Einführung in Transformer-Architekturen von Neuronalen Netzen

Ausarbeitung Integrationsseminar

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Wirtschaftsinformatik Software Engineering

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Alexander Meinecke

Abgabedatum:	27. Januar 2025
Bearbeitungszeitraum:	Dezember 2024 - 27. Januar 2025
Matrikelnummer, Kurs:	1522347, WWI22SEB
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Gutachter:	Alexander Lütke

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Ausarbeitung Integrationsseminar mit dem Thema:

Einführung in Transformer-Architekturen von Neuronalen Netzen

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, den 3. Januar 2025

Meinecke, Alexander

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

BEHG	Brennstoffemissionshandelsgesetz
CBAM	Carbon Border Adjustment Mechanism (dt. Grenzausgleichsmechanismus)
DEHSt	Deutsche Emissionshandelsstelle
EPA	United States Environmental Protection Agency (dt. Umweltschutzbehörde der Vereinigten Staaten)
ETS	Emission Trading System (dt. Emissionshandelssystem)
ICAP	International Carbon Action Partnership (dt. Internationale Partnerschaft für Emissionshandel)
MSR	Market Stability Reserve (dt. Marktstabilitätsreserve)
nEHS	Nationales Emissionshandelssystem
TNAC	Total Number of Allowances in Circulation (dt. Anzahl der Zertifikate im Umlauf)

1 Einleitung

Transformer-Architekturen haben die Landschaft der neuronalen Netze revolutioniert und neue Maßstäbe in Bereichen wie maschineller Übersetzung, Textgenerierung und Sprachverarbeitung gesetzt.

1.1 Motivation und Ziel der Arbeit

Der Grund, warum Transformer im Bereich der Textanalyse und -verarbeitung so gut sind, ist ihre einzigartige Architektur. Sie sind dadurch viel effizienter, schneller und erzielen auch bessere Ergebnisse. Die klassische Transformer-Architektur wurde 2017 in einem Paper mit dem Titel „Attention is All You Need“ vorgestellt. Diese Grundlagen zu verstehen, ermöglicht es auch, bekannte Modelle wie BERT, GPT oder T5 zu verstehen. Ziel dieser Arbeit ist es, dem Leser ohne viel Vorwissen die klassischen Konzepte der Transformer-Architektur zu verdeutlichen.

1.2 Aufbau der Arbeit

Zunächst werden im folgenden Kapitel dem Leser Grundlagen von neuronalen Netzen vermittelt, die helfen, die Notwendigkeit von Transformern in der Geschichte von neuronalen Netzen zu verstehen. Anschließend wird sich ein Kapitel mit dem Kernkonzept von Transformern beschäftigen, das dem Model hilft, den Kontext von Texten schnell und effizient zu analysieren. Dies wird zusätzlich mit einem praktischen Beispiel erläutert. Darauf folgt die Erklärung der Transformer-Architektur und ihrer Komponenten im Detail. Am Ende folgt noch ein Fazit, das noch einmal die wichtigsten Aspekte zusammenfasst.

2 Grundlagen

Um zu verstehen, wie Transformer funktionieren, muss zunächst ein kurzer Überblick über neuronale Netze gegeben werden.

2.1 Neuronale Netze

Neuronale Netze sind ein Teilgebiet des maschinellen Lernens. Sie sind trainierbar und können die gesammelten Erfahrungen auf neue Daten anwenden. Neuronale Netze simulieren dabei den Aufbau des menschlichen Gehirns. Sie bestehen aus mehreren Schichten von Neuronen.

Es gibt dabei eine Eingabe- und Ausgabeschicht sowie mehrere versteckte Schichten, in denen die eigentliche Verarbeitung stattfindet. Jedes Neuron verarbeitet seine eigenen Daten, gewichtet sie und addiert sie mit den Ergebnissen der vorherigen Schicht. Dieses Ergebnis wird dann an die Aktivierungsfunktion weitergegeben, um die nächsten verbundenen Neuronen anzusteuern. Diese Funktion bestimmt, ob das Neuron „feuert“ und das Signal weitergeleitet wird.

Über die Jahre haben sich neuronale Netze besonders für Anwendungen wie Bildanalysen als äußerst effektiv erwiesen. Doch ein Problem war lange die Textanalyse und -verarbeitung. Hier besteht die Herausforderung darin, dass Informationen nicht wie bei einem Bild simultan erfasst werden können. Um den Sinn eines Satzes zu verstehen, muss der Inhalt schrittweise verarbeitet werden.

