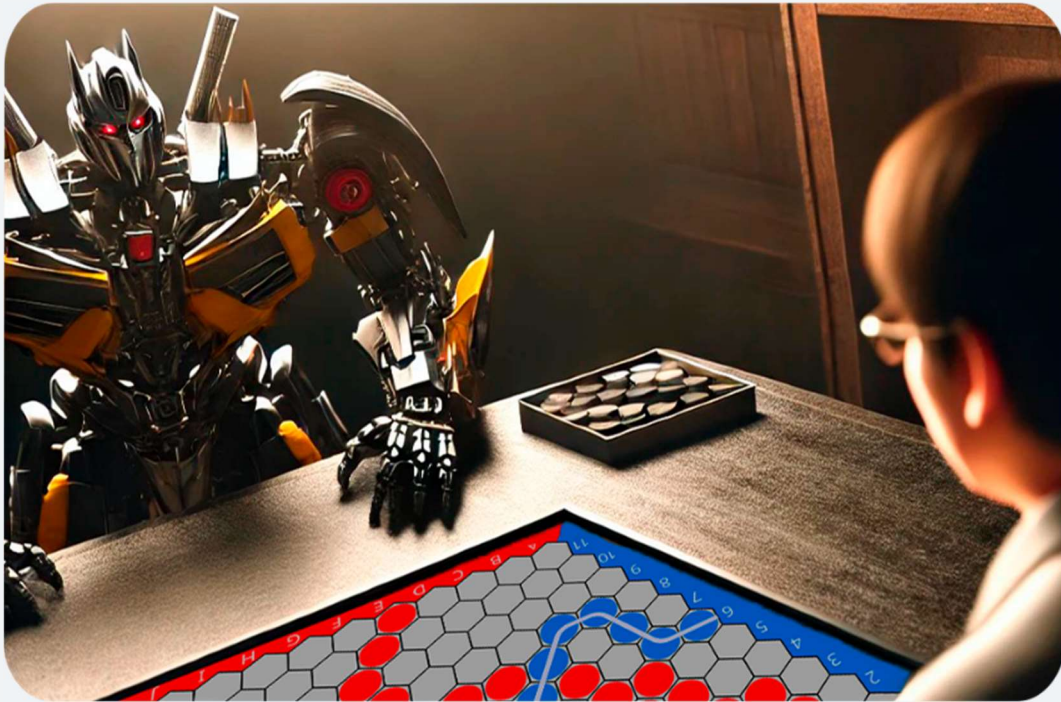


Applying Transformer Models to the Game of Hex

Einsatz von NLP-Techniken



Maturaarbeit von: Tymur Haivoronskyi
Kantonsschule Wattwil, Klasse 2021P
Speerstrasse 1, 8717, Benken SG

Betreuungsperson: Emil Müller, Jakub Adamek

Eingereicht im: Oktober 2024

„Wettbewerbsarbeit für Schweizer Jugend forscht“

«Jede hinreichend fortschrittliche Technologie ist
von Magie nicht zu unterscheiden.»

- Arthur Clarke

Inhaltsverzeichnis

1	Vorwort.....	4
2	Hintergrund	5
3	Einleitung und Problemstellung	6
4	Brettspiel Hex.....	7
4.1	Erfindung	7
4.2	Regeln	7
4.3	Warum Hex?.....	8
4.3.1	Das Spiel in eine Sequenz verwandeln	9
4.3.2	Kein Unentschieden	10
4.3.3	Der ideale Spieler.....	10
4.4	Dark Hex	11
5	Architekturen.....	13
5.1	Transformer Architektur.....	13
5.1.1	Tokenisierung.....	13
5.1.2	Input Embedding.....	15
5.1.3	Positional Encoding	16
5.1.4	Single Head Self-Attention	18
5.1.5	Multi-Head Attention	19
5.1.6	Add and Normalize	21
5.1.7	Feed Forward	22
5.1.8	Multi-Head Cross Attention	24
5.1.9	Masked Multi-Head Attention	24
5.1.10	Linear	25
5.1.11	Softmax	25
5.1.12	Trainingsverlauf	26
5.2	Aufbau von GPT	28
6	Projektumsetzung.....	30

6.1	Datenerfassung.....	30
6.2	Erstellung eines Tokenisers	32
6.3	Erstellung einer Trainingsdatei	32
6.4	Aufteilung in zwei Spieler	34
6.5	Der erste Versuch	34
6.6	Suche nach der optimalen Modellgrösse	35
6.7	Modellleistungstest	36
6.8	Das Konzept der Generationen	37
6.9	Misserfolg und Analyse	38
6.10	Fine-tuning an den ursprünglichen Daten	39
6.11	GUI zum Spielen von Hex	40
7	Ergebnisse	40
8	Weiterentwicklung.....	42
9	Diskussion	44
10	Schlusswort	45
11	Inhaltsverzeichnis.....	46
12	Abbildungsverzeichnis	49
13	Selbständigkeitserklärung	50
14	Anhang	51

1 Vorwort

Meine erste Bekanntschaft mit neuronalen Netzen machte ich im Alter von etwa 13 Jahren, als ich auf ein YouTube-Video stiess, in dem die Konstruktion des einfachsten Konzepts im Bereich der neuronalen Netze, des **Perzeptrons**, erklärt wurde. Dieses Konzept erschien mir damals wie eine verrückte Idee, die für meinen Verstand zu schwer zu begreifen war. Zunächst hörte ich immer häufiger vom Begriff „neuronale Netze« [2]. Die Ergebnisse dieser Netze verblüfften mich, aber ich hatte keine Ahnung, wie sie funktionierten. Damals begann ich gerade, darüber nachzudenken, welches Fachgebiet ich nach der Schule studieren wollte, und natürlich kam ich auf die Idee, maschinelles Lernen in Betracht zu ziehen. Im Laufe der Zeit ist dieser Bereich unbewusst ganz oben auf meiner Prioritätenliste gelandet und steht dort weiterhin. Diese Arbeit ist für mich der perfekte Zeitpunkt, um mit meiner Einführung in dieses Gebiet zu beginnen und zu erkennen, wie interessant es ist, nicht nur die Ergebnisse zu sehen, sondern direkt an der Erstellung dieser Ergebnisse beteiligt zu sein. Dank der Diskussionen mit meiner Betreuungsperson, Emil Müller, und seiner Ideen kamen wir auf das Thema dieser Maturaarbeit, nämlich neue Mechanismen des maschinellen Lernens zu nutzen, um einem Agenten das Spielen eines Brettspiels beizubringen. Diese Idee gefällt mir, und diese Arbeit dient als Demonstration der Leistungsfähigkeit von generativen LLMs (Large Language Models) bei der Vorhersage von Zügen in einem taktischen Brettspiel.

2 Hintergrund

Im Sommer 2022 waren wir begeistert von den Bildern, die mit neuronalen Netzen erstellt wurden. Diese schienen jeden Tag besser zu werden. Bald darauf erfuhr die Welt von ChatGPT, als eine neue Version veröffentlicht wurde, die nicht nur in der Lage war, geschriebenen Text zu vervollständigen, sondern auch Fragen zu beantworten. Kurz darauf gab es einen unglaublichen Qualitätssprung in der Bildklassifizierung, Objekterkennung, Spracherkennung und Stimmenklonung. Die letztgenannte Technologie konnte sowohl in unterhaltsamen Videos, in denen US-Präsidenten Minecraft spielen, als auch in öffentlichkeitswirksamen Skandalen eingesetzt werden [3]. Die kürzlich angekündigte künstliche Intelligenz Sora wird in der Lage sein, Videos zu erzeugen, die von echten Aufnahmen praktisch nicht zu unterscheiden sind [4]. Seit 2021 tauchen wie ein Sturm Hunderte von Unternehmen auf, die behaupten, einen bestimmten Bereich unseres Lebens zu erneuern. Und sie tun dies recht erfolgreich, indem sie die Aufmerksamkeit von Investoren und der Öffentlichkeit auf sich ziehen. All diese scheinbar unterschiedlichen Innovationen haben ein wichtiges Detail gemeinsam: Sie basieren alle auf der Transformer-Architektur.

In einer 2017 von Google-Forschern veröffentlichten Forschungsarbeit mit dem Titel "**Attention Is All You Need**" wurde eine neue Architektur für die maschinelle Übersetzung vorgestellt, die ein neues Konzept der Multi-Head Attention nutzt, das dem Modell ermöglicht, lange Datensequenzen effizient zu verarbeiten und dabei auf die wichtigsten Teile der eingegebenen Informationen zu achten. Diese Architektur erwies sich als ausserordentlich flexibel und skalierbar. Sie bildet die Grundlage für viele moderne Sprachmodelle wie GPT (**Generative Pre-trained Transformer**) von OpenAI, BERT von Google, Llama von Meta AI, Claude von Anthropic und viele andere. [5]

Während Sprachmodelle in der Regel nur für die Verarbeitung von Sprachen verwendet werden, wurden die Prinzipien von Transformers erfolgreich für die Verarbeitung von Bildern (z. B. in Computer Vision und Diffusion Models) und sogar für die Erzeugung von Musik und Sprache angepasst, was diese Architektur ziemlich einzigartig und anpassungsfähig macht.

3 Einleitung und Problemstellung

Das Brettspiel HEX wird von zwei Spielern (in der Regel rot und blau) gespielt, deren Ziel es ist, die gegenüberliegenden Seiten, die zu ihrer Farbe gehören, zu verbinden. Der Spieler, der für die rote Farbe spielt, muss die obere und untere Seite verbinden, während der Spieler, der für die blaue Farbe spielt, die linke und rechte Seite verbinden muss. Zwei Seiten gelten als verbunden, wenn es eine ununterbrochene Kette gibt, in der die Seiten miteinander und auch mit den gegenüberliegenden Kanten verbunden sind. Da anstelle von quadratischen Spielfeldern sechseckige Felder verwendet werden (d. h. Felder, die sechs statt vier Nachbarn haben können), wird das Spiel komplizierter und weniger trivial. Die Verwendung von sechseckigen Feldern verhindert auch, dass man Züge macht, die das Spiel unentschieden enden lassen, was bei einem Spiel mit quadratischen Feldern leicht vorstellbar wäre.

Diese Arbeit konzentriert sich auf die Fähigkeit von Transformern, Textdaten als Sequenz zu verarbeiten, und zielt darauf ab, ein Modell zu trainieren, das in der Lage ist, ein HEX-Brettspiel wie ein Mensch zu spielen. Zudem werden weitere Versuche unternommen, den Spielagenten zu verbessern, indem die vom Modell generierten Spiele sortiert werden und das Modell basierend darauf weitertrainiert wird.

Diese Arbeit kann in mehrere Phasen unterteilt werden, von denen jede eine wichtige Komponente zur Erreichung des Endziels darstellt, nämlich die Verbesserung der Qualität des Spielagenten. Natürlich gibt es keine Grenzen der Perfektion. Diese Arbeit zielt nicht darauf ab, einen perfekten Spielagenten zu entwickeln. Stattdessen wird eine alternative Methode zum Trainieren von Modellen für Spiele vorgeschlagen und untersucht, die als semantische Textsequenzen beschrieben werden können. Dabei wird vorgeschlagen, die Idee der Belohnung von Siegen und Niederlagen aufzugeben, wie es beim herkömmlichen Reinforcement Learning der Fall ist, und stattdessen zwei Modelle für jeden einzelnen Spieler zu verwenden und ein Modell einer bestimmten Farbe nur auf der Grundlage gewonnener Spiele weiter zu verbessern. Diese Methode steht im Gegensatz zum MCTS-Lernen, mit dem in den letzten Jahren erfolgreich hochmoderne Spielagenten wie Alpha Zero, AlphaGo Zero, KataHex usw. entwickelt wurden. Das beste Anzeichen dafür, dass ein Modell nach einem erneuten Training besser funktioniert, ist ein direkter Vergleich des Modells mit früheren Versionen in einem Spiel gegen einen Gegner, der das Spiel gut kennt (mit anderen Worten, gegen ein anderes Modell, das für die andere Farbe spielt), wobei wir uns für das Verhältnis von Siegen zu Verlusten und die durchschnittliche Spieldauer interessieren können. Je kürzer das Spiel, desto mehr überlegte Züge macht das Modell, wodurch das Spiel schneller beendet wird. Eine detaillierte Beschreibung der Schritte befindet sich im Abschnitt **Projektumsetzung**.

4 Brettspiel Hex

Auch wenn das Training des Modells kein Verständnis des Spiels voraussetzt, so ist es doch recht nützlich, sich mit den Grundprinzipien des Spiels und den Gewinnmechanismen vertraut zu machen, um die Ergebnisse des Modells aus menschlicher Sicht bewerten zu können. Zu diesem Zweck bietet sich ein Exkurs in die Geschichte des Spiels an, sowie ein kleiner theoretischer Teil zum Brettspiel.

4.1 Erfindung

Die Geschichte des Brettspiels Hex begann in den 1940er Jahren, als der dänische Mathematiker und Ingenieur Piet Hein das Spiel erstmals entwarf. Hein war bekannt für seine vielseitigen Beiträge zu Mathematik, Kunst und Design. 1942 stellte er das Spiel unter dem Namen „Polygon“ in einer dänischen Zeitung vor. Das Spiel wurde ursprünglich als Buch mit 50 Seiten vertrieben, von denen jede ein gedrucktes Feld hatte, und man ging davon aus, dass das Feld einmalig war und die Spieler ihre Züge mit einem Bleistift oder Kugelschreiber markieren mussten. Es ist anzunehmen, dass das Spiel ursprünglich als eine komplexere und strategisch anspruchsvollere Version von Tic-Tac-Toe gedacht war. Das Spiel breitete sich nur wenig aus, was sechs Jahre später, 1948, dazu führte, dass der amerikanische Mathematiker John Nash das Spiel wiederentdeckte und später behauptete, dass die Entdeckung unabhängig war. Diese Tatsache ist umstritten, denn es ist bekannt, dass einige von John Nashs dänische Kollegen mit dem von Piet Hein vorgestellten Spiel vertraut waren. Trotz akademischer Debatten über die Urheberschaft fand das Spiel einen kommerziellen Markt. So brachte Parker Brothers, die Firma, die für die Herstellung von Monopoly bekannt ist, 1952 die erste Version des Brettspiels mit einem Brett und Chips unter dem uns heute bekannten Namen **Hex** heraus. [6], [7]

4.2 Regeln

Im Vergleich zu Schach und sogar Go, mit dem Hex oft verglichen wird, erfordert dieses Brettspiel nicht viel Wissen, bevor man mit dem Spielen beginnen kann. Wenn man die Tauschregel ausser Acht lässt, lässt sich das Spiel leicht in wenigen Sätzen erklären. Aber um Missverständnisse zu vermeiden, ist es am besten, das Spiel mit der Regel zu verstehen.

Die Spieler müssen sich für die Farbe der Chips entscheiden, mit denen sie spielen wollen. Die Wahl besteht zwischen blau und rot, und derjenige, der rote Chips gewählt hat, darf den ersten Zug machen. Ein Zug besteht darin, einen eigenen Chip auf ein freies Feld zu setzen. Ein einmal besetztes Feld kann nicht ein zweites Mal von einem Gegner oder einem Spieler besetzt werden, d.h. es bleibt stehen. Ziel des Spiels ist es, eine zusammenhängende Kette von Spielsteinen zu bilden, die

die gegenüberliegenden Seiten des Spielbretts, die mit der Farbe des Spielers gekennzeichnet sind, verbindet. [8] Die Mathematiker stellten schnell fest, dass der rote Spieler durch die Befolgung solcher Regeln einen grossen Vorteil hat und mit höherer Wahrscheinlichkeit gewinnt als der blaue Gegner, was sie auf die Idee brachte, eine Regel zu erfinden, die das Spiel ausgleichen würde. Diese Regel war die Swap-Regel (**Swap Rule** deren Kern darin besteht, dass der blaue Spieler dem roten Spieler den ersten Zug wegnehmen kann. Es gibt zwei Versionen der Swap-Regel in der Spielgemeinschaft: eine, die **Swap Sides** genannt wird. Nach dem ersten Zug kann der zweite Spieler entscheiden, die Seite zu wechseln, so dass im Wesentlichen der blaue Spieler der rote Spieler wird. Abbildung 2 zeigt zur Verdeutlichung die Seitenwechselregel, ohne dass die Spieler physisch die Farben wechseln, in Wirklichkeit tauschen die beiden Spieler einfach die Chips. Diese Methode ist recht einfach und wird häufig in physischen Brettspielen verwendet, aber es gibt auch eine zweite Methode, **Swap Pieces**, bei der Blau den Chip des roten Spielers entfernen und seinen Chip auf ein Feld legen kann, das den roten Chip in Bezug auf die lange Diagonale des Spielbretts spiegelt. Die Swap Regel ist also der einzige Ausnahmefall, in dem es möglich ist, eine Figur zu entfernen. [9] In dieser Forschungsarbeit wird aus mehreren Gründen die zweite Regel verwendet. Mit diesen Regeln beträgt die minimale Spieldauer 21 Züge und die maximale Spieldauer 122 Züge (obwohl das Brett nur 121 Felder umfasst, zählt der Tauschregelzug für zwei Züge).

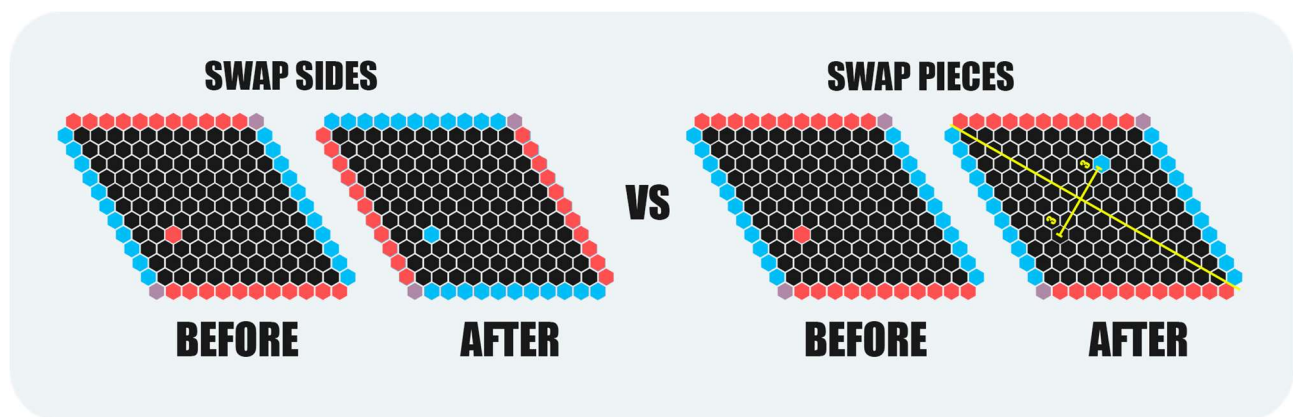


Abbildung 2: Vergleich von zwei Swap Regeln (eigene Darstellung)

4.3 Warum Hex?

Hex verfügt über einen enorm grossen Raum möglicher Züge, was die Suche nach optimalen Strategien zu einer komplexen Aufgabe macht und effiziente Algorithmen zur Bewertung von Positionen erfordert. Insgesamt könnte bei der Auswahl eines Brettspiels jede vollständige Informationsspiel wie Schach oder Go in Betracht gezogen werden. Doch im Gegensatz zu den genannten Spielen ist Hex im Kontext der KI weniger erforscht und gleichzeitig in mathematischen Kreisen sehr beliebt, da es eine reiche mathematische Grundlage besitzt, einschliesslich des Beweises des Hex-Theorems. Zudem sind die Regeln dieses Spiels sehr einfach und erfordern keine lange Einführung.

