

**BABES-BOLYAI UNIVERSITÄT CLUJ-NAPOCA
FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
STUDIENGANG: Informatik in deutscher Sprache**

BACHELORARBEIT

Robotersteuerung mit Large Language Models: Eine Machbarkeitsstudie

Betreuer: Conf. Dr. Christian Săcărea

Verfasser: Bogdan Oprisiu

2025

UNIVERSITATEA BABES-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA : Informatică în limba germană

LUCRARE DE LICENȚĂ

**Controlul roboților cu ajutorul
modelelor mari de limbaj: Un studiu de
caz**

Conducător științific: Conf. Dr. Christian Săcărea

Absolvent: Bogdan Opris

2025

Abstract

Large Language Models (LLMs) are playing an increasingly important role in modern artificial intelligence—particularly in the field of robotics. This Bachelor’s thesis explores to what extent a compact, specialized language model can be used for efficient robot control via natural language. The goal is to automatically translate natural language commands into a standardized JSON format that the robot can then process.

To achieve this, a custom dataset was created and a compact language model was trained on commodity GPUs as well as on Google Colab. Techniques such as *Knowledge Distillation* and *Quantization* were applied to dramatically reduce resource consumption. Currently, the model runs on an external computer that is connected via cable to an Arduino, which interprets and executes the control commands directly on the robot. Full autonomy—by deploying the model on a Raspberry Pi as an onboard controller—is left for future work.

For practical evaluation, an independent robot prototype was designed and built entirely by the author, from CAD in Fusion 360 through 3D printing to wiring. The results demonstrate that even compact LLMs can reliably and flexibly drive robot control, opening up new prospects for future developments in autonomous, resource-efficient robotics applications.

Inhaltsverzeichnis

1 Einleitung	4
2 Theoretische Grundlagen	6
2.1 Grundlagen des maschinellen Lernens	6
2.1.1 Gradientenabstieg (Gradient Descent)	7
2.1.2 Backpropagation	8
2.2 Verlustfunktionen (Loss Functions)	9
2.2.1 Kreuzentropieverlust (Cross Entropy Loss)	9
2.2.2 Kullback-Leibler-Divergenz (KLDiv Loss)	10
2.2.3 Kombination der Verlustfunktionen	11
2.3 Grundlagen der Quantisierung	12
2.3.1 Arten von Quantisierung	13
2.3.2 Implementierung der dynamischen Quantisierung	13
2.4 Distillation	14
2.5 Moderne NLP-Methoden: Transformers und Large Language Models	15
2.5.1 Einordnung von NLP und der Wandel durch Transformer .	15
2.5.2 Einführung in LLMs	15
2.5.3 Grundlagen der Transformer-Architektur	16
2.6 Komponenten der Transformer-Architektur	17
2.6.1 Aufmerksamkeitsmechanismus (Attention)	18
2.6.2 Feed-Forward-Netzwerk (FFN)	19
2.6.3 Positionelle Kodierung (Positional Encoding)	19
2.7 Tokenisierung	21
2.7.1 Zeichenbasierte vs. wortbasierte Tokenisierung	22
2.7.2 Byte Pair Encoding (BPE)	22
2.8 LoRA (Low-Rank Adaptation)	23
3 Praktische Umsetzung des Roboters	25
3.1 Elektrisches System	25
3.1.1 Verkabelung und Komponentenverbindung	26
3.2 Mechanische Konstruktion	26

3.2.1	Chassis & Innenstruktur	27
3.2.2	Gedruckte Module für Elektronik & Strom	28
3.3	Sensorik (geplant)	29
3.4	Objekterkennung (Future Work)	29
3.5	Steuerlogik	30
3.5.1	Mecanum-Räder – Frei bewegen auf vier Rollen	30
4	Entwicklung eines eigenen LLMs	31
4.1	Datensatz	31
4.1.1	Labelled Data	31
4.1.2	Unlabelled Data	32
4.1.3	Test Data	32
4.2	Tokenizer & Normalisierung	33
4.3	Distillation	34
4.3.1	Teacher – Supervised Fine-Tuning	34
4.3.2	Soft-Target-Extraktion	35
4.3.3	Ausblick	35
4.4	Quantisierung	35
4.5	Student-Modell und Training	36
4.5.1	Mini-T5-Architektur	36
4.5.2	Distillations-Setup	36
4.5.3	Trainingsablauf	37
4.6	Architektur	38
4.6.1	Allgemeine Übersicht	38
4.6.2	Residual- und Normalisierungsschicht	38
4.6.3	Feed-Forward-Netzwerk (FFN)	39
4.6.4	Aufmerksamkeitsmechanismus (Attention)	40
4.6.5	Positionskodierung	40
4.6.6	Einbettungs- und Ausgabeschicht	41
4.6.7	Cross-Attention	41
4.6.8	Encoder und Decoder	42
4.7	Training	45
4.7.1	Teacher – Supervised Fine-Tuning	45
4.7.2	Student-Training	46
4.8	Optimierung	47
5	Integration von Roboter und LLM	49
5.1	Systemübersicht	49
5.2	Webinterface	50
5.3	Vorverarbeitung der Nutzereingaben	51

5.3.1	Normalisierung und Rauschfilterung	52
5.3.2	Rechtschreibkorrektur	53
5.3.3	Numerische Verarbeitung	54
5.4	Struktur des JSON-Kommandos	56
5.5	REST-API Back-End	57
5.6	Robotersimulator	58
5.6.1	Implementierung des Simulators	58
5.6.2	Webinterface zur Steuerung	58
5.6.3	Software auf dem Roboter	59
6	Vergleich mit bestehenden Sprachmodellen	61
6.1	Gemini – Kommerzielles Modell mit Prompt-Engineering	61
6.2	Feinjustiertes Teacher-Modell – Gute Ergebnisse mit leichten Schwächen	61
6.3	Eigenes Modell – Herausforderungen und Grenzen	62
6.4	Verbesserung durch Pre- und Post-Processing	62
6.5	Zusammenfassung	63
7	Fazit und Ausblick	64

Kapitel 1

Einleitung

Die rasanten Fortschritte im Bereich der Künstlichen Intelligenz (KI), insbesondere im Teilgebiet des *Natural Language Processing* (NLP), haben in den letzten Jahren eine Vielzahl neuer Anwendungen ermöglicht. Eine der einflussreichsten Innovationen in diesem Kontext ist die Entwicklung sogenannter **Large Language Models** (LLMs). Diese hochskalierbaren, auf der Transformer-Architektur basierenden Modelle sind in der Lage, komplexe sprachliche Aufgaben wie Textgenerierung, maschinelle Übersetzung und Dialogführung auf menschlichem Niveau zu bewältigen [Vas+17; Cor25].

Gleichzeitig gewinnen flexible, sprachbasierte Steuerungssysteme auch in der Robotik zunehmend an Bedeutung. Während klassische Robotersteuerungen häufig auf regelbasierten Algorithmen und festen Kommandos basieren, bieten LLMs durch ihre Sprachverständnissfähigkeiten eine neue Dimension der Mensch-Maschine-Interaktion. Besonders relevant wird diese Entwicklung mit Blick auf ressourcenschonende Edge-Anwendungen, bei denen kompakte Modelle direkt auf mobilen oder eingebetteten Systemen eingesetzt werden können [HS+21; NK24].

Ziel dieser Arbeit ist es, die Machbarkeit einer solchen sprachgesteuerten Robotersteuerung mit einem kompakten LLM zu untersuchen. Hierfür wird ein eigenes **Large Language Model** entwickelt und auf einem eigens erstellten Datensatz trainiert. Das Modelltraining erfolgte lokal auf handelsüblichen GPUs sowie ergänzend auf Google Colab, um die Effizienz und Anpassungsfähigkeit des Sprachmodells sicherzustellen. Durch Techniken wie *Knowledge Distillation* [HVD15] und *Quantisierung* [`huggingface\quantization`] wird ein ressourcenschonendes Modell erzeugt. Momentan läuft dieses Modell auf einem Computer, der über ein Kabel direkt mit einem Arduino verbunden ist, welcher die Steuerbefehle für den Roboter interpretiert und umsetzt. Zukünftige Arbeiten umfassen die Portierung dieses Modells auf eine eigenständige Plattform, konkret auf einen Raspberry Pi, um eine vollständige Unabhängigkeit des Roboters von externen Systemen zu ermöglichen.

Zur praktischen Umsetzung wurde ein Roboterprototyp vollständig eigenständig entworfen und realisiert. Die komplette mechanische Konstruktion wurde mithilfe

der CAD-Software Fusion 360 entwickelt und mittels 3D-Druck aus PLA-Filament gefertigt. Sämtliche elektrische Komponenten sowie deren Verkabelung wurden ebenfalls eigenständig geplant und umgesetzt. Obwohl geplant ist, den Roboter künftig autonom zu machen – unter Einbeziehung von Ultraschallsensoren und einer KI-gestützten Kamera auf einem Raspberry Pi – musste aus zeitlichen Gründen aktuell auf eine vereinfachte Prototyp-Version zurückgegriffen werden, bei der die Steuerung per Laptop über ein Kabel mit einem Arduino erfolgt.

Ein maßgeblicher Impuls für diese Arbeit stammt aus der Forschungsarbeit von Narimani und Klarmann [NK24], welche den Einsatz von großen Sprachmodellen zur Fehlerdiagnose und Steuerung im industriellen Kontext untersucht. Die dort beschriebenen Ergebnisse verdeutlichen das Potenzial von LLMs, komplexe und dynamische Aufgabenstellungen erfolgreich zu lösen. Aufbauend auf diesen Erkenntnissen soll im Rahmen dieser Arbeit demonstriert werden, dass LLMs – in komprimierter Form – auch für die direkte Steuerung physischer Roboter praktikabel und effizient einsetzbar sind.

Hintweis: Für die sprachliche Formulierung und Strukturierung dieser Arbeit wurde teilweise die Unterstützung moderner KI-basierter Sprachmodelle genutzt, insbesondere Large Language Models (LLMs). Da Deutsch nicht meine Muttersprache ist, dienten diese Systeme dazu, meine Ideen klar und präzise zu reformulieren. Der Inhalt und die wissenschaftlichen Konzepte sind ausschließlich von mir selbst erarbeitet worden.

Kapitel 2

Theoretische Grundlagen

2.1 Grundlagen des maschinellen Lernens

Maschinelles Lernen (ML) ist ein zentraler Teilbereich der künstlichen Intelligenz, der es Systemen ermöglicht, eigenständig aus Daten Muster zu erkennen und darauf basierend Entscheidungen zu treffen, ohne explizit dafür programmiert zu sein. Ziel ist es, eine Funktion $f(x)$ zu erlernen, die Eingabedaten x möglichst genau auf Zielwerte y abbildet. Dies erfolgt typischerweise durch iterative Anpassung der Modellparameter mittels Optimierungsverfahren, um eine definierte Verlustfunktion $L(f(x), y)$ zu minimieren.

Ein typischer maschinelles Lernprozess umfasst dabei folgende Schritte (siehe Abbildung 2.1):

1. **Datenaufbereitung:** Reinigung und Strukturierung der Daten.
2. **Modelltraining:** Anpassung der Modellparameter durch Optimierungsmethoden wie den Gradientenabstieg.
3. **Modellvalidierung:** Bewertung der Modellleistung anhand einer geeigneten Verlustfunktion.
4. **Modellanpassung:** Gegebenenfalls Optimierungen zur Verbesserung der Modellgenauigkeit.
5. **Anwendung:** Nutzung des validierten Modells auf neuen Daten.

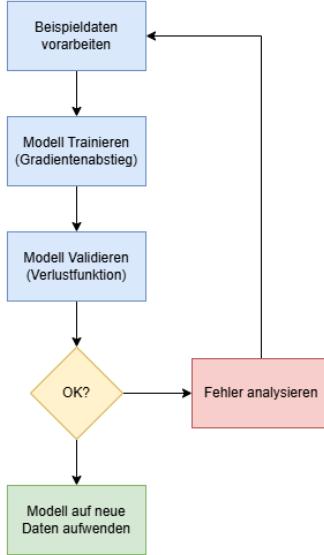


Abbildung 2.1: Vereinfachte Darstellung eines maschinellen Lernprozesses **Quelle:** [Eigene Darstellung](#)

Maschinelles Lernen unterteilt sich in verschiedene Paradigmen: Beim *überwachten Lernen* werden Modelle mit Datensätzen trainiert, die sowohl Eingabedaten als auch Zielwerte enthalten, während das *unüberwachte Lernen* Muster in Daten ohne vorgegebene Zielwerte erkennt. Beim *bestärkenden Lernen* lernt ein Agent durch Interaktion mit seiner Umgebung und anhand von Belohnungssignalen, optimale Entscheidungen zu treffen.

Der Erfolg eines Modells hängt stark von der richtigen Wahl der Modellarchitektur und einer geeigneten Verlustfunktion ab. Neben klassischen Modellen wie Entscheidungsbäumen oder Support-Vektor-Maschinen werden zunehmend neuronale Netzwerke verwendet, deren Komplexität durch stochastische Optimierungstechniken und Regularisierungsmethoden kontrolliert wird, um Überanpassung zu vermeiden und eine gute Generalisierungsfähigkeit sicherzustellen.

Weiterführende Informationen bietet Domingos [Dom12].

2.1.1 Gradientenabstieg (Gradient Descent)

Gradientenabstieg ist ein grundlegendes Optimierungsverfahren im maschinellen Lernen, bei dem die Parameter θ eines Modells iterativ angepasst werden, um eine Zielfunktion $J(\theta)$ zu minimieren. In jeder Iteration bewegt sich der Algorithmus dabei einen kleinen Schritt in Richtung des steilsten Abstiegs, also entgegen dem Gradienten der Zielfunktion:

$$\theta_{\text{neu}} = \theta_{\text{alt}} - \eta \cdot \nabla_{\theta} J(\theta) \quad (2.1)$$

Hier gibt die Lernrate η an, wie groß diese Anpassungsschritte ausfallen. Je nach Wahl dieser Rate nähert sich das Modell schneller oder langsamer dem optimalen Wert, der die besten Vorhersagen ermöglicht.

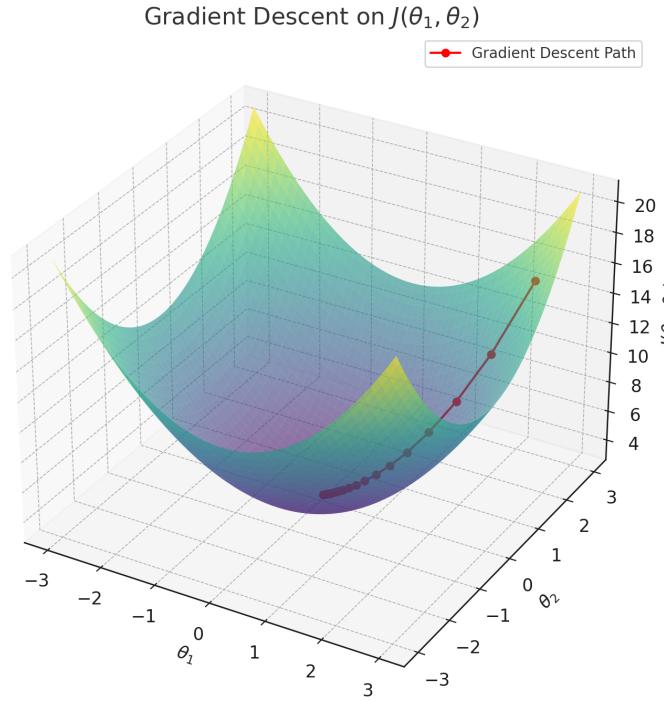


Abbildung 2.2: Gradientenabstieg auf einer konvexen Funktion $J(\theta_1, \theta_2) = \theta_1^2 + \theta_2^2 + 3$ in \mathbb{R}^2 . Die rote Linie zeigt den Pfad des Algorithmus in Richtung des globalen Minimums. **Quelle:** Eigene Darstellung

Abbildung 2.2 zeigt beispielhaft, wie der Algorithmus eine einfache, zweidimensionale konvexe Zielfunktion optimiert. Der rote Pfad stellt dar, wie sich die Parameter schrittweise zum Minimum der Funktion bewegen. Dieses Beispiel verdeutlicht intuitiv das Prinzip und die Konvergenz des Gradientenabstiegs.

Weitere Informationen zum Gradientenabstieg finden sich bei Domingos [Dom12].

2.1.2 Backpropagation

Backpropagation ist ein zentrales Verfahren, mit dem neuronale Netzwerke lernen. Dabei wird der Fehler, den das Netzwerk bei einer Vorhersage gemacht hat, schrittweise von der Ausgabeschicht zurück durch alle vorherigen Schichten geleitet. So lässt sich berechnen, wie stark jedes Gewicht und jeder Bias zum Fehler beigetragen hat.

Diese Berechnung erfolgt mithilfe der Kettenregel aus der Differentialrechnung. Die resultierenden Gradienten zeigen, in welche Richtung und wie stark die Gewichte angepasst werden müssen, um den Fehler beim nächsten Durchlauf zu verringern.

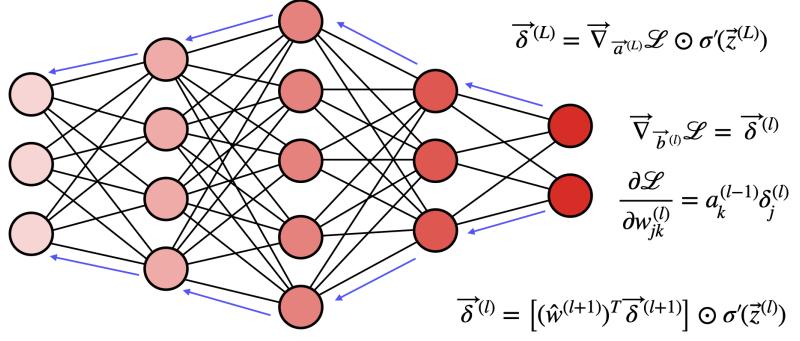


Abbildung 2.3: Schematische Darstellung des Backpropagation-Verfahrens in einem mehrschichtigen neuronalen Netzwerk. Die Pfeile in Blau zeigen die Vorwärtsausbreitung der Aktivierungen, während die roten Deltas $\delta^{(l)}$ die rückwärts gerichtete Ausbreitung des Fehlers (Gradienten) repräsentieren. Quelle: [For].

Abbildung 2.3 zeigt diesen Ablauf grafisch. Im sogenannten Forward Pass wird eine Vorhersage erzeugt, im anschließenden Backward Pass wird der Fehler durch das Netzwerk zurückgeführt. Dabei entstehen die Gradienten, die z.B. mit Gradientenabstieg genutzt werden, um das Modell zu verbessern. So lernt das Netzwerk schrittweise, bessere Vorhersagen zu treffen.

2.2 Verlustfunktionen (Loss Functions)

Eine Verlustfunktion quantifiziert den Unterschied zwischen der Vorhersage eines Modells und dem tatsächlichen Zielwert. Sie bildet das zentrale Optimierungskriterium im Trainingsprozess, indem das Modell so angepasst wird, dass der Verlust minimiert wird. In diesem Projekt werden zwei Verlustfunktionen kombiniert, um sowohl feste Zielwerte (Hard Targets) als auch zusätzliche Informationen aus weichen Zielwerten (Soft Targets) zu berücksichtigen.

