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 4. Mai 2023

Schiebelhut, Yannik

Abstract

- Deutsch -

Platzhalter

Abstract

- English -

Placeholder

Inhaltsverzeichnis

Abkürzungsverzeichnis **V**

Abbildungsverzeichnis **VI**

Quellcodeverzeichnis **VII**

1 Einleitung **1**

2 Grundlagen **2**
2.1 Machine Learning 2
2.2 Unity3D 5
2.3 Vektorgeometrie 11
2.4 Beschreibung der Projektbasis 13

3 State of the Art **17**

4 Konzeptionierung **20**
4.1 Einschränkungen und Übertragungsprobleme 20
4.2 Wahl der Simulationsumgebung 22
4.3 Geplante Realisierung 23

5 Implementierung **26**

6 Bewertung der Ergebnisse **27**

7 Fazit / Future Work **28**

Literaturverzeichnis **VIII**

Abkürzungsverzeichnis

CLI	Command Line Interface
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
SAC	Soft Actor-Critic

Abbildungsverzeichnis

2.1 Die Interaktion von Agent und Umgebung als Markov Decision Process (MDP) [2, S. 48] 3
2.2 Vereinfachtes Block-Diagramm des ML-Agents-Toolkits [8] 7
2.3 Vektor von Punkt A zu Punkt B 12
2.4 Zerlegung des zweidimensionalen Vektors $\overrightarrow{v}$ 12

3.1 Dezentrale Architektur in Anlehnung an Stabinsekten [19] 18
3.2 Transfer von Trainingsdaten zur Bewältigung komplexer Probleme [21] 19

Quellcodeverzeichnis

1 Einleitung

Die Robotik ist ein breites Forschungsfeld mit praktisch grenzenlosen Möglichkeiten. Die Fähigkeit, sich zu bewegen, ist dabei besonders spannend, da sie die Flexibilität der Einsatzmöglichkeiten für Roboter stark erhöht. Roboter mit Beinen stellen sich besonders heraus. Mit Inspirationen aus Mensch- und Tierreich bieten diese das Potenzial, sich in jedem denkbaren Terrain fortzubewegen, das auch für Menschen zugänglich ist. Im Vergleich zu anderen Fortbewegungsarten stellt die stabile Koordination von mehreren Beinen, die jeweils aus mehreren Gelenken bestehen, allerdings eine große Herausforderung für die technische Umsetzung dar. In der Regel sind für die Programmierung solcher Roboter eine sehr genaue Kenntnis der Maschine und deren Dynamik vonnöten.

In den vergangenen Jahren wird deshalb verstärkt erforscht, wie Roboter sich diese Fähigkeiten selbstständig mittels Reinforcement Learning beibringen können. In einer vorangegangenen Studienarbeit wurden am Beispiel eines eigens dafür gebauten, vierbeinigen Roboters Möglichkeiten erforscht, um diesen mittels selbst gelernter Bewegungsabläufe möglichst effizient und schnell geradlinig nach vorne zu bewegen. Um den Trainingsprozess zu beschleunigen wurde dabei der Roboter in der Simulationsumgebung „Unity“ nachgebaut und trainiert.

Für das Erfüllen eines praktischen Nutzens ist es jedoch in der Regel nicht ausreichend, wenn sich ein Roboter nur in eine feste Richtung bewegen kann. Im Rahmen dieser Arbeit wird deshalb untersucht, wie dem Roboter beigebracht werden kann, einem gezielt übergebenen Pfad zu folgen. Weiterhin soll der Roboter dabei Hindernisse, die sich auf diesem Pfad befinden, automatisch umsteuern und anschließend wieder auf den vorgebenen Pfad zurückkehren. Dazu werden zunächst die Arbeitsumgebung und Ergebnisse der vorherigen Arbeit rekonstruiert. Anschließend wird diskutiert, welche Änderungen am Roboter und dessen Simulationsumgebung vorgenommen werden müssen, um die erweiterten Anforderungen grundsätzlich erfüllen zu können. Außerdem wird erläutert, wie die Aufgabe in sinnvolle Teilaufgaben gegliedert werden kann. Zur Bearbeitung dieser Teilaufgaben wird dann ein Konzept erarbeitet, welches im Anschluss in der Simulationsumgebung, unter Zuhilfenahme des ML-Agents-Toolkits, mittels des Proximal Policy Optimization-Algorithmus ein Proof-of-Concept trainiert.

2 Grundlagen

2.1 Machine Learning

Machine Learning ist eine Unterkategorie der künstlichen Intelligenz und bezeichnet einen automatisierten Prozess, der es Computern ermöglicht, eigenständig aus Trainingsdaten zu lernen und sich mit der Zeit zu verbessern, ohne explizit zur Lösung einer Aufgabe programmiert zu werden. Machine Learning Algorithmen können Muster in Daten entdecken und aus ihnen Lernen, um eigene Prognosen und Entscheidungen zu treffen.

In der Regel wird Machine Learning in folgende Teilbereiche untergliedert [1]:

- **Supervised Learning:** Ein Modell wird anhand von gelabelten Trainingsdaten trainiert. Ein Datentupel besteht dabei aus einer Eingabe und der dazu gewollten Ausgabe. Der Algorithmus sucht beim Training nach Zusammenhängen und Abhängigkeiten um anschließend Ausgaben für unbekannte Eingaben generieren zu können. Üblicherweise wird Supervised Learning für Regressions- und Klassifikations-Probleme eingesetzt.
- **Unsupervised Learning:** Wird in der Regel für Clusterbildung von unbeschrifteten Trainingsdaten verwendet. Der Algorithmus muss dabei selbstständig nach Mustern in den Daten suchen. Unsupervised Learning kann dabei helfen, Einblicke in große Datensätzen zu erhalten, um etwa versteckte Trends zu entdecken.
- **Semi-Supervised Learning:** Für das Training werden beim Semi-Supervised Learning sowohl ein kleiner gelabelter, als auch ein großer ungelabelter Datensatz verwendet. Dabei werden die Vorzüge von Supervised und Unsupervised Learning miteinander verbunden. Interessant ist dies vor allem bei sehr großen Datensätzen (zum Beispiel bei Bild-Klassifizierung), da die Labelung der Daten in der Regel manuell erfolgen muss.
- **Reinforcement Learning:** Beim Reinforcement Learning kann ein Agent Aktionen tätigen, für die er entweder belohnt oder bestraft wird. Es ist sein Ziel, selbstständig ein bestmögliches Verhalten zu lernen, um seine Belohnung zu maximieren. Dabei

