sind. Wenn der durchschnittliche Gewichtswert bei 0.5 liegt, wird der Gradient für die Parameter der Eingabeschicht zu  $0.5^{32} \approx 0.0000000002$ . Das ist ein äußerst kleiner Wert. Nach der Multiplikation mit der Lernrate werden die Aktualisierungen in den frühen Schichten vernachlässigbar. Infolgedessen kann das Netz nicht mehr effektiv lernen.

Residualverbindungen bieten eine Lösung für das Problem der verschwindenden Gradienten, indem sie Abkürzungen im Pfad der Gradientenberechnung schaffen. Die Grundidee ist einfach: Anstatt nur die Ausgabe einer Schicht an die nächste weiterzuleiten, wird die Eingabe der Schicht zu ihrer Ausgabe addiert. Mathematisch lässt sich das wie folgt formulieren:

$$y = f(x) + x,$$

wobei  $x$  die Eingabe,  $f(x)$  die berechnete Funktion der Schicht und  $y$  die Ausgabe ist. Diese Addition bildet die Residualverbindung. [Abbildung 4.11](#) stellt dies grafisch dar.

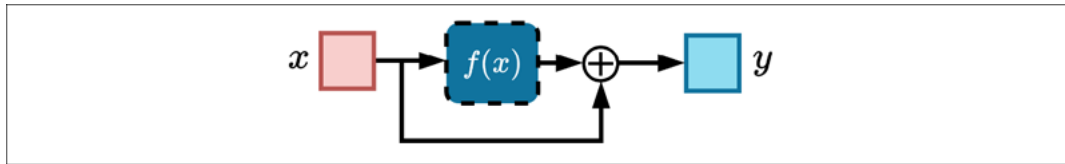


Abbildung 4.11: Schematische Darstellung einer Residualverbindung

Wie [Abbildung 4.11](#) zeigt, wird die Eingabe  $x$  sowohl durch die Schicht verarbeitet (als  $f(x)$  dargestellt) als auch direkt zur Ausgabe der Schicht addiert.

In unser 3-Schichten-Netz wollen wir nun Residualverbindungen einführen. Dabei werden Sie sehen, wie sich dies auf die Gradientenberechnung auswirkt und das Problem der verschwindenden Gradienten entschärft. Beginnend mit dem ursprünglichen Netz  $f(x) = f_3(f_2(f_1(x)))$ , fügen wir den Schichten 2 und 3 Residualverbindungen hinzu:

$$z \leftarrow f_1(x) \stackrel{\text{def}}{=} w_1x + b_1$$

$$r \leftarrow f_2(z) \stackrel{\text{def}}{=} w_2z + b_2 + z$$

$$y \leftarrow f_3(r) \stackrel{\text{def}}{=} w_3r + b_3 + r$$

Unsere zusammengesetzte Funktion wird zu:

$$f(x) = w_3[w_2(w_1x + b_1) + b_2 + w_1x + b_1] + b_3 + w_2(w_1x + b_1) + b_2 + w_1x + b_1$$

Nun berechnen wir den Gradienten des Verlusts  $L$  in Bezug auf  $w_1$ :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1}$$

Wir erweitern  $\frac{\partial f}{\partial w_1}$ :

$$\frac{\partial L}{\partial w_1} = \frac{\partial}{\partial w_1} [(w_3(w_2(w_1x + b_1) + b_2 + (w_1x + b_1)) + b_3) + (w_2(w_1x + b_1) + b_2 + (w_1x + b_1))] =$$

Demnach sieht der vollständige Gradient wie folgt aus:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot (w_3w_2 + w_3 + w_2 + 1) \cdot x$$

Vergleichen Sie dies mit unserem ursprünglichen Gradienten ohne Residualverbindungen:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot w_3 \cdot w_2 \cdot x$$

Wir stellen fest, dass Residualverbindungen drei zusätzliche Terme einführen:  $w_3$ ,  $w_2$  und 1. Der addierte konstante Term 1 garantiert, dass der Gradient nicht vollständig verschwindet, selbst wenn  $w_2$  und  $w_3$  klein sind.

Wenn zum Beispiel wie im vorherigen Fall  $w_2 = w_3 = 0.5$  ist:

- **ohne Residualverbindungen:**  $0.5 \cdot 0.5 = 0.25$
- **mit Residualverbindungen:**  $0.5 \cdot 0.5 + 0.5 + 0.5 + 1 = 2.25$

[Abbildung 4.12](#) stellt einen Decoder-Block mit Residualverbindungen dar.



Abbildung 4.12: Decoder-Block mit Residualverbindungen

Wie [Abbildung 4.12](#) zeigt, enthält jeder Decoder-Block zwei Residualverbindungen. Diese sind jetzt wie Python-Objekte benannt, die wir in Kürze implementieren. Außerdem sind zwei RMSNorm-Schichten hinzugekommen. Der folgende Abschnitt erläutert ihren Zweck.

## RMS-Normalisierung

Die RMSNorm-Schicht normalisiert den Eingabevektor mit dem quadratischen Mittelwert (*Root Mean Square Normalization*)<sup>3</sup>. Diese Operation findet statt, kurz bevor der Vektor in die Self-Attention-Schicht und das positionsbezogene MLP gelangt. Wir wollen dies anhand eines dreidimensionalen Vektors veranschaulichen.

Angenommen, wir haben einen Vektor  $\mathbf{x} = [x^{(1)}, x^{(2)}, x^{(3)}]^T$ . Um RMS-Normalisierung anzuwenden, berechnen wir zuerst die Wurzel aus dem quadratischen Mittelwert (RMS) des Vektors:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{3} \sum_{i=1}^3 (x^{(i)})^2} = \sqrt{\frac{1}{3} [(x^{(1)})^2 + (x^{(2)})^2 + (x^{(3)})^2]}$$

Dann normalisieren wir den Vektor, indem wir jede Komponente durch den RMS-Wert teilen, um  $\tilde{\mathbf{x}}$  zu erhalten:

$$\tilde{\mathbf{X}} = \frac{\mathbf{X}}{\text{RMS}(\mathbf{X})} = \left[ \frac{x^{(1)}}{\text{RMS}(\mathbf{X})}, \frac{x^{(2)}}{\text{RMS}(\mathbf{X})}, \frac{x^{(3)}}{\text{RMS}(\mathbf{X})} \right]^T$$

Schließlich wenden wir den Skalierungsfaktor  $\boldsymbol{\gamma}$  auf jede Dimension von  $\tilde{\mathbf{x}}$  an:

$$\bar{\mathbf{x}} = \text{RMSNorm}(\mathbf{x}) \stackrel{\text{def}}{=} \boldsymbol{\gamma} \odot \tilde{\mathbf{x}} = \left[ \gamma^{(1)} \tilde{x}^{(1)}, \gamma^{(2)} \tilde{x}^{(2)}, \gamma^{(3)} \tilde{x}^{(3)} \right]^T$$

Hier kennzeichnet  $\odot$  das *elementweise Produkt*. Der Vektor  $\mathbf{y}$  ist ein trainierbarer Parameter, und jede RMSNorm-Schicht hat ihren eigenen unabhängigen Parameter  $\mathbf{y}$ .

