DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Erlernen von Hindernisumfahrung mithilfe von Reinforcement Learning

Studienarbeit (T3_3101)

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Yannik Schiebelhut

Abgabedatum:	22. Mai 2023
Bearbeitungszeitraum:	14.10.2022 - 22.05.2023
Matrikelnummer, Kurs:	3354235, TINF20B1
Gutachter der Dualen Hochschule:	Florian Stöckl

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit (T3_3101) mit dem Thema:

Erlernen von Hindernisumfahrung mithilfe von Reinforcement Learning

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

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

Karlsruhe, den 22. April 2023

Schiebelhut, Yannik

Abstract

- Deutsch -

Platzhalter

Abstract

- English -

Placeholder

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

2 Grundlagen

2.0.1 Machine Learning

Besonderer Fokus auf Reinforcement Learning und Begriffsdefinitionen

2.0.2 Unity

kurze Einführung zur Unity-Engine allgemein ML-Agents, Historie und Funktionsweise

2.0.3 Vektorgeometrie

Abstandsbestimmung und Winkel zwischen Vektoren

3 State of the Art

4 Konzeptionierung

4.1 Beschreibung der Projekt-Basis

Die Zielsetzung dieser Arbeit soll aufbauend auf einer Vorgängerarbeit realisiert werden, welche vor einigen Jahren ebenfalls im Rahmen einer Studienarbeit durchgeführt wurde. Hier soll nun zunächst die Vorgehensweise der Vorgängerarbeit in ihren Grundzügen erläutert werden.

4.1.1 Aufbau und Simulation des Roboters

Kernelement der Arbeit ist ein vierbeiniger, 3D-gedruckter Roboter, der in seiner Anatomie einer Spinne gleicht. Der Roboter besteht aus einer rechteckigen Zentralplatte. An jeder Ecke dieser Platte ist ein Bein angebracht. Die Beine des Roboters bestehen jeweils aus 3 separaten Teilen. Folglich hat jedes Bein zwei Gelenke. [1, S. 52] Jedes dieser Gelenke wird mit einem Servomotor des Typs SG90 XY realisiert, welche sich in einem Aktionsradius von 180° bewegen können. Die Neutralstellung der Beine ist die jeweilige Mittelposition des Servomotors. Angegeben wird der Aktionsradius des Servos als Wert im Bereich 0° - 180°, die Mittelstellung befindet sich also bei 90°. [1, S. 38] Die verwendeten Servomotoren bewegen sich nur auf einer Achse. An der Stelle, an der die Beine mit dem Körper verbunden sind, befindet sich ein weiterer Servomotor. Mithilfe der verbauten Servomotoren ist es möglich, jedes Bein anzuziehen oder auszustrecken und es nach vorne beziehungsweise hinten zu bewegen.

Angesteuert werden die Servomotoren mit einem ESP8266 Mikrocontroller, welcher mittels I^2C Steuersignale an ein Servo-Breakout-Board sendet, an welchem wiederum die Servomotoren angeschlossen sind. Die Stromversorgung erfolgt über eine Batterie, damit sich der Roboter autark von einer Stromquelle im Raum bewegen kann. [1, S. 54] Die aufgelisteten Bauteile werden so mit Kabelbindern an der Zentralplatte befestigt, dass sie nicht mit der Bewegungsfreiheit der Beine interferieren. Am Roboter ist keinerlei Sensorik verbaut. [1, S. 36]