2.2 Neuronale Netze für Textverarbeitung

Für Textanalysen, Übersetzungen oder Zusammenfassungen wurden lange Zeit Recurrent Neural Networks (RNNs) verwendet. RNNs analysieren einen Text sequentiell und verknüpfen jedes Wort mit dem Kontext aus dem vorhergehenden Wort.

Doch RNNs haben Schwächen. Sie haben Schwierigkeiten, den Kontext von größeren Texten vollständig zu erfassen. Außerdem ist die sequentielle Verarbeitung ineffizient und schlecht auf moderne Hardware optimiert, die auf Parallelverarbeitung ausgelegt ist.

Um diese Probleme zu lösen, wurde eine neue Art von neuronalen Netzen entwickelt: der Transformer.

3 Die Funktionsweise von Self-Attention

Transformer arbeiten mit dem zentralen Baustein Self-Attention. Self-Attention ist ein Algorithmus der letztendlich Zusammenhänge zwischen Wörtern z.B. in einem Text aufzeigt. Wörter werden in Form von Tokens verarbeitet, die ganze Wörter oder Wortteile sein können. Jeder Token ist einzigartig und wird zunächst nur durch eine natürliche Zahl repräsentiert.

3.1 Generierung von Embeddings

Grundlage jedes Attention-Zyklus sind Eingabe-Tokens. Um die mathematische Vorgehensweise besser zu veranschaulichen, wird im Folgenden der Satz „Ich sitze auf der Bank“ als Beispiel verarbeitet. Jedes dieser Wörter ist ein eigener Token, der vor der Eingabe in den Attention-Zyklus vom Transformer übersetzt wurde: [„Ich“, „sitze“, . . . , „Bank“]. Diese Tokens könnten für das Modell wie folgt aussehen: [„243“, „645“, . . . , „316“]. Die Herausforderung für den Transformer besteht darin, aus dem Kontext der anderen Tokens zu erkennen, ob „Bank“ eine Sitzbank oder das Finanzinstitut Bank bedeutet.

Jeder Token bildet ein Schlüssel-Werte-Paar. Der korrespondierende Wert hinter einem Token ist ein Vektor, der die Bedeutung eines Tokens hinsichtlich mehrerer Dimensionen beschreibt. Diese Vektoren sind aus Trainingsdaten des Transformer-Modells entstanden.

Im ersten Schritt des Attention-Zyklus wird jeder Token in den dazugehörigen Vektor übersetzt. Diese Vektoren werden in der Matrix $\mathbf{X}$ gespeichert, wobei jede Zeile einen Token repräsentiert. Dieser Prozess wird als **Embedding** bezeichnet. Jeder dieser Vektoren hat gemäß der Literatur mindestens 512 Dimensionen. Es wird von einem $d_{\text{model}} = 512$ gesprochen. Es gilt: Je größer das d_{model}, desto präziser kann der Transformer die Zusammenhänge zwischen Tokens erkennen.

Wenn sich Tokens im Vektorraum nahe liegen, haben sie eher Gemeinsamkeiten im Vergleich zu Tokens, die weit auseinander liegen. Angenommen, es gäbe nur ein $d_{\text{model}} = 2$ für jeden Token, könnten diese zwei Dimensionen als Koordinaten genutzt werden, um Zusammenhänge visuell als Cluster in einem Koordinatensystem darzustellen. Hier wären beispielsweise die Tokens „Hund“ und „Katze“ nah beieinander.

Für das oben genannte Beispiel „Ich sitze auf der Bank“ nehmen wir der Übersichtlichkeit halber ein $d_{\text{model}} = 4$. So ergibt sich eine Embedding-Matrix $\mathbf{X}$ mit 5 Zeilen für 5 Tokens und 4 Spalten für jeweils 4 Dimensionen:

$$\mathbf{X} = \begin{bmatrix} 0.4 & 0.8 & 1.5 & 1.6 \\ 3.2 & 0.4 & 0.7 & 0.2 \\ 0.6 & 0.9 & 1.2 & 0.5 \\ 2.1 & 0.5 & 2.0 & 0.2 \\ 0.7 & 2.4 & 0.1 & 0.9 \end{bmatrix}$$