2.2.1 Kreuzentropieverlust (Cross Entropy Loss)

Der Kreuzentropieverlust quantifiziert die Differenz zwischen der vom Modell vorhergesagten Wahrscheinlichkeit und dem tatsächlichen Hard Label. Für die binäre Klassifikation gilt:

$$\text{Loss}_{y=1} = -\log(p), \quad \text{Loss}_{y=0} = -\log(1-p),$$

wobei p die vorhergesagte Wahrscheinlichkeit für $y = 1$ darstellt. Zur Veranschaulichung wurde der Verlust mit

$$\text{cross_entropy_loss} = 5 \cdot \exp(-0.03 \cdot \text{Iteration}) + 0.2 \cdot \text{Rauschen}$$

simuliert (negative Werte werden auf 0 begrenzt). Abbildung 2.4 zeigt den simulierten Verlauf, der den exponentiellen Abfall des Verlusts verdeutlicht.

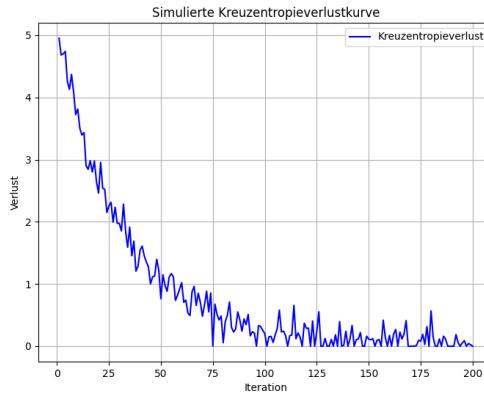


Abbildung 2.4: Simulierter Plot des Kreuzentropieverlusts **Quelle:** Eigene Darstellung

2.2.2 Kullback-Leibler-Divergenz (KLDiv Loss)

Die Kullback-Leibler-Divergenz misst die Abweichung zwischen zwei Wahrscheinlichkeitsverteilungen. Für eine fixe wahre Verteilung $P = (0.7, 0.3)$ wird sie definiert als:

$$D_{KL}(P\|Q) = 0.7 \cdot \log \frac{0.7}{q} + 0.3 \cdot \log \frac{0.3}{1-q}.$$

Zur Simulation des Trainingsverhaltens wurde die Funktion

$$\text{kl_div_loss} = 3 \cdot \exp(-0.02 \cdot \text{Iteration}) + 0.15 \cdot \text{Rauschen}$$

verwendet (negative Werte werden auf 0 begrenzt). Abbildung 2.5 zeigt, wie der Verlust von etwa 3 zu Beginn exponentiell abnimmt.

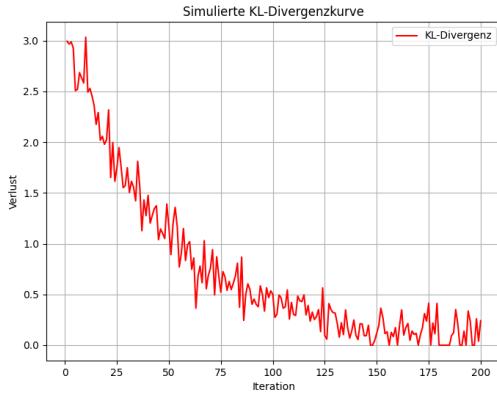


Abbildung 2.5: Simulierter Plot der Kullback-Leibler-Divergenz **Quelle: Eigene Darstellung**

2.2.3 Kombination der Verlustfunktionen

Um sowohl harte Zielwerte (Cross Entropy) als auch vom Teacher stammende Soft Targets (KL-Divergenz) zu berücksichtigen, kombiniert dieser Ansatz beide Terme:

$$\text{Loss}_{\text{train}} = \text{CE} + \text{KL}.$$

Zum Vergleich wurde zusätzlich eine gewichtete Simulation berechnet,

$$\text{Loss}_{\text{sim}} = 0.6 \text{ CE} + 0.4 \text{ KL},$$

um den Einfluss beider Komponenten anschaulich darzustellen. Abbildung 2.6 zeigt links den glatten Verlauf der Simulation und rechts echte Trainingswerte mit natürlichen Schwankungen. Beide Kurven – von mir selbst erstellt – konvergieren schnell gegen null und demonstrieren damit die Wirksamkeit der kombinierten Verlustfunktion.

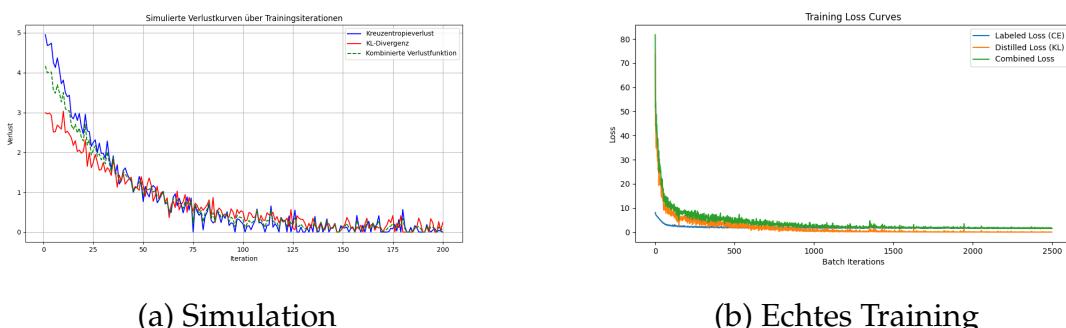


Abbildung 2.6: Verlauf der kombinierten Verlustfunktion in Simulation (a) und eigenem Training (b).

2.3 Grundlagen der Quantisierung

Quantisierung ist eine Technik zur Modellkompression, bei der die numerische Präzision der Modellgewichte oder Aktivierungen reduziert wird. Anstatt beispielsweise 32-Bit-Gleitkommazahlen (float32) zu verwenden, werden die Werte mit geringerer Bitanzahl – etwa 8-Bit-Ganzzahlen (int8) – dargestellt. Dies führt zu einer erheblichen Reduktion von Speicherbedarf und Rechenleistung, was insbesondere für den Einsatz in mobilen Geräten oder eingebetteten Systemen relevant ist [Facb].

Zur Veranschaulichung des Quantisierungsprozesses wurde ein Beispiel mit einer komplexen Wellenfunktion simuliert. Die Quantisierung erfolgt mit folgender Funktion:

```
def quantize_to_int8(x):  
    return np.int8(np.clip(np.round(x * 127), -128, 127))
```

Hierbei werden die float32-Werte zunächst mit 127 multipliziert (angenommener Maximalwert 1), gerundet und dann auf den Bereich [-128, 127] begrenzt. Zur Veranschaulichung werden die quantisierten Werte wieder in den Float-Bereich zurückkonvertiert.

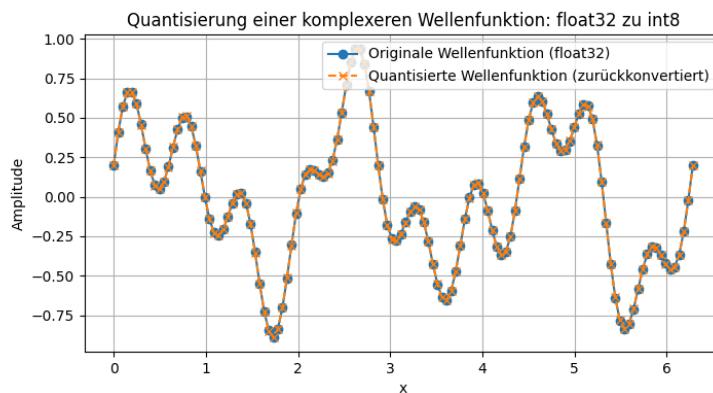


Abbildung 2.7: Simulation der Quantisierung einer Wellenfunktion: Vergleich zwischen der originalen float32-Wellenfunktion und der nach der Quantisierung rekonstruierten Version.

Diese Visualisierung zeigt, dass trotz der Reduktion der numerischen Präzision der allgemeine Verlauf der Wellenfunktion erhalten bleibt – feine Details gehen jedoch verloren. Dies entspricht dem typischen Verhalten der Quantisierung: Ein signifikanter Vorteil hinsichtlich Speicher- und Rechenersparsnis, jedoch verbunden mit einem gewissen Genauigkeitsverlust.

2.3.1 Arten von Quantisierung

In der Praxis werden vor allem drei Ansätze zur Quantisierung unterschieden, die je nach Anwendungsvorgaben unterschiedliche Vor- und Nachteile bieten:

- **Post-training Quantization:** Bei diesem Verfahren wird ein vollständig trainiertes Modell nachträglich in eine niedrigere numerische Präzision (z.B. int8) konvertiert. Der Vorteil liegt in der einfachen Implementierung, da keine Änderungen am Trainingsprozess erforderlich sind. Allerdings kann es zu einem Genauigkeitsverlust kommen, wenn das Modell nicht entsprechend kalibriert wird.
- **Quantization-aware Training:** Hier wird das Modell bereits während des Trainings auf die zukünftige Quantisierung vorbereitet. Das Training simuliert quantisierungsbedingte Effekte, sodass das Modell lernt, mit der reduzierten Präzision umzugehen. Dieser Ansatz führt in der Regel zu einer geringeren Einbuße der Modellgenauigkeit, erfordert jedoch eine Anpassung des Trainingsprozesses und kann den Trainingsaufwand erhöhen.
- **Dynamische Quantisierung:** Bei der dynamischen Quantisierung werden nur bestimmte Schichten (z.B. Linear-Schichten) zur Laufzeit während der Inferenz quantisiert, während andere Schichten in voller Präzision verbleiben. Dies ermöglicht einen guten Kompromiss zwischen Rechenleistung und Genauigkeit, da nur die rechenintensiven Teile des Modells quantisiert werden. In diesem Projekt kam diese Variante zum Einsatz, um die Inferenzgeschwindigkeit zu erhöhen, ohne dass die Modellgenauigkeit wesentlich beeinträchtigt wird.

Letztere Variante – die **dynamische Quantisierung** – kam in diesem Projekt zum Einsatz.

Weitere Informationen zur Quantisierung und deren praktischen Einsatz finden sich in der Hugging Face Dokumentation [Facb].

2.3.2 Implementierung der dynamischen Quantisierung

Zur Effizienzsteigerung beim Einsatz des t5-Modells habe ich dynamische Quantisierung mit PyTorch verwendet.

Dabei wurden gezielt nur Linear-Schichten (vollständig verbundene Schichten) in das 8-Bit-Format (qint8) überführt. Dies stellt einen guten Kompromiss dar: Die Rechenintensität wird reduziert, ohne dass die Modellgenauigkeit nennenswert darunter leidet.

Zusätzlich wurde für jedes generierte Token die Top-5-Wahrscheinlichkeiten aus der Ausgangsverteilung extrahiert und in float16-Präzision gespeichert. Dies

ermöglicht eine platzsparende Speicherung von **Soft Targets** für spätere Verfahren wie Distillation oder gewichtete Trainingsmethoden.

Durch den Einsatz von dynamischer Quantisierung konnte die Inferenzgeschwindigkeit des T5-Modells erhöht und der Speicherbedarf reduziert werden. Die zusätzlich eingesetzte float16-Quantisierung der Wahrscheinlichkeiten stellt zudem sicher, dass probabilistische Informationen effizient weiterverarbeitet werden können – ein zentraler Bestandteil für das geplante Distillation-Setup.

2.4 Distillation

Wissensdistillation (Knowledge Distillation) ist ein Verfahren zur Modellkompression, bei dem das Wissen eines großen, leistungsfähigen **Teacher-Modells** auf ein kleineres **Student-Modell** übertragen wird. Dabei dienen die vom Teacher generierten "Soft Targets" – also Wahrscheinlichkeitsverteilungen über die Klassen – als zusätzliche Informationsquelle während des Trainings des Student-Modells. So lernt das Student-Modell nicht nur, die harten Klassenlabels (Hard Targets) zu reproduzieren, sondern auch feine Unterschiede und Unsicherheiten zwischen den Klassen zu erfassen.

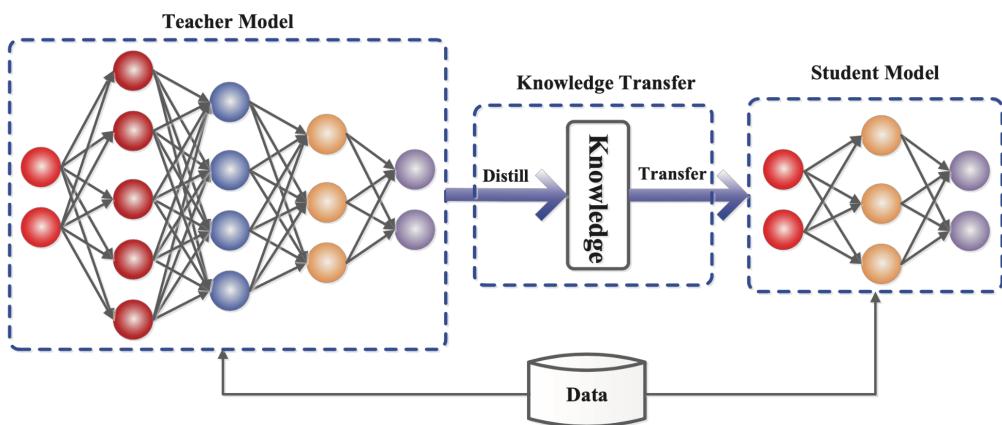


Abbildung 2.8: Schematische Darstellung des Knowledge-Distillation-Prozesses
Quelle: Neptuneai Blog [Nep].

Wie in den Abschnitten zur *Kreuzentropieverlust* (siehe Abbildung 2.4) und zur *Kullback-Leibler-Divergenz* (siehe Abbildung 2.5) erläutert, werden im Distillationsprozess beide Verlustfunktionen kombiniert:

- **Kreuzentropieverlust:** Misst die Differenz zwischen den harten Zielwerten und den Vorhersagen des Student-Modells.
- **Kullback-Leibler-Divergenz:** Quantifiziert die Abweichung zwischen der Soft

Target-Verteilung des Teacher-Modells und der vom Student-Modell berechneten Wahrscheinlichkeitsverteilung.

Durch die Kombination dieser beiden Ansätze lernt das Student-Modell, nicht nur die korrekten Klassen zuzuordnen, sondern auch die zugrunde liegenden Unsicherheiten und Beziehungen zwischen den Klassen zu erfassen. Dies führt zu einer verbesserten Generalisierungsfähigkeit, ohne dass ein großes, speicherintensives Modell für die Inferenz benötigt wird.

Die in diesem Projekt verwendete Distillationsmethode wurde maßgeblich durch die Arbeit von Hinton et al. [HVD15] inspiriert. Weitere Informationen und Inspirationen zum Thema Knowledge Distillation finden sich auch im Neptune.ai Blog [Nep]. Eine detaillierte Beschreibung der Implementierung und der spezifischen Trainingsparameter des Distillationsprozesses erfolgt in einem späteren Abschnitt dieser Arbeit.

2.5 Moderne NLP-Methoden: Transformers und Large Language Models

2.5.1 Einordnung von NLP und der Wandel durch Transformer

Die Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) stellt einen der zentralen Anwendungsbereiche im Feld der künstlichen Intelligenz dar. Ziel ist es, Maschinen in die Lage zu versetzen, menschliche Sprache in geschriebener oder gesprochener Form zu verstehen, zu interpretieren und zu generieren. Frühere Ansätze basierten auf regelbasierten Systemen oder klassischen statistischen Modellen, welche bei der Verarbeitung komplexer sprachlicher Strukturen schnell an ihre Grenzen stießen [Dom12].

Der Durchbruch erfolgte mit der Verbreitung tief neuronaler Netze, insbesondere durch die Einführung der Transformer-Architektur von Vaswani et al. [Vas+17]. Diese ermöglichte es erstmals, längere Texte parallel und kontextbewusst zu verarbeiten – ein bedeutender Fortschritt gegenüber rekurrenten Netzen, die sequentiell arbeiteten und unter dem Vanishing-Gradient-Problem litten [Ras21]. Damit wurde die Grundlage für heutige leistungsfähige Sprachmodelle geschaffen.

2.5.2 Einführung in LLMs

Large Language Models (LLMs) sind hochskalierte neuronale Netzwerke, die auf der Transformer-Architektur basieren und aus umfangreichen Textkorpora vielfältige Sprachmuster lernen. Bekannte Vertreter wie GPT (Generative Pre-trained Transfor-

mer) oder BERT (Bidirectional Encoder Representations from Transformers) demonstrieren eindrucksvoll, wie flexibel und leistungsfähig moderne LLMs bei Aufgaben wie Textgenerierung, maschineller Übersetzung oder Frage-Antwort-Systemen sind.

Dank ihrer Skalierbarkeit und Generalisierungsfähigkeit stellen sie heute das Fundament für viele fortgeschrittene Anwendungen in Forschung und Industrie dar. Im Rahmen dieser Arbeit liegt der Fokus auf einer vertieften Analyse solcher Modelle sowie auf der Entwicklung einer eigenen, angepassten Modellarchitektur zur Steuerung eines Roboters durch natürliche Sprache.

2.5.3 Grundlagen der Transformer-Architektur

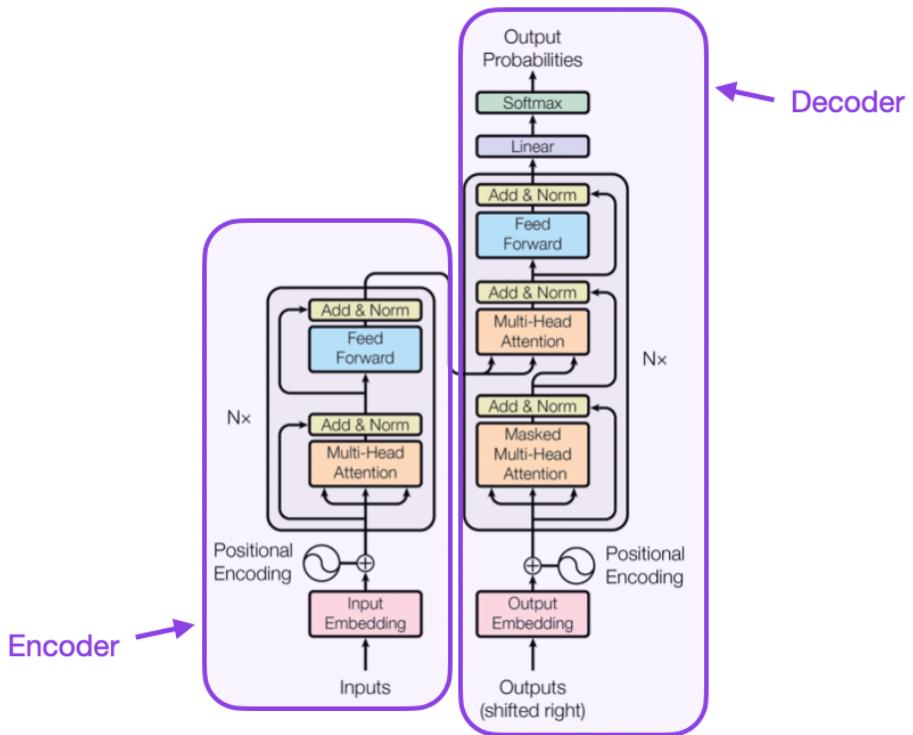


Abbildung 2.9: Schematische Darstellung der Transformer-Architektur nach Vaswani et al. [Vas+17]

Die Transformer-Architektur, eingeführt von Vaswani et al. [Vas+17], bildet die Grundlage moderner Large Language Models und hat sich durch ihre Effizienz und Skalierbarkeit als Standard in vielen NLP-Anwendungen etabliert. Im Gegensatz zu früheren rekurrenten oder konvolutionellen Ansätzen basiert der Transformer vollständig auf sogenannten Attention-Mechanismen.

Das Modell besteht aus zwei Hauptkomponenten: dem **Encoder** und dem **Decoder**. Beide sind aus mehreren identischen Schichten (sogenannten „Layers“) aufgebaut, die jeweils zentrale Bausteine wie Multi-Head Attention und Feed-Forward-

Netzwerke enthalten.