In erster Linie soll RMSNorm das Training stabilisieren, indem die Skala der Eingabe für jede Schicht konsistent gehalten wird. Dies verbessert die numerische Stabilität und hilft, übermäßig große oder kleine Gradientenaktualisierungen zu vermeiden.

Nachdem wir nun die wichtigsten Komponenten der Transformer-Architektur behandelt haben, wollen wir zusammenfassen, wie ein Decoder-Block seine Eingaben verarbeitet:

1. Die Eingabe-Embeddings  $\mathbf{x}_t$  durchlaufen zuerst eine RMS-Normalisierung.
2. Die normalisierten Embeddings  $\bar{\mathbf{x}}_t$  werden durch den Multi-Head-Self-Attention-Mechanismus verarbeitet, wobei RoPE auf Schlüssel- und Abfragevektoren angewendet wird.
3. Die Self-Attention-Ausgabe  $\mathbf{g}_t$  wird zur ursprünglichen Eingabe  $\mathbf{x}_t$  addiert (Residualverbindung).
4. Diese Summe,  $\hat{\mathbf{g}}_t$ , durchläuft erneut eine RMS-Normalisierung.
5. Die normalisierte Summe  $\bar{\hat{\mathbf{g}}}_t$  wird durch das MLP geleitet.
6. Die Ausgabe des Perzeptrons  $\mathbf{z}_t$  wird zum Prä-RMS-Normalisierungsvektor  $\hat{\mathbf{g}}_t$  addiert (eine weitere Residualverbindung).
7. Das Ergebnis  $\hat{\mathbf{z}}_t$  ist die Ausgabe des Decoder-Blocks und dient als Eingabe für den nächsten Block (oder die letzte Ausgabeschicht, wenn es der letzte Block gewesen ist).

Diese Sequenz wird für jeden Decoder-Block im Transformer wiederholt.

## Schlüssel-Wert-Caching

Während des Trainings kann der Decoder alle Positionen parallel verarbeiten, da er in jedem Block die Abfrage-, Schlüssel- und Wertmatrizen  $\mathbf{Q} = \mathbf{XW}^Q$ ,  $\mathbf{K} = \mathbf{XW}^K$  und  $\mathbf{V} = \mathbf{XW}^V$  für die gesamte Sequenz  $\mathbf{X}$  berechnet. Bei einer autoregressiven (von links nach rechts ablaufenden) *Inferenz* müssen die Tokens jedoch einzeln erzeugt werden. Normalerweise müssten wir jedes Mal, wenn wir ein neues Token generieren, folgende Schritte absolvieren:

1. Die Schlüssel-, Abfrage- und Wertvektoren für das neue Token berechnen.
2. Die Schlüssel- und Wertmatrizen für alle vorherigen Tokens neu berechnen.
3. Diese Matrizen mit den Schlüssel- und Wertvektoren des neuen Tokens zusammenführen, um die Self-Attention für das neue Token zu berechnen.

Das *Schlüssel-Wert-Caching* überspringt Schritt 2, indem es die Schlüssel- und Wertmatrizen früherer Tokens speichert und so wiederholte Berechnungen vermeidet. Da  $\mathbf{W}^K$  und  $\mathbf{W}^V$  nach dem Training fest sind, bleiben die Schlüssel- und Wertvektoren früherer Tokens während der Inferenz konstant. Diese Vektoren lassen sich speichern (»cachen«), nachdem sie einmal berechnet wurden. Für jedes neue Token werden

- seine Schlüssel- und Wertvektoren mit  $\mathbf{W}^K$  und  $\mathbf{W}^V$  berechnet,
- diese Vektoren zur Self-Attention an die gecachten Schlüssel-Wert-Paare angehängt.

Abfragevektoren werden jedoch nicht gecacht, weil sie vom aktuell verarbeiteten Token abhängen. Jedes Mal, wenn ein neues Token hinzugefügt wird, muss sein Abfragevektor spontan berechnet werden, um alle zwischengespeicherten Schlüssel und Werte zu berücksichtigen.

Durch diesen Ansatz ist es nicht mehr erforderlich, den Rest der Sequenz erneut zu verarbeiten, was den Rechenaufwand bei langen Sequenzen erheblich verringert. In jedem Decoder-Block werden die gecachten Schlüssel und Werte pro Attention-Kopf mit den Formen  $(L \times d_h)$  für beide Matrizen gespeichert, wobei  $L$  mit jedem neuen Token um eins wächst und  $d_h$  die Dimensionalität der Abfrage-, Schlüssel- und Wertvektoren für diesen Kopf ist. Bei einem Modell mit  $H$  Attention-Köpfen haben die kombinierten Schlüssel- und Wert-Caches in dem Decoder-Block die Form  $(H \times L \times d_h)$ .



RoPE wendet positionsabhängige Drehungen auf die Vektoren an, was jedoch mit dem Caching nicht in Konflikt kommt. Trifft ein neues Token ein, nimmt es einfach den nächsten verfügbaren Positionsindex (wenn die Sequenz  $L$  Tokens umfasst, bekommt das neue Token die Position  $L + 1$ ), während zuvor verarbeitete Tokens ihre ursprünglichen Positionen von 1 bis  $L$  beibehalten. Das bedeutet, dass die gecachten Schlüssel und Werte, die bereits entsprechend ihrer jeweiligen Position gedreht wurden, unverändert bleiben. Die Drehung wird nur auf das neue Token an der Position  $L + 1$  angewendet.

Nachdem Sie nun die Funktionsweise eines Transformers kennen, können wir mit der Programmierung beginnen.

