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 19. Mai 2023

Schiebelhut, Yannik

Abstract

- Deutsch -

Platzhalter

Abstract

- English -

Placeholder

Inhaltsverzeichnis

Abkürzungsverzeichnis

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

Abbildungsverzeichnis

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 oder sogar in Gebiete vorzudringen, die uns verwehrt bleiben. 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, spinnenartigen, 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. Anhand der Ergebnisse dieses Trainingsprozesses wird evaluiert, ob das entwickelte Konzept einen erfolgversprechenden Lösungsansatz darstellt und welche Verbesserungen vorgenommen werden sollten.

2. Grundlagen

Wie in Kapitel 1 bereits angedeutet, werden in dieser Arbeit Methoden aus dem Bereich der künstlichen Intelligenz verwendet, um das gestellte Problem zu lösen. Zum Verständnis der Ausarbeitung werden nachfolgend die wichtigsten Grundlagen erläutert. Weiterhin wird kurz auf die zur Simulation verwendete Software eingegangen. Im Rahmen der konzeptuellen Planung wird auch von mathematischen Grundlagen des dreidimensionalen Raums Gebrauch gemacht, welche deshalb ebenfalls kurz beleuchtet werden sollen.

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, die gelernten Daten generalisierend, 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

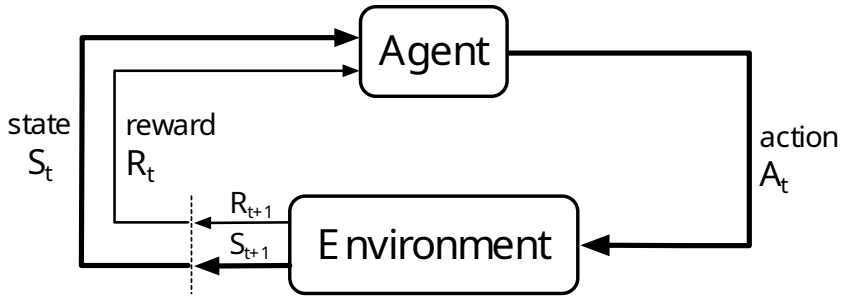

Abbildung 2.1.: Die Interaktion von Agent und Umgebung als MDP [2, S. 48]

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 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.

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 *Markov Decision Process (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.

Der wichtigste Bestandteil 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 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

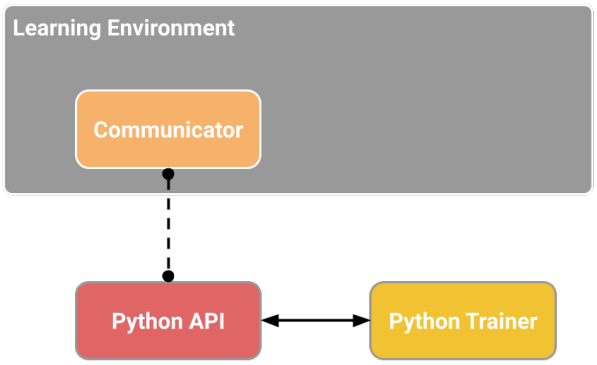

Abbildung 2.2.: Vereinfachtes Block-Diagramm des ML-Agents-Toolkits [8]

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.
- **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 *Cumulative 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 und keine starken Schwankungen aufweisen. 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 $\overrightarrow{AB}$

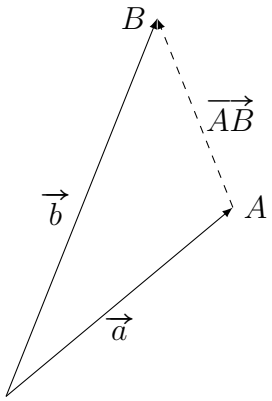

Abbildung 2.3.: Vektor von Punkt A zu Punkt B

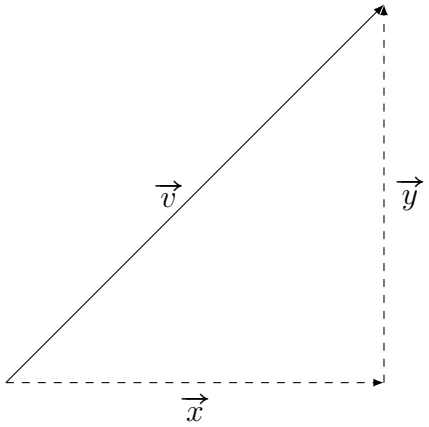

Abbildung 2.4.: Zerlegung des zweidimensionalen Vektors $\overrightarrow{v}$

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 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)).

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

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

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

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)).

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

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

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 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 [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 dauert und dabei außerdem die Gefahr droht, 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 Oszillatorsysteme 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

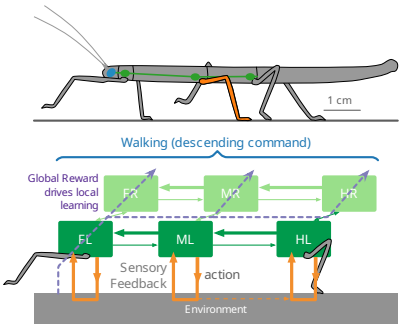

Abbildung 3.1.: Dezentrale Architektur in Anlehnung an Stabinsekten [19]

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,

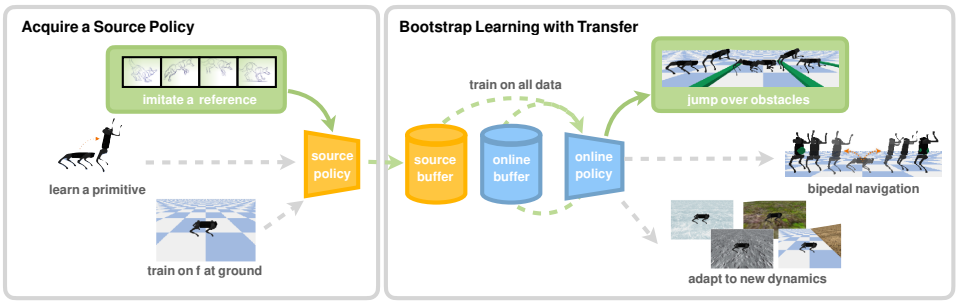

Abbildung 3.2.: Transfer von Trainingsdaten zur Bewältigung komplexer Probleme [21]

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 kann 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 kann, 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 muss für jeden Pfad, den der Roboter laufen soll, ein eigenständiges Modell trainiert werden, was einen praktischen Nutzen unmöglich macht – 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 kann diese Frage einfach mit „Nein“ beantwortet werden. Prinzipiell gesehen kann der Roboter seine aktuelle Position anhand seiner vergangenen Bewegungen vom Ausgangspunkt her berechnen. Andererseits setzt dies eine enorm hohe Präzision der

Bewegungen voraus. Außerdem kann der Roboter in der Realität auf dem Boden rutschen. Auch läuft das bisherige Modell nicht verlässlich geradeaus – nicht einmal in der Simulation –, 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 ist, 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 innerhalb 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 ist 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 ist 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 ist in der Realität der Einsatz komplexer Systeme nötigt. Die Integration solcher übersteigt jedoch den Umfang dieser Arbeit erheblich. Hier soll eher ein Proof-of-Concept für die selbstständige Umsteuerung von Hindernissen erarbeitet werden. Es soll daher 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, besteht außerdem das Problem, dass der Roboter im Sprung gegen diese knallen kann. Aus den genannten Gründen ist es sinnvoll, Maßnahmen zur Einschränkung der Bewegung des Roboters vorzunehmen. Möglich ist 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 ist. 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 wird, muss 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 kann 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 ist es denkbar, die Ergebnisse der verschiedenen Umgebungen zu vergleichen. Da jedoch keine Übertragung auf einen realen Roboter erfolgt, ist davon auszugehen, in diesem Kontext keinen spürbareren Unterschied zwischen den Physics-

Engines zu beobachten. Selbst falls ein messbarer Unterschied bestehen sollte, kann 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 wird die Umgebung auf den aktuellen Stand der Technik migriert, 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 können auch für die Pfadplanung des Roboters verwendet werden. Die für das Training notwendigen, zufälligen Pfade bilden sie bereits automatisch ab. Um dem trainierten Modell des Crawlers einen festen Pfad vorgeben zu können, muss lediglich eine kleine Modifikation der Dynamic Targets vorgenommen werden, um diese in einer festgelegten Reihenfolge anstatt zufällig erscheinen zu lassen. Dieses Prinzip soll auf diese Studienarbeit übertragen werden und es so dem Roboter ermöglichen, einem geplanten Pfad zu folgen.