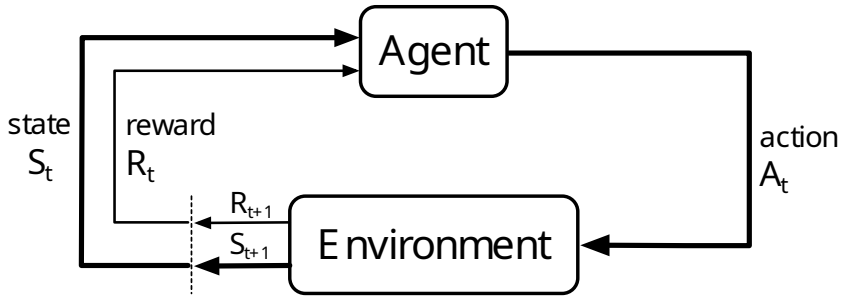

Abbildung 2.1: Die Interaktion von Agent und Umgebung als MDP [2, S. 48]

werden keine Trainingsdaten verwendet. Der Agent lernt ausschließlich aus seinen eigenen Erfahrungen und Fehlern. Reinforcement Learning findet vor allem in den Bereichen Robotik und Videospiele Einsatz und wird auch in dieser Arbeit verwendet werden.

2.1.1 Reinforcement Learning

Reinforcement Learning ist ein rechnerischer Ansatz, um zielorientierte Lern- und Entscheidungsprozesse zu verstehen und nachzubilden. Wie oben schon beschrieben, steht dabei ein *Agent* im Zentrum, der aus der direkten Interaktion mit seiner *Umgebung (engl. Environment)* lernt, ohne dabei eine beispielhafte Anleitung oder vollständige Modelle der Umgebung zu benötigen. Das formale Framework des *MDP* wird genutzt, um die Interaktion zwischen dem lernenden Agenten und seiner Umgebung zu definieren. MDPs sind eine mathematisch idealisierte Form eines Reinforcement Learning Problems, für die präzise, theoretische Aussagen getroffen werden können [2, S. 13]. Abbildung 2.1 stellt die grundlegende Struktur eines Reinforcement Learning Problems dar. Die einzelnen Bestandteile der Abbildung werden nachfolgend erläutert.

Agenten haben explizite Ziele, können Aspekte ihrer Umgebung wahrnehmen und *Aktionen A* auswählen, um mit ihrer Umgebung zu interagieren und diese zu beeinflussen. Es wird davon ausgegangen, dass Reinforcement Learning diejenige Strategie des Machine Learning ist, die dem natürlichen Lernen von Menschen und Tieren am nächsten kommt. Viele zentrale Algorithmen des Reinforcement Learnings sind ursprünglich durch biologische Systeme inspiriert [2, S. 4].

Besonders wichtig für Reinforcement Learning ist das Konzept von *Zuständen S*. Ein Zustand kann dabei als eine Art Signal verstanden werden, das dem Agent Informationen über den Zustand der Umgebung liefert. Weiterhin definieren folgende Elemente ein Reinforcement Learning Problem:

- **Policy π:** Die Policy definiert, wie sich der Agent zu einer gegebenen Zeit verhält. Sie stellt ein Mapping zwischen den wahrgenommenen Zuständen der Umgebung und den durchzuführenden Aktionen dar. Die Policy ist hinreichend, um das Verhalten des Agent zu bestimmen [2, S. 6].
- **Reward-Signal R:** Das Reward-Signal definiert das Ziel eines Reinforcement Learning Problems. Bei jedem *Zeitschritt t* sendet die Umgebung ein Skalar an den Agent. Das einzige Ziel des Agent ist die Maximierung des kumulativen Rewards. Der Reward ist die primäre Basis für Änderungen an der Policy [2, S. 6].
- **Value-Funktion V:** Die Value-Funktion legt fest, was auf lange Sicht gut ist. Der Value eines Zustands ist der kumulierte Reward, den ein Agent, ausgehend von diesem Zustand, in der Zukunft erwarten kann. Values geben die langfristige Attraktivität von Zuständen an. Die Wahl einer Aktion wird auf Basis der Value-Einschätzung des aktuellen Zustands getroffen. Im Vergleich zum Reward-Signal sind Values allerdings deutlich schwerer zu bestimmen, da diese anhand einer Sequenz von Observationen des Agenten geschätzt werden müssen [2, S. 6].
- **Modell der Umgebung (optional):** Manche Reinforcement Learning Systeme nutzen ein Modell der Umgebung. Dieses Modell erlaubt das Ziehen von Schlussfolgerungen über das Verhalten der Umgebung. So kann etwa eine Voraussage des nächsten Resultierenden Zustands und Rewards, ausgehend von einem gegebenen Zustand und einer Aktion getroffen werden. Genutzt werden diese Modelle zur Planung. Es werden also Entscheidungen für eine Folge von Aktionen auf Basis möglicher zukünftiger Situationen getroffen, bevor diese tatsächlich erlebt werden. Reinforcement Learning Methoden, die Modelle und Planung verwenden, werden als *modell-basiert* bezeichnet. Im Gegensatz dazu stehen *modell-freie* Methoden, welche explizit auf Basis von Trial-And-Error lernen [2, S. 7].