## Transformer in Python

Um den Decoder in Python zu implementieren, definieren wir zunächst die Klasse `AttentionHead`:

```
class AttentionHead(nn.Module):
    def __init__(self, emb_dim, d_h):
        super().__init__()
        self.W_Q = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_K = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_V = nn.Parameter(torch.empty(emb_dim, d_h))
        self.d_h = d_h
    def forward(self, x, mask):
        Q = x @ self.W_Q ❶
        K = x @ self.W_K
        V = x @ self.W_V ❷
        Q, K = rope(Q), rope(K) ❸
        scores = Q @ K.transpose(-2, -1) / math.sqrt(self.d_h) ❹
        masked_scores = scores.masked_fill(mask == 0, float("-inf")) ❺
        attention_weights = torch.softmax(masked_scores, dim=-1) ❻
        return attention_weights @ V ❼
```

Diese Klasse implementiert einen einzelnen Attention-Kopf im Multi-Head-Attention-Mechanismus. Im Konstruktor initialisieren wir drei trainierbare Gewichtsmatrizen: die Abfragematrix `W_Q`, die Schlüsselmatrix `W_K` und die Wertmatrix `W_V`. Jede dieser Matrizen ist ein `Parameter`-Tensor der Form `(emb_dim, d_h)`, wobei `emb_dim` die Eingabe-Embedding-Dimension und `d_h` die Dimensionalität der Abfrage-, Schlüssel- und Wertvektoren für diesen Attention-Kopf ist.

In der Methode `forward`

- multiplizieren die Zeilen ❶ und ❷ den Eingabevektor `x` mit den jeweiligen Gewichtsmatrizen, um die Abfrage-, Schlüssel- und Wertmatrizen zu berechnen. Da `x` die Form `(batch_size, seq_len, emb_dim)` hat, haben `Q`, `K` und `V` jeweils die Form `(batch_size, seq_len, d_h)`.
- wendet Zeile ❸ die rotierende Positionscodierung auf `Q` und `K` an. Nachdem die Abfrage- und Schlüsselvektoren gedreht wurden, berechnet Zeile ❹ die Attention-Scores. Im Einzelnen
  - vertauscht `K.transpose(-2, -1)` die letzten beiden Dimensionen von `K`. Hat `K` die Form `(batch_size, seq_len, d_h)`, wird die Matrix in die Form `(batch_size, d_h, seq_len)` transponiert. Damit ist `K` für die Matrixmultiplikation mit `Q` vorbereitet.
  - führt `Q @ K.transpose(-2, -1)` eine Matrixmultiplikation im Stapel durch, was einen Tensor von Attention-Scores der Form `(batch_size, seq_len, seq_len)` ergibt.
  - dividieren wir durch `sqrt(d_h)` aus Gründen der numerischen Stabilität (wie im Abschnitt »Self-Attention« auf [Seite 116](#) erwähnt).



Wird der Matrixmultiplikationsoperator `@` auf Tensoren mit mehr als zwei Dimensionen angewendet, verwendet PyTorch *Broadcasting*. Diese Technik behandelt Dimensionen, die nicht direkt mit dem Operator `@` kompatibel sind, der normalerweise nur für zweidimensionale Tensoren (Matrizen) definiert ist. In diesem Fall behandelt PyTorch die erste Dimension als Stapeldimension und führt die Matrixmultiplikation separat für jedes Beispiel im Stapel (Batch) aus. Dieses Verfahren wird als *Batch-Matrixmultiplikation* bezeichnet.

- Zeile ❺ wendet die kausale Maske an. Der Tensor `mask` hat die Form `(seq_len, seq_len)` und enthält Nullen und Einsen. Die Funktion `masked_fill` ersetzt alle Zellen in der Eingabematrix, bei denen `mask == 0` ist, durch negative Unendlichkeit `(-inf)`. Dies verhindert, dass zukünftige Tokens beachtet werden. Da `mask` die Stapeldimension fehlt, `scores` sie aber enthält, wendet PyTorch per Broadcasting `mask` auf die `scores` jeder Sequenz im Stapel an.

- Zeile ⑥ wendet Softmax auf die scores entlang der letzten Dimension an, was sie in Attention-Gewichte umwandelt. Dann berechnet Zeile ⑦ die Ausgabe, indem diese Attention-Gewichte mit V multipliziert werden. Die resultierende Ausgabe hat die Form (batch\_size, seq\_len, d\_h).

Ausgehend von der Klasse AttentionHead können wir nun die Klasse MultiHeadAttention definieren:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        d_h = emb_dim // num_heads ①
        self.heads = nn.ModuleList([AttentionHead(emb_dim, d_h) for _ in range(num_heads)] ②)
        self.W_0 = nn.Parameter(torch.empty(emb_dim, emb_dim)) ③
    def forward(self, x, mask):
        head_outputs = [head(x, mask) for head in self.heads] ④
        x = torch.cat(head_outputs, dim=-1) ⑤
        return x @ self.W_0 ⑥
```

Im Konstruktor

- berechnet Zeile ① mit d\_h die Dimensionalität jedes Attention-Kopfs, indem die Embedding-Dimensionalität des Modells emb\_dim durch die Anzahl der Köpfe geteilt wird.
- erzeugt Zeile ② eine ModuleList mit num\_heads Instanzen von AttentionHead. Jeder Kopf übernimmt als Eingabe die Dimensionalität emb\_dim und gibt einen Vektor der Größe d\_h aus.
- initialisiert Zeile ③ mit W\_0 eine lernbare Projektionsmatrix der Form (emb\_dim, emb\_dim), um die Ausgaben aus allen Attention-Köpfen zu kombinieren.

In der Methode forward

- wendet Zeile ④ jeden Attention-Kopf auf die Eingabe x der Form (batch\_size, seq\_len, emb\_dim) an. Die Ausgabe eines jeden Kopfs hat die Form (batch\_size, seq\_len, d\_h).
- verkettet Zeile ⑤ alle Ausgaben der Köpfe entlang der letzten Dimension. Das Ergebnis x hat die Form (batch\_size, seq\_len, emb\_dim), da num\_heads \* d\_h = emb\_dim ist.
- multipliziert Zeile ⑥ die verkettete Ausgabe mit der Projektionsmatrix W\_0. Die Ausgabe hat die gleiche Form wie die Eingabe.

Da wir nun Multi-Head-Attention realisiert haben, fehlt für den Decoder-Block nur das positionsbezogene mehrschichtige Perzeptron (MLP). Der folgende Code definiert es:

Im Konstruktor:

```
class MLP(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.W_1 = nn.Parameter(torch.empty(emb_dim, emb_dim * 4))
        self.B_1 = nn.Parameter(torch.empty(emb_dim * 4))
        self.W_2 = nn.Parameter(torch.empty(emb_dim * 4, emb_dim))
        self.B_2 = nn.Parameter(torch.empty(emb_dim))
    def forward(self, x):
        x = x @ self.W_1 + self.B_1 ①
        x = torch.relu(x) ②
        x = x @ self.W_2 + self.B_2 ③
        return x
```

Im Konstruktor initialisieren wir lernbare Gewichte und Bias-Werte. In der Methode forward

- multipliziert Zeile ① die Eingabe x mit der Gewichtsmatrix W\_1 und addiert den Bias-Vektor B\_1. Die Eingabe hat die Form (batch\_size, seq\_len, emb\_dim), sodass das Ergebnis die Form (batch\_size, seq\_len, emb\_dim \* 4) erhält.
- wendet Zeile ② die ReLU-Aktivierungsfunktion elementweise an und fügt somit Nichtlinearität hinzu.
- multipliziert Zeile ③ das Ergebnis mit der zweiten Gewichtsmatrix W\_2 und addiert den Bias-Vektor B\_2, was die Dimensionalität wieder zurück auf (batch\_size, seq\_len, emb\_dim) reduziert.

Die erste lineare Transformation erweitert die Embedding-Dimensionalität auf das Vierfache ( $emb\_dim * 4$ ), um dem Netz eine größere Kapazität für das Lernen komplexer Muster und Beziehungen zwischen Variablen zu verschaffen. Der Faktor  $4x$  sorgt für ein Gleichgewicht zwischen Expressivität und Effizienz. Nachdem die Dimensionalität erweitert wurde, wird sie wieder auf die ursprüngliche Embedding-Dimensionalität ( $emb\_dim$ ) komprimiert. Dadurch wird die Kompatibilität mit Residualverbindungen gewährleistet, die übereinstimmende Dimensionalitäten voraussetzt. Empirische Ergebnisse belegen, dass dieser Erweitern-und-Komprimieren-Ansatz einen effektiven Kompromiss zwischen Rechenkosten und Performance darstellt.

Da nun alle Komponenten definiert sind, können wir den vollständigen Decoder-Block einrichten:

```
class DecoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        self.norm1 = RMSNorm(emb_dim)
        self.attn = MultiHeadAttention(emb_dim, num_heads)
        self.norm2 = RMSNorm(emb_dim)
        self.mlp = MLP(emb_dim)
    def forward(self, x, mask):
        attn_out = self.attn(self.norm1(x), mask) ❶
        x = x + attn_out ❷
        mlp_out = self.mlp(self.norm2(x)) ❸
        x = x + mlp_out ❹
        return x
```

Die Klasse `DecoderBlock` verkörpert einen einzelnen Decoder-Block in einem Transformer-Modell. Im Konstruktor richten wir die erforderlichen Schichten ein: zwei `RMSNorm`-Schichten, eine `MultiHeadAttention`-Instanz (konfiguriert mit der Embedding-Dimensionalität und der Anzahl der Köpfe) und eine MLP-Schicht.

In der Methode `forward`

- wendet Zeile ❶ `RMSNorm` auf die Eingabe `x` an, die die Form  $(batch\_size, seq\_len, emb\_dim)$  hat. Die Ausgabe von `RMSNorm` behält diese Form bei. Dieser normalisierte Tensor wird dann an die Multi-Head-Attention-Schicht weitergeleitet, die einen Tensor mit der gleichen Form ausgibt.
- fügt Zeile ❷ eine Residualverbindung hinzu, indem die Attention-Ausgabe `attn_out` mit der ursprünglichen Eingabe `x` kombiniert wird. Die Form ändert sich nicht.
- wendet Zeile ❸ die zweite `RMSNorm` auf das Ergebnis der Residualverbindung an, wobei die gleiche Form beibehalten wird. Dieser normalisierte Tensor wird dann durch das MLP geleitet, das einen weiteren Tensor mit der Form  $(batch\_size, seq\_len, emb\_dim)$  ausgibt.
- fügt Zeile ❹ eine zweite Residualverbindung hinzu, die `mlp_out` mit seiner nicht normalisierten Eingabe kombiniert. Die endgültige Form der Ausgabe ist  $(batch\_size, seq\_len, emb\_dim)$ , bereit für den nächsten Decoder-Block oder die letzte Ausgabeschicht.

Nachdem wir den Decoder-Block definiert haben, können wir nun ein Decoder-Transformer-Sprachmodell aufbauen, indem wir mehrere Decoder-Blöcke sequenziell übereinander stapeln:

```
class DecoderLanguageModel(nn.Module):
    def __init__(
        self, vocab_size, emb_dim,
        num_heads, num_blocks, pad_idx
    ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_dim, padding_idx=pad_idx)
        ) ❶
        self.layers = nn.ModuleList([
            DecoderBlock(emb_dim, num_heads) for _ in range(num_blocks)
        ]) ❷
        self.output = nn.Parameter(torch.rand(emb_dim, vocab_size)) ❸
    def forward(self, x):
        x = self.embedding(x) ❹
```

```

_, seq_len, _ = x.shape
mask = torch.tril(torch.ones(seq_len, seq_len, device=x.device)) ❸
for layer in self.layers: ❹
    x = layer(x, mask)
return x @ self.output ❺

```

Im Konstruktor der Klasse `DecoderLanguageModel`

- erzeugt Zeile ❶ eine Embedding-Schicht, die die Indizes der Eingabetokens in dichte Vektoren umwandelt. Der Parameter `padding_idx` gibt die ID des Padding-Tokens an und stellt sicher, dass Padding-Tokens auf Nullvektoren abgebildet werden.
- erzeugt Zeile ❷ eine `ModuleList` mit `num_blocks` `DecoderBlock`-Instanzen, die den Stapel der Decoder-Schichten bilden.
- definiert Zeile ❸ eine Matrix, um die Ausgabe des letzten Decoder-Blocks auf Logits über das Vokabular zu projizieren und so die Vorhersage des nächsten Tokens zu ermöglichen.

In der Methode `forward`

- konvertiert Zeile ❹ die Indizes der Eingabetokens in Embeddings. Der Eingabe-Tensor `x` hat die Form `(batch_size, seq_len)`, und die Form der Ausgabe ist `(batch_size, seq_len, emb_dim)`.
- erzeugt Zeile ❺ die *kausale Maske*.
- wendet Zeile ❻ jeden Decoder-Block auf den Eingabe-Tensor `x` mit der Form `(batch_size, seq_len, emb_dim)` an und liefert damit einen Ausgabe-Tensor der gleichen Form. Jeder Block verfeinert die Sequenz und gibt sie an den nächsten, bis zum letzten Block, weiter.
- projiziert Zeile ❼ die Ausgabe des letzten Decoder-Blocks auf Logits in der Größe des Vokabulars, indem sie sie mit der Matrix `self.output`, die die Form `(emb_dim, vocab_size)` hat, multipliziert. Nach dieser Batch-Matrixmultiplikation hat die endgültige Ausgabe die Form `(batch_size, seq_len, vocab_size)` und liefert Scores für jedes Token im Vokabular an jeder Position in der Eingabesequenz. Diese Ausgabe kann dann verwendet werden, um die Vorhersagen des Modells zu erzeugen, worauf wir im nächsten Kapitel eingehen werden.

Die Trainingsschleife für `DecoderLanguageModel` ist die gleiche wie für das RNN (siehe den Abschnitt »Ein RNN-Sprachmodell trainieren« auf [Seite 105](#)), sodass wir sie hier der Kürze halber nicht wiederholen. Auch die Implementierungen von `RMSNorm` und `RoPE` überspringen wir. Die Trainingsdaten werden genau wie beim RNN vorbereitet: Die Zielsequenz wird um eine Position relativ zur Eingabesequenz verschoben, wie im Abschnitt »Die Klassen `Dataset` und `DataLoader`« auf [Seite 107](#) beschrieben. Den vollständigen Code für das Training des Decoder-Sprachmodells finden Sie im Notebook unter <https://www.thelmlbook.com/nb/4.1>.

Im Notebook habe ich folgende Hyperparameterwerte verwendet: `emb_dim = 128`, `num_heads = 8`, `num_blocks = 2`, `batch_size = 128`, `learning_rate = 0.001`, `num_epochs = 1` und `context_size = 30`. Mit diesen Einstellungen hat das Modell eine Perplexität von 55,19 erreicht und damit den RNN-Wert von 72,23 übertroffen. Dies ist ein gutes Ergebnis angesichts der vergleichbaren Anzahl von trainierbaren Parametern (8.621.963 für den Transformer vs. 8.292.619 für das RNN). Die eigentlichen Stärken der Transformer treten jedoch bei größeren Skalen von Modellgröße, Kontextlänge und Trainingsdaten deutlicher hervor. Es verbietet sich natürlich aus praktischen Gründen, Experimente in dieser Größenordnung für dieses Buch durchzuführen.

Sehen wir uns einige Fortsetzungen des Prompts »The President« an, die das Decoder-Modell in späteren Trainingsschritten erzeugt hat:

```

The President has been in the process of a new deal to make a decision on the issue.
The President 's office said the government had " no intention of making any mistakes
''.
The President of the United States has been a key figure for the first time in the past
## years.

```

Die #-Zeichen in den Trainingsdaten stehen für einzelne Ziffern. Zum Beispiel steht ## wahrscheinlich für die Anzahl der Jahre.

\*\*\*

Herzlichen Glückwunsch, wenn Sie es bis hierher geschafft haben! Sie verstehen jetzt die Funktionsweise von Sprachmodellen. Aber das Verständnis dieser Mechanismen allein genügt nicht, um zu begreifen, wozu moderne Sprachmodelle in der Lage sind. Für ein wirkliches Verständnis müssen Sie mit einem Modell arbeiten.

Im nächsten Kapitel untersuchen wir große Sprachmodelle oder LLMs (*Large Language Models*). Wir werden erörtern, warum sie als groß bezeichnet werden und was an ihrer Größe so besonders ist. Dann beschäftigen wir uns damit, wie

man ein vorhandenes LLM für praktische Aufgaben wie die Beantwortung von Fragen und Klassifizierung von Dokumenten optimiert und wie man LLMs für eine Reihe von praktischen Problemen einsetzt.

## Symbole

@ (Matrixmultiplikation) [135](#)

## A

Ableitungen

erste [29](#)

Gradientenabstieg [99](#)

partielle [29](#)

accuracy [152](#)

Achsenabschnitt [27](#)

additives Glätten [79](#)

Aktivierungsfunktionen [38](#)

Sigmoid [58](#)

Softmax [58](#)

tanh [101](#)

Algorithmen

Gradientenabstieg [47](#)

Random Forest [25](#)

Skip-Gramm- [66](#)

Solver [152](#)

word2vec [66](#)

All-gather [145](#)

Anwesenheitsstrafe [165](#)

APIs

Modul- [63](#)

sequenzielle [63](#)

Assertionen

negative Lookahead- [74](#)

negative Lookbehind- [74](#)

assistant [158](#)  
Attention  
    Flash [144](#)  
    Grouped-Query [144](#)  
    Multi-Head- [125](#)  
    Scores [117](#)  
AttentionHead (Klasse) [134](#)  
Attribute  
    .grad [52](#)  
    requires\_grad [168](#)  
Ausgabeschichten [40](#)  
Autograd [49](#)  
automatisches Differenzieren [49](#)  
AutoModelForCausalLM [170](#)  
AutoModelForSequenceClassification [170](#)  
autoregressive Sprachmodelle [77](#), [113](#), [153](#)

## **B**

Backoff [78](#)  
Backpropagation [52](#)  
Backward-Pass [52](#)  
Bag-of-Words [55](#), [150](#)  
Bahdanau, Dzmitry [119](#)  
Baseline [150](#)  
Batch-Matrixmultiplikation [135](#)  
BCELoss [53](#)  
Bearbeitungsdistanz [180](#)  
bedingte Wahrscheinlichkeiten [76](#)  
Beispiele [26](#)  
Belohnungsmodell [184](#)  
Bengio, Yoshua [119](#)  
Berechnungsgraphen [39](#)  
Bias [27](#)  
Bibliotheken  
    BLAS [35](#)  
    cuBLAS [35](#)  
    lineare Algebra [35](#)  
    Milvus [36](#)  
    PEFT [168](#)  
    Qdrant [36](#)

scikit-learn [150](#)  
Weaviate [36](#)  
binäre Klassifizierung [44](#), [55](#)  
binäre Kreuzentropie [45](#), [52](#), [58](#)  
BLAS [35](#)  
Bootstrap-Resampling [94](#)  
BoW (Bag-of-Words) [55](#), [150](#)  
Bradley-Terry-Modell [94](#)  
Broadcasting [135](#)  
Brown Corpus [82](#)  
build\_prompt [160](#)  
Byte-Paar-Codierung [71](#), [106](#)

## C

\_\_call\_\_ [161](#)  
Chain of Thought (CoT) [185](#)  
Chat-LMs [91](#), [141](#), [171](#)  
ChatML (Chat Markup Language) [157](#)  
Chat-Sprachmodelle [58](#), [68](#), [77](#), [171](#)  
choice [163](#)  
CLIP (Contrastive Language-Image Pretraining) [185](#)  
Codegenerierung [174](#)  
Codomäne [26](#)  
Colab [20](#)  
compute\_perplexity [85](#)  
CountLanguageModel [79](#)  
CountVectorizer [151](#)  
Cross-Attention [185](#)  
CrossEntropyLoss [53](#)  
cuBLAS [35](#)

## D

DARE [183](#)  
DataLoader [108](#)  
Dataset [108](#)  
Dateien  
    JSONL [108](#)  
    lesen [108](#)  
Datenparallelität [147](#)  
Datensätze [26](#)

- Dolma [146](#)
- Qwen 2.5 [146](#)
- DecoderBlock (Klasse) [137](#)
- DecoderLanguageModel (Klasse) [138](#)
- Deep Learning [128](#)
- Deep Neural Networks [128](#)
- dichte Schichten [40](#)
- dichte Vektoren [66](#)
- Differenzialrechnung
  - Faktorregel [31](#), [47](#)
  - Kettenregel [31](#)
  - Summenregel [31](#), [47](#)
- Differenzieren, automatisches [49](#)
- Dimensionalität [34](#)
  - Verringerung [69](#), [70](#)
- Dimensionen [34](#)
- diskrete Wahrscheinlichkeitsverteilung [59](#), [76](#)
- doc\_to\_bow [62](#)
- Dokumentation, synchronisieren [175](#)
- Dokument-Term-Matrix [57](#)
- Dolma [146](#)
- domänenspezifisches Vortraining [178](#)
- download\_and\_prepare\_data [81](#)
- Dropout [186](#)
- DTM (Dokument-Term-Matrix) [57](#)
- dünn besetzt [57](#)
  - MoE-Schichten [183](#)
  - One-Hot-Vektoren [66](#)

## **E**

- Early Stopping [186](#)
- Eingaben [28](#)
- Eingabeschichten [40](#)
- Eingabesequenzen [76](#)