Der Encoder verarbeitet die Eingabesequenz vollständig parallel. Jedes Token wird dabei zunächst in eine kontinuierliche Repräsentation (Embedding) überführt und erhält zusätzlich eine Positionsinformation (Positional Encoding). Anschließend werden die Token durch mehrere Schichten aus Selbstaufmerksamkeit und Feed-Forward-Netzwerken geleitet. Das Ziel des Encoders ist es, eine kontextualisierte Repräsentation der Eingabesequenz zu erzeugen, die relevante Beziehungen zwischen den Tokens berücksichtigt.

Der Decoder generiert die Zielsequenz Token für Token. Dabei greift er sowohl auf die bereits erzeugten Ausgaben zurück (über maskierte Selbstaufmerksamkeit), als auch auf die vom Encoder bereitgestellten Repräsentationen (über Encoder-Decoder-Attention). Durch diese Kombination ist der Decoder in der Lage, kohärente und kontextuell passende Ausgaben zu erzeugen – etwa in der maschinellen Übersetzung oder Textgenerierung.

Die Stärke des Transformers liegt in der expliziten Trennung von Eingabe- und Ausgabeverarbeitung sowie in der parallelen Verarbeitung von Sequenzen. Dies führt zu einer erheblichen Effizienzsteigerung gegenüber sequentiellen Modellen.

Die folgenden Abschnitte beleuchten die Schlüsselkomponenten dieser Architektur im Detail: den Aufmerksamkeitsmechanismus, das Feed-Forward-Netzwerk und die Positionelle Kodierung.

2.6 Komponenten der Transformer-Architektur

Die Leistungsfähigkeit von Transformer-Modellen basiert auf dem Zusammenspiel mehrerer spezialisierter Komponenten, die jeweils unterschiedliche Aspekte der Sprachverarbeitung übernehmen. Diese modulare Struktur erlaubt es, komplexe Beziehungen innerhalb von Textsequenzen effizient zu modellieren – selbst über große Distanzen hinweg.

In diesem Abschnitt werden die drei zentralen Bausteine der Architektur im Detail betrachtet: der Aufmerksamkeitsmechanismus (Attention), das Feed-Forward-Netzwerk (FFN) und die Positionelle Kodierung (Positional Encoding). Zusammen ermöglichen sie eine effektive Repräsentation und Transformation sprachlicher Informationen innerhalb jeder einzelnen Schicht des Modells.

2.6.1 Aufmerksamkeitsmechanismus (Attention)

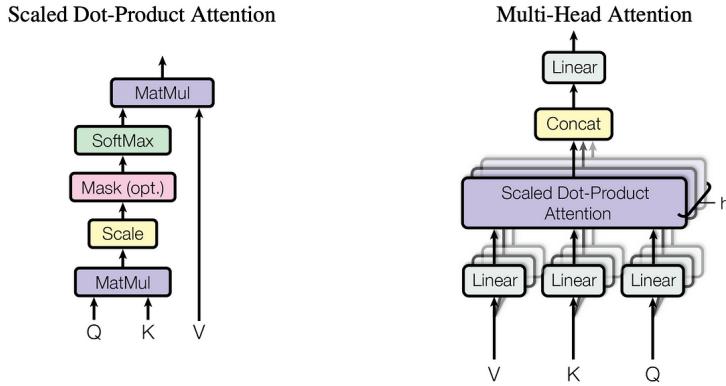


Abbildung 2.10: Darstellung des Multi-Head Attention-Mechanismus nach Vaswani et al. [Vas+17]

Der Aufmerksamkeitsmechanismus (Attention) bildet das Herzstück der Transformer-Architektur. Er ermöglicht es dem Modell, kontextuelle Beziehungen zwischen Tokens innerhalb einer Sequenz flexibel zu erkennen und zu gewichten. Jedes Token wird dabei durch ein Set von Vektoren beschrieben: *Query* (Q), *Key* (K) und *Value* (V).

Das zentrale Ziel besteht darin, zu bestimmen, wie stark ein Token auf andere Tokens in der Sequenz „achten“ soll. Mathematisch geschieht dies über die sogenannte **Scaled Dot-Product Attention**:

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

Hierbei steht d_k für die Dimensionalität der Key-Vektoren. Das Ergebnis ist eine gewichtete Summe der Value-Vektoren, wobei die Gewichtung durch die Ähnlichkeit zwischen Query und Key bestimmt wird.

Der Mechanismus wird im Modell durch die sogenannte **Multi-Head Attention** erweitert. Dabei werden mehrere Attention-Köpfe parallel berechnet, die jeweils unterschiedliche Aspekte der Sequenz erfassen. Dies ermöglicht dem Modell, verschiedene semantische Beziehungen gleichzeitig zu lernen und zu integrieren.

2.6.2 Feed-Forward-Netzwerk (FFN)

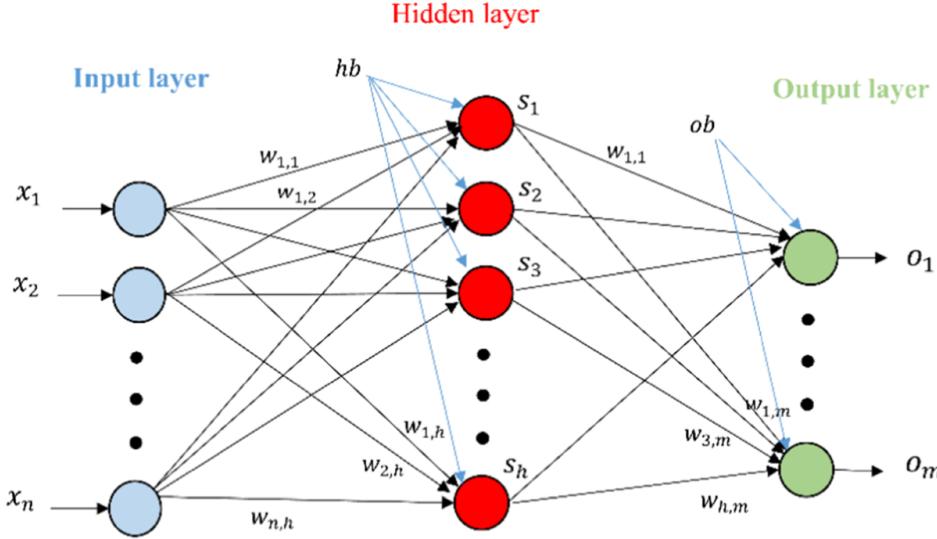


Abbildung 2.11: Struktur eines Feed-Forward-Netzwerks innerhalb einer Transformer-Schicht [Vas+17]

Jede Transformer-Schicht enthält neben der Attention-Einheit auch ein **Feed-Forward-Netzwerk (FFN)**. Dieses besteht aus zwei vollständig verbundenen (linearen) Schichten, zwischen denen eine nichtlineare Aktivierungsfunktion – typischerweise ReLU oder GELU – zum Einsatz kommt.

Das FFN verarbeitet die Ausgaben des Attention-Blocks weiter und transformiert sie in einen neuen Repräsentationsraum. Die Standardform dieses Netzwerks ist:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Dabei sind W_1, W_2 Gewichtsmatrizen und b_1, b_2 die zugehörigen Bias-Vektoren. Durch diese Architektur kann das Modell auch komplexe nichtlineare Beziehungen innerhalb der Token-Repräsentationen lernen und weiterverarbeiten.

Das FFN wird auf jedes Token einzeln, jedoch identisch angewendet („position-wise“), wodurch Effizienz und Parallelisierbarkeit erhalten bleiben.

2.6.3 Positionelle Kodierung (Positional Encoding)

Transformer-Modelle besitzen im Gegensatz zu rekurrenten Netzen keine inhärente Fähigkeit, die Reihenfolge von Token in einer Sequenz zu berücksichtigen. Um diese Positionsinformationen dennoch einzubringen, wird jeder Token-Repräsentation eine sogenannte *positionelle Kodierung* hinzugefügt. Diese Kodierung besteht aus deterministisch berechneten sinus- und cosinus-basierten Werten, die eindeutige Muster für jede Position im Satz erzeugen. Dadurch wird dem Modell ermöglicht, die relative und absolute Position eines Tokens zu interpretieren.

Mathematisch wird das Positional Encoding für jede Position pos und jede Dimension i wie folgt definiert:

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

Diese Struktur erlaubt es dem Modell, Positionsinformationen zu verarbeiten, ohne zusätzliche lernbare Parameter einzuführen. Die unterschiedlichen Frequenzen ermöglichen zudem, dass das Modell sowohl feingranulare als auch grobe Positionsverhältnisse erkennen kann.

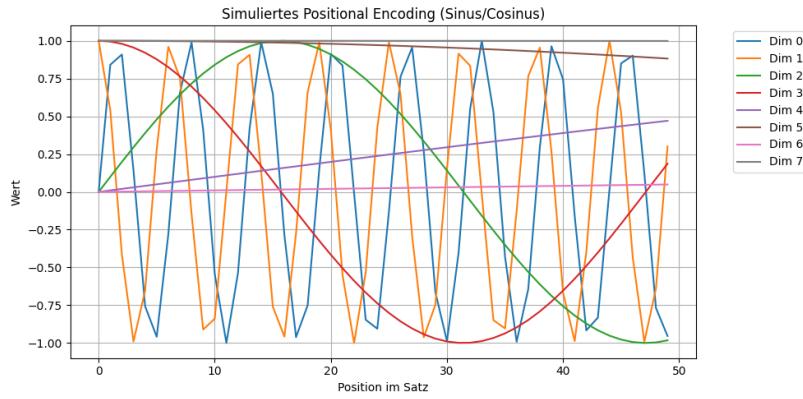


Abbildung 2.12: Simulierte Positional Encoding über 8 Dimensionen. Jeder Graph zeigt den Positionsverlauf einer Embedding-Dimension **Quelle: Eigene Darstellung**

Abbildung 2.12 zeigt den Verlauf der Positional-Encoding-Werte entlang der Sequenzposition für verschiedene Embedding-Dimensionen. Gut erkennbar ist, dass sich die Werte sinus- und cosinusförmig mit unterschiedlichen Frequenzen verändern, was dem Modell hilft, Positions muster eindeutig zu erfassen.

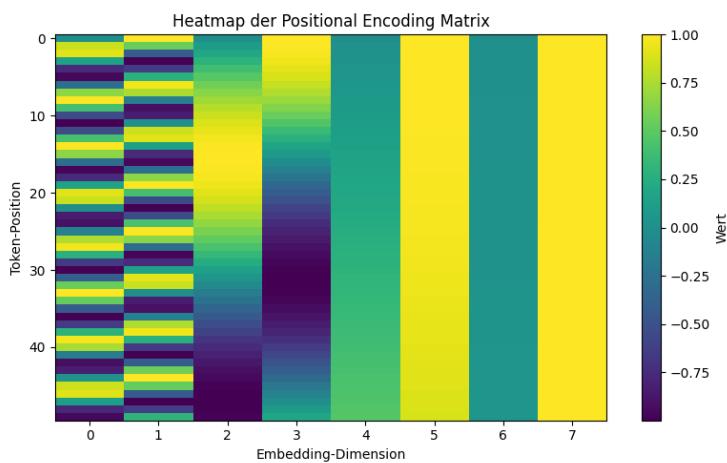


Abbildung 2.13: Heatmap der Positional Encoding Matrix mit 50 Token-Positionen und 8 Embedding-Dimensionen **Quelle: Eigene Darstellung**

Ergänzend veranschaulicht Abbildung 2.13 die Positional Encoding Matrix als Heatmap. Hier wird die systematische, regelmäßige Struktur der Kodierungen über Position und Dimension hinweg deutlich sichtbar. Diese Kodierung bildet die Grundlage für die Fähigkeit des Modells, Wortpositionen auch ohne rekursive Verarbeitung korrekt zu interpretieren.

2.7 Tokenisierung

Die Tokenisierung ist ein zentraler Vorverarbeitungsschritt in der Verarbeitung natürlicher Sprache (NLP). Sie bestimmt, wie ein Rohtext in kleinere Verarbeitungseinheiten – sogenannte *Tokens* – zerlegt wird. Diese Tokens sind die grundlegenden Eingabeeinheiten für NLP-Modelle und können einzelne Zeichen, Wörter oder Wortbestandteile sein.

Die Wahl der Tokenisierungsstrategie hat einen erheblichen Einfluss auf die Effizienz, Genauigkeit und Generalisierungsfähigkeit eines Sprachmodells. Besonders bei großen Sprachmodellen (LLMs) stellt die Tokenisierung einen entscheidenden Faktor für die Modellleistung dar, da sie bestimmt, wie Informationen in den Textdaten repräsentiert werden.

Grundsätzlich lassen sich drei Hauptansätze zur Tokenisierung unterscheiden:

- **Zeichenbasierte Tokenisierung:** Jeder Buchstabe, jedes Satzzeichen oder Sonderzeichen wird als einzelnes Token behandelt. Diese Methode verwendet ein kleines Vokabular und eignet sich gut für Sprachen mit komplexer Morphologie oder für Szenarien mit vielen unbekannten Wörtern. Sie führt jedoch zu langen Sequenzen und höherem Rechenaufwand.
- **Wortbasierte Tokenisierung:** Hierbei wird jedes Wort als einzelnes Token betrachtet. Diese Methode ist intuitiv und effizient, setzt aber ein großes Vokabular voraus und hat Schwierigkeiten mit unbekannten Wörtern (*Out-of-Vocabulary Problem*).
- **Subwort-Tokenisierung:** Diese Methode zerlegt Wörter in häufige Teilwort-einheiten oder Silben. Verfahren wie *Byte Pair Encoding* (BPE) oder *Unigram Language Model* kombinieren die Vorteile beider obiger Ansätze: ein kompaktes Vokabular bei gleichzeitig hoher Abdeckung und Robustheit gegenüber unbekannten Wörtern.

Im weiteren Verlauf dieses Kapitels werden die Unterschiede zwischen Zeichen-, Wort- und Subword-Tokenisierung detaillierter betrachtet.

2.7.1 Zeichenbasierte vs. wortbasierte Tokenisierung

Zwei klassische Ansätze in der Tokenisierung sind die **zeichenbasierte** und die **wortbasierte** Methode. Beide verfolgen unterschiedliche Strategien zur Zerlegung von Text und bieten jeweils spezifische Vor- und Nachteile (siehe Abbildung 2.14).

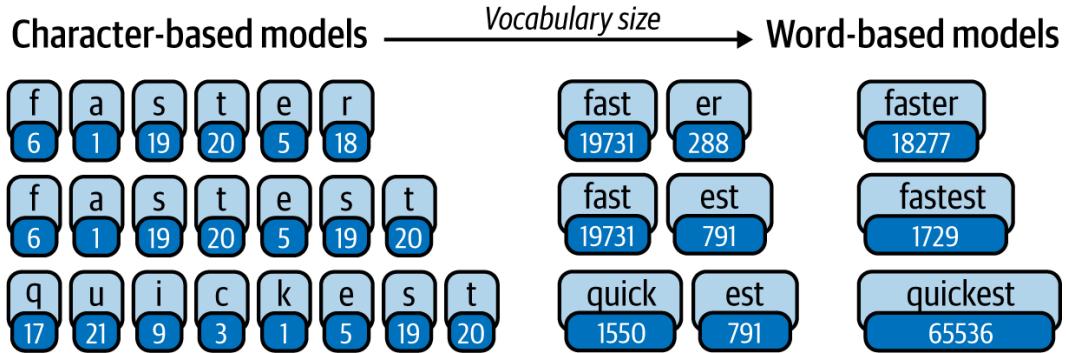


Abbildung 2.14: Vergleich zwischen zeichen- und wortbasierter Tokenisierung

Die **zeichenbasierte Tokenisierung** zerlegt Texte in einzelne Zeichen, was ein sehr kleines und flexibles Vokabular ermöglicht. Sie ist besonders robust gegenüber unbekannten Wörtern, führt aber zu längeren Sequenzen, die den Rechenaufwand erhöhen.

Die **wortbasierte Tokenisierung** arbeitet hingegen mit vollständigen Wörtern als Einheiten. Dadurch entstehen kürzere Sequenzen, was die Effizienz verbessert – allerdings auf Kosten der Generalisierung, da unbekannte Wörter nicht direkt verarbeitet werden können.

Da beide Ansätze Einschränkungen haben, wurden Methoden wie *Byte Pair Encoding* (BPE) entwickelt, um die Vorteile beider Verfahren zu kombinieren. Diese werden im nächsten Abschnitt näher erläutert.

2.7.2 Byte Pair Encoding (BPE)

Ein moderner und weit verbreiteter Ansatz zur Tokenisierung ist **Byte Pair Encoding (BPE)**. BPE kombiniert die Vorteile der zeichen- und wortbasierten Methoden, indem es häufig auftretende Zeichenfolgen zu stabilen Token-Einheiten zusammenfasst. Dadurch entsteht ein effizientes Token-Vokabular, das sowohl seltene als auch häufige Wörter kompakt darstellen kann, ohne den Kontextbezug zu verlieren.

BPE ist heute Standard bei vielen Large Language Models wie GPT oder BERT und wird von führenden Frameworks wie HuggingFace implementiert [Cor25; Ras21].

In dieser Arbeit wird ebenfalls BPE als zentrales Tokenisierungsverfahren verwendet, wobei ein eigener Tokenizer mithilfe der `tokenizers`-Bibliothek trainiert wurde. Die Details zur konkreten Umsetzung, einschließlich Trainingsdaten und Spezialtokens, werden in Kapitel *Implementierung* beschrieben.

Initiales Vokabular und Ablauf Zu Beginn setzt BPE auf ein minimales Vokabular, das ausschließlich aus einzelnen Zeichen besteht. Um Wortgrenzen zu kennzeichnen, wird häufig ein spezieller End-of-Word-Marker (_) verwendet. Beispielhafte Zerlegung:

```
"lower" → l o w e r _
"lowest" → l o w e s t _
"newer" → n e w e r _
```

Dieses initiale Zeichen-Vokabular wird iterativ durch häufige Zeichenpaare ergänzt, wodurch sich schrittweise ein subwortbasiertes Token-Vokabular ergibt.

Tabelle 2.1: Beispielhafte Schritte der Byte Pair Encoding Tokenisierung

Iteration	Häufigstes Paar	Tokensequenz
0	-	lower lowest newer
1	er	lower lowest newer
2	lo	lower lowest newer
3	low	lower lowest newer
4	es	lower lowest newer
5	est	lower lowest newer

Vokabularaufbau Nach fünf Iterationen hat sich das ursprüngliche Zeichen-Vokabular um folgende Subtoken erweitert:

```
V = {l, o, w, e, r, s, t, n, _, er, lo, low, es, est}
```

So entstehen robuste Token, die sowohl häufige Bestandteile als auch generalisierbare Subwörter abbilden. Selbst unbekannte Wörter können so sinnvoll in bekannte Einheiten zerlegt werden.

Für eine weiterführende und praxisnahe Einführung in BPE sei auf die Dokumentation von Hugging Face sowie auf einen anschaulichen Blogartikel von Weng verwiesen [Facc; Wen].

2.8 LoRA (Low-Rank Adaptation)

Low-Rank Adaptation (LoRA) ist eine effiziente Methode zur Feinabstimmung großer neuronaler Modelle. Ursprünglich wurde LoRA eingeführt, um den Resourcenverbrauch bei der Anpassung vortrainierter Transformer-Modelle deutlich zu reduzieren, indem es auf eine partielle Anpassung der Modellparameter setzt und dabei speziell auf eine niedrigrangige Matrixzerlegung zurückgreift [ibm\lora].

Diese Technik reduziert maßgeblich die Anzahl der Parameter, die während des Feinabstimmungsprozesses trainiert werden müssen, ohne die Leistungsfähigkeit des Modells signifikant einzuschränken.