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 kann etwa in der Reward-Funktion ein Faktor eingebracht werden, der den Roboter mit einem, an der Neigung der Zentralplatte skalierten, Wert bestraft.

Um den Roboter im Zuge der Pfadplanung 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 ist, die konkrete Bewegungsrichtung des Roboters zu bestimmen und ein Produkt mit der Geschwindigkeit des Roboters zu bilden. Diese Möglichkeit ermöglicht eine potenziell feinere Gewichtung der Faktoren. 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 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 wird voraussichtlich auch hier die Adaption der Reward-Funktion sein, 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 ist, bei jedem Simulationsschritt eine kleine Strafe zu vergeben, mit dem Ziel, den Roboter zu 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. Umsetzung

5.1. Rekonstruktion und Migration der Simulationsumgebung

Der Quellcode der alten Arbeit liegt auf GitHub[1] vor. Im ersten Schritt wird angestrebt, diesen Arbeitsstand zu rekonstruieren, sodass ein Training in der Unity-Umgebung möglich ist.

5.1.1. Programmversionen

Zur Projektbasis liegt neben der Ausarbeitung in [5] nur Quellcode vor. Leider werden dabei die verwendeten Programmversionen nicht dokumentiert, welche jedoch notwendigerweise fein aufeinander abgestimmt sein müssen, um ein Training zu ermöglichen und dessen Erfolg zu gewährleisten. Sowohl Unity als auch ML-Agents und das zugehörige Python-Toolkit wurden seit der Durchführung von [5] in unterschiedlicher Geschwindigkeit weiterentwickelt. Aufgrund von auftretenden Inkompatibilitäten empfiehlt es sich daher, zunächst die Originalversionen aufzusetzen und dies als Ausgangspunkt für weitere Anpassungen zu nutzen. Die Unity-Projektdatei enthält Informationen über die exakte Unity-Version, die zur Erstellung des Projekts verwendet wurde. Leider fehlt jedoch Dokumentation zur verwendeten Version von ML-Agents und des Python-Toolkits. Recherche in der Veröffentlichungshistorie von ML-Agents ergeben, das höchstwahrscheinlich Version 0.14.1 des Unity-Plugins verwendet wurde. Daraus ergeben sich Python-Abhängigkeiten, die darauf hindeuten, dass eine Python Version >= 3.6 und <3.7 verwendet wurde. Leider gestaltet sich der Versuch erfolglos, diese Versionen auf den verwendeten Entwicklungssystemen (OS X und Linux) zu installieren. Somit ist es auch nicht möglich, die exakte Entwicklungskonstellation von [5] zu rekonstruieren.

[1] https://github.com/MobMonRob/HindernisumfahrungRLStudien/tree/1c0a884c2133107562e4928bfa8bef2ee6e2ade0

Um die Ergebnisse dieser Arbeit möglichst nachhaltig zu machen, soll zur weiteren Entwicklung die aktuelle Version von ML-Agents verwendet werden (Unity-Paket in der Version 2.0.1, Stand: Mai 2023). Das korrespondierende Python-Plugin ist `mlagents` in der Version 0.30.0. (Zum Zeitpunkt des Schreibens ist es notwendig, das Paket `protobuf` in der Version 3.20.3 explizit zu installieren, da sonst die Installation von `mlagents` scheitert.) Die Dependencies dieses Pakets bedingen, dass als neuste Python-Version 3.10.8 verwendet werden kann, welche auch zur Entwicklung gewählt wird.

(Installation und Management spezifischer Patch-Versionen von Python kann kompliziert sein, da in der Regel ein Versionsmanagement nur anhand der Minor-Versionen vorgesehen ist. Im Rahmen dieser Arbeit hat sich der Einsatz des Programms `pyenv`[2] empfohlen, da sich damit sehr einfach spezifische Versionen von Python individuell kompilieren und managen lassen.)

Da es einerseits Problematiken bereiten kann, neue Plugins mit einer alten Version von Unity zu betreiben und zusätzlich die damals verwendete Version massive Fehler in Kombination mit OS X aufweist, wird auch Unity auf eine aktuelle Version angepasst. Dafür wird in diesem Fall die aktuellste LTS-Version zum Zeitpunkt der Entwicklung verwendet (2021.3.21.f1).

5.1.2. Änderungen der Codebasis

Da es sich bei ML-Agents um ein vergleichsweise neues Toolkit handelt, unterliegt es fortlaufend einer starken Entwicklung. Im Zeitraum seit der Vorgängerarbeit wurde das Plugin von einer Alphaversion zu einem offiziellen Release gebracht. In diesen Entwicklungsphasen kommt es bei Software häufig zu Breaking Changes. Auch bei ML-Agents ist dies der Fall und es kommt umgehend zum Kompilierungsfehlern, wenn das Unity-Projekt im Editor geöffnet wird. Der erste Schritt besteht deshalb darin, herauszufinden, welche Methoden davon betroffen sind. Dafür werden im Fehlerbericht die Methoden gesucht, die Fehler enthalten. Die Methodenköpfe können dann in die Versionshistorie von ML-Agents[3] gesucht werden. Im vorliegenden Fall sind somit alle an der Schnittstelle des Toolkit vorgenommenen Änderungen deutlich aufgeschlüsselt und geben Aufschluss darüber, wie die Kompatibilität des Quellcodes wiederhergestellt werden kann.

[2] https://github.com/pyenv/pyenv
[3] https://github.com/Unity-Technologies/ml-agents/releases

Für das Training wird in [5] eine Trainer-Config-YAML-Datei verwendet, wie sie in Unterabschnitt 2.2.3 beschrieben wird. Mit einer Aktualisierung des Python-Toolkits hat sich auch das Format dieser Datei verändert, weshalb die ursprüngliche Datei nicht mehr kompatibel zum nun verwendeten Tooling ist. Die Änderungen am Format sind vergleichsweise gering und die Datei von überschaubarer Größe, weshalb nach dem Vorbild der alten Datei und unter Anleitung von [14] die Datei neu gebaut werden kann. Der Codestand ist nach diesen Veränderungen[4] nun mit den aktualisierten Versionen des Toolings kompatibel und wird als Ausgangspunkt für die Experimente dieser Arbeit verwendet.