- eingebettete Vektoren [36](#)
- Einheiten [39](#)
- Einheitsvektoren [36](#)
- 1-aus-n-Vektor [60](#)
- elementweises Produkt [35](#), [132](#)
- Elman-RNN [98](#)

ElmanRNN, Klasse [101](#), [102](#)  
ElmanRNNUnit [101](#)  
Elo-Ratings [92](#)  
Embeddings  
    Dimensionalität [103](#)  
    Schichten [103](#)  
    Wort- [65](#)  
endliche Menge [59](#)  
EndTokenStoppingCriteria [160](#)  
Entscheidungsbäume [25](#)  
Epochen [107](#)  
Erklärbarkeit [181](#)  
erste Ableitung [29](#)  
eulersche Zahl [38](#), [47](#), [58](#), [83](#)  
Experten [183](#)

## **F**

Faktorregel [31](#), [47](#)  
FastText [69](#)  
Featurevektor [34](#), [55](#)  
Feedforward [40](#), [63](#)  
Fehler  
    mittlerer quadratischer [29](#)  
    quadratischer [28](#)  
Feintuning [149](#)  
    überwachtes [86](#), [147](#)  
    vortrainiertes Modell [149](#)  
Few-Shot-Prompting [171](#)  
fit [152](#)  
fit\_transform [151](#)  
FlashAttention [144](#)  
Flexionsformen [57](#)  
FLOPs (Floating-Point Operations) [146](#)  
forward [63](#), [101](#), [134](#), [136](#), [137](#), [138](#)  
Forward-Pass [52](#)  
Frankenmerges [184](#)  
Funktionen [26](#)  
    Aktivierung [38](#)  
    doc\_to\_bow [62](#)  
    get\_probability [85](#)

initialize\_weights [106](#)  
Komposition [30](#)  
lineare [27](#)  
set\_seed [105](#)  
Softmax [162](#)  
Verlust- [29](#)

## **G**

Gate-Netz [183](#)  
Gedankenbaum [185](#)  
Gedankenkette, explizite [185](#)  
Genauigkeit [88](#)  
Generalisierung [151](#), [186](#)  
get\_probability [85](#), [86](#)  
get\_vocabulary [62](#)  
\_\_getitem\_\_ [108](#)  
Gewichte [27](#)  
Gewichtsterm [27](#)  
GloVe [69](#)  
google/gemma-2-2b [148](#)  
google/gemma-2-2b-it [148](#)  
Google Colab [20](#)  
GPT-2  
    Feintuning [150](#)  
    Komplexität [86](#)  
    Perplexität [112](#)  
GPU-Stunden [146](#)  
.grad [52](#)  
Gradienten [47](#)  
    Verfolgung [65](#)  
Gradientenabstieg [47](#)  
    Konvergenz [100](#)  
    Mini-Batch- [99](#)  
Greedy Decoding [161](#)  
Größe  
    Batches [99](#)  
    N-Gramm [82](#)  
    Vektoren [36](#)  
    Vokabular [74](#), [103](#), [164](#)  
Groß-Sigma-Notation [34](#)

Ground Truth [87](#)  
Grouped-Query Attention [144](#)

## H

Halluzinationen [176](#)  
    verhindern [177](#)  
Häufigkeitsstrafe [165](#)  
Hauptkomponenten  
    erste [70](#)  
    zweite [70](#)  
Hauptkomponentenanalyse [70](#)  
Hugging Face Hub [148](#)  
Hyperparameter [48](#)  
    Epochen [107](#)

## I

Inferenz  
    Anweisung [161](#)  
    autoregressive [133](#)  
    Gradienten [65](#)  
    Kontextlänge [120](#)  
    Parameter [183](#)  
Informativität [90](#)  
initialize\_weights [106](#)  
Installation  
    LoRA [168](#)  
    scikit-learn [150](#)  
    transformers [105](#)  
is\_available() [106](#)  
Iterationen [48](#)

## J

Jailbreak-Angriffe [185](#)  
JSON [150](#)  
JSONL [108](#), [150](#)

## K

kausale Maske [117](#), [138](#)  
kausale Sprachmodelle [77](#)

Kernel-Methoden [26](#)  
Kettenregel [31](#), [46](#)  
Klassen  
    AttentionHead [134](#)  
    CountLanguageModel [79](#)  
    CountVectorizer [151](#)  
    DataLoader [108](#)  
    Dataset [108](#)  
    DecoderBlock [137](#)  
    DecoderLanguageModel [138](#)  
    ElmanRNN [101](#)  
    EndTokenStoppingCriteria [160](#)  
    MultiHeadAttention [135](#)  
    RecurrentLanguageModel [103](#)  
Klassifizierer [50](#)  
Klassifizierung  
    binäre [44](#), [55](#)  
    Mehrklassen- [55](#)  
Klassifizierungskopf [169](#)  
Koeffizient [27](#)  
Komponenten [34](#)  
Komposition [46](#)  
    Funktionen [30](#)  
    Modelle [39](#)  
Konfidenzintervalle [94](#)  
konstanter Term [27](#)  
konstitutionelle KI [184](#)  
Kontext [76](#)  
kontextbezogenes Lernen [171](#)  
Kontextfenster [110](#)  
Kontextparallelität [145](#), [147](#)  
Konvergenz [48](#), [100](#)  
Kookkurrenz [69](#)  
Korpus [56](#)  
Kosinus-Ähnlichkeit [36](#), [66](#), [180](#)  
Kreuzentropie [69](#), [107](#), [170](#)  
künstliche Intelligenz [23](#)  
künstliche Neuronen [39](#)

## **L**

L1 [186](#)  
L2 [186](#)  
Labeling [57](#)  
Länge  
    Eingabesequenz [116](#)  
    N-Gramme [87](#)  
    Sequenzen im Stapel [156](#)  
    Vektoren [36](#)  
längste gemeinsame Teilsequenz [88](#)  
Laplace-Glättung [79](#), [85](#)  
Lastenausgleich [183](#)  
Layer [39](#)  
lbfgs [152](#)  
\_\_len\_\_ [108](#)  
Lernen  
    unüberwachtes [28](#)  
    verstärkendes [28](#)  
Lernrate [48](#)  
Likelihood, negative Log- [83](#)  
LIMA [158](#)  
lineare Algebra [35](#)  
lineare Funktionen [27](#)  
lineare Regression [38](#), [45](#)  
Llama 3.1 [147](#)  
Loader [107](#)  