Beim Reinforcement Learning versucht der Agent, mit seinen ausgeführten Aktionen ein Reward-Signal zu maximieren. Dabei muss ein Kompromiss zwischen Nutzen des Gelerntem und Entdecken von Neuem gefunden werden. Es besteht das Dilemma, dass

weder das eine, noch das andere uneingeschränkt verfolgt werden kann, ohne bei der Ausführung der Aufgabe zu scheitern, denn beim Entdecken muss der Agent auch schlechte Aktionen ausführen, für die er keinen Reward erhält. Entdeckt er jedoch nichts, weiß er auch nicht, welche Aktionen einen hohen Reward erzeugen [2, S. 3].

2.1.2 Deep Reinforcement Learning

Wie auch andere Algorithmen haben Reinforcement Learning Algorithmen Skalierungsprobleme hinsichtlich ihrer Komplexität. So kommt es beispielsweise zu Schwierigkeiten, die Value- oder die Policy-Funktion abzubilden, wenn die Dimension des Reinforcement Learning Problems zu groß wird. Insbesondere bei hoch-dimensionalen, kontinuierlichen Zustands- und Aktionsräumen ist dies der Fall.

Die wichtigste Eigenschaft von Deep Learning sind Deep Neural Networks. Diese Netzwerke können automatisch kompakte, niedrig-dimensionale Repräsentationen von hoch-dimensionalen Daten finden. Beim Deep Reinforcement Learning werden Algorithmen und Technologien des Deep Learning in Reinforcement Learning eingebracht. Dabei werden Deep Neural Networks als Funktionsapproximatoren für die Value-Funktion oder die Policy verwendet. Diese neue Kombination macht eine Skalierung auf bislang unlösbare Entscheidungsprobleme möglich [3].

2.2 Unity3D

Unity3D ist eine plattformübergreifende Game Engine, die erstmal 2005 angekündigt wurde. Primärer Zweck der Unity Engine ist die Entwicklung von Videospielen für Computer, Konsolen und Mobilgeräte. Dabei ist Unterstützung für zwei- und dreidimensionale Grafik enthalten. VR Entwicklung ist ebenso möglich. Das Skripting innerhalb der Engine erfolgt primär in C# [4]. Neben dem Einsatz in der Spieleentwicklung ist Unity jedoch auch für den Einsatz in anderen Branchen geeignet, so zum Beispiel in der Architektur oder der Forschung [5, S. 30], wo mit Unity Simulationen der realen Welt erstellt werden können. Als Grafik-APIs werden unter anderem Direct3D (Windows), OpenGL (Linux, macOS, Windows) und WebGL unterstützt. Unity enthält einen Asset Store für die Entwickler-Community, über den Dritten das Hoch- und Herunterladen kommerzieller

Abkürzung

und freier Ressourcen (zum Beispiel Texturen, Modelle und Plugins) ermöglicht wird [4]. Der Einsatz von Unity für Projekte mit weniger als 100.000 $ jährlichem Gewinn ist kostenlos [6].

2.2.1 Unity Machine Learning Agents Toolkit

Das Unity Machine Learning Agents Toolkit (kurz *ML-Agents*) ist ein von Unity Technologies entwickeltes, quelloffenes Projekt, welches 2017 erstmals als Testversion veröffentlicht wurde, seitdem sehr aktiv weiterentwickelt wird und inzwischen die Produktreife erreicht hat. ML-Agents ermöglicht es Spiele und Simulationen, als Trainingsumgebung für intelligente Agenten zu dienen. Dabei werden State-of-the-Art, PyTorch-basierte Implementierungen gängiger Machine Learning Algorithmen angeboten, um ein einfaches Training mit möglichst geringer Einstiegshürde zu ermöglichen. Alternativ können auch eigene Algorithmen zum Training verwendet werden. Wie Unity selbst auch ist ML-Agents für den Einsatz in 2D-, 3D- und VR/AR-Umgebungen geeignet. Als Trainingsmethoden werden unter anderem Reinforcement Learning, Imitation Learning und Neuroevolution unterstützt [7].

ML-Agents besteht aus folgenden high-level Komponenten [8] (siehe Abbildung 2.2):

- **Trainingsumgebung (Learning Environment):** Die Trainingsumgebung enthält eine Unity Szene und sämtliche Game Charaktere. Die Unity Szene stellt dabei die Umgebung bereit, in der Agenten ihre Beobachtungen machen, handeln und lernen. Mithilfe des ML-Agents Unity SDK kann jede Unity Szene in eine Trainingsumgebung transformiert werden, indem Game Objects als Agenten definiert werden.
- **Python Low-Level API:** Die Python API enthält ein low-level Python Interface, welches die Aufgabe besitzt, mit der Trainingsumgebung zu interagieren und diese zu manipulieren. Diese Python API ist im Gegensatz zur Trainingsumgebung kein Teil von Unity, sondern kommuniziert mit Unity durch den Communicator.
- **Externer Communicator:** Der Communicator erfüllt die Aufgabe, die Python API mit der Trainingsumgebung zu verbinden.

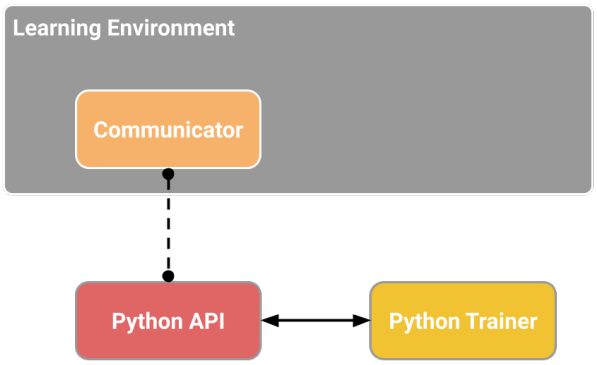

Abbildung 2.2: Vereinfachtes Block-Diagramm des ML-Agents-Toolkits [8]

- **Python Trainer:** In den Python Trainern sind alle Machine Learning Algorithmen enthalten, die ein Training der Agenten ermöglicht. Dieses Paket stellt die zum Training genutzte Command Line Interface (CLI) (`mlagents-learn`) bereit.

Die Trainingsumgebung wird durch zwei enthaltene Unity-Komponenten organisiert [8].