5.1.3. Durchführung des Trainings

Um ein Training in der Simulationsumgebung durchzuführen wird zuerst eine kompilierte Binary der Trainingsumgebung erstellt. Zwar ist es theoretisch möglich, direkt aus dem Unity-Editor das Training zu starten. Allerdings bringt dies einige Nachteile mit sich, wie etwa mangelnde Skalierbarkeit und Performanceverluste. Außerdem ist es so nicht möglich, das Training effizient auf einer unabhängigen Maschine durchzuführen, die eine weit höhere Trainingsrate ermöglicht. Um eine solche Binary zu erstellen, wird der Build Prozess in Unity unter File >Build Settings >Build gestartet. Dabei kann die gewünschte Zielplattform ausgewählt werden – in Abhängigkeit, wo das Training durchgeführt werden soll. Zum Cross-Compiling ist es allerdings vorausgesetzt, dass die entsprechenden Erweiterungen und Bibliotheken bei der Installation von Unity ausgewählt und geladen wurden. Sonst ist standardmäßig nur das Kompilieren für die Architektur und Plattform möglich, unter der der Editor ausgeführt wird.

Auf der Maschine, auf der das Training durchgeführt werden soll, muss dafür lediglich das Python-Toolkit installiert sein. Eine Installation von Unity ist dafür nicht erforderlich, was das Auslagern des Trainingsprozesses und somit ein effizientes Training deutlich vereinfacht. Für die Experimente im Rahmen dieser Arbeit wird ein virtuelles Rechencluster mit einer Intel Xeon (Gold 6240) CPU mit 8 Kernen, einer Nvidia vGPU V100D-8C und Ubuntu 20.04.1 LTS zum Einsatz. Deshalb wird das Environment für Linux x86_64 gebaut und es ist kein Cross-Compiling von der Entwicklungsmaschine zur Trainingsmaschine nötig. Uni-

[4] https://github.com/MobMonRob/HindernisumfahrungRLStudien/tree/ec8abbd161217c9a42adb42779e01c5b3dfeb209

ty produziert bei Kompilierung für Linux einen Ordner mit den Ausgabedateien, welche sowohl die eigentliche Binary als auch Artefakte und Libraries enthalten. Dieser Ordner sollte deshalb vollständig auf die Trainingsmaschine übertragen werden. Für die Durchführung des Trainingsprozesses können verschiedene Parameter von `mlagents-learn` angepasst werden. Es ist keine pauschale Angabe möglich, welche Parameter verlässlich und auf allen möglichen Trainingsmaschinen zu einem guten Ergebnis führen. Deshalb ist eine kleine Testreihe unerlässlich, bei der möglichst gute Werte für die Parameter ermittelt werden, um das spätere Training effizient zu gestalten. Der maßgebliche Performanceindikator ist hierbei die Anzahl der Trainingsschritte pro Sekunde. Zur Optimierung wird beispielsweise nach und nach die Anzahl der parallelen Trainingsumgebungen erhöht, bis die Anzahl der Schritte pro Sekunde nicht mehr weiter steigt. Die nachfolgende Befehlszeile scheint nach einigem Optimieren gute Ergebnisse für die Machienenkonfiguration zu liefern: `mlagents-learn --env binary-name/binary-name.x86_64 --run-id runName --no-graphics --torch-device cuda --num-envs 4 --time-scale 1 trainer.yaml`. Dabei werden 4 parallele Trainingsumgebungen (mit jeweils 30 Agenten) ohne Generierung von grafischen Artefakten betrieben. ML-Agents greift intern auf Tensorflow und PyTorch zurück. Grundsätzlich bieten beide Bibliotheken die Möglichkeit, eine CUDA-fähige GPU zur Grafikbeschleunigung der Trainingsalgorithmen zu verwenden. Im Kontext des verwendeten Trainingsalgorithmus (PPO) ist infrage zu stellen, ob ein ernstzunehmender Vorteil durch die Verwendung einer GPU erzielt werden kann. Ressourcen im Internet sind sich hierüber uneinig. Da es jedoch nicht zu einer Verschlechterung und bestenfalls zu einer Verbesserung kommen kann, und die GPU zur Verfügung stand, wird sie dem Trainingsprozess auch als Ressource zur Verfügung gestellt. Weiterhin wird die Wahl getroffen, die `time-scale` des Trainingsprozesses auf 1 zu setzen. Standardmäßig wird dieser Parameter von ML-Agents auf 20 gesetzt, was den Trainingsprozess beschleunigen soll. Vergleichende Tests ergeben jedoch, dass es im Kontext der verwendeten Trainingsumgebung und Trainingsmaschine keinen Unterschied für die Geschwindigkeit der Trainingsoperationen zu machen scheint, auf welchen Wert der Parameter gesetzt wird. Es gibt jedoch Hinweise, dass bestimmte Arten von physikalischen Berechnungen von der zeitlichen Skalierung beeinflusst werden [27]. Da unklar ist, ob solche Berechnungen hier Verwendung finden und gegebenenfalls sogar eine Ursache für die eigenartige Fortbewegungsart des Roboters im bisherigen Training sein könnten, wird für das Training, wie bereits erwähnt, eine nicht-verzerrte Zeitskala verwendet. (Da jedoch der Parameter die Schritte pro Sekunde nicht zu beeinflussen

scheint, ist zu hinterfragen, ob er in der verwendeten Version der Toolkits überhaupt korrekt interpretiert wird.)

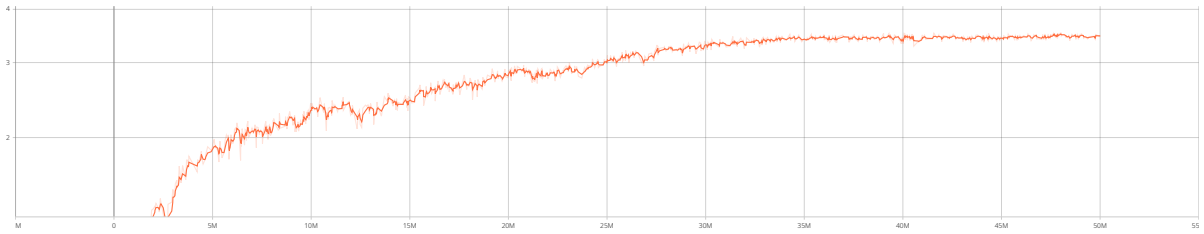

Abbildung 5.1.: Cumulative Reward der Rekonstruktion von [5]

Der Verlauf des Cumulative Rewards, der nun beim Training entsteht und in Abbildung 5.1 dargestellt wird, gleicht im Wesentlichen den Resultaten von [5, S. 50]. Auch der gelernte Bewegungsablauf, der bei der Inferenz zu beobachten ist, gleicht den Resultaten der vorangegangenen Arbeit, weshalb die Rekonstruktion als erfolgreich bewertet wird.

Video timeScale1

5.2. Stabilisierung des Laufverhaltens

Wie in Abschnitt 4.1 ausgeführt und von den Ergebnissen der Rekonstruktion zusätzlich veranschaulicht wird, bewegt sich der Roboter aktuell mit einer sprungartigen Bewegung fort. Diese ist, wie beschrieben, Ursache für viele potenzielle Probleme. Deshalb soll nun das Laufverhalten stabilisiert werden. In diesem Zusammenhang wird darunter verstanden, die Schwankungen, die die Körpermitte bei der Bewegung vollführt, auf ein Minimum zu begrenzen. Wird das Vorbild der Spinnen betrachtet, so ist davon auszugehen, dass diese Einschränkung kein Hindernis für eine effiziente Fortbewegung darstellt.