Diese Informationen allein reichen aus, um das Spiel als Option in Betracht zu ziehen. Es gibt jedoch noch einige weitere wichtige Punkte, die die Entscheidung zugunsten dieses Spiels beeinflussen haben.

4.3.1 Das Spiel in eine Sequenz verwandeln

Die Verwendung der Transformer-Architektur, die im nächsten Kapitel ausführlich beschrieben wird, beinhaltet die Arbeit mit logischen Datensequenzen. Im LLM sind solche Sequenzen Texte, aber wenn man ein Spiel betrachtet, stellt sich die Frage, wie man eine Spielposition auf dem Feld als dieselbe Sequenz darstellen kann. Hex ist ein zweidimensionales Feld mit Zellen. Die Chips bewegen sich nicht und bleiben an ihrem Platz, und das Spiel ist eine Folge von Zügen. Dies macht die Aufzeichnung von Zügen zu einer sehr einfachen Aufgabe. Um die Verwechslung von x- und y-Koordinaten zu vermeiden, werden horizontale Koordinaten in der Ebene mit den Buchstaben A bis K bezeichnet (für ein Feld von 11 mal 11), während vertikale Koordinaten auf dem Spielbrett mit den Zahlen 1 bis 11 bezeichnet werden. Wir drücken also jeden platzierten Chip durch die Koordinaten aus, auf denen er platziert wurde, und erhalten einen Satz von 121 Zügen von A1 bis K11.

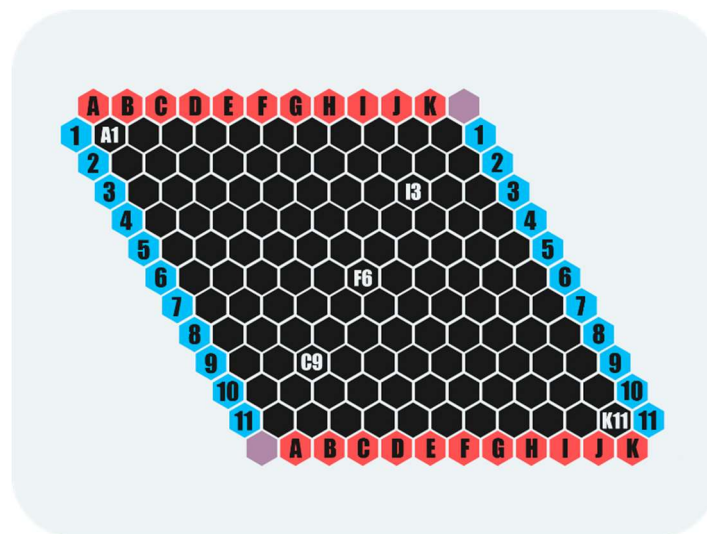


Abbildung 3: Koordinaten auf dem Brett (eigene Darstellung)

Stellen wir uns vor, der rote Spieler setzt seinen Chip auf F6, der blaue Spieler wiederum setzt seinen Chip auf I3 und im dritten Zug wählt der rote Spieler das Feld A1. Damit haben wir das Spiel F6 I3 A1 gebildet, das die Anforderungen der logischen Reihenfolge erfüllt, da es Informationen über die Reihenfolge der Züge und auch die vollständige Beschreibung der Züge enthält. Da das Spiel für zwei Spieler bestimmt ist, ist es nicht einmal notwendig, anzugeben, welcher Spieler einen Zug macht, denn alle gepaarten Züge gehören dem blauen Spieler, und alle ungepaarten Züge gehören dem roten Spieler.

Es ist nicht ganz klar, wie man die ersten beiden Züge aufzeichnet, wenn der blaue Spieler beschliesst, die Tauschregel anzuwenden. Angenommen, Rot belegt XY, wobei X die horizontale Koordinate von A bis K und Y die vertikale Koordinate von 1 bis 11 ist. In diesem Fall hat der gespiegelte Zug den Wert YX, z. B. wird das Feld C9 zu 9C, aber nach den oben genannten Regeln zur Interpretation von Zügen muss 9 in den entsprechenden Buchstaben umgewandelt werden, in diesem Fall I, und der Buchstabe C muss in die entsprechende Zahl umgewandelt werden, in diesem Fall 3, als Ergebnis haben wir den Zug I3. Es entsteht ein Problem, weil die resultierenden Züge C9 I3 mit und ohne Anwendung der Tauschregel interpretiert werden können (weil der blaue Spieler die Regel anwendet oder nicht). Um die Verwendung zusätzlicher Symbole zur Anzeige eines Tausches zu vermeiden, wird vorgeschlagen, eine ähnliche Notation C9 C9 zu verwenden, die offensichtlich anzeigt, dass es einen Zugwechsel gibt, da sonst dasselbe Feld nicht zweimal besetzt werden kann. Der grafische Interpretier, der eine solche Wiederholung der Koordinaten XY XY am Anfang sieht, sollte verstehen, dass es notwendig ist, den roten Chip vom Feld XY zu entfernen und den blauen Chip auf das Feld YX zu setzen.

4.3.2 Kein Unentschieden

Beim HEX-Spiel gibt es kein anderes Ergebnis als den Sieg eines der Spieler. Es gibt viele Möglichkeiten zu beweisen, dass das Spiel nie unentschieden endet. Die meisten dieser Beweise basieren auf der sechseckigen Geometrie des Spielfelds. Diese Tatsache ist recht intuitiv, denn wenn auf einem vollständig gefüllten Spielbrett ein Chip in einer Kette fehlt, die für uns hätte gewonnen werden können, dann hat unser Gegner diesen Chip garantiert benutzt, um den Sieg zu erringen. Der Beweis wird im Hex-Theorem angeführt, das wiederum dem Fixpunktsatz von Brouwer entspricht. Obwohl es nicht möglich ist, mit Sicherheit zu sagen, wie sich das Vorhandensein von Unentschieden auf das Training mit dem transformerbasierten Texterzeugungsmodell auswirken würde, bedeutet es beim Reinforcement Learning oft eine Vereinfachung der Belohnungsfunktion, da nur zwei mögliche Ergebnisse die Belohnungsfunktion vereinfachen. Das Modell erhält ein klares Signal darüber, ob sein Zug erfolgreich war oder nicht, was die Anpassung der Strategie erleichtert. Im Falle dieser Arbeit vereinfacht dieser Punkt die Verarbeitung und Analyse der Daten und die Bewertung der Modellleistung. [10]

4.3.3 Der ideale Spieler

Das Spiel Hex kann auf beliebig grossen Spielbrettern gespielt werden, und es gibt kein Brett, das als Standard bezeichnet werden kann. Die beliebtesten Grössen sind **11 × 11**, **13 × 13**, **15 × 15** und **19 × 19**. [11] Der grosse Verzweigungsfaktor in diesem Spiel macht es schwierig, es mit Computeralgorithmen der Brute-Force-Suche zu lösen. Bislang wurden HEX-Spielbretter bis zur Grösse **9 ×**

9 auf diese Weise vollständig gelöst, so dass das **11 × 11-Brett** ein schlecht gelöstes Problem darstellt. Bekannt ist lediglich, dass bei zwei Spielern, die mit perfekter Taktik und ohne Tauschregel spielen, der erste Spieler garantiert gewinnen wird. Diese Aussage lässt sich ganz einfach mit dem von John Nash in den 1940er Jahren vorgeschlagenen Argument des Strategiediebstahls beweisen. Das Argument stützt sich auf zwei Lemmas:

1. Das Spiel kann nicht unentschieden enden, was durch das Hex-Theorem bewiesen wird. [10]
2. Es kann dem Spieler nicht schaden, zusätzliche/zufällige Steine der eigenen Farbe auf dem Brett zu haben.

Angenommen, der **zweite Spieler** (Spieler B) hat eine Gewinnstrategie, d. h. eine Spielweise, die ihm den Sieg garantiert, unabhängig von den Aktionen des ersten Spielers. Der **erste Spieler** (Spieler A) beginnt das Spiel und macht einen **zufälligen ersten Zug**, indem er seine Figur auf ein beliebiges freies Feld des Brettes setzt. Dieser Zug kann willkürlich sein und ist nicht unbedingt mit einer Strategie verbunden. Nach dem ersten Zug von Spieler A macht Spieler B seinen Zug. Spieler A beschliesst nun, die vermeintliche Gewinnstrategie von Spieler B zu „stehlen“. Spieler A beginnt, die Strategie zu verfolgen, von der wir angenommen haben, dass Spieler B sie hat. Jedes Mal, wenn Spieler A am Zug ist, reagiert er auf den Zug von Spieler B so, wie Spieler B auf den vorherigen Zug von Spieler A reagiert hätte, wenn sie die Plätze getauscht hätten. Wenn Spieler A aufgrund seiner Strategie einen Chip auf ein bereits besetztes Feld setzen muss (einschliesslich seines zufälligen ersten Zuges), macht er einfach einen beliebigen anderen Zug, da ihm ein zusätzlicher Chip nicht schaden kann (dies ist durch die Eigenschaften des Hex-Spiels gerechtfertigt). Unter der Annahme, dass Spieler B eine Gewinnstrategie hat und Spieler A diese nach seinem zufälligen Zug anwenden kann, gewinnt Spieler A also garantiert. Dies widerspricht der ursprünglichen Annahme, dass Spieler B eine Gewinnstrategie hat. Da unsere Annahme zu diesem Widerspruch geführt hat, ist sie falsch. **Folglich kann der zweite Spieler keine Gewinnstrategie haben.** In Anwendung des No-Draw-Lemmas ist die einzig mögliche Schlussfolgerung, dass **der erste Spieler eine Gewinnstrategie hat.** [12], [13]

4.4 Dark Hex

Es gibt viele Varianten von Hex, bei denen die Form und Grösse des Spielbretts und die Spielregeln verändert werden. Eine solche Version von Hex ist Dark Hex, bei der weder die Regeln noch das Spielbrett verändert werden. Das Besondere an diesem Spiel ist die unvollständige Information über den Zustand des Brettes. Jeder der Spieler kann nur seine eigenen Züge sehen. Informationen über ein Feld, das von einem Gegner besetzt ist, erhält man nur, wenn ein Spieler versucht, dieses Feld zu besetzen. Das Spiel geht davon aus, dass es eine Art Schiedsrichter gibt, dem alle Informationen

über das Spiel zur Verfügung stehen und der in das Spiel eingreifen kann, um zu erklären, dass ein Zug nicht ausgeführt werden kann. Es besteht die Möglichkeit, dass sich diese Version des Spiels besser für das Training des transformerbasierten Modells eignet, das in **8. Diskussion** vorgestellt wird. [14]

5 Architekturen

Um den Mechanismus zu verstehen, durch den Large Language Models das nachfolgende Token erzeugen, ist zunächst ein **intuitives Verständnis** der Transformer-Architektur und des Konzepts der Multi-Head Attention erforderlich.

5.1 Transformer Architektur

In einem von Google im Jahr 2017 veröffentlichten Forschungspapier wird der Transformer aus zwei Hauptkomponenten zusammengesetzt:

1. **Encoder** (linke Seite der Architektur): wandelt die Eingangssequenz in eine latente Darstellung um.
2. **Decoder** (rechte Seite der Architektur): erzeugt eine Ausgabesequenz unter Verwendung der versteckten Encoder-Darstellung und früherer Ausgaben.

Beide Komponenten bestehen aus mehreren Schichten, von denen jede Aufmerksamkeitsmechanismen und positionsabhängige, vollständig verbundene Schichten enthält. Diese Struktur ermöglicht eine effiziente Satzübersetzung unter Berücksichtigung des Kontexts und bietet ein tieferes Verständnis des übersetzten Textes als andere maschinelle Übersetzungsmodelle. [15]

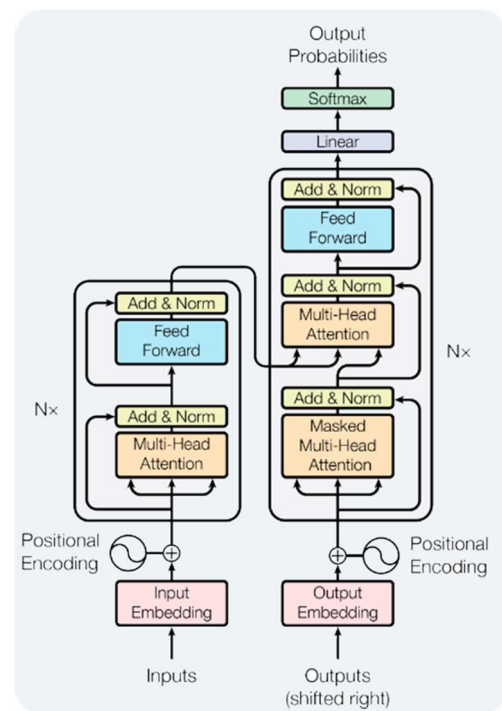


Abbildung 4: Aufbau des Transformers [16]

Der einfachste Weg, um zu verstehen, wie Transformer strukturiert ist, besteht darin, die einzelnen Schritte der Datenverarbeitung in der Transformer-Architektur durchzugehen.

5.1.1 Tokenisierung

Die Tokenisierung ist ein fundamentaler Schritt in der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP). Sie bezeichnet den Prozess, bei dem roher Text in kleinere, handhabbare Einheiten, sogenannte **Tokens**, zerlegt wird. Diese Tokens können Wörter, Subwörter oder sogar einzelne Zeichen sein. Die gewählte Methode der Tokenisierung hat einen erheblichen Einfluss auf die Leistungsfähigkeit des nachfolgenden Modells, da sie die Basis für die weitere Verarbeitung und das Verständnis des Textes bildet.

Der Tokenisierungsprozess lässt sich in mehrere aufeinanderfolgende Schritte unterteilen, die jeweils eine spezifische Funktion erfüllen. Bevor der eigentliche Zerlegungsprozess beginnt, erfolgt

eine grundlegende Vorverarbeitung des Textes. Dieser Schritt umfasst hauptsächlich die Normalisierung und optional die Entfernung von Stoppwörtern. Die **Normalisierung** beinhaltet die Umwandlung aller Zeichen in Kleinbuchstaben, das Entfernen von Sonderzeichen sowie die Vereinheitlichung von Zeichensätzen. Durch diese Massnahmen wird die Konsistenz des Textes erhöht und die Variabilität reduziert, was die nachfolgende Verarbeitung erleichtert. Optional kann auch die **Entfernung von Stoppwörtern** erfolgen. Stoppwörter sind häufig vorkommende Wörter wie "und", "oder" und "aber", die oft wenig semantische Bedeutung tragen und die Relevanz der verbleibenden Tokens erhöhen können, indem sie das Rauschen im Datensatz reduzieren.

Nach der Vorverarbeitung wird der Text in kleinere Einheiten zerlegt, die als Tokens bezeichnet werden. Es gibt verschiedene Methoden der Tokenisierung, die je nach Anwendungsfall und Modellarchitektur unterschiedlich geeignet sind:

1. **Wortbasierte Tokenisierung:** Bei dieser Methode wird der Text anhand von Leerzeichen und Satzzeichen in einzelne Wörter zerlegt. Der Satz „Das ist ein Beispiel.“ wird folgendermassen tokenisiert: "Das ist ein Beispiel." → ["Das", "ist", "ein", "Beispiel"]
2. **Subwort-Tokenisierung:** Diese Methode zerlegt seltene oder unbekannte Wörter in häufig vorkommende Subworteinheiten. Verfahren wie Byte-Pair Encoding (BPE) oder WordPiece sind hierfür gängig. Diese Methode wird am häufigsten verwendet, da sie eine optimale Tokenisierung ermöglicht, indem die Algorithmen sich adaptiv an die Trainingsdaten anpassen. Beispielsweise könnte das Wort „Beispiel“ in Subworteinheiten zerlegt werden:
"Das ist ein Beispiel." → ["Das", "ist", "ei", "n", "Bei", "spiel"]

OpenAI scheint genau diese Art von Tokenizer in ihren Modellen zu verwenden. Er kann unter <https://platform.openai.com/tokenizer> getestet werden. Hier sieht man, dass bei der Eingabe von grossen Wörtern zwei oder mehr Tokens verwendet werden, um das Wort zu definieren.

3. **Zeichenbasierte Tokenisierung:** Hierbei wird der Text in einzelne Zeichen zerlegt. Dies kann besonders nützlich sein für Sprachen mit komplexen Wortstrukturen oder für Anwendungen, die eine feinkörnige Analyse erfordern. Zum Beispiel wird das Wort „Beispiel“ wie folgt tokenisiert: "ein Beispiel" → ["e", "i", "n", " ", "B", "e", "i", "s", "p", "i", "e", "l"]

Nach der Zerlegung des Textes in Tokens wird ein **Vokabular** erstellt, das eine Liste aller einzigartigen Tokens enthält, die im Datensatz vorkommen. Insbesondere bei der Subwort-Tokenisierungsmethode werden häufige Subworteinheiten priorisiert, um die Anzahl der unbekannten Tokens (Out-of-Vocabulary, OOV) zu minimieren. Jedes Token im Vokabular wird einer eindeutigen numerischen ID zugewiesen, um eine effiziente Verarbeitung durch das Modell zu ermöglichen.

Zusätzlich zu den regulären Tokens werden **spezielle Tokens** eingeführt, die spezifische Funktionen erfüllen. Beispiele hierfür sind **[PAD]** für Padding-Zwecke, **[CLS]** für Klassifizierungsaufgaben und **[SEP]** zur Trennung von Sätzen. Diese speziellen Tokens sind essenziell für bestimmte Aufgaben und helfen dem Modell, Kontext und Struktur innerhalb der Daten zu erkennen. Ein gut strukturiertes Vokabular ist eine gute Basis für die spätere numerische Darstellung der Tokens. [17]

Im Zusammenhang mit der Arbeit mit Spielen wird es wichtig sein, den Beginn und das Ende des Spiels wie in einem Text zu markieren, damit das Modell die Positionen korrekt ansteuern kann. Zu diesem Zweck werden die Tokens **<|startoftext|>** und **<|endoftext|>** in einem zukünftigen Tokenisierer verwendet werden. Das Fehlen dieser Token im Sprachmodell kann zu einem kritischen Fehler führen. Es spielt keine Rolle, wie sie geschrieben werden, zum Beispiel **<s>** oder **sentence_starts** anstelle von **<|startoftext|>** wäre in Ordnung, um einen Tokenisierer zu erstellen, wichtig ist, dass das gleiche Token in jedem Fall von Satzanfang verwendet wird und nicht jedes Mal ein neues.