Logarithmus, natürlicher [45](#)  
logistische Regression [45](#)  
logistischer Verlust [45](#)  
Logits [58](#)  
Longest Common Subsequence (LCS) [88](#)  
Long Short-Term Memory (LSTM) [112](#)  
LoRA (Low-Rank Adaptation) [166](#)  
LoRA-Adapter [167](#)

## **M**

Machine Learning [26](#)  
    Datensätze [26](#)  
maskierte Scores [117](#)  
maskierte Sprachmodelle [77](#)  
Matrixmultiplikation [101](#)

- Batch- [135](#)
- Matrix-Vektor-Multiplikation [43](#)
- Matrizen [42](#)
  - dünn besetzt [57](#)
  - elementweises Produkt [132](#)
  - Matrix-Vektor-Multiplikation [43](#)
  - Produkt [42](#)
  - Rotations- [121](#)
  - Summe [42](#)
  - transponieren [43](#)
- Maximum Likelihood Estimate (MLE) [78](#)
- Maximum-Likelihood-Schätzung [78](#)
- Mehrklassenklassifizierung [55](#)
- mehrschichtiges Perzeptron [40](#)
- Mengen, endliche [59](#)
- mergekit [184](#)
- Methoden
  - `__call__` [161](#)
  - `__getitem__` [108](#)
  - `__len__` [108](#)
  - `build_prompt` [160](#)
  - `choice` [163](#)
  - `compute_perplexity` [85](#)
  - `download_and_prepare_data` [81](#)
  - `fit` [152](#)
  - `fit_transform` [151](#)
  - `forward` [63](#), [101](#), [134](#), [136](#), [137](#), [138](#)
  - `get_probability` [86](#)
  - `get_vocabulary` [62](#)
  - `is_available()` [106](#)
  - `predict_next_token` [80](#)
  - `tokenize` [62](#)
  - `train` [80](#)
  - `transform` [151](#)
- Metriken
  - `accuracy` [152](#)
  - Perplexität [83](#)
  - ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [87](#)
- Milvus [36](#)
- Mini-Batch-Gradientenabstieg [99](#)
- miniSTM [112](#)

mittlerer quadratischer Fehler [29](#)  
Mixtral 8x7B [183](#)  
Mixture of Experts (MoE) [183](#)  
MLP  
    Klasse [136](#)  
    Mixture of Experts (MoE) [183](#)  
    Multilayer Perceptron [114](#)  
Modelle [26](#)  
    Berechnungsgraphen [39](#)  
    Generalisierung [151](#)  
    google/gemma-2-2b [148](#)  
    google/gemma-2-2b-it [148](#)  
    Komposition [39](#)  
    LIMA [158](#)  
    logistische Regression [45](#)  
    Mixtral 8x7B [183](#)  
    Open-Weight- [143](#)  
    vortrainierte [86](#)  
Modellkomprimierung [184](#)  
Modellparallelität [149](#)  
Model Merging [183](#)  
Model Soups [183](#)  
Modul-API [50](#), [63](#)  
MPS (Apple Metal) [106](#)  
Multi-Head-Attention [125](#), [126](#)  
MultiHeadAttention (Klasse) [135](#)  
Multilayer Perceptron (MLP) [114](#), [119](#)

## **N**

Nadel-im-Heuhafen-Test [145](#)  
natürlicher Logarithmus [45](#)  
negative Log-Likelihood [83](#)  
negative Lookahead-Assertionen [74](#)  
negative Lookbehind-Assertionen [74](#)  
neuronalen Netze [37](#)  
    Feedforward [40](#), [63](#)  
N-Gramme [65](#)  
    Länge [87](#)  
NLP (Natural Language Processing) [113](#)  
Norm [36](#)

Nucleus-Sampling [164](#)  
Nullvektoren [36](#), [66](#), [98](#)  
Nur-Decoder-Transformer [113](#)

## O

One-Hot-Vektor [60](#)  
Open-Weight-Modelle [143](#)  
Overfitting [25](#), [81](#), [186](#)

## P

paarweise Vergleiche [92](#)  
Padding [98](#)  
Parallelität, Modell- [149](#)  
Parameter [27](#)  
    Anzahl [142](#)  
    Inferenz [183](#)  
    Temperatur [162](#)  
Parameter-Efficient Finetuning (PEFT) [168](#)  
partielle Ableitungen [29](#)  
Partitionen  
    Test- [81](#)  
    Trainings- [81](#)  
Passthrough [184](#)  
PEFT (Parameter-Efficient Finetuning) [168](#)  
Perplexität [83](#)  
    berechnen [85](#)  
Perzeptron [25](#)  
Phi 3.5 mini [106](#)  
Pipeline-Parallelität [147](#)  
Post-Training-Quantisierung [184](#)  
Precision [88](#)  
predict\_next\_token [80](#)  
Principal Component Analysis (PCA) [70](#)  
Problem der verschwindenden Gradienten [128](#)  
Produkt, Matrizen [42](#)  
Prompts [76](#)  
    Engineering [171](#)  
    Injection [185](#)  
Public Domain [179](#)  
Punktprodukt [34](#), [118](#)

## PyTorch

- BCELoss [53](#)
- Modul-API [50](#)
- sequenzielle API [50](#)
- Tensoren [51](#)
- torch.nn [50](#)
- torch.optim [50](#)

## Q

- Qdrant [36](#)
- QLoRA (Quantized Low-Rank Adaptation) [184](#)
- quadratischer Fehler [28](#)
- Quantisierung [51](#)
- Qwen 2.5 [146](#)

## R

- RAG (Retrieval-Augmented Generation) [177](#)
- Random Forest [25](#)
- Rang [166](#)
- Ranking [92](#)
- ReAct [185](#)
- Recall [87](#)
- RecurrentLanguageModel [103](#)
- Regeln, Funktionen [26](#)
- Regression
  - lineare [45](#)
  - logistische [45](#)
- reguläre Ausdrücke [62](#)
- Regularisierung [186](#)
  - Dropout [186](#)
- Reinforcement Learning [28](#)
- Reinforcement Learning from Human Feedback (RLHF) [184](#)
- Reproduzierbarkeit [62](#)
- requires\_grad [168](#)
- Residualverbindungen [128](#)
- Retrieval-Augmented Generation [177](#)
- RMSNorm [131](#)
