INF
Informatik

Bachelorarbeit

Erstellen von Reinforcement Learning Agenten in Unity mithilfe von ML-Agents

Luca Rommler
Matrikel-Nr.: 766788

Erstprüfer: Prof. Dr. Benjamin Himpel

ZweitprüferIn: Prof. Uwe Kloos

Abgabedatum: 30.06.2022

Abstract:
In dieser Ausarbeitung werden Grundlagen und verschiedene Vorgehensweisen erläutert und untersucht, welche eingesetzt werden um selbst-lernende Agenten auf eine vordefinierte Problemstellungen zu trainieren. Die Vorgehensweisen und Grundlagen welche eingesetzt werden sind dabei generell gültig und anwendbar. Die Umsetzung der Agenten geschieht in Unity mit Hilfe der von ′Unity Technologies′ ′Open Source′ erstellten ′MLAgents′-Bibliothek und der von dem ′Google Brain Team′ ′Open Source′ erstellten KI-Bibliothek ′TensorFlow′. Diese beiden Bibliotheken bieten viele nützliche Werkzeuge zum erstellen von Künstlichen Intelligenzen und selbst lernenden Agenten aller Art.
Im Verlauf der Ausarbeitung werden verschiedene generelle Vorgehensweisen zum erstellen dieser selbst-lernenden Agenten anhand von Fallbeispielen genauer erläutert und demonstriert. Hierbei werden verschiedene Aspekte wie Eingabemethoden, Hyperparameter und Trainingsvarianten mithilfe einzelner Fallbeispiele genauer beleuchtet. Es werden weiter Problemstellungen dargelegt welche während der Planung, Implementierung und des Trainings beachtet werden sollten um gute Ergebnisse zu erhalten.

Hierbei werden die Vorgehensweisen und Erkenntnisse anhand von speziell ausgewählten Fallbeispielen dargelegt. Fragen die sich während der Umsetzung dieser Agenten kristallisiert haben werden zudem angegeben. Das betrachten und beantworten dieser Fragen während der eigenen Problembehandlung führt zu strukturiertem Vorgehen und generell zu besseren Ergebnissen. Typische Problemstellungen die während der Planung, Umsetzung oder des Trainings der Agenten auf treten können, wie z.B. ′Overfitting′ und ′Curriculum Learning′ werden zusätzlich von Fallbeispielen offengelegt und erläutert.

Am Ende der Ausarbeitung hat der Leser ein gutes Verständnis über Vorgehensweisen, Planung und ′Troubleshooting′-Ansätzen im Bereich des Reinforcement Learning.

Inhaltsverzeichnis

1 Motivation

Künstliche Intelligenz ist bei weitem keine neue Erfindung. Das erste mal wurden KIs in der Mitte der 1950er Jahre einwickelt [27, 28] um mehrere mathematische Lehrsätze zu beweisen. Von hier an entwickelte sich KI nur weiter.
In heutiger Zeit ist KI mittlerweile in fast allen größeren Software-Projekten implementiert. Sei es durch einfache ZustandsMachinen die anhand Ihres Zustandes gewisse Ereignisse unterschiedlich verarbeiten oder Selbst-lernende Künstliche Intelligenzen welche verschiedenste Algorithmen nutzen um eigenständig oder überwach lernen gewisse Problemstellungen zu lösen.
Machine Learning findet sich hierbei nahezu überall wo Daten in irgendwelcher Art oder Form verarbeitet werden. Sei es in der Wirtschaft für Modelle welche den Einfluss von COVID-19 auf die US-Motor-Nachfrage darstellen [11], in der Medizin um Gehirntumore besser erkennen zu können [13], in der Sprache um Zeichen- zu gesprochener Sprache umzuwandeln [12] oder im Kundenservice zum Erstellung von Chatbots zu initialen Kategorisierung von Kundenanfragen.
Die Verwendungszwecke von Machine Learning sind reichlich und vielfältig.

Die Branche der Spiele Entwicklung hat sich in den letzten Jahrzehnten stetig vergrößert und ist mittlerweile profitabler als Musik- und Filmindustrie kombiniert. KI ist in dieser Branche jedoch nichts neues, da schon in älteren Spielen wie ′Pac-Man′ (1980) [10] KI genutzt wurde um den Gegnern der Spielers verschiedene Charaktereigenschaften zu geben um den diesen herauszufordern. Wenn es zum nutzen von Machine Learning in Video-spielen kommt, hat die Branche jedoch noch einen weiten weg vor sich um diese Technologie effektiv und profitable einzusetzen. Allerdings hat sich hier in den letzten Jahren durch die weitere Entwicklung von Systemen und Tools vieles getan um mehr Aufmerksamkeit und einen einfacheren Zugang für das Thema Machine Learning in Video spielen zu ermöglichen.

Eines dieser Tools ist das ′ML-Agents′ Toolkit welches von ′Unity Technologies′ als ′Open-Source-Software′ entwickelt wird. Dieses ermöglicht uns Agenten in selbst erstellten Umgebungen mit Daten (welche aus der Umbegung gesammelt werden) zu trainieren. Diese Ausarbeitung soll hierbei einen guten Einblick verschaffen wie dieses Toolkit grundlegend funktioniert. Im genaueren welche Technologien/Algorithmen genutzt werden, welche Möglichkeiten das Toolkit zum Umsetzen von Agenten ermöglicht, generelle Vorgehensweisen zur Erstellung von

Trainingsumgebungen und häufige Problematiken welche während der Planung, Implementierung und des Trainings auftreten können.

2 Grundlagen

Bevor wir mit den Grundlagen anfangen, soll gesagt sein, dass in dieser Ausarbeitung Unity und ML-Agents genutzt wird. Diese werden hauptsächlich dafür verwendet um Umgebungen zu erstellen in denen dann ein (oder mehrere) Agent(en) trainiert werden sollen.
Die Anwendung/Darstellung ist hierbei spezifisch auf Unity und ML-Agents ausgelegt. Jedoch ist die Theorie und die Vorgehensweisen die in dieser Ausarbeitung beschrieben werden generell gültig und kann auch außerhalb von Unity und ML-Agents genutzt werden um Umgebungen zu erstellen und Agenten zu trainieren.
Unity und ML-Agents dienen hier lediglich als Beispiel Toolkit, da dieses uns ermöglicht, schnell und effektiv einfache Umgebungen zu erstellen um gewisse Aspekte des Prozesses zu demonstrieren.