5.1.2 Input Embedding

Nachdem der Text in Tokens zerlegt und numerisch kodiert wurde, erfolgt der nächste Schritt in der Transformer-Architektur: die **Eingabeeinbettung** (Input Embedding). Dieser Prozess ist fundamental, um die numerischen Token-IDs in eine für das Modell verständliche Form zu überführen. Die Eingabeeinbettung dient dazu, die diskreten Token-IDs in kontinuierliche Vektoren fester Grösse zu transformieren. Da neuronale Netzwerke besser mit kontinuierlichen Daten arbeiten können, ermöglicht die Einbettung eine effiziente Verarbeitung und das Erfassen semantischer Beziehungen zwischen den Tokens. Ohne diesen Schritt wären die Token-IDs lediglich isolierte Zahlen ohne inhärente Bedeutung oder Struktur, was das Lernen und die Generalisierung des Modells erheblich erschweren würde.

In vielen Transformer-Modellen, einschliesslich des ursprünglichen Modells aus *"Attention Is All You Need"*, beträgt die Dimension der Einbettungsvektoren $d_{model} = 512$. Diese hohe Dimension ermöglicht es dem Modell, komplexe Muster und Beziehungen zwischen den Tokens zu erfassen. Ein 512-dimensionaler Vektor kann als ein Punkt in einem 512-dimensionalen Raum betrachtet werden, wobei jede Dimension eine spezifische Eigenschaft oder ein Merkmal des Tokens repräsentieren kann. Vergleichbar mit einem mehrdimensionalen Merkmalsraum in anderen Bereichen des maschinellen Lernens, erlaubt diese hohe Dimensionalität eine feinkörnige und differenzierte Darstellung der Eingabedaten.

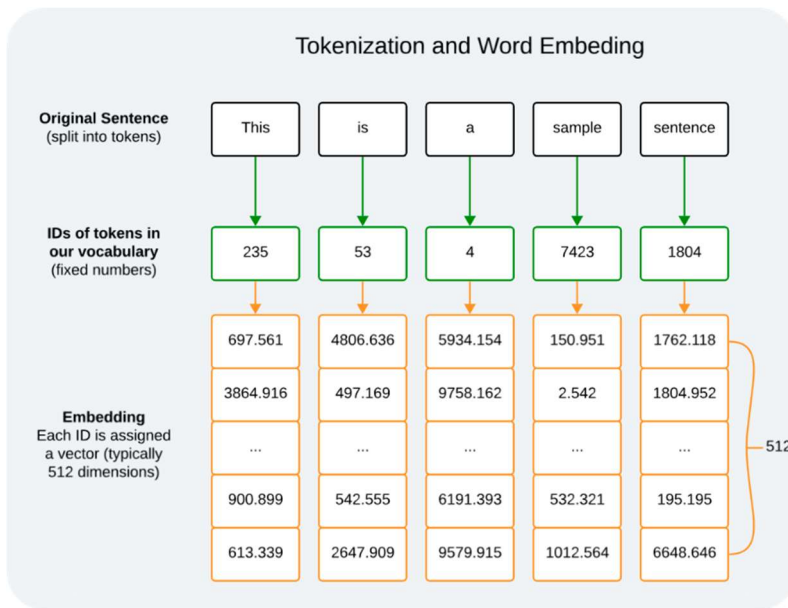


Abbildung 5: Beispiel für Token-Embedding (eigene Darstellung)

$$\text{Token-Embedding: } e_i = E[t_i]$$

Hierbei ist t_i die Token-ID des i -ten Tokens und e_i der entsprechende Einbettungsvektor.

Die Eingabeeinbettung ist ein lernbarer Teil des Transformer-Modells, was bedeutet, dass die Embedding-Matrix E während des Trainings angepasst wird, um die Leistung des Modells zu optimieren. **Backpropagation** ist der Algorithmus, der verwendet wird, um die Gradienten der Verlustfunktion bezüglich der Modellparameter zu berechnen und diese Parameter entsprechend zu aktualisieren. Während des Trainingsprozesses fließt der Fehler vom Ausgang des Modells zurück durch alle Schichten, einschliesslich der Eingabeeinbettung. Die Gradienten, die durch die Backpropagation berechnet werden, ermöglichen es dem Modell, die Einbettungen und weitere Modellgewichte so zu optimieren, dass semantisch ähnliche Tokens im Vektorraum nahe beieinander liegen. Dies verbessert die Fähigkeit des Modells, Kontext und Bedeutungsnuancen zu erfassen, was letztlich die Qualität der generierten Ausgaben steigert. [16]

5.1.3 Positional Encoding

Nachdem die Token-IDs in kontinuierliche Vektoren umgewandelt wurden, ist es für das Transformer-Modell unerlässlich, die Reihenfolge der Tokens in der Eingabesequenz zu berücksichtigen. Bei der Betrachtung der Vektoren fällt uns jedoch nicht auf, dass das Modell Informationen über die Positionen erhält. Um dem Modell dieses Wissen zu vermitteln, wird ein Positionsvektor hinzugefügt, der die Position jedes Tokens in der Sequenz kodiert. Diese positionelle Kodierung (**Positional Encoding**), die mit PE bezeichnet wird, hat ebenfalls die Dimension d_{model} .

$$\text{Positional Encoding: } p_i = PE[i]$$

Mit diesen Informationen kann man vermuten, wie der Prozess der Zuweisung von Vektoren zu jedem Token aussehen sollte: Jedes Token wird mithilfe einer Embedding-Matrix, häufig bezeichnet als E , in einen d_{model} -dimensionalen Vektor umgewandelt. Für ein Vokabular mit V einzigartigen Tokens und einer Einbettungsdimension von d_{model} hat die Embedding-Matrix die Dimension $V \times d_{model}$.

Bemerkung: In den letzten zwei Schritten wurden zwei verschiedene Begriffe verwendet, die auf den ersten Blick dasselbe bedeuten könnten. Mit **Embedding** meint man diejenige Kodierung, die im Laufe des Trainings auch gelernt wird. **Encoding** auf der anderen Seite bedeutet, dass die Werte einmalig berechnet und dann fix zugewiesen werden.

Im ursprünglichen Transformer-Modell wurden **Sinusoidal Positional Encodings** verwendet. Sie basieren auf Sinus- und Cosinus-Funktionen unterschiedlicher Frequenzen und ermöglichen eine Extrapolation auf längere Sequenzen als die, die während des Trainings gesehen wurden. Die Berechnung geschieht in der Regel in der ersten Schicht des Modells und bleibt während des gesamten Trainings und der Inferenz konstant.

Die Positionskodierung wird wie folgt berechnet:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

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

Hierbei gilt:

- pos ist die Position des Tokens in der Sequenz (beginnend bei 0).
- i ist die Dimension innerhalb des Einbettungsvektors (beginnend bei 0).
- d_{model} ist die Gesamtzahl der Dimensionen der Einbettung (z.B. 512).

Die finale Eingabedarstellung z_i für jedes Token i wird durch die Addition des Token-Embeddings e_i und des Positional Encodings p_i berechnet, bevor sie in den Encoder oder Decoder eingespeist werden. [16], [18]

$$z_i = e_i + p_i$$

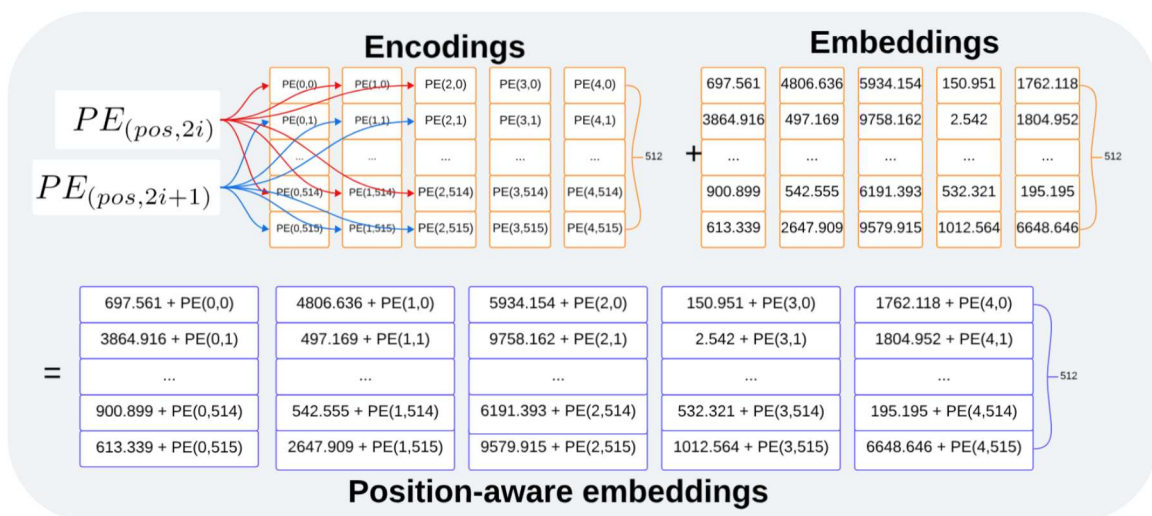


Abbildung 6: Addition von Kodierungen und Einbettungen (eigene Darstellung)

5.1.4 Single Head Self-Attention

Sobald die Eingabedarstellungen erhalten wurden, werden sie vom nächsten Modul der Architektur empfangen (Abbildung 4). Die Multi-Head Attention ist genau das Element, das das Modell so effizient macht, dass es sich mühelos eine riesige Menge an Informationen einprägen kann. Dieser Block wiederum besteht hauptsächlich aus Single Head Self-Attentions. Dieses Konzept ist wichtig für ein intuitives Verständnis der Architektur des Transformers

Die Self-Attention ermöglicht es dem Modell, Beziehungen zwischen verschiedenen Positionen in einer Eingabesequenz zu erfassen. Im Gegensatz zu traditionellen sequenziellen Modellen wie z. B. **RNNs** (Recurrent Neural Networks) kann Self-Attention globale Abhängigkeiten in der Sequenz berücksichtigen, unabhängig von deren Position. Die grundlegende Idee hinter der **Self-Attention** besteht darin, dass jedes Token in der Eingabesequenz mit jedem anderen Token interagieren kann, um kontextuelle Informationen zu sammeln. Dies ermöglicht es dem Modell, Abhängigkeiten über lange Distanzen hinweg zu erfassen und relevantere Repräsentationen der Tokens zu erzeugen.

Im Fall der **Single Head Self-Attention** wird ein einzelner Aufmerksamkeitsmechanismus verwendet, um die Gewichtungen zwischen allen Positionen in der Sequenz zu berechnen. Dieser Mechanismus wird mit Hilfe der drei Matrizen Query (**Q**), Key (**K**) und Value (**V**) berechnet, die aus der Eingabeeinbettungen **Z** und den Gewichtungsmatrizen für Query W^Q , Key W^K und Value W^V bestehen. Gewichtungsmatrizen sind die Parameter, die vom Modell während des Trainings trainiert werden. Zu Beginn des Trainings werden sie nach dem Zufallsprinzip generiert, und eine der Aufgaben des Modells besteht darin, die richtigen Werte zu finden, die sicherstellen, dass die Aufmerksamkeit richtig funktioniert und es den Kontext der Eingabedaten korrekt verarbeiten und analysieren kann. Matrizen W^Q , W^K und W^V haben immer die Dimensionen von $d_{model} \times d_{model}$, so dass man bei der Multiplikation der Matrix Eingabeeinbettungen **Z** der Dimension $seq \times d_{model}$ (seq bedeutet hier die Anzahl der Token, die aus der Text-zu-Token-Konvertierung erhalten wurden) eine Matrix der gleichen Dimension $seq \times d_{model}$ erhält.

Die Prozesse der Single Head Self-Attention lassen sich wie folgt darstellen:

1. Berechnung der Query-, Key- und Value-Matrizen:

$$Q = ZW^Q, K = ZW^K, V = ZW^V$$

2. Berechnung der Aufmerksamkeitsmatrix:

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

In der Formel für die Attention bedeutet K^T eine transponierte Matrix, die aus der ursprünglichen Matrix K durch Ersetzen von Zeilen durch Spalten mit den Dimensionen $d_{model} \times seq$. [16]

5.1.5 Multi-Head Attention

Bei der Verwendung von Single Head Self-Attention erhalten wir am Ausgang nur eine Aufmerksamkeitsmatrix. Google-Forscher haben sich mit der Idee beschäftigt, die Anzahl der Aufmerksamkeitsmatrizen zu erhöhen, und sind zu einer eleganten Lösung gelangt. Anstatt die Aufmerksamkeit nur einmal über die gesamte Einbettungsdimension zu berechnen, teilten sie die Einbettungsdimension durch die Anzahl der Köpfe und berechneten die Aufmerksamkeit für jede dieser kleineren Dimensionen separat. Bei einer Einbettungsdimension von 512 und 4 Köpfen beträgt die Dimension jeder Kopf-Einbettung 128.

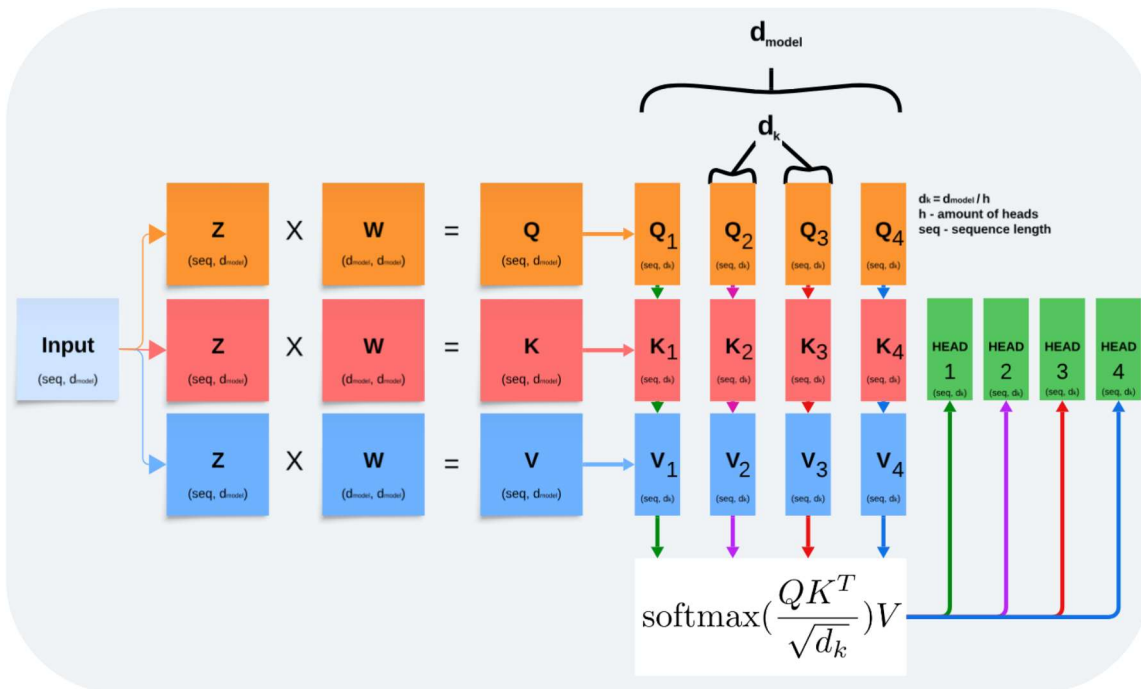


Abbildung 7: Visualisierung der Kopferfassung bei Multi-Head Attention (eigene Darstellung)

Abbildung 7 zeigt den Prozess der Umwandlung der ursprünglichen Eingabedarstellungen für einen Text in separat gezählte Köpfe für jede Einbettungsgruppe. Nachdem die Kopfdaten vorliegen, muss noch eine weitere Aktion durchgeführt werden, bevor die Daten zum nächsten Block weitergeleitet werden, und zwar die Anwendung der Formel

$$MultiHead(Q, K, V) = Concat(head_1, head_1 \dots head_n)W^O$$

um die endgültige Matrix zu erhalten. **Concat()** ist nichts anderes als ein einfaches Aneinanderreihen aller Köpfe, um auf die ursprüngliche Dimension von $seq \times d_{model}$ zu kommen, wonach die Matrizen mit zusätzlichen Parametern W^O der Dimension $d_{model} \times d_{model}$ multipliziert werden, die ebenso wie andere Modellgewichte während des Trainings modifiziert werden.

Betrachtet man Multi-Head Attention mathematisch, sind die Berechnungen nicht besonders kompliziert. Eine Matrix wird vom Multi-Head Attention Block übernommen, einige Modifikationen dieser Matrix durchgeführt, und es erfolgen Multiplikationen mit Gewichten, die als statische Parameter des trainierten Modells in Matrizen zusammengefasst sind. Die einzige Operation, die neu und unklar sein kann, ist softmax, aber sie ist lediglich für die Umsetzung Vektor von reellen Zahlen in eine Wahrscheinlichkeitsverteilung zuständig und wird in Abschnitt 5.1.10 näher erläutert. Am Ende produziert der Block eine Matrix derselben Dimension wie die vom Block angenommene und es ist nicht intuitiv möglich zu sagen, was diese Matrix bedeutet.

Um die Aktionen im Block zu interpretieren, gehen wir davon aus, dass wir mit einem trainierten Modell arbeiten, das über angepasste Parameter verfügt und gelernt hat, den Kontext zu verstehen. Berechnen wir die Aufmerksamkeit nach Anwendung von Softmax, erhalten wir die Matrix $seq \times seq$.

