F A C H H O C H S C H U L E W E D E L

Bachelorthesis

in der Fachrichtung
Computer Games Technology
SS 2024

Thema:

AI Coding Addon für die Godot Engine

Eingereicht von:	Torben Oschkinat (CGT104590) Uhdeweg 17 22607 Hamburg Tel. 01578 4921244 E-Mail: torben@steffen-oschkinat.de
Erarbeitet im:	10. Semester
Abgegeben am:	11. Juli 2024
Betreuer:	Prof. Dr. Christian-Arved Bohn Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel. 04103 8048 40 E-Mail: bohn@fh-wedel.de
Zweitprüfer:	Prof. Dr. Dennis Säring Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel. 04103 8048 43 E-Mail: dsg@fh-wedel.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AI - Artificial Intelligence

KI - Künstliche Intelligenz

MBPP - Mostly Basic Python Programming

NLP - Natural Language Processing

RMSNorm - Root Mean Square Layer Normalization

UI - User Interface

VM - Virtuelle Maschine

Gender-Hinweis

Um eine schlechtere Lesbarkeit des Textes zu vermeiden, wird hier nur die männliche Form gewählt. Sie steht selbstverständlich ebenso stellvertretend für die weibliche Form.

1. Einleitung

In der heutigen digitalisierten Welt spielt die Künstliche Intelligenz (KI) eine immer wichtigere Rolle. Sie wird in vielen Bereichen wie dem autonomen Fahren aber auch dem Software-Engineering eingesetzt. Dabei setzen Entwickler mittlerweile immer häufiger darauf, sich von KIs Ideen, Code Erklärungen oder sogar ganze Codestücke generieren zu lassen. Dies trägt unter anderem zur Ideenanregung bei. Die beiden bekanntesten Tools in diesem Bereich sind ChatGPT und Copilot. Beide Tools sind Chatbots, die als virtueller Assistent dienen und den Nutzer bei allen möglichen Problemen unterstützen.

Die Godot Engine ist eine plattformübergreifende Spiele-Engine zur Erstellung von 2D- und 3D-Spielen, die vor allem gerne von Indie Developern verwendet wird. Sie enthält einen umfassenden Baukasten an Tools, so dass Spiele einfach und intuitiv entworfen werden können. Sie ist frei nutzbar und unterliegt der MIT-Lizenz. Diese erlaubt es den Nutzern, die Software sowohl für Open Source als auch Closed Source Software zu verwenden. Innerhalb der Godot Engine wird unter anderem die hauseigene Programmiersprache GDScript benutzt. Ihre Syntax ähnelt der von der Programmiersprache Python und hat die Vorteile, dass sie benutzerfreundlich und sehr Engine-nah ist.

Angesichts der zunehmenden Bedeutung von KI und der einfachen Zugänglichkeit der Godot Engine liegt die Kombination beider Themen nahe. Im Rahmen dieser Thesis soll die Entwicklung eines AI Coding Addons für die Godot Engine im Vordergrund stehen. Dieses Projekt umfasst die Erstellung und Integration einer selbst trainierten KI in die Godot Engine durch ein eigens programmiertes Addon. Dieses Addon soll die Nutzung von KI für Entwickler so zugänglich und nützlich wie möglich machen, so dass sie von jedem Entwickler eingesetzt werden kann, um Code zu generieren. Die KI soll in der Lage sein, dem Nutzer bei der Codegenerierung zu helfen, wodurch der Entwicklungsprozess vereinfacht und beschleunigt werden soll. Letztlich soll

dieses Addon nicht nur die Effizienz steigern, sondern auch die Kreativität der Entwickler unterstützen, indem es ihnen neue Wege und Möglichkeiten aufzeigt, ihre Ideen umzusetzen.

2. Anforderungen

Im Folgenden werden die Anforderungen an den praktischen Teil der Thesis genauer definiert.

2.1. Zielsetzung

Die Thesis umfasst zwei zentrale Themenbereiche. Zum einen die Erstellung eines Addons für die Godot Engine, welches die Schnittstelle zwischen dem Nutzer und der KI darstellt. Zum anderen wird eine KI entwickelt, die fähig sein soll, aus einem gegebenen englischen Prompt GDScript-Code zu generieren. Der Schwerpunkt der Arbeit liegt dabei auf der Entwicklung und dem Training der KI.

2.2. Forschungsfragen

1. Wie funktioniert die Erstellung eines Godot Addons?
2. Wie funktioniert die Erstellung eines KI-Modells zur Code Generierung?
3. Inwiefern ist das Tool, welches mit begrenzten Mitteln erstellt wurde, im praktischen Einsatz hilfreich?

2.3. Eingrenzung der Anforderungen

Durch das Addon soll es dem Nutzer möglich gemacht werden, mit der auf einem Server befindlichen KI zu kommunizieren. Dafür muss das Addon ein Inputfeld enthalten und eine Möglichkeit, dem Nutzer die Antwort der KI darzustellen. Dazu muss dieses Addon einheitlich in die Godot Engine implementiert und eingegliedert werden. Es wird im Rahmen dieser Arbeit kein Wert auf die Verschlüsselung der Verbindung zum Server oder die Geschwindigkeit der Antwort der KI gelegt.

Die KI soll in der Lage sein, mit einem Datensatz trainiert zu werden, der aus englischen Prompts und GDScript Code Paaren besteht. Dabei soll aus einer Instruktion GDScript Code generiert werden. Die KI soll dafür vorerst kein bereits existierendes vortrainiertes Modell benutzen, sondern lediglich auf die Struktur und das Training des selbst implementierten Modells zurückgreifen. Sollte dieses Modell nicht erfolgreich Code generieren, wird auf ein vortrainiertes Modell zurückgegriffen.

2.4. Aufbau der Thesis

Die Thesis ist in drei Teile gegliedert:
1. Die Erstellung und das Design des Addons.
2. Die Erstellung und das Training der KI
3. Das Aufsetzen der KI auf einem Server und die Kommunikation zum Addon in der Engine.

Diese drei Teile werden systematisch in verschiedenen Phasen der Thesis bearbeitet und im Nachhinein zu einem Gesamtprojekt zusammengefügt. Das Gesamtensemble bildet den praktischen Teil dieser Arbeit.

3. Addon

Das Addon ist der erste Teil der Arbeit, da es das Grundgerüst innerhalb der Godot Engine darstellt und somit der maßgebliche Bestandteil ist. Bei der Erstellung geht es darum, dem Nutzer mit einem simplen Design und einfachen Bedienelementen den Zugang zu der KI so einfach wie möglich zu machen.

3.1. Design

Das Addon passt sich an die bestehende Designsprache der Godot Engine an und verschmilzt mit den bestehenden Elementen in der Engine. Dabei werden nicht nur die einfachen rechteckigen Formen genutzt, sondern auch die für die Godot

Engine typischen Farben. Die Hauptfarben in der Godot Engine sind im Folgenden aufgelistet:

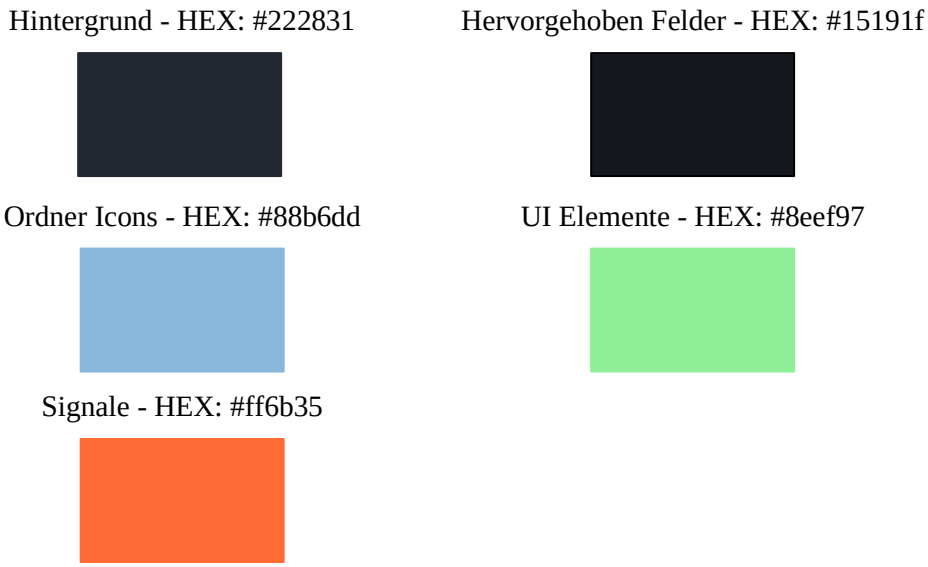

Abbildung 1: Hauptfarben der Godot Engine

Die Hintergrundfarbe und die Farbe für hervorgehobene Felder haben dabei die gleiche Aufgabe wie in der Godot Engine selbst erhalten. Sie grenzen die einzelnen Bereiche innerhalb des Addons systematisch ab und schaffen eine aufgeräumte und saubere Entwicklungsumgebung.

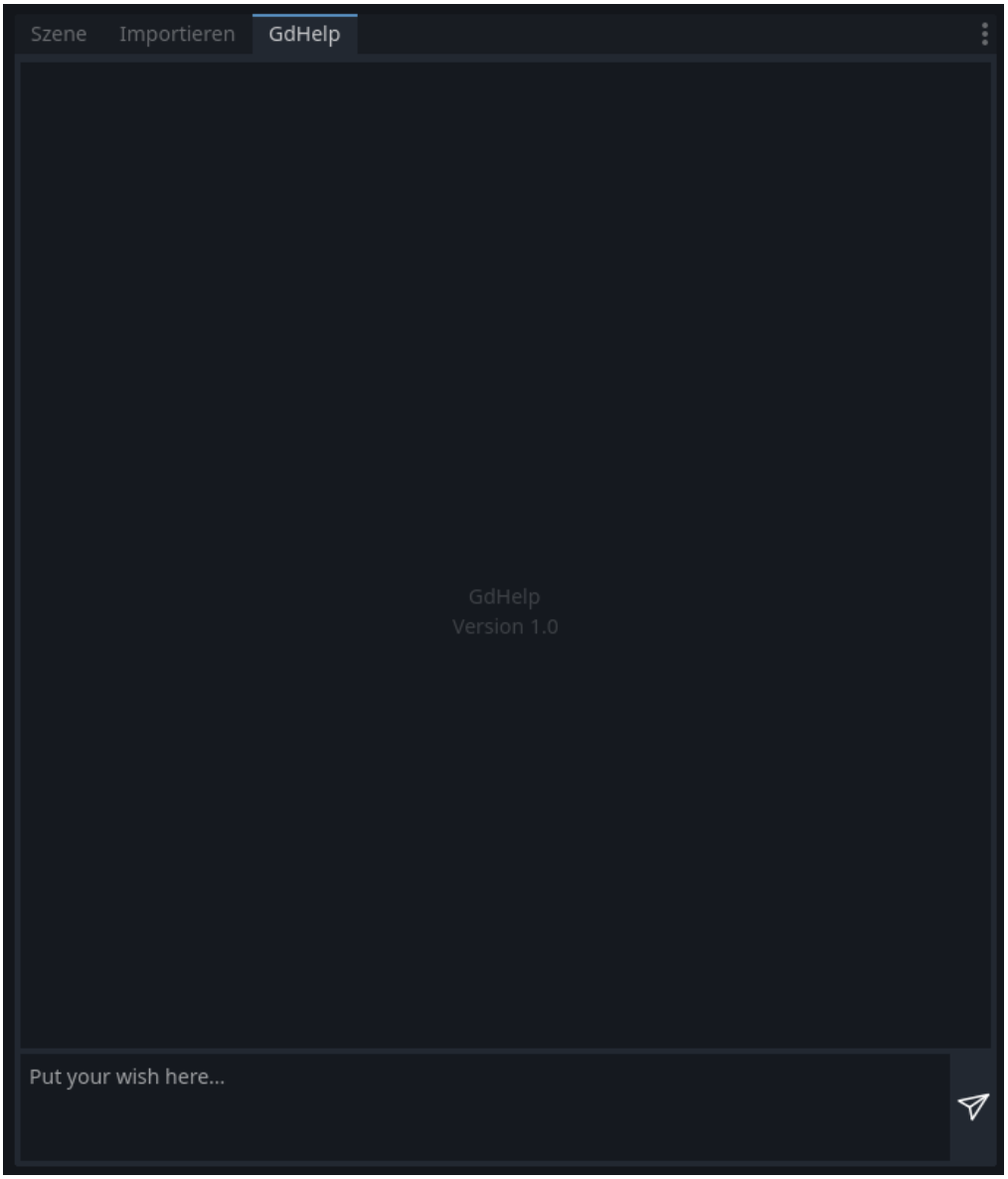

Abbildung 2: Addon in der Godot Engine

Die ursprüngliche Farbe des Ordner Icons wurde wiederverwendet, um einen Hover-Effekt auf die Knöpfe zu legen und dem Nutzer somit visuelles Feedback zu geben, sobald er eine Aktion auslösen kann. Hierdurch bleibt das User Interface (UI) weiterhin intuitiv bedienbar.