2.1 Stand der Technik

Machine Learning wird heutzutage in vielen verschiedenen Themenbereichen mit den unterschiedlichsten Zielen, Methodiken und Algorithmen genutzt [19]. Je nach Ziel und Umfeld werden hier verschiedene Kombinationen aus diversen Attributen gewählt um eine gewisse Problemstellung zu automatisieren oder zu lösen. Die 3 Hauptattribute hierbei sind die Architektur des Neutralen Netzwerkes, die Machine Learning Methodik/Vorgehensweise und die genutzten Algorithmen welche während des Trainings genutzt werden um die Daten zu verarbeiten.

Bei den Architekturen gibt es eine Vielzahl an verschiedenen Strukturen, mit verschiedenen Komponenten, welche verschiedene Verhaltensweisen aufzeigen. Diese sind sehr gut an einem Schaubild des Äsimov Instituteßu erkennen, welches unter ′Asimov Institue - Neural Network Zoo′ gefunden werden kann. Hier können je nach Aufbau des Neuronalen Netzwerkes Dinge wie Gedächtnis oder ′Muster′-Erkennung nachgebildet werden. Eine genauere Erklärung des Neuronalen Netzwerkes kann unter Neuronale Netzwerke′ gefunden werden.

Ein weiterer wichtiger Aspekt bilden die Algorithmen welche genutzt werden um die Daten innerhalb des Neuronalen Netzwerkes zu verarbeiten. Hier können je nach Aufgabenstellung und Architektur des Netzes verschiedene Algorithmen genutzt werden um ein gewünschtes Verhalten zu erzielen. Dabei gibt es heutzutage eine Vielzahl an verschiedenen Algorithmen. Hierbei haben verschiedene Reinforcement Learning Methodiken verschiedene Algorithmen welchen zur Verfügung stehen. In der folgenden Aufzählung sind die meist genutzten Algorithmen Aufgezählt [19].

- Lineare Regression
- Entscheidungsbäume
- 'Random Forest'
- Logistische Regression
- K-Nächster Nachbar
- Anomalie Erkennung
- Hauptkomponentenanalyse
- 'Proximal Policy Optimization'
- 'Soft Actor Critic'

Der letzte große Aspekt sind die 4 verschiedenen Methodiken wie Machine Learning heutzutage Praktiziert wird. Diese werden im nachfolgenden Kapitel genauer erläutert.

Wie schon oben beschrieben ergeben sich aus verschiedenen Kombinationen der verschiedenen Aspekte unterschiedliche Machine Learning Konstrukte, welche genutzt werden können um eine gigantische Vielzahl an Aufgabenstellung entweder zu automatisieren oder zu lösen. Einige Beispiele hierzu sind: Gesichtserkennung im Smartphone, autonomes Fahren, E-Mail-Spam-Erkennung, Spracherkennung, Stöber-Verhalten' von Kunden, Recommender Systems, Data Mining, ... [19, 21, 22].

2.2 Maschine Learning

Machine Learning ist eine Unterkategorie von Künstlicher Intelligenz. Machine Learning kann wie folgt definiert werden: ′Machine Learning ist das Anwenden von rechnerischen Methoden, welche Erfahrung nutzen, um ihre Leistung zu verbessern oder um genaue Vorhersagen treffen zu können′ [15]. Mit Erfahrungen sind hier frühere Informationen gemeint, welche dem Agenten zu Verfügung gestellt wurden. Diese werden typischerweise in digitaler Form von dem Agenten analysiert und werden typischerweise entweder durch menschlich erstellte, sortierte und markierte Trainingsdatensäten oder dem direkten auslesen spezifischer Daten des Trainingsumfeldes gewonnen. Egal wie hierbei diese Informationen gewonnen werden, die Qualität und Quantität sind hier entscheidend für den Erfolg des Machine Learning Agenten.

Wie in anderen Bereichen der Informatik, wird die Qualität von Machine Learning Agenten nicht nur an der Qualität des Endergebnisses gemessen, sonder auch an der Zeit- und Speicherplatzeffektivität dieser Agenten. In Machine Learning benötigen wir jedoch die Trainingsdatenkomplexität als zusätzliches Qualitätsmerkmal, welcher die die Größe und Komplexität der benötigten Trainingsdaten beschreibt um den Agenten erfolgreich zu trainieren.

Der Einsatz von Machine Learning ist sehr weitläufig. Text- oder Dokumentenklassifikation, Verarbeitung von natürlicher Sprache, ′Computer Vision′-Applikationen, Betrugs Analyse im Online-Banking, Network Intrusion, das Lernen von Spielen wie Schach, autonomes Fahren, Unterstützung von medizinischen Diagnosen und Information Extraktion sind nur einige von unzähligen Bereichen in denen Machine Learning Algorithmen genutzt werden. Diese Nutzung lässt sich in folgende Kategorien unterteilen [15]:

- **Klassifikation**
  Ist das Problem der Kategorisierung von nicht kategorisierten Daten

- **Regression**
  Ist das Problem der Vorhersage von ′realen′ Werten für Gegenstände

- **Wertung**
  Ist das Problem des Ordnen von nicht geordneten Daten

- **Clutering**
  Ist das Problem des Ordnens nicht geordneten Daten in mehrere homogener Teilmengen

- **Dimensionsreduktion oder vielfältiges Lernen**
  Ist das Problem der Umwandlung einer initialen Repräsentation von Gegenständen in eine niedriger-dimensions Repräsentation wobei gewisse Eigenschaften der initialen Repräsentation behalten werden (z.B.: Verarbeitung Digitaler Bilder in ´Computer Vision´-Aufgaben

Das praktische Hauptziel von Machine Learning besteht aus der Erstellung von akkuraten Vorhersagen von nicht betrachteten Gegenständen/Entitäten und dem designen von effizienten und robusten Algorithmen um diese Vorhersagen zu treffen.
Um diese Ziele zu Erreichen gibt es verschiedene Machine Learning Modelle welche genutzt werden um gewisse Problemstellungen zu realisieren. [19]

- **Supervised learning**
  Hier werden vorgefertigte Datensätze verwendet um den Agenten zu trainieren. Diese Datensätze sind so sortiert und markiert, sodass der Agent anhand dieser Datensätze lernt wie er ungesehene, unbekannte Verarbeiten soll

- **Unsupervised learning**
  Hier werden (sowie bei supervised Learning) vorgefertigte Datensätze verwendet um den Agenten zu trainieren. Diese Datensätze sind nicht sortiert oder markiert und der Agent soll eigenständig diese Datensätze in Kategorien unterteilen. Dies geschieht mit Hilfe von Nachahmung der bereitgestellten Trainingsdaten

- **Semi-supervised learning**
  Hierbei handelt es sich im einen Mix aus supervised und unsupervide learning. Hier werden dem Agenten eine kleine Menge an markierten und sortierten Daten, sowie eine große Menge an unsortierten, unmarkierten Daten während des Trainings bereit gestellt. Anhand dieser Daten soll der Agent dann diese in Kategorien unterteilen.