- **Agents** sind an ein Unity GameObject geknüpft (beliebiger Charakter innerhalb einer Szene). Sie sind gleichzusetzen mit dem Agent eines Reinforcement Learning Problems. Agents generieren die Observations (Beobachtungen) des GameObjects, welche dem Reinforcement Learning Algorithmus zugeführt werden, führen die vom Algorithmus empfangenen Aktionen aus und weisen den Reward zu. Jeder Agent ist mit einem Behavior verknüpft.
- **Behaviors** (Verhalten) definieren Attribute des Agenten, so auch die Anzahl der Aktionen, die der Agent entgegennehmen kann. Ein Behavior kann als Funktion verstanden werden, welche Observations und Reward des Agents als Eingabeparameter enthält und auszuführende Aktionen zurückliefert. Behaviors werden in drei Typen unterschieden: *Learning*, *Heuristic* und *Inference*. Learning Behaviors sind noch nicht definiert, können aber trainiert werden. Heurisitc Behaviors werden mittels manuell implementierter Regeln im Quellcode definiert. Inference Behavior werden von trainierten Neural-Network-Dateien (entsprechen der finalen, trainierten Policy) repräsentiert. Nachdem ein Learning Behavior trainiert wurde, wird es zum Inference Behavior.

Herausstellenswert hierbei ist, dass ML-Agents die Möglichkeit bietet, mehrere Agents in einer Trainingsumgebung zu platzieren, die jedoch mit demselben Behavior verknüpft

sein können. Dies kann dafür genutzt werden, das Training zu parallelisieren und damit zu beschleunigen.

2.2.2 Reinforcement Learning Algorithmen

Von ML-Agents werden zwei Trainingsalgorithmen bereitgestellt, die sich dem (Deep) Reinforcement Learning zuordnen lassen. Dies sind Proximal Policy Optimization (PPO) und Soft Actor-Critic (SAC) [8]. In [5] wurden diese Algorithmen bereits verglichen. PPO ist der Standardalgorithmus von ML-Agents, da er sich, verglichen mit vielen anderen Reinforcement Learning Algorithmen, als für den allgemeinen Einsatz besser geeignet gezeigt hat [9, 10]. PPO ist ein on-policy Algorithmus. Das bedeutet, dass in jeder Iteration des Lernvorgangs nur aus Erfahrungen gelernt wird, die mit der aktuellen Version der Policy gesammelt wurden. SAC hingegen ist ein off-policy Algorithmus und lernt somit aus allen Erfahrungen, die er jemals während des gesamten Trainingsvorgangs gesammelt hat [11, 12]. Daraus ergeben sich für beide Algorithmen unterschiedliche Vor- und Nachteile. On-policy Algorithmen haben in der Regel einen deutlich stabileren Lernfortschritt, als off-policy Algorithmen. Andererseits brauchen on-policy Algorithmen in der Regel deutlich mehr Trainingsschritte, um nennenswerte Ergebnisse zu erzielen [8]. Im Zuge der Vorgängerarbeit wurde mit beiden Algorithmen gearbeitet, mit dem Ergebnis, dass der PPO-Algorithmus auch für das konkrete Problem im Rahmen dieser Studienarbeit deutlich bessere Resultate liefert [5, S. 48].

2.2.3 Training und Hyperparameter

Wenn die Trainingsumgebung erstellt ist, kann mit dem Training der Agenten begonnen werden. Der Einstiegspunkt hierzu ist immer die CLI `mlagents-learn`, welche Teil der Python-Umgebung von ML-Agents ist. Das Training kann dann entweder direkt im Unity-Editor oder mittels einer kompilierten Umgebung erfolgen. Falls letztere verwendet werden soll, kann sie mittels des Parameters `--env=<env_name>` spezifiziert werden. Einem Training sollte in der Regel auch ein Run-Identifier gegeben werden (`--run-id=<run-identifier>`). Über diesen können die Ergebnisse besser zugeordnet werden und es ist beispielsweise auch möglich, ein unterbrochenes Training wiederaufzunehmen [13].

Optional kann das Training mithilfe einer YAML-Datei konfiguriert werden. In dieser können sowohl allgemeine Aspekte des Trainingsvorgangs festgelegt werden (etwa wie viele Trainingsschritte durchgeführt werden sollen), als auch Hyperparameter eingestellt werden, die spezifisch für das jeweils durchzuführende Training sind. Für ein Training mit dem PPO-Algorithmus sind vor allem die folgenden Hyperparameter von Bedeutung [14]:

- **batch_size:** Anzahl der Erfahrungen in jeder Iteration des Gradientenabstiegs.
- **buffer_size:** Anzahl der Erfahrungen, die gesammelt werden, bevor das Policy Modell aktualisiert wird, also wie viele Erfahrungen gesammelt werden, bevor gelernt wird.
- **learning_rate:** Initiale Lernrate für den Gradientenabstieg. Damit kann gesteuert werden, wie schnell das Modell am Anfang lernt. Sollte so groß wie möglich gewählt werden, um ein schnelles Training durchzuführen. Wenn das Training jedoch instabil verläuft, sollte dieser Wert verringert werden.
- **learning_rate_schedule:** Gibt an, wie die Lernrate über die Zeit verändert wird. Für PPO wird diese in der Regel linear verringert, damit das Training möglichst stabil konvergiert.
- **beta:** Faktor um die Entropie des Trainingsprozesses zu regulieren. Die Entropie sollte gegenläufig zum Reward langsam im Laufe des Trainingsprozesses fallen. Mit einer Erhöhung des beta-Werts wird die Entropie länger auf einem höheren Level gehalten und umgekehrt.
- **epsilon:** Begrenzt die Veränderung der Policy während des Trainingsprozesses. Wird epsilon klein gewählt, so werden Änderungen an der Policy kleinschrittiger vorgenommen, was das Training stabilisiert, aber auch mehr Trainingsschritte benötigt.
- **lambd:** Parameter zur Regularisierung während der Berechnung des „Generalized Advantage Estimate“. Prinzipiell gibt der Parameter an, wie stark der Agent auf seine aktuelle Schätzung des Values aufbaut, während die Schätzung des Values aktualisiert wird. Bei einem niedrigen Wert von lambd liegt das Gewicht beim aktuell geschätzten Value (der einen hohen Bias haben kann) und bei einem hohen Wert werden die eigentlichen Rewards höher gewichtet (welche jedoch einer hohen Varianz unterliegen können).