Abbildung 3: Hover Effekt auf dem „Senden Knopf"

Die Farbe der Signale wird wiederverwendet, um zu signalisieren, dass im Verlauf der Anfrage ein Problem aufgetreten ist und die Anfrage möglicherweise nicht bearbeitet werden kann. Zudem kann das gesamte Addon auf Wunsch vom Nutzer frei an eine gewünschte Position in der Engine verschoben werden, so dass ein individuell auf den Nutzer abgestimmter Workflow entstehen kann.

3.2. Funktionalität

Das Addon ist systematisch in zwei Bereiche aufgeteilt. Einerseits die Input Sektion im unteren Bereich, welche sich aus einem Input Feld und einem “Senden” Knopf zusammensetzt. Andererseits die im oberen Bereich liegende Output Sektion, welche aus einem Chatverlauf besteht und sowohl die Anfragen als auch die Antworten anzeigt. Statt lediglich die Anfragen im eigentlichen Addon anzuzeigen und die Antworten der KI direkt dem Code des Nutzers hinzuzufügen, wird auf ein klassisches Chat Layout gesetzt. Um die Option des einfachen Kopierens aber dennoch beizubehalten, hat die Antwort im Gegensatz zur Anfrage einen weiteren Knopf, der den Code direkt in die Zwischenablage des Nutzers kopiert. Dadurch bleibt der Nutzer mobil und muss gleichzeitig keinen potenziellen falschen Code nach der Anfrage wieder aus dem bereits implementierten Code herauslöschen.

4. Künstliche Intelligenz

Intelligenz kann als Fähigkeit betrachtet werden, effektive Lösungsstrategien zu erlernen und anzuwenden. Ein vollständig funktional programmierter Roboter ist zwar präzise, aber nicht intelligent. Der Begriff der Künstlichen Intelligenz wurde erstmals 1955 von Stanford-Professor John McCarthy geprägt und von ihm als die Wissenschaft und Technik, intelligente Maschinen zu bauen definiert. Heutzutage sind Künstliche Intelligenzen darauf ausgelegt, ähnlich wie Menschen eigenständig zu lernen.[1]

[1] Vgl. Manning, Christopher (2020)

Die Forschung rund um Künstliche Intelligenz hat in den letzten Jahrzehnten enorme Fortschritte gemacht, insbesondere im Bereich der Verarbeitung von natürlicher Sprache. Ein besonderer Meilenstein in der Entwicklung von KI-Modellen war die Einführung der Transformer-Architektur, wie sie im Paper "Attention is All You Need" aus dem Jahre 2023 von Vaswani und seinem Team beschrieben wird.

Rekurrente neuronale Netzwerke sind derzeit führend in der Sequenzmodellierung, haben jedoch Einschränkungen bei der Parallelisierung. Der Transformer hingegen verzichtet auf Rekurrenz und nutzt stattdessen ausschließlich Aufmerksamkeitsmechanismen, um Abhängigkeiten herzustellen. Dieses ermöglicht eine höhere Parallelisierung und bessere Leistung in der maschinellen Übersetzung. Der Transformer stellt einen bedeutenden Fortschritt dar, indem er die Berechnung von Repräsentationen effizienter gestaltet und die Abhängigkeiten unabhängig von der Entfernung in der Sequenz modelliert.

Für die Erstellung meiner eigenen KI wurden zwei verschiedene Wege gewählt. Zuerst habe ich ein eigenes, nicht vortrainiertes Transformer Modell erstellt, welches dann auf einen Datensatz trainiert wurde. Das Modell und die Umsetzung im Code sind im Folgenden genau erläutert.

4.1. Nicht vortrainierter Transformer

Der Transformer wie er in dem Paper „Attention is all you need" beschrieben wird, ist ein neuronales Netz, welches ausschließlich Aufmerksamkeitsmechanismen benutzt, um „Sequenz zu Sequenz" Aufgaben zu lösen und dabei trotzdem mit Abhängigkeiten umgehen kann. Dadurch, dass das Modell nicht der herkömmlichen sequenziellen Struktur folgt, erleichtert es nicht nur die parallele Datenverarbeitung, sondern beschleunigt ebenfalls das Training, vor allem beim Verarbeiten von langen Sequenzen. Deshalb ist es für Aufgaben wie die Generierung von Sequenzen in unbestimmter Länge genau richtig geeignet.

4.1.1. Modell-Architektur

Das Transformer Modell besteht aus zwei hauptsächlichen Bauteilen, dem Decoder und Encoder, wobei der Encoder eine Repräsentation des Ausgangssatzes aufbaut und diese dann dem Decoder im zweiten Multi-Head Attention Layer übergibt, welcher auch als Cross-Attention oder Encoder-Decoder Attention Layer bezeichnet wird.

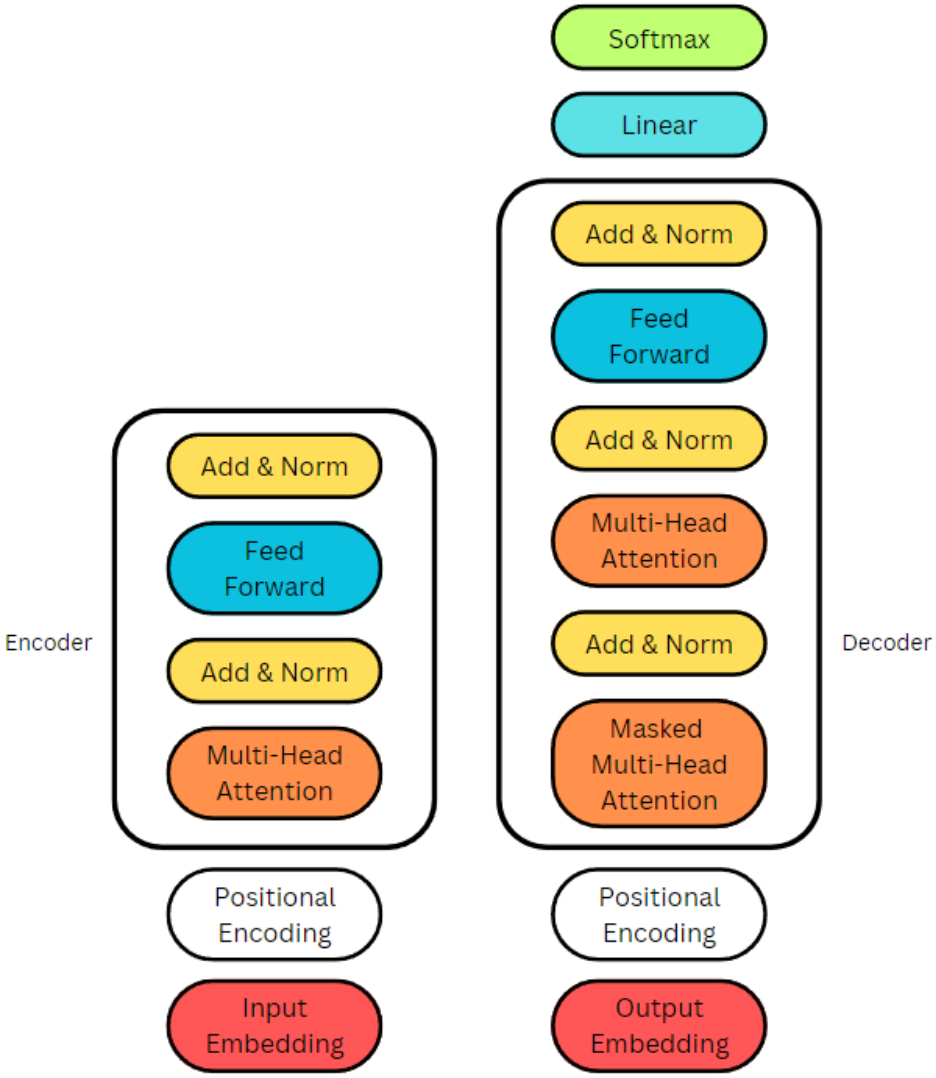

Abbildung 4: Transformer Architektur[2]

[2] Krüger, Ralph (2021)

4.1.1.1. Word Embedding

Word Embedding ist eine Vektorrepräsentation in einem höherdimensionalen Raum, die semantische Zusammenhänge zwischen Wörtern abbildet. Dabei liegen ähnliche Wortvektoren beieinander, wie zum Beispiel die Vektoren der Wörter „Buch“ und „Lesen“. Die Vektoren der Wörter, deren Bedeutung sich weniger ähnelt, liegen weiter auseinander, wie zum Beispiel „Buch“ und „Pferd“.

Ein Vektor entsteht dabei durch die Analyse eines Textes, bei der die Wahrscheinlichkeit berechnet wird, mit der Worte in einem gegebenen Kontext auftreten. Hierdurch wird die Semantik zwischen den Wörtern bestimmt. Ein Vektor eines Wortes besteht nach dem Embedding aus einer Reihe von Wahrscheinlichkeiten, mit der das Wort gemeinsam mit anderen Worten auftritt. Durch die Umwandlung der Wörter in Wortvektoren, werden sie in einen Vektorraum eingebettet, woher auch der Name des Embeddings kommt.[3]

Im Kontext des Transformers wird im ersten Schritt sowohl auf den Input als auch auf den Output ein solches Word Embedding angewendet. Dazu wird als erstes ein Tensor mit der Länge des Inputs erstellt, der daraufhin in das Embedding Layer weitergeleitet wird.

```
inputs = layers.Input(shape=(input_length,))
enc_embedding = layers.Embedding(vocab_size,
                head_size,)(inputs)
```

Abbildung 5: Embedding – Inputs

```
dec_inputs = layers.Input(shape=(target_length,))
dec_embedding = layers.Embedding(vocab_size,
                head_size)(dec_inputs)
```

Abbildung 6: Embedding – Targets

[3] Vgl. IT-P GmbH (o.J.)

4.1.1.2. Positionial Encoding

Transformer verarbeiten sequenzielle Daten wie natürliche Sprache parallel in einem einzigen Schritt, wodurch ihnen die Informationen zur Wortposition fehlen. Würde der Input lediglich vom Word Embedding verarbeitet werden, würde er das alte System des „Bag of Words“ benutzen, welches die Wörter ungeachtet ihrer Reihenfolge darstellen würde. Folgende drei Sätze würden dann alle gleich dargestellt werden:

Ich esse Kuchen
Kuchen esse Ich
Ich Kuchen esse

Die Wortpositionen müssen daher durch einen Positional Encoding Vektor zur Verfügung gestellt werden.[4] Dieser wird mit der folgenden Funktion errechnet:

```
def get_positional_encoding(length, depth):
    positions = np.arange(length)[:, np.newaxis]
    depths = np.arange(depth // 2)[np.newaxis, :] /
             (depth // 2)
    angle_rads = positions / (10000**depths)
    pos_encoding =
             np.concatenate([np.sin(angle_rads),
             np.cos(angle_rads)], axis=-1)
    return tf.cast(pos_encoding, dtype=tf.float32)
```

Abbildung 7: Funktion Positional Embedding

Um Positional Encoding Vektoren für die einzelnen Word Embeddings der Sequenzen zu erzeugen, werden die Positionial Encoding Vektoren für die Embeddings, die einen geraden Index besitzen, mithilfe einer Sinus-Funktion berechnet. Für die Embeddings mit ungeradem Index, erfolgt die Berechnung

[4] Vgl. TensorFlow (2024)

mithilfe einer Kosinus-Funktion. Nah beieinanderliegende Wörter haben nach der Berechnung ähnliche Positional Encoding Vektoren. Die daraus entstandenen Vektoren haben die gleiche Dimensionalität wie die Word Embeddings, weshalb sie anschließend mit den Word Embeddings addiert werden, um mit Positionsinformationen angereicherte Word Embeddings zu erstellen.[5] Im Code geschieht dies mit einer einfachen Addition.

```
enc_outputs = enc_embedding + enc_positional_encoding
```

Abbildung 8: Addition Embedding und Positional Encoding – Inputs

```
dec_outputs = dec_embedding + dec_positional_encoding
```

Abbildung 9: Addition Embedding und Positional Encoding – Outputs

Nach dieser Addition bestimmt die Nähe der Word Embedding Vektoren im Raum sowohl die semantische Ähnlichkeit als auch ihre absolute Position innerhalb der Sequenz. Dabei entsteht eine Matrix, der Form: Anzahl der Samples, Sequenz Länge und Tiefe (auch Modell Größe genannt).