- **Reinforcement learning**
  Hierbei wird der Agent während des Training konstant bewertet. Je nach Bewertung erkennt der Agent positive oder negative Aspekte in seinem Verhalten und passt sich so an um das positivste Verhalten zu verstärken.

Im Bereich der Spieleentwicklung und Simulation macht es hier am meisten Sinn Reinforcement Learning zu nutzen, da man nur schwer und in speziellen Fällen vorgefertigte Daten dem Agenten zur Verfügung stellen können. Das bewerten von Aktionen des Agenten in einem Spielumfeld ist hier jedoch Recht einfach und bietet sich perfekt für diese Problemstellung an.

2.3 Neuronale Netzwerke

Bevor wir zu Reinforcement Learning kommen und genauer betrachten wie dies funktioniert, sollten wir Grundlegend verstehen was Neuronale Netzwerke sind, wie diese aufgebaut sind und wie diese ´lernen´ können. Neuronale Netzwerke sind ein grundlegender Baustein von Machine Learning und werden in allen Methodiken genutzt. Es gibt dabei eine große Anzahl an verschiedenen Neuronalen Netzwerk Architekturen, welche verschiedene Eigenschaften aufweisen und in verschiedenen Situationen genutzt werden, jedoch basieren alle Architekturen auf einen prinzipiellen Aufbau. Diesen möchte Ich hier genauer erklären. In der Nachfolgenden Abbildung ist ein typisches simples Neuronales Netzwerk dargestellt.

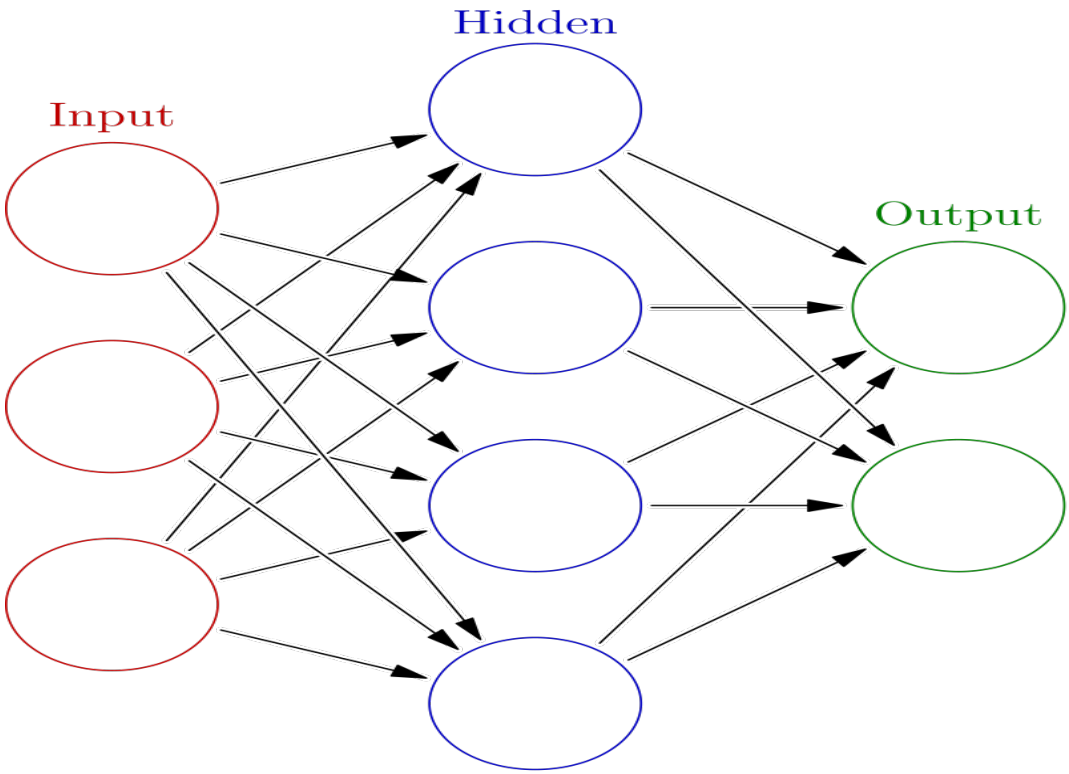

Abbildung 2.1: Grundlegender Aufbau eines Neuronalen Netzwerkes [23]

Neuronale Netzwerke sind in sogenannte Schichten (Layers) unterteilt. Diese Schichten bestehen immer aus einer definierten Anzahl an Neuronen (Neurons). Die Neuronen sind hierbei mit Neuronen aus der vorherigen, sowie der nachfolgenden Schicht verbunden (dargestellt mit Pfeilen). Diese Verbindungen geben an welche Informationen ein Neuron von den Neuronen der vorherigen Schicht

erhält und an welche Neuronen verarbeiteten Informationen an die kommenden Schicht von Neuronen weiter gegeben werden. Die Schichten des Neuronalen Netzwerkes können hierbei in 3 Teile unterteilt werden. Die Eingangsschicht (Input-Layer), Versteckte Schicht(en) (Hidden Layers) und die Ausgangsschicht (Output-Layer). Die Eingangsschicht besteht aus Neuronen, welche nur einen einzigen Wert besitzen. Dieser Wert wird entweder aus präparierten Datensätzen oder direkt aus der Umgebung ausgelesen. Die Ausgangsschicht enthält (wie die Eingangsschicht) nur Neuronen, welche einen einzigen Wert enthalten. Diese Werte werden als Informationen genutzt, wie das Neuronale Netzwerk handeln soll. Die Versteckten Schichten bestehen hierbei aus speziellen Neuronen, welche eine Funktion beinhalten. Diese Funktion errechnet anhand von den Informationen, welche das Neuron von den verbundenen Neuronen der vorherigen Schicht erhält, neue Informationen welche an die Neuronen der nachfolgenden Schicht weitergegeben werden. Dieser Kreislauf aus Informationen der vorherigen Schicht sammeln, errechnen von neuen Informationen und weiter geben der errechneten Informationen wird solange wiederholt, bis jede Schicht die Informationen verarbeitet hat und wir Werte in der Ausgangsschicht auslesen können.
Die Funktion welche die Neuronen der Versteckten Schichten nutzen sind je nach Algorithmus welcher genutzt wird unterschiedlich. Eine der beliebtesten Funktionen ist die ′Sigmoid′-Funktion. Die Informationen der vorherigen Schicht werden meist in den Neuronen zusammen addiert, dann auf die ′Sigmoid′-Funktion abgebildet. Der neu erhaltene Wert wird dann an die nächste Schicht weiter gegeben und das ganze wird wiederholt bis man an der Ausgangsschicht angelangt ist.

