TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Fakultät für Mathematik und Informatik
Institut für Informatik
Professur für Virtuelle Realität und Multimedia

Masterarbeit

Konzeption und Implementierung eines Einkaufroboters in einem simulierten Supermarkt auf Basis von Reinforcement Learning

Marc Hoffmann

Master Angewandte Informatik
Matrikel: 61 090

14. März 2024

Betreuer/1. Korrektor:
Prof. Dr. Bernhard Jung

2. Korrektor:
Dr. Stefan Reitmann

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Versicherung bezieht sich auch auf die bildlichen Darstellungen.

14. März 2024 Marc Hoffmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Während die Übernahme des Einkaufes für manche eine reine Zeitersparnis wäre, löst es für körperlich eingeschränkte Personen ein bedeutendes Problem. Das Transportieren eines schweren Wocheneinkaufs stellt für ältere und eingeschränkte Personen, die häufig keinen Zugang zu Autos haben eine Herausforderung dar. Die Übernahme des Einkaufes könnte auch diesen Aspekt vereinfachen, da der Transport des Einkaufes durch Drittunternehmen wie beispielweise Flaschenpost ermöglicht wird, ohne ein eingeschränktes Angebot in Kauf nehmen zu müssen.
Jedoch ist für viele der Einkauf ein üblicher Bestandteil des Lebens, weswegen hier kein Raum geschaffen werden kann, in dem Mensch und automatisierte Roboter voneinander getrennt agieren können. Ebenso ist eine Trennung nicht im Interesse der Supermärkte, da dies zu weniger impulsiven Käufen und somit zu weniger Gewinnen führen würde. Deswegen ist ein Kontakt des Einkaufsroboters mit Menschen unausweichlich.
Andere Universitäten haben hier ebenso ein großes Potential erkannt. Mit dem Projekt I-RobEka verfolgt die TU Chemnitz einen Ansatz zur Implementierung eines voll umfänglichen Einkaufsroboters. Ein anderer Zweig in dem Forschungsbereich sind unterstützende Einkaufsroboter, deren Aufgaben weniger weitreichend sind. Meist befassen sich diese mit dem Tragen des Einkaufs oder sie geben Einkaufenden Wegbeschreibungen, damit diese die Produkte schneller finden können. Mehr zu dem derzeitigen Forschungsstand im Bereich Reinforcement Learning und Robotik im Einkauf berichte ich im Kapitel 2.
Das Ziel dieser Arbeit ist herauszufinden, ob mithilfe von Reinforcement Learning ein lernender Agent geschaffen werden kann, der dazu in der Lage ist in einem simulierten Supermarkt einen Einkauf zu übernehmen. Hierfür werden unterschiedliche Szenarien getestet und analysiert. Gesichtspunkte sind Realisierbarkeit in der Realität sowie Dauer des Einkaufs und die Fähigkeit Hindernissen auszuweichen.
Einkaufsroboter müssen dazu in der Lage sein, mit den sich verändernden Verhältnissen innerhalb des Supermarktes umzugehen. Dies gilt vor allem im Bezug auf die Wegplanung. Während das grundsätzliche Kartografieren des Supermarktes simpel ist, kann dies nicht über die Position der tausenden Artikel gesagt werden. Die Implementierung einer solchen Datenbank führt zu einem erheblichen Aufwand und zu einer bestehen bleibenden Wartungsaufgabe. Dennoch wird für diese Projekt angenommen, dass die Position der verschiedenen Artikel innerhalb des Supermarkts bekannt ist. Die Information zur Position der Artikel wird daraufhin genutzt, um den kürzesten Weg für den gesamten Einkauf zu berechnen. Dieser Weg kann wahrscheinlich nicht so genutzt werden, da nicht nur relativ statische Hindernisse wie Einkaufswägen, sondern auch Menschen den Weg versperren können. Somit muss der Roboter entscheiden, wie er den Hindernissen effizient ausweicht, während er weiterhin versucht alle Artikel der Einkaufsliste zu kaufen. Die Datenbank mit den Positionen der Artikel ist nur eine von vielen Annahmen zur Umsetzung dieses Projektes. Mehr zu den Annahmen kann im Kapitel 4 gefunden werden.

Die Bewegungssteuerung des Roboters wird durch Reinforcement Learning trainiert. Dafür muss der Agent lernen, den berechneten Weg so gut wie möglich zu folgen. Das Einsammeln der Artikel wird simplifiziert. Die Wahrnehmung der Umgebung erfolgt durch am Agenten angebrachte simulierte Sensorik auf die im Kapitel 5 eingegangen wird. Weiterhin erhält der Roboter Informationen über sich selbst wie bspw. die eigene Geschwindigkeit.

2 Forschungsstand in den jeweiligen Bereichen

2.1 Reinforcement Learning

2.2 Robotik im Einkauf

3 ML-Agents und Unity

Für die Umsetzung des RL-Agenten wird das Open Source Projekt ML-Agents und die Spiel-Engine Unity verwendet. Unity wird primär zur Entwicklung von 3D Umgebungen für Spiele, Simulationen, Filme und vieles mehr.[1] In Unity erstellt man häufig Objekte, die daraufhin mit Komponenten ausgestattet werden. Zu diesen Komponenten gehören beispielsweise Materialien, Collision Shapes (dt.: Kollisionsformen) aber auch Skripte, die das Verhalten des Objekts bestimmen. Somit wird ein Objekt mit vielen Funktionalitäten ausgestattet. Den Objekten können noch weitere sogenannte Child Objects (dt.: Kind Objekte) angehangen werden. Somit baut sich ein Objekt wie ein Baum mit seinen Verästlungen auf. Dieser Ansatz ist für Spiele-Engins nicht unüblich und wird beispielsweise auch in Godot verwendet. Warum also ausgerechnet Unity nutzen?
Die Engine dient als Basis für dieses Projekt, da hierfür das vorher angesprochene Machine Learning Agents Toolkit erstellt wurde. Das ML-Agents Toolkit ist ein Open-Source-Projekt, bei dem jeder der möchte, zur Weiterentwicklung beitragen kann. Es ermöglicht eine einfache Implementierung von Umgebungen zum Trainieren von intelligenten Agenten.[2] Dafür liefert ML-Agents Grundbausteine für die Implementierung von Reinforcement Learning in Unity. Dazu gehört eine Vielzahl an Klassen, Komponenten und Skripts.
Zusätzlich lassen sich die erworbenen Daten einfach via TensorBoard visualisieren.[3] TensorBoard kann verschiedene Metriken des trainierten Agenten in Grafiken wiedergeben und somit den aktuellen Stand des Trainingsprozesses darstellen. Dazu gehört der Reward zu einem gegebenen Zeitpunkt, die durchschnittliche Dauer der Episoden, die Lernrate und vieles mehr. Dafür muss folgender Befehl in der Command Promp an der Position des Projekts ausgeführt werden:

```
C:\PfadzumProjekt> venv\Scripts\activate
(venv) C:\PfadzumProjekt> tensorboard --logdir results
```

Der erste Befehlt starte die virtuelle Umgebung (eng.: virtual environment) und der zweite Befehl öffnet die gespeicherten Daten in dem <results> Ordner. In diesem werden die Ergebnisse automatisch während des Trainingsprozesses gespeichert.
Des Weiteren bietet ML-Agents einen einfachen Einstieg, da eine Menge an Beispielszenen zur Verfügung gestellt werden. Durch die Betrachtung verschiedener Problemstellungen in diesen Beispielen, lassen sich Orientierungshilfen für dieses Projekt ableiten. Die Beispiele geben Anhaltspunkte über die Nutzung von Observation (dt.: Beobachtung oder Umgebungswahrnehmung), Decision (dt.: Entscheidungen), Action (dt.: Handlungen), Reward (dt.: Belohnungen) und Trainingskonfiguration. Diese sind, wie im Kapitel... angesprochen, die Hauptaspekte, die das Training des Agenten maßgeblich beeinflussen. Im Folgenden werden die wichtigsten Erkenntnisse aus den Beispielen zusammengefasst. Die Erkenntnisse werden dann im Kapitel 5 aufgegriffen und auf den hier implementierten Einkaufsroboter angewendet.