4.1.1.3. Multi-Head Attention

Im Transformer werden die Berechnung im Attention Layer parallel mehrfach ausgeführt. Jede dieser Wiederholungen wird als Attention Head bezeichnet, woher auch der Name der Multi-Head Attention kommt. Dabei werden die drei Parameter Query, Key und Value N-fach aufgeteilt, wobei N die Anzahl der Attention Heads darstellt. Die einzelnen Berechnungen in den einzelnen Attention Heads werden am Ende kombiniert, um die finalen Aufmerksamkeitswerte zu erhalten. Durch diese Werte werden die Beziehungen für jedes Wort kodiert.[6]

Für jeden der Parameter Query, Key und Value existieren separate Lineare Layer mit ihren eigenen Gewichtsmatrizen. Der Input des Attention Layer wird in

[5] Vgl. Krüger, Ralph (2021)
[6] Vgl. Doshi, Ketan (2021)

diesen drei linearen Layer eingespeist und mit den Gewichtsmatrizen multipliziert, um die drei Matrizen für Query, Key und Value zu erzeugen.

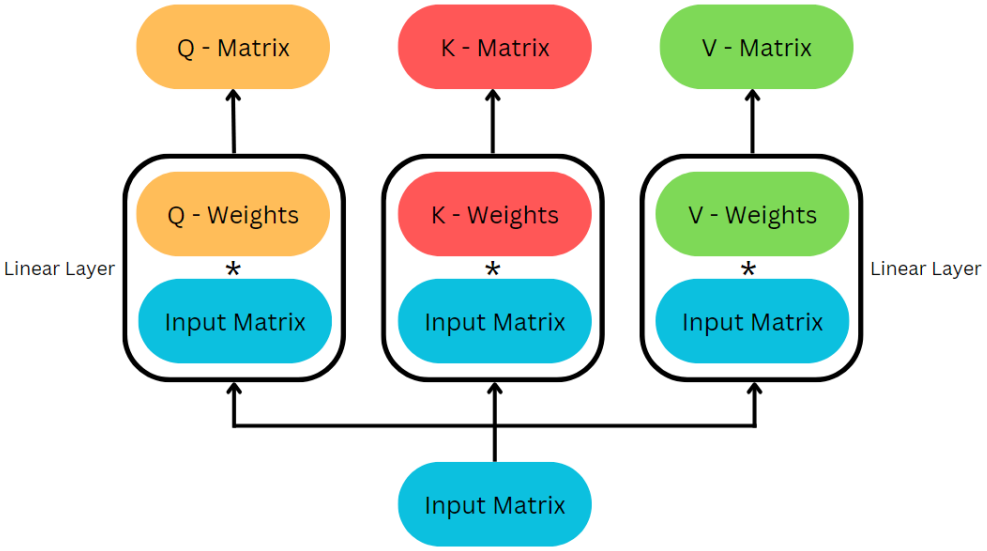

Abbildung 10: Erstellung der Q, K und Matrizen im Attention Layer

Dabei ist wichtig zu verstehen, dass bei der Aufteilung auf die Köpfe des Attention Layers die Matrizen nicht wirklich in separate Matrizen unterteilt werden. Es wird eine große Datenmatrix verwendet, die logische getrennte Abschnitte hat.[7]

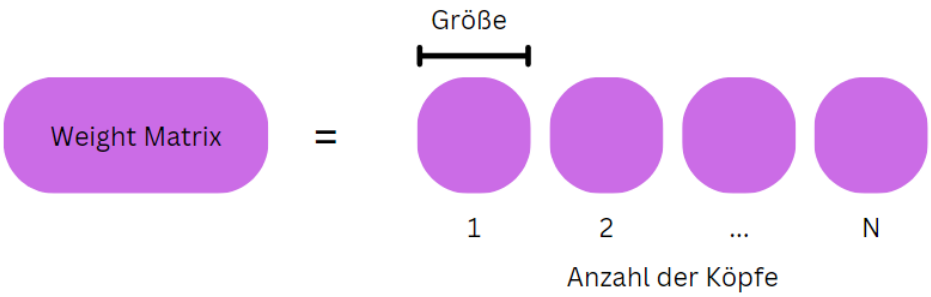

Abbildung 11: Aufteilung Gewichtsmatrix

[7] Vgl. Doshi, Ketan (2021)

Genau so wird auch nur ein linearer Layer verwendet für alle Attention Heads, wobei jeder Head in seinem eigenen Abschnitt der Matrix arbeitet. Die Berechnung für alle Köpfe können somit gleichzeitig durgeführt werden, so dass nicht nur weniger lineare Schichten benötigt werden, sondern auch gleichzeitig die Effizienz der Berechnung erhöht wird.

Um die eigentlichen Aufmerksamkeitswerte zu berechnen, müssen zuerst die beiden Matrizen Q und die transponierte Matrix K miteinander multipliziert werden. Danach wird das Ergebnis mit einer Maske addiert. Diese wird im einfachsten Fall dazu verwendet, dass das Attention Layer die Padding Token ignoriert, mit der die Sequenz auf eine einheitliche Länge gebracht wurden. Die daraufhin entstehende Matrix wird skaliert, indem sie durch die Quadratwurzel der Größe dividiert und ein Softmax auf das Ergebnis angewendet wird. Danach wird eine weitere Matrix Multiplikation zwischen dem Ergebnis und der letzten Matrix V durchgeführt. Das Ergebnis enthält die fertigen Aufmerksamkeitswerte.[8]

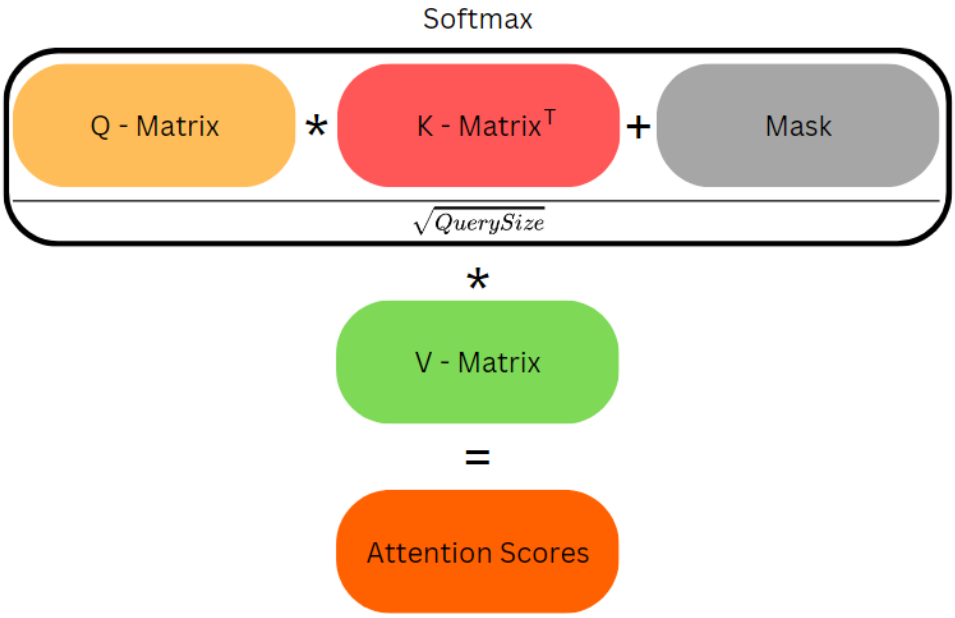

Abbildung 12: Berechnung Attention Scores

[8] Vgl. Doshi, Ketan (2021)

Das Encoder und Decoder Attention Layer funktioniert genau gleich, nur dass bei dem Decoder im ersten Attention Layer noch eine weitere Maske zusätzlich zu der Padding Maske verwendet wird: die „Causal Mask“.

Bei dem zweiten Attention Layer im Decoder, welcher auch Encoder-Decoder Layer genannt wird, wird der Output des Decoders im Attention Layer mit dem Output des ersten Attention Layers des Decoders kombiniert. Dabei wird für Key und Value der Output des Encoders verwendet, während für Query der Output des vorherigen Layers auf dem Decoder verwendet wird.[9]

4.1.1.3.1. Causal Mask

Da dem Transformer während des Trainings bereits die gesamte Output Sequenz zur Verfügung gestellt wird und er durch die parallele Verarbeitung auf das Wort an der aktuellen Position und die zukünftigen Wörter zugreifen könnte, wird eine Maske erstellt, die dies verhindert: die „Causal Mask“. Dabei werden alle Werte oberhalb der Matrix Diagonalen ignoriert, indem sie in der Maske auf 0 gesetzt werden.

```
causal_mask = 1 -
              tf.linalg.band_part(tf.ones((seq_length,
               seq_length)), -1, 0)
```

Abbildung 13: Erstellung der Causal Mask

Bei einer Sequenz Länge von 5 würde die dazu passende Causal Mask folgendermaßen aussehen:

[1. 0. 0. 0. 0.]

[1. 1. 0. 0. 0.]

[1. 1. 1. 0. 0.]

[1. 1. 1. 1. 0.]

[1. 1. 1. 1. 1.]

[9] Vgl. Doshi, Ketan (2021)

4.1.1.3.2. Padding Mask

Die Padding Maske sorgt lediglich dafür, dass die Padding Token bei der Berechnung der Aufmerksamkeitswerte nicht mit einbezogen werden. Dafür werden genau wie bei der Causal Mask die nicht zu beachtenden Stellen in der Maske auf 0 gesetzt.
Eine Tokenized Padded Sequence sieht beispielhaft wie folgt aus:
[1 5 11 0 0]
Die dafür erstellte Maske sieht daraus folgend wie folgt aus:
[1. 1. 1. 0. 0.]

Im Code wird dies über folgende Zeile geregelt, die automatisch alle Stellen in der Maske auf 0 setzt, wo in der Sequenz ebenfalls 0 steht:

```
enc_zero_mask = tf.cast(tf.not_equal(inputs, 0),
                dtype=tf.float32)[:, tf.newaxis,
                tf.newaxis, :]
```

Abbildung 14: Erstellung der Zero Mask

4.1.1.4. Add & Norm

Für das Add & Norm Layer werden zwei Komponenten benötigt: Einerseits der Output des Sublayers wie zum Beispiel eines Multi-Head Attention Layers. Andererseits die per Residual Verbindung um das Sublayer herumgeführte Matrix-Kopie. Diese werden zuerst miteinander addiert und anschließend wird eine Layer Normalisierung durchgeführt. Durch die Normalisierung kann die Trainingszeit des neuronalen Netzwerks verkürzt sowie die Generalisierungsleistung erhöht werden.[10]

```
x = layers.Add()([inputs, x])
x = layers.LayerNormalization(epsilon=1e-6)(x)
```

Abbildung 15: Layer Normalization und Addition

[10] Vgl. Krüger, Ralph (2021)

4.1.1.5. Feed Forward

Das Netzwerk des Feed Forward Layers besteht aus zwei linearen Transformationen mit einer ReLU-Aktivierung. Dabei wird nach den beiden linearen Transformationen ein Dropout Layer hinzugefügt, welches einige der Elemente zufällig auf Null setzt, um Überanpassung zu vermeiden.

```
x = layers.Conv1D(filters=ff_dim, kernel_size=1,
    activation="relu")(x)
x = layers.Conv1D(filters=inputs.shape[-1],
      kernel_size=1)(x)
x = layers.Dropout(dropout)(x)
```

Abbildung 16: Feed Forward Mechanismus mit Dropout

Das Dropout Layer setzt während des Trainings zufällig Eingabeeinheiten mit einer Häufigkeit von der Dropout-Rate auf 0, was dazu beiträgt, Overfitting zu verhindern. Eingaben, die nicht auf 0 gesetzt werden, werden mit 1 / (1 - dropout rate) skaliert, so dass die Summe über alle Eingaben unverändert bleibt.[11]

4.1.1.6. Linear und Softmax

Durch die lineare Schicht wird der Output des letzten Decoder-Blocks in einen Vektor transformiert, dessen Dimensionalität der Größe des zielsprachlichen Vokabulars des Transformers entspricht. Jede Dimension dieses Vektors repräsentiert ein Wort des Zielvokabulars und erhält einen entsprechenden Zahlenwert. Dieser „Vokabularvektor“ wird durch eine Softmax-Funktion normalisiert, sodass die Werte eine Wahrscheinlichkeitsverteilung bilden, die in Summe 1 ergibt.