Es existieren innerhalb der Simulation zwei elementare C#-Skripte, die für die in diesem Kapitel beschriebenen Änderungen von besonderer Relevanz sind: `SpiderAgent.cs` und `SpiderController.cs`. In `SpiderAgent.cs` befinden sich die für den Reinforcement Learning Prozess relevanten Konfigurationen des Agenten. Dort werden Beobachtungen gesammelt und dem Lernalgorithmus zur Verfügung gestellt, die Aktionen vom Lernalgorithmus entgegengenommen, die Lern-Episoden verwaltet und vor allem der Reward zugewiesen. Der `SpiderController` ist wiederum einem `SpiderAgent` zugeordnet und verwaltet dessen Skripte für die Servomotoren, koordiniert die allgemeine Durchführung der Bewegung und führt das Zurücksetzen der Komponenten durch. Da

```
1  public float getReward() {
2      var addReward = 0f;
3      if (isTurned()) {
4          addReward = -0.1f;
5      }
6
7      var change = getProgress() - lastProg;
8      lastProg = getProgress();
9      if (change < 0) change = 0;
10     return change + addReward;
11 }
12
13 public bool isTurned() {
14     var up = center.forward;
15     var angle = Vector3.Angle(up, Vector3.up);
16     return angle > 90;
17 }
```

Listing 5.1: Ürsprüngliche Berechnung des Reward-Signals

ein `SpiderController` somit auf die direkten Werte des Roboterzustands zugreifen kann, wird hier auch der Reward berechnet, der in `SpiderAgent.cs` zugewiesen wird.

Für die Stabilisierung der Bewegung bestehen nun zwei verschiedene Ansätze, die ausprobiert und verglichen werden. Beide werden über Änderungen in `SpiderController.cs` realisiert. Die dort implementierte, allgemeine Bewegungskoordination des Roboters ist eine seiner physikalischen Grundeigenschaften und wird deshalb im Rahmen dieser Arbeit nicht manipuliert. Auch wird so eine kontinuierliche Basis für die einzelnen Versuche und deren Evaluation geboten. Jedoch stellt der in dieser Klasse berechnete Reward die zentralste Stellschraube des Reinforcement Learning Problems dar.

In Listing 5.1 ist die ursprüngliche Berechnung des Rewards zu sehen, wie sie in `SpiderController.cs` implementiert ist. Dabei wird zunächst der Reward mit 0 initialisiert. Anschließend wird abgeprüft, ob sich der Roboter um mehr als 90° zur horizontalen Achse gedreht hat und eine negative Belohnung vergeben, falls dies der Fall ist. Mit dieser Prüfung wird verhindert, dass der Roboter sich überschlägt und dabei seine empfindliche Elektronik beschädigt. Anschließend wird geprüft, wie weit der Roboter auf der Karte in Vorwärtsrichtung von seinem Ausgangspunkt entfernt ist und die Differenz zum vorherigen Resultat dieser Berechnung gebildet. Diese Differenz – sowohl wenn diese

```
1  public float getReward() {
2      var addReward = 0f;
3      if (isTurned()) {
4          addReward = -0.1f;
5      }
6      addReward += getAngle() * -0.01f;
7
8      var change = getProgress() - lastProg;
9      lastProg = getProgress();
10     if (change < 0) change = 0;
11     return change + addReward;
12 }
13
14 public float getAngle() {
15     var up = center.forward;
16     var angle = Vector3.Angle(up, Vector3.up);
17     return angle % 90;
18 }
```

Listing 5.2: Reward-Funktion mit Stabilisierung

positiv ausfällt, als auch, wenn sie negativ ist – wird dann mit der eventuellen Strafe für ein Überschlagen verrechnet und als Reward zurückgegeben.

Der offensichtliche und einfache Weg ist es nun, in der Methode `isTurned()` den Schwellwert der Rückgabe von 90° auf einen kleineren Wert, zum Beispiel 5°, zu reduzieren. Mit dieser Modifikation wird automatisch ein negativer Summand in den Reward eingebracht, wenn die Schwankung der Körpermitte den nun deutlich geringeren Schwellwert überschreitet. Das Problem hierbei besteht darin, dass die Methode `isTurned()` nicht nur zur Berechnung des Rewards verwendet wird, sondern auch in `SpiderAgent.cs` die aktuelle Episode terminiert wird, wenn sich der Roboter auf den Rücken dreht. Wird die beschriebene Modifikation an dieser Methode vorgenommen, so wird auch eine kleine Schwankung der Körpermitte bereits als Überschlag interpretiert und die laufende Trainings-Episode fälschlicherweise beendet. Wie sich bei einem Trainingsversuch zeigt, führt dies dazu, dass der Roboter kein Laufverhalten lernen kann, da ihm keine Gelegenheit geboten wird, seinen Fehler zu erkennen, zu erforschen und zu korrigieren. Für eine möglichst stabile Körpermitte sollte der Rotations-Schwellwert möglichst gering sein. Jedoch wird bei diesem Ansatz ein Training gegenläufig zum sinkenden Schwellwert praktisch unmöglich.

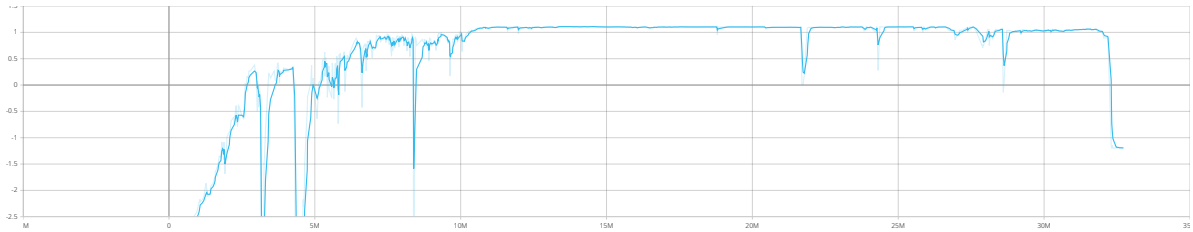

Abbildung 5.2.: Cumulative Reward mit Stabilisierung

Eine alternative Lösung ist in Listing 5.2 dargestellt. Zentrales Element dieser Lösung ist die neu eingeführte Methode `getAngle()`. Der Methodenrumpf ist fast identisch mit dem der Methode `isTurned()` aus Listing 5.1. Der Unterschied besteht dabei darin, dass die neue Methode nicht prüft, ob der berechnete Winkel größer als 90° ist und den resultierenden Wahrheitswert zurückgibt, sondern stattdessen den Rest der Division des Winkels durch 90 bestimmt und als Fließkommazahl zurückliefert. Anschließend wird der mit der neuen Methode berechnete Winkel mit einem kleinen negativen Faktor multipliziert und dieser Summand in der Berechnung des Rewards ergänzt (Zeile 6). Somit wird eine Strafe vergeben, deren Betrag proportional mit der Rotation der Körpermitte zunimmt. Die anschließende Belohnung für die zurückgelegte Distanz und die Verrechnung mit der Bestrafung bleibt unverändert.

Das mit dieser Reward-Funktion trainierte Modell legt die Wirksamkeit der Maßnahme dar: beim Laufverhalten des Roboters ist zu beobachten, dass die Körpermitte perfekt in einer horizontalen Lage gehalten wird. Abbildung 5.2 zeigt den Verlauf des Cumulative Rewards während des Trainings. Im Vergleich zu Abbildung 5.1 fällt auf, dass der Trainingsprozess zwar zu einem stabilen Wert konvergiert, jedoch besonders am Anfang deutlich instabiler verläuft. Außerdem ist der Wert, zu dem das Training konvergiert, fast um zwei Drittel gemindert. Betrachtet man das Laufverhalten des Roboters, so fällt weiterhin auf, dass der Roboter weiterhin eine sprungartige Fortbewegung wählt. (Diese fällt allerdings weitaus stabiler aus, als der in [5] erlernte Bewegungsablauf.) Dabei werden allerdings nur die beiden vorderen und eins der hinteren Beine verwendet, das letzte Bein ist eingekringelt und berührt den Boden nicht.