Da das Training des Roboters in der Realität zu lange dauern würde und dabei außerdem die Gefahr drohen würde, den Roboter zu beschädigen, wurde eine Simulationsumgebung aufgebaut, in der das Training des Roboters erfolgt. Diese ist in Unity realisiert. In Unity ist es möglich, dasselbe 3D-Modell zu importieren, das auch für den Druck der Teile des Roboters verwendet wird, was eine akkurate Umsetzung der Dimensionen sicherstellt. Die größte Schwierigkeit einer Simulation des Roboters besteht darin, die Servomotoren abzubilden. Im Gegensatz zu einem Computerprogramm ist es Bauteilen in der realen Welt nicht möglich, instantan einen gezielten Zustand anzunehmen. Das heißt, wird für einen Servomotor ein neuer Winkel vorgegeben, so braucht dieser eine bestimmte Zeit, bis er diesen erreicht hat. Diese Zeit ist einerseits von der normalen Bewegungsgeschwindigkeit des Servomotors abhängig, andererseits stellt die Last, die der Motor bewegt, einen weiteren Einflussfaktor dar. Die Vorgängerarbeit implementiert eine Software-Simulation der Servomotoren, bei denen zumindest der Aspekt der normalen Bewegungsgeschwindigkeit berücksichtigt wird, sowie verwandte, allgemeine Charakteristika der Bewegung. [1, S. 37] Zusätzlich werden in Unity die ungefähren Gewichte der einzelnen Teile eingetragen, um eine akkurate Simulation der physikalischen Kräfte zu ermöglichen, die auf den Roboter einwirken. Die Simulation dieser allgemeinen Physik wird bereits als Grundfunktion der Unity-Engine bereitgestellt und muss nicht separat implementiert werden.

irgendwo mehr zu Unity schreiben

4.1.2 Training des Roboters

Mit der fertig aufgebauten Simulation kann nun der Roboter für alle erdenklichen Szenarien trainiert werden. Dazu müssen Rahmenbedingungen des Trainings definiert werden. Primär sind dies die Beobachtungen, die der Trainingsalgorithmus macht, die Aktionen, die er ausführen kann und die Belohnung, die er als Feedback für seine Handlungen erhält. Die Zielsetzung der Vorgängerarbeit bestand aus einer reinen Vorwärtsbewegung. Da der reale Roboter über keine Sensorik verfügt und lediglich die Ansteuerwinkel seiner Servomotoren kennt, waren dies auch die einzigen Beobachtungen, die dem Algorithmus zugänglich gemacht wurden. Als Aktionen kann der Algorithmus beliebige Zielwinkel für die Servomotoren setzen. Die Belohnung für den Roboter bestand in der Vorgängerarbeit aus der Streckendifferenz, die er nach vorne zurückgelegt hat. Bestraft wurde der Roboter für ein Umkippen, um zu verhindern, dass bei der späteren Übertragung auf den realen Roboter die Steuerelektronik beschädigt wird. Um das Training zu beschleunigen, sind in

der Vorgängerarbeit mehrere Agenten nebeneinander in derselben Umgebung instanziiert. Nach einer Einstellung der Hyperparameter wurden Modelle mit den Reinforcement-Learning-Algorithmen SAC und PPO trainiert. Dabei wurde festgestellt, dass in diesem Anwendungsfall die Ergebnisse des PPO-Algorithmus denen des SAC-Algorithmus deutlich überlegen sind. [1, S. 48]

Die Bewegungsart, die der Roboter sich bei diesen Trainingsbedingungen antrainiert, ist keine klassische Form des natürlichen Laufens. Stattdessen führen die Ergebnisse des Trainings dazu, dass der Algorithmus den Roboter mit einem der Hinterbeine einknicken lässt und ihn dann sprungartig nach vorne katapultiert. [1, S. 51]

4.1.3 Übertragung in die Realität

Die Implementierung der Vorgängerarbeit sieht auch eine Übertragung der Ergebnisse auf den realen Roboter vor. Zu diesem Zwecke können die Steuersignale, die der Roboter in der Simulation erhält, über eine serielle Verbindung übertragen werden und direkt auf dem realen Roboter angewandt werden.