Mathematisch lässt sich LoRA folgendermaßen formulieren: Gegeben sei eine ursprüngliche Gewichtsmatrix $W \in \mathbb{R}^{d \times k}$. LoRA ergänzt diese durch zwei kleinere, trainierbare Matrizen $B \in \mathbb{R}^{d \times r}$ und $A \in \mathbb{R}^{r \times k}$, wobei der Rang r deutlich kleiner als die Dimensionen der ursprünglichen Matrix gewählt wird ($r \ll \min(d, k)$). Damit ergibt sich die adaptierte Gewichtsmatrix zu:

$$W' = W + BA$$

Diese niedrigdimensionale Zerlegung erlaubt es, nur eine geringe Zahl zusätzlicher Parameter anzupassen, was sowohl den Speicherbedarf als auch die Rechenlast erheblich reduziert. Der Grundgedanke dahinter ist, dass für viele Feinabstimmungsaufgaben nur kleinere Anpassungen notwendig sind, um eine signifikante Leistungsverbesserung auf spezifischen Datensätzen zu erzielen. Somit vermeidet LoRA eine vollständige Neuberechnung oder umfassende Änderungen der bestehenden Gewichte.

Im Rahmen dieser Bachelorarbeit wird LoRA verwendet, um ein großes, vortrainiertes Sprachmodell auf einem spezifischen Datensatz effizient zu feinjustieren. Die dabei erzielten optimierten Modellparameter dienen anschließend als Grundlage für weitergehende Verfahren und Experimente. Weitere ausführliche Erläuterungen sowie praktische Anwendungsbeispiele zur LoRA-Technik bietet die Dokumentation von Hugging Face [Faca].

Kapitel 3

Praktische Umsetzung des Roboters

3.1 Elektrisches System

Das aktuelle Setup bleibt bewusst schlicht: Ein **Arduino Uno** (ATmega328P) steuert alle Komponenten; ein Raspberry-Pi-Upgrade für Netzwerk / Bildverarbeitung bleibt *Future Work*.

Stromkreise in der Zwischenlösung

- **Leistungskreis** – Ein 7.4 V Li-Po-Akku versorgt den Motortreiber und damit die vier DC-Motoren.
- **Logikkreis** – Der Uno und alle Sensorprototypen hängen pragmatisch am 5 V/500 mA-USB-Port des Laptops. Eine gemeinsame Masse verbindet beide Kreise sauber.

Sensor-Prototyping

Weil die 20 verfügbaren Uno-Pins knapp sind, simuliere ich fehlende Ultraschall- und Entfernungssensoren vorerst mit einfachen Drucktastern. Sie liefern nur „Hindernis ja/nein“, genügen aber für erste Fahrtests. Später werden echte Sensoren entweder direkt am Uno (I^2C / UART) oder über den Pi (ROS-Node) eingebunden.

Ausblick

Sobald der Platzbedarf geklärt ist, kommt ein Raspberry Pi als *Edge-Server* dazu. Er zieht die gleichen 5 V vom USB-Rail, spricht per UART mit dem Uno und öffnet dem Roboter den Weg ins WLAN (MQTT, OTA-Updates).

Das elektrische System ist damit jetzt bewusst minimal, aber schon auf eine spätere Aufrüstung vorbereitet.

3.1.1 Verkabelung und Komponentenverbindung

Abbildung 3.1 zeigt sowohl den *Ist-Aufbau* (Arduino-Prototyp) als auch das *Soll-Konzept* mit späterem Raspberry-Pi-Zusatz. Das Layout bleibt bewusst übersichtlich und leicht erweiterbar.

- **Steuerzentrale (heute)** – Ein Arduino Uno hängt per 5 V-USB am Laptop, verteilt GND und treibt alle Logik signale.
- **Sensor-Mock-up** – Vier einfache Drucktaster ersetzen Ultraschallsensoren; sie melden nur „Hindernis ja/nein“ und sparen Pins für den Start.
- **Motorantrieb** – Ein H-Bridge-Modul bekommt PWM + DIR vom Uno, Fahrstrom liefert eine 7.4 V Li-Po. Gemeinsame Masse verbindet Logik- und Leistungskreis.
- **Steuerzentrale (später)** – Ein Raspberry Pi soll per UART/I²C andocken, Kamera und echte Ultraschallsensoren übernehmen und das System ans WLAN anbinden. Dafür genügt ein zusätzliches vieradriges Datenband.

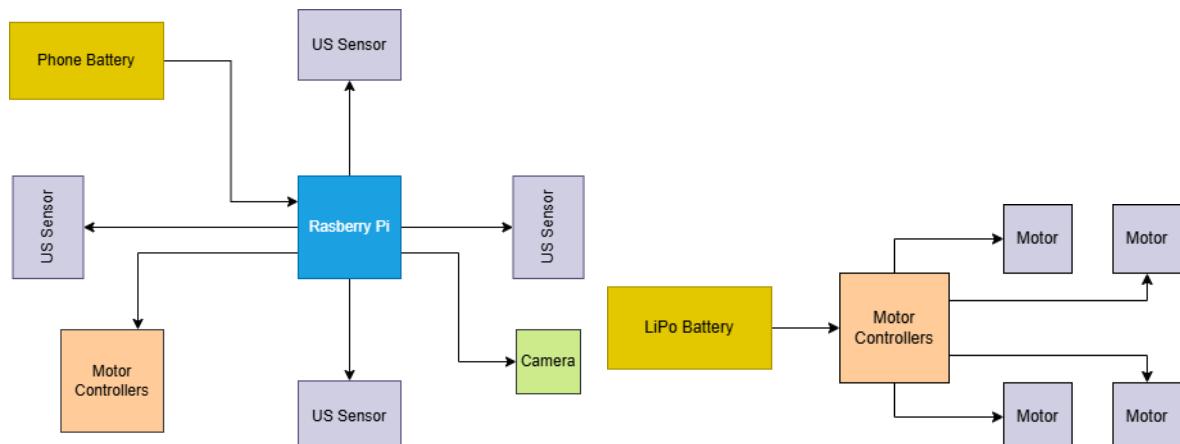


Abbildung 3.1: Links: geplanter Endausbau mit Raspberry Pi und Ultraschallsensoren. Rechts: unveränderte Stromführung für Motor + Treiber.

So bleibt die aktuelle Verdrahtung minimal, lässt sich aber ohne Kabeleingriff auf die finale Architektur aufrüsten.

3.2 Mechanische Konstruktion

Alle Gehäuse- und Halterungsteile stammen aus dem 3D-Drucker. Ich modellierte sie in *Fusion 360*, prüfte die Passform per digitaler „Joint“-Simulation und fertigte sie anschließend auf meinem FDM-Drucker (Ender 3) aus PLA.

Die Leitidee war eine modulare, leicht wartbare Plattform:

- **Steckbares Chassis** – vier gedruckte Segmente ergeben nach dem Verkleben eine stabile Basis, die trotzdem in das Druckbett passt.
- **Integrierte Aufnahmen** – Motoren, Sensoren, Akku und Elektronik sitzen in passgenauen Mulden; zusätzliche Schrauben oder Adapter entfallen.
- **Saubere Kabelführung** – interne Kanäle halten Leitungen kurz und schützen sie vor der Mechanik.

Das Ergebnis ist ein kompakter Rover mit vier Gleichstrommotoren, der sich gleichermaßen für Labortests und mobile Einsätze eignet – und sich dank der STL-Dateien im Anhang jederzeit nachdrucken oder anpassen lässt.

3.2.1 Chassis & Innenstruktur

Das komplette Gehäuse entstand in *Fusion 360* und wurde auf meinem Ender-3 in vier Steckmodulen gedruckt – so passt es trotz des kleinen Druckbetts aufs Gerät und bleibt später leicht zu warten. Beim Entwurf standen zwei Ziele im Mittelpunkt:

- **Modularität** – Motoren, Sensoren, Akku und Controller sitzen in passgenauen Mulden. Jedes Teil lässt sich einzeln tauschen, ohne den Rest zu zerlegen.
- **Robustheit** – Eine innere Rahmenkonstruktion trennt Elektronik (oberes Deck) von Antrieb (unten). Kurze Kabelwege, klarer Luftstrom und genug Steifigkeit für mobile Tests.

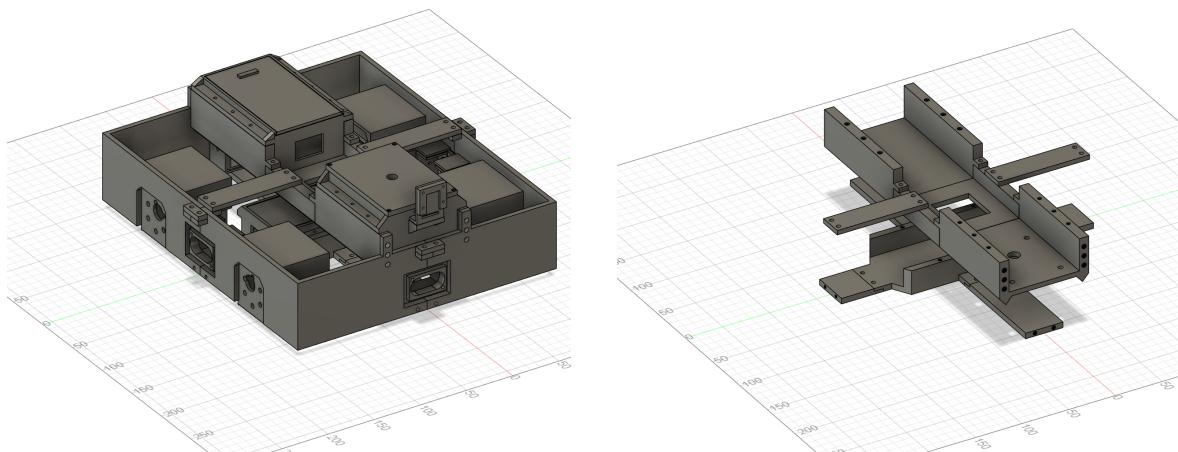


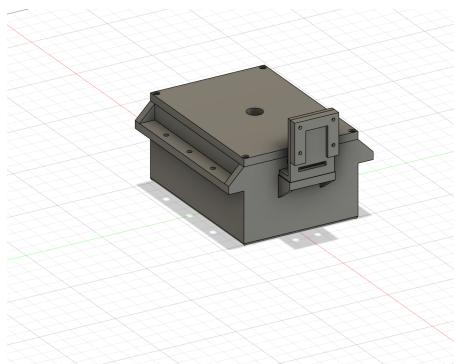
Abbildung 3.2: Chassis Version 3.37 mit integrierten Halterungen und zwei stöckigem Rahmen

Das Ergebnis ist eine kompakte, aber stabile Basis, die sich dank der STL-Dateien im Anhang jederzeit nachdrucken oder an neue Komponenten anpassen lässt.

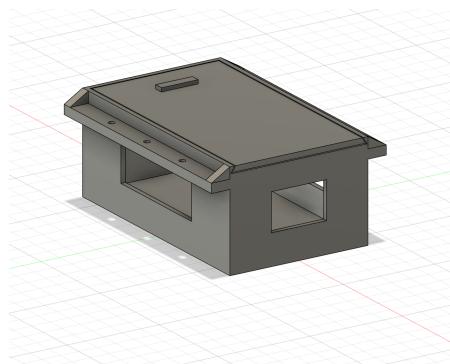
3.2.2 Gedruckte Module für Elektronik & Strom

Damit alles sauber ins Chassis passt, habe ich für jede Baugruppe ein eigenes kleines Gehäuse gedruckt (Abb. 3.3). Noch steckt dort zwar ein **Arduino Uno** statt des geplanten Raspberry Pi, und der Akku-Schacht ist leer – das Setup ist eben Work-in-Progress – aber die Hüllen sind schon bereit für das Upgrade.

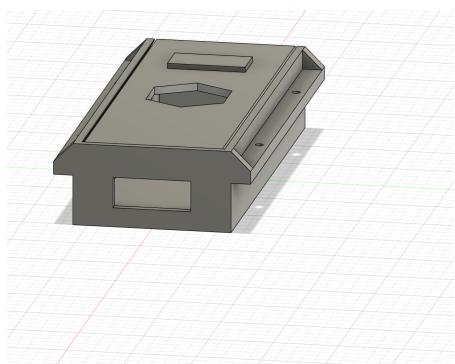
- **(a) Controller-Box** – Aktuell beherbergt sie den Uno; später soll hier der Pi samt Kamera einziehen. Alle Ports bleiben frei zugänglich.
- **(b) Motor-Treiber-Gehäuse** – Sorgt für geordnete Kabel und schützt die H-Bridge vor Vibrationen.
- **(c) Akku-Schacht** – Noch ungenutzt, weil der Uno per USB gefüttert wird; Platz ist aber schon da, um den Li-Po tiefer zu setzen und den Schwerpunkt zu senken.
- **(d) Sensor-Halter** – Snap-Fit-Teil für den Ultraschallsensor. Solange die Pins fehlen, steckt hier nur eine Platzhalter-Platte.



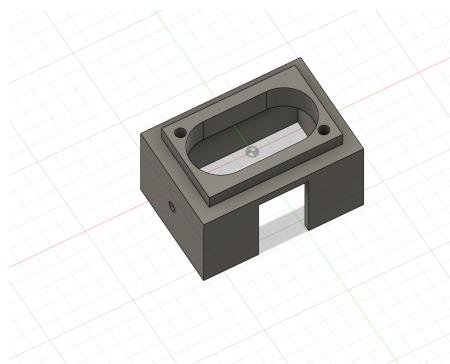
(a) Controller-Box



(b) Motor-Treiber-Gehäuse



(c) Akku-Schacht



(d) Sensor-Halter

Abbildung 3.3: Gedruckte Module – schon fertig, teils noch Platzhalter

So bleibt das Innenleben aufgeräumt, und sobald mehr Hardware da ist, muss ich nur noch die Deckel auf –*nicht* das ganze Chassis auseinander nehmen.

3.3 Sensorik (geplant)

Eigentlich sollen vier Ultraschallsensoren (z. B. HC-SR04) die Hinderniserkennung übernehmen. Wegen der noch knappen Uno-Pins simuliere ich sie zurzeit jedoch mit einfachen Drucktastern (siehe Abschnitt 3.1.1). Die Hardware-Halterungen sind schon gedruckt, die Elektronik wird nachgerüstet, sobald der Pin-Engpass gelöst ist.

Messprinzip in Kürze

Der Sensor sendet einen 40 kHz -Impuls, misst die Laufzeit t des Echos und berechnet die Distanz

$$d = \frac{v_{\text{Schall}} t}{2}$$

mit $v_{\text{Schall}} \approx 343 \text{ m s}^{-1}$. Die Division durch zwei berücksichtigt Hin- und Rückweg.

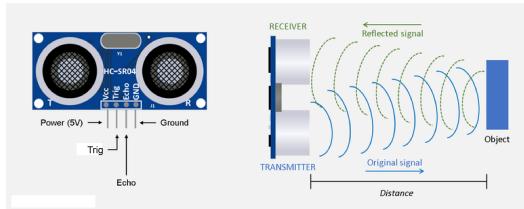


Abbildung 3.4: Ultraschallsensor – Prinzip

In der fertigen Version triggert der Uno (später der Pi) den Trig-Pin, liest die Echo-Zeit ein und entscheidet je nach Abstand, ob der Rover bremst, ausweicht oder stoppt. Bis dahin liefern die Drucktaster nur ein „Hindernis ja/nein“ – genug für die ersten Fahrversuche.

3.4 Objekterkennung (Future Work)

Sobald der geplante Raspberry Pi in die *Controller-Box* (Abb. 3.3 a) einzieht, kann der Rover mehr als nur Distanz messen: Die Kamera öffnet die Tür für klassische OpenCV-Algorithmen oder ein kleines CNN (z. B. MobileNet) zur Objekterkennung. Damit ließen sich laterale Hindernisse, farbige Markierungen oder sogar QR-Tags identifizieren und in die Fahrplanung einbeziehen.

Aktuell ist dieses Feature noch nicht implementiert – die Halterung ist allerdings schon gedruckt, und der Software-Stack lässt sich bei Bedarf über pip nachrüsten.

3.5 Steuerlogik

Momentan fährt der Rover noch *open loop*: Die verbauten N20-Motoren besitzen keine Encoder, daher lassen sich Geschwindigkeit und Drehung nur grob per PWM steuern. Geplant ist ein späterer Tausch gegen Varianten mit integrierten Hall-Encodern; damit kann ich dann einen echten PID-Regler aufsetzen und die Bewegungen präzise nachregeln. Bis es so weit ist, dient dieser Abschnitt als theoretische Grundlage.

3.5.1 Mecanum-Räder – Frei bewegen auf vier Rollen

Die schräg montierten Rollen eines Mecanum-Rads leiten den Antriebsschub seitlich ab. Kombiniert man die Drehrichtungen der vier Räder, entstehen beliebige Bewegungen:

- **Vor / zurück** – alle Räder drehen gleichsinnig.
- **Seitwärts (Strafing)** – Räder auf einer Diagonale vorwärts, die anderen rückwärts.
- **Rotation** – linke Räder vorwärts, rechte rückwärts (oder umgekehrt).

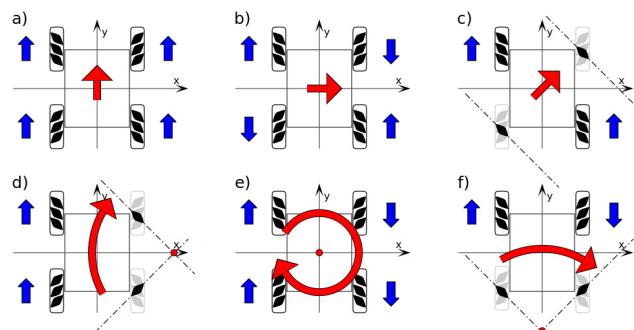


Abbildung 3.5: Kraftvektoren bei verschiedenen Mecanum-Rad kombinationen

Die endgültige Radgeschwindigkeits matrix soll später ein PID-Regler berechnen; bis dahin nutze ich feste PWM-Profilen, um das Fahrwerk zu testen und Referenzdaten für die spätere Regelung zu sammeln.

Kapitel 4

Entwicklung eines eigenen LLMs

4.1 Datensatz

Für das LLM habe ich drei Datensorten gebaut:

- **Labelled** – synthetische Befehle → JSON-Ziele (Tab. 4.1, 4.2)
- **Unlabelled** – rohe Befehle ohne Annotation (Tab. 4.3)
- **Test** – handgetippte, bewusst schräge Beispiele (Tab. 4.4)

Alle JSON-Ausgaben folgen der Struktur aus Abschnitt 5.4; so kann ich später ohne Bruch neue Felder hinzufügen. *Warum so viel Unlabelled?* Rohe Textbefehle sammelt man schnell (Web-Scraper + ChatGPT-Paraphrasen), sauber gelabelte Paare dauern. Das feingetunte Teacher-Modell erzeugt daher zur Laufzeit Pseudo-Labels (*Distillation*, vgl. §4.7.1) und spart Man-Power.

4.1.1 Labelled Data

Zwei Schärfegrade:

1. **Basic** – nur das Nötigste (distance, angle,...)
2. **Full** – plus Extras (momentan acceleration)

Erzeugung. Die Scripte `generate_full_labelled_data.py` und `generate_basic_labelled_data.py` mischen

- *Templates* (z. B. "move \$DIR \$DIST \$UNIT")
- *Parameter-Sampler* für Distanz, Winkel, Beschleunigung
- kleines Synonym-Lexikon (forward ↔ ahead ...)

Damit entstehen rund **64 068** JSON-Befehle (42 000 Basic + 22 068 Full).

