KI-Ansätze bei der Entwicklung lernender Roboter – Implementierungsmöglichkeiten und Grenzen

Maja Temerinac-Ott

Hochschule Furtwangen
Robert-Gerwig-Platz 1, 78120 Furtwangen
E-Mail: maja.temerinac-ott@hs-furtwangen.de

1 Einführung

Roboter-Programme müssen für jede neue Umgebung und Aufgabe angepasst werden. Dabei müssen neue Positionen des Roboters zeitaufwendig erfasst werden bzw. neue mathematische Formeln für die Bewegungen der Roboter berechnet werden. Dabei stellt sich offensichtlich die Frage: Wie kann man Roboter einfacher programmieren?

Die Robotik gehört von Anfang an zu den aktiven Forschungsgebieten in der KI. Befeuert durch Literatur und Film entstand der Eindruck, dass Roboter immer mehr unseren Alltag bestimmen und bald schon die Menschheit beherrschen würden [1]. Tatsächlich sind wir noch weit weg davon, dass Roboter wirklich intelligent sind und in unserem Alltag die dominierende Rolle spielen. In der Industrie gibt es Roboter, die erfolgreich in der Produktion eingesetzt werden [2] , und im Haushalt haben sich Staubsauger-Roboter und Rasenmäher-Roboter etabliert. Im Bereich des autonomen Fahrens gibt es große Fortschritte [3] und auch in der Energieerzeugung, Gesundheit [4] und Ernährung [5] spielen Roboter eine immer wichtigere Rolle.

Das maschinelle Lernen hat das intelligente Verarbeiten von Sensordaten stark vereinfacht. Dabei haben sich als vorherrschende Methode künstliche Neuronale Netze (kNN) etabliert, da sie es ermöglichen einen Roboter zu program-

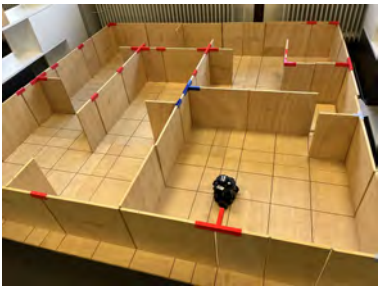
mieren, ohne dabei explizit eine Entscheidungslogik zu definieren. Ein kNN kann nur mit Hilfe von Datenbeispielen trainiert werden und imitiert somit auch das menschliche Lernen. Obwohl kNNs unglaublich gute Ergebnisse erzeugen können, scheitern sie häufig, sobald man die erlernten Algorithmen in einer leicht abgeänderten Umgebung ausprobieren möchte. Die große Herausforderung bei den kNNs ist die gelernten Modelle erklärbar zu machen und die Sicherheit der Algorithmen zu garantieren.

In diesem Beitrag werden wir auf die Programmierumgebungen (ROS [6]) und die Schnittstellen (keras/Tensorflow [7]) eingehen, die es ermöglichen Roboter mit Hilfe von maschinellem Lernen zu trainieren. Dabei werden wir insbesondere die Möglichkeiten vorstellen, wie man einen Roboter in der Simulation (gazebo) trainieren kann, um die trainierten Modelle auf echte Roboter zu übertragen. Das Testszenario besteht aus einem Holz-Labyrinth und einem Turtlebot Roboter, der mit Laser Range Scanner und einer 2D-Kamera ausgestattet ist. Dabei soll der Roboter lernen, autonom den Weg zur angegebenen Zielposition zu planen ohne dabei gegen ein Hindernis zu fahren. Es wird hierbei untersucht in wie weit die trainierten Modelle in leicht abgeänderten Szenarien funktionsfähig bleiben.