Video adjusted Angles

```
1  public float getReward() {
2      if (isTurned()) {
3          return -10.0f;
4      }
5
6      var reward = -0.0025f;
7      reward += getAngle() * -0.0025f;
8
9      var change = getProgress() - lastProg;
10     lastProg = getProgress();
11     return change + reward;
12 }
```

Listing 5.3: Optimierte Reward-Funktion

5.2.1. Optimierung der Reward-Funktion

In einem Versuch, das Laufverhalten weiter zu optimieren und eine natürlichere Laufbewegung zu erhalten, wird die Berechnung des Rewards leicht umgestaltet. Die optimierte Version ist in Listing 5.3 abgebildet. Die Prüfung, ob der Roboter sich überschlagen hat, bleibt bestehen, wobei hier die Bestrafung erhöht wird, um den Überschlag als schlimmstmöglichen Fall vom instabilen Laufverhalten abzugrenzen. Tritt dieser Fall ein, so findet auch keine weitere Berechnung des Rewards statt, sondern der Wert der Strafe wird umgehend zurückgegeben. Danach wird eine kleine Strafe eingeführt, die mit jedem Schritt vergeben wird. Diese stellt eine generelle Optimierung der Reward-Funktion dar, da eine solche Strafe bewirkt, dass ein Agent danach strebt, sein Ziel möglichst schnell zu erreichen [28].

Durch die starke Modifikation der Gewichte in der Reward-Funktion können die Beträge der Cumulative Reward Funktion während des Trainings nicht mehr als vergleichende Metrik eingesetzt werden. Der in Abbildung 5.3 dargestellte Kurvenverlauf zeigt jedoch, dass der Trainingsprozess bedeutend stabiler verläuft verglichen mit Abbildung 5.2. Das in Folge der vorgenommenen Maßnahmen trainierte Modell zeigt ein erstaunlich natürliches Bewegungsmuster, sehr ähnlich zur Fortbewegung eines Spinnentiers.

Video 4-debug-extended-vector-size

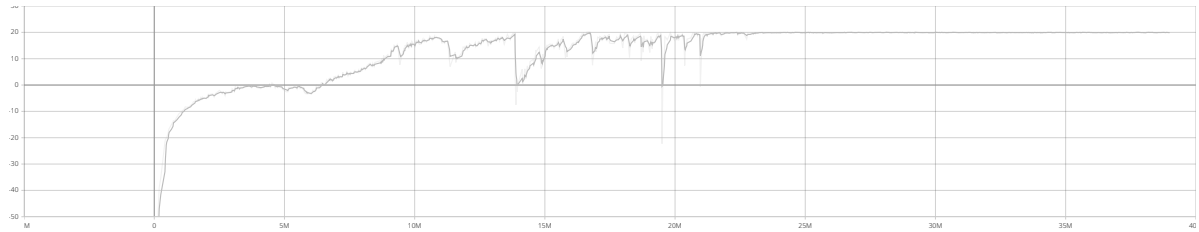

Abbildung 5.3.: Cumulative Reward mit Optimierung der Reward-Funktion

```
1 public void MoveTargetToRandomPosition() {
2     var newTargetPos = m_startingPos + ↙
        ↳ (Random.insideUnitSphere * spawnRadius);
3     newTargetPos.y = m_startingPos.y;
4     transform.position = newTargetPos;
5 }
```

Listing 5.4: Respawnen des DynamicTargets

5.3. Pfadplanung

Der nächste Implementierungsschritt ist die Ergänzung von Pfadplanung. Hierfür wird zunächst ein Proof of Concept erstellt und dieses im Anschluss auf den Roboter übertragen.

5.3.1. Proof of Concept: Crawler

Wie in Abschnitt 4.3 erwähnt, bietet das *Crawler-Example* des ML-Agents-Toolkits eine interessante Referenz für die Pfadplanung. Im Beispiel wurde ein Agent darauf trainiert, nacheinander zu zufällig erscheinenden Zielen zu laufen. Wenn der Agent ein Ziel (*DynamicTarget*) erreicht, so verschwindet es und taucht an anderer Stelle wieder auf. Der Agent hat dabei keinen Einfluss auf das DynamicTarget und kennt nur dessen aktuelle Position.

Wenn ein DynamicTarget initialisiert wird oder in Kontakt mit einem Unity-GameObject kommt, welches mit dem Tag „agent“ versehen ist, wird die in Listing 5.4 dargestellte Methode ausgeführt, welche das DynamicTarget an einen zufälligen Ort teleportiert.

Mittels der Modifikation aus Listing 5.5 ist es möglich, Einfluss auf die Reihenfolge zu nehmen, in der die DynamicTargets erscheinen. Die Kontrollstruktur basiert auf

```
1  private int i = 0;
2  public List<Vector3> positions;
3
4  public void MoveTargetToRandomPosition() {
5      if (i < positions.Count) {
6          transform.position = positions[i++];
7      } else {
8          var newTargetPos = m_startingPos + ↙
               ↳ (Random.insideUnitSphere * spawnRadius);
9          newTargetPos.y = m_startingPos.y;
10         transform.position = newTargetPos;
11     }
12 }
```

Listing 5.5: Modifiziertes Respawnen des DynamicTargets

einem Zähler und einer Liste von Positionen (Zeile 1f). Solange der Zähler mit einem Listenindex korrespondiert (Zeile 5), wird als neue Position des Ziels die in der Liste hinterlegte Position verwendet. Wurde die Liste vollständig abgearbeitet, erscheinen die Ziele wieder zufällig. In der Element-Detailansicht von Unity können anschließend beliebig viele Zielpunkte frei eingegeben werden (Abbildung 5.4).

Da der Agent im Crawler-Example die Ziele nicht selbst kontrolliert, ist kein erneutes Trainieren des Modells nötig, sondern die Modifikationen sind direkt wirksam und der Agent läuft entlang des vorgegebenen Pfades.

Video crawler-poc

5.3.2. Übertragung auf den SpiderAgent

Um dieses Prinzip für den Roboter in dieser Studienarbeit zu nutzen, werden der `TargetController` und das dazugehörige Prefab aus dem Beispielprojekt des ML-Agents-Toolkits in die Simulationsumgebung kopiert. `SpiderAgent.cs` wird im Anschluss um den in Listing 5.6 dargestellten Code ergänzt. Dieser stellt eine Verknüpfung zwischen dem Agent und dem Target her und ermöglicht das initiale Spawnen des Ziels. Weiterhin wird sämtlichen Körperteilen des Roboters der Tag „agent“ zugewiesen. Für das Training wird die Positionsliste leer gelassen, um die Targets zufällig auf der Trainingsplattform erscheinen zu lassen, damit der Roboter nicht einen vorgegebenen Pfad auswendig lernt.