Dies alleine reicht jedoch noch nicht um das Neuronale Netzwerk lernfähig zu gestalten. Hierzu benötigen wir noch eine extra Komponente Namens Tendenz (Bias). Die Tendenz gibt einem Neuron der Versteckten Schicht an wie stark gewichtet diese die Informationen eines Neuron der vorherigen Schicht zu werten hat. Hierbei erhält jede Verbindung eines Neurons zur vorherigen Schicht eine eigene Tendenz. Die Tendenz besteht meist aus einem Numerischen Wert zwischen 0 und 1. Die Tendenz wird dann einfach mit dem Wert des vorherigen Neuron multipliziert und dann mithilfe der Informationen der anderen Neuronen und der beinhalteten Funktion weiter verarbeitet. An diesem Punkt schalten sich die Verschiedenen Algorithmen ein, welche genutzt werden können um Neuronale Netzwerke zu trainieren. Diese Algorithmen berechnen diese Tendenzen in regelmäßigen Abständen erneut um das Verhalten des Netzwerkes so zu ändern um eine gewisse Problemstellung zu automatisieren oder zu lösen. Je nach Algorithmus geschieht das auf unterschiedliche Art und Weise. Die Tendenzen sind dann letzten Endes das, was das Neuronale Netzwerk auszeichnet. Das Training des Netzwerkes ist prinzipiell, das entdecken und erhalten von Tendenzen, welche die Informationen der Eingangsschicht so gewichten und verarbeiten, sodass die Ausgangsschicht Informationen enthält, welche die gegebene Problemstellung lösen.

Dieser generelle Kreislauf von Informationen sammeln, gewertete Informationen mithilfe von Funktionen in neue Informationen umwandeln, Informationen weitergehen und in regelmäßigen Abstände die Tendenzen (Wertungen) überarbeiten zeichnet Machine Learning aus. Die Architektur/Algorithmen welche einzelne Neuronale Netzwerke nutzen haben hierbei nur wenig bis keine Auswirkungen auf dieses grundsätzlichen Vorgehen.

2.4 Reinforcement Maschine Learning

Reinforcement Learning (RL) ist ein Teilgebiet von Maschine Learning und somit auch ein Teilgebiet der Künstlichen Intelligenz. Beim RL werden dem Agenten üblicherweise Daten direkt aus dem Umfeld gegeben in welchem er Trainiert werden soll. Hierbei interagiert der Agent über Zeit mit dem Umfeld.

Der folgende Absatz ist eine sinngemäße Übersetzung basierend auf der Ausarbeitung 'Deep Reinforcement Learning' Kapitel 2.4.1 von Yuxi Li (2018) [9]

Nach jedem Zeitintervall t erhält der Agent einen Zustand (State) s_t und wählt eine Aktion a_t aus einem Aktionsraum (State Space) S, welche einer Regel (Policy) $\pi(a_t|s_t)$ folgt. Diese Regel spiegelt das Verhalten des Agenten wieder. Das Verhalten kann auch als eine Abbildung von Zustand s_t auf eine Aktion a_t beschrieben werden.
Der Agent erhält anschließend eine skalare Belohnung (Reward) r und geht in den nächsten Zustand s_{t+1} über.
Dies ist in folgender Abbildung zum besseren Verständnis dargestellt.