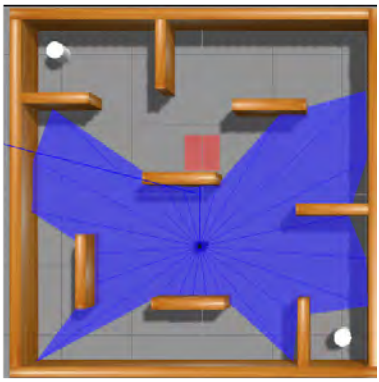
2 Maschinelles Lernen in der Simulation

2.1 Verwandte Arbeiten

Die Kombination von Deep Learning und Simulation ist vielversprechend, da sich die beiden Technologien ergänzen: Die Simulation kann sehr viele Daten erzeugen und die kNNs benötigen viele Beispiele, um sinnvolle Modelle zu lernen. In dem kNNs mit simulierten Daten trainiert werden, lernen sie komplexe Aufgaben bei der Steuerung von Robotern zu lösen [8]. Dabei kann man ohne den Roboter zu beschädigen in der Simulation viele unterschiedliche Szenarien und Parameter erforschen. Zudem bietet die Simulation die Möglichkeit mit dem Roboter zu arbeiten ohne physisch in der Anlage oder im Labor präsent zu sein. Allerdings stellt es noch immer eine große Herausforderung



(a) Turtlebot3 im Holzlabyrinth.



(b) Darstellung von Roboter und Simulationsumgebung in gazebo (Stage 4).

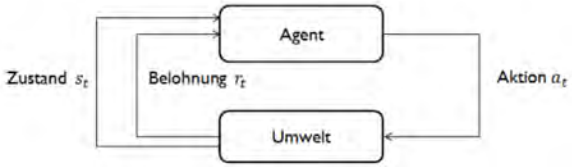
Bild 1: RL- Testumgebung für Sim2Real.

dar, die trainierten Modelle erfolgreich bei der Steuerung von echten Robotern einzusetzen [9].

2.2 ROS und Gazebo

Die Open Source Plattform ROS [6] ermöglicht es Programme für die Steuerung von Robotern zu schreiben, die sowohl auf echten Robotern als auch auf simulierten Robotern eingesetzt werden können, ohne dabei viele Änderungen im Programm durchführen zu müssen. Die Plattform basiert auf einer Master-Slave Architektur, die auf dem Publisher-Subscriber Prinzip beruht. Die Software wird in Modulen (Nodes) erstellt, die zu einem Graphen verbunden werden und über Messages miteinander kommunizieren können. Diese Messages müssen in eine Topic publiziert oder von einer Topic ausgelesen werden. ROS unterstützt verschiedenen Programmiersprachen, wobei für ML häufig die Programmiersprache Python zum Einsatz kommt.

Gazebo [10] ist eine Open Source Simulationsplattform, die leicht innerhalb von ROS installiert werden kann und seit 2004 entwickelt wird. Mittlerweile haben auch große Firmen das Potential von Simulationsplattformen erkannt und investieren in eigene Simulationsumgebungen für die Robotik i.e. [11].



(a) Modellierung der Parameter eines RL-Algorithmus.

Bild 2: RL- Kreislauf zum Erlernen von Aktionen, welche die Belohnung über die Zeit optimieren.

Gazebo stellt fertige Simulationsumgebungen zur Verfügung (z.B. in Bild 1b), erlaubt es aber auch eigene neue Simulationsumgebungen zu kreieren.

2.3 Reinforcement Learning Umgebung

Eine RL Umgebung bietet die Möglichkeit Aktionen zu erlernen, die zu einem gewünschten Ziel führen. Dabei wird die Umgebung mit Hilfe von Zuständen (s_t) modelliert, die sich über die Zeit verändern können, in dem man Aktionen (a_t) ausführt und dabei Belohnungen r_t erhält [12]. Entscheidend für die Maximierung der Belohnung in Bild 10 ist die Wahl des Agenten und die Modellierung der Umgebung. Der DQN-Agent [13] hat bei 49 Atari Computerspielen erstaunlich gute Ergebnisse erzielt und wird deswegen in dieser Arbeit eingesetzt werden. Bei der Simulation von Robotern beschränkt sich die Modellierung der Zustände auf die Modellierung der Sensordaten, wohingegen die Aktionen durch die Steuerungsbefehle, die an den Roboter gesendet werden, dargestellt werden. In Bild 1 ist der Roboter in einem Labyrinth dargestellt. Der Roboter soll mit Hilfe des RL-Algorithmus erlernen zu dem Ziel (rotes Rechteck in Bild 1b) zu gelangen ohne dabei gegen die Wand oder ein anderes Objekt zu fahren.