Wenn wir diese Matrix mit Hilfe einer Heatmap darstellen (ein Bild, das mit einer Farbtintensität den in der Matrix in dieser Zelle gespeicherten Wert zeigt), sehen wir ein interessantes Muster. Wir können dies als den Grad der Aufmerksamkeit interpretieren, den das Modell jedem der Wörter schenkt, wenn ein bestimmtes Wort genannt wird. So sehen wir zum Beispiel, dass bei dem Wort Er die Aufmerksamkeit sowohl auf das Wort Er als auch auf das Wort Bericht gerichtet ist, weil das Modell davon ausgeht, dass diese Wörter miteinander

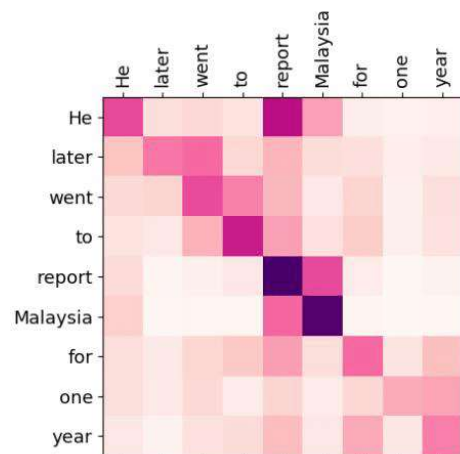


Abbildung 8: visuelle Darstellung der Score-Matrix [19]

verwandt sind. Wenn wir also beobachten, welchen Wörtern das Modell Aufmerksamkeit schenkt, können wir Muster erkennen, die es verwendet, um den Kontext zu verstehen.

Multipliziert man diese Matrix mit den Werten V , so erhält man eine neue Matrix der Dimension $seq \times d_{model}$, in der jedem Token nicht nur die ursprüngliche Einbettung, sondern auch seine Beziehung zu anderen Wörtern in der Sequenz zugeordnet ist.

Ein Beispiel dafür ist der Satz "Er schreibt einen Bericht". Wenn man sich auf das Wort "schreibt" konzentriert, kann man den Wörtern "Er" und "Bericht" mehr Aufmerksamkeit schenken, weil sie mit der durch dieses Wort ausgedrückten Handlung in Verbindung stehen. In der Zeile der Aufmerksamkeitsmatrix, die dem Wort „schreibt“ entspricht, werden die Gewichte für „Er“ und „Bericht“ höher und für die anderen Wörter niedriger sein. Wenn diese Zeile mit V multipliziert wird, werden

die Gewichte verwendet, um die Einbettung der anderen Wörter zu gewichten, was zu einer angereicherten Darstellung von „schreibt“ führt, die Subjekt- („Er“) und Objektinformationen („Bericht“) enthält.

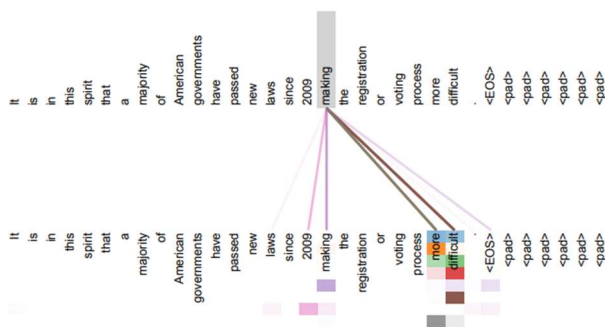


Abbildung 9: Multi-Head Score-Matrizen für ein Wort [16]

Wenn wir mehrere Köpfe betrachten, erhalten wir so viele ähnliche Bilder mit gegenseitigen Beziehungen, wie wir Köpfe haben, und es macht Sinn, nicht alle Matrizen zu berücksichtigen, sondern die Menge der Aufmerksamkeit auf ein konkretes Wort nach dem Durchgang verschiedener Köpfe, also in Abbildung 9 ist

es sichtbar, dass verschiedene Köpfe im trainierten Modell in der Mehrheit Wortpaaren den Vorzug geben werden, aber es wird auch Köpfe geben, die andere Zusammenhänge sehen werden, die durch andere Einbettungen verursacht werden, die in einem Kopf ankommen. [16]

5.1.6 Add and Normalize

Anhand des Namens dieses Blocks lässt sich leicht erraten, dass er eigentlich aus zwei Schritten besteht, nämlich **add und normalize**. Diese beiden Konzepte stellen sehr unterschiedliche Ideen dar, haben aber ein gemeinsames Ziel: ein schnelleres und effizienteres Training eines neuronalen Netzes durch die Manipulation von Matrixwerten.

Residualverbindungen (Add), die auch als Skip-Connections bekannt sind, wurden erstmals im Rahmen der ResNet-Architektur eingeführt. Sie adressieren das Problem des vanishing gradients, das beim Training sehr tiefer neuronaler Netze auftritt.

Vanishing gradient ist ein Problem, das beim Training tiefer neuronaler Netze auftritt, insbesondere bei Netzen mit einer grossen Anzahl von Schichten. Es besteht darin, dass die Gradienten (d. h. die Ableitungen der Fehlerfunktion nach Modellparametern) während der Backpropagation des Fehlers sehr klein werden, was zu einer Verlangsamung oder einem vollständigen Abbruch des Trainings führt.

Add löst dieses Problem durch Hinzufügen einer Restbeziehung, die in Abbildung 10 zu sehen ist und die Multi-Head Attention umgeht. Der Add-Schritt führt also eine einfache **stückweise Addition** zwischen den ursprünglichen Embeddings und den von Multi-Head Attention zurückgegebenen angereicherten Repräsentationen durch:

$$\text{Output} = \text{MultiHead}(Q, K, V) + Z$$

Durch die Residualverbindung kann der Gradient ohne Dämpfung direkt durch die Schicht hindurchgehen, auch wenn die Gradientenwerte in der Masse der Schicht abnehmen.

Die aus der Residualverbindung resultierende Ausgabe durchläuft anschliessend einen Layer-Normalisierungsprozess. Diese von Jimmy Lei Ba und Kollegen entwickelte Normalisierungstechnik operiert über die Feature-Dimensionen und wurde spezifisch für die Anforderungen sequenzieller Modellarchitekturen wie des Transformers konzipiert. Durch die Standardisierung der Eingaben wird verhindert, dass extreme Werte die nachfolgenden Berechnungen dominieren. Die lernbaren Parameter γ und β ermöglichen es dem Modell, die Normalisierung optimal an die Daten anzupassen und den Informationsfluss nicht unnötig einzuschränken. Die Layer-Normalisierung standardisiert die Ausgaben eines Layers, indem sie den Mittelwert und die Varianz der Eingabedaten berechnet und diese Werte zur Normalisierung verwendet. Dies geschieht für jedes einzelne Element der Reihe nach über alle Parameter hinweg, und es wird Formel

$$x_i^{new} = \gamma \left(\frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta$$

verwendet. Dabei sind γ und β , wie schon oben erwähnt wurde, lernbare Parameter, ϵ ist ein kleiner Wert, der für die numerische Stabilität hinzugefügt wird. Der Parameter ϵ ist nur empirisch bestimmbar und sein Wert liegt normalerweise zwischen 10^{-5} und 10^{-8} . x_i ist der ursprüngliche Wert (eine von d_{model} Einbettungen des Elements, mit dem zu dem Zeitpunkt gearbeitet wird) und x_i^{new} ist der Wert, der anstelle des ursprünglichen Werts gespeichert wird.

μ und σ^2 sind der Mittelwert des Eingangssignals bzw. die Varianz des Eingangssignals und werden mit Hilfe der Formeln

$$\mu = \frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i$$

und

$$\sigma^2 = \frac{1}{d_{model}} \sum_{i=1}^{d_{model}} (x_i - \mu)^2$$

berechnet. [16], [20], [21]

5.1.7 Feed Forward

Sowohl im Decoder als auch im Encoder gibt es neben dem Multi-Head Attention Block und dem Add & Normalise Block eine weitere wichtige Komponente, das Feed Forward Network (FFN). Im Kern ist das FFN ein vollständig verbundenes positionsbasiertes neuronales Netz (Abbildung 10).

Die Aufgabe des FFNs ist es, die Ausgabe des Aufmerksamkeitsmechanismus zu verfeinern, indem es eine Transformation durchführt, die positionsspezifische Informationen einfängt und verstärkt. Die positionsbasierte Verarbeitung stellt sicher, dass das FFN, obwohl jedes Token mit anderen durch den Aufmerksamkeitsmechanismus interagiert, diese Interaktionen unabhängig voneinander verfeinern kann, ohne dass eine Abhängigkeit zwischen verschiedenen Positionen in der Sequenz besteht.

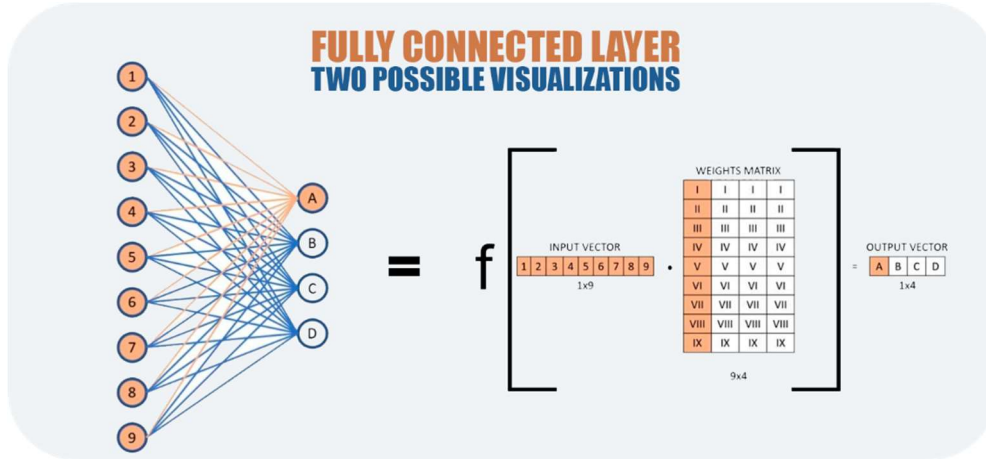


Abbildung 10: zwei Möglichkeiten zur Darstellung einer vollständig verbundenen Schicht [22]

Das FFN besteht aus zwei vollständig verbundenen Schichten. Die erste Schicht projiziert die Eingabedaten linear und erweitert sie von der Dimensionalität der d_{model} Einbettung auf einen höherdimensionalen Raum mit der Bezeichnung d_{ff} (in der Regel 4-mal grösser als d_{model}). Diese Erweiterung ermöglicht es dem Netz, eine komplexere, höherdimensionale Darstellung der kontextuellen Beziehungen jedes Tokens zu erfassen. Die zweite Schicht projiziert diese erweiterte Repräsentation anschliessend zurück in die ursprüngliche Dimensionalität d_{model} , wodurch die Informationen bei entsprechendem Training genauer und komprimierter werden, um sie dann an den Aufmerksamkeitsmechanismus zurückzugeben. Zwischen diesen beiden vollständig gekoppelten Schichten erfolgt die Verarbeitung mit Hilfe einer nichtlinearen **ReLU**-Aktivierungsfunktion. Diese Nichtlinearität, mathematisch definiert als

$$ReLU(x) = \max(0, x),$$

verleiht dem Modell einen gewissen Grad an Komplexität, der es ihm ermöglicht, komplexe Abhängigkeiten innerhalb der Eingabesequenz zu lernen. Die **ReLU**-Aktivierung wird Element für Element angewandt und verbessert daher die Fähigkeit des FFN, nichtlineare Beziehungen darzustellen, was für die Erfassung komplexer sprachlicher Nuancen unerlässlich ist.

Die endgültige Formel beinhaltet die Multiplikation des Vektors jedes Tokens mit einer $d_{model} \times 4 * d_{model}$ **Gewichtsmatrix**, die Addition eines Biases, die Anwendung von **ReLU** und die erneute Multiplikation des resultierenden Vektors mit einer weiteren $4 * d_{model} \times d_{model}$

Gewichtsmatrix und die Addition eines anderen Biases. Das Endergebnis ist eine Matrix mit den ursprünglichen Dimensionen, deren Bedeutung schwer zu interpretieren ist, die aber für die Architektur des Transformer ein wichtiger Schritt ist. [16], [23]

Der gesamte Prozess lässt sich wie folgt zusammenfassen:

$$FFN(X) = ReLU(XW_1 + b_1)W_2 + b_2,$$

wobei:

- X der Eingabevektor ist (Repräsentation des Tokens),
- W_1, W_2 die Gewichtsmatrizen des ersten bzw. des zweiten linearen Layers sind,
- b_1, b_2 die Bias-Vektoren sind,
- $ReLU$ die Aktivierungsfunktion Rectified Linear Unit ist, wie bereits erwähnt.

5.1.8 Multi-Head Cross Attention

In den vorangegangenen Abschnitten wurden alle Details behandelt, die zum Verständnis des Encoders erforderlich sind. Die Architektur mag verwirrend erscheinen, und das ist sie auch. Aber das Wichtigste ist zu verstehen, dass es einen Encoder-Block gibt, der eine Eingabe entgegennimmt und sie in eine nützliche Darstellung umwandelt – eine Matrix einer bestimmten Dimension. Die Ausgabe des Encoders ist ebenfalls eine Matrix derselben Dimension, aber mit anderen Werten, die mehr Informationen enthalten als die Eingabe. Ein Blick auf die Architektur (Abbildung 4) zeigt, dass die Ausgangsmatrix nach dem Ausgang des Encoders der Multi-Head Attention zugeführt wird, die sich im Decoder befindet. Tatsächlich wird Self-Attention nicht überall in der Transformer-Architektur verwendet, denn dieser Multi-Head Attention-Block nimmt zwei Parameter auf, nämlich die Matrix zur Bildung von Keys K und Values V aus dem Encoder-Block und die Matrix zur Bildung von Query Q aus der vorherigen Schicht in seinem Decoder-Block. Diese Tatsache ändert nur, dass wir jetzt nicht mehr nur eine einzige Matrix Z haben, die dann mit Gewichtsmatrizen multipliziert wird, sondern wir haben jetzt sowohl Z_{Enc} als auch Z_{Dec} , die aus verschiedenen Quellen stammen. Diese werden separat für die Berechnung von K , V und Q verwendet. Ansonsten unterscheidet sich das, was innerhalb der Multi-Head Cross Attention-Schicht passiert, nicht von der oben beschriebenen Multi-Head Attention. [16], [24]

5.1.9 Masked Multi-Head Attention

Die Besonderheit dieser Schicht ist die Kausalität. Das bedeutet, dass die Ausgabe an einer bestimmten Position nur von den Wörtern an den vorherigen Positionen abhängen sollte. In einer herkömmlichen Multi-Head-Attention-Schicht kann ein Wort alle Wörter in der Eingabe sehen und mit ihnen in Beziehung treten, während diese Architektur nur auf die vorherigen Wörter beschränkt ist.

Maskierte Aufmerksamkeit gewährleistet die **Kausalität des Modells**, indem sie sicherstellt, dass die Vorhersage an Position i nur von den Positionen abhängt. Dies ist von entscheidender Bedeutung für Sequenzgenerierungsaufgaben wie LLM oder maschinelle Übersetzung.

Das Erreichen dieses Effekts ist recht einfach und ersetzt einfach alle Werte, die auf das aktuelle Wort folgen, durch $-\infty$, so dass bei Anwendung von softmax die Wahrscheinlichkeit Null ist und die Quadrate im Wesentlichen ignoriert werden, wie in Abbildung 11 mit grauen Quadraten in der Matrix dargestellt.

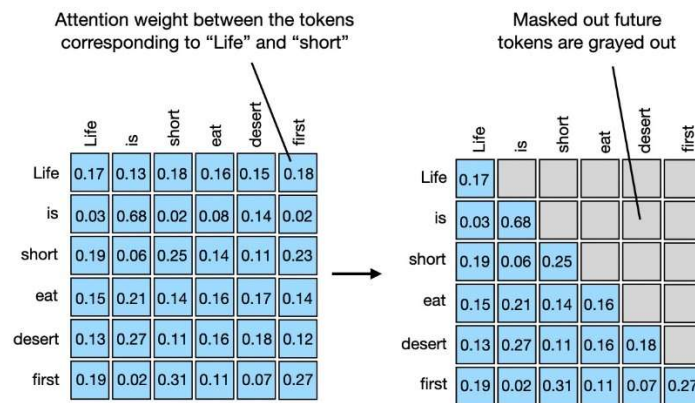


Abbildung 11: Masked Multi-Head Attention Score Matrix [25]

5.1.10 Linear

Die Ausgabe des vorherigen Layers des Decoders hat die Dimension $seq \times d_{model}$. Das "Linear"-Layer projiziert die Ausgabe des Decoders in den Vokabularraum. Wenn das Vokabular Y einzigartige Tokens enthält, hat die Gewichtsmatrix W^L des "Linear"-Layers die Dimension $d_{model} \times Y$. Die beiden Matrizen werden multipliziert, um eine endgültige Matrix der Dimension $seq \times Y$ zu erhalten, und sie an die Softmax-Schicht weiterzugeben.

Bemerkung: Fast überall, wo mit Gewichten multipliziert wird, ist es möglich, auch Bias zusätzlich zu verwenden, und in den meisten Fällen ist das auch der Fall, denn Bias ist im Grunde ein Bias, aber um den Leser nicht zu verwirren und sich auf die Schlüsselkonzepte in Attention All You Need und in dieser Beschreibung der Architektur, die dem Beispiel folgt, zu konzentrieren, wird Bias weggelassen.