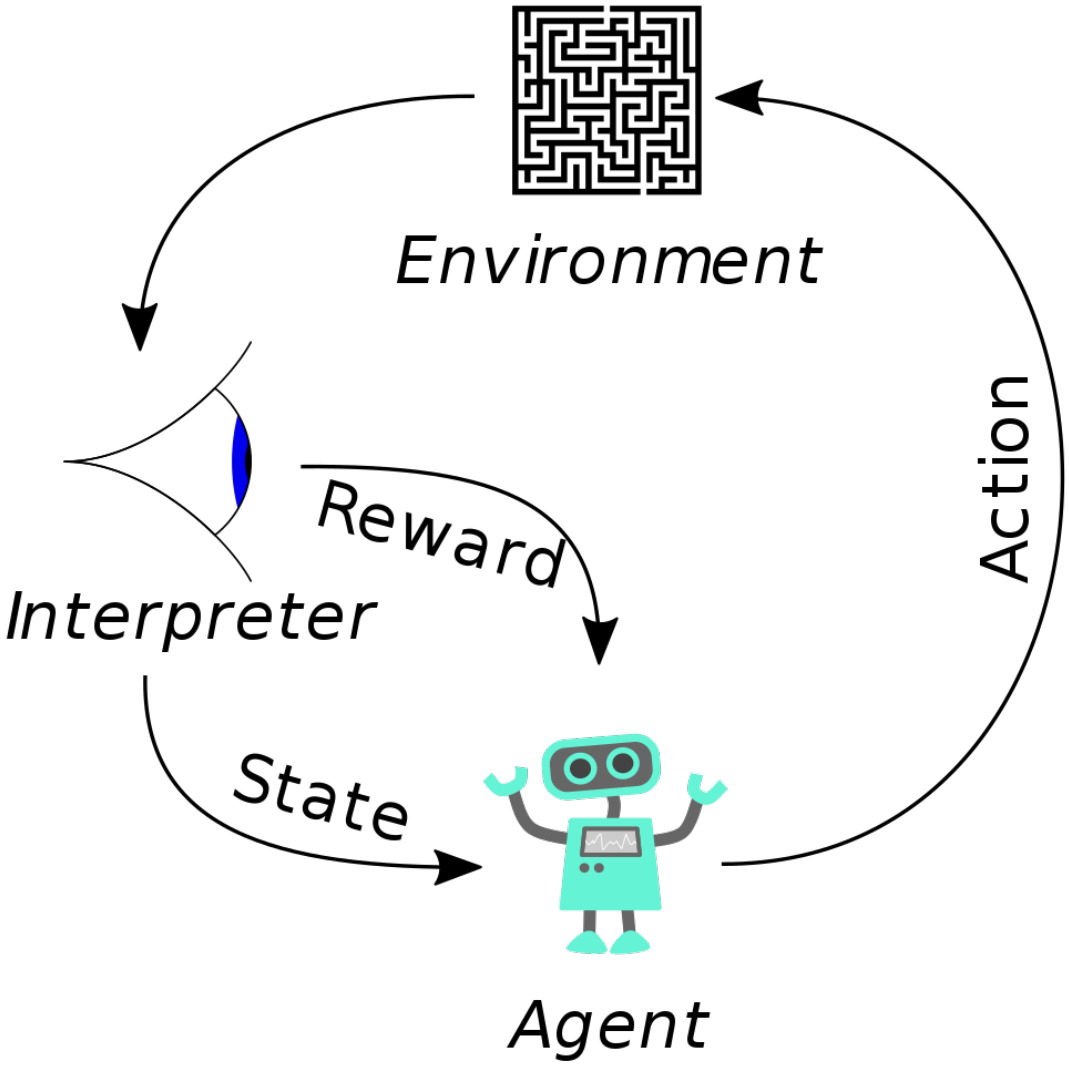

Abbildung 2.2: Reinforcement Learning Darstellung [29]

In einer episodischen Umgebung wiederholt sich dieser Vorgang solange, bis der Agent eine terminalen Zustand erreicht. Danach startet die Umgebung von Anfang an erneut.
Am ende einer Episode hat der Agent eine reduzierte, gesamt Belohnung mit einem Reduzierungsfaktor von $\gamma \in (0, 1]$. Daraus kann eine Formel zu Berechnung der Episoden-Belohnung abgeleitet werden.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Der Agent zielt darauf ab, die Erwartung einer solchen langfristigen Episoden-Belohnung von jedem Zustand zu maximieren. Diese Problemstellung ist hier in einem diskreten Zustand und Zustandsräume aufgebaut. Es ist nicht schwer diese zu erweitern um kontinuierliche Zustände zu erhalten.

2.5 Unity und ML-Agents

Unity ist eine ′Cross-Plattform Game Engine′ welche von ′Unity Technologies′ entwickelt wird. Unity biete eine Vielzahl an Features, welche dazu genutzt werden können um 2D oder 3D Umgebungen zu erstellen. Diese werden in der Industrie hauptsächlich dazu genutzt um Viedeospiele zu entwickeln, jedoch kann man mit solch eine ′Game Engine′ für Software in vielen anderen Branchen auch genutzt werden. Unity kann auch als Visualisierungstool verwendet werden um z.B. Computer assisted Design, VR-Applikationen oder AR-Applikationen interaktiv darzustellen. In unserem Fall nutzen wir Unity zum erstellen der 3D Umgebungen, aus denen wir Informationen sammeln können um unsere Agenten in diesen zu trainieren. Unity bietet uns hierbei das von ′Unity Technologies′ selbst erstelle Toolkit ′ML-Agents′ welches uns mithilfe von Python und Tensorflow ermögliche Agenten einfach und effektiv zu trainieren.

Das ′Unity Machine Learning Agents Toolkit′ (ML-Agents) ist ein ′Open-Source′ Projekt welches Spiele- und Simulationsumgebungen das Trainieren von intelligenten Agenten ermöglicht. Agenten können hierbei mithilfe von Reinforcement Learning (RL), Imitation Learning (IL), ′Neuroevolution′ oder anderen Machine Learning Methoden über eine Python API trainiert werden. Zusätzlich werden (basierend auf PyTorch) ′state-of-the-art′ implementierte Algorithmen bereit gestellt, um Entwicklern und Hobbyisten ein einfaches trainieren von intelligenten Agenten für 2D, 3D und VR/AR-Applikationen zu ermöglichen. Diese Agenten können für eine Vielzahl von verschiedenen Zwecken genutzt werden. Wie z.B. Kontrollieren von ′Non-Playable-Charachters′ (NPCs) und automatisiertes testen von Softwarebuilds.
ML-Agents nutzt hierbei 2 spezielle RL Algorithmen. Den ′Proximal Policy Optimization′ (PPO) und den ′Soft Actor Critic′ (SAC) Algorithmus. Diese werden detaillierter im folgenden Kapitel beschrieben.
Tensorflow ist eine kostenlose, ′Open-Source′ Softwarebibliothek für Machine learning und Künstliche Intelligent. Diese behandelt eine weite Vielzahl an verschiedenen Aufgaben, hat aber ein speziellen Fokus auf das Trainieren und Interferieren von neuralen Netzwerken.

2.6 PPO und SLA

Proximal Prolicy Optimization (PPO) ist ein RL Algorithmus welche ein neurales Netzwerk nutzt um die ideale Regeln/Funktionen (Policy) zu schätzen, welche die Observation eines Agenten auf die best möglichste Aktion in jeden gegebenen Zustand projiziert [1]. Dieser PPO Algorithmus ist in TensorFlow umgesetzt und läuft in einem seperaten Python Prozess (welcher über einen Socker mit Unity kommuniziert).

Soft Actor-Critic (SAC) ist ein öff-policyÄlgorithmus, welcher im Kontrast zu PPO, von vergangenen Erfahrungen lernen kann. Diese können zu egal welchem Zeitpunkt während des Training gesammelt worden sein. Während des sammeln dieser Erfahrungen werden diese ein einen Replay Buffer"gespeichert. Aus diesem wird dann während des weiteren Trainings zufällige Erfahrungen gezogen und zum trainieren genutzt. Dies macht SAC signifikant effizienter, was oftmals dazu führt, dass 5-10 mal weniger Zustände (States) benötigt werden um gleiche Lernergebnisse zu erzielen als Agenten welche mit PPO trainiert wurden. Jedoch neigt SAC dazu mehrere Modell-Aktualisierungen zu benötigen. D.h. die internen Regeln, welche der Agent nutzt müssen öfters überarbeitet werden.

Anhand dieser Vor- und Nachteile wird PPO generell genutzt, da dieser generell bessere Ergebnisse liefert. SAC kann hierbei in langsameren Umgebungen genutzt werden. PPO agiert hier am effektivsten mit Zeitintervallen (Step) von 20 Millisekunden. SAC solle erst genutzt werden wenn eine Umgebung gegeben wird in welcher ein Zeitintervall von 100ms oder mehr ausstreichend ist.