[11] Vgl. Keras (o.J.) Dropout Layer

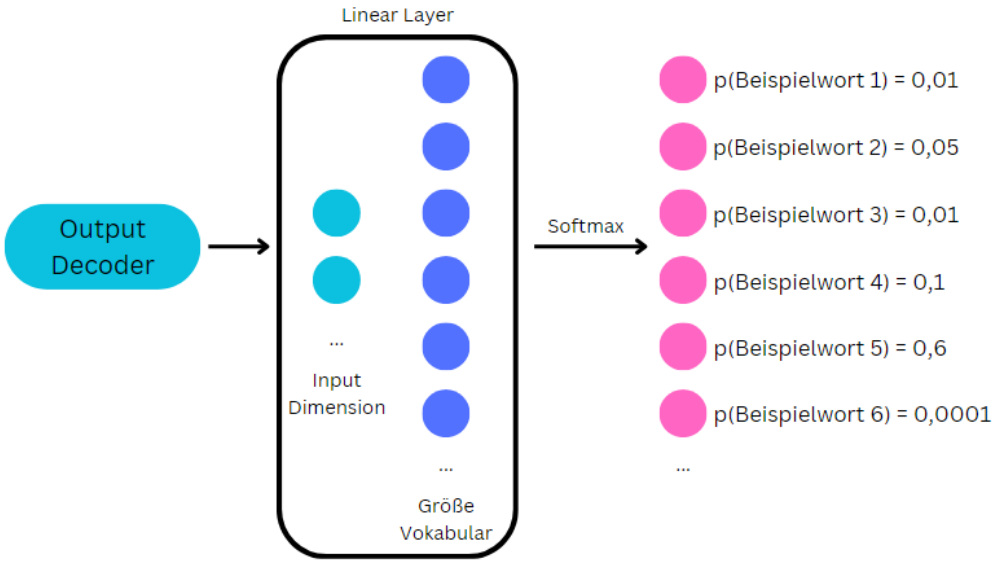

Abbildung 17: Linear und Softmax Umwandlung[12]

Danach wird im Normalfall das Wort mit der höchsten Wahrscheinlichkeit ausgegeben. Dies kann aber dazu führen, dass die KI im Verlauf des Satzes einen Fehler macht, der danach nicht mehr rückgängig gemacht werden kann. Dieser Effekt wird auch Garden-Path-Effekt genannt.[13] Im Code werden das Linear und Softmax Layer durch ein Dense Layer realisiert.

```
outputs = layers.Dense(vocab_size,
          activation="softmax")(dec_outputs)
```

Abbildung 18: Linear und Softmax Code

4.1.1.7. Encoder

Der Encoder Block setzt sich zusammen aus einem Multi-Head Attention Layer mit einer Padding Mask, gefolgt von einem Add & Norm Layer. Dem folgt eine Feed Forward Schicht, ebenfalls gefolgt von einem weiteren Add & Norm Layer. Dabei werden mit zwei Residual Verbindungen die zwei unveränderten Matrizen

[12] Krüger, Ralph (2021)
[13] Vgl. Krüger, Ralph (2021)

um die Hauptlayer der Multi-Head Attention und dem Feed Forward Network herumgeführt und in dem Add & Norm Layer wieder vereint. Im Code passiert dies durch einfaches Zwischenspeichern auf einer Variable „res_x“.

```
def transformer_encoder(inputs, zero_mask, head_size,
      num_heads, ff_dim, dropout=0.1):
    # Multi-Head Self-Attention
    x = layers.MultiHeadAttention(key_dim=head_size,
        num_heads=num_heads, dropout=dropout)(inputs,
        value=inputs, attention_mask=zero_mask)
    # Add & Norm
    x = layers.Add()([inputs, x])
    x = layers.LayerNormalization(epsilon=1e-6)(x)
    # Residual connection
    res_x = x
    # Feed-forward Network
    x = layers.Conv1D(filters=ff_dim, kernel_size=1,
        activation="relu")(x)
    x = layers.Conv1D(filters=inputs.shape[-1],
        kernel_size=1)(x)
    x = layers.Dropout(dropout)(x)
    # Add & Norm
    x = layers.Add()([res_x, x])
    return layers.LayerNormalization(epsilon=1e-
            6)(x), zero_mask
```

Abbildung 19: Transformer Encoder

4.1.1.8. Decoder

Der Decoder besteht aus zwei Multi-Head Attention Layern, denen jeweils eine Add & Norm Schicht folgt. Das erste Multi-Head Attention Layer erhalt dabei zusätzlich zu der Padding Mask eine Causal-Mask. Das zweite Multi-Head Attention Layer ist die Verbindung zum Encoder und erhält deshalb als Input ebenfalls den Output des Encoders, kombiniert mit einer Padding Mask. Auch hier werden die unveränderten Matrizen per Residual Verbindung um die Layer herumgeführt und in der Add & Norm Schicht wieder vereint.

```
def transformer_decoder(inputs, enc_outputs,
      dec_zero_mask, enc_zero_mask, head_size,
      num_heads, ff_dim, dropout=0.1):
    # Masked Multi-Head Self-Attention
    input_shape = tf.shape(inputs)
    seq_length = input_shape[1]
    causal_mask =
             tf.linalg.band_part(tf.ones((seq_length,
             seq_length)), -1, 0)
    combined_mask = tf.minimum(dec_zero_mask,
        causal_mask)
    x = layers.MultiHeadAttention(key_dim=head_size,
        num_heads=num_heads, dropout=dropout)(inputs,
        value=inputs, attention_mask=combined_mask)
    # Add & Norm:
    x = layers.Add()([inputs, x])
    x = layers.LayerNormalization(epsilon=1e-6)(x)
    # Residual connection
    res_x = x
    # Multi-Head Attention with Encoder Outputs
    x = layers.MultiHeadAttention(key_dim=head_size,
        num_heads=num_heads,
        dropout=dropout)(query=x,
```

```
        key=enc_outputs, value=enc_outputs,
        attention_mask=enc_zero_mask)
    # Add & Norm
    x = layers.Add()([res_x, x])
    x = layers.LayerNormalization(epsilon=1e-6)(x)
    # Residual connection
    res_x = x
    # Feed-forward Network
    x = layers.Conv1D(filters=ff_dim, kernel_size=1,
        activation="relu")(x)
    x = layers.Conv1D(filters=inputs.shape[-1],
        kernel_size=1)(x)
    x = layers.Dropout(dropout)(x)
    # Add & Norm
    x = layers.Add()([res_x, x])
    return layers.LayerNormalization(epsilon=1e-
            6)(x), dec_zero_mask
```

Abbildung 20: Transformer Decoder

4.1.1.9. Setup

Für die Erstellung des Modells wurde die Bibliothek Tensorflow in der Version 2.10 verwendet, die die grundlegenden Bestandteile für das Erstellen von Künstlichen Intelligenzen bereitstellt. Des Weiteren wurde die Bibliothek Pandas verwendet, um den Datensatz aus der Excel-Datei herauszulesen.

4.1.1.10. Model Complexity

Bei der Model Complexity handelt ich sich um die Komplexität des Modells. Dabei wird unter anderem die Dimension, die Anzahl der in Reihe geschalteten Transformer Blöcke und die Anzahl der Köpfe in den Multi-Head Attention Layern bestimmt. Durch die Anpassung dieser Hyperparameter kann sich die Performance stark verändern.

Für diese KI wurden folgende Hyperparameter verwendet:

Batch Size: 2

Head Size: 64

Number of Heads: 4

Feed-Forward Dimension: 256

Number of Transformer Blocks: 4

Dropout Rate: 0.1

Das daraus entstandene Modell hat damit eine Gesamtanzahl von 14 Millionen Parametern, die ebenfalls alle auch trainierbar sind.

4.1.2. Datensatz

Der Datensatz, der zum Trainieren des Modells verwendet wurde, ist ein bereits bestehender Datensatz[14] aus über 60.000 Prompts und den dazugehörenden GDScript Code Snippets. Aus Gründen der Übersichtlichkeit und einer stichprobenartigen Überprüfung der Codequalität und der Qualität des Datensatzes an sich, wurde das ursprüngliche json Dokument mit einem Script in eine Excel Tabelle umgewandelt.

```
# Open the JSON file and load the data
with open(json_file, mode='r', encoding='utf-8') as
       f:
    data = json.load(f)

# Open the CSV file for writing
with open(csv_file, mode='w', newline='',
       encoding='utf-8') as file:
    writer = csv.writer(file, delimiter=';')
```

[14] Minosvasilias (o.J.)

```
    # Write the header
    writer.writerow(['prompt', 'gdscript_code',
      'source'])

    # Process each entry in the JSON data
    for entry in data:
        prompt = entry['instruction']
        gdscript_code = entry['output']

        # Write the row to the CSV file
        writer.writerow([prompt, gdscript_code, ''])
```

Abbildung 21: Json to CSV Converter

Um den Datensatz im Hauptprogramm benutzen zu können, wird er mit Hilfe der Pandas Bibliothek geladen, sodass er im Folgenden weiterverarbeitet werden kann.

```
dataset_name = 'data.csv'
df = pd.read_csv(dataset_name, sep=';',
      encoding='latin1')
```

Abbildung 22: Laden des Datensatzes

Bevor ein Tokenizer erstellt wird, werden die Inputtexte einmal alle vorverarbeitet, um Lücken zwischen Worten und Sonderzeichen zu schaffen, da der Tokenizer diese sonst als ein großes Wort verstehen würde. Beispielsweise würde aus „_add():“ lediglich ein Token werden. Nachdem die Sonderzeichen vorher aufgetrennt werden, wird jedes der folgenden Zeichen ein einzelner Token: _, add, (,).

Der Tokenizer wird auf die Input- und Target Texte angepasst, so dass er danach ein Wörterbuch über alle Zeichen und Wörter enthält, die in den gesamten Texten vorkommen. Dabei erhält jedes Zeichen oder Wort eine eindeutige Zahl,

anhand der es zugeordnet werden kann. Anschließend wird der Tokenizer mit Pickle gespeichert, um ihn bei der Codegenerierung wiederzuverwenden.

```
tokenizer = Tokenizer(filters='', lower=False,
            oov_token='<OOV>')
tokenizer.fit_on_texts(all_texts)

with open('tokenizer.pkl', 'wb') as f:
    pickle.dump(tokenizer, f)
```

Abbildung 23: Tokenizer füllen und speichern

Bevor der Tokenizer die Texte verarbeitet wird außerdem an das Ende jedes GD-Script Code Snippets ein “End Of Sentence” Token gehängt, der der KI im Training signalisiert, dass diese am Ende des Satzes angekommen ist, bevor die Padding Token starten. Dieses wird verwendet, damit die KI ebenfalls ein Ende des Satzes erstellen kann, ohne die gesamte Sequenzlänge während der Prediction zu erstellen, wenn dies nicht nötig ist.

```
eos_token = '<EOS>'
df['gdscript_code'] =
df['gdscript_code'].apply(lambda x: x +
                      ' ' + eos_token)
```

Abbildung 24: Hinzufügen EOS Token

Nachdem die Texte vorbereitet wurden, werden sie mit dem Tokenizer in Sequenzen von Zahlen, die zum im Tokenizer gespeicherten Wörterbuch passen, umgewandelt.

```
input_sequences = tokenizer.texts_to_sequences(
                  df['prompt'].tolist())
target_sequences = tokenizer.texts_to_sequences(
                  df['gdscript_code'].tolist())
```

Abbildung 25: Umwandlung Text in Sequenz

Eine beispielhafte Umwandlung von einem Prompt in eine Sequenz sieht so aus:

Prompt:
“Free up the memory used by the current node instance .”