5.1.11 Softmax

Softmax nimmt als Eingabe Logits - nicht-normierte Werte, die positiv, negativ oder Null sein können - und wandelt sie in Wahrscheinlichkeiten um. Die Wahrscheinlichkeiten liegen zwischen 0 und 1 und alle Wahrscheinlichkeiten in einer Reihe summieren sich zu 1, so dass sie sich für die

Interpretation als Vorhersagen von Wahrscheinlichkeiten für Klassen eignen. Obwohl wir uns immer für das nächste Token interessieren, dessen Vorhersage wir wollen, weil die vorherigen verfügbar sind, erzeugt die Transformer-Architektur Vorhersagen für Token ab dem zweiten bis zum letzten Token, das wir wollen. Dies ist leicht zu verstehen, da die Ausgabe von Linear kein Vektor, sondern eine Matrix ist. So enthält beispielsweise die erste Zeile der Softmax-Matrix die Wahrscheinlichkeiten für alle Token, die an der **zweiten Position** in der Sequenz ausgewählt werden können. Die zweite Zeile ist eine Vorhersage der Wahrscheinlichkeiten für die **dritte Position** und so weiter, bis die Wahrscheinlichkeiten für alle Positionen in der Sequenz generiert sind. [16]

Softmax wird grundsätzlich zur komfortableren Darstellung von Daten verwendet, da es immer bequemer ist, mit Wahrscheinlichkeiten zu arbeiten als mit Zahlen, die in einem riesigen Bereich verstreut sind. Aber in der Softmax-Formel gibt es noch ein weiteres Merkmal, das nach Belieben hinzugefügt werden kann, und das heisst Temperatur τ . Betrachten wir dies für die resultierende Matrix der Dimension $seq \times Y$. Wie wir uns erinnern, ist Y die Länge unseres Wörterbuchs, und in der Tat für jede der Zeilen von der ersten bis seq müssen wir Wahrscheinlichkeiten erhalten. Hier wird jede Zeile in seq als ein Vektor $\mathbf{z} = (z_1, \dots, z_Y)$ betrachtet. Hier wird jede Zeile in seq als ein Vektor Z betrachtet. Dann wird die Wahrscheinlichkeit für das aktuelle Wort im Wörterbuch nach der Formel

$$P_i = \frac{e^{\frac{z_i}{\tau}}}{\sum_{k=1}^Y e^{\frac{z_k}{\tau}}}$$

berechnet. [26]

Die Temperatur ist eine Möglichkeit, die Entropie einer Verteilung zu kontrollieren, wobei die relativen Ränge der einzelnen Ereignisse erhalten bleiben. Abbildung 12 zeigt, dass mit steigendem Parameter τ die Wahrscheinlichkeiten derjenigen Werte zunehmen, die bei niedrigeren Temperaturen vernachlässigbare Werte haben. Je höher die Temperatur, desto grösser die Entropie (die Unordnung) [27]

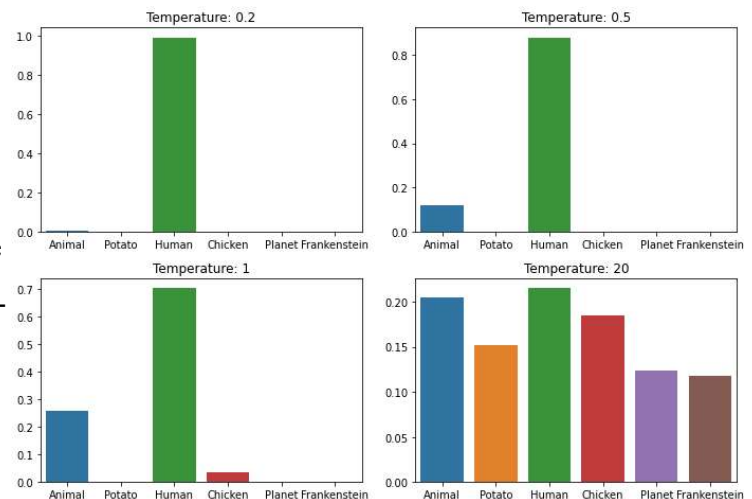


Abbildung 12: Abhängigkeit der Wahrscheinlichkeiten von der Temperatur [28]

5.1.12 Trainingsverlauf

Das Training des Transformationsmodells zielt darauf ab, seine Parameter (Gewichte) so einzustellen, dass das Modell das nächste Token in der Sequenz genau vorhersagen kann. Dieser Prozess

umfasst mehrere wichtige Schritte: Feedforward-Propagation, Berechnung der Verlustfunktion, Backpropagation und Gradientenabstieg. Betrachten wir nun jede dieser Komponenten und ihre Beziehungen untereinander im Detail.

In der Feedforward-Phase werden die Eingabedaten durch das Modell geleitet, und die Ausgabe sind die vorhergesagten Wahrscheinlichkeiten für jedes mögliche nächste Token in der Sequenz. Alle oben genannten Details der Transformer-Architektur sind im Wesentlichen die Details dessen, was mit den Daten in dieser Phase geschieht.

Sobald die vorhergesagten Wahrscheinlichkeiten ermittelt sind, muss bewertet werden, wie nahe sie an den tatsächlichen Werten liegen. Dazu wird eine Verlustfunktion verwendet, bei Transformers in der Regel der Cross-Entropy-Loss. Sie misst die Differenz zwischen der vorhergesagten Wahrscheinlichkeitsverteilung und der tatsächlichen Verteilung nach der Formel

$$L = - \sum_{i=1}^Y p_i^{desired} \log(p_i^{actual}),$$

die das Produkt der gewünschten Wahrscheinlichkeiten (1 für den wahren Wert, der im Beispiel vorkommt, und 0 für alle anderen) mit dem Logarithmus der vom Modell erzeugten Wahrscheinlichkeitswerte addiert. Im Ergebnis liefert die Verlustfunktion eine quantitative Schätzung, wie gut das Modell die richtigen Token vorhersagt, und dient auch als Signal, um die Modellparameter in Richtung einer Fehlerreduzierung anzupassen. [29]

Der nächste Schritt ist die **Backpropagation**, ein Prozess, der es dem Modell ermöglicht zu verstehen, wie seine Parameter (Gewichte und Verzerrungen) angepasst werden müssen, um die Verlustfunktion zu minimieren und die Vorhersagegenauigkeit zu verbessern. Bei der **Backpropagation** werden die Gradienten der Verlustfunktion in Bezug auf die einzelnen Modellparameter berechnet. Dies geschieht mit Hilfe einer Ableitungskettenregel, die auf die Abfolge der Operationen, aus denen das Modell besteht, angewendet wird. Alle neuronalen Netze beruhen auf dem Prinzip der Backpropagation, und das Konzept ist recht schwierig zu beschreiben. [30]

Der beste Weg, um Backpropagation zu verstehen, sind Beispiele, die das Verständnis des Konzepts unglaublich verbessern. Es gibt viele Materialien auf YouTube und im öffentlichen Bereich, die klar beschreiben, wie Backpropagation funktioniert, wie z. B. [diese Playlist auf YouTube](#), die im ersten Video ein Verständnis des Konzepts der Backpropagation vermittelt und in weiteren Videos den ML- Lernhorizont weiter ausweitet.

Bei Trainingsdaten wird die Dauer des Trainingsprozesses in der Regel durch Epochen beschrieben, die im Wesentlichen eine Durchquerung des gesamten Trainingsdatensatzes darstellen. Es liegt auf

der Hand, dass der Transformer nicht den gesamten Datensatz auf einmal durchlaufen kann. Daher werden die Daten innerhalb einer Epoche in Stapel aufgeteilt, die durch den Wert `batch_size` definiert sind. Wenn wir also die Länge des Datensatzes durch `batch_size` teilen, erhalten wir die Anzahl der Batches sowie die Anzahl der Iterationen (in fast allen Metriken ist dieser Wert die Masseinheit für den Trainingsfortschritt des Modells), die zum Durchlaufen einer Epoche erforderlich sind. Jetzt können wir die gewünschte Anzahl der vollständigen Durchläufe durch den Datensatz angeben, die durch den Parameter `num_train_epochs` bestimmt wird. Multipliziert man diesen Parameter mit der Anzahl der Iterationen in einer Epoche, erhält man die Gesamtzahl der Schritte, die für den Abschluss des Trainings erforderlich sind, wobei jeder Schritt einen Durchlauf der Daten durch das Transformer-Modell bedeutet.

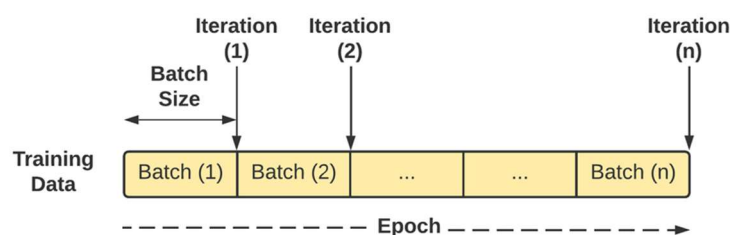


Abbildung 13: visuelle Darstellung der Batches [31, S. 1]

Wenn ein Modell trainiert wird, werden die Gewichte zufällig generiert. Es gibt jedoch auch eine Möglichkeit, die Gewichte eines bereits bestehenden Modells zu laden und das Training darauf aufzubauen, was deutlich schneller verläuft und nicht so viele Rechenressourcen erfordert wie das Training von Grund auf. Dieser Prozess wird Fine-Tuning genannt.

5.2 Aufbau von GPT

Zu Beginn des Kapitels wurde erwähnt, dass die Architektur ursprünglich für maschinelle Übersetzungsaufgaben entwickelt wurde, für die zwei Blöcke entwickelt und das Konzept der kreuzweisen mehrköpfigen Aufmerksamkeit eingeführt wurde. Die Forscher bei Google erkannten wahrscheinlich, dass die Architektur auch für andere, noch komplexere Aufgaben verwendet werden könnte. Daher begann Google nach einiger Zeit, Modelle nur mit Encoder zu testen. Etwas früher begann auch OpenAI damit, Variationen dieser Architektur zu testen, und wenn wir uns die GPT-Architektur ansehen, werden wir die Verwendung von Encoder dort nicht bemerken, was die Architektur zu einer Decoder-basierten Architektur macht. Wir bemerken die Verwendung von Masked Multi-Head Attention und anderen Schichten, die namentlich aus der Beschreibung der ursprünglichen Architektur bekannt sind.

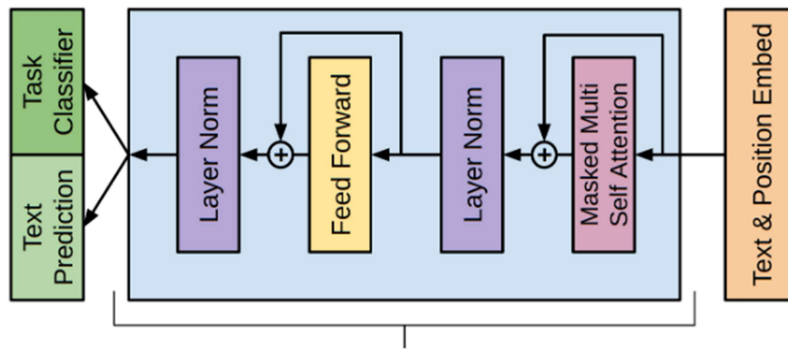


Abbildung 14: vereinfachte GPT-Struktur [32]

Tatsächlich hat OpenAI die Struktur der Schichten leicht verändert. Beispielsweise sieht man in der detaillierten Darstellung der Architektur, dass ReLU im Feed-Forward durch GELU (Gaussian Error Linear Units) ersetzt wurde. Ausserdem wurden Dropouts hinzugefügt, die einige Werte der Matrix zufällig auf 0 setzen, um Overfitting zu verhindern. Der Transformer-Head sieht ebenfalls etwas anders aus, und Linear, das fast nach jedem Schritt zu sehen ist, bedeutet einfach die Multiplikation der Modellvektoren mit den Gewichten und das Hinzufügen eines Bias, was nichts Neues darstellt.

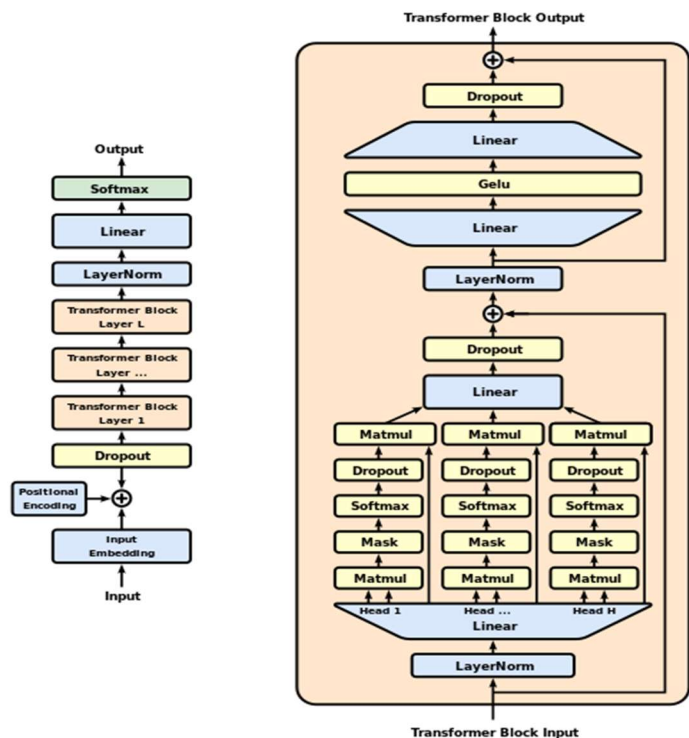


Abbildung 15: detaillierte Struktur der GPT-Architektur [32]

6 Projektumsetzung

Alles beginnt mit einer Idee, und dieses Projekt ist keine Ausnahme. Die Arbeit an der Modellentwicklung lässt sich kaum als streng geplant bezeichnen, wie es oft bei Projekten im Bereich des maschinellen Lernens der Fall ist. Der Weg zur Zielerreichung ist eher empirisch als theoretisch und rein logisch begründet. Die gesamte Entwicklung verlief im Format von Versuch und Irrtum, und sehr oft, insbesondere in der Phase des Modelltrainings, musste auf eine völlig andere Strategie zurückgegriffen werden als ursprünglich geplant.

Diese Arbeit basiert auf zwei Hauptideen. Erstens soll demonstriert werden, dass Transformermodelle wirklich vielseitig anwendbar sind, indem ein Agent trainiert wird, der die Logik eines Spiels verstehen lernt, als wäre es ein Satz, basierend auf Spielen als Datensatz, die diese Logik demonstrieren. Die zweite Idee ist es, mögliche Ansätze zu erforschen, wie die Transformer Architektur genutzt werden kann, um das Modell nicht nur auf die im Datensatz enthaltene Logik zu trainieren, sondern es auch zur Weiterentwicklung dieser Logik anzuregen, sodass am Ende eine bessere Strategie entsteht als bisher.

6.1 Datenerfassung

Jedes Large Language Model wird aus Trainingsdaten trainiert, daher war der erste Schritt auf dem Weg dorthin das Sammeln von Daten. Im Falle des Schachspiels wäre es im Allgemeinen kein Problem, die richtigen Daten zu finden, da es seit langem begeisterte Seiten im Internet gibt, die Spieldaten sammeln, einschliesslich Informationen über Züge, Elo-Level und Spielresultate. Es wäre kein Problem, z. B. eine Million qualitativ hochwertiger Partien zu finden, was auf die Beliebtheit von Brettspielen in der Welt zurückzuführen ist, da die Daten meist nur eine Aufzeichnung von Partien sind, die von echten Spielern bei Turnieren, auf Websites usw. gespielt wurden.

Hex ist bei weitem nicht so populär wie Schach. Dies erschwert die Datenerfassung. Der Versuch, einen bereits bestehenden Datensatz zu finden, war erfolglos, woraufhin beschlossen wurde, einen eigenen Datensatz zu erstellen. Theoretisch könnten Spiele durch die Zusammenstellung von zufällig generierten Zügen generiert werden. Da aber in solchen Zügen keine Strategie steckt und das Training eines Transformer-basierten Modells bedeutet, dass das Modell lernt, Züge zu generieren, die denen im Trainingssatz ähnlich sind, würde dies bedeuten, dass das Modell nach dem Training kein strategisches Denken mehr hat.

Das Problem wurde gelöst, als eine Website gefunden wurde, die aktiv Turniere und einfache Partien eines Hex-Spiels namens Little Golem veranstaltet. Die Website ermöglicht es, alle Spieler anzuzeigen, die jemals ein bestimmtes Brettspiel gespielt haben, das auf der Website verfügbar ist, und ermöglicht es auch, die Züge aller Partien herunterzuladen, die von einem bestimmten Benutzer

in einem bestimmten Spiel gespielt wurden. Auf diese Weise wurde auf die Partien von über 3000 Benutzern, die Hex gespielt haben, zugegriffen. Nach dem Schreiben kleiner, auf Java-Skript basierender Skripte zur Automatisierung des Prozesses der Datenerfassung auf der Seite war es möglich, alle auf der Website verfügbaren Spiele herunterzuladen. Die Skripte wurden mit der Tampermonkey-Browsererweiterung geschrieben, mit der Skripte erstellt und zu jeder beliebigen Seite im Internet hinzugefügt werden können, wodurch neue Funktionen implementiert werden. Die Verwendung von Code zur Datenerfassung anstelle der manuellen Erfassung reduzierte den Zeitaufwand erheblich.

Nach der Bearbeitung der Partien durch ein Skript, bei dem unter anderem unnötige Informationen wie Spielernamen entfernt, Züge in das gewünschte Format konvertiert und sich wiederholende Partien entfernt wurden, betrug die Zahl der eindeutigen Partien etwa 25000. Diese Anzahl von Beispielen kann definitiv als unzureichend für das Training eines Transformer-Modells bezeichnet werden, aber es war die beste der Möglichkeiten. Abgesehen von der Menge hatten die Daten noch einen weiteren bedeutenden Makel, nämlich dass die überwiegende Mehrheit der Spiele nicht zu Ende gespielt wurde. Das liegt ganz einfach daran, dass sich Hex relativ leicht analysieren lässt, nachdem man die grundlegenden Strategien des Spiels gelernt hat, und ein Spieler, der sieht, dass er keine Chance hat, gibt auf, anstatt Züge zu machen, die offensichtlich zum Verlust führen. Aber beim Training hat das Modell nur Zugang zu Daten, und ohne genau zu sehen, wie man das Spiel zu Ende bringt, kann das Modell es nicht lernen.

Um dieses Problem zu lösen, wurden die vorhandenen unterdurchschnittlichen Spiele mit Hilfe der Website <https://lazymuttgames.com/games/Hex/> beendet, die eine künstliche Intelligenz mit einem ziemlich guten Verständnis des Spiels bietet. Das Skript fügte das unterspielte Spiel in das Eingabefeld ein und wartete, bis das Modell das Spiel beendet hatte. Auf diese Weise war es möglich, 25000 Spiele zu beenden und einen Datensatz mit den ersten Zügen zu erhalten.

Bemerkung: Die künstliche Intelligenz auf der angebotenen Seite hat keine Variabilität bei der Generierung von Partien, d.h. wenn man sie eine Partie von Anfang bis Ende spielen lässt, wird sie nur wenige Muster produzieren, was die Möglichkeit verhindert, eine grosse Anzahl von Partien mit dieser Seite zu generieren. Gleichzeitig werden offensichtlich zu wenig gespielte Partien mit unterschiedlichen Ausgangspositionen von dem Modell unterschiedlich generiert.

Nach der Erstellung des Datensatzes wurde im nächsten Schritt mit dem Training des Modells begonnen. Dies erforderte das Schreiben von Code zur Erstellung eines Tokenisers sowie von Modelltrainingscode, der die GPT-Architektur und den erstellten Tokeniser zum Trainieren des Modells verwenden würde.

6.2 Erstellung eines Tokenisers

Die Datei `tokenizer.py` ist für das Training und Speichern des Tokenizers verantwortlich. Um den Tokenizer nicht von Grund auf neu zu entwickeln, wurden bereits vorhandene Werkzeuge aus den Bibliotheken `tokenizers` und `transformers` verwendet. Diese Bibliotheken bieten eine breite Auswahl an Konfigurationen für das Training von Modellen. In diesem speziellen Fall wurde die Bibliothek `tokenizers` genutzt, um das Modell anhand des Datensatzes mithilfe des Byte-Pair-Encoding (BPE)-Modells zu trainieren, das dafür sorgt, dass häufig vorkommende Wortteile und Zeichenfolgen kompakt dargestellt werden.