Um ein Beispiel zum besseren einschätzen zu nennen an welchem wir entscheiden welchen Algorithmus man nutzen sollten, stelle man sich ein autonom fahrendes Auto vor. Mit dem PPO Algorithmus würde das Auto alle 20 ms seine Umgebung wahrnehmen und Aktionen wie Bremsen oder Lenken anhand der Umgebung errechnen. Dies hat zur Folge, dass das Auto $(100km/h : 3,6 = 27,78m/s * 0.02s =)$**0.556 *m*** 'blind' fährt bevor es seine neue Umgebung wahrnimmt und darauf reagiert. Im Vergleich SAC welcher all 100ms ein Update erhält würde $(100km/h : 3,6 = 27,78m/s * 0.1s =)$**2,778 *m*** zwischen den Erneuerungen der Umgebung (und Aktionen) fahren.
Somit hat man eine Differenz von 2,22m zwischen den beiden Algorithmen. Im Straßenverkehr kann dies über Leben und Tot entscheiden.

Dieses Beispiel soll klar machen, dass man sich im Voraus Gedanken machen sollte wie die gegebene Umgebung aussieht und wie der Agent mit dieser Interagiert. Zudem kommt dann noch die Zeit welche zur Verfügung steht um den Agenten zu trainieren.
Wenn relativ viel Zeit zur Verfügung steht um einen Agenten zu trainieren und

ein Umfeld besitzt in welchem der Agent schnell auf Änderungen innerhalb dieses Umfeldes reagieren soll, dann sollte der Agent mit PPO trainiert werden. Wenn man jedoch nur begrenzt Zeit hat und zeitsparend einen Agenten trainieren soll, sollte man SAC verwendet werden. Hier muss dann jedoch ein Umfeld geschaffen werden in dem der Agent Spielraum hat auf Änderungen innerhalb dieses zu reagieren.

3 Installation und Erklärung von ML-Agents

3.1 Installation von ML Agents

Für die erfolgreiche Installation von ML-Agents werden folgende Softwarekomponenten benötigt:

- Unity (2020.3 oder neuer)
- ML-Agents (Kann in Unity direkt über Unity's 'PacketManager' installiert werden)
- Python (3.6.1 oder neuer)
- 'mlagents' Python Paket
- 'pytorch' Python Paket
- (OPTIONAL) Wenn zum Trainieren des Agenten eine CUDA fähige NVIDIA-GPU vorliegt kann zudem auch CUDA installiert werden. (Dies ermöglicht Tensorflow das Nutzen dieser während des Trainings)

Bevor man mit dem Installieren dieser Komponenten anfängt muss auf die Versionen-Kompatibilität dieser geachtet werden, da nur gewisse Versionen von ML-Agents mit gewissen Versionen der anderen Komponenten kompatibel ist. Wir hier nicht darauf geachtet, kann es unter Umständen sein dass die Kommunikation zwischen Unity und Python nicht funktioniert, was das Training unmöglich gestaltet.
Eine List mit welche Versionen mit welchen Kompatibel sind, kann in der offiziellen ML-Agents Dokumentation im ML-Agents Repository auf GitHub gefunden werden.

Um nun die Installation umzusetzen sollte zuerst eine Unity-Version 2020.3 oder neuer (vorzugsweise über Unity Hub) installiert werden. Danach sollte in Unity,

über den PacketManager, aus dem Unity-Registry eine Version von ML-Agents installiert werden. Anhand dieser können dann, mithilfe der GitHub-Version Dokumentation, die korrekten Versionen der Restlichen Komponenten installiert werden.

Zuerst installiert man die korrekte Python Version. Danach wird optional empfohlen, in den Ordner zu navigieren in dem das Unity Projekt erstellt wurde und eine virtuelle Python Umgebung zu erstellen, in dem man die restlichen Komponenten (Pytorch und ML-Agents Python Paket) installiert.
Dies hat den Vorteil dass alle Ergebnisse und "Gehirne"(welche nach dem Training entstehen) ohne großen Mehraufwand in dem Unity Projekt abgelegt sind. Wenn man mehrere verschiedene Projekte besitzt, sollte dies unbedingt beherzigt werden. Ansonsten kann hier schnell der Überblick verloren gehen.

Um die Installation in Unity zu verifizieren überprüfen wir in der Komponenten-Liste nach ob wir einen Reiter mit ′ML-Agents′ finden.
Zum verifizieren der Python Installationen kann der Befehl ′mlagents-learn -h′ oder ′mlagents-learn –help′ in einer Konsole ausgeführt werden. Es kann hierbei mehrere Sekunden dauern, bis ein Ergebnis in der Konsole zu sehen ist. Dieser Befehl sollte nun eine List an weiteren Befehlen liefern. Am Anfang dieser List können eventuell noch Version-Meldungen stehen. Sollten jedoch keine Meldungen vorhanden sein, ist die Installation erfolgreich und man kann mit der Training des Agenten starten.

Falls eine CUDA Kompatible Nvidia GPU vorliegt kann zusätzlich noch CUDA installiert werden. Dies ist etwas aufwendiger und erfordert eine kostenlose Anmeldung bei dem ′Nvidia Developer Program′. Eine detaillierte Installation kann in der Offiziellen ML-Agents Dokumentation unter ′Installing ML-Agents for Windows′ gefunden werden. Die Installation von CUDA ist jedoch komplett optional. ML Agents kann auch ohne diese erfolgreich Agenten trainieren.

Für diese Ausarbeitung werden folgende Versionen genutzt:

- Unity v2020.3.27f1
- ML-Agents v2.0.1
- Python v3.9.9
- PyTorch v1.11.0
- mlagents v0.20.0 (Python Paket)
- CUDA v11.0 (März 2020)

3.2 Erklärung der ML Agents Komponenten

Bevor wir mit der Implementierung unseres ersten Agenten anfangen werden hier noch diverse Begrifflichkeiten, sowie die Hauptkomponenten welche von ML-Agents während des Verlauf des Trainings genutzt werden beschrieben und erklärt.

Das Training der Agenten wird in sogenannten Schritten (Steps) durchgeführt. Mit einem solchen Schritt ist eine gewisse Zeitspanne gemeint. Diese kann manuell in den Einstellungen der Unity-Physik-Einstellungen definiert werden. Hier wird für den Schritt die ′FixedUpdate′-Zeit in den Physik-Einstellungen des Projektes definiert. Standardmäßig ist diese Zeit auf 20ms gesetzt. Während des Trainings sollte diese Zeit konstant bleiben.