Tokenized Prompt:
[1205, 228, 10, 856, 366, 72, 10, 46, 31, 175, 5]

Ebenfalls abgespeichert werden die maximale Länge der Input- und Output-Sequenzen, um auch diese bei der separaten Codegenerierung wiederverwenden zu können. Anschließend werden alle Sequenzen mit Padding Token auf die maximale Länge aufgefüllt, damit die KI sie verarbeiten kann. Dabei werden Padding Token (eine 0), in diesem Fall per Parameter definiert (padding=‘post‘), an das Ende der tokenisierten Sequenz angehangen, bis die maximale Länge erreicht ist.

```
input_sequences = pad_sequences(input_sequences,
                  maxlen=max_input_len,
                 padding='post')
target_sequences = pad_sequences(target_sequences,
                   maxlen=max_target_len,
                   padding='post')
```

Abbildung 26: Padding der Sequenzen

Nachdem die Padding Token hinzugefügt wurden sieht die Tokenized Prompt aus dem vorherigen Beispiel, folgendermaßen aus:
[1205 228 10 856 366 72 10 46 31 175 5 0 0 0 0 0 0 0 0 0]

Die Größe des Wortschatzes wird im nächsten Schritt um eins erhöht, um das Padding-Token zu berücksichtigen.

```
vocab_size = len(tokenizer.word_index) + 1
```

Abbildung 27: Erhöhung des Wortschatzes um Padding Token

Nachdem die Sequenzen vorbereitet sind, wird der Datensatz in einen Trainings- und Validierungsdatensatz aufgeteilt. Dabei wird ein Verhältnis von 80 Prozent für das Training und 20 Prozent für die Validierung verwendet. Zuerst wird dafür ein Array mit Indizes erstellt, welches Indizes bis zur maximalen Anzahl der Sequenzen enthält:

```
indices = np.arange(input_sequences.shape[0])
np.random.shuffle(indices)
```

Abbildung 28: Erstellen des Indizes-Arrays

Danach wird der Split Index berechnet, an dem die Daten aufgeteilt werden und der Datensatz in den Trainings und Validation Datensatz aufgeteilt:

```
split_index = int((1 - SPLIT_RATIO) *
              input_sequences.shape[0])

train_indices = indices[:split_index]
val_indices = indices[split_index:]
```

Abbildung 29: Datensatz in Training und Validation aufteilen

Im Folgenden werden die einzelnen Teile für die X und Y Komponenten des Tensorflow-Datensatzes erstellt.

```
X_train = input_sequences[train_indices]
y_train = target_sequences[train_indices]
```

```
X_val = input_sequences[val_indices]
y_val = target_sequences[val_indices]
```

Abbildung 30: X und Y Komponenten Datensatz erstellen

Zum Erstellen des finalen Datensatzes, der in der KI beim Training als Input und zur Validierung eingesetzt wird, wurde folgende Funktion definiert:

```
def create_dataset(inputs, targets, batch_size):
    dataset =
             tf.data.Dataset.from_tensor_slices(
             (inputs, targets))
    dataset =
dataset.shuffle(buffer_size=len(inputs))
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(buffer_size=
              tf.data.experimental.AUTOTUNE)
    return dataset
```

Abbildung 31: Funktion zur Erstellung eines Tensorflow Datensatzes

In dieser wird als erstes ein Tensorflow-Datensatz aus den Eingabe- und Zielsequenzen erstellt. Danach werden die Daten noch einmal durchgemischt und anschließend in Batches aufgeteilt. Zum Abschluss wird „prefetch“ benutzt, um die Leistung zu verbessern. Dabei bestimmt der Parameter „buffer_size“ die optimale Puffergröße automatisch. Diese Funktion wird anschließend verwendet, um die finalen Datensätze zu erstellen:

```
train_dataset = create_dataset((X_train, y_train[:,
                  :-1]), y_train[:, 1:], BATCH_SIZE)
val_dataset = create_dataset((X_val, y_val[:, :-1]),
                  y_val[:, 1:], BATCH_SIZE)
```

Abbildung 32: Finale Datensätze erstellen

Generell ist zu sagen, dass je größer der Datensatz ist, desto besser wird die KI auch performen, da sie eine breitere Basis hat, von der sie sich bedienen kann. Allerdings hat ein größerer Datensatz nicht nur Vorteile. Denn je größer der Datensatz wird, desto mehr Leistung wird gebraucht, um die KI letzten Endes zu trainieren. Zudem steigt die erforderliche Leistung ebenfalls mit der Batchgröße, weshalb in diesem Fall eine kleine Batchgröße gewählt wurde.

Als Batchgröße bezeichnet man die Anzahl an Datenpunkten, in diesem Fall Sequenzen, die gleichzeitig während des Trainingsprozesses verarbeitet werden. Dabei wird der eigentliche Datensatz nochmal in kleine Teile aufgeteilt. Die Größe der Batches beeinflusst direkt die Geschwindigkeit des Trainings sowie die Qualität des Modells. Einer größere Batchsize kann zu schnellerem Training führen, weil gleichzeitig mehr Daten verarbeitet werden, aber gleichzeitig wird das Modell weniger gut generalisiert, da es weniger häufig aktualisiert wird.

4.1.3. Training

Zum Trainieren wurde am Anfang der Bachelorthesis lediglich der eigene PC benutzt. Allerdings wurde mit einem Hochskalieren der Parameter die KI nicht nur komplexer, sondern auch die benötigte Rechenpower immer größer. Aus diesem Grund wurde das Training der KI über die Google Cloud ausgelagert.

4.1.3.1. Google Cloud

Google Cloud ist ein Service von Google, welcher unter anderem virtuelle Maschinen bereitstellt. Diese wurde genutzt, um die schnell hoch skalierende benötigte Rechenleistung der KI zu handhaben.

4.1.3.2. Setup

Für die virtuelle Maschine wurden folgende Komponenten verwendet:

- 4 GPUs a 16GB
- 1n8 mit 32GB RAM
- Debian Linux mit Cuda
- Anaconda Prompt zur Erstellung der Umgebung auf der VM

- Python 3.10
- cudatoolkit 11.2
- cudnn 8.1.0
- Tensorflow 2.10

4.1.3.3. Loss und Accuracy

Damit der Loss und die Accuracy richtig bestimmt werden können, müssen diese ebenfalls mit einer Maske berechnet werden, damit die Padding Token nicht beachtet werden. Dazu wurde jeweils eine eigene Funktion erstellt, die das übernimmt.

```
def masked_loss(label, pred):
    mask = tf.cast(label != 0, dtype=pred.dtype)
    spars_cat_cross =
      tf.keras.losses.SparseCategoricalCrossentropy(
      reduction='none')
    loss = spars_cat_cross(label, pred)
    masked_loss = tf.reduce_sum(loss * mask) /
                  tf.reduce_sum(mask)
    return masked_loss
```

Abbildung 33: Masked Loss Function[15]

Als erstes wird eine Maske erstellt, die wie die Padding Mask funktioniert. Danach wird die SparseCategoricalCrossentropy berechnet, wobei der Loss für jedes Element einzeln berechnet wird. Anschließend wird das Ergebnis mit der Maske multipliziert, wodurch diese angewendet wird.

Die Masked Acucuracy wird auf dem gleichen Prinzip berechnet. Zuerst wird ebenfalls eine Padding Maske erstellt. Danach wird der höchste Wert der letzten Achse der Vorhersage gespeichert. Nun werden die vorhergesagten Klassen mit

[15] TensorFlow (2024) Neural machine translation with a Transformer and Keras

denen der tatsächlichen verglichen. Dadurch entsteht ein Tensor mit Wahrheitswerten. 1 für korrekte Vorhersagen und 0 für falsche. Dieser wird abschließend mit der Padding Mask multipliziert, um diese anzuwenden. Im Return wird dann als Ergebnis der Durchschnitt der richtigen Vorhersagen ausgegeben.

```
def masked_accuracy(label, pred):
    mask = tf.cast(label != 0, dtype=tf.float32)
    pred = tf.argmax(pred, axis=-1,
             output_type=tf.int32)
    label = tf.cast(label, tf.int32)
    correct = tf.cast(tf.equal(pred, label),
             dtype=tf.float32)
    masked_correct = correct * mask
    return tf.reduce_sum(masked_correct) /
             tf.reduce_sum(mask)
```

Abbildung 34: Masked Accuracy Function[16]

4.1.3.4. Early Stopping

Early Stopping ist eine Technik zur Vermeidung von Overfitting des Trainings eines KI-Modells. Es handelt sich dabei um einen Callback-Mechanismus, der das Training stoppt, sobald die Leistung auf einer Validierungsdatenmenge über mehrere Epochen hinweg nicht mehr verbessert wird. Dies geschieht typischerweise durch die Überwachung einer bestimmten Metrik.

```
early_stopping = EarlyStopping(monitor='val_loss',
      patience=PATIENCE, restore_best_weights=True)

transformer.fit(train_dataset, epochs=EPOCHS,
      validation_data=val_dataset,
```

[16] TensorFlow (2024) Neural machine translation with a Transformer and Keras

```
        callbacks=[early_stopping])
```

Abbildung 35: Early Stopping

In diesem Fall wird der val_loss als Metrik verwendet, welcher der Validierungsverlust ist. Diese wird verwendet, um zu prüfen, ob das Training gestoppt werden soll. Patience ist die Anzahl der Epochen ohne Verbesserung des überwachten Metrik, bevor das Training gestoppt wird. Wenn nach der Anzahl der Epochen keine Verbesserung erfolgt, wird das Training gestoppt. Restore_best_weights gibt an, dass die Gewichte auf den Zustand zurückgesetzt werden, in dem sie den besten Wert der überwachten Metrik erreicht haben.

4.1.3.5. Trainingszeiten

Aufgrund der großen Komplexität des Modells und des Datensatzes, hat die KI auf der VM 5 ½ Stunden trainiert für eine Epoche. Insgesamt wurde die KI über 2 Epochen trainiert.

4.1.4. Fazit nicht vortrainierter Transformer

Ein Modell direkt auf einen Datensatz mit einer Größe von nur 60.000 zu trainieren, führt in der Regel zu keinen relevanten und guten Ergebnissen bei Vorhersagen des KI-Modells. Die KI ist lediglich in der Lage eine, ohne jegliche Syntax geformte Sequenz aus Unterstrichen, Zeilenumbrüchen und Klammern zu erzeugen. Dieser Code ist weder in irgendeiner Art und Weise syntaktisch wertvoll noch im realen Programmieralltag zu gebrauchen. Aus diesem Grund wurde im zweiten Schritt eine vortrainierte KI benutzt, die auf den Datensatz sensibilisiert wurde.

4.2. Vortrainiertes Modell

Um bessere Ergebnisse zu erzielen wurde im zweiten Schritt auf ein vortrainiertes Modell gesetzt, welches ebenfalls die Transformer Struktur als Basis für das Modell verwendet. Um ein möglichst gutes Ergebnis zu erreichen,

wurde das Code Llama Modell[17] von Meta aus der Huggingface Bibliothek verwendet, welches unter anderem von Grund auf darauf ausgelegt ist, Code zu generieren und auf Instruktionen zu reagieren. Code Llama ist ein weiterentwickeltes Modell und basiert auf dem Grundmodell Llama 2 von Meta. Das Modell ist in diesem Zustand bereits in der Lage Code anhand einer Instruktion zu generieren, fügt allerdings noch ergänzende Informationen zu dem entwickelten Code hinzu. Des Weiteren ist das Modell noch nicht auf GDScript gefinetuned worden. Das Modell Code Llama gibt es in den Größen 7B, 13B, 34B und 70B Parametern, wobei das 13 Milliarden Parameter Modell im Verhältnis der Größe zu den anderen Modellen die besten Resultate erzielt. Es ist in HumanEval und MBPP besser als das deutlich größere 70B Basismodell Llama 2.[18]

HumanEval	pass@1	pass@10	pass@100
Llama 2 – 70B	30,5%	59,4%	87,0%
Code Llama – Instruct – 13B	42,7%	71,6%	91,6%
MBPP	pass@1	pass@10	pass@100
Llama 2 – 70B	45,4%	66,2%	83,1%
Code Llama – Instruct – 13B	49,4%	71,2%	84,1%

Tabelle 1: pass@ Ergebnisse für HumanEval und MBPP[19]

Bevor es diese beiden Metriken gab, wurde die Korrektheit des Codes anhand des BLEU-Scores ermittelt, welche die Ähnlichkeit zwischen Textmengen misst. Diese Metrik misst allerding nicht, ob ein Code ein bestimmtes Problem lösen kann. Bei bestimmten Problemen kann die Ähnlichkeit zur Lösung komplett abweichen, aber trotzdem funktional korrekt sein.

[17] HuggingFace (o.J.) Code Llama
[18] Vgl. Roziere, Baptiste u.a. (o.J.)
[19] Vgl. Roziere, Baptiste u.a. (o.J.)