Die Tokenisierung erfolgt auf Zugebene (z. B. Zug A11 - 12 Token). Es ist möglich, die Tokenisierung auf der Ebene der Koordinaten darzustellen, wobei jede alphabetische und numerische Koordinate durch separate Token dargestellt wird, wodurch die Gesamtzahl der Token verringert und folglich die Berechnung des Transformers beschleunigt wird. In der Praxis ist die Verwendung eines solchen Tokenisers in einer Spielsituation jedoch nicht ganz angemessen, da der wahrscheinlichste Buchstabe und die wahrscheinlichste Zahl, die darauf folgen, nicht immer der wahrscheinlichste Zug sind (z.B. Die Wahrscheinlichkeiten von A und B sind 0.6 bzw. 0.4, 1 nach dem A – 0.5, 2 nach dem A – 0.4, 1 nach dem B – 0.9). Um dieses Problem zu lösen, müssten wir die Gesamtwahrscheinlichkeit für jede Zahl mit jedem Buchstaben berechnen, was zu einer längeren Berechnungszeit führt, obwohl es nicht ausgeschlossen ist, dass diese Methode der Tokenisierung ihre Vorteile hat, z. B. eine Leistungssteigerung.

Nach dem Training wird der Tokenizer in ein `PreTrainedTokenizerFast`-Objekt umgewandelt, was für die Integration in die `transformers`-Bibliothek, die in den weiteren Schritten verwendet wird, wichtig ist. Dieses Format bietet standardisierte Funktionen für die weitere Nutzung des Tokenizers. Der trainierte Tokenizer wird gespeichert, damit er später einfach wiederverwendet werden kann, ohne eine erneute Erstellung durchführen zu müssen.

6.3 Erstellung einer Trainingsdatei

Die nächste Aufgabe bestand darin, einen Code zu erstellen, der die Decoder-Only-Architektur verwendet und am Ende ein Modell ausgibt, das getestet werden kann. Theoretisch wäre es möglich gewesen, eine eigene Architektur zu entwickeln, aber diese Idee wurde ziemlich schnell verworfen, da die Erstellung nicht nur ein vollständiges Verständnis der Transformer-Architektur erfordert, sondern auch das Verständnis anderer nicht-trivialer Konzepte aus dem Bereich des maschinellen Lernens. Aber selbst mit diesem Wissen wäre es unwahrscheinlich gewesen, etwas Besseres zu erreichen als das, was bereits von Forschern in diesem Bereich erreicht wurde. In Anbetracht des Erfolgs

der in GPT verwendeten Architektur und der leichten Zugänglichkeit von Tutorials und Lehrmaterialien, die diese Architektur nutzen, wurde beschlossen, sie zu verwenden.

Die absolute Mehrheit der Projekte im Bereich des maschinellen Lernens wählt Python. Laut Schätzungen verschiedener Quellen wird Python in 50–75 % der Fälle in dieser Nische verwendet. In der Realität nutzt jedoch niemand reines Python zum Trainieren von Modellen. Stattdessen ermöglicht Python die Verwendung von Bibliotheken, die im Wesentlichen vordefinierte Funktionen und Klassen bereitstellen, die einen einfachen Zugriff darauf ermöglichen, was die Lesbarkeit des Codes und den Programmierprozess erheblich vereinfacht. Die Auswahl an Bibliotheken ist immens, aber besonders wichtig für das maschinelle Lernen sind NumPy für verbesserte und beschleunigte Verarbeitung verschiedener mathematischer Operationen einschliesslich Matrixmultiplikation, Matplotlib für die Visualisierung von Daten und Ergebnissen mithilfe verschiedener Grafiken und Diagramme sowie die Deep-Learning-Frameworks für die Implementierung der Machine-Learning-Modelle, ohne die es schwierig ist, Architekturen zu schreiben. Insgesamt können drei Frameworks hervorgehoben werden, nämlich TensorFlow, Keras und PyTorch, von denen jedes seine Vor- und Nachteile hat, aber im Bereich Computer Vision und Natural Language Processing hat sich aus vielen Gründen PyTorch an die Spitze gesetzt. [33]

Obwohl man theoretisch die gesamte Architektur in PyTorch ohne zusätzliche Bibliotheken schreiben kann und im Internet viele detaillierte Anleitungen zur Implementierung von Transformer- und GPT-Architekturen in PyTorch zu finden sind, die es definitiv wert sind, angesehen zu werden, gibt es tatsächlich eine Möglichkeit, die eine oder andere beliebte Architektur einfach zu verwenden, indem man sie aus der entsprechenden Bibliothek importiert. Da die Aufgabe dieser Arbeit das Training des Modells und nicht die Erstellung der Architektur selbst ist, wurde aus Zeitersparnisgründen beschlossen, die fertige GPT-Architektur zu verwenden, die bei Hugging Face verfügbar ist.

Die geeignete Architektur für die Aufgabe der Vorhersage des nächsten Zuges im Spiel war die GPT2LMHeadModel. Im Wesentlichen ist diese Architektur die grundlegendste und eignet sich für Aufgaben des Natural Language Processing. Von GPT2Model unterscheidet sie sich durch das Vorhandensein eines Language Modeling Head, der aus Linear und Softmax besteht und in Abbildung 15 dargestellt ist. Die Aufgabe dieses Teils, der die Architektur vor der Ausgabe der Ergebnisse abschliesst, besteht darin, die versteckten Zustände aus dem Basismodell zu erhalten und sie in Logits für jedes Token des Vokabulars zu transformieren, was es dem Modell ermöglicht, die Wahrscheinlichkeit des Auftretens jedes möglichen Tokens an der nächsten Position vorherzusagen.

Nach dem Laden dieser Architektur und des Tokenizers besteht die Möglichkeit, die Anzahl der Heads, die Anzahl der Schichten, die Anzahl der Embeddings und das maximale Datenvolumen zu konfigurieren, das das Modell in einer Sequenz berücksichtigen kann. Für das erste Training wurde

beschlossen, keine Parameter ausser der maximalen Länge zu ändern. Die maximale Länge wurde auf 130 reduziert, was mit Reserve die maximale Länge des Spiels und zusätzliche Anfangs- und End-Tokens berücksichtigte, die jeder Sequenz vorangestellt und angehängt wurden. Alle anderen Parameter wurden auf den Standardwerten belassen, auf denen OpenAI GPT-2 trainiert hat. Die Verwendung der Bibliothek Transformers ermöglicht es auch, die Anzahl der Epochen, die Häufigkeit der Logging, die Häufigkeit des Speicherns des Modells usw. festzulegen. Im Code wird auch eine Aufteilung in Trainings- und Validierungsdatensätze vorgenommen, die für die korrekte Bewertung des Modells erforderlich sind. Der Trainingsdatensatz dient als Lernmaterial für das Modell, und um sicherzustellen, dass das Modell nicht nur lernt, wie man genau diese Daten generiert, sondern auch neue generieren kann, bewerten wir das Modell am Ende jeder Epoche auf diesen während des Trainings unsichtbaren Daten. [34]

Aufgrund der geringen Datenmenge wurde zur Vermeidung einer zufälligen Batch-Grösse eine adaptive Batch-Grösse verwendet, die jedes Mal den aktuellen Stapel des vom Modell zu verarbeitenden Brettspiels berücksichtigte, so dass pro Iteration ein Spiel durch das GPT geleitet wurde.

6.4 Aufteilung in zwei Spieler

Indem man der einfachen Idee folgt, dass der gesamte Prozess des Modelltrainings darauf abzielt, den Agenten zu lehren, Daten aus dem Datensatz zu wiederholen und in Situationen mit bisher unbekannten Daten Züge zu machen, die den Zügen aus den Trainingsdaten ähneln, kann man zu dem Schluss kommen, dass es unmöglich ist, ein Modell auf Daten von Siegen sowohl des blauen als auch des roten Spielers zu trainieren und von ihm zu erwarten, dass es die stärkstmöglichen Züge macht. Daher wurde die logische Entscheidung getroffen, zwei Agenten auf zwei verschiedenen Datensätzen zu trainieren, die durch Aufteilung des allgemeinen Datensatzes in Spiele, in denen Blau gewinnt, und Spiele, in denen Rot gewinnt, entstehen. Auf diese Weise haben gewinnbringende und verlierende Züge weniger Konkurrenz untereinander, und die Generierung eines strategisch gewinnbringenden Zuges wird bei dem Modell wahrscheinlicher, das darauf trainiert wurde, in der Rolle der jeweiligen Farbe zu gewinnen. So wurde der Datensatz, der ursprünglich 25.000 Partien umfasste, nach der Sortierung der Textdatei für jedes der Modelle etwa halbiert.

6.5 Der erste Versuch

Das Training des ersten Modells mit der Architektur und den Standardparameterwerten dauerte nicht allzu lange, genauer gesagt etwa 3-4 Stunden. Wenn man jedoch berücksichtigt, dass die Anzahl der Epochen nur 10 betrug und der Datensatz lediglich etwa 12.000 Spiele umfasste, erhält man ein recht hohes Verhältnis der auf eine Iteration verwendeten Zeit. Der Code für das erste Training berechnete ausserdem den Verlust (Loss) für das Validierungsset nicht und speicherte ihn auch

nicht. Trotz aller Mängel war es jedoch notwendig, das trainierte Modell zu starten und seine Fähigkeiten zu testen. Es wurde ein kleines Skript für manuelle Tests entwickelt, das einen Checkpoint (so wird das Modell auch genannt) lud, eine Sequenz entgegennahm und seine Vorhersage für den nächsten Zug ausgab. Nach dem Start des Programms und der Generierung des ersten Zugs war es schwer, die Überraschung zu verbergen, denn trotz des unvollkommenen Trainingscodes sowie des hohen Loss-Wertes auf den Trainingsdaten (etwa 8, was im Vergleich zu den nachfolgenden Modellen enorm gross ist), schienen die Züge sinnvoll zu sein. Natürlich konnte der erste Versuch nicht so erfolgreich sein, und beim 11. Zug stiess man auf den ersten Mangel: einen sich wiederholenden Zug. Das heisst, das Feld, das das Modell besetzen wollte, war bereits belegt. Daher musste die Anzahl der angezeigten Vorhersagen von eins auf fünf erhöht werden, um im Test die Möglichkeit zu haben, den technisch wahrscheinlichsten Zug auszuwählen. Das zweite Problem waren Züge, die keinen Sinn ergaben. Obwohl das Modell in den meisten Fällen zu Beginn der Partie Anzeichen von Spielverständnis zeigte und trotz der Fähigkeit der Transformer-Modelle zu extrapolieren, verlor es nach einer bedingten Grenze von 20 Zügen die Orientierung und begann abwechselnd oder manchmal gleichzeitig Züge auf bereits besetzte Felder oder Züge, die offensichtlich zum Verlust führten, auszugeben. Die erste Erklärung, die bereits oben als mögliche Ursache für die Probleme genannt wurde, war ein zu kleiner Datensatz, aber das zweite Problem war die unzureichende Trainingszeit des Modells. Um es weiter zu trainieren, war eine enorme Menge an Zeit erforderlich, da der Loss-Wert in der Regel irgendwann beginnt, sich einem lokalen Minimum zu nähern, was die Rate der Verringerung der fehlerhaft vorhergesagten Züge verlangsamt.

6.6 Suche nach der optimalen Modellgrösse

Indem man verstand, dass die Anzahl der Parameter in der ursprünglichen GPT-Architektur höchstwahrscheinlich für monatelanges Training auf leistungsstarken Rechnern berechnet wurde, und indem man erkannte, dass für ein so kleines Vokabular, das aus lediglich 121 möglichen Zügen besteht, eine so grosse Anzahl von Parametern nur eine Behinderung im Training darstellen kann, wurde die Entscheidung getroffen, die Anzahl der Schichten, Köpfe und Embeddings zu reduzieren. Nach mehreren Versuchen wurde die optimalste Variante mit 128 Embeddings, 4 Köpfen und 4 Schichten gefunden. Mit diesen Einstellungen verringerte sich die Gesamtanzahl der Parameter des Modells um das Zehnfache, was es ermöglichte, die Grösse eines Checkpoints von 500 Megabyte auf 10 Megabyte zu reduzieren. Gleichzeitig beschleunigte sich der Trainingsprozess um das 50- bis 60-fache, und bereits nach 20 Minuten Training hatte das neue Modell einen geringeren Loss als der Checkpoint mit einer grösseren Anzahl von Parametern. Weitere Tests zeigten, dass die zusätzliche Reduzierung der Embeddings keine weiteren Verbesserungen im Loss brachte und auch keinen grossen Einfluss auf die Trainingsgeschwindigkeit hatte. Daher wurde beschlossen, die gleichen

Parameter beizubehalten. Ausserdem wurde ein Teil des Skripts überarbeitet, der für die Berechnung und Speicherung des Loss im Ordner logs verantwortlich war.

Die Modelle mit einem Gewicht von 10 Megabyte spielten besser als die grösseren Modelle, aber in vielen Situationen gelang es auch ihnen nicht, die Verbindung zwischen dem Spiel und dem logischsten Zug zu erkennen. Ein strategisch gewinnbringender Zug konnte nach 4 sinnlosen Zügen folgen, und obwohl dies seltener vorkam, gab es immer noch Zugwiederholungen.

In diesem Stadium wurde ein zusätzliches Problem identifiziert, nämlich dass am Anfang und am Ende des Spiels keine Start- und Endspiel-Tokens eingefügt wurden. Dadurch wurde der erste Zug völlig anders generiert, als er es statistisch hätte tun sollen. Nachdem dieser Fehler erkannt und behoben wurde, wurde der erste Zug tatsächlich korrekt generiert. Eine interessante Beobachtung war jedoch, dass sowohl der korrigierte als auch der fehlerhafte Code, wenn ihnen eine Zugsequenz (mehr als ein Zug) gegeben wurde, denselben Zug mit derselben Wahrscheinlichkeit ausgaben. Dies erklärt sich wahrscheinlich durch die dynamische Batchgrösse, die das Spiel immer von Anfang bis Ende nahm, was es ermöglichte, sich korrekt auf die Positions-Tokens für alle Positionen ausser der ersten einzustellen. Nach dieser Änderung wurde das Trainingsskript der Modelle nicht mehr verändert und nur noch für Inferenzzwecke in der weiteren Arbeit mit den Modellen verwendet.

6.7 Modelleleistungstest

In jeder Epoche speicherte der Code einen Checkpoint (also einen Snapshot) des Modells nach einer bestimmten Generierungsepoche. Nachdem zwei Modelle für die roten und blauen Spieler jeweils 60 Epochen trainiert worden waren, wurden 120 Checkpoints erhalten. Die Aufgabe bestand nun darin zu verstehen, welche der roten und welche der blauen Checkpoints am besten spielen. Ein Indikator für ein gutes Spiel könnte der Verlust (Loss) sein. Da die Spiele aus dem Trainingssatz jedoch keine Beispiele für eine ideale Strategie darstellten, ist es theoretisch möglich, dass ein Checkpoint mit einer besseren Strategie einen höheren Verlust aufweist als ein Checkpoint, der besser spielt. Dafür wurden Varianten von Leistungstests für die Modelle entwickelt. In erster Linie blieb in den Ordnern mit den Checkpoints jeder fünfte Checkpoint (ein Unterschied von weniger als 5 Epochen verändert das Modell nicht besonders kritisch) übrig, um Zeit beim Testen zu sparen. Die erste Variante, die zur Verwendung angenommen wurde, bestand darin, eine $N \times M$ -Tabelle zu erstellen, in der jedes Element N ein roter Agent und jedes Element M ein blauer Agent ist. Jeder rote Agent sollte gegen jeden blauen Agenten 121 Spiele mit jedem der 121 Anfangszüge spielen. Diese Methode zeigt nicht die tatsächliche Überlegenheit des ersten Spielers gegenüber dem zweiten oder umgekehrt, da der rote Spieler alle möglichen Züge im ersten Zug machen muss. Wenn wir jedoch die Anzahl der gewonnenen Spiele für jedes Modell einer Farbe vergleichen, erhalten wir recht

genaue Effizienzkennzahlen des Checkpoints im Vergleich zu anderen Checkpoints derselben Farbe. Zusätzlich zum Skript, das diese Matrix erstellt und sie füllt, indem es Spiele zwischen den Modellen spielt, wurde mit Hilfe von ChatGPT ein Skript erstellt, das die ELO-Wertung basierend auf der Matrix berechnete. Die zweite getestete Methode bestand darin, den Zugvorhersage-Output als Liste zu verwenden und die Züge entsprechend ihrer Wahrscheinlichkeiten aus der Liste zu wählen, so oft wie nötig. Diese Option ermöglichte eine grössere Anzahl von Spielen, auf deren Grundlage die Modellleistung beurteilt werden konnte. Da der Vergleich jedoch selbst bei 121 Zügen pro Zugpaar ressourcenintensiv war und zudem kleinere Schwierigkeiten beim Erstellen des Skripts für das zweite Szenario auftraten, wurde beschlossen, bei der ersten Variante zu bleiben und diese zu verwenden.

6.8 Das Konzept der Generationen

Angesichts der Tatsache, dass manchmal die tatsächlich besten Züge die zweiten oder dritten in der Wahrscheinlichkeitverteilung waren und nach weniger logischen Zügen folgten, entstand die Idee, ein Skript zu erstellen, das die beiden stärksten Modelle auswählt, die leicht mit den Skripten aus dem vorherigen Abschnitt ermittelt werden konnten, und anstatt immer die wahrscheinlichste Vorhersage des Modells zu wählen, die ersten zwei oder drei wahrscheinlichsten Züge auswählt. Als Grundlage für dieses Skript diente das Skript, das zur Erstellung der Matrizen verwendet wurde. Die Parameter, die im Skript geändert werden konnten, waren zwei Modelle, die gegeneinander spielen sollten, und eine Liste, die mit Zahlen gefüllt werden konnte. Die Zahl n an der Stelle g in der Liste bedeutet, dass beim Zug g die n wahrscheinlichsten Züge ausgewählt und mit allen von ihnen ein Spiel gespielt werden müssen. Wenn man in die Felder der Matrix Zahlen ausser eins einträgt, wächst die Anzahl der zu generierenden Spiele mit jedem Zug exponentiell. Eine detaillierte Umsetzung, einschliesslich der Überprüfung der Spiele auf den Gewinner, wird im Code beschrieben.