Des weiteren werden Informationen, die während des Trainings aus dem Unity Umfeld gesammelt werden, an die sogenannte ′Akademie′ (Academy) gesendet. Diese organisiert alle Agenten- und Gehirn-Objekte innerhalb einer Unity Szene. Jede Szene welche einen Agenten enthält muss zwangsweise auch eine Akademie enthalten. Die Akademie übernimmt das Trainieren des Agenten, das zurücksetzen der Umgebung bei einem terminalen Zustand und die Entscheidungsfindung der Agenten.

ML-Agents kommt mit verschiedenen Komponenten und einer Bibliothek, welchen man utilisieren muss um Agenten zu erstellen, modifizieren und zu trainieren. In der Version die in dieser Ausarbeitung genutzt wird kommt ML-Agents mit 12 Komponenten (Behaviour Parameters, Buffer Sensor, Camera Sensor, Decision Requester, Demonstration Recorder, Grid Sensor, Match 3 Auctuator, Match 3 Sensor, Ray Perception Sensor 2d, Ray Perception Sensor 3D, Render Texture Sensor und Vector Sensor). 8 dieser 12 Komponenten sind jedoch sogenannte Sensoren (alle Komponenten die das Wort ′Sensor′ beinhalten), welche uns verschiedene Methoden geben um verschiedene Informationen aus dem Umfeld für die Verarbeitung zu sammeln. Einige dieser werden in den Fallbeispielen genutzt und genauer erklärt. Die anderen 4 Komponenten (Behaviour Parameters, Decision Requester, Match 3 Auctuator, Demonstration Recorder) sind wichtige Bestandteile gewisser Agenten und sind teilweise Notwendig um unsere Agenten zu erstellen und zu trainieren. Zusätzlich benötigen wir noch ein Skript (welches selbst erstellt wird) welches von der Klasse ′Agent′ aus der ′Unity.MLAgents′ Bibliothek erbt.

Die 2 wichtigsten Komponenten sind die ′Behaviour Parameters′ und das eigens erstelle Skript, da diese beiden das Herzstück jedes Agenten bilden.

Bei den ′Behaviour Parameters′ handelt es sich um diverse Einstellungen, wie unser Agent handelt, wie viele Inputs unser Neuronales Netzwerk erhält, welche Arten von Output (und die Anzahl dieser) wir fordern, welches Gehirnmodell der Agent nutzten soll (falls schon ein Verhaltensmodell vorhanden ist), und noch weitere auf die später genauer eingegangen werden. Kurz gesagt handelt es sich hier um diverse Einstellungen welche man in Unity am Agenten vornehmen kann.

Das selbst erstellte Skript, gibt uns die Möglichkeit diverse Funktionen aus der ′Agent′ Klasse zu überschreiben und eigene Implementation zu liefern welche Umgebungsabhängig gestaltet werden müssen. Dies beinhaltet Dinge wie das Zurücksetzen der Umgebung bei terminalen Zuständen, das ′Reward′-System des Agenten, oder das ′Verzufälligen′ des Umfeldes (um ′Overfitting′ zu Bekämpfen, später dazu mehr). Wie genau dieses Skript aussieht und welche Funktionen wichtig sind, wird bei dem ersten Fallbeispiel genauer erläutert.

Bei dem ′Decition Requester′ handelt es sich um eine optionale jedoch sehr empfehlenswerte Komponenten. Diese Komponenten fordert in gewissen Zeitabständen Entscheidungen von der Akademie an. Dies kann jedoch auch manuell in unseren selbst erstellten Skript mithilfe der ′RequestDecision()′-Funktion durchgeführt werden, falls notwendig.

Die Restlichen Komponenten werden genauer erläutert, wenn diese in einem Fallbeispiel genutzt werden.

4 Genereller Workflow zum erstellen eines Agenten am Fallbeispiel ′BallBalancer′

In diesem Kapitel wird das Umsetzen eines selbst trainierten Agenten mithilfe von ML Agents behandelt. Im genaueren wird beschrieben wie das generelle Vorgehen, das Erstellen des Umfeldes, das Erstellen des Agenten, die Einstellungen des neuronalen Netzwerkes und das Nutzen von Tensorflow und PyTorch gehandhabt werden. Anhand der Fallbeispiele sollen die verschiedene Arten dargestellt werden, wie Agenten Informationen aus dem Umfeld entnehmen können (Vector-basierend, Bild-basierend, ...).

Der Fokus liegt hierbei auf ML-Agents. Es wird vorausgesetzt, dass Grundkenntnisse der Unity Engine bekannt sind (GameObjects, Komponenten, Skripte, ...). Die Komponenten von ML-Agents sowie die Nutzung dieser werden jedoch genauer beschrieben und es werden keine Vorkenntnisse von ML-Agents, TensorFlow oder Machine Learning vorausgesetzt.
Anhand des Fallbeispieles ′BallBalancer′ sollen die wichtigsten Vorgehensweisen demonstriert und erläutert werden, um einen Agenten auf ein gewisses Problem zu trainieren.

4.1 Beschreibung des Problems

Bevor man sich Gedanken über irgendetwas anderes macht, muss im klaren sein wie die Problemstellung um genauen aussieht, welches der Agent automatisieren oder bewältigen soll. Es ist wichtig bei der Definition des Problems so wenig wie möglich Spielraum für Interpretationen zu lassen. Je klarer das Problem und die dazugehörigen Eigenschaften definiert werden desto einfacher und effizienter kann das Umfeld und der Agent erstellt werden. Generell gilt es folgende Fragen mit der Definition zu beantworten:

- Was soll der Agent erreichen?
- Wie soll der Agent dies erreichen?
- Hat der Agent Einschränkungen?
- Was sind die Kriterien für Erfolg/Fehlschlag?