HumanEval ist ein Datensatz bestehend aus 164 Programmieraufgaben, der die pass@k Metrik verwendet. Mit dieser kann die funktionale Korrektheit des Codes evaluiert werden. Die pass@k Metrik ist die Wahrscheinlichkeit, dass mindestens eines der ersten k generierten Codestücke die Unit-Tests bestehen kann. Die Metrik ist dabei an dem Vorgehen von menschlichen Entwicklern inspiriert, welche die Korrektheit von ihrem Code ebenfalls anhand von Unit-Tests überprüfen.[20]

Der zweite Datensatz MBPP (Mostly Basic Python Programming) ist eine Sammlung von ungefähr 1.000 Python Programmieraufgaben, die von Programmieranfängern gelöst werden könnten. Jede Aufgabe besteht aus einer Aufgabenbeschreibung, einer Musterlösung und 3 Tests.[21]

Anhand dieser beiden Datensätze und der pass@k Metrik kann die Fähigkeit funktionalen Code zu generieren, evaluiert werden.

4.2.1. Code Llama

Code Llama basiert auf dem Basismodell Llama 2, welches anders als herkömmliche Code Modelle nicht nur auf Code trainiert wurde, sondern auf generellen Text und Code Daten. Es handelt sich bei Code Llama um eine optimierte autoregressive Transformer Architektur.

4.2.1.1. Modellarchitektur

Dieses Modell ist eine Variante der Transformerstruktur aus Kapitel 4.1. Sie basiert dabei auf dem Decoder-Only Konzept, wobei lediglich der Decoder Block aus dem Transformer benutzt wird.

[20] Vgl. Deepgram (2023)
[21] Vgl. Meta AI (o.J.)

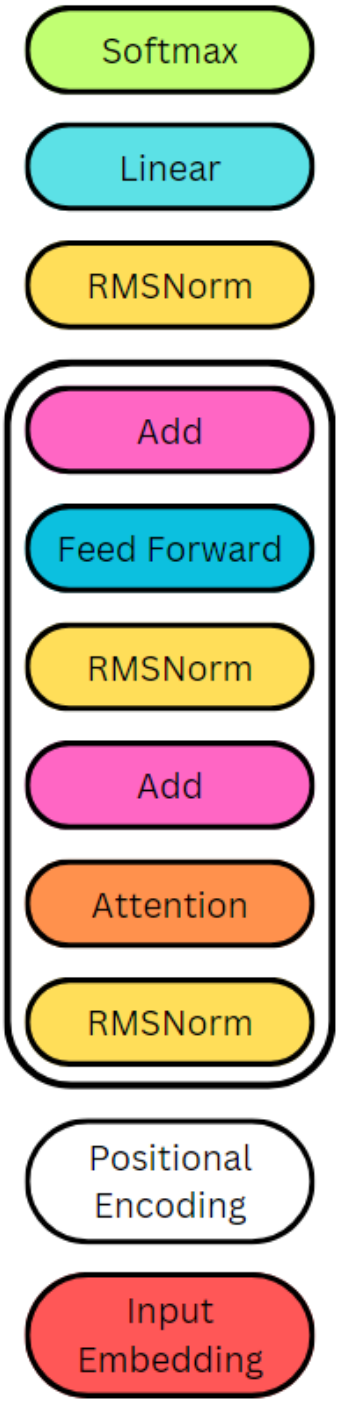

Abbildung 36: Code Llama Modellarchitektur

Der größte Unterschied zu dem normalen Encoder-Decoder Transformer ist, dass dieser aus einer Kombination von Encodern und Decodern besteht, während das Modell von Meta lediglich aus einer Kombination von Decodern besteht. Aus diesem Grund entfällt das im Standard Transformer verwendete zweite

Attention Layer (Cross-Attention Layer). Der Decoder besteht lediglich aus zwei RMSNorm Layern, einem Feed Forward Layer und zwei Add Layern, die die Residual Verbindung zu dem Ergebnis des vorherigen Layer addieren.

4.2.1.1.1. Normalisierung

Anders als im normalen Transformer, wo die Layernormalisierung nach dem eigentlichen Layer stattfindet, wird eine Pre-Normalisierung mit einem RMSNorm (Root Mean Square Layer Normalization) Layer verwendet. Diese ist eine vereinfachte Version der Layernormalisierung und verbessert die Trainingsstabilität und Generalisierung. Abgesehen davon ist sie effizienter in der Berechnung.

4.2.1.1.2. SwiGLU-Aktivierungsfunktion

Statt der ReLU-Aktivierungsfunktion verwendet Code Llama die SwiGLU-Aktivierungsfunktion in ihren Feed Forward Layern. SwiGLU ist ein elementweises Produkt aus zwei linearen Transformationen, wovon auf einer eine Swish-Aktivierung angewendet wurde. Mit drei Matrixmultiplikationen ist SwiGLU zwar rechenintensiver als eine RELU-Aktivierungsfunktion, aber zeigt im Vergleich eine bessere Leistung.[22]

4.2.1.1.3. RoPE Positional Encoding

Zusätzlich wird beim Positional Encoding das Rotary Positional Encoding verwendet und nicht das absolute Position Encoding. Rotary Positional Encoding ist eine Art des Positional Encodings, welche die absolute Positionsinformationen mit einer Rotationsmatrix kodiert und eine explizite relative Positionsabhängigkeit mit in die Self-Attention einbezieht.[23] Die Rotationsmatrix rotiert dabei den Wort-Vektor um einen bestimmten Winkel und besteht aus mehreren 2D Rotationsmatrizen.

[22] Vgl. Wolfe, Cameron R. (2023)
[23] Vgl. Kianyew, Ngieng (2024)

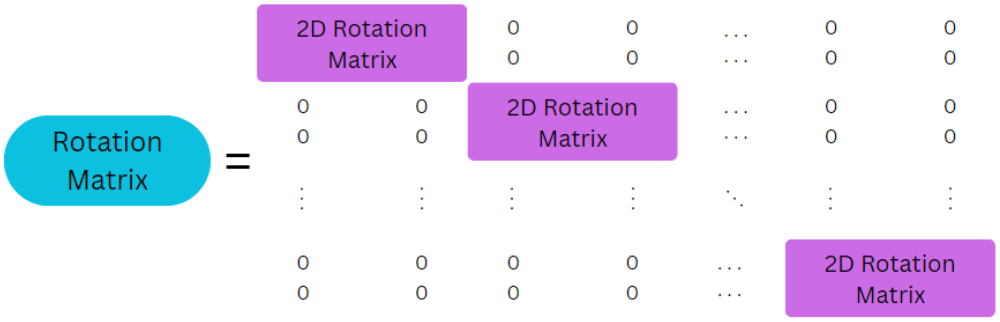

Abbildung 37: Rotationsmatrix[24]

Diese 2D Matrizen werden von den beiden trigometrischen Eigenschaften Sinus und Cosinus abgeleitet.

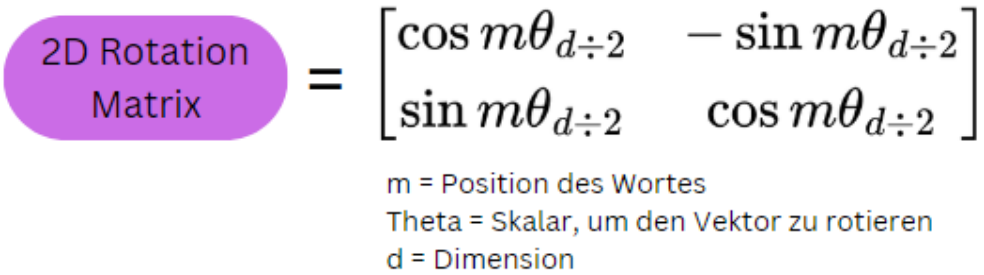

Abbildung 38: 2D Rotationsmatrix

Diese Rotationsmatrix wird verwendet, um eine D-dimensionale Matrix zu rotieren, wobei der Drehwinkel jeder Matrix durch m * Theta bestimmt wird. m entspricht der Position des Wortes und Theta ist ein Skalar, welches bei allen Wörtern gleichbleibt und eine exponentiell aussehende Kurve erzeugt. Wenn m zunimmt, verschiebt sich die Kurve nach oben, um einen größeren Positionsabstand anzuzeigen.[25]

Durch diese Methode vereint Rotary Positional Encoding die beiden Methoden des Absoluten und Relativen Positional Encodings. Ersteres wird in dem Standard-Transformer verwendet, bezieht allerdings die relativen

[24] Vgl. Su, Jianlin (2023)
[25] Vgl. Kianyew, Ngieng (2024)

Positionsinformation nicht mit ein. Dieses wird beim Relativen Positional Encoding zwar angewendet, führt aber dazu, dass es berechnungsintensiver ist und nicht für Inferenzen geeignet ist. Das Rotary Positional Encoding vereint die Vorteile beider Verfahren und hat darüber hinaus ebenfalls den Vorteil, dass es durch die Matrix einen rechnerisch effizienten Weg zur Berechnung bietet.[26]

4.2.1.1.4. Grouped-Query Attention

Des Weiteren benutzt das Code Llama Modell eine Grouped-Query Attention, die den Inferenzprozess im Modell beschleunigt. Es ist eine modifizierte Version des normalen Multi-Head Attention Layers mit Causal Self-Attention.

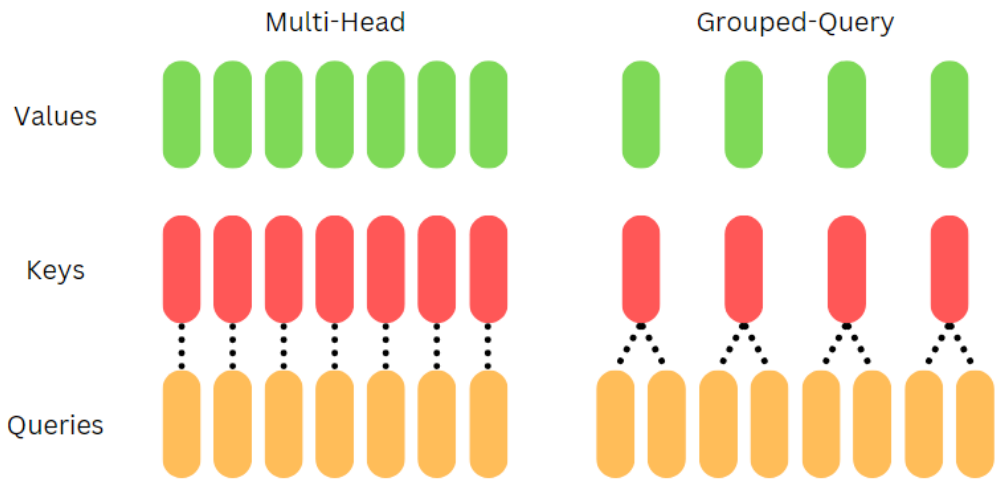

Abbildung 39: Vergleich Multi-Head- und Grouped-Query Attention

Bei der Grouped-Query Attention werden die N Attention-Köpfe in Gruppen unterteilt, wobei die Values und Keys innerhalb jeder Gruppe gemeinsam genutzt werden. Die nächste Stufe wäre eine Multi-Query Attention, die eine einzige Projektion von Values und Keys für alle N Köpfe verwendet.[27] Die Grouped-Query ist somit ein Zwischenmodell, welches die Leistung der normalen Multi-Head Attention beibehält und eine vergleichbare Effizienz zur Multi-Query Attention hat.

[26] Vgl. Kianyew, Ngieng (2024)
[27] Vgl. Wolfe, Cameron R. (2023)

4.2.1.2. Trainingsdatensatz

Das Grundmodell Llama 2 wurde auf einen Mix aus öffentlich zugänglichen Online-Daten trainiert, die insgesamt eine Größe von zwei Billionen Token haben. Das Code Llama Instruction Modell wurde auf diesem Basismodell weiter trainiert und gefinetuned.