3.1 Rewards

Nach Betrachtung der Beispielszenen lässt sich für den Bereich der Rewards sagen, dass diese generell spärlich vergeben werden. Dies könnte an der geringen Komplexität der Beispielaufgaben liegen. Ein Projekt betrachtet, das Balancieren eines Balles auf der oberen Ebene eines Würfels. Der Würfel ist hier der trainierte Agent, welcher sich um sein Zentrum so rotieren muss, dass der Ball nicht von der oberen Ebene herunterrollt. Für jeden Zeitschritt, in dem der Ball auf seinem „Kopf“ bleibt, bekommt der Agent eine kleine Belohnung. Sollte der Ball herunterfallen, bekommt er eine hohe Bestrafung. Die Vergabe von Rewards erfolgt mithilfe von Gleitkommazahlen. In dem eben beschriebenen Beispiel erhält der Agent +0.1 für jeden Zeitschritt, in dem der Ball nicht herunterrollt und -1.0, falls der Ball fällt. Belohnungen und Bestrafungen werden in dem Toolkit einheitlich als Reward bezeichnet. Darum wird das im Folgenden ebenso von mir benutzt. Generell gelten für die Vergabe von Rewards diese Ansätze:

- Existenzabzüge, also das Verbleiben in einer ungewollten Situation
- Existenzbelohnung, das Bestehenbleiben in einer erwünschten Situation
- Erreichen eines Ziels oder den Abschluss einer Aufgabe, wobei unterschieden werden kann, wie gut die Aufgabe abgeschlossen wurde
- Erreichen eines Zwischenziels oder einer Teilaufgabe
- Abzüge für das Verlassen eines vorgegebenen Bereichs
- Für strukturell komplexere Agenten wie Ragdolls (dt.: Lumpenpuppe):
  - Ausrichtung des Agenten stimmt mit dem des Ziels überein
  - Geschwindigkeit des Agenten gleicht dem des Ziels

3.2 Observation und Decision

Bei der Betrachtung der Observation wird zwischen zwei Bereichen unterschieden. Einerseits die sogenannte Vector Observation (dt.: Vektor Beobachtung) und andererseits Visual Observation (dt.: visuelle Beobachtung). Zu den üblichen Vector Observationen gehören:

- Position und Rotation des Agenten
- Position und möglicherweise die Geschwindigkeit des Ziels
- Geordnete Liste mit Positionen mehrerer Ziele
- Zusätzlich Winkelgeschwindigkeit von Gliedmaßen bei Ragdoll Agenten
- Boolean, ob eine Situation zutrifft. Ein Beispiel hierfür wäre, ob sich der Agent auf dem Boden befindet

Eine andere Variante der Vector Observation ist die Verwendung von sogenannten „Ray Perception Sensor“ (dt.: Strahlen-Wahrnehmungssensor). Diese sind eine von ML-Agents mitgelieferte Komponente, die man am ehesten mit den in der Realität benutzten Lidar Sensoren vergleichen kann. Ein Lidar Sensor liefert durch das Aussenden von Laserstrahlen Daten über die Entfernung zu Objekten in der Umgebung. Diese Werte können in einer Punktwolke gespeichert werden, die daraufhin für die Visualisierung der Daten benutzt kann. Die „Ray Perception Sensor“ ermöglichen die Erkennung von verschiedenen Objekten. Hierfür werden sogenannte Tags (dt.: Marke) verwendet. Diese kann man Objekten in der Szene hinzufügen. Zum Beispiel kann ein Regal den Tag „Regal“ bekommen. Offensichtlich versteht der Agent nicht, was ein Regal ist, aber andere Informationen können so gelernt werden. Hierzu gehört möglicherweise die Breite eines Regals oder auch die Folgen, wenn der Agent in ein Regal hineinfährt.
Zu den Visual Observation gehört die Nutzung einer Kamera. Diese hat meistens eine niedrige Auflösung und stellt eine Draufsicht, Seitenansicht oder die Sichtweise von der Perspektive des Agenten dar.
Die Decision sind verhältnismäßig unkompliziert. Der Agent muss nach einer vorgegebenen Anzahl von Steps (dt.: Zeitschritten) eine Auswahl hinsichtlich seiner Action-Parameter treffen. In Unity beträgt die Dauer eines Steps 0.02 Sekunden. Die vorgegebene Anzahl an Steps wird dann mit den 0.02 Sekunden multipliziert und so erhält man den ML-Agents spezifischen Environment Step (dt.: Umgebungszeitschritt). Genauer ausgedrückt, sammelt der Agent bei jedem Environment Step Informationen über seine Umgebung, leitet diese an seine entscheidungstreffende Policy weiter und erhält dann einen Aktionsvektor zurück. Die Anzahl der Unity Steps können im Agenten eingestellt werden. Der Environment Step wird außerdem für die TensorBoard Visualisierung verwendet.

3.3 Action

Die Actions werden in Continuous Actions (dt.: kontinuierliche Aktion) und Discreate Branches (dt.: diskrete Zweige) unterschieden. Die Continuous Actions können Werte zwischen minus eins und eins annehmen. Dagegen werden die Werte, die ein Discreate Branch annehmen kann, durch seine Größe beeinflusst. Ein Branch der den Wert drei besitzt, kann nur den Wert null, eins oder zwei annehmen. Diese Werte werden dann im Code zu Bewegungs- oder Handlungsaktionen umgewandelt. Beispielsweise könnte hier minus eins für volle Geschwindigkeit Rückwärts und eins für volle Geschwindigkeit vorwärts stehen. Die Werte dazwischen reihen sich dann ein. Somit beschreiben diese Werte den Entscheidungsraum des Agenten.

3.4 Trainingskonfiguration

Die Trainingskonfigurationen stellen einen der wichtigsten Aspekte des Trainingsprozesses dar. Sie bestimmen darüber, welcher Algorithmus zum Training verwendet wird. Des Weite-

ren können hier Einstellungen zum Aufbau des neuronalen Netzes getätigt werden. Wie hoch die Lernrate ist, wie schnell diese mit der Zeit abnimmt und vieles mehr. Die Anpassung der Werte innerhalb der Trainingskonfigurationen können einen massiven Einfluss auf das Ergebnis haben.
Aber bevor ich auf die wichtigsten Parameter eingehe, zurück zum Anfang, um zu entscheiden welcher Lernalgorithmus verwendet wird. Das Toolkit liefert drei Reinforcement Learning Algorithmen mit sich. Hierzu gehören „Proximal Policy Optimisation“ (Abk.: PPO), Soft Actor-Critic (Abk.: SAC) und MultiAgent POsthumous Credit Assignment (Abk.: MA-POCA).
Wie schon vom Namen abzuleiten ist, wird MA-POCA für Trainingsumgebungen verwendet, in denen mehrere Agenten kooperativ versuchen eine Aufgabe zu lösen. Da für dieses Projekt, aber die nur die Nutzung von einem Agenten pro Umgebung vorgesehen ist, kann dieser Algorithmus ausgeschlossen werden.[4]
SAC trainiert mithilfe von aufgenommenen Erfahrungen, welche immer wieder abgespielt werden, wodurch dieser Algorithmus effizienter hinsichtlich der Trainingsdaten ist.[5]
PPO wird eher für allgemeine Aufgaben verwendet und ist zuverlässiger.[4] Wegen der fehlenden Beispieldaten, welche erst für den SAC-Algorithmus generiert werden müssten und der höheren Zuverlässigkeit von PPO, wird in diesem Projekt der PPO Lernalgorithmus verwendet.
Ein weiterer Betrachtungspunkt sind die Hyperparameter. In der Tab.1 sind die bedeutendsten Parameter einmal zusammengefasst und kurz erklärt. Anhand der Trainingsbeispiele und der ausführlichen Beschreibungen auf dem ML-Agents Github Repository[6] können daraufhin Entscheidungen getroffen werden, wie diese Parameter für die jeweilige Aufgabe eingestellt werden sollten. Um das einmal an einem Parameter zu erklären, betrachten wir einmal den Gamma Wert. Dieser soll klein sein, wenn die richtige Handlung direkt zu einem Reward führt. Ein Beispiel hierfür ist die Basic Umgebung die ML-Agents mitliefert. Hier muss sich der Agent, wie in der Abb.1 zu sehen, nach links oder rechts bewegen, um ein optimales oder suboptimales Ziel zu erreichen. Wenn er beim Ziel ankommt, startet der Prozess von vorn.