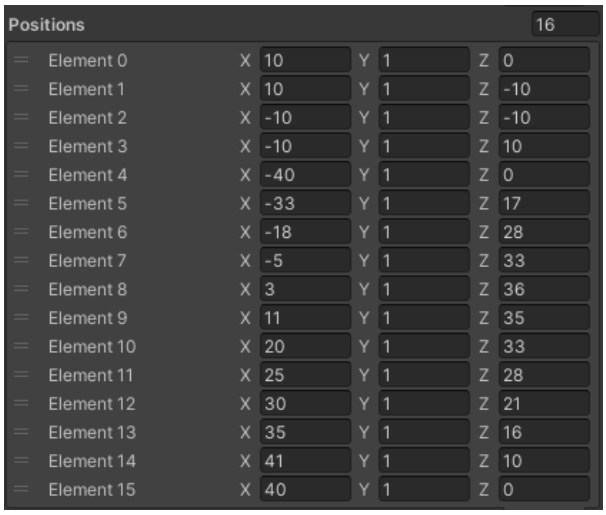

Abbildung 5.4.: Eingabefeld der Positionen

```
1  public Transform TargetPrefab;
2  private Transform m_Target;
3
4  public override void Initialize() {
5      SpawnTarget(TargetPrefab, transform.position);
6  }
7
8  private void SpawnTarget(Transform prefab, Vector3 pos) {
9      m_Target = Instantiate(prefab, pos, Quaternion.identity, ↙
           ↳ transform.parent);
10 }
```

Listing 5.6: Ergänzung für Spawnen des Targets (SpiderAgent.cs)

```
1  public override void CollectObservations(VectorSensor sensor) {
2      for (int i = 0; i < 12; i++) {
3          sensor.AddObservation(
4              normalize(controller.allServos[i].currentAngle));
5      }
6
7      var robotPosition = controller.getCenterPosition();
8      sensor.AddObservation(robotPosition);
9
10     var targetPosition = m_Target.transform.position;
11     sensor.AddObservation(targetPosition);
12 }
```

Listing 5.7: Erweiterung des Observation Space (SpiderAgent.cs)

Bislang verfügt der Roboter nur über die aktuellen Winkel der Servomotoren als Observations. Für das weitere Training sollen ihm nun zusätzlich seine eigene Position und die Position des Ziels zugänglich gemacht werden, da es für den Algorithmus andernfalls nicht möglich ist, seine Aktionen auf das Erreichen des Ziels auszurichten. Alternativ könnte dem Roboter ein Richtungsvektor übergeben werden, der vom Roboter auf das Ziel zeigt. Hinsichtlich ihres Informationsgehalts sind beide Angaben identisch, da sie rechnerisch ineinander überführt werden können. Da bei der manuellen Berechnung des Richtungsvektors jedoch leichter Fehler auftreten können, werden absolute Positionen übergeben. Außerdem wird dadurch zukünftig das Verbinden eines beliebigen Ortungssystems potenziell einfacher gehalten.

Als Voraussetzung, um die zusätzlichen Informationen zur Verfügung zu stellen, muss in Unity die Größe des Observation Space um sechs Elemente erhöht werden, da beide zu übergebenden Positionen jeweils von einem dreidimensionalen Vektor repräsentiert werden. Anschließend wird die in Listing 5.7 dargestellte Methode zum Sammeln der Observations modifiziert. Vor der Modifikation bestand die Methode lediglich aus der Schleife, die die einzelnen Werte der Servomotoren ausliest und in den VectorSensor eingibt. Diese Informationen werden um die beiden zusätzlichen Positionen ergänzt. Die gesamte Anzahl aller Einträge, die in dieser Methode zugewiesen werden muss exakt der in Unity hinterlegten Größe des Observation Space entsprechen, ansonsten erzeugt die Methode Fehler.

5.3.3. Geometrische Reward-Funktion

Um den Roboter nun dazu zu animieren, nicht mehr geradeaus, sondern zu den Zielen zu laufen, wird die Reward-Funktion angepasst. Wie in Abschnitt 4.3 beschrieben, existieren dafür verschiedene Ansätze. Als der vielversprechendste erscheint dabei die Verwendung einer sogenannten geometrischen Funktion. Kennzeichnend für diese ist, dass die einzelnen Bestandteile des Rewards nicht wie bislang aufsummiert werden, sondern ein Produkt aller Faktoren gebildet wird. Damit soll vermieden werden, dass sich der Agent darauf konzentriert, den einfachsten Reward zu maximieren und damit etwa in einem lokalen Maximum stecken zu bleiben. Um das Produkt zu maximieren, müssen zwangsläufig alle einzelnen Faktoren maximiert werden.

Es werden Trainings für drei solcher Funktionen durchgeführt. Die Funktionen haben verschiedene Komplexitäten und Ansätze, um eine allgemeine Eignung einer solchen Reward-Funktion abzuschätzen.

- Für den ersten Ansatz wird die Distanz gemessen, die der Roboter seit dem letzten Schritt zurückgelegt hat. Ausschlaggebend für die Messung ist dabei nur die Körpermitte. Als zusätzlicher Faktor wird der Winkel zwischen der „Blickrichtung“ der Körpermitte und der relativen Richtung, in der das Ziel liegt, gebildet und auf einen Bereich von 180° normalisiert.
- Die zweite Variation verwendet die in Unity-GameObjects hinterlegte Geschwindigkeit der einzelnen Körperteile und bildet daraus den Durchschnitt. Dieses Vorgehen soll plötzliche und ruckartige Bewegungen des Roboters, die beim Training auftreten, ausgleichen. Der nächste Faktor ist, wie bei der ersten Variante, ein normalisierter Wert an dem abgelesen werden kann, ob die Bewegung in Richtung Ziel erfolgte. Weiterhin wird hier wieder eine Belohnung eingeführt, die umso höher ausfällt, desto stabiler die Körpermitte in der Horizontalen gehalten wird.
- Der letzte Ansatz ist sehr ähnlich zum zweiten. Allerdings wird hier auf das berechnete Produkt noch eine konstante Strafe für jeden Schritt summiert, um eine schnelle Zielerreichung voranzutreiben.

Trotz der Unterschiede der Ansätze zeigen alle ein ähnliches Ergebnis des Trainings: der Roboter fällt mit seiner Körpermitte auf den Boden und bewegt seine Beine in der Luft.

Der Cumulative Reward ist dabei relativ konstant, das Modell lernt auch nach mehreren Millionen Simulationsschritten nicht.

5.3.4. Klassischer Reward

Aus diesen Gründen wird die Entscheidung getroffen, die Reward-Funktion neu zu entwerfen und dabei möglichst wenig konzeptionelle Änderungen ausgehend vom stabilisierten Laufverhalten vorzunehmen. Ziel dieser Vorgehensweise ist es, die Quellen potenzieller Fehler möglichst weit einzuschränken. Es wird grundlegend dieselbe optimierte Reward-Funktion wie in Listing 5.3 genutzt. Der Unterschied besteht hierbei jedoch darin, dass nicht mehr die zurückgelegte Distanz entlang der x-Achse gemessen wird, sondern die zurückgelegte Distanz in Richtung des Ziels. Dafür wird die jeweils letzte Position des Roboters zwischengespeichert. Es werden dann die Distanzen von der letzten Position zur Zielposition sowie von der aktuellen Position zur Zielposition gebildet. Anschließend wird die Differenz der beiden Distanzen so bestimmt, dass die Berechnung in einer positiven Belohnung resultiert, wenn der Roboter sich dem Ziel genähert hat.

Dieser Ansatz zeigt bessere Ergebnisse als die mit geometrischer Reward-Funktion, jedoch wird im Training sehr früh ein lokales Maximum erreicht. Dabei lernt der Roboter, dass er eine Belohnung erzielt, wenn er sich in Richtung des Ziels rollen lässt, überschlägt sich dabei allerdings und startet somit die Trainingsepisode neu. Versuche, die Strafen für Überschlagen und Körperrotation anzuheben, schlagen ebenfalls fehl. Wird diese Maßnahme ergriffen, verschlechtern sich die Resultate sogar. Anstatt sich in Richtung des Ziels zu rollen, rollt sich der Roboter auf dem Boden ein und vollführt nur minimale Bewegungen.