3.2 Lineare Transformation in Query-, Key- und Value-Matrizen

Die Embedding-Matrix $\mathbf{X}$ wird durch drei Gewichtungsmatrizen $\mathbf{W_Q}$, $\mathbf{W_K}$ und $\mathbf{W_V}$, die aus dem Training des Transformer-Modells stammen, in drei neue Matrizen transformiert:

$$\mathbf{Q} = \mathbf{X} \cdot \mathbf{W_Q}$$

$$\mathbf{K} = \mathbf{X} \cdot \mathbf{W_K}$$

$$\mathbf{V} = \mathbf{X} \cdot \mathbf{W_V}$$

Die drei Matrizen haben im Attention-Zyklus umgangssprachlich formuliert folgende Funktionen:

- **Query-Matrix (Q)**: Was fragt ein Token?
- **Key-Matrix (K)**: Welche Tokens im Kontext antworten am besten auf die Frage?
- **Value-Matrix (V)**: Erlernten Informationen über ein Token.

Im Beispiel könnten die jeweiligen Zeilen der Matrizen $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ für das Token „Bank“ folgendermaßen aussehen:

$$Q_{\text{Bank}} = [1.0, 0.7, 0.9, 1.1], \quad K_{\text{Bank}} = [0.8, 0.6, 1.0, 0.9], \quad V_{\text{Bank}} = [0.9, 0.5, 0.7, 1.0]$$

3.3 Berechnung und Einbeziehung von Attention-Scores

Um die Relevanz zwischen Q und K zu messen, wird jeweils das Skalarprodukt zwischen jedem Tokenvektor von Q_{T} und K_{T} gebildet. Also im Beispiel wird unter anderem der Tokenvektor Q_{Bank} mit jedem Tokenvektor K_{T} multipliziert.

$$
\begin{aligned}
\text{Score}_{\text{Bank,Ich}} &= [1.0, 0.7, 0.9, 1.1] \cdot [0.4, 0.5, 0.1, 0.3] \\
&= 0.4 + 0.35 + 0.09 + 0.33 && = 1.17 \\
\text{Score}_{\text{Bank,Sitze}} &= [1.0, 0.7, 0.9, 1.1] \cdot [0.9, 0.8, 0.75, 0.8] \\
&= 0.9 + 0.56 + 0.675 + 0.88 && = 3.015 \\
&\vdots \\
\text{Scores}_{\text{Bank}} &= [1.17, 3.015, 2.92, 1.12, 2.98]
\end{aligned}
$$

Die berechneten Attentionscores müssen noch zwei Verfahren unterlaufen. Einmal ist das die Fokussierung und Normalisierung der Attentionscores mit der **Softmax-Funktion**.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

x ist der jeweilige aktuell zu betrachtende Attention-Score-Vektor. i ist der aktuell zu betrachtende Werteindex in diesem Vektor. j ist die Gesamtanzahl an Werten in x.

Bei der **Fokussierung** werden höhere Attention-Score-Werte zwischen **Q** und **K** exponentiell bevorzugt. Analog dazu werden niedrigere Attention-Score-Werte exponentiell nach unten bewertet. Bei der zweiten Aufgabe der Softmax-Funktion, der **Normalisierung**, werden die Attention-Score-Werte pro Score-Vektor in Wahrscheinlichkeiten zwischen 0 und 1 transformiert, wobei die Summe jedes Attention-Score-Vektors immer 1 ist.

Damit die Softmax-Funktion aber optimal funktionieren kann, müssen die Werte in den Attention-Score-Vektoren erst einmal dimensioniert werden. Bei geläufigen Transformermodellen wird wie oben beschrieben, ein d_{model} von mindestens 512 verwendet. Durch diese großen Dimensionen entehen bei der Berechnung von den Attention-Scores durch die Aufsummierung bei der Bildung des Skalarprodukte sehr große Werte. Diese großen Werte sorgen dafür, dass die Softmax-Funktion viele Q-K-Beziehungen sehr hoch bewertet und so der Transformer nicht sich auf die tatsächlich vielversprechenden Verbindungen konzentrieren kann und so die Weiterverarbeitung ungenau wird. Diese Werte fallen bei dem oben gerechneten Beispiel nicht auf, da hier nur mit einem d_{model} von vier gerechnet wird.