Die unternommenen Versuche waren jedoch leider nicht erfolgreich. Als Ursache fällt die Vermutung auf Hardwarefehler, da der Roboter – in der Luft gehalten – die Bewegungen korrekt spiegelt. Wird der Roboter jedoch auf den Boden gestellt, knickt er unter seinem Gewicht sofort ein und kann die gelernte Laufmethodik nicht anwenden. [1, S. 58]

4.2 Einschränkungen und Übertragungsprobleme

Bislang sollte der Roboter nur geradeaus laufen. Dabei verfügt der verwendete Roboter über keinerlei Sensorik. Einzig die aktuellen Zielwinkel der Servomotoren, welche die Bewegung der Beine ermöglichen, sind dem Lernalgorithmus zugänglich. Das Training des Roboters erfolgte jedoch rein in der Simulation. Da die Simulationsumgebung nicht nur den Roboter, sondern dessen komplettes Umfeld abbilden muss, sind alle Informationen, die ein beliebiger Sensor liefern könnte theoretisch in der Simulation vorhanden, wurden dem Algorithmus jedoch nicht zugänglich gemacht. Der Roboter kennt seinen eigenen Zustand nicht, beziehungsweise nur bedingt. Das trainierte Modell wird lediglich in der Praxis angewandt, unter der Annahme, dass bei einem korrekt gelernten Modell keinerlei

Zusatzinformationen notwendig sind, um den Roboter seine Aufgabe erfüllen zu lassen: geradeaus zu laufen.

Die Zielsetzung dieser Studienarbeit erweitert nun jedoch diese Aufgabe des Roboters, was Probleme aufwirft. Der Roboter soll lernen, einem vorgegebenen Pfad zu folgen. Dabei soll der Pfad für jeden Durchlauf dem Roboter individuell vorgegeben werden können. Besonders wichtig ist hierbei zu beachten, dass der Roboter unter keinen Umständen einen bestimmten Pfad auswendig lernen soll, denn dann müsste für jeden Pfad, den der Roboter laufen soll, ein eigenständiges Modell trainiert werden, was einen praktischen Nutzen unmöglich machen würde – schließlich kann nicht für eine Bewegungsanweisung an einen Roboter jedes Mal mehrere Stunden Rechenzeit aufgebracht werden. Dem Roboter muss also ein Pfad mitgegeben werden. Außerdem muss für das Training des Roboters mehrfach dieser übergebene Pfad neu generiert werden, um ein Auswendiglernen zu verhindern.

Dass der Roboter einem frei angegebenen Pfad folgen können soll, wirft die Frage auf, ob er dazu Informationen über seine Position im Raum benötigt. Rein theoretisch betrachtet könnte diese Frage einfach mit „Nein“ beantwortet werden. Prinzipiell gesehen kann der Roboter seine aktuelle Position anhand seiner vergangenen Bewegungen vom Ausgangspunkt her berechnen. Andererseits würde dies eine enorm hohe Präzision der Bewegungen voraussetzen. Außerdem könnte der Roboter in der Realität auf dem Boden rutschen. Auch läuft das bisherige Modell – nicht einmal in der Simulation – verlässlich geradeaus, sondern hat dabei immer einen leichten Drall zur Seite. Aus diesen Gründen wird der Schluss gezogen, dass durch minimale, nicht vermeidbare Abweichungen die Ausführung der Aufgabe nur sehr ungenau möglich wäre, wenn keine Positionierungsinformationen zur Verfügung gestellt werden.

In der Simulation gestaltet sich eine mögliche Lösung des Problems sehr einfach: Der Roboter ist ein Game-Objekt innerhalt der Unity-Engine. Als solches besitzt er automatisch Koordinaten innerhalb der Simulation, welche einfach für den Roboter freigegeben werden können. Bei einer späteren Übertragung in die Realität können diese Informationen durch andere, konkrete Ortungssysteme geliefert werden. Es wäre lediglich ein Mapping erforderlich, um das Informationsformat eines konkreten Sensors in das Koordinatenformat von Unity umzuwandeln. Diese Informationen können dann dem Modell für die Inferenz zur Verfügung gestellt werden. Somit ist es möglich, ein Modell zu trainieren, welches unabhängig von der später eingesetzten Ortungstechnologie ist.