Die Idee hinter diesem Code besteht darin, dass, wenn man eine grosse Anzahl von Spielen mit Variationen der Züge für einen der Spieler mithilfe einer Verzweigungsliste wie [1, 2, 1, 2, 1, 2, 1, 2...] generiert, man das Modell anschliessend weitertrainieren (sogenanntes Fine-Tuning) kann anhand der Spiele, in denen es durch solche Verzweigungen gewonnen hat. Dadurch erhalten wir ein Modell, das Fortschritte macht, weil es mehr Beispiele sehen kann, in denen es den Sieg davonträgt. Dieses Modell, das über ein grosses Wissen über das Spiel verfügt, tritt in die nächste Generation von Modellen ein, und indem man so die Agenten trainiert, die für beide Seiten spielen, erhalten sie ein immer besseres Verständnis des Spiels, bis schliesslich die Generationen eine optimale Strategie erreichen.

Ursprünglich war geplant, diesen Algorithmus zur Generierung von zwei Millionen Spielen innerhalb weniger Stunden zu verwenden, aber in der Praxis erwies sich das Skript als nicht so schnell wie erwartet, und die Generierung von **519.288 Spielen** mit Möglichkeiten für **19 Verzweigungen** dauerte **bis zu 9 Stunden**. Von der Gesamtzahl erwiesen sich etwa 1/5 als Spiele, in denen der gewünschte Spieler gewann, und auf diesen 1/5 Daten wurde das Fine-Tuning durchgeführt. Das Training des Modells des Gegners auf 4/5 der Daten hätte kein Ergebnis gebracht, weil alle Züge des Gegners ohne Verzweigungen generiert wurden, weshalb das Modell gelernt hätte, Vorhersagen zu bevorzugen, die ohnehin Top-Vorhersagen sind.

Aus dem Verhältnis von Siegen zu Niederlagen lässt sich schliessen, dass ein geringerer Teil der Verzweigungen tatsächlich zum Sieg führt und qualitative Verbesserungen darstellt, weshalb es wahrscheinlich weniger sinnvoll ist, Züge für beide Spieler gleichzeitig zu verzweigen, als für jeden einzeln (paarweise Züge für den blauen Spieler, unpaarweise Züge für den roten Spieler).

6.9 Misserfolg und Analyse

Nach der Generierung einer neuen Generation stellte sich heraus, dass sie tatsächlich bessere Ergebnisse in den Spielen erzielte, und die folgende Generation war ebenfalls besser als die vorherige. Allerdings erschien der Agent, wenn er in einem echten Spiel gegen einen Menschen getestet wurde, dümmer als der Agent der ersten Generation. Der Agent verstand nicht besonders gut, was das Ziel des Spiels war, was bei der ersten Generation sehr deutlich zu erkennen war. Anstatt zu versuchen, das Spiel zu beenden und den Sieg zu erringen, erinnerten sich die Züge eher an zufällig generierte.

Nachdem eine Vergleichsmatrix der Modelle erstellt wurde, die alle drei Generationen umfasste, ergab sich alles an seinem Platz. Die zweite Generation war besser als die erste, die dritte Generation war besser als die zweite, verlor jedoch gleichzeitig gegen die erste Generation. Ein solches Verhalten könnte darauf zurückzuführen sein, dass ursprünglich versucht wurde, nicht den Algorithmus, sondern das Sprachmodell zu übertreffen, das nicht genügend Beispiele gesehen hatte, um Fehler zu vermeiden. Im Falle eines Algorithmus besteht der einzige Weg, besser zu spielen, darin, einen besseren Algorithmus zu finden, was im Falle der Verzweigung von Zügen durchaus möglich wäre.

Wenn jedoch ein Modell, das die Logik des Spiels nicht vollständig verstanden hat, versucht, ein anderes solches Modell zu übertreffen, anstatt nach starken Zügen zu suchen, die helfen, die Taktik des Gegners zu übertreffen, ist es viel einfacher, Züge zu finden, auf die das gegnerische Modell mit schwachen oder unlogischen Zügen reagiert. Das heisst, unter allen Spielen, in denen der Agent

gewinnt, wird die Anzahl der Spiele, in denen der Agent tatsächlich gut spielt, im Vergleich zu der Vielzahl von Spielen, in denen der Gegner dumme Züge generiert, unverhältnismässig gering sein.

6.10 Fine-tuning an den ursprünglichen Daten

Viele Tage wurden darauf verwendet, zunächst eine logische Erklärung für den misslungenen Versuch zu finden und dann über mögliche Wege nachzudenken, wie dieser Fehlschlag behoben werden kann. Man kann mit Sicherheit sagen, dass kurze Spiele die besten Beispiele für das Training sind, während lange Spiele, obwohl sie im Datensatz vorhanden sein sollten, mehr Entropie enthalten und nicht optimal für das Training sind. Dies führt zu der Überlegung, ein schrittweises Fine-Tuning durchzuführen, bei dem das Modell mit kurzen Sequenzen beginnt und schrittweise zu längeren übergeht, wodurch es die notwendigen Parameter anpasst, um das Spiel mit optimalen Zügen zu spielen. Eine solche Datenverarbeitung und das anschliessende schrittweise Fine-Tuning würden dem Modell eindeutig Vorteile gegenüber Modellen bieten, bei denen das Fine-Tuning ohne vorherige Datenverarbeitung durchgeführt wird. Die Frage ist, ob dies helfen würde, die im vorherigen Abschnitt dargestellten Probleme zu überwinden.

Eine unerwartete Lösung des Problems war ein zusätzliches Fine-Tuning auf den ursprünglichen 12.000 Spielen, die von Websites erhalten wurden. Interessant ist auch der Rückgang der Loss-Werte beim Fine-Tuning auf Werte, die beim ersten Training nicht beobachtet wurden, was theoretisch das Vorhandensein eines lokalen Minimums bedeuten kann. In diesem Fall verschiebt das erste Fine-Tuning auf der Basis neuer Daten den Gradient Descent und ermöglicht es, ein neues lokales Minimum zu finden. Ein solches Verfahren, bei dem die besten roten und blauen Modelle einer bestimmten Generation verwendet werden, um neue Spiele mit Verzweigungen zu generieren, gefolgt von deren Sortierung und Verwendung für das Fine-Tuning dieser besten Modelle und einem abschliessenden Fine-Tuning auf den ursprünglichen Daten (12000 Spiele) über drei Generationen hinweg, führte zu einer Leistungsverbesserung des Modells.

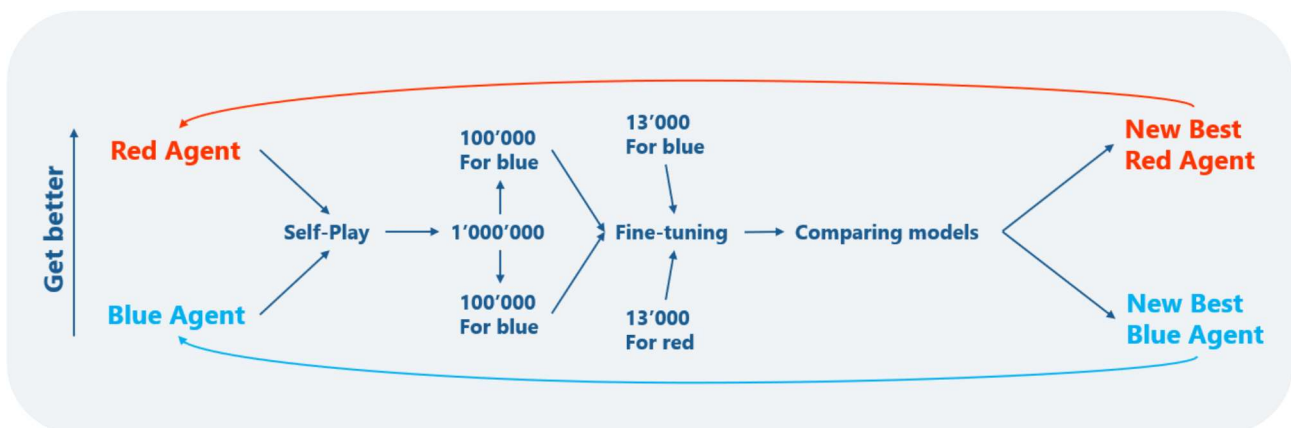


Abbildung 16: Iterativer Modellverbesserungsprozess (eigene Darstellung)

6.11 GUI zum Spielen von Hex

Der letzte Schritt bestand darin, eine visuelle Benutzeroberfläche mit Gewinnerüberprüfung zu erstellen, die es ermöglicht, direkt gegen die besten Checkpoints des roten und des blauen Spielers zu spielen. Dafür wurde Flask verwendet, um einen lokalen Server zu erstellen und problemlos die Generierung des nächsten Zuges basierend auf der aktuellen Zugsequenz anzufordern. Tatsächlich begann die Entwicklung der Website noch bevor die Modelle trainiert wurden, aber ohne korrekt funktionierende Modelle war die Schnittstelle nicht sehr nützlich. Die im Code integrierten Überprüfungen wählen den nächsten möglichen Zug aus, wenn der vorgeschlagene Zug bereits besetzt ist oder ein ungültiges Token darstellt.

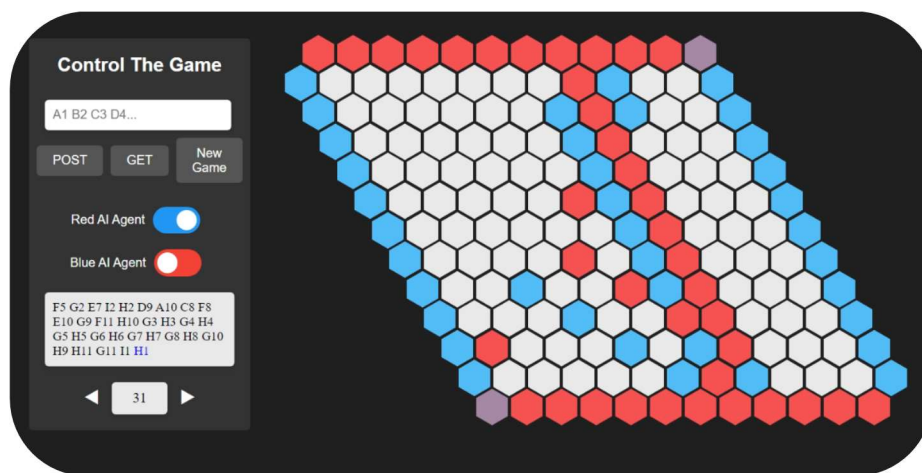


Abbildung 17: Website-Interface zum Spielen von Hex (eigene Darstellung)

7 Ergebnisse

Das Ergebnis dieses Projekts waren vier erfolgreich trainierte Generationen von Modellen, einschliesslich der ursprünglichen Generation, die auf dem anfänglichen Spieldatensatz trainiert wurde. Jede Generation umfasst zwei Pakete von Modellen: eines für den roten und eines für den blauen Spieler. Abhängig von der Generation sind für jeden Spieler zwischen 6 und 12 Checkpoints gespeichert. Um die besten Modelle der ersten Generation zu bestimmen, werden alle Checkpoints der ersten Generation in Spielen gegen sich selbst getestet. Die Top-3 Checkpoints jeder Farbe gelangen in eine Vergleichstabelle der Modelle, und alle Checkpoints der zweiten Generation werden mit ihnen verglichen. Diejenigen Checkpoints der zweiten Generation, die gegen die der ersten Generation am besten abschneiden, gelangen ebenfalls in die Vergleichstabelle und treten wiederum in den Vergleich mit der dritten Generation ein. So erreicht die Vergleichstabelle für vier Generationen eine Grösse von 12 mal 12, und darin werden abschliessende Tests der Modelle durchgeführt, um die Beziehungen zwischen den Siegen und Niederlagen jedes einzelnen zu verstehen. Letztendlich

werden nach dem Test die Daten von einem Code verarbeitet, der jedem Modell einen ELO-Rating zuweist. Der endgültige ELO-Rating sieht wie folgt aus:

```
Top Red Checkpoints by Elo Rating:
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_4_checkpoint-765960_ft_r: Elo 1009.70
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_4_checkpoint-702130_ft_r: Elo 1007.90
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_3_checkpoint-765960_ft_gen_3_r: Elo 999.45
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_4_checkpoint-574470_ft_r: Elo 999.16
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_3_checkpoint-638300_ft_gen_3_r: Elo 998.56
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_2_checkpoint-510640_ft_gen_2_r: Elo 995.46
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_2_checkpoint-319150_ft_gen_2_r: Elo 993.97
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_3_checkpoint-510640_ft_gen_3_r: Elo 989.33
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_2_checkpoint-446810_ft_gen_2_r: Elo 988.57
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_1_checkpoint-765960_r: Elo 977.54
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_1_checkpoint-638300_r: Elo 968.30
C:\Users\Timur\Downloads\hex_agony\mini_red_test\gen_1_checkpoint-446810_r: Elo 965.03

Top Blue Checkpoints by Elo Rating:
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_3_checkpoint-589095_ft_gen_3_b: Elo 1025.73
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_4_checkpoint-589095_ft_b: Elo 1022.38
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_4_checkpoint-392730_ft_b: Elo 1022.27
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_3_checkpoint-654550_ft_gen_3_b: Elo 1020.76
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_3_checkpoint-392730_ft_gen_3_b: Elo 1020.72
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_2_checkpoint-523640_ft_gen_2_b: Elo 1019.75
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_4_checkpoint-720005_ft_b: Elo 1016.87
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_2_checkpoint-392730_ft_gen_2_b: Elo 1012.94
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_2_checkpoint-261820_ft_gen_2_b: Elo 1003.67
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_1_checkpoint-523640_b: Elo 979.22
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_1_checkpoint-654550_b: Elo 978.92
C:\Users\Timur\Downloads\hex_agony\mini_blue_test\gen_1_checkpoint-392730_b: Elo 978.90
```

Abbildung 18: Ergebnisse des Vergleichs der besten Modelle der verschiedenen Generationen (eigene Darstellung)

In manuellen Tests zeigen die Modelle gute Ergebnisse, und wenn man gegen sie mit einer Strategie spielt, scheint es, dass das Modell die Strategie tatsächlich versteht. Allerdings besteht die Schwäche der Checkpoints weiterhin darin, dass sie nicht verstehen, wie sie auf unlogische Züge reagieren sollen, die im Datensatz überhaupt nicht vorkommen. Das heisst, der beste Weg, diese Modelle zu besiegen, besteht darin, seltsame Strategien zu spielen oder zufällig zu ziehen, bis das Modell nicht mehr versteht, wie es reagieren soll. Ausserdem wurde ein kleiner manueller Vergleich von `gen_4_checkpoint-765960_ft_r` mit einem Referenz-KI-Modell auf der Website durchgeführt, und in den meisten Fällen werden nicht nur Züge, sondern auch ganze Spiele von beiden Modellen identisch generiert, wie zum Beispiel in dem Spiel C4 F6 G6 G5 I4 J1 D7 C10 B10 E5 C6 C5 D5 E3 D4 D2 B3 C1 A2 B9 C9 D6 C7 C8 D8 C3 B4 D3 C2 D1 B2 B1 A1 B11 A11.

Beim Betrachten der Elo-Tabelle wird deutlich, dass kein einziges der blauen Modelle der vierten Generation das Modell der dritten Generation übertroffen hat. Das bedeutet nicht, dass das Modell eine Regression durchläuft, sondern vielmehr, dass sich ein Muster abzeichnet, bei dem ein Modell einer bestimmten Generation Fortschritte macht, während das andere zurückfällt. Bereits in der dritten Generation wurde festgestellt, dass das Modell `gen_2_checkpoint-382980_ft_gen_2_r` besser spielt als jeder Checkpoint aus der dritten Generation (obwohl die aktuelle Elo-Tabelle dies nicht direkt widerspiegelt, zeigt der direkte Vergleich der roten Checkpoints der dritten Generation mit dem oben genannten Checkpoint der zweiten Generation einen Vorteil für das Modell der zweiten Generation, auch wenn die Checkpoints der dritten Generation im Gesamten besser erscheinen als

die der zweiten Generation. Dieser Umstand weist auf die Unvollkommenheit der in dieser Arbeit verwendeten Vergleichsmethoden hin).

Gleichzeitig zeigten die blauen Modelle der dritten Generation im direkten Vergleich zur zweiten Generation bessere Ergebnisse. Nach dem Prinzip, die stärksten Checkpoints zur Erstellung der vierten Generation auszuwählen, wurde beschlossen, das zweite Generation der roten Modelle und die dritte Generation der blauen Modelle zu verwenden. Auf diese Weise spielte der blaue Agent, der stärkere Spieler, gegen den roten Agenten, den schwächeren Spieler. Während der rote Agent lernte, komplexere Mechanismen zu besiegen, konnte der blaue Agent sich erlauben, weniger optimale Züge zu machen und trotzdem zu gewinnen.

Sollte das oben Beschriebene zutreffen, dann ist zu erwarten, dass, wenn man die stärksten verfügbaren Checkpoints zur Erstellung einer fünften Generation heranzieht, es zu einer Verschlechterung der roten Checkpoints der fünften Generation und zu einer Verbesserung der blauen Checkpoints der fünften Generation kommen wird.

8 Weiterentwicklung

Einer der wichtigsten Teile der Projektverfeinerung ist das Testen der trainierten Modelle gegen den externen Algorithmus, der für die Beendigung der Spiele verwendet wurde. Die Auswertung verwendet zwei Testfälle: 500 Eröffnungsproben von einem der trainierten Modelle und 625 Eröffnungen ausserhalb der Distribution von einer externen Open-Source-Baseline (<https://github.com/harbecke/HexHex>), um die Generalisierung zu beweisen, wobei die Modelle gegen den Algorithmus antreten.