- **num_epoch:** Wie viele Durchläufe durch die gesammelten Erfahrungen gemacht werden können, wenn ein Gradientenabstieg durchgeführt wird. Ein kleiner Wert führt zu stabilerem, aber auch langsamerem Training.
- **gamma:** Diskontinuierungsfaktor für zukünftige Rewards. Gibt an, wie weit der Agent bei Spekulation auf mögliche Rewards in die Zukunft „denken“ soll.
- **hidden_units:** Anzahl der Neuronen in den Hidden Layers des zu trainierenden Neural Network. Sollte mit der Komplexität des Problems nach oben skaliert werden.
- **num_layers:** Anzahl der Hidden Layers des Neural Networks. Analog zur Anzahl der Neuronen sollte dieser Wert mit der Komplexität des Problems skaliert werden. Weniger Layer trainieren in der Regel schneller, können aber unter Umständen zur Abbildung komplexer Probleme nicht ausreichen.
- **normalize:** Gibt an, ob die eingegebenen Observation Vektoren normalisiert werden. Bei Problemen mit komplexen, kontinuierlichen Beobachtungs- und Aktionsräumen kann dies hilfreich sein.

Die in der Regel verwendeten Befehle zur Initiierung des Trainings folgen grundlegend folgendem Aufbau [13]:

```
mlagents-learn <yaml-config> --env=<env_name> --run-id=<run-id>
```

2.2.4 Bewertungskriterien

Der Verlauf des Trainings kann mittels des Tools *Tensorboard* überwacht werden. Tensorboard startet dabei einen lokalen Webserver, der Auswertungen der Trainingsdaten im Browser darstellt. Um das Ergebnis eines Trainings objektiv zu bewerten, muss natürlich das eigentliche, resultierende Modell betrachtet werden, jedoch geben auch einige der in Tensorboard dargestellten Metriken frühzeitig Aufschluss über den potenziellen Erfolg des Trainings. Allen voran steht der *Kumulierte Reward*. Dieser Wert gibt den mittleren kumulierten Reward innerhalb einer Episode von allen Agenten an. Natürlich hängt der Verlauf dieser Kurve stark davon ab, wie die individuelle Reward-Funktion gewählt wird, allerdings sollte sich dieser Wert bei einem erfolgreichen Training stetig erhöhen. Die *Episodenlänge* zeigt an, wie lange die Episoden im Mittel gedauert haben. Wird die Umgebung etwa beim unwiderruflichen Scheitern eines Agenten zurückgesetzt, so kann

hier abgelesen werden, wie lange es im Mittel dauert, bis es zu einem fatalen Scheitern kommt. *Policy Loss* hängt davon ab, wie stark die Policy sich verändert. Der Betrag dieser Funktion sollte im Laufe des Trainings abnehmen, wenn die Policy zu einer stabilen Funktion konvergiert. Schließlich gibt *Value Loss* an, wie stark die vorhergesagten Values von den tatsächlich erhaltenen Rewards abweichen. Bei einem erfolgreichen Training sollte dieser Wert zu Beginn ansteigen und dann im Anschluss gegen einen niedrigen Wert konvergieren [15, 16].

Weiterhin werden in Tensorboard die Hyperparameter im Laufe der Zeit dargestellt. Entsprechend den Beschreibungen in Unterabschnitt 2.2.3 können auch diese Werte live mitverfolgt und gegebenenfalls korrigiert werden.

2.3 Vektorgeometrie

Da Vektoren im dreidimensionalen Raum ein essenzieller Bestandteil der Arbeit mit Unity sind, sollen die dafür wichtigen Konzepte hier kurz vorgestellt werden.

Grundsätzlich gesehen ist ein Vektor ein mathematisches Konstrukt und hat eine Länge und eine Richtung. Im Gegensatz zu Punkten geben Vektoren keinen festen Ort an, sondern beschreiben den Weg von einem Punkt zu einem anderen [17, S. 6]. Dabei besitzt ein Vektor für jede Dimension eine Komponente. Trotzdem ist eine Ortsangabe mit einem Vektor möglich (und wird in Unity für die Game Objekte verwendet). Dieses Konstrukt besteht aus einem Vektor, der an einem bestimmten Punkt beginnt und nennt sich Ortsvektor [17, S. 21]. Im Weiteren Verlauf wird davon ausgegangen, dass Ortsvektoren dem Koordinatenursprung entstammen.

Abbildung 2.3 stellt die Ortsvektoren $\overrightarrow{a}$ und $\overrightarrow{b}$ dar, welche vom Koordinatenursprung auf die Punkte A und B zeigen. Der Vektor $\overrightarrow{AB}$ stellt den Vektor dar, der von A zu B führt. Addiert man mehrere Vektoren, so erhält man den daraus resultierenden Vektor [17, S. 11]. Wie in Gleichung (2.1) ersichtlich ist, kann durch Addition von $\overrightarrow{a}$ und $\overrightarrow{AB}$ wiederum $\overrightarrow{b}$ bestimmt werden. Mittels eines Umstellens der Gleichung kann aber auch aus den beiden Ortsvektoren der Verbindungsvektor $\overrightarrow{AB}$ bestimmt werden [17, S. 12].