Weiterhin umfasst die Aufgabenstellung, dass der Roboter Hindernisse auf seinem Weg erkennen und gezielt umgehen können soll. Anschließend soll er auf den vorgegebenen Pfad zurückkehren, wobei die Abweichung möglichst gering ausfallen sollte. Hierfür wird eine Hindernis- beziehungsweise Kollisionserkennung benötigt. Mit der aktuellen technischen Ausstattung des Roboters ist auch diese Aufgabe nicht umsetzbar. In der Simulation soll vereinfacht für die Hinderniserkennung die Kollisionserkennung der Beine verwendet werden. (Diese Kollisionserkennung kann Unity für sämtliche Game-Objekte durchführen.) Dadurch muss der Roboter mit seinem Hindernis zusammenstoßen, um es wahrzunehmen. In der Realität wäre natürlich eine Hinderniserkennung sinnvoll, mit der Hindernisse bereits vor einer Kollision erkannt werden können (zum Beispiel LiDAR, Kameras oder ähnliche Systeme).

Sowohl für die Ortung, als auch für die Hinderniserkennung wäre in der Realität der Einsatz komplexer Systeme nötigt. Die Integration solcher würde jedoch den Umfang dieser Arbeit erheblich übersteigen. Hier soll eher ein Proof-of-Concept für die selbstständige Umsteuerung von Hindernissen erarbeitet werden. Es soll keine Übertragung der in der Simulation trainierten Modelle auf den realen Roboter stattfinden.

Die Springbewegung, die der Roboter sich bislang antrainiert hat, führt zu weiteren potenziellen Problemen. Zwar wurde, wie oben beschrieben, der Einsatz eines Ortungssystems festgelegt, jedoch bringt diese Bewegungsform trotzdem massive Genauigkeitsprobleme mit sich. Sie ist sehr kraftaufwändig und instabil, der Roboter schwankt dabei stark um seine horizontale Achse und kann seine exakte Landeposition nur bedingt steuern. Da der Roboter Hindernisse über eine Kollision mit diesen erkennen soll, würde außerdem das Problem bestehen, dass der Roboter im Sprung gegen diese knallen könnte. Aus den genannten Gründen ist es sinnvoll, Maßnahmen zur Einschränkung der Bewegung des Roboters vorzunehmen. Möglich wäre zum Beispiel eine Anpassung der Belohnungsfunktion, wonach starke Höhenänderungen oder Neigungen der Zentralplatte des Roboters bestraft werden.

4.3 Wahl der Simulationsumgebung

In der Vorgängerarbeit wurden bereits drei mögliche Simulationsumgebungen ausführlich gegenübergestellt. [1, S. 27] Die beschriebenen Bedingungen haben sich dabei leicht

verändert. Aus dem Vergleich ging hervor, dass MuJoCo aufgrund seiner äußert realistischen Simulation der Physik und einem Fokus auf Gelenksimulation eine gute Wahl für eine Trainingsumgebung wäre. Jedoch war der Einsatz von MuJoCo damals mit erheblichen Lizenzgebühren verbunden, was einer der Gründe war, diese Software nicht zu verwenden. Mittlerweile ist MuJoCo allerdings frei und quelloffen verfügbar [2, 3], was dafür sprechen würde, die Programmwahl neu zu überdenken.