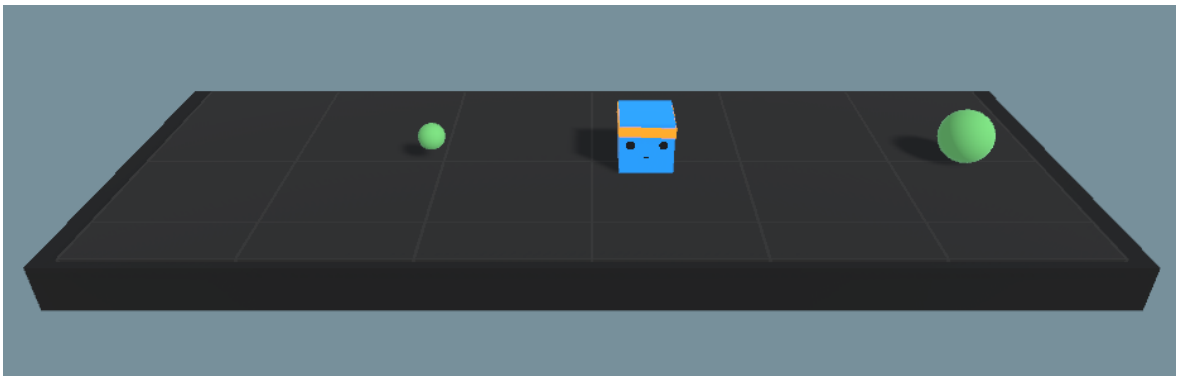

Abb. 1: Screenshot der Basic Beispielumgebung von ML-Agents. Der Agent(blau) muss durch die Bewegung nach links oder rechts ein Ziel(die grünen Sphären) erreichen. Der Kontakt mit der großen Sphäre gibt dabei einen höheren Reward als die kleine Sphäre.

Die Startposition des Agenten sowie die Position der zwei Ziele ist immer gleich. Die Aufgabe

Parameter	Default-Wert	Wertebereich	Erklärung
`batch_size`	64	32 - 512	Anzahl der Erfahrungen, die für eine Iteration der Aktualisierung des Gradientenabstiegs verwendet wird.
`buffer_size`	10240	2048 - 409600	Anzahl an Erfahrungen, die gesammelt werden sollen, bevor die Policy aktualisiert wird.
`time_horizon`	64	32 - 2048	Anzahl der Erfahrungen, die gesammelt werden sollen, bevor sie dem Buffer (`buffer_size`) hinzugefügt werden.
`learning_rate`	3e-4	1e-5 - 1e-3	Die Lernrate für den Gradientenabstieg am Anfang des Trainings.
`hidden_units`	128	32 - 512	Anzahl der Neuronen pro versteckter Ebene in dem Neuronalen Netz.
`num_layer`	2	1 - 3	Anzahl der versteckten Ebenen im Neuronalen Netz.
`gamma`	0.99	0.8 - 0.995	Beschreibt, wie weit sich der Agent um mögliche Belohnungen in die Zukunft kümmern soll.
`max_steps`	500000	5e5 - 1e7	Anzahl der zu durchlaufenden Steps bis der Trainingsprozess beendet wird.
`keep_checkpoints`	5	/	Anzahl von Modellen, die zwischengespeichert werden.
`checkpoint_interval`	500000	/	Anzahl an Erfahrungen, die gesammelt werden, bis ein Modell zwischengespeichert wird.

Tab. 1: Zusammenfassung der wichtigsten Hyperparameter für PPO frei übersetzt von [6]

ist sehr simpel und die Handlungen führen ohne Umweg zu einem Reward Signal. Deswegen wird hier der Gamma Wert niedrig gewählt. Das Lernen aus Beispielen ist nur eine Option die möglichst optimalen Werte zu finden. Ein anderer Ansatz wäre über Brute Force verschiedene Parametereinstellungen zu testen und die Ergebnisse zu vergleichen. Aus eigener Erfahrung führen suboptimale Hyperparameterwerte nicht dazu, dass der Agent keinen Fortschritt im Training macht. Meist fällt der Kumulative Reward nur etwas schlechter aus. Anders sieht das bei der Observation, der Reward Vergabe und den Aktionsparametern aus. Diese haben einen deutlich größeren Einfluss auf das Ergebnis und können bei suboptimalen Einstellungen bis zum Fehlschlag des Trainings führen. Mehr dazu im Kapitel Training im Wandel der Zeit.

3.5 Installation und Nutzung

Zum Start des Projekts war ML-Agent Release 20 die aktuellste Version, welche am 19.11.2022 veröffentlicht wurde. Der Installationsprozess kann im ML-Agents Github unter Installation[7] nachgelesen werden. Um ML-Agents nutzen zu können, sind folgende Schritte notwendig. Die Versionen in der Klammer beschreiben die verwendeten Versionen in diesem Projekt:

- Unity 2021.3 oder höher installieren (2021.3.31f1 LTS)
- Python 3.8.13 oder höher installieren (3.9.13)
- ML-Agent Github Repository klonen, um die Beispielszenen nutzen zu können (Optional)
- Pytorch installieren (1.7.1)
- mlagents Python Package installieren (0.30.0)
- in Unity im Package Manager:
  - Die package.json aus dem com.unity.mlagents Ordner hinzufügen
  - Die package.json aus dem com.unity.mlagents.extensions Order hinzufügen (Optional)
  - InputSystems Package im Projekt einbinden (Unity eigenes Package)
- Unter <ProjektPfad>Packages/manifest.json "com.unity.nuget.newtonsoft-json": "3.0.1" hinzufügen

Um zu testen, ob die Installation erfolgreich war, die Windows-Eingabeaufforderung im Projekt öffnen und die virtuelle Umgebung aktivieren. Danach den nächsten Befehl ausführen:

```
(venv) C:\PfadzumProjekt> mlagents-learn --help
```

Nun sollten mögliche Befehle, die die mlagents-learn.exe erlaubt dargestellt werden. Da die Übersichtlichkeit in der Eingabeconsole jedoch begrenzt ist, folgen ein paar Befehlsketten, die für dieses Projekt häufig genutzt wurden. Um den Lernprozess zu starten:

```
(venv) C:\PfadzumProjekt> mlagents-learn
```

Hierbei wird eine default ID erstellt und danach muss die Szene in Unity gestartet werden und der Agent trainiert mit den Standardwerten der Trainingskonfiguration. Für mehr Kontrolle wird der nachfolgende Befehl verwendet:

```
(venv) C:\PfadzumProjekt> mlagents-learn config(Position der
   Datei)\config.yaml(Dateiname) --run-id=ErstesTraining(ID)
```

Bei dieser Anweisung werden die Trainingskonfigurationen aus der config.yaml geladen und die Ergebnisse des Trainings werden in dem Ordner <ErstesTraining> gespeichert. Ml-Agents

bittet noch viele weitere Befehlsketten an. So können alte Trainingsdaten überschreiben oder ein bereits trainiertes Modell verbessert werden, indem man von diesem aus, das Training startet.