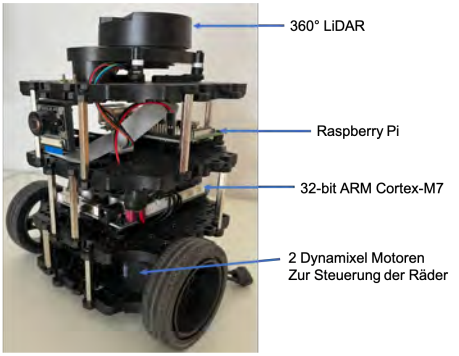
3 Experimentaufbau

3.1 Turtlebot

Der erste Turtlebot wurde 1960 am MIT entwickelt und war der Namensgeber für eine Reihe von Robotern, die mit ROS programmiert werden können und die in der Lehre und Forschung eingesetzt werden. Wir verwenden für die Experimente Turtlebot3 burger der Firma ROBOTIS [14]. In Bild 4 sieht man den Roboter, so wie die Beschriftung der wichtigsten Komponenten des Roboters. Durch den LiDar Sensor werden Distanzmessungen erzeugt und durch die Topic '/scan' innerhalb von ROS bereitgestellt und für die Modellierung des Zustands s_t verwendet. In dem man Werte in die Topic '/cmd_vel' publiziert, können die Räder angesteuert werden. Die Topic '/cmd_vel' erhält Nachrichten vom Typ *Twist*, die aus einer linearen und einer winkelbasierten Geschwindigkeit bestehen. In den Experimenten wird die lineare Geschwindigkeit konstant gehalten ($0.15m/s$) bis das Ziel erreicht wird, während die Winkelgeschwindigkeit von dem Agenten verändert werden kann. Dabei sind folgende 5 Aktionen für a_t möglich: (1) -1.5 rad/sec, (2) -0.75 rad/sec, (3) 0 rad/sec, (4) 0.75 rad/sec und (5) 1.5 rad/sec. Durch das Verändern der Winkelgeschwindigkeit ändert der Roboter seine Bewegungsrichtung.

3.2 Installation und Modellierung des Versuchsaufbaus

Sowohl auf dem Turtlebot3 (Raspberry Pi/ ARM-Controller) als auch auf dem Steuerungs PC wurden die ROS-Pakete zu der Version ROS-kinetic installiert. Zusätzlich wurde auf dem Steuerungs-PC Tensorflow/keras installiert. Zum Trainieren des DQN-Agenten wurde die Anleitung in [14] befolgt und der DQN-Agent für Stage 1 und Stage 2 (statische Objekte) trainiert. Zu beachten ist, dass der Roboter hierbei keine Karte seiner Umgebung erstellt, sondern lediglich die Sensoreingaben benutzt, um zum Ziel zu steuern ohne dabei gegen ein Hindernis zu fahren. Die Belohnung beträgt $r_t = 200$ für das Erreichen des Ziels und $r_t = -200$ für die Kollision mit einem Hindernis. Für das kollisionsfreie Fahren wird eine Belohnung in Abhängigkeit von der Lage des Roboters zum Ziel definiert: Fährt der Roboter in Richtung des Ziels ist r_t



(a) Hardware: LiDAR kann Distanzen im 360° Winkel messen, wobei die Anzahl der Messungenv(z. B. 24) spezifiziert werden muss.