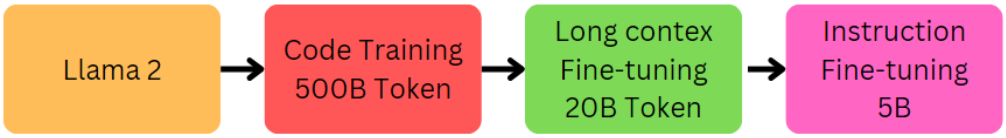

Abbildung 40: Code Llama Instruction Pipeline[28]

Dabei kommen zu den ursprünglichen zwei Billionen Token insgesamt weitere 525 Milliarden Token hinzu. Die 500 Milliarden Token aus dem ersten Code Training bestehen dabei aus folgender Aufschlüsselung:

Art der Daten	Anteil	Epochen	Größe
Code	85%	2,03	859 GB
Natürliche Sprache in Bezug auf Code	8%	1,39	78 GB
Natürliche Sprache	7%	0,01	3,5 TB

Tabelle 2: Anteile Datensatz Code Training[29]

Zum Schluss der Pipeline wird das Modell auf fünf Milliarden Token mit Instruktionen gefinetuned. Diese haben die Aufgabe, dass das Modell besser menschlichen Instruktionen folgen kann.

[28] Vgl. Roziere, Baptiste u.a. (o.J.)
[29] Vgl. Roziere, Baptiste u.a. (o.J.)

4.2.2. Vorbereitung Datensatz

Anders als im nicht vortrainierten Transformer wird in der Huggingface Bibliothek mit Datensätzen gearbeitet. Diese müssen zunächst ebenfalls erstmal vorbereitet werden. Dazu wird ein bereits in der Huggingface Bibliothek existierender Datensatz mit 62.500 Einträgen, bestehend aus einer englischen Instruktion und dem dazu passenden Code, verwendet.

```
raw_dataset = load_dataset(DATASET_NAME)
```

Abbildung 41: Datensatz von Huggingface laden

Nach dem Herunterladen enthält der Datensatz die Instruktionen und den Output. Dieser muss im nächsten Schritt zuerst in das richtige Format gebracht werden. Da das Code Llama Modell ein autoregressiver Transformer ist, dessen Aufgabe es ist, den nächsten Token vorherzusagen, werden die Instruktionen und der Output nicht auf Input_ids und Labels aufgeteilt, sondern in einer Kombination gemeinsam in den Input_ids übergeben. Dazu erhält der Input folgende Formatierung:

“““{PROMPT_TEMPLATE}

Problem: {instruction}
[/INST]
[GDSCRIPT]
{output}
[/GDSCRIPT]”””

Das PROMPT_TEMPLATE ist dabei wie eine Anweisung an die KI zu verstehen, die die generelle Aufgabe genau schildert und der KI erklärt, dass der zu generierende Code zwischen den eckigen GDSCRIPT-Tags stehen und in

welcher Programmiersprache der Code generiert werden soll. Daraufhin folgt die eigentliche Instruktion in Kombination mit dem Code.[30]

Nachdem der Input formatiert wurde, muss er im nächsten Schritt tokenisiert werden. Dazu wird für jedes Sample im Datensatz nun eine Funktion aufgerufen, die den Text mit dem Tokenizer tokenisiert. Dazu wird der Text an den Tokenizer übergeben, der ebenfalls eine Padding-Maske erstellt und die Sequenzen von vorne aus auf die maximale Länge mit Pad-Token auffüllt. Da es keinen richtigen Output mehr gibt, werden die Labels auf die Input_ids gesetzt, da das Modell den nächsten Token vorhersagen soll. Intern werden die Labels dazu um einen geshiftet, wodurch zu dem aktuellen Wort der Output das eigentlich nächste Wort in der Sequenz ist.

```
def tokenize_function(example, tokenizer):
    inputs = tokenizer(
        example['text'],
        padding=PADDING_TYPE,
        truncation=TRUNCATION,
        max_length=MAX_SEQ_LENGTH,
        return_tensors=RETURN_TENSORS,
    )
    inputs['labels'] = inputs['input_ids'].clone()
    return inputs
```

Abbildung 42: Tokenizer Dataset Funktion

Nachdem dies für den gesamten Datensatz geschehen ist, enthält der Datensatz folgende Spalten: Text, Input_ids, Attention_mask und Labels. Die ursprünglichen Texte in natürlicher Sprache werden für den weiteren Verlauf nicht mehr benötigt, weshalb sie aus dem Datensatz gelöscht werden. Im nächsten Schritt muss der Datensatz in einen Trainings- und einen

[30] Vgl. Roziere, Baptiste u.a. (o.J.)

Validierungsdatensatz aufgeteilt werden. Dazu wird automatisiert ein Training- und Test-Split erstellt und anschließend in einen kombinierten Datensatz umgewandelt.

```
df = tokenized_datasets.to_pandas()

train_df, val_df = train_test_split(df,
test_size=TEST_SIZE, random_state=RANDOM_STATE)

data = DatasetDict({
    "train": Dataset.from_pandas(train_df),
    "val": Dataset.from_pandas(val_df)
})
```

Abbildung 43: Aufteilung in Trainings- und Validierungsdatensatz

4.2.3. Finetuning des Modells

Um das schon vorhandene Modell auf den Datensatz zu verfeinern, wird ein Trainer benötigt. Dieser erhält einmal das Modell, alle Parameter für das weitere Training und die beiden Datensätze für das Training und die Evaluierung. Außerdem wird mit der Peft-Bibliothek ein Kopf auf das vorhandene Modell gesetzt, damit dieses überhaupt gefinetuned werden kann.[31]

```
trainer = Trainer(
    model=model,
    args=training_args,
    tokenizer=tokenizer,
    train_dataset=data["train"],
    eval_dataset=data["val"],
)
```

Abbildung 44: Erstellung des Trainers

[31] Vgl. HuggingFace (o.J.) Load adapters with HuggingFace PEFT

Im Anschluss wird das Modell trainiert und im Folgenden abgespeichert, damit es im weiteren Verlauf wiederverwendet werden kann.

```
trainer.train()
save_model_and_tokenizer(trainer, tokenizer)

def save_model_and_tokenizer(trainer, tokenizer):
    trainer.save_model(MODEL_SAVE_PATH)
    print(f"Model saved to {MODEL_SAVE_PATH}")
    tokenizer.save_pretrained(TOKENIZER_SAVE_PATH)
    print(f"Tokenizer saved to
              {TOKENIZER_SAVE_PATH}")
```

Abbildung 45: Fine Tuning und Abspeichern des Modells und Tokenizers

4.2.4. Trainingszeiten

Aufgrund fehlender Serverkapazität wurde statt Google Cloud der Service von Fluidstack in Anspruch genommen. Mit Fluidstack können innerhalb von Minuten GPU Instanzen und GPU Cluster erstellt werden. Für das Finetuning der KI wurde eine A100 GPU mit 80GB vRAM verwendet. Das Modell wurde insgesamt eine Epoche lang trainiert und die Trainingszeit lag bei 3 ½ Stunden inklusive Validierung.

4.2.5. Fazit vortrainiertes Modell

Das vortrainierte Modell gibt nach dem Finetuning auf den Godot Datensatz gute und verwertbare Ergebnisse aus. Es hält dabei die vorgegebene Struktur ein und gibt den Code in dem dafür vorgesehenen Platz aus. Der Code ist dabei syntaktisch korrekt und kann ohne weiteres in ein Godot Modul eingesetzt werden.

5. Serverkommunikation

Da das Addon so einfach wie möglich gehalten werden sollte, wird die KI auf einen Server ausgelagert. Zudem benötigte das Generieren der Antwort ebenfalls eine große Menge an Leistung. Dieses würde auf vielen PCs dazu führen, dass das Addon eine sehr lange Antwortzeit hätte.

5.1. Godot Addon

Die Funktionalität einer HTTP-Request wird in Godot direkt über das eigene „HTTPRequest“ Element geregelt. Dabei wird diese zuerst instanziiert und im Folgenden dem Szenen Baum als Kind Objekt hinzugefügt. Danach wird festgelegt, welche Funktion aufgerufen werden soll, sofern die Request abgeschlossen ist. Zusätzlich wird der eigentliche Prompt (hier „request“) vorbereitet und dann abgeschickt.

```
var http_request = HTTPRequest.new()
add_child(http_request)
http_request.request_completed.connect(
      _on_request_completed)
var body = JSON.stringify(request)
http_request.request(url, [], HTTPClient.METHOD_POST,
      body)
```

Abbildung 46: Godot Request erstellen[32]

Die Funktion behandelt die Antwort des Servers auf die Anfrage. Dabei wird zuerst geprüft, ob es einen serverseitigen Fehler gab. Sollte es serverseitig keinen Fehler geben, dann wird die Antwort des Servers dem Nutzer direkt angezeigt. Sollte ein Fehler entstanden sein, dann wird dem Nutzer dies ebenfalls mitgeteilt.

[32] Vgl. Godot community (2024)

```
func _on_request_completed(result, response_code,
headers, body):
    if response_code == 200:
        var response = body.get_string_from_utf8()
        curr_answer._set_text(response, true)
        curr_answer._set_time()
    else:
        curr_answer._set_text("An error accured during
              your request!", false)
        curr_answer._set_error_color()
```

Abbildung 47: Godot Request Completed Funktion

Zusätzlich zu dem eigentlichen Text in der Antwort wird dem Nutzer ein Fehler auch visuell durch die Veränderung der Farbe in der Antwort dargestellt.

Abbildung 48: Anzeige Fehler bei Request

5.2. Server

Das Skript für den Server ist darauf ausgelegt, dass es lediglich gestartet werden muss und daraufhin alle Post Requests verarbeitet. Im ersten Schritt wird das Modell und der Tokenizer geladen, damit dies nicht bei jeder Anfrage an den Server erneut gemacht wird. Danach wird der HTTP-Server gestartet, welcher kontinuierlich läuft und somit auf alle eingehenden Anfragen hört.

```
def run(server_class=HTTPServer,
            handler_class=handler, port=8000):
    load_model_and_tokenizer()
    print(f'Successfully loaded the model and
```

```
            tokenizer...')
    server_address = ('', port)
    httpd = server_class(server_address,
            handler_class)
    print(f'Starting http on port {port}...')
    httpd.serve_forever()
```

Abbildung 49: Modell laden und HTTP-Server erstellen[33]

Der „Handler“ ist dabei dafür verantwortlich, dass die Post Abfrage bearbeitet wird. Dafür wird zunächst die Nutzlast der Anfrage ausgelesen. Diese wird zuerst in dem gleichen Format wie beim Training vorbereitet, danach tokenisiert und der KI als Input gegeben, um ein Ergebnis vorherzusagen. Das Resultat der Vorhersage wird wieder enttokenisiert. Da der Output der KI nach der Generierung immer noch die Formatierung des Inputs hat, wird lediglich der Code zwischen den beiden Tags extrahiert und daraufhin an den Client zurückgesendet.

```
class handler(BaseHTTPRequestHandler):
    def do_POST(self):
        self.send_response(200, "OK")
        content_length = int(self.headers['Content-
            Length'])
        post_data = self.rfile.read(content_length)
        json_data = json.loads(post_data.decode('utf-
            8'))
        formatted_prompt = create_formatted_prompt(
            json_data)
        inputs = tokenizer(formatted_prompt,
            return_tensors="pt")
        inputs = {key: value.to(device) for key,
```

[33] Vgl. Python Software Foundation (o.J.)

```
            value in inputs.items()}
        sample = model.generate(**inputs,
            max_length=512)
        model_output = tokenizer.decode(sample[0],
            skip_special_tokens=True)
        formatted_output = extract_gdscript_content(
            model_output)
        response_data = formatted_output
        self.send_header('Content-Length',
            str(len(response_data)))
        self.end_headers()
        self.wfile.write(response_data.encode('utf-
            8'))
```

Abbildung 50: Request Handler

6. Evaluation

Das erste nicht vortrainierte Transformer Modell gibt nach dem Training keine verwertbaren Ergebnisse zurück, weshalb es in der Praxisanwendung des Addons nicht benutzt werden kann und es keinerlei Sinn ergibt, das Modell in dieser Form weiterzuentwickeln. Die vortrainierte KI hingegen liefert verwertbare Ergebnisse und ist damit in Verbindung mit dem Addon eine Hilfe im Programmieralltag. Dazu sei gesagt, dass dieses Tool ein Tool für Programmierer ist, die bereits Erfahrung im Bereich der Programmierung haben und die grundlegende Syntax und den Aufbau von Code verstehen. Die KI gibt mit Absicht keinerlei natürliche Sprache als Erklärung zu dem Code aus. Das Tool kann dabei helfen, wenn der Nutzer keinen Ansatz für ein Problem hat oder er eine einfache Funktion nicht selbst schreiben möchte. Für beide Fälle muss eine eindeutig formulierte Anweisung an die KI gestellt werden. Die KI ist nicht dafür ausgelegt, wie in einem Chat Fragen zu beantworten, sondern erstellt Code zu einem vorhandenen Problem. Im Folgenden sind ein paar Beispiele der