4 Generierung der Simulationsumgebung

Die Generierung der Trainingsumgebung ist ein Hauptaspekt des Projekts. Der Agent soll in einer möglichst realistischen Supermarkt-Simulation lernen einen Einkauf abzuschließen. Hierfür werden Regeln und Annahmen benötigt auf deren Basis die Generierung erfolgt.
Um die Annahmen besser nachvollziehen zu können, muss kurz erklärt werden, wie die Erstellung der Umgebung ablaufen soll. Grundsätzlich wird das Umfeld mithilfe mehrerer 2D Grids (dt.: Gitternetze) generiert. Diese beinhalten Informationen über die Position der Regale und Hindernisse. Mit diesen Daten werden dann Assets wie modellierte Regale geladen, welche an die gespeicherte Position gesetzt werden. Somit wird eine dreidimensionale Umgebung erstellt. Die Regale sind dabei modulare Elemente. Ziel ist es in jeder Iteration der Erstellung des Supermarkts eine neue zufällige Umgebung zu generieren, so dass der Agent keine Umgebung einfach auswendig lernt. Diese Zufälligkeit ist jedoch nur begrenzt umsetzbar und es kann nicht gewährleistet werden, dass die Umgebung nicht schon einmal so erstellt wurde. Dies stellt kein Problem dar, solange sich das Umfeld zwischen jeder Iteration ausreichend verändert. Im Folgenden werden die verwendeten Annahmen, sowie die Visualisierung der Umgebung erklärt. Zusätzlich wird genauer beschrieben, wie das Simulationsumfeld generiert und genutzt wird.

4.1 Getroffene Annahmen für die Implementierung des Projekts

Für die Implementierungen dieses Projekts wurden diverse Annahmen getroffen, um die Modellierung dieser Aufgabe zu vereinfachen und überhaupt erst zu ermöglichen. Die Annahmen können hierbei in zwei Bereiche unterteilt werden. Zum einen betreffen diese die Strukturierung des Supermarktes. Zum anderen die Aktionen und die Sensorik des Agenten.
Um die Simulation des Supermarktes in der Wirklichkeit zu verankern, habe ich mir mehrere Supermärkte angeschaut und den hier ansässigen Edeka ausgemessen. Dabei habe ich einige grundsätzliche Informationen zusammentragen können.

- Die Abstände zwischen den Regalen liegen bei ungefähr 2 Metern, hierbei gibt es einige wenige Ausnahmen in Verbindung mit Aktionsaufstellern, diese werden aber außenvorgelassen.
- Strukturell beginnen die meisten Supermärkte mit frischen Produkten wie Obst, Gemüse, Brot, usw. Danach folgen häufig Kühlbereiche mit Fertigprodukten. Daraufhin länger haltbare Nahrungsmittel wie Süßigkeiten, Dosen, orientalische Produkte, usw. Zum Schluss folgen oft Getränke.
- Es gibt keine Sackgassen. Der Weg in das Abteil ist nie der einzige weg hinaus.
- Meist wird versucht ein möglichst hohes Maß an Abdeckung mit Regalen zu erreichen.

Auch wenn dies im Realen nicht immer gilt, wird für die Simulation angenommen, dass der Grundschnitt des Supermarktes sowie die Aufteilungen der Abteile quadratisch oder rechteckig sind. Die Grundfläche wird aus dem zweiten Stichpunkt abgeleitet, in vier Abteile aufgeteilt. Dazu gehören Eingang, Obst und Gemüse Bereich, Abteil mit länger haltbare Güter und Getränke Bereich.
Des Weiteren wurde für die Berechnung der Wegplanung angenommen, dass der Supermarkt mit der derzeitigen Regalstruktur kartografiert wurde. Zu dieser Karte gehören nur statische Elemente, die sich über die Zeit hinweg nicht ändern. Hindernisse wie Einkaufswägen und Pappboxen mit neuen Artikeln werden nicht miteinbezogen. Sollte sich die Regalanordnung ändern, muss auch die Karte erneuert werden. Ein Ansatz mit einer dynamischen Karte, in der Hindernisse während des Trainingsprozesses vom Agenten hinzugefügt oder gelöscht werden, wird in dieser Arbeit nicht betrachtet. Grundlage hierfür ist, einerseits die Rechenersparnis hinsichtlich der festgelegten Wegplanung. Andererseits ist die Betrachtung der Reaktion des Agenten auf unbekannte Hindernisse ein wichtiger Bestandteil der Arbeit. Das Kartografieren des Supermarktes ist simpel und realistisch umsetzbar. Jedoch gilt dies nicht für die Position der Artikel innerhalb des Supermarktes. Hierfür wird angenommen, dass eine Datenbank besteht, die diese Position eingespeichert hat und innerhalb der Karte darstellen kann. Diese Annahme ist eher unrealistisch, da die Datenbank ständig angepasst werden müsste, sobald die Produkte innerhalb der Regale umsortiert werden, oder neue Artikel hinzukommen. Des Weiteren ist dies besonders fehleranfällig, da der Agent davon ausgeht, dass der ihm übergebene Weg, der Richtige ist. Falls aber die Datenbank nach dem Umräumen nicht vollständig angepasst wurde, würde der Agent ein falsches Produkt einsammeln. Dennoch wurde diese Annahme getroffen, um in der vorgegebenen Zeit dieses Projekt umsetzen zu können. Eine Alternative zu diesem Ansatz, wäre die Entwicklung einer Künstlichen Intelligenz, welche die Produkte mithilfe einer Kamera erkennen kann. Ohne die potenziellen Probleme dieser Vorgehensweise zu beschreiben, wäre hierfür zunächst ein Datensatz mit tausenden Bildern der jeweiligen Produkte aus verschiedenen Perspektiven notwendig. Die Implementierung einer solchen KI als Instanz vor der eigentlichen Untersuchung übersteigt den Rahmen einer Masterarbeit.
Zusätzlich wird das Einsammeln der Artikel simplifiziert und dabei wird getestet, ob der Roboter zur berechneten Position fährt. Das physische Einsammeln von Objekten verschiedener Größe, während der Roboter nicht unbedingt perfekt ausgerichtet ist, ist hochkomplex, weswegen diese Form der Vereinfachung gewählt wurde.
Dies sind die hauptsächlichen Annahmen, die getroffen wurden. Des Weiteren gibt es zusätzliche Anpassungen, um die Berechnung der Simulation zu optimieren. Für die Kollisionsberechnung der Regale und der Kasse wurde jeweils eine große Hitbox benutzt. Für mehr Realismus hätte man hier die genau Form der Regale durch Hitboxen nachbauen können.
Des Weiteren wurden keine zufälligen Fehler zu den Sensordaten und auch der Steuerung hinzugefügt. Alle Berechnungen wurden genau übergeben. Grund hierfür war, dass der Agent

lange kein sinnvolles Verhalten trainiert hat und das Hinzufügen von Fehlern keinen Mehrwert gebracht hätte.
Bei der Sensorik wurden zusätzlich noch weitere Vereinfachungen gewählt. Für das Projekt wurden keine realistisch funktionierenden Lidar- oder Ultraschallwellensensoren benutzt. Stattdessen wurde die Ray Perception Sensor Komponente verwendet, welche ML-Agents mitliefert. Diese ist für die Nutzung mit Agents ausgelegt und arbeitet ähnlich wie ein Lidar. Darum liegt der Grad der Abstraktion für diese Arbeit im Rahmen. Zusätzlich wird angenommen, dass die Position des Agenten sowie die Position des Ziels genau bekannt ist. Diese Werte sind notwendig, um zu erkennen, ob der Agent das Zier erreicht hat. Dies könnte durch die Nutzung von Kameradaten in einem realen Supermarkt zu einem gewissen Grad umgesetzt werden. Auf die Sensorik sowie die anderen Observationen wird genauer im Kapitel 5.3 eingegangen.
Hinsichtlich der Visualisierung wäre die Modellierung tausender Artikel extrem zeitaufwendig und würde von einem Trainingsaspekt aus nichts ändern. Deswegen werden die Regale mit Platzhaltermodellen ausgestatte. Diese sehen je nach Bereich unterschiedlich aus, um die Abteile besser visuell unterscheiden zu können.
Weitere Daten, die ich durch die Ausmessung erhielt, habe ich in die Modellierung des Supermarktes einfließen lassen. Dazu gehört beispielsweise die Höhe der Regale, die Höhe des höchsten Regalfaches, die Abstände zwischen den Fächern und weitere Elemente.