5.3.5. Optimierung der Hyperparameter

Aufgrund der geringen Abweichung der aktuellen Reward-Funktion von Listing 5.3 und des simplen Aufbaus der Funktion wird ein Logikfehler in der Funktion als Ursache ausgeschlossen. Zur Fehlersuche werden erneut Testreihen ausgehend vom Stand des stabilisierten Laufverhaltens durchgeführt und dabei die Änderungsschritte zusätzlich verkleinert. Dieser letzte Stand enthält als Observationen nur die Winkel der zwölf Servomotoren. Im ersten Schritt wird nun die Größe des Observation Space erneut, wie

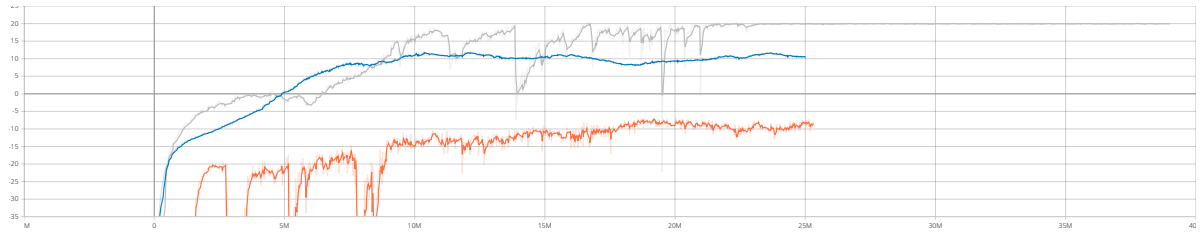

Abbildung 5.5.: Schrittweise Erweiterung und Optimierung der Hyperparameter: (grau) hinzugefügte Nullvektoren; (orange) zusätzliche Beobachtungen bereitgestellt; (blau) optimierte Hyperparameter

oben beschrieben, erweitert. Dabei werden allerdings nicht die Position des Roboters und die des Ziels hinzugefügt, sondern zwei dreidimensionale Nullvektoren als Platzhalter. Wie erwartet ist der Cumulative Reward des Trainingsdurchlaufs praktisch deckungsgleich mit dem Stand vor der Veränderung. Der nächste Schritt fügt nun bei weiterhin gleichbleibender Reward-Funktion die beiden Positionen als Beobachtung anstelle der Nullvektoren hinzu. Auch wenn zusätzliche Informationen, die allerdings nicht in den Reward einfließen, den Trainingsprozess eigentlich nicht negativ beeinflussen dürften, erreicht der Trainingsprozess mit Zugriff auf die Positionen keinen positiven Cumulative Reward.

Die wahrscheinlichste Ursache für ein solches Verhalten sind suboptimal eingestellte Hyperparameter. Zusätzlich scheint die Anzahl der Neuronen des verwendeten Neuralen Netzes für die gesteigerte Eingabekomplexität nicht mehr auszureichen. Aufgrund der Ähnlichkeit des Problems mit dem Crawler-Example, werden die Hyperparameter beider Probleme miteinander abgeglichen und optimiert zusammengeführt. Abbildung 5.5 zeigt eine Übersicht der Cumulative Rewards der drei Testläufe. Die optimierten Hyperparameter werden in Abschnitt B dargestellt.

Auf Basis der neu eingestellten Hyperparameter werden erneut die Änderungen von Unterabschnitt 5.3.4 angewandt. Als Resultat kann zwar eine stabil ansteigende Kennkurve des Cumulative Reward beobachtet werden, jedoch trifft dies nicht auf das gelernte Bewegungsverhalten zu. Der Roboter vollführt eine Mischung aus Sprüngen und Schritten, die ihn teilweise zielstrebig auf das Ziel zubewegen, in anderen Situationen jedoch vollkommen zufällig erscheinen. In der Dokumentation von ML-Agents wird als Maßnahme für einen stabilen Trainingsprozess empfohlen, den Betrag des Rewards möglichst auf einen Betrag von 1 zu skalieren. Einige Versuche mit verschiedenen Skalierungen zeigen jedoch

video 14a

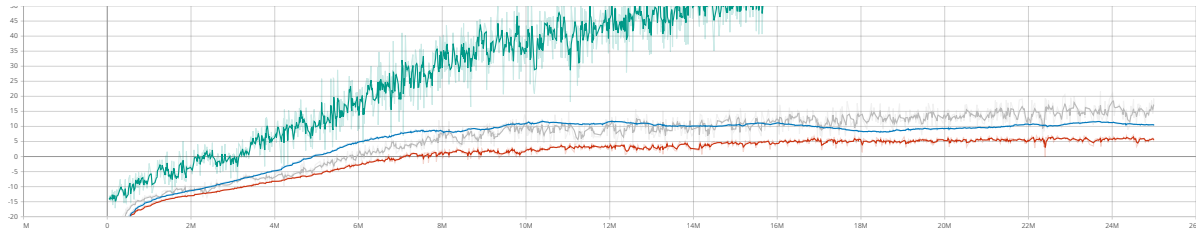

Abbildung 5.6.: Vergleich verschiedener Skalierungen des Rewards: (blau) Vergleichskurve geradeaus laufen; (rot) Belohnung für Bewegung in Richtung des Ziels; (grün) skalierter Reward; (grau) skalierter Reward mit höheren Strafen für Schieflage der Körpermitte

keine abweichenden Ergebnisse. Auffällig ist jedoch, dass die Cumulative Rewards der Trainingsdurchläufe mit skalierten Rewards deutlich stärker schwanken (Abbildung 5.6).

5.4. Hindernisumfahrung

Da Hindernisse nur im Kontext eines vorgesehenen Pfades sinnvoll umsteuert werden können, ist eine funktionierende Pfadplanung eine notwendige Voraussetzung für ein erfolgreiches Training eines Modells, dass den Roboter Hindernisse umsteuern lässt. Da die im vorherigen Kapitel trainierte Pfadverfolgung keine verwendbare Basis darstellt, werden im Folgenden ein paar grundlegende Aspekte thematisiert und demonstriert.

Um zu sehen, wie der Roboter auf einem einfachen Pfad auf ein statisches Hindernis reagiert, werden ein paar Experimente auf Basis des laufstabilisierten Modells durchgeführt. Als Pfad ist hier das Laufen entlang der x-Achse zu betrachten, wobei anzumerken ist, dass es keinen Faktor gibt, der eine seitliche Abweichung entlang der z-Achse korrigieren würde. Stellt man dem normal trainierten Modell ein Hindernis orthogonal zu seiner Laufrichtung in den Weg, so kann man beobachten, dass der Roboter bis an das Hindernis läuft und auch nach der Kollision seine Gangbewegung noch unverändert ausführt, dabei seine Beine allerdings nur auf der Stelle über den Boden schiebt. Von solch einem Ergebnis ist ohne explizites Training auch auszugehen, da der Roboter nie lernt, zur Seite zu gehen.

Wird nun allerdings mit unveränderter Reward-Funktion und zusätzlich mit dem eingefügten Hindernis trainiert, kann ein interessantes Ergebnis beobachtet werden. Wie in Abschnitt 4.3 schon ausgeführt, lernt der Roboter seine Umgebung auswendig, wenn sich diese nicht verändert. Dies ist auch hierbei zu beobachten. Was allerdings verwundert, ist,

dass der Roboter schon zu Beginn anfängt, eine Kurve zu laufen, mit der er knapp am Hindernis vorbeikommt – jedoch wird diese Kurve nicht abgebrochen, wenn der Roboter am Hindernis vorbeilaufen kann, obwohl er dadurch ausschließlich negative Rewards bekommen kann. Dies widerspricht eigentlich einer der grundlegendsten Maxime von Reinforcement Learning: den Reward immer und um jeden Preis zu maximieren.