(b) Modelierung des Zustandsraums

Bild 3: Turtlebot3 burger

positiv; entfernt er sich vom Ziel wird r_t negativ. Das DQN wird trainiert, in dem der aktuelle Zustand s_t, die ausgeführte Aktion a_t und der neue Zustand s_{t+1} so wie die erhaltene Belohnung r_t an das kNN übermittelt werden. Dabei soll das kNN lernen die beste Aktion a_t für jeden Zustand s_t auszuführen. Um die beste Aktion ermitteln zu können, muss der Q-Wert $Q(s,a)$ gelernt werden, in dem sehr viele Aktionen in der Simulation ausgeführt werden.

3.3 Training des kNN

Die Architektur des verwendeten kNN ist in Fig. 4 abgebildet. Beim Deep Q-Learning werden mehrere Episoden gespielt. Jede Episode wird benutzt, um die Gewichte des Neuronalen Netzwerks anzupassen. Je mehr Episoden gespielt werden, desto besser sollte die Reward Funktion für jede Episode werden. Am Anfang sollten möglichst viele Aktionen zufällig ausgewählt werden. Mit steigender Zahl der Episoden sinkt die Anzahl der zufälligen Aktionen. Damit ältere Episoden nicht komplett vergessen werden, werden diese teilweise in der replay memory gespeichert. Diese stellt sicher, dass es keine zu abrupten Wechsel in den berechneten Parametern des Neuronalen Netzwerks

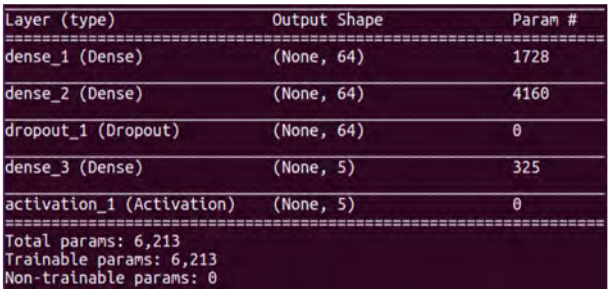


Bild 4: Tensorflow-Ausgabe der kNN Archtiektur

gibt. Die Parameter des DQN sind in Tab. 1 abgebildet. Das DQN beseht aus zwei identischen kNNs, wobei die Gewichte des einen Netzwerks nach einer bestimmten Anzahl an Episoden von Source zu Target Netzwerk kopiert werden. Das Target Netzwerk trifft die Entscheidungen, während die Gewichte des Source Netzwerks mit jeder Episode neu berechnet werden. Damit das Netzwerk lernen kann, wird immer eine bestimmt Anzahl an Aktionen zufällig ausgewählt, die am Anfang noch sehr groß ist und mit der Dauer des Trainings immer kleiner wird. Bei der Aktualisierung der Gewichte im Netzwerk, wird der Einfluß der zukünftigen Aktionen durch den Parameter Discount berücksichtigt. Ein Discount nahe eins bezieht immer auch das Ende der Episode in das laufende Update der Gewichte mit ein.

4 Ergebnisse

4.1 Ergebnisse der Simulation

Die Ergebnisse zeigen, dass komplexere Umgebungen längere Trainingszeiten verlangen. In der Simulation in Bild 5 zeigen, das der Roboter nach 100 Episoden in Stage 1 (einfache Umgebung) und 1000 Episoden in Stage 2 (komplexe Umgebung) gelernt hat möglichst viele Ziele innerhalb einer Episode zu besuchen und die erhaltene Belohnungen zu maximieren. Stage 1 und Stage 2 sind Testumgebungen, welche in dem ROS-Paket 'turtlebot3_gazebo' bereits

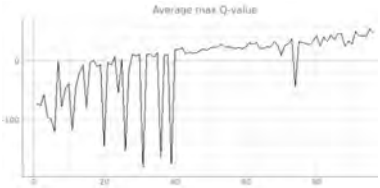
Hyperparameter	Wert
Zeitschritte pro Episode	6000
Update Rate für das Target Netzwerk	2000
Discount	0.99
Lernrate	0.00025
Startwert für die Auswahl einer Zufallsaktion	1.0
Abfallrate für die Auswahl einer Zufallsaktion	0.99
Minimaler Wert für Zufallsaktionen	0.05
Batchgröße	64
Größe der Replay Memory	1 Million (s_t, a_t)-Paare
Minimale Größe in der Replay Memory	64

Tabelle 1: Hyperparameter für das Training des DQN-Agenten.