Für die Länge eines Vektors (auch Norm genannt), gibt es verschiedene Definitionen. Die hier relevante und erklärte ist die sogenannte *euklidische Norm*. Zur Bestimmung dieser

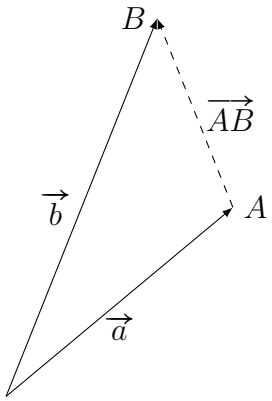

Abbildung 2.3: Vektor von Punkt A zu Punkt B

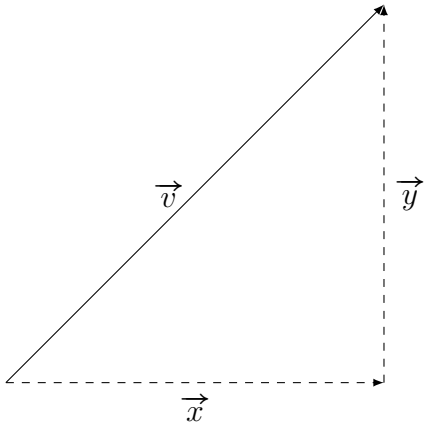

Abbildung 2.4: Zerlegung des zweidimensionalen Vektors $\overrightarrow{v}$

wird die Summe aus den Quadraten der Komponenten gebildet und aus dieser Summe im Anschluss die Quadratwurzel gezogen [17, S. 30] (Gleichung (2.3)). Der Hintergrund liegt hierbei im Satz des Pythagoras, was ersichtlich wird, wenn man sich die Zerlegung eines zweidimensionalen Vektors in seine einzelnen Komponenten anschaut (Abbildung 2.4, Gleichung (2.2)).

$$\overrightarrow{a} + \overrightarrow{AB} = \overrightarrow{b} \quad \Rightarrow \quad \overrightarrow{AB} = \overrightarrow{b} - \overrightarrow{a} \tag{2.1}$$

$$\overrightarrow{v} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \Rightarrow \quad |\overrightarrow{v}| = \sqrt{x^2 + y^2} \tag{2.2}$$

$$\overrightarrow{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \Rightarrow \quad |\overrightarrow{v}| = \sqrt{x^2 + y^2 + z^2} \tag{2.3}$$

Dadurch, dass Vektoren eine Richtung darstellen, ist es ebenfalls möglich, den Winkel β zu bestimmen, der zwischen diesen Vektoren liegt, wenn man sie aufeinander stellt. Dafür wird zunächst das sogenannte Skalarprodukt der beiden Vektoren gebildet [17, S. 45] (Gleichung (2.4)). Anschließend wird das Skalarprodukt durch das Produkt der Längen beider Vektoren geteilt und in die arccos Funktion eingesetzt [17, S. 60] (Gleichung (2.5)).

$$\overrightarrow{v} \cdot \overrightarrow{w} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 \tag{2.4}$$

$$\beta = \arccos\left(\frac{\overrightarrow{v} \cdot \overrightarrow{w}}{|\overrightarrow{v}| \cdot |\overrightarrow{w}|}\right) \tag{2.5}$$

2.4 Beschreibung der Projektbasis

Die Zielsetzung dieser Arbeit soll aufbauend auf einer Vorgängerarbeit [5] 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.

2.4.1 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 drei separaten Teilen. Folglich hat jedes Bein zwei Gelenke [5, 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° [5, 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 I²C 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 [5, 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 [5, 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 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 mittleren Bewegungsgeschwindigkeit berücksichtigt wird, sowie verwandte, allgemeine Charakteristika der Bewegung [5, 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.

2.4.2 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 besteht aus einer reinen Vorwärtsbewegung. Da der reale Roboter über keine Sensorik verfügt und lediglich die Ansteuerwinkel seiner Servomotoren kennt, sind dies auch die einzigen Beobachtungen, die dem Algorithmus zugänglich gemacht werden (kontinuierlicher Beobachtungsraum). Als Aktionen kann der Algorithmus beliebige Zielwinkel für die Servomotoren setzen (kontinuierlicher Aktionsraum). Die Belohnung für den Roboter besteht in der Vorgängerarbeit aus der Streckendifferenz, die er nach vorne zurückgelegt hat. Bestraft wird 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 [5, S. 48].

Die Bewegungsart, die der Roboter sich bei den vorgegebenen 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 [5, S. 51].

2.4.3 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 nachahmt. Wird der Roboter jedoch auf den Boden gestellt, knickt er unter seinem Gewicht sofort ein und kann die gelernte Laufmethodik nicht anwenden [5, S. 58].

3 State of the Art

Aktuell gibt es in der Bewegungsforschung von Robotern mit mehreren Beinen drei vorherrschende Ansätze [18]:

- **Central Pattern Generators** (sind Oszilatorsysteme oder neurale Netze, die eine rhythmische Ausgabe erzeugen, ohne eine bestimmte Eingabe vorauszusetzen.)
- **lernende Ansätze** (darunter fällt auch der in dieser Arbeit thematisierte Reinforcement Learning Ansatz)
- **sensorbasierte Ansätze** (sind besser erklärbar im Vergleich zu lernenden Ansätzen. Normalerweise sind sie relativ anpassbar an sich verändernde Umgebungen.)

Simmering et al. [18] erforschen fortgeschrittene Laufmethoden der sensorbasierten Walknet-Architektur und sehen vor allem ein Potenzial in der Verbindung mehrerer dieser Ansätze.

Während Machine Learning in den letzten Jahren erfolgreich auf viele Aufgaben angewandt wurde, treffen lernende Ansätze in der Anwendung bei realen, mehrbeinigen Robotern – welche kontinuierlichen Kontroll-Aufgaben darstellen – noch auf einige Probleme [19]. Einerseits werden Roboter aus Praktikabilitäts- und Sicherheitsgründen meist in Simulationen trainiert. (Häufig werden auch zunächst in der Simulation Grundsteine gelegt und dann die daraus gewonnenen Modelle manuell für eine bestimmte Aufgabe feingeschliffen [19].) In der Folge kann allerdings der Transfer auf reale Probleme erschwert werden, da etwa in der realen Anwendung deutlich mehr (Signal-)Rauschen von Eingabesensoren auftritt, was zur Hinterfragung führt, ob tatsächlich ein fester MDP vorliegt [19]. Andererseits ist es ein fundamentales Problem, dass beim (Deep) Reinforcement Learning gelernt wird, den Reward möglichst gut auszunutzen, weshalb diese Art von Lernproblemen zu Overfitting neigt. Das bedeutet, dass Deep Reinforcement Learning dazu tendiert, Nischenlösungen zu finden, die meist nicht dazu in der Lage sind, adaptiv auf neue, vor allem unvorhergesehene Situationen zu reagieren [19].