Tabelle 4.1: Example Basic Labelled Data

Input Text	Expected JSON Output
move forward 50 cm.	{ "command": "forward", "parameters": {"distance": 50} }
stop.	{ "command": "stop", "parameters": {} }

Tabelle 4.2: Example Full Labelled Data

Input Text	Expected JSON Output
move forward 50 cm with acceleration 12.3 cm/s ² .	{ "command": "forward", "parameters": { "distance": 50, "acceleration": 12.3}}
rotate left 30 deg with angular acceleration 25.6 deg/s ² .	{ "command": "rotate", "parameters": { "angle": 30, "direction": "left", "acceleration": 25.6}}

4.1.2 Unlabelled Data

153 742 unlabelte Befehle – zusammengesetzt aus

- 100 k Basic-Prompts (Web-Crawler + Rule-Sampler),
- 45 k Full-Prompts (siehe Script `generate_full_unlabelled_data.py`),
- 8 k ChatGPT-Paraphrasen.

Nach Dedup-Filter verbleiben die oben genannten 153 k, auf die der Teacher Soft-Targets projiziert.

Tabelle 4.3: Vergleich Basic vs. Full Unlabelled Data

Basic	Full
move forward 50 cm.	move forward 50 cm with acceleration 12.3 cm/s ² .
rotate left 30 deg.	rotate left 30 deg with angular acceleration 25.6 deg/s ² .

4.1.3 Test Data

Nicht der Synthesizer, sondern echte Leute: Dialekt, Tippfehler, verschachtelte Anweisungen. Aktuell 1 032 handkurierte Beispiele (Tab. 4.4), um früh zu prüfen, wo das Modell stolpert.

Tabelle 4.4: Auszug aus den hand gepflegten Test Data

Eingabe	Besonderheit
Could you move a bit to the left?	Alltagssprache, indirekt
Please proceed backward 25 cm, then rotate left 60 deg.	Mehrfachbefehl
Stop now!!!	Dringlichkeit, Satzzeichen-Spam
Advance 20 deg?	Unklare Einheit (Winkel vs. Distanz)
Go 100 forward quickly	Fehlendes Einheitstoken, Adverb

Der Datensatz deckt damit die volle Bandbreite von *sauber annotiert* bis *komplett roh* ab – ideal für Supervised Fine-Tuning, Distillation und robuste Härtetests.

4.2 Tokenizer & Normalisierung

Weg vom Eigenbau. Mein erster Versuch war ein selbst trainierter BPE-Tokenizer mit einer recht aufwendigen Regex-Normalisierung. Er funktionierte, hatte aber drei Nachteile: viel Pflege, spürbar langsamer beim Vorverarbeiten großer Korpora und trotzdem noch Lücken bei exotischen Eingaben.

Aktueller Stand. Inzwischen nutze ich einfach den *Fast-Tokenizer des Basismodells* (`google/t5-v1_1-large`). Der bringt einen robusten Byte-Level-Normalizer, Unicode-Support und die wichtigsten JSON-Tokens von Haus aus mit – völlig ohne Extra-Skripte. Fehlen später Sondertokens (z. B. neue Einheiten), kann ich sie in zwei Zeilen nachträglich hinzufügen; das Basis setup bleibt aber schlank und wartungsfrei.

Beispielhafte Ausgabe

Tabelle 4.5: So zerteilt der aktuelle Tokenizer zwei typische Befehle

Eingabe	Tokenfolge (gekürzt)
Move forward 50 cm.	<code><s>_Move _forward _50 _cm. </ s></code>
Rotate left by 30 deg with acceleration 25.67 deg/s ² .	<code><s>_Rotate _left _by _30 _deg _with _acceleration _25.67 _deg/s2. </ s></code>

Kurz gesagt: Der Google-Tokenizer erledigt den Job zuverlässiger als mein Eigenbau und spart mir Wartung, also bleibt er vorerst im Einsatz.

Post-Processing & Ausblick

Nach dem reinen Tokenisieren hängt die Pipeline derzeit noch ein kleines *Post-Processing* dran: Vor jedem Prompt kommt automatisch `<s>`, ans Ende ein `</ s>`.

Das reicht, weil T5 die JSON-Sequenz sowieso als Fließtext ausgibt.

Für die nächste Iteration möchte ich das jedoch direkt in das Tokenizer-Training ziehen. Ziel:

- ein eigenes <sep>-Token, damit Eingabe und erwartetes JSON klar getrennt sind;
- BOS/EOS und Padding als fester Bestandteil des Tokenizers, nicht mehr als Hack im Data-Collator;
- optionale Slot-Tags wie <cmd> oder <param> für feinere Kontrolle, sobald das Modell komplexere Rückmeldungen liefern soll.

Sobald das ansteht, werde ich den Google-Tokenizer als Basis nehmen, Spezialtokens anhängen und das Vokabular per *post-training* nachjustieren – die Grundstabilität bleibt, der Output wird präziser.

4.3 Distillation

Zweck. Ein ausgewachsener Encoder-Decoder wie Flan-T5-Small reicht für die Roboterbefehle völlig aus – ist aber immer noch zu groß, um später auf einem Mikro-GPU-Board in Echtzeit zu laufen. Darum komprimiere ich sein Wissen in ein noch kleineres Student-Netz: (1) Teacher fein tunen, (2) Soft-Targets für unlabelte Befehle erzeugen.

4.3.1 Teacher – Supervised Fine-Tuning

- **Daten** – 120 k synthetische Label-Paare.
- **Training** – 2 Epochen, Batch 4, LR 3×10^{-4} , *constant* Scheduler ohne Warm-up.
- **Sequenzlänge** – 512 Token (Ein- und Ausgabe).
- **LoRA** – Rank 8, Alpha 16, Dropout 0.05; Basis gewichte in 8-bit (Quantisierung → §2.3).
- **Tokenizer** – Standard-Tokenizer des Basismodells (s. §4.2).
- **Checkpoint** – runs/teacher_v3/.

Tabelle 4.6: Teacher-Hyperparameter (aktuelle Konfiguration)

Parameter	Wert
Basismodell	google/flan-t5-small
Parameterzahl	~80 M
Epochen	16
Batchgröße	8
Lernrate	3×10^{-4}
Scheduler	constant (kein Warm-up)
Max. Seq-Länge	512
Optimizer	AdamW
Weight Decay	0.01
LoRA Rank / Alpha	8 / 16
8-bit Basis	ja

4.3.2 Soft-Target-Extraktion

- **Korpus** – 110 k deduplizierte unlabelte Befehle.
- **Inference** – greedy Decode, max_new_tokens 128, Top-5 Probs bei Temperatur 1.0.
- **Speicher** – komprimierte JSON-Liste `soft_targets_v3.json.gz` (~ 20 MB).

Die Top-5-Wahrscheinlichkeiten je Token dienen als weiche Ziele fürs Student-Training; das kleine Modell muss den Teacher dabei nie laden.

4.3.3 Ausblick

Genauigkeits- und Speichervergleiche folgen in Kapitel ???. Wichtig hier: Dank 8-bit-Gewichten passt der Teacher in < 3 GB VRAM, und die Destillation liefert einen handlichen Soft-Target-Korpus als Basis für die nächste Trainingsphase.

4.4 Quantisierung

Die Gewichte des Teacher- und später auch des Student-Netzes lagern nicht komplett in FP32, sondern werden vor dem Training auf **8-Bit** komprimiert. Warum das funktioniert und wann es schief-geht, habe ich in §2.3 (Grundlagenkapitel) zusammengefasst – hier nur das Kurz-How-To für mein Setup:

- **BitsAndBytes** (`bnn_literal_quant`) legt beim AutoModel-Laden alle Linear-Layer in 8-Bit an.
- Die eigentlich zu lernenden **LoRA**-Adapter bleiben in 16-Bit – sonst kollabiert die Gradientenpräzision.

- Optimizer (AdamW) und Checkpoints behandeln Quant-Layer transparent; beim Speichern landet nur der LoRA-Delta, die 8-Bit-Basismatrix bleibt unverändert.
- Ergebnis: GPU-RAM halbiert, Throughput \uparrow ca. 40

Damit passt selbst `flan-t5-small` in eine einzelne Laptop-GPU – und das Feintuning kostet eher Minuten als Stunden.

4.5 Student-Modell und Training

Das Student-Modell baut auf einer kompakten Mini-T5-Architektur auf, um trotz geringer Größe eine effiziente und zuverlässige Steuerung des Roboters zu gewährleisten. Die wesentlichen Aspekte sind:

4.5.1 Mini-T5-Architektur

- **Basismodell:** Die Architektur basiert auf dem vortrainierten `flan-t5-small`-Modell, das auf insgesamt 4 Encoder- und 4 Decoder-Schichten sowie eine reduzierte interne Dimension von $d_{\text{model}} = 256$ gekürzt wurde. Damit ist es ideal auf Ressourcenbeschränkungen zugeschnitten (vgl. Theorie zur Transformer-Architektur in Abschnitt 2.6).
- **Tokenizer:** Verwendet wird derselbe Tokenizer wie beim Teacher-Modell (siehe § 4.2), wodurch eine problemlose und direkte Übernahme des Vokabulars und damit eine konsistente Eingabeverarbeitung sichergestellt ist.
- **Parameterbudget:** Das finale Modell umfasst rund 42 Mio. Parameter und wird zudem 8-Bit quantisiert gespeichert und ausgeführt, um Speicherplatz und Rechenzeit erheblich einzusparen.

4.5.2 Distillations-Setup

- **Datengrundlage:** Trainiert wird auf insgesamt etwa $1,5 \times 10^5$ unannotierten Befehlen, ergänzt um weiche Lernziele (Soft-Targets) aus dem Teacher-Modell, wobei stets die Top-5 Wahrscheinlichkeiten bei Temperatur $T = 1$ verwendet werden (vgl. Abschnitt `refdistillation_theory`).

- **Verlustfunktion:** Die Trainingsverluste setzen sich zusammen aus einer gewichteten Kombination von harter Cross-Entropy (CE) für klare Zielvorgaben sowie einer weichen Kullback-Leibler-Divergenz (KL) zur besseren Generalisierung auf semantischer Ebene:

$$\text{Loss} = 0.6 \cdot \text{CE}_{\text{hard}} + 0.4 \cdot \text{KL}_{\text{soft}}$$

Detaillierte theoretische Grundlagen zu diesen Verlustfunktionen finden sich in Abschnitt `flossFunctions_theory`.

4.5.3 Trainingsablauf

Der Trainingsprozess umfasst folgende Eckpunkte:

- **Batchgröße und Optimierung:** Trainiert wird mit einer Batchgröße von 32 und einer Lernrate von 5×10^{-4} über insgesamt 12 Epochen. Als Optimizer kommt AdamW zum Einsatz, mit gemischter Präzision (FP16) zur Beschleunigung des Trainings.
- **Checkpointing und Early-Stopping:** Nach jeder Epoche wird ein Checkpoint gespeichert. Sollte die Validierungsleistung über zwei aufeinanderfolgende Epochen nicht verbessert werden, erfolgt ein vorzeitiger Trainingsabbruch (Early-Stopping).

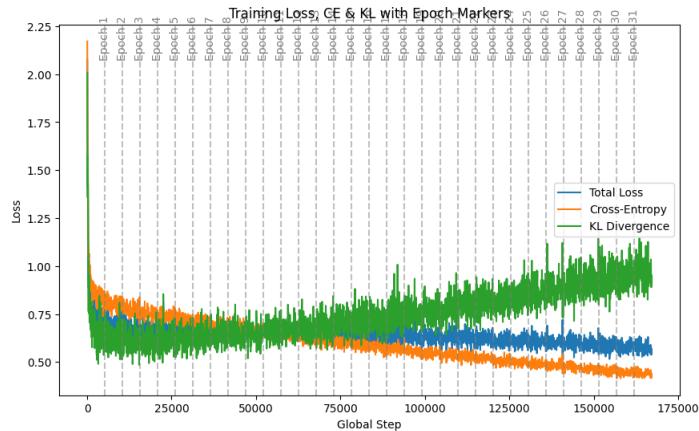


Abbildung 4.1: Trainingsverlauf – Cross-Entropy (CE) und KL-Divergenz.

Dieses sorgfältig abgestimmte Setup erlaubt dem Student-Modell, komplexe Steuerbefehle zuverlässig umzusetzen und von den robusten semantischen Fähigkeiten des Teacher-Modells zu profitieren.

4.6 Architektur

4.6.1 Allgemeine Übersicht

Meine geplante LLM-Architektur folgt dem klassischen Transformer-Design von Vaswani et al. [Vas+17] und gliedert sich in vier Bausteine (siehe Abb. 4.2):

1. **Tokenisierung & Input-Embedding:** Rohtext → Token → Vektoren.
2. **Encoder:** Self-Attention gefolgt von Feed-Forward-Netzwerk für kontextualisierte Repräsentationen.
3. **Decoder:** Maskierte Self-Attention plus Cross-Attention auf den Encoder-Output.
4. **Output-Embedding:** Projektion der Decoder-Ausgabe auf eine Wahrscheinlichkeitsverteilung über das Vokabular.

Optional lässt sich über Weight-Tying die Zahl der Embeddings reduzieren und so die Modellgröße etwas drücken. Leider reichte der lokale VRAM nicht aus, um dieses volle Setup wirklich in Originalgröße zu trainieren – hier musste ich stark verkleinern oder Filtervarianten auf Colab ausprobieren.

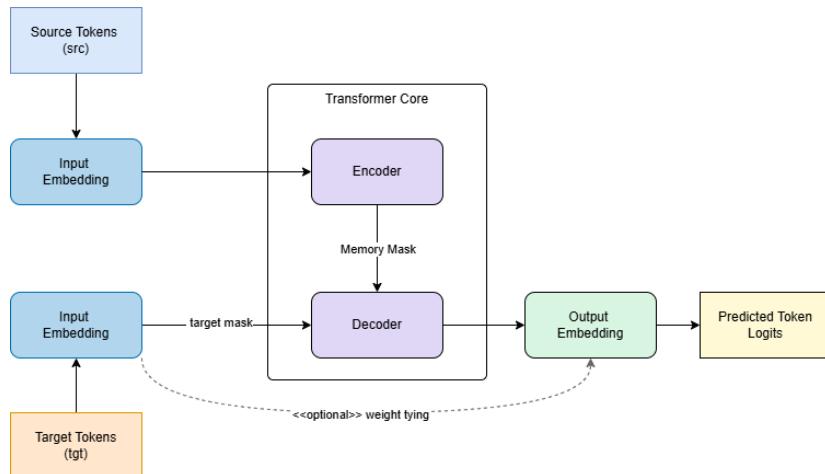


Abbildung 4.2: High-Level Übersicht der Transformer-Architektur des LLMs **Quelle: Eigene Darstellung**

Im Folgenden werden die einzelnen Architekturkomponenten detaillierter beschrieben und erläutert.

4.6.2 Residual- und Normalisierungsschicht

Diese Schicht kombiniert zwei wesentliche Schritte: Zuerst wird der ursprüngliche Eingang mit der Ausgabe eines Sublayers addiert, um den direkten Informationsfluss beizubehalten (Residualverbindung). Anschließend wird durch Dropout eine

Regularisierung erzielt und mittels Layer-Normalization der kombinierte Output standardisiert. Diese Vorgehensweise stabilisiert das Training und verbessert den Gradientenfluss in tiefen Netzwerken.

4.6.3 Feed-Forward-Netzwerk (FFN)

In jeder Transformer-Schicht steckt ein kleines MLP (FFN), das jedes Token einzeln „aufpeppt“:

$$x \rightarrow \text{Linear} \rightarrow \text{GELU} \rightarrow \text{Dropout} \rightarrow \text{Linear} \rightarrow x'.$$

Damit können nichtlineare Zusammenhänge abgebildet werden (vgl. Theorie in Abschnitt 2.6.2).

Ich nutze zwei Varianten:

- **Standard-FFN:** Klassischer Ablauf Linear → GELU → Dropout → Linear.
- **DeepSeek-FFN:** Fügt vorab eine Layer-Normalization und ein Gating hinzu, um den Gradientenfluss noch robuster zu machen.

Für schnelle Prototypen reicht meist schon das klassische FFN, während die DeepSeek-Option bei Bedarf zugeschaltet wird.

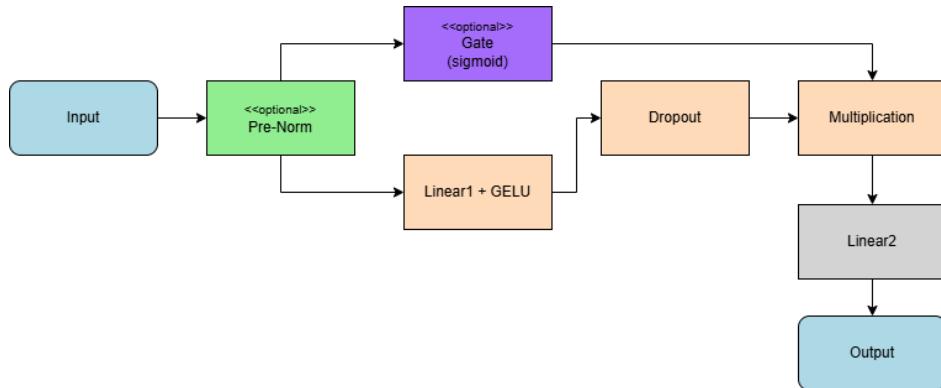


Abbildung 4.3: Übersicht des DeepSeek-FFN. Quelle: eigene Darstellung

In Abb. 4.3 sieht man rechts den klassischen FFN-Pfad (Linear+GELU → Dropout → Linear). Links sind zwei optionale Extras eingezeichnet:

- *Pre-Norm*, das den Input vor den linearen Schritten normalisiert,
- ein *Gate* (sigmoid), das die Aktivierung aus dem ersten Linear-GELU-Block skaliert.

Beide Mechanismen zielen darauf ab, den Gradientenfluss zu stabilisieren und das Training robuster zu machen.“

4.6.4 Aufmerksamkeitsmechanismus (Attention)

Attention ist das Herz des Transformers: Jede Position in der Eingabesequenz vergleicht sich mit allen anderen und zieht relevante Informationen heran. Formal sieht das so aus:

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

wobei Q , K und V aus den Token-Embeddings abgeleitet werden und d_k die Dimension der Keys ist (vgl. Abb. 2.10 und Theorie in Abschnitt 2.6.1).

Multi-Head Attention teilt diesen Prozess in mehrere „Köpfe“ auf, damit das Modell gleichzeitig unterschiedliche Kontext-Aspekte lernt (siehe ebenfalls [Vas+17]).

4.6.5 Positions kodierung

Damit der Transformer auch die *Reihenfolge* versteht, bekommt jedes Token eine Positionsinfo dazu (vgl. Theorie in Abschnitt 2.6.3). Ich habe zwei gängige Varianten getestet:

- **Sinusoidal (fix):** Vordefinierte sin / cos-Kurven – schnell, braucht keine Extra-Parameter.
- **Learned:** Das Modell lernt die Positionsvektoren selbst; passt sich besser an Daten an, kostet aber ein paar Parameter.