4.2 Visualisierung

Zwar würden für die Implementierung des Versuchsaufbaus die von Unity vorgegebenen Primitive ausreichend sein, um die reine Funktionalität des Ansatzes zu testen. Jedoch wäre dies für den Nutzer wenig anschaulich und kann teilweise, zu Fehlern in der Anwendung führen. Dies kann passieren, wenn nicht genügend visuelle Anhaltspunkte gegeben werden, um die Richtigkeit der Anwendung damit zu überprüfen. Die Betrachtung des Fehlerelements ist aber ein beiläufiger Aspekt und soll deswegen nicht weiter in diesem Kapitel betrachtet werden.
Um die Visualisierung anschaulicher zu gestalten, müssen verschiedene Modelle erstellt werden, die in ihrem Stil kohärent sind. Dies erhöht die Immersion des Nutzers. Ich habe mich für eine stilisierte Umgebung entschieden, die realistische Elemente mit einem Cartoon Style paart. Ziel war es, einen verspielten Stil zu implementieren, der die Erkennbarkeit der Objekte jedoch nicht erschwert. Für die Modellierung wurde die Open Source Software Blender verwendet. In dieser können sowohl 3D wie auch 2D Objekte erstellt werden. Blender liefert eine Vielzahl an Funktionalitäten, von denen für dieses Projekt nur ein Bruchteil verwendet wurde. Modellierung, Sculpting, VFX, Animation, Rigging und vieles mehr, um einmal einige Funktionen aufzuzählen, ermöglicht Blender.[8] Im Folgenden wird genauer auf den Ablauf bei der Modellierung der Szenenelemente und den Aufbau des Roboters eingegangen. Die Erklärungen umfassen größtenteils Abstrahierungen und sind keine Schritt für Schritt Anleitung zum Nachbauen.

4.2.1 Visualisierung der Umgebung

Bei der Modellierung der Umgebungsobjekte gab es zwei wichtige Faktoren. Einerseits die Umsetzung des geplanten Stils und andererseits die Anzahl der “Vertices“ möglichst klein zu halten. Üblicherweise werden, um das Training zu beschleunigen, mehrere Instanzen der Simulationsumgebung erstellt. Eine detailgetreue Modellierung würde dann dazu führen, dass sich deutlich mehr “Vertices“ in der Szene befinden. Diese benötigen bei der Darstellung mehr Rechenleistung und können somit den Trainingsprozess negativ beeinflussen. Beispielsweise könnte eine Folge des höheren Rechenaufwands weniger Instanzen der Szene sein. Dadurch würde die Dauer des Trainings sich um Minuten bis Stunden verzögen. Was wiederum dazu führt, dass weniger verschiedene Ansätze getestet werden können. Somit entsteht ein Dilemma zwischen der Generierung einer visuell ansprechenderen Umgebung und der damit einhergehenden erhöhten Trainingsdauer. Da der Fokus der Arbeit aber auf der Implementierung eines RL-Agenten liegt, ist der visuelle Aspekt eher zweitrangig. Somit muss ein Mittelweg gefunden werden, der eher in Richtung Performance getrimmt ist.
Mein Ziel für den stilistischen Teil der Modellierung war, dass keine oder nur selten scharfe Kanten zu erkennen sind. Dies lässt sich durch die Funktion SShade Auto Smoothïn der Blender Version 4.0 und höher umsetzen. In der Abbildung kann man den Unterschied noch einmal genauer erkennen. Um die klar erkennbaren Übergänge zu verhindern, muss der Winkel zwischen den Kanten eines Objektes, eine gewisse Gradzahl unterschreiten. Dafür kann man einerseits den Winkel in den Einstellungen anpassen. Wird der Winkel zu hoch gewählt, sieht die Beleuchtung des Objektes unnatürlich aus. Ist die Einstellung zu gering, sind die Kanten wieder erkennbar. Die andere Option ist die Erhöhung der Anzahl der Kanten, um die Übergänge natürlicher aussehen zu lassen. Dies führt aber wieder zu dem vorher angesprochenen Dilemma.
Nun gibt es zwei Ansätze für unterschiedliche Objektformen. Bei rechteckigen Objekten lohnt es sich den sogenannten "Bevel Modifier" zu verwenden. Dieser fügt an den äußeren Kanten neue Topologie hinzu, um diese abzurunden. Dadurch reflektiert das einfallende Licht deutlich natürlicher und das modellierte Objekt wirkt realistischer. Dafür reicht es schon wenige Kanten hinzuzufügen, wodurch die Menge an „Vertices“ nur geringfügig steigt. Runde Modelle sind wiederum ein wenig komplizierter. Diese wirken bei genauerer Betrachtung erst rund, wenn die Anzahl an Kanten verhältnismäßig hoch ist. Dies führt automatisch dazu, dass runde Modelle mehr „Vertices“ benötigen als Rechteckige. Hier muss eher beachtet werden aus welcher Entfernung das Objekt betrachtet wird und wie hoch die Auflösung sein sollte, damit die Kanten nicht auffallen.
Meine Erfahrung in diesem Bereich liefert mir ungefähre Anhaltspunkte, welche Anzahl an „Vertices“ in Abhängigkeit von der Objektform gerechtfertigt sind. Für kleine rechteckige Objekte wie zum Beispiel die Artikel des Supermarktes habe ich mir eine Grenze von 1000 Vertices gesetzt. Runde Artikel hingegen können bis zu 3000 Vertices haben. Die Anzahl ist nicht besonders hoch und befindet sich eher im Low Poly Bereich.

4.2.2 Visualisierung des Agenten

Diese Grenzen gelten jedoch nicht für Objekte, die eher im Fokus liegen wie der Roboter. Hier sind deutlich höhere aufgelöste Topologien in Ordnung, da diese auch nur einmal pro Simulationsumgebung vorkommen. Für die Modellierung des Einkaufsroboters habe ich mir den Prototypen der TU Chemnitz als Anhaltspunkt ausgesucht siehe Abb. Der Roboter ist mit einem Greifarm ausgestattet und besitzt einen Korb in dem die eingesammelten Objekte gelagert werden können.

4.3 Generierung und Parameter

Zusammengefasst für die Generierung. Aus den Erkenntnissen wird für die Simulation abgeleitet, dass der Abstand zwischen den Regalen mindestens 2 Meter betragen muss. Das verhindert automatisch die Generierung von Sackgassen. Außerdem wird versucht, die Abdeckung mit Regalen möglichst hoch zu wählen.

4.4 stationäre Hindernisse

5 Struktur der trainingsbezogenen Elemente des Agenten

Um mithilfe von ML-Agents einen Agenten in Unity zu erstellen, muss dem Robotermodell ein Script hinzugefügt werden, welches von der Klasse "Agent" erbt. Dadurch erstellt sich die "Behavior Parameters" Komponente, welche in der Abbildung 2 zu sehen ist. In der Abbildung gibt es viele Parameter die genauer erklärt werden müssen, um den Aufbau des Agenten besser zu verstehen. Hierzu gehören die Vector Observation, die Actions und die Use Child Sensors (dt.: nutze Kind Sensoren). Auf diese wird in den Unterkapiteln eingegangen.

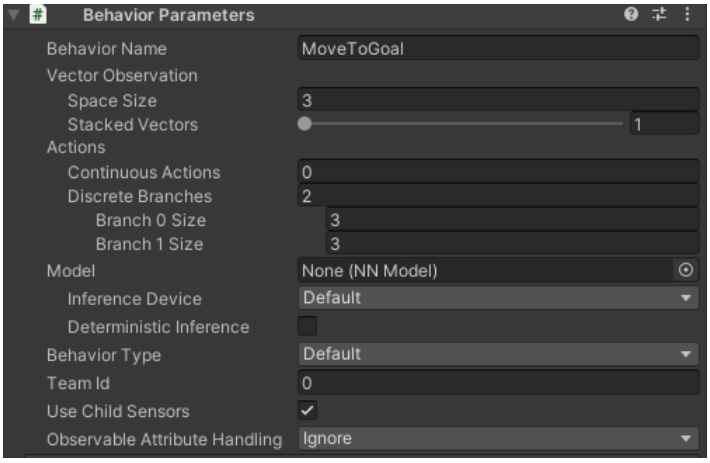

Abb. 2: Die Behavior Parameters Komponente von ML-Agents.