Andererseits soll diese Arbeit an die vorangegangene anknüpfen. Wenn die Simulationsumgebung gewechselt würde, müsste im Grunde genommen von vorne begonnen werden, da die meisten Aspekte der Vorarbeit, wenn nicht sogar alle, nicht einfach außerhalb der Unity-Umgebung genutzt werden können. Deshalb wird die Wahl getroffen, weiterhin Unity zu verwenden. Unity bietet jedoch die Möglichkeit, die standardmäßig verwendete Physics-Engine gegen andere, über Plugins bereitgestellte, auszutauschen. MuJoCo ist ebenfalls als ein solches Plugin verfügbar. [4] Insofern könnte bei Bedarf theoretisch mit geringem Aufwand in Betracht gezogen werden, MuJoCo als Physics-Engine in Unity einzubinden, um somit das Gesamtergebnis des Trainings durch eine bessere Physiksimulation zu verbessern. Auch wäre es denkbar, die Ergebnisse der verschiedenen Umgebungen zu vergleichen. Da jedoch keine Übertragung auf einen realen Roboter erfolgt, dürfte in diesem Kontext vermutlich kein spürbarer Unterschied zwischen den Physics-Engines zu beobachten sein. Selbst falls ein messbarer Unterschied bestehen sollte, könnte dieser nicht hinsichtlich seiner Aussagekraft eingeordnet werden.

4.4 Geplante Realisierung

Die Umsetzung der Arbeit lässt sich in mehrere Aufgabenpakete gliedern. Diese sollen folgend beschrieben werden.

4.4.1 Rekonstruktion

Der erste Schritt besteht darin, die Simulationsumgebung und Lernergebnisse der vorherigen Arbeit zu rekonstruieren. Diese Rekonstruktion bringt Probleme mit sich. Einige der verwendeten Komponenten sind einer starken Entwicklung unterlegen – insbesondere das erst 2017 vorgestellte ML-Agents Toolkit, welches sich zum Zeitpunkt der Bearbeitung

des Basisprojekts noch in der Pre-Release-Phase befand. [5] Deshalb sind auf jeden Fall Änderungen nötig, um die bestehenden Ergebnisse überhaupt sichten zu können. Außerdem empfiehlt es sich, die Umgebung auf den aktuellen Stand der Technik zu migrieren, um von potenziellen Verbesserungen im verwendeten Tooling profitieren zu können. Auch wird damit eine zukunftssicherere Basis geboten, auf der die Ergebnisse dieser Arbeit weitergenutzt werden können.

4.4.2 Entwurf der Pfadplanung

Im Anschluss an die Konstruktion einer verwendbaren Simulationsumgebung muss ein Format entworfen werden, wie dem Roboter ein Pfad mitgeteilt werden kann, dem dieser folgen soll. Ein Pfad besteht aus mathematischer Sicht aus einer geordneten Aufreihung an Punkten. Der Roboter muss diese der Reihe nach ansteuern. Wie in Abschnitt 4.2 bereits ausgeführt, muss verhindert werden, dass der Roboter einen vorgegebenen Pfad auswendig lernt. Als logische Folgerung muss bei jeder Trainingsepisode ein zufälliger Pfad vorgegeben werden. Beim Verflogen eines Pfades ist zu jeder Zeit nur der als nächstes anzusteuernde Punkt von Relevanz. Für das Training ergibt dies Parallelen zu einem Beispielszenario des ML-Agents-Toolkits. Beim Crawler-Example muss eine Kreatur lernen, ein zufällig spawnendes Ziel (sogenannte *Dynamic Targets*) zu erreichen. Kommt der Crawler mit einem Dynamic Target in Berührung, teleportiert es sich an einen zufälligen Ort. [6] Dabei tauch die Dynamic Targets als kleine Würfel in der Umgebung des Crawlers auf. Die Dynamic Targets könnte auch für die Pfadplanung des Roboters verwendet werden. Die für das Training notwendigen, zufälligen Pfade bilden sie bereits automatisch ab. Um später einen festen Pfad vorgeben zu können, müsste lediglich eine kleine Modifikation der Dynamic Targets vorgenommen werden, um diese in einer festgelegten Reihenfolge anstatt zufällig erscheinen zu lassen.

4.4.3 Anpassung der Belohnungsfunktion

Bislang wird der Roboter nur für eine zurückgelegte Distanz entlang der x-Achse belohnt und erhält eine Bestrafung, wenn er sich auf den Rücken dreht. Im ersten Schritt soll das Laufverhalten des Roboters stabilisiert werden, damit sich dieser nicht mehr springend

fortbewegt. Dazu könnte etwa in der Belohnungsfunktion ein Faktor eingebracht werden, der den Roboter mit einem, an der Neigung der Zentralplatte skalierten, Wert bestraft.