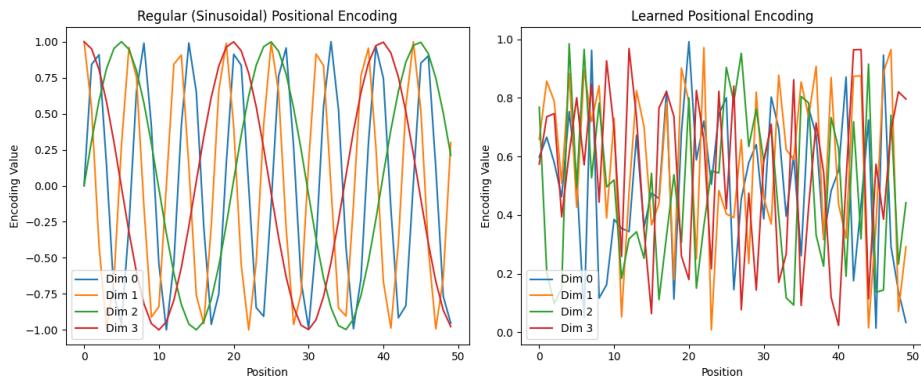


Abbildung 4.4: Regular vs Gelernt

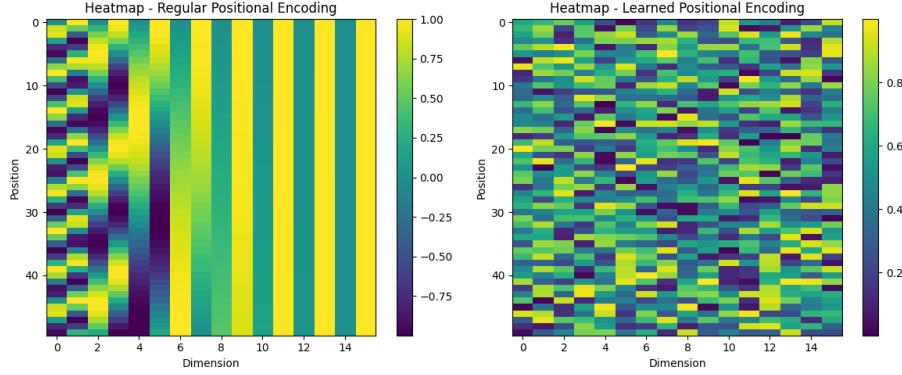


Abbildung 4.5: Regular vs Gelernt Heatmap

Man sieht: links schön periodisch, rechts wild und flexibel. Für kleine Modelle reicht meist die sinusförmige Variante; ob die gelernte Kodierung bei größerem VRAM-Budget tatsächlich Mehrwert bringt, muss ich erst noch praktisch testen.

4.6.6 Einbettungs- und Ausgabeschicht

Die Einbettungsschicht (*Embedding-Layer*) wandelt Eingabetoken-Indizes in kontinuierliche, dichte Vektorrepräsentationen fester Größe (d_{model}) um. Dabei integriert sie zusätzlich eine Positions kodierung, welche entweder sinusförmig oder gelernt sein kann. Dies ermöglicht dem Modell, Sequenzpositionen zu erfassen (vgl. Abschnitt 4.6.5).

Die Ausgabeschicht projiziert die vom Decoder erzeugten Repräsentationen zurück in den Tokenraum, indem sie eine lineare Transformation von d_{model} zur Größe des Tokenvokabulars durchführt. Optional können hierbei die Gewichte der Eingabe- und Ausgabeeinbettung zur Effizienzsteigerung gekoppelt werden (*Weight Tying*). Beide Schichten sind essenziell für die Transformation diskreter Eingabedaten in eine für das Modell nutzbare Form und deren anschließende Rücktransformation zur diskreten Ausgabe.

4.6.7 Cross-Attention

Die Cross-Attention stellt die Verbindung zwischen Encoder und Decoder in der Transformer-Architektur her. Während der Decoder zunächst mithilfe der maskierten Self-Attention seine bisher erzeugten Tokens verarbeitet, greift er anschließend mittels Cross-Attention zusätzlich auf die Informationen des Encoders zurück. Konkret verwendet die Cross-Attention die Ausgaben des Encoders als *Key*- und *Value*-Vektoren, während die aktuellen Decoder-Token als *Query*-Vektoren dienen. So kann jedes Decoder-Token relevante Informationen aus der gesamten Eingabesequenz

abrufen, unabhängig von deren Position. Die genaue Berechnung erfolgt mittels des Dot-Product-Attention-Mechanismus (siehe Abschnitt 4.6.4).

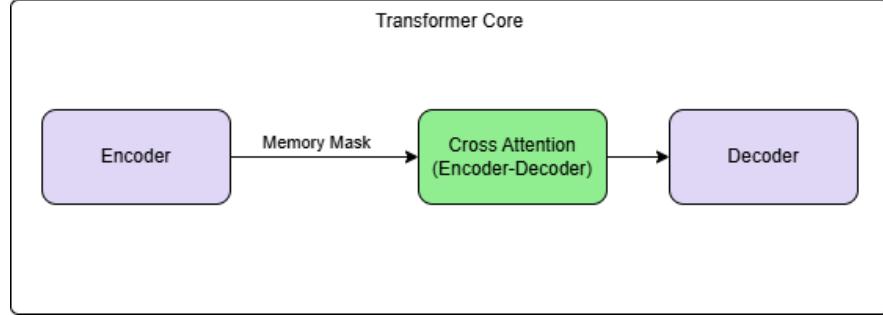


Abbildung 4.6: Darstellung der Cross-Attention innerhalb des Transformers. **Quelle:** Eigene Darstellung

Abbildung 4.6 verdeutlicht diesen Ablauf: Die Encoder-Ausgaben werden zusammen mit einer optionalen *Memory Mask* an die Cross-Attention übergeben. Diese Maske ermöglicht es, gezielt unwichtige Tokens wie Padding-Elemente zu ignorieren. Die Decoder-Queries wählen dann über Attention-Gewichte genau jene Eingabeinformationen aus, die für die aktuelle Ausgabe relevant sind. Auf diese Weise generiert der Decoder kontextsensitiv und eingabebezogen – eine zentrale Eigenschaft für Aufgaben wie maschinelle Übersetzung oder das Umsetzen von natürlichen Befehlen in ausführbare Anweisungen.

4.6.8 Encoder und Decoder

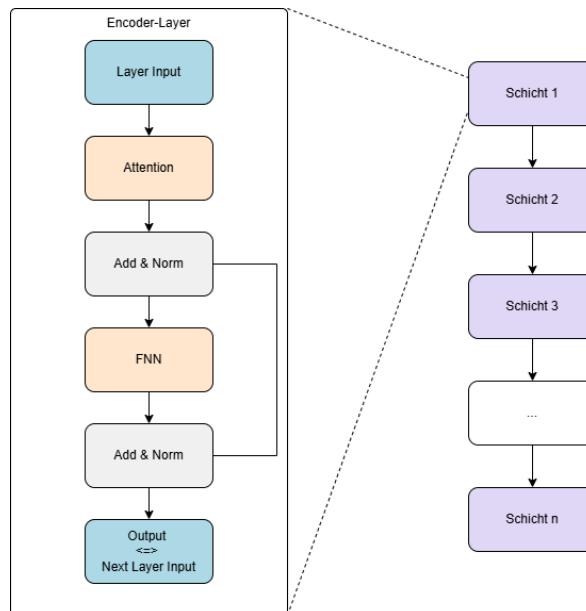


Abbildung 4.7: Aufbau einer Encoder-Schicht und deren Einbettung in den Encoder-Stack. **Quelle:** Eigene Darstellung

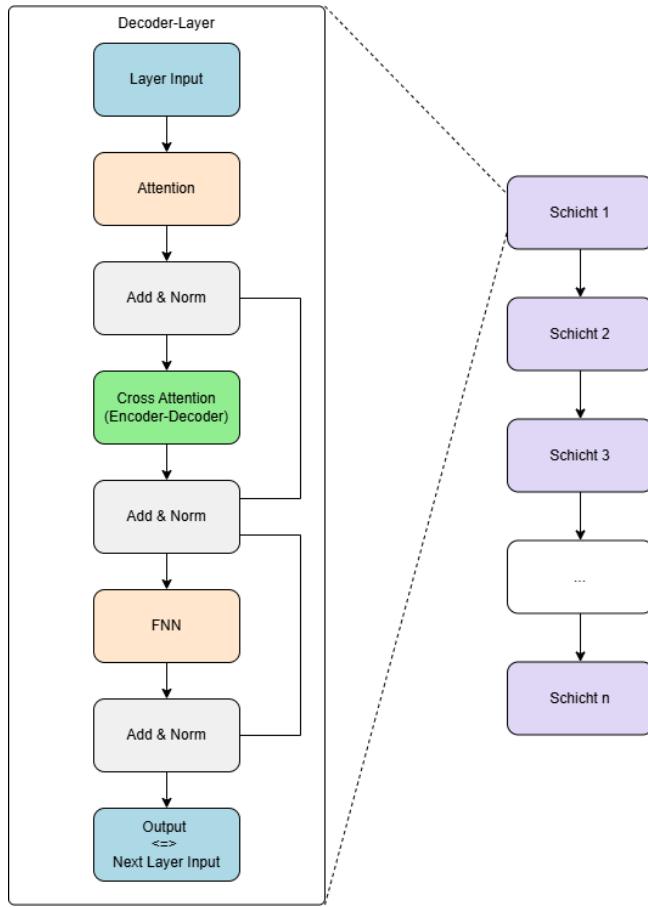


Abbildung 4.8: Aufbau einer Decoder-Schicht und deren Einbindung in den Decoder-Stack. **Quelle: Eigene Darstellung**

Wie bereits in Abschnitt 2.5.3 erläutert, besteht ein Transformer-Modell grundsätzlich aus zwei Hauptkomponenten: dem Encoder und dem Decoder. Der Encoder verarbeitet eine Eingabesequenz und erzeugt eine kompakte, kontextabhängige Darstellung der Daten. Der Decoder nutzt diese Repräsentation anschließend, um schrittweise eine passende Zielsequenz zu generieren.

Encoder

Der Encoder setzt sich aus mehreren identischen Schichten zusammen. Jede dieser Schichten enthält eine sogenannte *Multi-Head Self-Attention* sowie ein Feed-Forward-Netzwerk (FFN), wie in Abbildung 4.7 dargestellt. Die Eingabesequenz wird zunächst um Positionskodierungen erweitert, damit das Modell auch die Reihenfolge der Tokens berücksichtigt (siehe Abschnitt 2.6.3). Danach berechnet die Self-Attention für jedes Token, wie relevant jedes andere Token in der Sequenz ist. Das erlaubt dem Modell, wichtige Zusammenhänge in der Eingabesequenz zu erfassen.

Nach der Attention folgt eine Add & Norm-Schicht, bei der das Ausgangssignal mit dem Eingangssignal kombiniert (Residualverbindung) und anschließend normalisiert wird. Dies verbessert die Stabilität während des Trainings. Danach verarbeitet

das Feed-Forward-Netzwerk jedes Token individuell weiter. Es transformiert die Token in einen höher dimensionalen Raum, aktiviert sie nichtlinear und transformiert sie anschließend zurück in die ursprüngliche Dimension. Zum Abschluss erfolgt erneut eine Add & Norm-Schicht. Diese Struktur nennt man die Post-Norm-Variante, bei der die Normalisierung nach den jeweiligen Operationen ausgeführt wird.

Encoder-Stack

In der Praxis enthält der Encoder mehrere solcher identischen Schichten, die nacheinander geschaltet sind (vgl. Abb. 4.7). Dadurch werden die Eingabeinformationen schrittweise vertieft und die entstandenen Repräsentationen zunehmend kontextreich. Typischerweise umfasst ein Encoder zwischen 6 und 12 solcher Schichten, um komplexe Beziehungen innerhalb der Eingabe gut abzubilden. Als letzten Schritt nutzt man eine finale Layer-Normalisierung, um die erzeugten Repräsentationen stabil zu halten und optimal auf den Decoder vorzubereiten.

Decoder

Der Decoder generiert schrittweise eine Ausgabesequenz auf Basis der vom Encoder gelieferten Informationen. Jede Schicht im Decoder besteht ebenfalls aus mehreren Komponenten: einer maskierten Self-Attention, einer Cross-Attention und einem Feed-Forward-Netzwerk (vgl. Abb. 4.8).

Die maskierte Self-Attention stellt sicher, dass jedes Token im Decoder nur Informationen über bereits generierte Tokens verwenden kann und zukünftige Tokens ausgeblendet bleiben. Dies ist essenziell für eine korrekte schrittweise (autoregressive) Erzeugung der Zielsequenz. Nach dieser Attention folgt wiederum eine Add & Norm-Schicht, die für Stabilität sorgt.

Im nächsten Schritt erfolgt die Cross-Attention. Hier verwenden die aktuellen Decoder-Tokens die Encoder-Ausgaben, um kontextrelevante Informationen zu erhalten. Die Decoder-Tokens sind dabei Queries, während die Encoder-Ausgaben als Keys und Values dienen (siehe Abschnitt 4.6.7). Auch hier erfolgt anschließend eine weitere Add & Norm-Schicht.

Abschließend folgt ein Feed-Forward-Netzwerk, das jedes Token weiter transformiert und erneut von einer Add & Norm-Schicht abgeschlossen wird.

Decoder-Stack

Ähnlich wie der Encoder besteht der Decoder aus einer Reihe identischer Schichten (vgl. Abb. 4.8). Durch das wiederholte Anwenden dieser Schichten entsteht ein immer besseres Verständnis der Zielsequenz unter Berücksichtigung der Encoder-Informationen. Meist verwendet man auch hier zwischen 6 und 12 Schichten. Zum

Abschluss des Decoder-Stacks erfolgt nochmals eine Layer-Normalisierung, um stabile und hochwertige Repräsentationen für die finale Ausgabe zu erzeugen.

Insgesamt ermöglicht die hier beschriebene Encoder-Decoder-Architektur, dass komplexe semantische Zusammenhänge zwischen Eingabe- und Zielsequenz effektiv modelliert und in der Ausgabe berücksichtigt werden können.

4.7 Training

4.7.1 Teacher – Supervised Fine-Tuning

- **Daten:** $\approx 64\,068$ synthetische Label-Paare (Basic + Full; vgl. §4.1.1).
- **Modell:** google/flan-t5-small (~ 80 M Parameter).
- **Training:** 16 Epochen, Batchgröße 8, Lernrate $\eta = 3 \times 10^{-4}$, konstanter Scheduler (kein Warm-up).
- **Optimierer:** AdamW ($\beta_1 = 0.9, \beta_2 = 0.999$, Weight-Decay = 0.01).
- **Sequenzlänge:** max. 512 Token (Input + Output).
- **Quantisierung:** 8-Bit für alle Basismatrizen (BitsAndBytes, vgl. §2.3).
- **Tokenizer:** Der offizielle Flan-T5-Tokenizer (§4.2).
- **Checkpoint:** Speicherung unter `runs/teacher_v3/`.

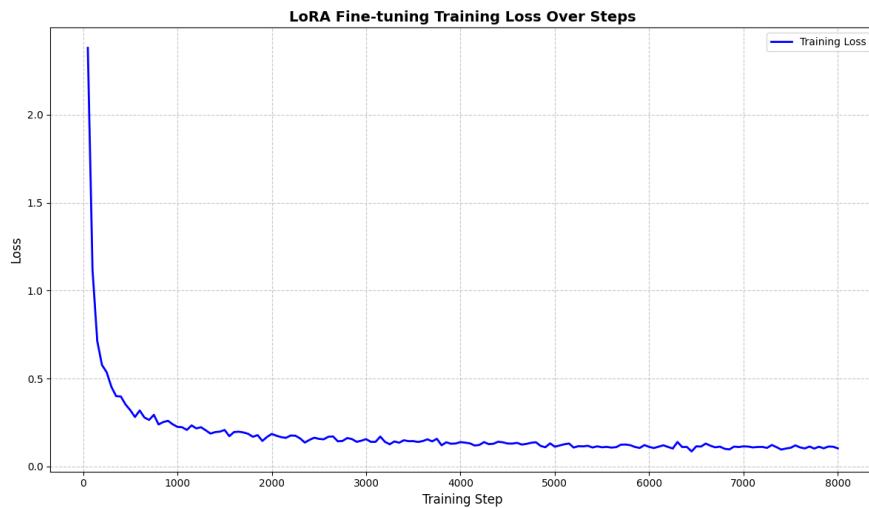


Abbildung 4.9: Trainingskurve: Cross-Entropy für LoRA

Die in Abbildung 4.9 dargestellte Trainingskurve zeigt den Verlauf der Cross-Entropy (CE) während des LoRA-Fine-Tunings des Modells google/flan-t5-small.

Es ist erkennbar, dass der Verlust in den ersten Trainingsschritten stark absinkt und anschließend zunehmend flacher wird. Nach etwa 2000 Schritten stabilisiert sich der Verlust auf einem niedrigen Niveau, was darauf hinweist, dass das Modell zu diesem Zeitpunkt bereits gut auf die Trainingsdaten angepasst ist. Die geringen Schwankungen im späteren Verlauf signalisieren, dass das Modell eine stabile und robuste Anpassung an die Trainingsdaten erreicht hat.

4.7.2 Student-Training

- **Architektur:** Mini-T5 (4 Encoder- / 4 Decoder-Layer, $d_{\text{model}} = 256$, ~ 42 M Parameter), identischer Tokenizer.
- **Distillation:** ~ 150 k unlabeled Beispiele + Top-5 Soft-Targets ($T = 1$, vgl. §2.4), Loss = $0.6 \text{CE}_{\text{hard}} + 0.4 \text{KL}_{\text{soft}}$ (§2.2).
- **Training:** 12 Epochen, Batchgröße 32, Lernrate 5×10^{-4} , AdamW, FP16.
- **Regularisierung:** Gradient Clipping bei Norm 1.0, Early-Stopping nach 2 Ep. ohne Val-Gain.
- **Logging & Checkpoints:** Google Drive für Checkpoints; wandb + TensorBoard für Metriken.

Das Setup läuft in Google Colab Pro; ein kompletter Durchlauf benötigt weniger als eine Stunde und passt komfortabel in 16 GB GPU-RAM.

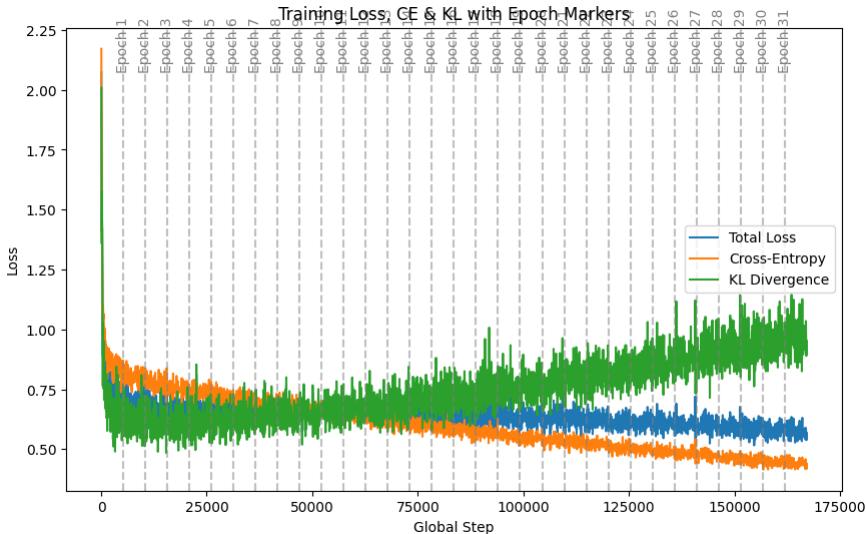


Abbildung 4.10: Trainingskurve: Cross-Entropy KL-Divergenz.