Schilling et al. [19] schlagen deshalb einen dezentralisierten Ansatz vor, der dem Nervensystem von Insekten ähnelt (Abbildung 3.1). Im Gegensatz zu [5] wird dabei nicht ein

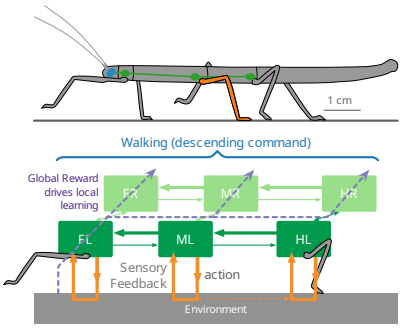

Abbildung 3.1: Dezentrale Architektur in Anlehnung an Stabinsekten [19]

zentraler Controller verwendet, der alle Beine steuert und sämtliche Zustandsinformationen erhält, sondern jedes Bein wird durch einen eigenen Controller mit reduzierten Zustandsinformationen gesteuert. Vorteilhaft ist hieran vor allem, dass die Komplexität/Dimensionalität des Problems deutlich verringert wird verglichen mit Ansätzen, die einen zentralen Controller verwenden. Versuche resultierten in der Erkenntnis, dass die erzielten Resultate mit beiden Algorithmen ähnlich gut ausfielen. Jedoch hatte der dezentrale Ansatz eine rapide, fast halbierte Trainingszeit, um dieses Ergebnis zu erreichen [19]. Verwendet wurde in beiden Fällen der PPO-Algorithmus, welcher in der aktuellen Forschung allgemein gute Ergebnisse bei kontinuierlichen Problemen liefert, ohne dabei extensive Anpassungen der Hyperparameter zu erfordern [19]. Im Kontext dieser Arbeit ist festzustellen, dass der im Experiment verwendete Roboter über deutlich mehr und detailliertere Eingangsdaten verfügte, die unter anderem über Sensoren gesammelt wurden (z.B. auch Sensoren für Bodenkontakt der Beine und Körperneigung). Aufgrund der sehr eingeschränkten Informationen, die dem Roboter in dieser Arbeit zur Verfügung stehen, scheint die Verfolgung eines dezentralen Ansatzes leider nicht erfolgversprechend.

Tsounis et al. [20] verfolgen einen bislang komplett neuen Ansatz, der vor allem für die Fortbewegung in unwegsamen Gelände erfolgversprechend scheint. Dabei ist die Idee, die Schrittplanung als Aufgabe von der eigentlichen Durchführung der Schritte abzukoppeln. Dadurch können zwei unabhängige Modelle trainiert und verwendet werden, die jeweils eine deutlich geringere Dimension haben, als es bei einem gemeinsamen Überproblem der Fall wäre. Außerdem bieten solche Ansätze die Möglichkeit, einzelne Komponenten unabhängig voneinander zu optimieren und auszutauschen. Weiterhin evaluieren Tsounis et al. für das Training der Schrittplanung nur die physikalische Machbarkeit der Schritte,

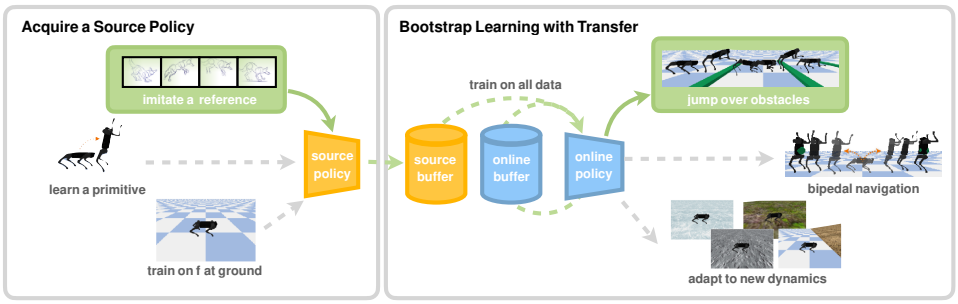

Abbildung 3.2: Transfer von Trainingsdaten zur Bewältigung komplexer Probleme [21]

anstatt die Umgebung vollständig zu simulieren. Dabei ist eine deutlich höhere Genauigkeit bei weniger Rechenaufwand möglich, als wenn die Simulation eines Schritts Frame für Frame geladen und evaluiert würde. Die erzielten Resultate sind weit besser als die von vergleichbaren Ansätzen – insbesondere für das Überwinden von Klüften. Allerdings ist dazu auch anzumerken, dass die Modelle auf dem Terrain trainiert wurden, auf das sie später angewandt wurden. Dennoch konnte eine gute Verallgemeinerung festgestellt werden. Ein wichtiges Fazit der Arbeit war, dass eine ausreichend hohe Entropie während des Trainings maßgeblich für den Gesamterfolg verantwortlich ist.