Die restlichen Parameter haben eine geringer Bedeutung in diesem Projekt. Der Behavior Name beschreibt den Namen des zukünftig erstellten neuronalen Netzes.
Dem Model Element kann ein bereits trainiertes NN hinzugefügt werden. Daraufhin kann man die Simulation starten und des Verhalten des trainierten Modells innerhalb der Umgebung betrachten.
Das Inference Device beschreibt mit welchem Verfahren das Modell trainiert werden soll. Hier kann zwischen "Default", "GPU", "CPU" und "Burst" unterschieden werden. Da in früheren Projekten keine relevanten Unterschiede erkennbar waren, wurde in diesem Projekt der Default Wert verwendet.
Der Behavior Typ bestimmt darüber, ob der Agent vom Modell oder vom Nutzer gesteuert werden soll. Wenn der Default Wert ausgewählt ist, wird automatisch der Agent genutzt, falls unter Model ein trainiertes neuronales Netz hinzugefügt wurde.
Über die Team Id kann eingestellt werden, zu welchem Team ein Agent gehört. Dies wird notwendig, wenn man eine kompetitive Simulationsumgebung betrachtet, in denen Agenten gegeneinander antreten. Dazu würde beispielsweise die Soccer Umgebung von ML-Agents gehören.
Über Observable Attribute Handling können weitere Beobachtungen dem Agenten hinzugefügt werden. Ein Beispiel hierfür könnten die derzeitigen Lebenspunkte eines Agenten in einem Spiel sein. Die Einstellung wurde ebenfalls nicht benutzt.

Im Kapitel 3.4 wurden die wichtigsten Punkte hinsichtlich der Betrachtung des Agenten angeschnitten. Aus den Erkenntnissen lassen sich Ableitungen für die Implementierung eines Einkaufsroboters finden. Diese werden im Folgenden ausführlich betrachtet.

5.1 Die Steuerung des Agenten

Die Basis der Steuerung kann grundsätzlich in zwei Bereiche unterteilt werden. Hierzu gehört einerseits das Einlenken nach links und rechts. Andererseits die Beschleunigung vorwärts sowie rückwärts. Auf diese Basis aufbauend können dann verschiedene Stärkegrade der jeweiligen Handlung implementiert werden. Wie bei einem Auto, wo nicht nur voll nach links eingelenkt werden kann, sondern auch den Raum dazwischen.
Um dieses Verhalten zu ermöglichen, wurde initial eine Steuerung über Continuous Actions eingebaut. Dafür werden zwei verschiedene Actions benötigt, um den gesamten Bewegungsraum abdecken zu können. Bei den Continuous Actions kann der Agent bei jeder Entscheidung einen Wert zwischen minus eins und eins wählen. Diese Zahlen werden dann direkt auf die Steuerung gemappt. Beispielsweise steht der Wert minus eins dann für voll links einlenkten. Somit kann der Agent den Zwischenraum nutzen, um die Stärke seiner Beschleunigung und Lenkung zu beeinflussen.
Dies führte im Training jedoch zu sehr diskontinuierlich Verhalten. Der Agent fuhr keine zielgerichtet Linie, sondern wechselte die Richtung in jedem Entscheidungsschritt. Dadurch wirkte das Verhalten sehr unnatürlich, da der Roboter innerhalb einer Sekunde mehrmals seine Richtung komplett änderte. Diesem Problem hätte man im Code entgegenwirken können, in dem man den Grad der Richtungsänderung begrenzt. Jedoch würde dies, in Kombination mit der flacheren Lernkurve von Continuous Actions, zu deutlich längeren Trainingszeiten führen.
Deswegen wechselte ich nach einiger Zeit zu den Diskrete Branches. Hier wurden wieder zwei verschiedene Actions benötigt, die ebenso für das Einlenken und Beschleunigen benutzt werden. Anders als bei den Continuous Actions kann sich der Agent in jedem Entscheidungsschritt nur zwischen drei Werten entscheiden. Null steht für nichts unternehmen, eins für links einlenken und zwei für nach rechts lenken. Gleiches gilt für die Beschleunigung. Um den Stärkegrad der jeweiligen Aktion beeinflussen zu können, wird hierfür die maximale Dauer zwischen den Entscheidungen ausgenutzt. Der Agent kann somit leicht einlenken, wenn er sich nur einmal für das Lenken entscheidet. Möchte er stärker in eine Richtung fahren, muss er sich mehrmals hintereinander dafür entscheiden. In der Abbildung 3 sieht man die Decision Requester Komponente von ML-Agents. Diese muss einem Agenten hinzugefügt werden, damit dieser Entscheidungen treffen kann.

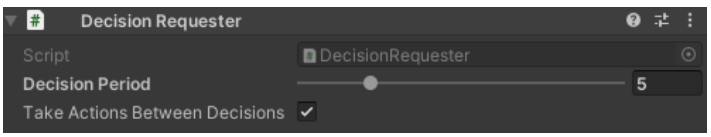

Abb. 3: Die Behavior Parameters Komponente von ML-Agents.

Wie man sehen kann, ist die Checkbox Take Actions Between Decision (dt.: Führe Aktionen zwischen Entscheidungen aus.) ausgewählt. Dadurch werden, wie der Name schon sagt, die Aktionen zwischen den Entscheidungen ausgeführt. Die Decision Period beschreibt, wie viele Unity Steps vergehen sollen, bis der Agent die nächste Entscheidung trifft. In diesem Beispiel beträgt die Zeit zwischen Entscheidungen 0.1 Sekunden. Dadurch wird jede getroffene Aktion 0.1 Sekunden ausgeführt. An diese Dauer kann nun die Berechnung der Steuerung abgestimmt werden, damit sie in der Simulationsumgebung möglichst natürlich wirkt.
Um die Physics Engine von Unity zu verwenden, wurde dem Roboter eine Rigidbody Komponente angehangen. Die letztendlich berechneten Werte, die auf Basis der Entscheidung des Agenten beruhten, wurden dann über Funktionen des Rigidbody ausgeführt. Dem Agenten wurde zusätzlich eine gewisse Menge an Drag (dt.: Widerstand) hinzugefügt, um zu verhindern, dass er unendlich schnell wird.

5.2 Reward Vergabe

Die Reward Vergabe wird entlang der Zielsetzung des Projekts modelliert. Diese wurde im Kapitel 1 erwähnt. Grundsätzlich soll der Einkaufsroboter möglichst schnell alle Artikel einsammeln und an einem vorgegebenen Ort abgeben. Unterstützend hierfür wird dem Agenten der kürzeste Weg zum nächsten Produkt angezeigt. Zusätzlich soll der Roboter nicht mit der Umgebung kollidieren, um Schäden an sich selbst und dem Umfeld zu verhindern.
Aus dieser Zielsetzung in Kombination mit den Erkenntnissen aus dem Kapitel 3.1 in dem auf die Reward Vergabe in den Beispielszenen eingegangen wurde, können nun die optimalen Rewards gebildet werden. Somit werden folgende Rewards vergeben:

- Existenzabzüge, um einen Anreiz zu geben, die Umgebung möglichst schnell abzuschließen
- kleine Belohnung beim Einsammeln von Wegpunkten
- größere Belohnung für das Erreichen der Artikel
- sehr große Belohnung für das Erreichen des Abgabebereichs
- größere Abzüge bei Kollision mit den Regalen und Hindernissen
- kleine Abzüge beim kontinuierlichen Beistehenbleiben in einer Kollision

Die genauen Werte der Rewards veränderten sich während des Trainings. Teilweise wurden gewählte Rewards verworfen und später wieder hinzugefügt. Auf die genauen Veränderungen wird im Kapitel 6.3 eingegangen.