Im Falle des Crawler-Examples, anhand dessen ein Proof of Concept zur Pfadplanung implementiert wurde, kann ähnliches beobachtet werden wie bei dem Modell des SpiderAgents, welches nicht explizit für das Hindernis trainiert wurde. Aufgrund seiner physikalischen Simulationseigenschaften federt der Crawler etwas von der Wand zurück, gegen die er wiederholt läuft. Dabei wird er jedes Mal leicht zur Seite abgedrängt, wodurch er nach einiger Zeit das Hindernis überwindet. Dieser Vorgang ist jedoch weder eine gezielte Handlung noch ansatzweise effizient.

Video crawler wand

Diese Experimente zeigen anschaulich, dass auch bei bereits funktionierender Pfadplanung in jedem Fall ein gesondertes Training mit einem zufällig generierten Hindernisparcour notwendig ist, um dem fertigen Modell ein Umsteuern von Hindernissen zu ermöglichen. Für den Fall, dass das Erlernen von Pfadplanung und das gleichzeitige Umsteuern von Hindernissen ein zu komplexes Lernproblem darstellt, könnte gegebenenfalls Curriculum Learning eingesetzt werden, um die Probleme aufeinander aufbauend zu lösen. Insgesamt dürfte eine Hindernisumsteuerung langfristig nur sinnvoll sein, wenn sie mit Sensorik unterstützt wird. Da dies das Problem und dessen Komplexität deutlich verändert, wird dann allerdings ohnehin ein gesondertes Training notwendig sein.

6. Auswertung der Ergebnisse

Insgesamt wurden einige Testreihen zur Stabilisierung des Laufverhaltens, zur Pfadplanung sowie zur Umsteuerung von Hindernissen durchgeführt. Während die Stabilisierung erfolgreich durchgeführt wurde, zeigen insbesondere die Testreihen zur Pfadverfolgung, auf denen die Hindernisumsteuerung basiert, nicht die erwünschten Ergebnisse. Zwar steuert der Roboter manchmal zielstrebig auf das DynamicTarget zu, allerdings ist sein Laufverhalten dabei stark instabil und die Aufgabe wird nicht verlässlich erfüllt. Da die dabei verwendete Reward-Funktion und somit das gestellte Reinforcement Learning Problem fundamental sehr ähnlich zur Projektbasis [5] aufgebaut sind, ist die Ursache nur schwer zu bestimmen.

Abbildung 6.1 stellt eine Übersicht der in Unterabschnitt 2.2.4 beschriebenen Bewertungskriterien für die einzelnen Trainingsdurchläufe dar. Hierbei existieren einige Auffälligkeiten. Die Kurve des Policy Loss sollte eigentlich im Laufe des Trainings abnehmen, wenn die Policy zu einer stabilen Funktion konvergiert. Abbildung 6.1 zeigt ein breites Spektrum, über das die verschiedenen Policy Losses verteilt liegen. Auffällig ist dabei jedoch, dass die Losses zwar in der Regel einer starken Schwankung unterliegen, diese sich jedoch bis auf wenige Ausnahmen auf einen recht engen Wertebereich für jede Kurve beschränken. Im Falle des Value Losses sollte der Wert zu Beginn hoch liegen und dann im Laufe des Trainings zu einem niedrigen Wert konvergieren. Auch bei diesen Kurven können bei den meisten Trainingsdurchläufen hohe Schwankungen anstatt der gewünschten Konvergenz beobachtet werden. Zwar gibt es auch einige Kurven, die grob dem erwarteten Verlauf entsprechen, jedoch konvergieren diese im Verhältnis zur Länge des Trainingsdurchlaufs viel zu früh.

Sowohl beim Policy als auch beim Value Loss kann beobachtet werden, dass einige Kurven sehr stabile und niedrige Werte aufweisen. Diese Kurven entstanden nach der Korrektur der Hyperparameter, weshalb auf eine potenzielle Überkorrektur geschlossen werden kann.

Zwar würden fehlerhaft optimierte Hyperparameter und die Verläufe der Losses theoretisch die schlechten Trainingsergebnisse erklären, jedoch muss in diesem Zusammenhang auch

Abbildung 6.1.: Policy Losses (oben) und Value Losses (unten) der Trainingsdurchläufe

darauf hingewiesen werden, dass die Stabilisierung des Laufverhaltens äußerst erfolgreich trainiert wurde, allerdings ebenso Verläufe der Losses zeigt, die eigentlich nicht zu einem erfolgreichen Training passen. Dadurch wird die oben geübte Kritik infrage gestellt.

6.1. Future Work

Als Future Work gilt es, die Fehlerquellen der aktuellen Problemstellung und Simulationsumgebung zu finden, um das implementierte Training erfolgreich und mit stabilem Ergebnis durchführen zu können. Dafür können verschiedene Konfigurationen von Hyperparametern versucht werden, welche das Problem potenziell verringern oder lösen könnten. Weiterhin dürfte es lohnenswert sein, ein Augenmerk auf die Unterschiede und Parallelen der vorliegenden Problemstellung zum wiederholt erwähnten Crawler-Example zu legen. Allerdings dürfen auch fundamentale Unterschiede der beiden Szenarios nicht außer Acht gelassen werden, die eine direkte Übertragung unmöglich machen. Primär sind dies die unterschiedliche Anzahl und Art der Gelenke genauso wie die zur Verfügung gestellten Eingabedaten. Auch Fehler in der Simulation der Servomotoren können als Ursache nicht ausgeschlossen werden, da diese etwa das zielgenaue Koordinieren der Gelenke erschweren könnte und somit zwar zur Bewältigung einzelner Aufgaben genügt, jedoch der komplexer werdenden Problemstellung nicht mehr gewachsen ist.

Sollten mit den aufgelisteten Maßnahmen kein Erfolg erreicht werden, so könnte es auch interessant sein, den dezentralen Ansatz von Schilling et al. [19] dediziert im Kontext der vorliegenden Problemstellung zu erforschen.

Ein elementarer Unterschied, der diese Arbeit von anderen abgrenzt, die dem State of the Art entsprechend, sind die stark beschränkten Eingabedaten. Den Trainingsalgorithmen in vergleichbaren Projekten stehen bedeutend mehr Daten zu sich selbst und ihrer Umgebung zur Verfügung. Deshalb wird der Schluss gezogen, dass wahrscheinlich sämtliche oben genannten Maßnahmen nur eine eingeschränkte Wirksamkeit entfalten können, solange die Sensorik des Roboters nicht erweitert wird. Deshalb sollte zukünftig weiterhin evaluiert werden, mit welchen Sensoren der Roboter in der Realität sinnvoll ergänzt werden könnte, um dem Roboter hilfreiche Informationen über seinen eigenen Zustand zur Verfügung zu stellen.

7. Fazit

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).

[27] Zhang, J. *Ultimate Volleyball: A multi-agent reinforcement learning environment built using Unity ML-Agents.* https://towardsdatascience.com/ultimate-volleyball-a-3d-volleyball-environment-built-using-unity-ml-agents-c9d3213f3064. 2021. (Einsichtnahme: 16. 05. 2023).

[28] *Agents.* https://unity-technologies.github.io/ml-agents/Learning-Environment-Design-Agents/#rewards. (Einsichtnahme: 18. 05. 2023).

A. Hinweis

Links einfügen und Text fomulieren

- Quellcode Projekt
- Ergebnisse Repo
- Videos
- Hinweis: zum Durchführung der Inferenz eines bestimmten Standes muss das Repository in diesem Commit ausgecheckt sein, die Observations müssen mit denselben Umgebungsdaten bestückt werden, wie während des Trainingsprozesses

B. Trainer Config YAML

titel

```yaml
behaviors:
  SpiderBrain:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 3.0e-4
      beta: 5.0e-3
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 3
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        strength: 1.0
        gamma: 0.995
    max_steps: 25.0e6
    time_horizon: 1000
    summary_freq: 10000
```