Abschließend soll hier noch der Ansatz von Smith et al. [21] erwähnt werden. Die Idee hier ist, agile und komplexe Bewegungsfähigkeiten durch Übertragung von Lernerfahrungen zu erlernen und anzupassen. Der Hintergrundgedanke beruht darauf, dass selbst einfache Aufgaben schon sehr komplex modulierte Reward-Funktionen benötigen können, um eine gezielte Bewegung zu erhalten. Wenn nun eine noch deutlich komplexere Bewegungsform von Null trainiert wird, kann dies teilweise fast unmöglich zu bewältigen sein, da nur ein sehr bestimmtes Verhalten Reward nach sich ziehen kann. Für ein spezifisches Zielproblem zu trainieren mag zwar scher sein, doch häufig ist es zumindest möglich, mit einfacheren Trainingsumgebungen und Problemen zumindest relevante Daten zu erhalten, welche für einen Lernprozess von Interesse sind. So sollen Daten von beliebigen Trainingsprozessen transformiert und transferiert werden können, um neue Fähigkeiten einfacher und in einem Bruchteil der Zeit erlernen zu können (Abbildung 3.2). Zum Beispiel können Daten von einem Roboter, der gelernt hat, auf zwei Beinen zu stehen, dabei helfen, ihm das Laufen auf diesen zwei Beinen beizubringen. Auch mit diesem Ansatz wurden bereits vielversprechende Ergebnisse erzielt. Sollten sich die Versuche dieser Arbeit schwierig gestalten, so könnte es zukünftig interessant sein, einen solchen Ansatz zu erforschen, um die Bewältigung der komplexeren Aufgabenstellung im Vergleich zu [5] zu ermöglichen.

4 Konzeptionierung

4.1 Einschränkungen und Übertragungsprobleme

In der Vorgängerarbeit sollte der Roboter bislang 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 Ausgangs-

punkt 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.

4.2 Wahl der Simulationsumgebung

In der Vorgängerarbeit wurden bereits drei mögliche Simulationsumgebungen ausführlich gegenübergestellt [5, 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 [22, 23], 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 [24]. 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.

4.3 Geplante Realisierung

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

4.3.1 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 [25]. 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.

4.3.2 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 4.1 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 [26]. 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.

4.3.3 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.

4.3.4 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 Trai-

ningsepisode 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] *An Introduction to Machine Learning.* https://monkeylearn.com/machine-learning/. (Einsichtnahme: 26. 04. 2023).

[2] Sutton, R. S./ Barto, A. G. *Reinforcement Learning: An Introduction (second edition).* Hrsg. von Bach, F. The MIT Press, 2018.

[3] Arulkumaran, K. u. a. „Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6 (11/2017), S. 26–38. URL: https://doi.org/10.1109%2Fmsp.2017.2743240.

[4] *Unity Game Engine Guide: How to Get Started with the Most Popular Game Engine Out There.* https://www.freecodecamp.org/news/unity-game-engine-guide-how-to-get-started-with-the-most-popular-game-engine-out-there/. 2020. (Einsichtnahme: 30. 04. 2023).

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

[6] *Unity Personal.* https://unity.com/products/unity-personal. (Einsichtnahme: 30. 04. 2023).

[7] *Unity ML-Agents Toolkit.* https://unity-technologies.github.io/ml-agents/. (Einsichtnahme: 01. 05. 2023).

[8] *ML-Agents Overview.* https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/. (Einsichtnahme: 01. 05. 2023).

[9] Schulman, J. u. a. *Proximal Policy Optimization Algorithms.* 2017. arXiv: 1707.06347 [cs.LG].

[10] Schulman, J. u. a. *Proximal Policy Optimization.* https://openai.com/research/openai-baselines-ppo. 2017. (Einsichtnahme: 01. 05. 2023).

[11] Suran, A. *On-Policy v/s Off-Policy Learning.* https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f. 2020. (Einsichtnahme: 01. 05. 2023).

[12] Sagar, R. *On-Policy VS Off-Policy Reinforcement Learning.* https://analyticsindiamag.com/reinforcement-learning-policy/. 2020. (Einsichtnahme: 01. 05. 2023).

[13] *Training ML-Agents*. https://unity-technologies.github.io/ml-agents/Training-ML-Agents/. (Einsichtnahme: 01. 05. 2023).

[14] *Training Configuration File*. https://unity-technologies.github.io/ml-agents/Training-Configuration-File/. (Einsichtnahme: 01. 05. 2023).

[15] AurelianTactics. *Understanding PPO Plots in TensorBoard*. https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2. 2018. (Einsichtnahme: 29. 04. 2023).

[16] *Using TensorBoard to Observe Training*. https://github.com/Unity-Technologies/ml-agents/blob/28ec36a74384f37eea75753b50a01436931dcaca/docs/Using-Tensorboard.md, (Einsichtnahme: 29. 04. 2023).

[17] Kohn, T. *Vektorgeometrie*. https://tobiaskohn.ch/files/Vektorgeometrie.pdf. 2012. (Einsichtnahme: 03. 05. 2023).

[18] Simmering, J. u. a. „Adaptation of a Decentralized Controller to Curve Walking in a Hexapod Robot“. In: 01/2023, S. 264–275.

[19] Schilling, M. u. a. *Decentralized Deep Reinforcement Learning for a Distributed and Adaptive Locomotion Controller of a Hexapod Robot*. 2020. arXiv: 2005.11164 [cs.RO].

[20] Tsounis, V. u. a. *DeepGait: Planning and Control of Quadrupedal Gaits using Deep Reinforcement Learning*. 2020. arXiv: 1909.08399 [cs.RO].

[21] Smith, L. u. a. *Learning and Adapting Agile Locomotion Skills by Transferring Experience*. 2023. arXiv: 2304.09834 [cs.RO].

[22] *MuJoCo – Advanced Physics Simulation*. https://mujoco.org/. (Einsichtnahme: 22. 04. 2023).

[23] *GitHub - deepmind/mujoco: Multi-Joint dynamics with Contact. A general purpose physics simulator.* https://github.com/deepmind/mujoco. (Einsichtnahme: 22. 04. 2023).

[24] *Unity Plug-in - MuJoCo Documentation*. https://mujoco.readthedocs.io/en/latest/unity.html. (Einsichtnahme: 22. 04. 2023).

[25] *Announcing ML-Agents Unity Package v1.0! — Unity Blog*. https://blog.unity.com/technology/announcing-ml-agents-unity-package-v1-0. 2020. (Einsichtnahme: 22. 04. 2023).

[26] Technologies, U. *Example Learning Environments - Unity ML-Agents Toolkit.* https://unity-technologies.github.io/ml-agents/Learning-Environment-Examples/#crawler. (Einsichtnahme: 22. 04. 2023).