5.3 Wie sieht der Agent die Umgebung?

In diesem Projekt wurden zwei Methoden zur Wahrnehmung der Umgebung genauer betrachtet. Hier spielen die anfangs angesprochenen Vector Observation und die Use Child Sensors eine große Rolle.
Unter Vector Observation befindet sich in der Abbildung 2 die Space Size (dt.: Größe des Platzes) und die Stacked Vectors (dt.: gestapelter Vektor). Space Size beschreibt die Anzahl an Werten, die an den Agenten übergeben werden. Eine Space Size von drei erlaubt, die Übergabe von drei Gleitkommawerten. Diese könnten beispielsweise die derzeitige lokale Position des Agenten im dreidimensionalen Raum darstellen.
Der Stacked Vectors Parameter beschreibt, wie viel Space Size an Daten für die nächste Entscheidung betrachtet werden soll. Bei einem Stacked Vector von eins ist nur die derzeitige Position von Bedeutung. Ein Stacked Vector von zwei übergibt die jetzige Position und die Position vor einem Zeitschritt. Damit kann der Agent möglicherweise auf Zusammenhänge schließen.
Angenommen wir betrachten das Basic Beispiel aus der Abbildung 1. Nur das diesmal ein Hindernis zufällig auf einer Seite vor dem Ziel platziert wird. Der Agent hat durch seine eingeschränkten Bewegungsoptionen nur die Möglichkeit, in das andere Ziel zu fahren. Würde man nur die derzeitige lokale Position des Agenten übergeben, erkennt dieser nicht, dass er nicht weiterkommt. Übergibt man aber zusätzlich die vorherige Position. Könnte der Agent darauf schließen, dass er sich zwischen dem letzten Zeitschritt und dem jetzigen nicht von der Stelle bewegt hat. Dadurch könnte er zielgerichteter die Entscheidung treffen in die andere Richtung zu fahren.
Wichtig anzumerken ist, dass durch längeres Training der Agent ohne Stacked Vector trotzdem auf die Lösung kommen würde und in das andere Ziel fährt. Jedoch gilt dies nur für diese einfache sehr deterministische Umgebung. Bei Simulationen in denen sich das Umfeld in jeder Iteration stark ändert und Bewegungen im zwei oder dreidimensionalen Raum stattfinden, kann es schnell passieren, dass der Agent steckenbleibt.
Die in den Beispielszenen verwendeten Vector Observation beinhalten häufig Informationen über die eigenen Parameter des Agenten und die des Ziels. Die Werte reichen aber nicht aus, um sich in einer Umgebung mit Hindernissen zurecht zu finden. Dafür sind die Use Child Sensors von Bedeutung. Damit können dem Agenten Sensoren als Kind Objekte angeheftet werden und der Agent benutzt automatisch die erhaltenen Daten. Für die Wahrnehmung der Umgebung eignen sich die Ray Perception Sensoren. Diese senden, wie in Abbildung 4 zu sehen, Strahlen aus, welche bei Kontakt mit einem Objekt Daten an den Agenten weiterleiten. Hierzu gehören die Tags, welche für Objekte in der Szene ausgewählt werden können. So kann man entscheiden, worauf der Agent reagieren und was er als unterschiedliches Objekt wahrnehmen soll.
Da ein Ziel war, die Fähigkeiten des Agenten in der Wirklichkeit zu verankern, wurde diese Erkennung auf das Minimum reduziert. Für dieses Projekt wurde angenommen, dass ein

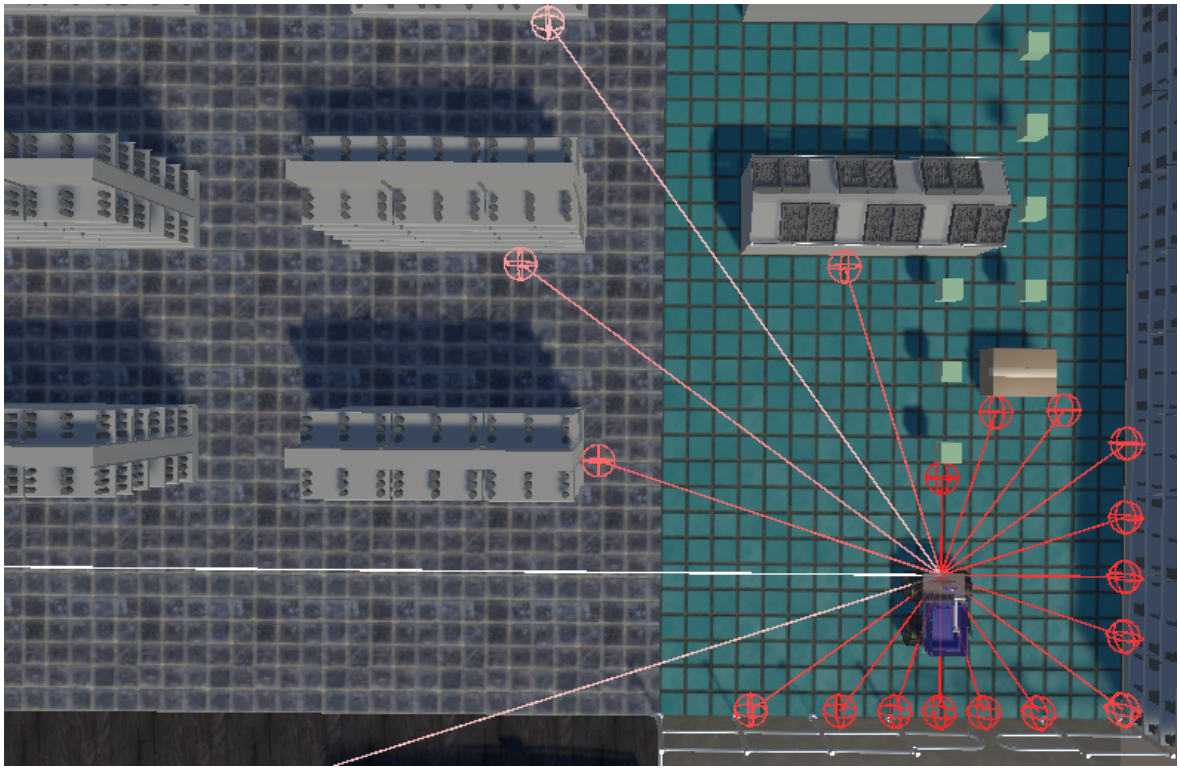

Abb. 4: Ray Perception Sensor in der Simulationsumgebung. Die rot markierten Strahlen zeigen, dass diese mit einem Objekt kollidieren. Weiße Strahlen hingegen treffen auf kein Objekt.

ausgesendeter Strahl, welcher ein Hindernis oder eine Wand trifft, immer den selben Tag zurückgibt. Die einzige Ausnahme stellen die erstellten Wegpunkte und das Ziel dar. Diese wurden jeweils mit unterschiedlichen Tags ausgestattet. Der Grund hierfür liegt darin, dass der Agent lernen soll den Hindernissen auszuweichen und den Wegpunkten zu folgen. Hätten beide denselben Tag würde er lernen, den Wegpunkten nicht zu folgen, da er den Tag voraussichtlich mit einer Bestrafung in Verbindung setzt.
Die Einstellungsmöglichkeiten des Ray Perception Sensor sind in der Abbildung 5 zu sehen. Unter Detectable Tags (dt.: erkennbare Marken) können die Tags hinzugefügt werden, die der Sensor erkennen soll.
Rays Per Direction (dt.: Strahlen pro Richtung) beschreibt die Anzahl an Strahlen die ausgesendet werden. Dabei teilen sich die Strahlen in zwei Richtungen, wodurch sich die eingestellte Zahl verdoppelt.
Die Max Ray Degrees (dt.: maximaler Strahlenwinkel) könnte man auch als Field of View (Abk.: FOV) betrachten. Eine Einstellung von 40 wurde so zu einem 80 Grad FOV führen. Die Ausrichtung der Strahlen erfolgt hierbei von einem Punkt aus in eine zusammenhängende Richtung.
Den Sphere Cast Radius (frei dt.: zusätzliche Sphärenradiuserkennung) kann man am besten Anhand der Abbildung 4 erklären. Hier werden anstelle eines Strahls Sphären in die jeweiligen Richtungen ausgesendet, bis diese mit einem Objekt kollidieren oder die eingestellt maximale

Reichweite des Strahl (eng.: Ray Length) erreichen. Die Sphären erhöhen den Bereich in dem ein Objekt erkannt wird. Dadurch soll verhindert werden, dass ein Strahl gerade so an einem Objekt vorbeifliegt und es somit nicht erkennt.
Ein weiterer wichtiger Parameter ist die Ray Layer Mask (dt.: Strahlen Ebenen Maske). Hier können bestimmte Umgebungstypen eingestellt werden, die der Sensor nicht erkennen soll. Dieser Parameter ist notwendig, um verschiedene Sensoreinstellungen zu testen. Tags, die nicht zu den Detectable Tags gehören, können zwar nicht erkannt werden, aber der Strahl kollidiert trotzdem mit dem Objekt. Wenn dem Objekt jetzt eine Maske gegeben wird, die nicht bei der Ray Layer Mask ausgewählt ist, fliegt der Strahl durch das Objekt hindurch und wird somit ignoriert. Die restlichen Werte sind selbsterklärend oder für das Training nicht weiter bedeutend.