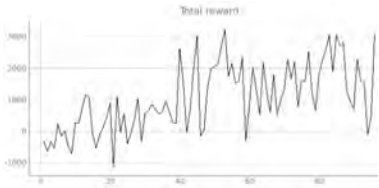
enthalten sind. Bei Stage 1 kann der Roboter das Ziel direkt anfahren, während er bei Stage 2 lernen musste, Hindernisse zu umfahren, um zum Ziel zu gelangen, was zu einer längeren Trainingszeit führt. Bei Stage 2 wird die Kollision weniger bestraft als bei Stage 1 ($r_t = -150$). Zudem werden bei der Modellierung von s_t bei Stage 2 zwei zusätzliche Parameter betrachtet: die minimale vom Sensor gemessene Distanz und der Winkel, aus dem diese Distanz gemessen wurde.

4.2 Ergebnisse im Holzlabyrinth

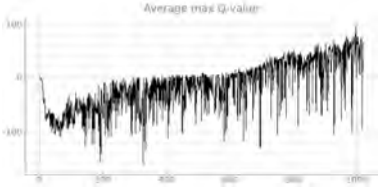
Um das trainierte Modell auf dem realen Turtlebot ausführen zu können, müssen kleinere Anpassungen getroffen werden: Die Sensordaten müssen abgetastet werden, um aus 360 Distanzmessungen des Sensors 24 Messungen für die Modellierung von s_t zu erhalten. Zudem gibtder simulierte Sensor in Gazebo den Rückgabewert *Inf* zurück, wenn der Roboter kein Hindernis detektiert, wohingegen bei dem realen Roboter der Rückgabewert Null ist. Die Angabe der Zielposition erfolgt relativ zu der Ausgangslage des Roboters. Die lineare Geschwindigkeit und die Winkelgeschwindigkeit bei dem realen Roboter entspricht der Geschwindigkeit in der Simulation.



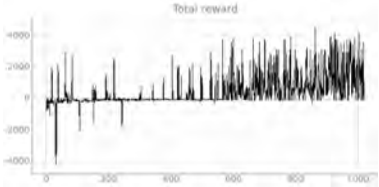
(a) Stage 1 Q-Werte während 100 Episoden des DQN-Algorithmus.



(b) Stage 1 Rewards während 100 Episoden des DQN-Algorithmus



(c) Stage 2 Q-Werte während 1000 Episoden des DQN-Algorithmus.



(d) Stage 2 Rewards während 1000 Episoden des DQN-Algorithmus

Bild 5: Training des DQN-Algorithmus

Die Simulationsumgebungen Stage 1 und Stage 2, so wie das vereinfachte Holz-Labyrinth sind in Bild 6 zu sehen. Wenn das trainierte DQN-Modell (Stage 1 oder Stage 2) auf diese neue Umgebung angewendet wird, so gelingt es dem Roboter ohne Probleme Ziele in einem Umkreis des Roboters zu erreichen, die sich auf freier Bahn befinden. Die Trefferquote entspricht der Trefferquote in der Simulation. Interessanterweise können auch Ziele erreicht werden, deren Distanz größer ist als die maximale Distanz während dem Training in der Simulation.

Soll ein Ziel wie etwa der grüne Würfel in Bild 6a erreicht werden, so bleibt der Roboter an der Abtrennung, die mit dem weißen Pfeil markiert ist hängen. Wird diese entfernt, so kann der Roboter das Ziel erreichen, allerdings nur mit dem Stage 2 DQN-Modell. Der Roboter hat dann jedoch das Problem zu wenden, wenn er wieder zurück an die Ausgangsposition fahren möchte. Hier reicht das trainierte DQN-Modell nicht aus, um alle Ziele zuverlässig zu erreichen.