Um hohe Attention-Score-Werte zu normalisieren, werden die Attention-Score-Vektor-Werte durch $\sqrt{d_{\text{model}}}$ geteilt und so für die Softmax-Funktion in einen stabilen Bereich gebracht.
So kann ein Transformer auch kleine Relevanzunterschiede in der Token-Beziehung berücksichtigen. Hier beispielsweise für das Attention-Score-Array von „Bank“:

$$\frac{\text{Scores}_{\text{Bank}}}{\sqrt{4}} = [0.585, 1.5075, 1.46, 0.56, 1.49]$$

$$\text{Softmax}\left(\frac{\text{Scores}_{\text{Bank}}}{\sqrt{4}}\right) = [0.107, 0.269, 0.256, 0.104, 0.264]$$

Damit diese nun umgewandelten Attention-Score-Wahrscheinlichkeiten in den weiteren Verarbeitungsschritten berücksichtigt werden können, werden sie mit der V-Matrix multipliziert. Das zeigt dem Modell, zu wie viel Prozent der erlernten Informationen zu einem Token im nächsten Schritt einfließen. Insgesamt sieht das Verfahren folgendermaßen aus:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_{\text{model}}}}\right)V$$

4 Transformer-Architektur im Detail

Im ursprünglichen Werk *Attention Is All You Need* , in dem das grundlegende Transformer-Architektur cit vorgestellt wurde, besteht ein Transformer aus zwei Hauptmodulen: dem Encoder und dem Decoder. Um die Architektur eines Transformers besser zu verstehen, müssen zunächst die Komponenten betrachtet werden, die in Encoder und Decoder eingesetzt werden.

4.1 Multi-Head-Attention

Wie schon im vorherigen Kapitel beschrieben, ist Self-Attention ein zentraler Baustein in der Transformer-Architektur. Um Transformer noch effizienter zu gestalten, wird Self-Attention in Form der **Multi-Head-Attention** angewendet. Dabei beschreibt jeder einzelne Kopf einen eigenen Attention-Zyklus, wie oben beschrieben.

Jeder Kopf verwendet unterschiedliche Gewichtungsmatrizen, um den Fokus bei der Analyse der Zusammenhänge zwischen Tokens auf verschiedene Schwerpunkte zu setzen. So kann beispielsweise ein Kopf die Tokens hinsichtlich syntaktischer Beziehungen analysieren, während ein anderer die semantischen Beziehungen betrachtet. Für jeden Attention-Kopf h werden dabei unterschiedliche Gewichtungsmatrizen $\mathbf{W}_{Qh}$, $\mathbf{W}_{Kh}$ und $\mathbf{W}_{Vh}$ verwendet.

Viele Transformermodelle verwenden acht Attention-Köpfe, die parallel ausgeführt werden. Nach der parallelen Verarbeitung werden die jeweiligen Ergebnisse der Köpfe zusammengefügt (auch als Konkatenieren bezeichnet) und mit Hilfe einer weiteren Gewichtungsmatrix $\mathbf{W}_O$ zusammengefasst. $\mathbf{W}_O$ ist die Matrix, welche die Ergebnisse der Attention-Köpfe unterschiedlich gewichtet. Umgangssprachlich formuliert bildet sie eine einheitliche Schlussfolgerung für den Transformer aus dem Multi-Head-Attention-Zyklus.

4.2 Positional Encodings

Ein Problem bei der einfachen Verwendung von Self-Attention ist, dass die tatsächliche Reihenfolge der Tokens verloren geht. Der Grund dafür ist, dass bei dem Self-Attention-Algorithmus alle Zusammenhänge zwischen Token parallel ermittelt werden und nicht Wort für Wort. Das Ergebnis ist somit unabhängig von der Reihenfolge der Tokens. Der Algorithmus kommt so für die Wortfolge „Katze jagt Hund“ und „Hund jagt Katze“ zum den gleichen Ergebnis, obwohl die semantische Bedeutung offensichtlich eine andere ist.

Hier kommt das **Positional Encoding** ins Spiel. Dabei werden zu den ursprünglichen Embeddings, die aus den Tokens generiert wurden, Positionsdaten hinzugefügt, die der Transformer erkennen kann. Hierfür verwendet man abwechselnd Sinus- und Cosinus-Funktionen:

$$PE_{(pos,2i)} = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Hierbei ist pos die Position des Tokens in der Wortfolge, i ist die aktuell zu betrachtende Dimension für diesen Token, und d_{model} ist die Anzahl der Dimensionen im Model.