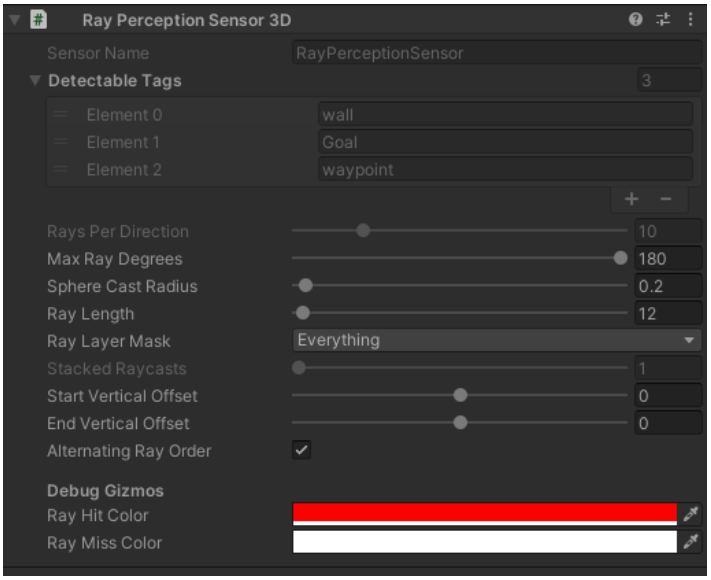

Abb. 5: Die Behavior Parameters Komponente von ML-Agents.

Anders als bei Lidar Sensoren wirkt es so, als würde der Ray Perception Sensor keine Entfernungen berechnen und dem Agenten übergeben. Scheinbar wird nicht viel mehr als der erkannte Tag an den Agenten gesendet. Zwar wird in der Dokumentation[9] beschrieben, wie die Endposition und andere Werte ausgegeben werden können. Jedoch deuten die Ergebnisse aus dem eigenen Training daraufhin, dass der Agent diese Werte nicht erhält. Genauso wird in der Zusammenfassung zu den verschiedenen Beispielumgebungen von ML-Agents[10] bei den Beispielen angegeben wie groß die Vector Observation jeweils sind. Die angegebenen Werte für die Ray Perception Sensor sind zu klein, um alle Werte aus der Dokumentation an den Agenten zu übergeben.
Der Ray Perception Sensor wird trotzdem durch unterschiedliche Einstellungen im Folgenden als vereinfachter Lidar und Ultraschallwellen Sensor verwendet. Diese Einstellungen haben sich während des Trainings immer wieder geändert, weswegen diese nicht hier sondern in dem Kapitel 6.3 erklärt werden.

6 Trainingsablauf

6.1 Anfänge...

6.2 Curriculum Learning

6.3 Training im Laufe der Zeit

6.4 Erkenntnisse und Learnings

Würde ich eher in die Auswertung packen.

7 Auswertung

8 Ausblick

Ein wichtiger Bestandteil der Betrachtung, die hier wegen fehlender Zeit ausgelassen wurde, ist der Umgang des Agenten mit dynamischen Hindernissen. Einkaufsroboter müssen dazu in der Lage sein mit dem Verhalten von Menschen umzugehen. Diese weisen in einem realen Umfeld unterschiedliche Muster auf. Teilweise würden sie dem Roboter ausweichen, andere stellen sich ihm vielleicht aktiv in den Weg oder bleiben an Ort und Stelle stehen. Herauszufinden wie der Roboter dieser vorm von Hindernissen effizient ausweicht, während er weiterhin versucht, alle Artikel der Einkaufsliste zu kaufen, könnte Bestandteil einer weiterführenden Arbeit sein.
Ebenso ist die Implementierung eines Vergleichsmodell von Interesse. Dieses könnte beispielsweise ein regelbasiertes System sein, welches gleichbleibend mit Hindernissen umgeht. Für die Betrachtung der verschiedenen Systeme würde man diese in getrennten Umgebungen, die über dieselben Hindernisse und zu kaufende Artikel verfügen, gegeneinander antreten lassen. Durch die Zufälligkeit der Umgebung kann nicht gewährleisten werden, dass die simulierten Menschen nach Kontakt mit dem Roboter gleich agieren. Dennoch würde die Nutzung eines Vergleichsmodells Anhaltspunkte über die Effizienz des KI-Agenten geben. Durch diese Form der Betrachtung könnte man einfacher die Stärken und Schwächen der jeweiligen Ansätze aufschlüsseln. Hierfür sollten Gesichtspunkte wie Dauer des Einkaufs, Anzahl an Kollisionen und Verärgerung der simulierten Menschen analysiert werden. Wenn man den Agenten auf Basis dieser Gesichtspunkte optimiert, wäre das Ergebnis, ein Roboter, der so wenig wie möglich Fahrtweg zurücklegt, während er die anderen Einkäufer minimal stört und die Dauer des Einkaufs gering hält.
Zusätzlich sollten Vereinfachungen, die in dieser Implantierung vorgenommen wurden, sukzessive abgebaut werden. Des Weiteren könnten noch komplexere Regalstrukturen, wie man sie aus dem Edeka oder Rewe kennt, hinzugefügt werden. Desto weniger Simplifizierungen verwendet werden, umso aussagekräftiger sind die Ergebnisse.

Literatur

[1] *Erstellen*. Unity. URL: https://unity.com/de (besucht am 06. 03. 2024).

[2] *Unity ML-Agents Toolkit*. Unity. URL: https://github.com/Unity-Technologies/ml-agents (besucht am 06. 03. 2024).

[3] *TensorBoard: TensorFlow's visualization toolkit*. TensorFlow. URL: https://www.tensorflow.org/tensorboard (besucht am 06. 03. 2024).

[4] *ML-Agents Toolkit Overview*. Unity. URL: https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/ML-Agents-Overview.md#training-in-cooperative-multi-agent-environments-with-ma-poca (besucht am 06. 03. 2024).

[5] *Training with Soft-Actor Critic*. Unity. URL: https://github.com/yosider/ml-agents-1/blob/master/docs/Training-SAC.md (besucht am 06. 03. 2024).

[6] *Training Configuration File*. Unity. URL: https://github.com/Unity-Technologies/ml-agents/blob/release_20_branch/docs/Training-Configuration-File.md#ppo-specific-configurations (besucht am 06. 03. 2024).

[7] *Installation*. Unity. URL: https://github.com/Unity-Technologies/ml-agents/blob/release_20_branch/docs/Installation.md (besucht am 06. 03. 2024).

[8] *Blender 4.0*. Blender. URL: https://www.blender.org/ (besucht am 06. 03. 2024).

[9] *Struct RayPerceptionOutput.RayOutput*. Unity Technologies. 18. Okt. 2023. URL: https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/Unity.MLAgents.Sensors.RayPerceptionOutput.RayOutput.html (besucht am 09. 03. 2024).

[10] *Example Learning Environments*. Unity. URL: https://github.com/Unity-Technologies/ml-agents/blob/release_20_branch/docs/Learning-Environment-Examples.md (besucht am 09. 03. 2024).