(a) Einfaches Holz-Labyrinth.

(b) Stage 1

(c) Stage 2

Bild 6: Reale Testumgebung (links) für Stage 1 und Stage 2 (rechts).

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, dass es möglich ist RL-Agenten in der Simulation zu trainieren und die trainierten Modelle auf echte Roboter zu übertragen. Sind sich Simulation und die reale Szene sehr ähnlich kann das Modell direkt übernommen werden. Um das Modell auf neue Szenarien anzupassen, könnte Active Learning hilfreich sein. Dabei sollen insbesondere Episoden zum Trainieren benutzt werden, bei denen der Mensch dem Agenten Aktionen vorgibt, um zum Ziel zu gelangen.

Weiterhin ist es interessant zu erforschen in wie weit man die trainierten Modelle von einem Roboter auf den anderen Roboter mittels Transferlearning übertragen kann. Bei dem turtlebot3 gibt es ein weiteres Modell (waffel) mit einem Greifarm, das andere Dimensionen als der turtlebot3 burger hat. Anstatt den Lernvorgang von vorne zu starten, wäre es viel effizienter die trainierten Modelle an die neuen Dimensionen anzupassen.

Schließlich ist eines der größten Risiken der kNN, das wir schlecht erklären können, was das Modell nun gelernt hat und wie zuverlässig das Modell sein

wird. Eine interessante Forschungsrichtung wäre es Methoden zu entwickeln, die das trainierte Modell erklärbar machen.

Literatur

[1] S. Russel. „Human Compatible: AI and the Problem of Control“. Viking. 2019.

[2] S. Robla-Gómez et al. „Working Together: A Review on Safe Human-Robot Collaboration in Industrial Environments“. In: *IEEE Access* 5. 2017.

[3] Y. Almalioglu, M. Turan, N. Trigoni und A. Markham„Deep learning-based robust positioning for all-weather autonomous driving“. In: *Nature Machine Intelligence* 2022.

[4] M. Kyrarini, et al. „A Survey of Robots in Healthcare“. In: *Technologies* 9 (1), 8 2021.

[5] C Rizzardo, S Katyara, M Fernandes and F Chen. „The importance and the limitations of sim2real for robotic manipulation in precision agriculture“. arXiv preprint arXiv:2008.03983. 2020.

[6] M. Quigley et al. „ROS: an open-source Robot Operating System“. In: *ICRA workshop on open source software* 3 (3.2), 5 2009.

[7] M. Abadi et al. „TensorFlow: Large-scale machine learning on heterogeneous systems“, arXiv:1603.04467 [cs.DC], 2016. Software available from tensorflow.org.

[8] J. Jesus, J. Bottega, M. Cuadros and. D. Gamarra. „Deep Deterministic Policy Gradient for Navigation of Mobile Robots in Simulated Environments“. In: *Proc., 19th International Conference on Advanced Robotics (ICAR)* 2019.

[9] S. Höfer et al. „Sim2Real in Robotics and Automation: Applications and Challenges“. In: *IEEE Transactions on Automation Science and Engineering* 18 (2). 2021.

[10] N. Koenig und A. Howard. „Design and use paradigms for Gazebo, an open-source multi-robot simulator“. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* 2004.

[11] „NVIDIA Isaac Sim“. 2022. [Online]. Available: https://developer.nvidia.com/isaac-sim (accessed Sept. 14, 2022).

[12] S. Russell und P. Norvig. „Artificial intelligence: a modern approach“. 2002.

[13] V. Mnih et al. „Human-level control through deep reinforcement learning“. In: *Nature* 518. 2015.

[14] „ROBOTIS eManual“. [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. [Accessed: Sept. 14, 2022].