Die unterschiedlichen Frequenzen der Sinus- und Cosinus-Funktionen ermöglichen es, dass jede Dimension des Embeddings anders auf die Position reagiert. Sinus und Cosinus haben eine periodische Struktur, die es dem Transformer erlaubt, die Position der Tokens in der ursprünglichen Eingabe sowie die Positionsunterschiede zwischen Tokens zu ermitteln. Die abwechselnde Verwendung dieser Funktionen sorgt dafür, dass sie sich nie überlagern und somit unabhängige Informationen darstellen.

4.3 Addition und Normalisierung

Die Addition-und-Normalisierung-Komponente bewahrt originale Eingabeinformationen, die durch die schrittweise Verarbeitung im Transformer verloren gehen könnten. Nach jeder Verarbeitung, beispielsweise durch die Multi-Head-Attention, werden die Ausgabewerte mit den ursprünglichen Eingabewerten x_i addiert. Dadurch stehen auch für nachfolgende Komponenten die Originalinformationen weiterhin zur Verfügung.

$$z_i = x_i + \text{SubkomponentenOutput}_i$$

Bei der Normalisierung wird berechnet, wie viele Standardabweichungen ein Wert z_i vom Mittelwert entfernt ist. Dadurch werden die Werte standardisiert, indem die Verteilung zentriert (Mittelwert μ wird 0) und skaliert (Standardabweichung σ wird 1) wird. Anschließend werden die Werte abhängig vom Modelltraining durch Streckung (mit γ) und Verschiebung (mit β) angepasst. Diese Anpassung macht die Normalisierung flexibel und trainierbar.

$$\hat{z}_i = \frac{z_i - \mu}{\sigma}$$
$$y_i = \gamma \cdot \hat{z}_i + \beta$$

4.4 Feed-Forward-Schicht

Während **Multi-Head-Attention** lineare Zusammenhänge zwischen Tokens berechnet, benötigt der Transformer eine zusätzliche Komponente, um nicht-lineare Zusammenhänge zu erfassen. Hierfur wird die **Feed-Forward-Schicht** verwendet, die die addierten und normalisierten Ergebnisse aus der Multi-Head-Attention-Schicht weiterverarbeitet.

Die Feed-Forward-Schicht ist ein neuronales Netzwerk, das aus zwei linearen Transformationen mit einer nicht-linearen Aktivierungsfunktion in der Mitte besteht. Im ersten Schritt wird die Eingabematrix mit der Gewichtungsmatrix W_1 multipliziert, was die erste Schicht des neuronalen Netzes darstellt. Dabei entsteht eine Matrix mit viermal so vielen Dimensionen wie der Eingabematrix, die anschließend um einen Bias-Wert b_1 erweitert wird. Danach wird die erste Schicht durch die nicht-lineare Aktivierungsfunktion **ReLU** (eng. Rectified Linear Unit) mit der zweiten Schicht verbunden. Die ReLU-Funktion hat eine einfache nicht-lineare Eigenschaft: Wenn ein Eingabewert negativ ist, wird er zu 0; wenn er positiv ist, bleibt er unverändert und es bleibt bei einem linearen Zusammenhang:

$$\text{ReLU}(x) = \max(0, x)$$

Somit werden nur die positiven Werte in der weiteren Verarbeitung berücksichtigt.

In der zweiten Schicht werden die Werte aus der ReLU-Funktion mit der zweiten Gewichtungsmatrix W_2 multipliziert und somit wieder auf die ursprüngliche Eingabegröße skaliert. Auch hier wird der resultierende Vektor durch einen Bias-Wert b_2 ergänzt. Insgesamt kann die Feed-Forward-Schicht mathematisch so ausgedrückt werden:

$$\text{Feed-Forward}(x) = \text{ReLU}(W_1 x + b_1)W_2 + b_2$$

4.5 Encoder

Der **Encoder** übersetzt die Eingabewörter in eine abstrahierte Repräsentation. Ziel ist es hierbei, die semantischen und syntaktischen Informationen des Textes zu extrahieren und zu kodieren.

Am Beginn jeder Textverarbeitung wird, wie auch im Self-Attention Kapitel beschrieben, der Eingabe- ve
text tokenisiert und in Embeddings umgewandelten. Anschließend werden einmalig Positional-Encodings hinzugefügt, die Positionsdaten der Tokens im Eingabetext ergänzen.

Nun werden diese Embedding-Informationen an die erste Encoder-Schicht weitergeleitet. Ein Encoder hat laut Literatur mindestens 6 Encoder-Schichten. In einer Encoder-Schicht wird zunächst Multi-Head-Attention angewendet. Das Ergebnis der Multi-Head-Attention, also die relevanten, weiterzuanalysierenden

Informationen über Tokens werden mit den ursprünglichen Eingabe-Embeddings der Encoder-Schicht der Addition und Normalisierung unterzogen.