Um den Roboter dazu zu bringen, zum nächsten Zielpunkt zu laufen, muss die bisherige Belohnung für die Distanz ersetzt werden. Denkbar sind hierfür mehrere Ansätze. Eine Möglichkeit besteht darin, die Distanz zwischen Roboter und Zielpunkt zu bestimmen und mit der Distanz vor der letzten Aktion zu vergleichen. Die andere Möglichkeit wäre, die konkrete Bewegungsrichtung des Roboters zu bestimmen und ein Produkt mit der Geschwindigkeit zu bilden. Diese Möglichkeit würde eventuell eine feinere Gewichtung ermöglichen. Der erste Berechnungsansatz stellt hingegen eine wesentlich kleinere Veränderung zur Ausgangssituation dar, weshalb hierbei die möglichen Fehlerquellen besser eingegrenzt werden können. Es sollen beide Ansätze ausprobiert und miteinander verglichen werden.

4.4.4 Ergänzen von Hindernissen

Als letzter Schritt sollen noch Hindernisse ergänzt werden. Hierfür können einfache Kisten in Unity verwendet werden. Auch die Position dieser Hindernisse darf natürlich nicht von Trainingsalgorithmus auswendig gelernt werden, weshalb die Kisten in jeder Trainingsepisode zufällig platziert werden müssen. Wenn diesen Kisten nun Kollisionsmodelle hinzugefügt werden, kann der Roboter automatisch nicht mehr durch diese Hindernisse hindurchgehen. Die größte Herausforderung sollte auch hier die Adaption der Belohnungsfunktion werden, damit diese den Roboter nicht daran hindert, den Pfad zu verlassen. Gleichzeitig soll der Roboter auch nicht dazu animiert werden, den vorgegebenen Pfad großräumig zu verlassen. Die Anpassungsmöglichkeiten sind hier jedoch relativ eingeschränkt. Möglich wäre, bei jedem Simulationsschritt eine kleine Strafe zu vergeben. Diese könnte den Roboter animieren, nicht vor einem Hindernis stehenzubleiben, weil dies auf lange Sicht eine sehr schlechte Belohnung für ihn bedeutet. Andererseits ist es auch wichtig, eine hohe Entropie für den Roboter zu haben, damit dieser nach Wegen um das Hindernis sucht, anstatt dort im lokalen Maximum steckenzubleiben.

5 Implementierung

6 Bewertung der Ergebnisse

7 Fazit / Future Work

ein Titel oder ggf zwei Kapitel?

Literaturverzeichnis

[1] Waidner, D. *Erlernen von Bewegungsabläufen durch Reinforcement Learning.* Techn. Ber. DHBW Karlsruhe, 2020.

[2] *MuJoCo – Advanced Physics Simulation — mujoco.org.* https://mujoco.org/. [Accessed 22-Apr-2023].

[3] *GitHub - deepmind/mujoco: Multi-Joint dynamics with Contact. A general purpose physics simulator. — github.com.* https://github.com/deepmind/mujoco. [Accessed 22-Apr-2023].

[4] *Unity Plug-in - MuJoCo Documentation — mujoco.readthedocs.io.* https://mujoco.readthedocs.io/en/latest/unity.html. [Accessed 22-Apr-2023].

[5] *Announcing ML-Agents Unity Package v1.0! — Unity Blog — blog.unity.com.* https://blog.unity.com/technology/announcing-ml-agents-unity-package-v1-0. [Accessed 22-Apr-2023].

[6] Technologies, U. *Example Learning Environments - Unity ML-Agents Toolkit — unity-technologies.github.io.* https://unity-technologies.github.io/ml-agents/Learning-Environment-Examples/#crawler. [Accessed 22-Apr-2023].