- RNN (rekurrentes neuronales Netz) [82](#), [97](#)
  - Elmann-RNN [98](#)
- Rollen

- assistant [158](#)
- system [158](#)
- user [158](#)
- Rotary Position Embedding (RoPE) [120](#)
- Rotationsfrequenz [123](#)
- Rotationsmatrizen [121](#)
- Rotierendes Positions-Embedding (RoPE) [120](#)
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [87](#)
- ROUGE-1 [87](#)
- ROUGE-L [88](#)
- ROUGE-N [87](#), [88](#)
- Routernetz [183](#)

## **S**

- Schichten [39](#)
  - dichte [40](#)
  - vollständig verbundene [40](#)
- Schlüssel-Wert-Caching [133](#)
- Schritte [48](#)
- scikit-learn [150](#)
- Selbstkonsistenz [185](#)
- Self-Attention
  - Funktionsweise [116](#)
  - Parameter [166](#)
  - Punktprodukt [118](#)
  - Transformer [115](#)
- semantische Ähnlichkeit [69](#)
- sequenzielle API [50](#), [63](#)
- set\_seed [105](#)
- Sigmoid [58](#)
- Skalar [55](#)
- Skalarprodukt [34](#)
- skalierte Scores [117](#)
- Skip-Gramme [66](#)
- Skip-Verbindungen [128](#)
- SLERP (Spherical Interpolation That Maintains Parameter Norms) [183](#)
- Softmax [58](#), [110](#), [162](#)
- Solver [152](#)
  - lbfgs [152](#)
- Spaltenvektor [34](#)

Sprachmodelle [76](#)  
    autoregressive [77](#), [113](#), [118](#), [153](#)  
    bewerten [83](#)  
    Chat- [77](#)  
    kausale [77](#)  
    Kontextfenster [110](#)  
    maskierte [77](#)  
    Perplexität [83](#)  
    zählbasierte [77](#)  
StackOverflow [174](#)  
Steigung [27](#)  
Strukturen [27](#)  
strukturiertes Pruning [184](#)  
Summe, Matrizen [42](#)  
Summenregel [31](#), [47](#)  
Supervised Learning [33](#), [147](#)  
Support Vector Machines [26](#)  
SVMs, Kernel-Methoden [26](#)  
system [158](#)

## **T**

tanh [101](#)  
Task-Vektor-Algorithmen [183](#)  
Teilwörter [56](#), [71](#)  
Temperatur [162](#)  
Tendenz zur Mitte [90](#)  
Tensoren [51](#)  
    Broadcasting [135](#)  
Tensor-Parallelität [147](#)  
Testdatensatz [186](#)  
Testpartitionen [81](#)  
tiefe neuronale Netze (Deep Neural Networks) [128](#)  
TIES-Merging [183](#)  
Tokenisierung [56](#)  
tokenize [62](#)  
Tokenizer, Phi 3.5 mini [106](#)  
Tokens [56](#)  
    längste gemeinsame Teilsequenz [88](#)  
Top-k-Routing [183](#)  
Top-k-Sampling [163](#)

Top-p-Sampling [164](#)  
torch.nn [50](#)  
torch.optim [50](#)  
train [80](#)  
Training  
    autoregressive Sprachmodelle [113](#)  
    Decoder-Sprachmodell [139](#)  
    Deep Learning [128](#)  
    Partitionen [81](#)  
    Schleife [106](#)  
    Tokenizer [106](#)  
    Verlust [111](#)  
    Vor- [141](#)  
Trainingsmenge [33](#)  
Trainingsverlust [33](#)  
transform [151](#)  
Transformer [113](#)  
    Architektur [113](#)  
    Multi-Head-Attention [125](#)  
    Nur-Decoder- [113](#)  
    Self-Attention [115](#)  
transformers, Installation [105](#)  
transponieren [43](#)  
Tree of Thought (ToT) [185](#)

## U

Überanpassung [25](#), [81](#), [146](#), [186](#)  
überparametrisiert [184](#)  
überwachtes Feintuning [86](#), [147](#)  
unstrukturiertes Pruning [184](#)  
Unsupervised Learning [28](#)  
unüberwachtes Lernen [28](#)  
Urheberrecht [179](#)  
user [158](#)

## V

Validierungsdatensatz [186](#)  
Vektoren [34](#)  
    1-aus-n- [60](#)  
    dichte [66](#)

- Dimensionalität [34](#)
- dünn besetzte One-Hot- [66](#)
- Einheits- [36](#)
- elementweises Produkt [35](#)
- Embedded [36](#)
- Feature- [34](#)
- Größe [36](#)
- Länge [36](#)
- Norm [36](#)
- Null- [36](#), [66](#)
- One-Hot- [60](#)
- Spalten- [34](#)
- Verallgemeinerung [81](#)
- Verbindungen
  - Residual- [128](#)
  - Skip- [128](#)
- verborgene Zustände [98](#)
- Vergleiche
  - Elo-Ratings [92](#)
  - paarweise [92](#)
- Verkettungs- und Projektionsschicht [127](#)
- Verlustfunktionen [29](#), [45](#)
  - binäre Kreuzentropie [45](#), [52](#), [58](#)
  - CrossEntropyLoss [53](#)
  - Kreuzentropie [69](#), [107](#)
- Versionskontrollsysteme [175](#)
- verstärkendes Lernen [28](#)
- Vertrauensintervalle [94](#)
- 4D-Parallelität [147](#)
- Vision-Encoder [185](#)
- Vision Language Models (VLMs) [185](#)
- Vokabular aktualisieren [72](#)
  - anfängliches [72](#)
  - erstellen [62](#)
  - Größe [103](#), [164](#)
  - Sampling [163](#)
- vollständig verbundene Schichten [40](#)
- Vorhersagewert [45](#)
- vortrainierte Modelle [86](#)
- Vortraining [141](#)

## **W**

Wahrscheinlichkeiten bedingte [76](#)

    Softmax [110](#)

    Verteilungen, diskrete [59](#), [76](#)

Weaviate [36](#)

Wissensdestillation [184](#)

word2vec [66](#), [174](#)

WordNet [70](#)

Wort-Embeddings [65](#), [66](#)

    FastText [69](#)

    GloVe [69](#)

    word2vec [69](#)

## **X**

Xavier-Initialisierung [106](#)

xLSTM [112](#)

## **Z**

zipfsches Gesetz [57](#)