Im zweiten Teil der Encoder-Schicht werden die Werte dann mit Feed-Forward-Komponente weiterverarbeitet. Am Schluss werden diese weiterverarbeiteten Werte nocheinmal mit den Werten vor der Feed-Forward-Komponente addiert und normalisiert. Nun werden die Werte an die nächste Encoder-Schicht weitergegeben, welche die Prozedur wiederholt.

Nach der letzten Schicht liefert der Encoder eine kontextbewusste Darstellung der Eingabesequenz, in der jedes Token nicht nur seine ursprüngliche Bedeutung, sondern auch die relevanten Zusammenhänge mit anderen Tokens der Sequenz berücksichtigt. Nun können diese Informationen an den Decoder weitergegeben werden.

4.6 Decoder

Der **Decoder** ist die zweite Komponente des Modells und nutzt die abstrahierte Repräsentation des Encoders, um passende Outputs zu generieren. In jeder Iteration berechnet er das nächste Token, das zum Output hinzugefügt werden soll, wobei er den zuletzt generierten Token als zusätzlichen Kontext berücksichtigt.

Zu Beginn eines Decoder-Zyklus wird, analog zum Encoder, der Eingabetext in Embeddings umgewandelt und mithilfe des Positional Encodings mit Positionsinformationen angereichert. Der Unterschied liegt jedoch darin, dass der Eingabetext in der ersten Iteration des Decoders nur aus einem „Start-Token“ besteht, da noch keine weiteren Tokens generiert wurden. Diese initialen Embeddings werden dann an die Decoder-Schichten weitergeleitet, wo die eigentliche Verarbeitung stattfindet.

Im ersten Schritt jeder Decoder-Schicht wird eine spezielle Variante von Multi-Head-Attention angewendet: die **Masked Multi-Head Attention**. Hierbei werden zukünftige Tokens durch eine Maske verdeckt, sodass der Decoder nur bereits generierte Tokens verwenden kann. Dies verhindert, dass der Decoder Informationen vorwegnimmt. Während des Trainings wird der Decoder mit der gesamten Zielsequenz versorgt, wobei durch Maskierung nur bereits bekannte Tokens sichtbar sind. Dies simuliert den schrittweisen Vorhersageprozess. Die Ergebnisse der Masked Multi-Head Attention werden anschließend mit den ursprünglichen Eingabewerten addiert und normalisiert.

Im zweiten Schritt wird erneut Multi-Head Attention angewendet, allerdings in einer modifizierten Form, die als **Cross-Attention** bezeichnet wird. Hier werden die Queries (Q) regulär aus den Embeddings des Decoders berechnet, während die Keys (K) und Values (V) aus den Ergebnissen des Encoders stammen. Durch die separaten K und V aus dem Encoder kann der Decoder die im Eingabetext gespeicherten Kontextinformationen nutzen, um die Ausgabe gezielt zu steuern. Dies ist besonders wichtig, wenn der

Eingabetext viele Details enthält, die in der Antwort berücksichtigt werden müssen. Die Ergebnisse der Cross-Attention werden ebenfalls addiert und normalisiert.

Im dritten Schritt durchlaufen die Werte eine Feed-Forward-Schicht, die die Berechnungen weiter verfeinert. Auch hier folgt eine Addition mit den ursprünglichen Werten und eine Normalisierung.

Die resultierenden Embeddings werden entweder an die nächste Decoder-Schicht weitergeleitet, um den Prozess zu wiederholen, oder sie werden für die Ausgabe vorbereitet. Bei der Ausgabe wird jedes Embedding einem Token aus dem Vokabular zugeordnet, das als ganze natürliche Zahl dargestellt wird. Anschließend berechnet die Softmax-Funktion die Wahrscheinlichkeiten für alle möglichen Tokens und erzeugt dabei eine Wahrscheinlichkeitsverteilung über das gesamte Vokabular. Das Token mit der höchsten Wahrscheinlichkeit wird ausgewählt und zur finalen Ausgabe hinzugefügt. Dieses neue Token wird dann als Eingabe in den nächsten Decoder-Zyklus eingespeist. Der Vorgang wird fortgesetzt, bis der Decoder ein „End of Sequence“-Token (*EOS*) generiert, das das Ende der Ausgabe markiert.

5 Fazit