In Abbildung 4.10 ist der Trainingsverlauf der Verlustfunktion dargestellt. Diese setzt sich aus zwei Komponenten zusammen: der Cross-Entropy (CE) und der

Kullback-Leibler-Divergenz (KL-Divergenz). Die Gesamtverlustfunktion ist eine gewichtete Kombination beider Terme:

$$\text{Loss}_{\text{gesamt}} = w_{\text{CE}} \cdot \text{CE} + w_{\text{KL}} \cdot \text{KL}$$

Dabei werden die Gewichte über die Trainingsdauer linear angepasst (*Annealing*). Konkret startet der Trainingsprozess mit einer stärkeren Gewichtung der KL-Divergenz, die schrittweise zugunsten der CE reduziert wird. Dies erfolgt gemäß der folgenden Formel:

$$w_{\text{KL}}(\text{Epoch}) = KL_{\max} - \frac{\text{Epoch}}{\text{Epochs}_{\max} - 1} \cdot (KL_{\max} - KL_{\min})$$

Mit den gewählten Werten von $KL_{\max} = 0,60$ und $KL_{\min} = 0,25$ bedeutet dies, dass die Wichtigkeit der KL-Divergenz im Laufe des Trainings kontinuierlich sinkt, während gleichzeitig der Einfluss der Cross-Entropy zunimmt:

- Zu Beginn (Epoch = 0): $w_{\text{KL}} = 0,60$, $w_{\text{CE}} = 0,40$
- Zum Ende hin (Epoch = $\text{Epochs}_{\max} - 1$): $w_{\text{KL}} = 0,25$, $w_{\text{CE}} = 0,75$

Die Abbildung verdeutlicht, dass sich die beiden Verlustkomponenten gegenläufig entwickeln. Die Cross-Entropy nimmt im Trainingsverlauf stetig ab, was zeigt, dass das Modell zunehmend besser darin wird, die harten Zielwerte präzise vorherzusagen. Gleichzeitig steigt die KL-Divergenz tendenziell an. Dies deutet darauf hin, dass sich das Modell mit zunehmender Dauer vom probabilistischen Lehrer-Modell entfernt, was angesichts der steigenden Dominanz der CE-Komponente ein erwarteter und erwünschter Effekt ist.

Die Varianz in der KL-Kurve ist deutlich höher als in der CE-Kurve. Dies liegt darin begründet, dass die KL-Divergenz weiche Wahrscheinlichkeitsverteilungen vergleicht und daher empfindlicher auf kleinere Schwankungen der Modellvorhersagen reagiert. Insgesamt verdeutlicht der Verlauf eine gesunde Trainingsdynamik: das Modell nutzt zunächst die Lehrer-Vorhersagen als Orientierungshilfe und lernt im weiteren Verlauf eigenständige und präzise Vorhersagen anhand der harten Zielwerte zu treffen.

4.8 Optimierung

- **Quantisierung.** Wie in §2.3 erklärt, laufen alle Basisgewichte in 8-Bit, während die LoRA-Adapter in 16-Bit bleiben – bester Kompromiss aus Speed und Genauigkeit.

- **Gradient Clipping.** Norm = 1.0 verhindert sporadische Explosionsspitzen bei CE + KL-Mix.
- **Learning-Rate-Finder.** Einmalig vor dem ersten Run; beste Plateaukante bei $LR = 5 \times 10^{-4}$.
- **Early-Stopping.** Val-Loss sinkt nach Epoche 8 kaum noch; daher Training → max. 12 Epochen.
- **Zukünftige Tweaks.**
 - 4-Bit NF4-Quantisierung testen (ggf. noch $\times 1.3$ Speed).
 - EMA Weights speichern, um das leicht bessere „moving-average“-Netz für Inferenz zu haben.
 - Learning-Rate-Warm-Restart statt konstantem Scheduler – sollte das übertrainieren der letzten Epochen weiter senken.

Kurzum: Colab + 8-Bit-Weights + LoRA liefern ein alltagstaugliches Setup, das ohne teure Hardware auskommt und trotzdem in unter einer Stunde konvergiert.

Kapitel 5

Integration von Roboter und LLM

5.1 Systemübersicht

In meinem Prototyp steuere ich den Roboter per natürlicher Sprache – alles in Python umgesetzt. Über eine einfache Streamlit-Weboberfläche tippe ich meinen Befehl ein. Dieser Text wird automatisch bereinigt (Rechtschreibung, Einheiten, Zahlen) und anschließend an mein lokal laufendes LLM weitergeleitet (vgl. Kapitel 4), das ihn in ein kompaktes JSON mit Parametern wie `direction`, `distance` und `speed` übersetzt.

Anschließend sende ich das JSON per FastAPI an meinen lokalen Server (derzeit ein PC). Dort prüfe ich die Nutzereingabe und leite sie per HTTP an den Arduino weiter, der die Motoren direkt ansteuert. Eine einfache Validierung fängt dabei ungültige Befehle ab.

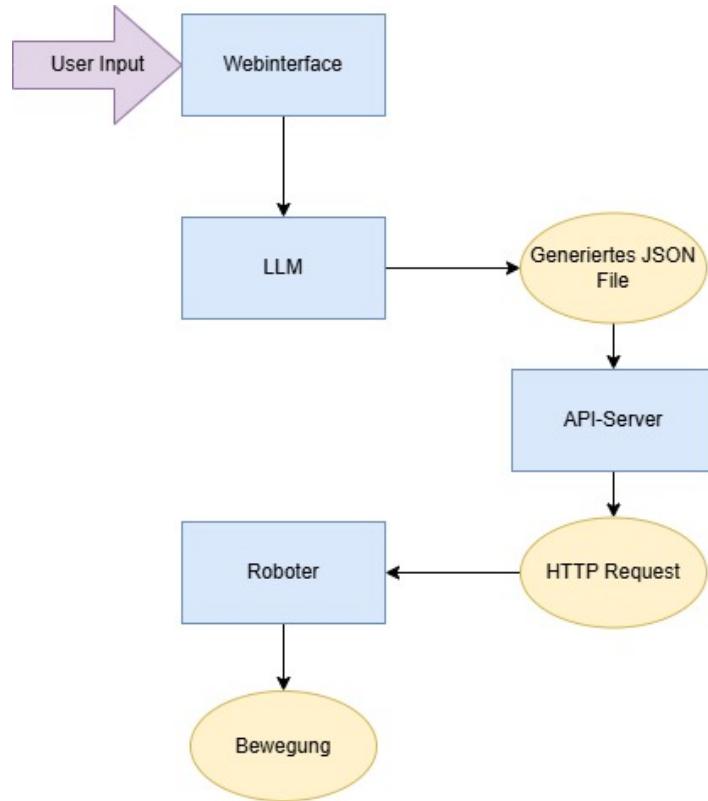


Abbildung 5.1: End-to-End–Datenfluss: Von meiner Spracheingabe über das LLM bis zur Ausführung auf dem Arduino. **Quelle:** Eigene Darstellung

In Zukunft plane ich, den Arduino durch einen Raspberry Pi zu ersetzen und die API auf einen externen Server auszulagern, um zusätzliche Sensorik und Rückmeldemechanismen zu integrieren.“

5.2 Webinterface

Für die Mensch-Maschine-Interaktion nutze ich weiterhin **Streamlit**. Damit bleiben Front- und Backend komplett in Python, die Installation ist minimal und Änderungen sind in Sekunden live.

Chat-Workflow mit Bot-Auswahl

1. Der Benutzer wählt im linken Dropdown den gewünschten Autobot: *Bumble Bee* (Cloud-Gemini), *Optimus Prime* (lokales Student-LLM) oder *Primus* (Teacher-LLM).
2. Beim ersten Aufruf werden alle Modelle im Hintergrund vor-geladen (Warm-Up). Ein späterer Wechsel geschieht ohne Wartezeit.
3. Jede Benutzereingabe erscheint sofort in der Chat-History.

- Für **Optimus** und **Primus** wird der Text zuerst durch die in Kapitel ?? beschriebenen Pre-Processing-Schritte geleitet (*normalisation* → *spell check* → *unit conversion*).
 - **Bumble Bee** verarbeitet den Rohtext unverändert, um Dialog-Freiheit zu demonstrieren.
4. Das jeweilige Modell liefert ein JSON-Kommando zurück; Streamlit leitet es per HTTP an die FastAPI-Schicht und von dort an den Arduino-Controller.

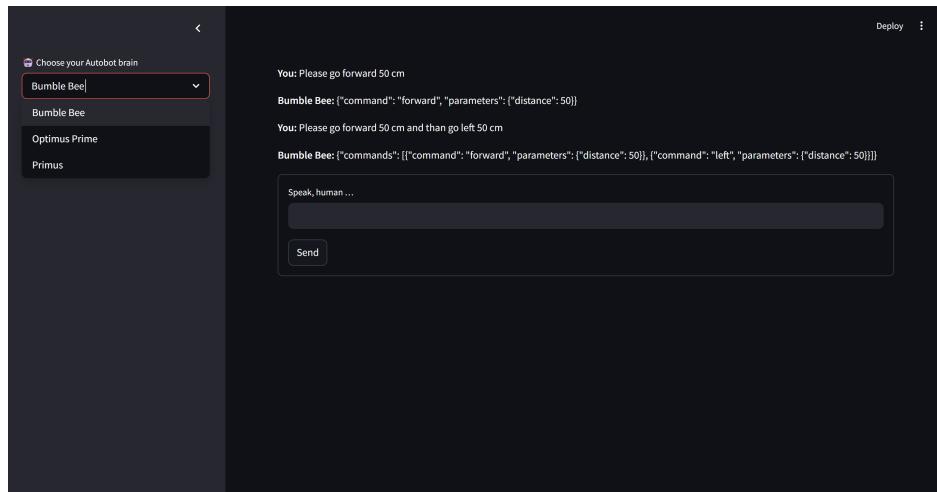


Abbildung 5.2: Aktualisierte Streamlit-Oberfläche mit Bot-Auswahl und Chat-Verlauf.
Quelle: Eigene Darstellung

Vorteile des Ansatzes

- **Modularität:** UI, LLM-Wrapper und Roboter-API sind klar getrennt, kommunizieren jedoch über einfache Python-Objekte.
- **Performance:** Dank des Einmal-Warm-Ups und Pre-Processings nur bei lokalen Modellen bleibt die Latenz niedrig.
- **Erweiterbarkeit:** Neue Bots oder zusätzliche Vorverarbeitungs-Schritte können per Plug-in-Modul ergänzt werden (vgl. `models/<name>.py`).

Somit erfüllt das Interface alle Anforderungen: einfache Bedienung, geringe Wartezeiten und eine saubere Abstraktion zwischen NLP-Logik und Robotik-Hardware.

5.3 Vorverarbeitung der Nutzereingaben

Bevor eine Benutzereingabe vom Sprachmodell verarbeitet wird, durchläuft sie eine mehrstufige Vorverarbeitung, die sicherstellt, dass das Modell robuste und fehlerfreie

JSON-Kommandos erzeugt. Die Pipeline basiert auf drei zentralen Schritten, die im Folgenden näher erläutert werden:

1. **Normalisierung und Rauschfilterung:** In diesem Schritt werden alle Eingaben vereinheitlicht, indem beispielsweise der gesamte Text in Kleinbuchstaben umgewandelt und überflüssige oder fehlerhafte Zeichen entfernt werden. Weitere Details finden Sie in Abschnitt 5.3.1.
2. **Rechtschreibkorrektur:** Mithilfe eines auf `pyspellchecker` basierenden Moduls werden häufige Tippfehler automatisch korrigiert, ohne dabei spezifische Einheitstoken zu verändern. Eine ausführliche Beschreibung finden Sie in Abschnitt 5.3.2.
3. **Numerische Verarbeitung:** Dieser Schritt umfasst die Umwandlung ausgeschriebener Zahlen in numerische Werte sowie die Erkennung, Standardisierung und Umrechnung physikalischer Einheiten (z. B. cm, cm/s, rad). Weiterführende Informationen dazu sind in Abschnitt 5.3.3 zu finden.

Zur Veranschaulichung des Verarbeitungskonzepts zeigt Tabelle ?? Beispiele, wie unterschiedliche Eingaben vorverarbeitet werden. Die Beispiele sind in deutscher Sprache formuliert, während das Modell tatsächlich ausschließlich englischsprachige Eingaben verarbeitet.

5.3.1 Normalisierung und Rauschfilterung

In diesem Schritt der Vorverarbeitung wird der ursprüngliche Texteingabetext standardisiert und von störenden Zeichen bereinigt, um eine konsistente Basis für die weiteren Verarbeitungsschritte zu schaffen. Dieser Prozess umfasst mehrere Teilschritte:

- **Lowercase-Normalisierung:** Der gesamte Text wird in Kleinbuchstaben konvertiert, was die Vergleichbarkeit und Verarbeitung vereinfacht.
- **Erhaltung spezifischer Asterisk-Kontexte:** Mittels regulärer Ausdrücke werden Asterisks in relevanten Kontexten (z. B. in mathematischen Ausdrücken oder Einheitendarstellungen wie $\text{cm} * \text{s}^{-2}$) erkannt und durch einen Platzhalter ersetzt. Dadurch bleibt die Bedeutung mathematischer Ausdrücke erhalten.
- **Rauschfilterung:** Über einen Whitelist-Ansatz werden alle nicht wesentlichen oder störenden Zeichen entfernt. Dabei werden grundlegende Satzzeichen, Ziffern, Buchstaben, mathematische Operatoren und Platzhalter beibehalten, während überflüssige Interpunktionszeichen und Sonderzeichen eliminiert werden.

- **Konsolidierung:** Mehrfache Interpunktionszeichen und überflüssige Leerzeichen werden vereinheitlicht, sodass der resultierende Text klare und einheitliche Satzstrukturen aufweist.

Zur Veranschaulichung der Wirkung dieses Prozesses zeigt Tabelle 5.1 einige Beispieltransformationen von Original- zu bereinigten Texteingaben.

Tabelle 5.1: Beispiele für Normalisierung und Rauschfilterung

Originaltext	Nach Normalisierung und Rauschfilterung
Move LEFT 50 centimeters!!! ???	move left 50 centimeters.
3 * 4 plus some stray * asterisks *** and cm*s^-2.	3 * 4 plus some stray * asterisks * and cm * cm*s^-2.
Hello \$@^#\$ End	hello end

Die Ergebnisse dieses Schrittes stellen sicher, dass die nachfolgenden Module, wie die Rechtschreibkorrektur und die numerische Verarbeitung, auf konsistenten und bereinigten Texteingaben arbeiten können.

5.3.2 Rechtschreibkorrektur

In diesem Abschnitt wird die automatische Rechtschreibkorrektur beschrieben, die auf der Bibliothek `pyspellchecker` basiert. Das Modul korrigiert häufig auftretende Tippfehler in alphabetischen Wörtern, während spezifische Einheitstoken (z. B. km, m, m/s, m/s²) unverändert bleiben. Dadurch wird sichergestellt, dass die Bedeutung von Maßeinheiten erhalten bleibt, während gleichzeitig die Qualität der Texteingaben verbessert wird.

Zur Veranschaulichung der Funktionalität wird in Tabelle 5.2 ein Vergleich zwischen Original- und korrigierten Texteingaben dargestellt.

Tabelle 5.2: Beispiele für die Rechtschreibkorrektur

Originaltext	Korrigierter Text
This is an exampel of a sentense with erors.	This is an example of a sentence with errors.
The quik brown fox jumpd over the lazy dog!	The quick brown fox jumped over the lazy dog!
Spelling mistaks are common in humann input.	Spelling mistakes are common in human input.
Move 50 km and then go 10 m/s.	Move 50 km and then go 10 m/s.
Accelerate 9.81 m/s^2 when needed.	Accelerate 9.81 m/s^2 when needed.

5.3.3 Numerische Verarbeitung

Dieser Abschnitt beschreibt den Prozess der numerischen Verarbeitung, der in drei Hauptschritte unterteilt ist. Ziel ist es, ausgeschriebene Zahlen in Ziffern zu konvertieren, physikalische Einheiten zu erkennen und diese in ein standardisiertes Format umzuwandeln. Die drei Schritte werden im Folgenden erläutert:

1. Frühe Normalisierung:

Dieser Schritt ersetzt Konstanten (z. B. "pi" → 3.14), wandelt ausgeschriebene Zahlen in Ziffern um, tokenisiert den Text, führt das Zusammenführen von mehrwortigen Einheitenausdrücken durch und standardisiert ausgeschriebene Einheiten in ihre Kurzformen.

Tabelle 5.3 zeigt ein Beispiel für diesen Verarbeitungsschritt.

2. Unit Inference:

Bei diesem Schritt werden, falls nach einer Zahl kein Einheitenbezeichner erkannt wird, standardmäßig Einheiten basierend auf dem letzten bekannten Befehlswort eingefügt. Zum Beispiel wird aus "turn 30äutomatisch" "turn 30 deg".

Ein Beispiel ist in Tabelle 5.4 dargestellt.

3. Einheitenumrechnung:

In diesem letzten Schritt werden die normierten numerischen Ausdrücke in die finalen Standards konvertiert. Dabei werden Entferungen in Zentimeter (cm),

Geschwindigkeiten in Zentimeter pro Sekunde (cm/s), Beschleunigungen in Zentimeter pro Sekunde-Quadrat (cm/s²) und Winkel in Radian (rad) umgerechnet.

Tabelle 5.5 illustriert diesen Umrechnungsprozess.

Zur Verdeutlichung wird im Folgenden ein finales Beispiel gezeigt, das die Kombination aller drei Verarbeitungsschritte veranschaulicht.

Beispiel 1: Frühe Normalisierung

Tabelle 5.3: Frühe Normalisierung: Umwandlung ausgeschriebener Zahlen und Standardisierung von Einheiten

Originaltext	Nach Früher Normalisierung
Move fifty centimeters forward.	move 50 centimeters forward.
Walk one hundred and twenty meters.	walk 120 meters.

Beispiel 2: Unit Inference

Tabelle 5.4: Unit Inference: Hinzufügen von Standard-Einheiten bei fehlender Angabe

Originaltext	Nach Unit Inference
turn 30	turn 30 deg
move 50	move 50 cm

Beispiel 3: Einheitenumrechnung

Tabelle 5.5: Einheitenumrechnung: Konvertierung in standardisierte Einheiten

Originaltext	Nach Einheitenumrechnung
9.81 m/s ²	981.00 cm/s ²
10 m/s	1000.00 cm/s
2 km/h	55.56 cm/s

Finale Kombination: Gesamte Pipeline

Nach Anwendung aller drei Schritte ergibt sich aus einem komplexen Texteingabebeispiel ein vollständig normierter Text, in dem ausgeschriebene Zahlen, fehlende Einheiten und variable Einheitenumrechnungen zusammengeführt werden.

Tabelle 5.6: Finale Ausgabe nach vollständiger numerischer Verarbeitung

Originaltext	Finale Ausgabe
Move fifty centimeters forward and turn 30. Then, accelerate at 9.81 m/s ² .	move 50 cm forward and turn 30 deg. then, accelerate at 981.00 cm/s ² .

5.4 Struktur des JSON-Kommandos

Die Kommunikation zwischen Sprachmodell und Roboter erfolgt über ein standardisiertes JSON-Format. Dieses Format beschreibt die vom Nutzer beschriebene Aktion in strukturierter Form und ist so konzipiert, dass es maschinenlesbar, flexibel und erweiterbar ist.

Allgemeiner Aufbau Das JSON-Objekt besteht aus zwei Hauptkomponenten:

- `commandLanguage`: Eine Liste definierter Kommandos mit zugehörigen Parametern.
- `errors`: Eine Liste möglicher Fehlermeldungen und deren Bedeutung.

Befehlsstruktur Jeder Eintrag unter `commands` beschreibt einen konkreten Steuerbefehl. Ein Beispiel ist der Befehl `forward`, der den Roboter auffordert, sich vorwärts zu bewegen. Jedes Kommando enthält:

- `name` – Name des Kommandos (z. B. `forward`, `rotate`).
- `description` – Eine kurze Beschreibung der beabsichtigten Bewegung.
- `parameters` – Ein Dictionary mit spezifischen Parametern (z. B. `distance`, `angle`, `acceleration`).