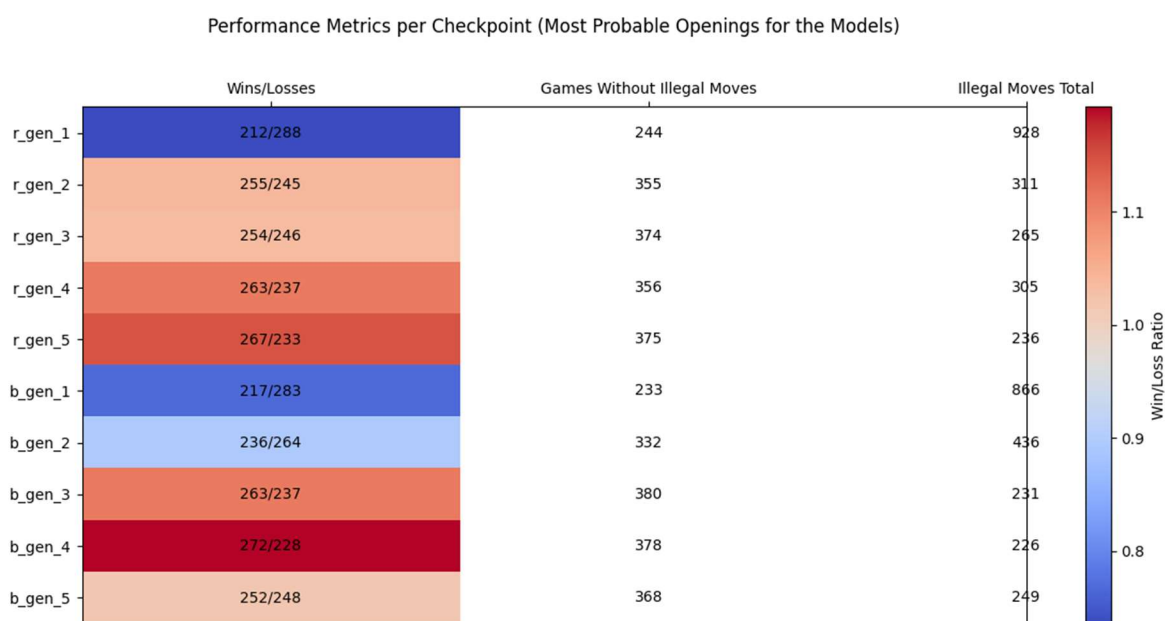


Abbildung 19: Erzeugungsraten in Spielen mit Eröffnungen, die den ursprünglichen Daten ähnlich sind

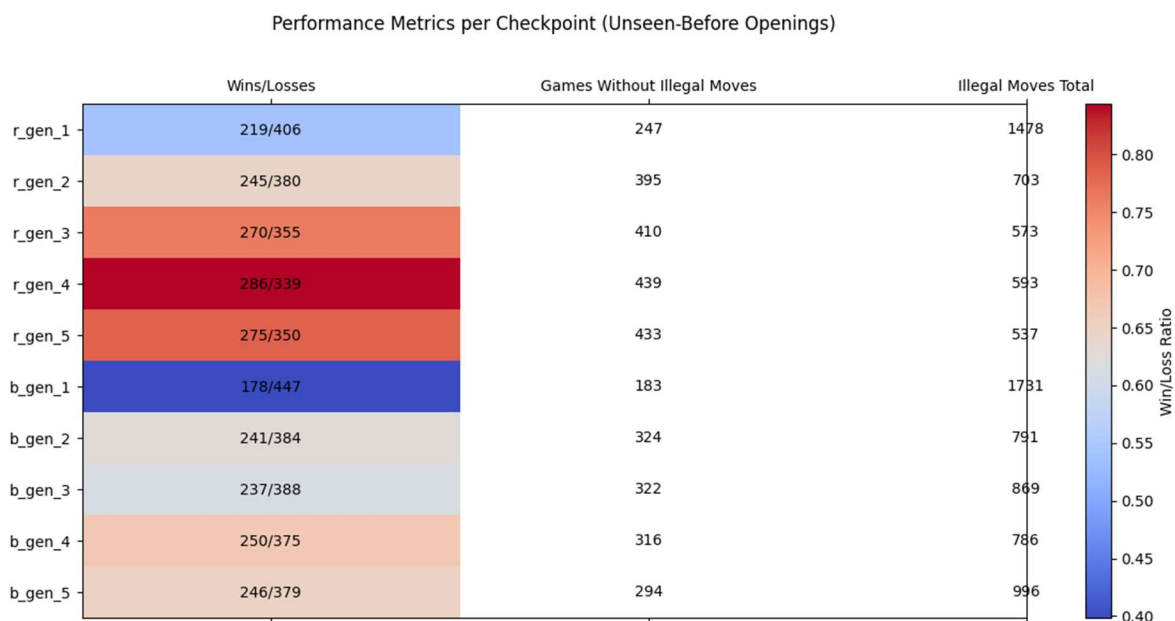


Abbildung 20: Erzeugungsraten in Spielen mit Eröffnungen, die die Modelle noch nie gesehen haben

Die Testergebnisse demonstrieren, dass der iterative Fine-Tuning-Ansatz, umgesetzt über fünf aufeinanderfolgende Iterationen, die Leistung sowohl bei bekannten als auch bei unbekannten Daten signifikant verbessert. Bei den bekannten Eröffnungen steigt die Gewinnrate des roten Modells von 42,4 % (212/500) am ersten Checkpoint auf 53,4 % (267/500) in der letzten Iteration, während die Anzahl der unerlaubten Züge nahezu fünfmal von 928 auf 236 sinkt. Entsprechend verbessern sich die blauen Modelle von 43,4 % auf 54,4 % und reduzieren unerlaubte Züge von 866 auf 226.

Bei unbekannten Daten steigt die Gewinnrate des roten Modells von etwa 35 % (219/625) in der ersten Generation auf rund 44 % (275/625) in der fünften Generation, wobei die unerlaubten Züge von 1478 auf 537 abnehmen. Das blaue Modell zeigt einen ähnlichen Trend: Die Gewinnrate erhöht sich von ungefähr 28,5 % auf etwa 39–40 % in den Generationen 4–5, und die unerlaubten Züge sinken von 1731 auf unter 1000.

Obwohl es zwischen den Iterationen zu leichten Schwankungen kommt, ist der Gesamteffekt eine deutliche Verbesserung sowohl der Gewinnrate als auch der Legalität der Züge. Bemerkenswert ist, dass der Anteil der Spiele, die ohne unerlaubte Züge generiert werden, von etwa 50 % in den Anfangsmodellen auf fast 70 % in der fünften Generation ansteigt.

9 Diskussion

Die unbestrittene Errungenschaft dieser Arbeit ist die Methode der Ausbildung von Modellen auf den von ihnen erzeugten Daten und die Idee der Verwendung von Generationen. Die Schwäche dieser Arbeit lediglich ein intuitives Verständnis und eine Interpretation dessen, was im Trainingsprozess geschieht. In dem Experiment konnte gezeigt werden, dass Modelle bis zu einem gewissen Grad die Logik des Spiels bedienen können. Im Allgemeinen wurde beobachtet, dass die Modelle gut abschnitten, wenn sie gegen Agenten mit der gleichen Logik spielten, die in den Trainingsdaten verwendet wurde. Eine kleine Änderung in der Strategie des Gegners ist für das Modell schwer zu verstehen und führt zu möglichen Fehlern bei der Ermittlung des besten Zuges. Ein Problem, das eine Stärke von Transformer bei der Texterstellung ist, aber verhindert, dass er bei der Erstellung von Spielen die gewünschten Ergebnisse erzielt, ist die Bedeutung der Reihenfolge der Chips vor dem zu erstellenden Zug. Das Modell achtet sehr stark auf die Positionen bestimmter Züge und nicht auf die Position des Brettes. Daher sind die letzten Züge für das Modell wahrscheinlich wichtiger als die Züge zu Beginn, und das Modell wird eher auf der Grundlage der letzten Züge handeln.

Zurückkehrend zur Variante des Spiels Dark Hex möchte ich sagen, dass es theoretisch durchaus möglich ist, ein transformerbasiertes Modell nur auf der Grundlage der eigenen Züge zu trainieren, die letztendlich den Agenten zum Sieg führen. Das weitere Spielen der Modelle gegeneinander kann mit Python-Skripten realisiert werden sowie durch die Anwendung eines leicht modifizierten Skripts zur Generierung neuer, einzigartiger Spiele. Aus Beobachtungen beim Spielen gegen einen Gegner, dessen Züge keine besondere Logik haben und nicht taktisch sind, fällt auf, dass das Modell nicht versteht, wie man solche Sequenzen berechnet, und nicht besonders sicher ist, welchen richtigen Zug es machen soll. Das Verstecken der Züge des Gegners könnte helfen, das Problem der Reaktion auf unsinnige Züge zu vermeiden, weil es intuitiv scheint, dass, wenn das Modell lernt, intelligente Gegner zu besiegen, ohne deren Züge zu sehen, es keine Probleme im Spiel gegen sinnlose Züge haben sollte.

10 Schlusswort

Diese Arbeit war eine unglaubliche Gelegenheit, die Welt der KI, neue Konzepte, Architekturen und Bibliotheken kennenzulernen.

Die Arbeit an dieser Maturaarbeit ermöglichte mir einen tiefen Einblick in die faszinierende Welt der Künstlichen Intelligenz und der transformerbasierten Modelle. Von den theoretischen Grundlagen über die Struktur und Funktionsweise neuronaler Netzwerke bis hin zur praktischen Umsetzung eines Hex-Spielagenten war der Lernprozess reich an Herausforderungen, aber ebenso belohnend. Insbesondere die Vielseitigkeit der Transformer-Architektur und ihr Potenzial, Muster und Strategien auch in unerwarteten Kontexten wie Brettspielen zu erkennen, haben mich begeistert.

Rückblickend auf die Projektarbeit bin ich froh, dass ich dieses Thema gewählt habe, denn was in der Arbeit beschrieben ist, werde ich weiterführen, um weitere Ergebnisse zu erzielen.

Ich möchte mich bei meinem Betreuer Emil Müller dafür bedanken, dass er mir geholfen und die Richtung vorgegeben hat. Dank ihm wurde diese Arbeit mit Ideen gefüllt und ich war voller Motivation, dieses Projekt durchzuführen.

Vielen Dank auch an Janis Steiner, der einige Seiten meiner Arbeit in letzter Minute durchgesehen und mich auf sprachliche Fehler hingewiesen hat.

Im Rahmen meiner Teilnahme am Wettbewerb „Schweizer Jugend Forscht“ möchte ich mich auch bei dem zugeteilten Experten Jakub Adamek bedanken, der mir bei der Fertigstellung dieser Arbeit sehr geholfen hat und mir geholfen hat, die Schwächen und Stärken meiner Arbeit zu finden.

Für diese Maturaarbeit wurde der Online-Übersetzer DeepL aktiv zur Übersetzung ins Deutsche verwendet. ChatGPT wurde zur Korrektur der verfassten Texte, für Feedback, zur Erstellung und Analyse von Code sowie zur Beantwortung von Fragen genutzt, die mir klar werden mussten.

11 Inhaltsverzeichnis

- [1] «Profile der Zukunft. Über die Grenzen des Möglichen - Clarke, Arthur C.: 9783453019058 - ZVAB». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.zvab.com/9783453019058/Profile-Zukunft-Grenzen-M%C3%B6glichen-Clarke-3453019059/plp>
- [2] «Emergent tool use from multi-agent interaction | OpenAI». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://openai.com/index/emergent-tool-use/>
- [3] P. Verma und W. Oremus, «AI voice clones mimic politicians and celebrities, reshaping reality», *Washington Post*, 13. Oktober 2023. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.washingtonpost.com/technology/2023/10/13/ai-voice-cloning-deepfakes/>
- [4] «Sora | OpenAI». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://openai.com/index/sora/>
- [5] «How BERT and GPT models change the game for NLP | IBM». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.ibm.com/think/insights/how-bert-and-gpt-models-change-the-game-for-nlp>
- [6] «Parker Brothers», *Wikipedia*. 2. Oktober 2024. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Parker_Brothers&oldid=1248947873
- [7] «History of Hex - HexWiki». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.hexwiki.net/index.php/History_of_Hex
- [8] «The game of Hex». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.maths.ed.ac.uk/~csangwin/hex/index.html>
- [9] «Swap rule - HexWiki». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.hexwiki.net/index.php/Swap_rule
- [10] «SP.268 - Coursenotes». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://web.mit.edu/sp.268/www/coursenotes.html>
- [11] «Board size - HexWiki». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.hexwiki.net/index.php/Board_size
- [12] «Strategy-stealing argument», *Wikipedia*. 27. Februar 2024. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Strategy-stealing_argument&oldid=1210594970

- [13] «Hex (board game)», *Wikipedia*. 4. Oktober 2024. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: [https://en.wikipedia.org/w/index.php?title=Hex_\(board_game\)&oldid=1249396697](https://en.wikipedia.org/w/index.php?title=Hex_(board_game)&oldid=1249396697)
- [14] «Dark Hex: A Large Scale Imperfect... | ERA». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://era.library.ualberta.ca/items/aa9475b5-8125-4eeb-baa0-f23bb21c53ad>
- [15] «How Transformers Work: A Detailed Exploration of Transformer Architecture». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.datacamp.com/tutorial/how-transformers-work>
- [16] «[1706.03762] Attention Is All You Need». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://arxiv.org/abs/1706.03762>
- [17] «Tokenizer». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://huggingface.co/docs/transformers/en/main_classes/tokenizer
- [18] T. Gumpions, «Transformer Architecture Simplified», Medium. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://medium.com/@tech-gumpions/transformer-architecture-simplified-3fb501d461c8>
- [19] «Figure 7. Attention matrix visualization: (a) weights in BERT Encoding...», ResearchGate. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.researchgate.net/figure/Attention-matrix-visualization-a-weights-in-BERT-Encoding-Unit-Entity-BERT-b_fig5_359215965
- [20] M. Sharma, «What is Add & Norm, as soon as possible?», Medium. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://molgorithm.medium.com/what-is-add-norm-as-soon-as-possible-178fc0836381>
- [21] «Batch vs Layer Normalization - Zilliz blog». Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://zilliz.com/blog/layer-vs-batch-normalization-unlocking-efficiency-in-neural-networks>
- [22] «Fully Connected Layer vs Convolutional Layer: Explained», Built In. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://builtin.com/machine-learning/fully-connected-layer>
- [23] S. Herath, «The Feedforward Network (FFN) in The Transformer Model», Data Science and Machine Learning. Zugriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://medium.com/image-processing-with-python/the-feedforward-network-ffn-in-the-transformer-model-6bb6e0ff18db>

- [24] S. R. PhD, «Understanding and Coding Self-Attention, Multi-Head Attention, Cross-Attention, and Causal-Attention in LLMs». Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention>
- [25] S. R. PhD, «Understanding and Coding Self-Attention, Multi-Head Attention, Cross-Attention, and Causal-Attention in LLMs». Zugegriffen: 31. Oktober 2024. [Online]. Verfügbar unter: <https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention>
- [26] H. Sharma, «Softmax Temperature», Medium. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://medium.com/@harshit158/softmax-temperature-5492e4007f71>
- [27] Conic, «Answer to «What is the role of temperature in Softmax?»», Cross Validated. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://stats.stackexchange.com/a/527081>
- [28] «The need for sampling temperature and differences between whisper, GPT-3, and probabilistic model's temperature», Shivam Mehta. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://shivammehta25.github.io/posts/temperature-in-language-models-open-ai-whisper-probabilistic-machine-learning/>
- [29] «Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy». Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning>
- [30] Andrej Karpathy, *The spelled-out intro to neural networks and backpropagation: building micrograd*, (17. August 2022). Zugegriffen: 29. Oktober 2024. [Online Video]. Verfügbar unter: <https://www.youtube.com/watch?v=VMj-3S1tku0>
- [31] «Figure 1. Illustration of batch size, iteration, and epoch.», ResearchGate. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.researchgate.net/figure/Illustration-of-batch-size-iteration-and-epoch_fig1_378880342
- [32] «Intramodal self-Attention unit», ResearchGate. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://www.researchgate.net/figure/Intramodal-self-Attention-unit_fig4_373917749
- [33] V. Yadoshchuk, «Why Use Python for AI and Machine Learning», Waverley. Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: <https://waverleysoftware.com/blog/python-for-ai-and-ml/>
- [34] «OpenAI GPT2». Zugegriffen: 29. Oktober 2024. [Online]. Verfügbar unter: https://huggingface.co/docs/transformers/en/model_doc/gpt2#transformers.GPT2LMHeadModel.config

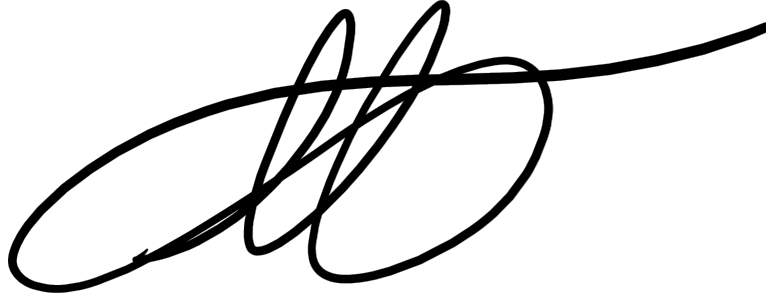
12 Abbildungsverzeichnis

Abbildung 1: Titelbild (generiert von Dall-E), [1].....	1
Abbildung 2: Vergleich von zwei Swap Regeln (eigene Darstellung).....	8
Abbildung 3: Koordinaten auf dem Brett (eigene Darstellung).....	9
Abbildung 4: Aufbau des Transformers [16]	13
Abbildung 5: Beispiel für Token-Embedding (eigene Darstellung)	16
Abbildung 6: Addition von Kodierungen und Einbettungen (eigene Darstellung)	17
Abbildung 7: Visualisierung der Kopferfassung bei Multi-Head Attention (eigene Darstellung).....	19
Abbildung 8: visuelle Darstellung der Score-Matrix [19]	20
Abbildung 9: Multi-Head Score-Matrizen für ein Wort [16].....	21
Abbildung 10: zwei Möglichkeiten zur Darstellung einer vollständig verbundenen Schicht [22]....	23
Abbildung 11: Masked Multi-Head Attention Score Matrix [25].....	25
Abbildung 12: Abhängigkeit der Wahrscheinlichkeiten von der Temperatur [28]	26
Abbildung 13: visuelle Darstellung der Batches [31, S. 1].....	28
Abbildung 14: vereinfachte GPT-Struktur [32]	29
Abbildung 15: detaillierte Struktur der GPT-Architektur [32]	29
Abbildung 16: Iterativer Modellverbesserungsprozess (eigene Darstellung).....	39
Abbildung 17: Website-Interface zum Spielen von Hex (eigene Darstellung).....	40
Abbildung 18: Ergebnisse des Vergleichs der besten Modelle der verschiedenen Generationen (eigene Darstellung).....	41
Abbildung 19: Erzeugungsraten in Spielen mit Eröffnungen, die den ursprünglichen Daten ähnlich sind	42
Abbildung 20: Erzeugungsraten in Spielen mit Eröffnungen, die die Modelle noch nie gesehen haben	43

13 Selbständigkeitserklärung

Ich habe diese Maturaarbeit unter Benützung der angegebenen Quellen selbständig entworfen, abgefasst, gestaltet und geschrieben.

Wattwil, 30.10.2024

A stylized, handwritten signature in black ink. It features a large, sweeping loop on the left, followed by several smaller, more intricate loops and a long, horizontal stroke extending to the right.

Tymur Haivoronskyi

14 Anhang

Der Grossteil des in dieser Arbeit verwendeten Codes ist in Google Drive unter dem Link https://drive.google.com/drive/folders/1CrDiHGF984xFBW-dmHZLkH5qP-IGRVF3?usp=drive_link zu finden.

Wenn Sie weitere Informationen wünschen oder die Verfügbarkeit anderer Teile des Codes oder anderer Checkpoints prüfen möchten, schreiben Sie bitte an die folgende E-Mail-Adresse:

timurgavarun@gmail.com

oder

tymur.haivoronskyi@kantiwattwil.ch