Generierungsfähigkeiten der KI dargestellt, anhand der beurteilt werden kann, dass die KI sowohl einfache, als auch komplexere Probleme lösen kann:

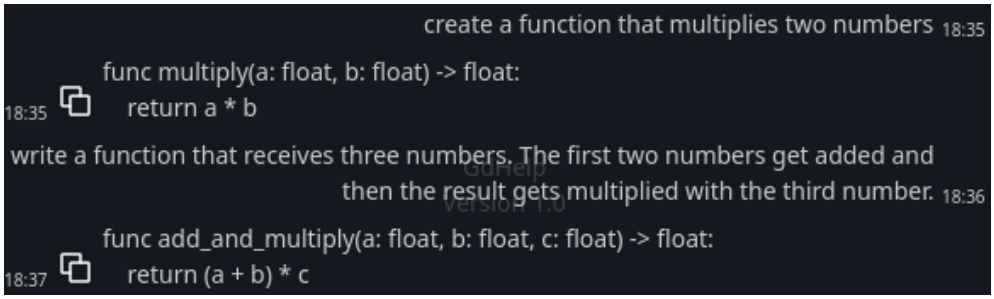

Abbildung 51: Generierung von einfachen mathematischen Funktionen

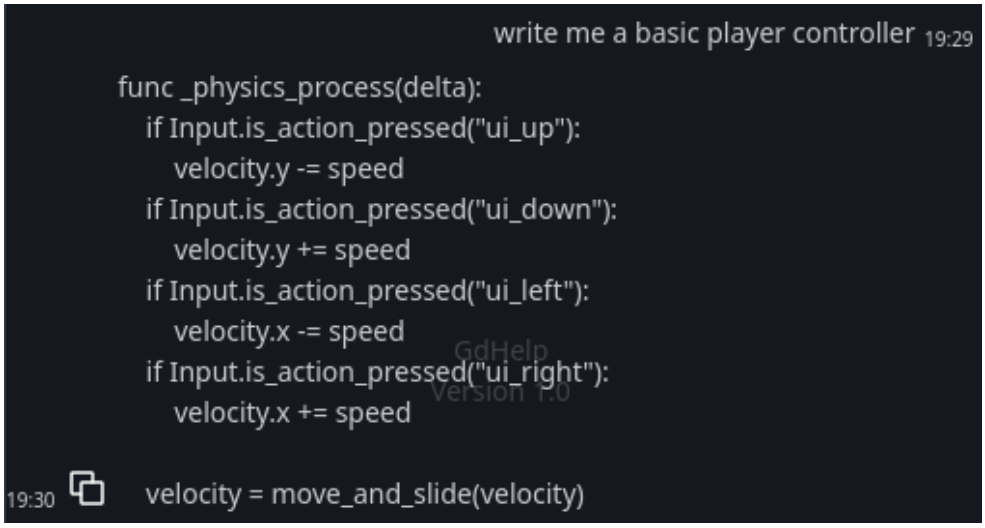

Abbildung 52: Generierung eines Player Controllers

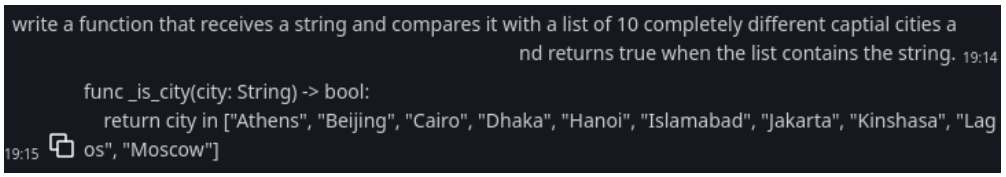

Abbildung 53: Generierung "Contains" Funktion

7. Schlussfolgerung und Ausblick

Das Addon erfüllt die Anforderungen, da es anhand einer englischen Instruktion GDScript Code generieren kann. Die Forschungsfragen für die Arbeit konnten in umfangreicher Form beantwortet werden. Generell gilt bei dieser KI, je mehr Daten sie im Training zur Verfügung hat, desto besser wird sie am Ende performen. Das Finetuning auf einen Datensatz von 62.500 fügt der KI weitere Informationen hinzu, die nur auf das Finetuning von Instruktionen zu Godot ausgelegt sind. Eine Erweiterung dieses Datensatzes würde eine noch bessere Performance und eine breitere Aufstellung bedeuten. Die aktuelle Performance und die Integration in der Godot Engine haben einen Nutzen für alle Programmierer, die in der Godot Engine arbeiten und gerne auf Code Synthesis zurückgreifen. Die KI kann nicht nur Ideen für eine Umsetzung im Code geben, sondern vor allem lästige kleinere Programmierarbeiten übernehmen. Der Programmierer kann sich dadurch die Zeit sparen, einfache Funktionen zu schreiben. Aufgrund des Knopfes an der KI-Antwort, kann der generierte Code in Sekundenschnelle ohne neues Formatieren in den bereits bestehenden Code übernommen werden.

Das Tool könnte insofern noch weiterentwickelt werden, indem ein noch größeres Datenset erstellt wird, auf das das Modell trainiert wird. Dazu könnten öffentliche Quellen für GDScript Code benutzt werden und mit Instruktionen versehen werden. Des Weiteren könnte das Modell ebenfalls darauf ausgelegt werden, auf bereits gegebene Antworten zurückzugreifen oder Code vom Benutzer mit in die Generierung einzubeziehen. Alle diese Features sind allerdings mit einem größeren Zeit- und Rechenaufwand und Kosten verbunden.

Literaturverzeichnis

Azhar (2024), Rotary Positional Embeddings: A Detailed Look and Comprehensive Understanding, https://medium.com/ai-insights-cobet/rotary-positional-embeddings-a-detailed-look-and-comprehensive-understanding-4ff66a874d83 (Abruf: 01.07.2024).

DataScientest (2024), Das Transformer neural Network, eine Revolution im NLP-Bereich, https://datascientest.com/de/das-transformer-neural-network-eine-revolution-im-nlp-bereich (Abruf: 19.06.2024).

Deepgram (2023), HumanEval: Decoding the LLM benchmark for Code Generation, https://deepgram.com/learn/humaneval-llm-benchmark (Abruf: 19.06.2024).

Devansh (2023), What are the Different Types of Transformers in AI, https://machine-learning-made-simple.medium.com/what-are-the-different-types-of-transformers-in-ai-5085275664e8#:~:text=Autoregressive%20models%2C%20popularized%20by%20GPT,decoder%20component%20of%20the%20transformer (Abruf: 25.06.2024).

Doshi, Ketan (2021), Transformers Explained Visually (Part 3): Multi-head Attention, deep dive, https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853 (Abruf: 25.06.2024).

Godot Foundation (2024), License, https://godotengine.org/license/ (Abruf: 11.06.2024).

Godot community (2014), Einführung, https://docs.godotengine.org/de/4.x/about/introduction.html (Abruf: 11.06.2024).

Godot community (2024), HTTPRequest, https://docs.godotengine.org/en/stable/classes/class_httprequest.html (Abruf: 26.06.2024).

HuggingFace (o.J.), Code Llama, https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf (Abruf: 26.06.2024).

HuggingFace (o.J.), godot-code, https://huggingface.co/datasets/dotfantasy/godot-code (Abruf: 26.06.2024).

HuggingFace (o.J.), Load adapters with HuggingFace PEFT, https://huggingface.co/docs/transformers/peft (Abruf: 27.06.2024).

Iconoir (o.J.), Icon Library, https://iconoir.com/ (Abruf: 01.06.2024).

IT-P GmbH (o.J.), Word Embedding, https://www.it-p.de/lexikon/word-embedding/#:~:text=Word%20Embedding%2C%20auch%20als%20Wortvektoren,semantische%20Zusammenh%C3%A4nge%20zwischen%20W%C3%B6rtern%20abbilden (Abruf: 18.06.2024).

Keras (o.J.), Dense Layer, https://keras.io/api/layers/core_layers/dense/ (Abruf: 25.06.2024).

Keras (o.J.), Dropout Layer, https://keras.io/api/layers/regularization_layers/dropout/ (Abruf: 25.06.2024).

Keras (o.J.), Embedding layer, https://keras.io/api/layers/core_layers/embedding/ (Abruf: 25.06.2024).

Keras (o.J), MultiHeadAttention layer, https://keras.io/api/layers/attention_layers/multi_head_attention/ (Abruf: 25.06.2024).

Kianyew, Ngieng (2024), Understanding Rotary Positional Encoding, https://medium.com/@ngiengkianyew/understanding-rotary-positional-encoding-40635a4d078e (Abruf: 01.07.2024)

Krüger, Ralph (2021), Die Transformer-Architektur für Systeme zur neuronalen maschinellen Übersetzung – eine popularisierende Darstellung in: *trans-com*, 14, S. 278 – 324, https://www.trans-kom.eu/bd14nr02/trans-kom_14_02_05_Krueger_NMUe.20211202.pdf (Abruf: 10.06.2024).

Manning, Christopher (2020), Artificial Intelligence Definitions, Standford University, https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf (Abruf: 10.06.2024).

McCarthy, J. u.a. (1955), A PROPOSAL FOR THE DARTHMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE, https://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html (Abruf: 10.06.2024).

Meta AI (o.J.), MBPP (Mostly Basic Python Programming), https://paperswithcode.com/dataset/mbpp (Abruf: 27.06.2024).

Minosvasilias (o.J.), Godot-Dodo, https://github.com/minosvasilias/godot-dodo (Abruf: 05.06.2024).

Python Software Foundation (o.J.), HTTP servers, https://docs.python.org/3/library/http.server.html (Abruf 20.06.2024).

Roziere, Baptiste u.a. (o.J.), Code Llama: Open Foundation Models for Code, https://scontent-ham3-1.xx.fbcdn.net/v/t39.2365-6/440937359_1249838219330505_1104237120116944930_n.pdf?_nc_cat=109&ccb=1-7&_nc_sid=3c67a6&_nc_ohc=Ji2b2Pyfs8cQ7kNvgEIPVB-&_nc_ht=scontent-ham3-1.xx&oh=00_AYASOM2XAQZhDPS-zWmssfhEkJATn6auMi0eAPEejoARkQ&oe=668739E0 (Abruf: 01.07.2024).

Su, Jianlin (2023), ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING, https://arxiv.org/pdf/2104.09864 (Abruf: 22.06.2024).

TensorFlow (2024), Neural machine translation with a Transformer and Keras, https://www.tensorflow.org/text/tutorials/transformer#the_embedding_and_positional_encoding_layer (Abruf: 13.06.2024).

TensorFlow (2024), tf.keras.layers.LayerNormalization, https://www.tensorflow.org/api_docs/python/tf/keras/layers/LayerNormalization (Abruf: 13.06.2024).

Touvron, Hugo (o.J.), Llama 2: Open Foundation and Fine-Tuned Chat Models, https://scontent-ham3-1.xx.fbcdn.net/v/t39.2365-6/10000000_662098952474184_2584067087619170692_n.pdf?_nc_cat=105&ccb=1-7&_nc_sid=3c67a6&_nc_ohc=9iYQ0hxrQFkQ7kNvgFOAz-i&_nc_ht=scontent-ham3-1.xx&oh=00_AYAoN-GqClVPTXKJTJAPxUPPvcab1xsVqxCwtXIW-bmO_w&oe=668B2B3F (Abruf: 02.07.2024).

Vaswani, Ashsish u.a. (2023), Attention Is All You Need, https://arxiv.org/pdf/1706.03762 (Abruf: 05.06.2024).

Wawrzko, Michal (2024), Learn X in Y Minutes Where X=GDScript, https://learnxinyminutes.com/docs/gdscript/#:~:text=GDScript%20is%20a%20dynamically%20and,perfect%20fit%20for%20game%20development (Abruf: 18.06.2024).

Wolfe, Cameron R. (2023), LLaMa-2 from the Ground Up, https://cameronrwolfe.substack.com/p/llama-2-from-the-ground-up (Abruf: 02.07.2024).

Zhu, Scott und Cholett, Francois (2023), Understanding masking & padding, https://www.tensorflow.org/guide/keras/understanding_masking_and_padding (Abruf: 26.06.2024).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sowie durch eine künstliche Intelligenz wie ChatGPT erstellte oder bearbeitete Inhalte sind als solche kenntlich gemacht.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Torben Oschkinat
Hamburg, den 11. Juli 2024