Beispiel Im folgenden Beispiel soll der Roboter 30 cm rückwärts fahren:

```
{
  "command": "back",
  "parameters": {
    "distance": 30,
    "acceleration": 10
  }
}
```

Verfügbare Kommandos Die aktuelle Implementierung unterstützt folgende Steuerbefehle:

- `forward / back` – Lineare Bewegung (mit `distance` und optionaler `acceleration`).
- `left / right` – Seitwärtsbewegung (Strafing mit Mecanum-Rädern).
- `rotate` – Rotation um die eigene Achse mit `angle`, `direction` (`left/right`) und optional `acceleration`.
- `stop` – Sofortiges Anhalten ohne Parameter.

Alle Parameter sind eindeutig beschrieben: Sie besitzen Typangaben (`number`, `string`), Einheiten (`cm`, `rad`, `cm/s2` etc.) und teilweise Standardwerte.

Fehlerstruktur Zur robusten Verarbeitung enthält das Format auch definierte Fehlerarten unter dem Schlüssel `errors`. Diese decken typische Probleme wie fehlende Parameter, ungültige Werte oder Syntaxfehler ab. Beispiele:

- `MISSING_PARAMETER` – Ein Pflichtwert wurde nicht übergeben.
- `UNSUPPORTED_UNIT` – Eine Einheit wurde nicht erkannt.
- `INVALID_COMMAND` – Das Kommando ist nicht bekannt.

Vorteile des Formats Dieses JSON-Schema ist bewusst klar strukturiert, um sowohl von einem LLM generiert als auch vom Roboter verlässlich interpretiert werden zu können. Es bildet die Schnittstelle zwischen sprachlicher Eingabe und maschineller Ausführung und erlaubt eine einfache Erweiterung für zukünftige Funktionen wie z. B. Sensorabfragen oder Rückmeldungen vom Roboter.

5.5 REST-API Back-End

Die REST-API wird mit FastAPI implementiert, einem modernen, schnellen (high-performance) Webframework für Python. FastAPI bietet zahlreiche Vorteile, wie etwa:

- **Hohe Performance:** Durch die asynchrone Programmierung ermöglicht FastAPI eine schnelle und effiziente Verarbeitung von HTTP-Anfragen.
- **Einfache Entwicklung:** Dank der automatischen Generierung von interaktiver API-Dokumentation (z.B. mit Swagger UI) und Pydantic für Datenvielfältigung wird die Entwicklung vereinfacht.

- **Modularität:** Die API-Architektur lässt sich leicht in einzelne, wiederverwendbare Komponenten gliedern, was Wartung und Erweiterung unterstützt.
- **Einfachheit und Klarheit:** FastAPI nutzt Standard-Python-Typannotationen, wodurch der Code übersichtlich und intuitiv zu verstehen ist.

5.6 Robotersimulator

Um die Entwicklung und das Testen der sprachgesteuerten Robotersysteme zu erleichtern, wurde ein 2D-Robotersimulator implementiert. Dieser Simulator dient als kritische Umgebung, um sowohl die Funktionalität des Roboters als auch die Wirksamkeit der Sprachsteuerungsbefehle in einer kontrollierten Umgebung zu validieren. Zukünftig ist geplant, diesen Simulator als Plattform für das Training und die Erprobung autonomer Fahrfähigkeiten des Roboters zu nutzen.

5.6.1 Implementierung des Simulators

Der Robotersimulator wurde komplett in Python mit Pygame entwickelt. Er zeigt den Roboter als rotes Rechteck, das sich durch eine 2D-Umgebung mit grünen Hindernissen bewegt. Eine einfache Physik-Engine ermöglicht realistische lineare und rotatorische Manöver, inklusive Strafing, wie es bei Mecanum-Rädern typisch ist. Bei Kollisionen oder dem Verlassen der Spielfläche endet die Runde, und ein Punktezähler zeichnet die Überlebensdauer auf: je länger der Roboter unbeschadet bleibt, desto mehr Punkte sammelt er.

Die Integration des Simulators in das Gesamtsystem erfolgt über eine REST-API, die mit FastAPI implementiert wurde. Die API liefert bewusst keine vollständigen internen Zustände, sondern emuliert Sensor-Inputs: Endpunkte geben simulierte Sensordaten (z. B. Abstände, Winkel, Geschwindigkeiten) sowie den aktuellen Punktestand zurück. Außerdem lassen sich darüber Steuerbefehle an den Roboter senden und das Spiel zurücksetzen, wodurch externe Anwendungen wie das Webinterface den Simulator programmgesteuert steuern und direktes Feedback erhalten.

5.6.2 Webinterface zur Steuerung

Das Webinterface zur Steuerung des Simulators wurde in React Native implementiert. Dies ist die einzige Projektkomponente, die nicht in Python geschrieben ist, da der Autor zu diesem Zeitpunkt ein mobiles App-Projekt mit React Native abgeschlossen hatte. Die Oberfläche ist chatähnlich gestaltet und ermöglicht es Benutzern, Befehle einzugeben und Echtzeit-Feedback (Score, Sensordaten) von der Simulation zu erhalten.

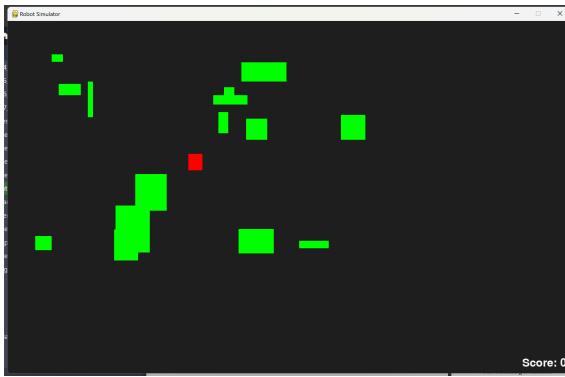


Abbildung 5.3: Pygame-basierte Roboter-simulation.

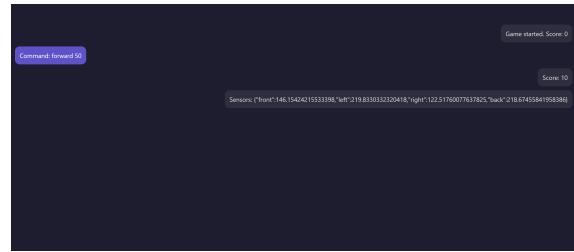


Abbildung 5.4: React Native-basiertes Webinterface.

Abbildung 5.5: Der Robotersimulator und sein Webinterface. Quelle: Eigene Darstellung

5.6.3 Software auf dem Roboter

Die gesamte Firmware und Steuerlogik auf dem Roboter ist in Python realisiert und nutzt die Bibliothek Pymata4 zur Kommunikation mit einem Arduino-Board. Auf diese Weise bleibt das komplette System im gleichen Tech-Stack und lässt sich direkt aus dem API-Server heraus ansteuern.

- **API-Modelle:** Die JSON-Schemas für Einzel- und Batch-Kommandos sowie die Status-Antworten werden mit Pydantic in `api_models.py` definiert :contentReference[oaicite:0]index=0. Das sorgt für automatisches Validation und klare Typen.
- **REST-API:** Mittels FastAPI (`main_api.py`) werden die Endpunkte `/drive` und `/drive_batch` bereitgestellt. Beim Start wird eine globale `RobotInterface`-Instanz aufgebaut, die alle Hardware-Zugriffe übernimmt :contentReference[oaicite:1]index=1.
- **Konfiguration:** In `robot_config.py` sind sämtliche Hardware-Parameter abgelegt – COM-Port, Baud-Rate, Motor-Pins, Limit-Switch-Pins und Umrechnungsfaktoren für Distanzen und Winkel :contentReference[oaicite:2]index=2.
- **RobotInterface:** Dieser zentrale Baustein in `robot_interface.py` kümmert sich um
 - serielle Verbindung und Pin-Setup,
 - nicht-blockierende Zeitsteuerung der Motoren (`drive_timed`),
 - Limit-Switch-Callbacks,
 - Reaktionslogik bei Hinderniserkennung (inkl. Rückwärtsfahrt) :contentReference[oaicite:3]index=3.

Auf diese Weise läuft auf dem Roboter selbst kein eigener HTTP-Server oder komplizierter Microcontroller-Code, sondern eine schlanke Python-Bibliothek, die über die zentrale FastAPI-Instanz auf meinem Rechner per HTTP angesprochen wird. Dieses Setup erlaubt schnelle Updates, gemeinsame Logik mit dem Backend und nahtlose Debug-Möglichkeiten – ganz ohne proprietäre Firmwaresprache.”

Kapitel 6

Vergleich mit bestehenden Sprachmodellen

In diesem Kapitel wird die Leistung des eigens trainierten Modells mit etablierten, bestehenden Sprachmodellen verglichen. Dabei werden insbesondere das kommerzielle Modell Gemini sowie das feingetunte Teacher-Modell (`google/flan-t5-small`) betrachtet. Ziel ist es, Unterschiede in der Leistung, Stärken und Schwächen sowie den Einfluss der zur Verfügung stehenden Ressourcen zu verdeutlichen.

6.1 Gemini – Kommerzielles Modell mit Prompt-Engineering

Im Rahmen der Untersuchung zeigte sich Gemini als besonders leistungsstark. Das Modell lieferte sehr gute Ergebnisse bereits mit minimalem Prompt-Engineering, also ohne aufwändiges Vor- oder Nachverarbeiten der Eingaben und Ausgaben. Diese robuste Leistung lässt sich auf die umfangreichen Trainingsressourcen und die enorme Größe des Modells zurückführen, die Gemini eine umfassende Sprachkompetenz verleihen.

Die Stärke von Gemini liegt insbesondere darin, dass es Zahlenangaben zuverlässig erkennt und kontextgerecht interpretiert. Dadurch eignet sich Gemini ideal für direkte Anwendungen, bei denen präzise und verlässliche Ergebnisse erwartet werden, ohne zusätzliche Verarbeitungsschritte implementieren zu müssen.

6.2 Feinjustiertes Teacher-Modell – Gute Ergebnisse mit leichten Schwächen

Das feingetunte Teacher-Modell (`google/flan-t5-small`) zeigt grundsätzlich solide Leistungen. Es gelingt dem Modell, natürliche Sprache gut zu verstehen

und zu übersetzen. Allerdings zeigen sich gelegentlich Schwierigkeiten, wenn es darum geht, Zahlen korrekt zu identifizieren und präzise zu verarbeiten. Diese Schwäche tritt vor allem dann auf, wenn keine spezielle Vor- oder Nachverarbeitung angewendet wird.

Trotz dieser Einschränkungen lieferte das Teacher-Modell zufriedenstellende Ergebnisse, die durch kleinere Anpassungen in der Eingabeformulierung (Prompt) oder leichte Nachbearbeitungen signifikant verbessert werden konnten. Dies unterstreicht die Bedeutung von ergänzendem Prompt-Engineering, um optimale Resultate zu erzielen.

6.3 Eigenes Modell – Herausforderungen und Grenzen

Das in dieser Arbeit trainierte Modell zeigte hingegen deutliche Einschränkungen. Die Ergebnisse waren insbesondere bei der Verarbeitung numerischer Werte und präzisen Anweisungen noch verbesserungsfähig. Diese Schwächen lassen sich hauptsächlich auf die begrenzten Ressourcen zurückführen, die für das Training zur Verfügung standen. Die Modellgröße und die Trainingsdauer mussten erheblich eingeschränkt werden, da für das Training lediglich ein handelsüblicher Laptop und eine Google-Colab-Umgebung zur Verfügung standen.

Die deutlich limitierte Rechenleistung führte dazu, dass nur eine kleinere Modellvariante mit deutlich weniger Parametern trainiert werden konnte. Ebenso war es nur möglich, das Modell über eine geringe Anzahl von Epochen zu trainieren. Dies hatte zur Folge, dass es sich nicht optimal auf die komplexe Aufgabe der numerischen Verarbeitung und kontextuellen Präzision spezialisieren konnte.

6.4 Verbesserung durch Pre- und Post-Processing

Um die Schwächen des eigens trainierten Modells zu kompensieren, wurden Techniken zur Vor- und Nachbearbeitung implementiert. Durch einfache Regeln und Verarbeitungsschritte (z. B. reguläre Ausdrücke zur Zahlenextraktion oder heuristische Korrekturen) konnten viele der fehlerhaften Ausgaben verbessert werden. Diese gezielten Nachbearbeitungsschritte ermöglichen es, die praktischen Ergebnisse des Modells deutlich aufzuwerten und näher an das Niveau des Teacher-Modells heranzuführen.

Zusätzlich zeigte sich, dass durch besseres Prompt-Engineering, also durch gezielte Anpassungen der Eingabeanweisungen, die Ergebnisse ebenfalls spürbar verbessert werden konnten. Dies bestätigt, dass auch ein Modell mit begrenzten Ressourcen und Parametern brauchbare Resultate liefern kann, wenn der Fokus auf begleitenden

Verarbeitungsschritten liegt.

6.5 Zusammenfassung

Der Vergleich zeigt, dass Gemini klar die beste Leistung liefert und nahezu ohne weitere Anpassungen sofort einsetzbar ist. Das feinjustierte Teacher-Modell erreicht bereits zufriedenstellende Ergebnisse, hat jedoch bei numerischen Details kleinere Schwächen. Das eigene,ressourcenbeschränkte Modell hingegen zeigt deutliche Grenzen, lässt sich aber durch gezielte Vor- und Nachverarbeitung erheblich verbessern. Dieser Vergleich verdeutlicht, wie stark die Leistung von Modellen von den verfügbaren Ressourcen und dem Umfang des Trainings abhängt, unterstreicht jedoch gleichzeitig, dass durch intelligente Verarbeitungsschritte auch schwächere Modelle sinnvoll genutzt werden können.

Kapitel 7

Fazit und Ausblick

Diese Arbeit zeigt, dass **sprachbasierte Robotersteuerung mit Large-Language-Models (LLMs)** heute schon machbar ist – selbst mit sehr begrenzter Hardware. Im Folgenden fasse ich die wichtigsten Punkte zusammen und gebe einen kurzen Ausblick.

Wichtigste Ergebnisse

- **Machbarkeitsnachweis**

Ein kleiner, distillierter *Tiny-T5* wandelt natürliche Befehle zuverlässig in das benötigte JSON-Schema um und bewegt damit den Mecanum-Rover (siehe Kapitel 3 und 5).

- **Ressourcenschonung**

Durch LoRA-Feintuning, 8-Bit-Quantisierung und Distillation läuft das Modell sogar auf einem normalen Laptop oder einer kostenlosen Colab-GPU (Kapitel 4).

- **End-to-End-Prototyp**

Hardware, Webfrontend und REST-Backend greifen sauber ineinander – vom Sprachbefehl bis zum fahrenden Roboter (Kapitel 5).

- **Modellvergleich**

Kapitel 6 zeigt: Ein kommerzielles Modell wie *Gemini* überzeugt schon mit einfacherem Prompt-Engineering, während das Teacher-Modell bei Zahlen schwächelt. Mein eigenes Student-Modell ist spürbar schwächer, lässt sich aber durch Vor- und Nachverarbeitung deutlich aufwerten.

Wesentliche Erkenntnisse

1. Prompt-Engineering statt Komplett-Retraining

Für viele Firmen reicht ein großes Foundation-Modell plus guter Prompt. Das spart Wochen an Datenaufbereitung und GPU-Zeit.

2. Eigene Modelle haben ihre Nische

Wo Daten sehr speziell sind oder Datenschutz wichtig ist, kann ein kleines, fein-tuntes Modell mit geringerer Latenz punkten – trotz kleinerer Parameterzahl.

3. Hardware bleibt der Flaschenhals

Die größten Hürden waren nicht Algorithmen, sondern GPU-Zeit und RAM. Techniken wie Quantisierung und Distillation sind daher Pflicht.

Ausblick

- **Edge-Deployment**

Nächster Schritt: Das Student-Modell auf einen Raspberry Pi oder Jetson Nano bringen – ganz ohne Cloud.

- **Besseres Zahlenverständnis**

Gezielte *numerical augmentation* und *unit normalisation* sollen typische Fehler bei Entfernung und Einheiten verringern.

- **Sensor-Fusion**

Kamera- und Ultraschalldaten könnten künftig direkt in die Sprachausgabe einfließen, etwa um Hindernisse mitzuteilen.

- **Human-in-the-Loop**

Ein leichtes Reinforcement-Learning mit Nutzerfeedback verspricht zusätzliche Robustheit – ganz ohne große neue Datensätze.

Kurz gesagt: *LLMs werden die Robotik verändern.* Ein komplettes Training von Grund auf ist für Einzelpersonen kaum realistisch, doch der clevere Einsatz bestehender Modelle – kombiniert mit gezieltem Fine-Tuning, Prompt-Engineering und kleinen Optimierungstricks – eröffnet schon heute neue Wege für flexible, sprachgesteuerte Robotersysteme.

Literatur

- [Cor25] Coralogix. *Exploring Architectures and Capabilities of Foundational LLMs*. Zugriff am 27. März 2025. 2025. URL: <https://coralogix.com/ai-blog/exploring-architectures-and-capabilities-of-foundational-l1lms/>.
- [Dom12] Pedro Domingos. „A Few Useful Things to Know about Machine Learning“. In: *Communications of the ACM* 55.10 (2012), S. 78–87.
- [Faca] Hugging Face. *LoRA Fine-tuning*. Zugriff am 29. Mai 2025. URL: <https://huggingface.co/docs/diffusers/training/lora>.
- [Facb] Hugging Face. *Quantisierung in Transformers*. Zugriff am 27. März 2025. URL: https://huggingface.co/docs/transformers/en/main_classes/quantization.
- [Facc] Hugging Face. *Tokenizers: Training — Byte-Pair Encoding*. Zugriff am 27. März 2025. URL: <https://huggingface.co/learn/nlp-course/chapter6/5>.
- [For] Roblox Developer Forum. *Understanding Neural Network Backpropagation & Learning Algorithm*. Zugriff am 27. März 2025. URL: <https://devforum.roblox.com/t/understanding-neural-network-backpropagation-learning-algorithm/1838115>.
- [HS+21] Edward J. Hu, Yelong Shen u. a. „LoRA: Low-Rank Adaptation of Large Language Models“. In: *arXiv preprint arXiv:2106.09685* (2021). Zugriff am 27. März 2025. URL: <https://arxiv.org/pdf/2106.09685.pdf>.
- [HVD15] Geoffrey Hinton, Oriol Vinyals und Jeff Dean. „Distilling the Knowledge in a Neural Network“. In: *arXiv preprint arXiv:1503.02531* (2015). Zugriff am 27. März 2025. URL: <https://arxiv.org/pdf/1503.02531.pdf>.
- [Nep] Neptune.ai. *Knowledge Distillation: Everything You Need to Know*. Zugriff am 27. März 2025. URL: <https://neptune.ai/blog/knowledge-distillation>.

- [NK24] Ali Narimani und Steffen Klarmann. „Integration von großen Sprachmodellen für Echtzeit-Fehlerbehebung in industriellen Umgebungen basierend auf Retrieval-Augmented Generation (RAG)“. In: *7th European Industrial Engineering and Operations Management Conference* (2024). Zugriff am 27. März 2025. URL: <https://index.ieomsociety.org/index.cfm/article/view/ID/17488>.
- [Ras21] Sebastian Raschka. *Einführung in Deep Learning*. Zugriff am 27. März 2025. 2021. URL: <https://sebastianraschka.com/blog/2021/dl-course.html#101-introduction-to-deep-learning>.
- [Vas+17] Ashish Vaswani u. a. „Attention Is All You Need“. In: *Advances in Neural Information Processing Systems* 30 (2017). Zugriff am 27. März 2025. URL: <https://arxiv.org/pdf/1706.03762.pdf>.
- [Wen] Hsinhung Weng. *Understanding Byte Pair Encoding*. Zugriff am 27. März 2025. URL: <https://medium.com/@hsinhungw/understanding-byte-pair-encoding-fd196ebfe93f>.