In unserem Fallbeispiel dem ′BallBalancer′ soll ein Agent erstellt werden, welcher auf einer quadratischen Fläche einen runden Ball mit bestimmter Größe so lange wie möglich balancieren soll (Was?). Der Agent darf hierbei nur die horizontale Rotation (X- und Z-Achse) beeinflussen (Wie?). Die Position der Fläche ist fest gesetzt und kann nicht geändert werden (Einschränkungen?). Das einzigste Bewertungsmerkmal ist die Zeit (Von Start bis zum ′Fallen lassen′ des Balles) welche der Agent den Ball erfolgreich balancieren kann. Sobald sich die Position des Balls unterhalb der Fläche befindet, gilt der Ball als ′Fallen gelassen′. Andere Bewertungsmerkmal gibt es nicht (Erfolg/Fehlschlag?). Um das Problem so simpel wie möglich zu halten, ändern sich die Eigenschaften der Fläche (z.B. Größe oder Form), sowie des Balls (z.B: Masse oder Startpunkt) nicht.

4.2 Erstellen des Umfeldes

Da wir jetzt genau wissen was die Problemstellung ist, können wir anfangen die Umgebung in Unity zu erstellen, in der unser Agent trainiert werden soll. Diese ist in unseren Beispiel recht einfach. Diese besteht nur aus 2 Objekten. Einer Fläche und einem Ball. Wichtig hier ist es dass eines der beiden Objekten eine ′Rigidbody′ Komponente und beide Objekte einen ′Collider′ besitzen. Dies ermöglicht es der Fläche den Ball mithilfe der Unity-Physik-Engine zu balancieren. Der Ball sollte hierbei zentral über der Fläche positioniert sein und er sollte kleiner als die Fläche sein.

Im Anhang unter den Abbildungen 16 - 18 sind Darstellungen zu finden welche den genaueren Aufbau der Umgebung, des Balles und des Agenten abbilden.

4.3 Erstellen des Agenten

Nach dem erstellen des Umfeldes kommen wir zum erstellen unseres Agenten. Hierzu sollte ein Agentenobjekt gewählt werden in welchem alle wichtigen Komponenten implementiert sind, welche dem Agenten seine Funktion ermöglichen. Das Agentenobjekt, kann hierbei theoretisch frei gewählt werden. Um jedoch gut strukturiert und logisch zu arbeiten ist es empfehlenswert ein Objekt aus der Umgebung/Szene zu wählen welches mit den meisten (oder allen) Erfolgsfaktoren direkt interagiert. Alle Objekte die als Erfolgsfaktor oder als Hilfsobjekt in der Umgebung dienen sind hierbei eine schlechte Wahl für das Agentenobjekt.
Um ein geeignetes Agentenobjekt zu wählen gibt es 2 einfache Möglichkeiten:

- Das Systematisches ausschließen von Objekten, bis nur noch eine gute Wahl übrig bleibt.

- Suchen des Objektes, welches mit den meisten (oder am besten allen) Erfolgskriterien direkt interagiert.

In unserem Beispiel können wir also entweder den Ball als Agenten ausschließen, da dieser ein Erfolgskriterium darstellt. (Somit bleibt nur noch die Fläche als Agentenobjekt übrig.) Oder wir Fragen uns welches Objekt am meisten mit dem Ball interagiert. Es wird schnell klar, dass die Fläche am meisten mit dem Ball interagiert. Somit ist die Fläche eine gute Wahl für die Repräsentation unseres Agenten.

Ist das Agentenobjekt gewählt gilt es die die korrekten Komponenten in dem Objekt zu implementieren. Hierzu muss sich zuerst gefragt werden, welche Informationen unser Agent aus der Umgebung benötigt um das bestehende Problem erfolgreich lösen zu können. Eine gute Hilfe um alle benötigten Informationen zu finden ist zu Fragen, was man an Informationen benötigt um das Problem selbst zu lösen (also was für Informationen benötige Ich wenn Ich dieser Agent in diesem Umfeld wäre). Hierzu gibt es meist mehrere Lösungen die zum Erfolg frühen und hier kann, je nach zur Verfügung stehenden Zeit, frei ausgewählt und getestet werden. Generell gilt aber so viel wie nötig, so wenig wie möglich. Während der Umsetzung der Fallbeispiele dieser Ausarbeitung wurde schnell klar, dass die Anzahl der zu verarbeitenden Daten (Inputs) in direkter Korrelation mit der daraus folgenden Trainingsdauer steht.

ML-Agents bietet uns eine viel zahl an verschiedenen Sensoren an, welche Informationen aus dem Umfeld lesen und es dem Agenten zur Verarbeitung zur Verfügung stellen. Später in der Ausarbeitung werden die häufigst genutzten Sensoren in anderen Fallbeispielen erläutert und demonstriert. Um das Fallbeispiel

'BallBalancer' so simpel wie möglich zu halten werden wir hier keines dieser Sensoren nutzen. Wir lesen mithilfe von etwas Code und einem Skript selbst direkt Informationen aus der Umgebung. Bei komplexeren Problemen wird meist eine Mischung aus Sensoren und dem selbstständigen Auslesen genutzt.

Um auf unser Beispiel nun zurück zu kommen, könnten wir z.B. dem Agenten die Position, Rotation oder die Geschwindigkeit des Balles (jeweils mit einem Vector3 repräsentiert) übergeben. Alle Inputs werden zu guten Ergebnissen führen. Da wir mehrere der selben Agenten in der Szene haben werden (um schneller zu trainieren), nutzen wir die Rotation des Balls, da diese immer gleich und nicht Positionsabhängig ist.

Nun da alle benötigten Informationen erfasst sind, können wir die Implementierung der Komponenten starten. Als erstes erstellen wir ein neues C#-Skript mit einem beliebigen Namen. In dem Beispiel nennen wir es 'BallBalancerController', da dieses Skript später mit der Akademie kommuniziert um Input an diese weiter zu geben und Entscheidungen dieser zu verarbeiten. Das neu erstellte Skript erbt momentan von der 'MonoBehaviour' Klasse der 'UnityEngine'-Bibliothek. Dies muss auf die Klasse 'Agent' aus der 'Unity.MLAgents'-Bibliothek geändert werden. Funktionalitäten der 'MonoBehaviour' Klasse sind immer noch gegeben da 'Agent' ein Kind von 'MonoBehaviour' ist. Das erben von dieser 'Agenten' Klasse indiziert, dass es sich hierbei um ein Agentenobjekt handelt. Dies ist notwendig, da andere Komponenten dieses zwingend benötigen um zu funktionieren. Der Kopf des 'BallBalancerController'-Skriptes sollte dann wie folgt aussehen:

```
using System;
using Unity.MLAgents;
using UnityEngine;

public class BallBalancerController : Agent
{
...
}
```

Die Klasse 'Agent' ermöglicht das überschreiben vieler elementarer Funktionen, welche uns ermöglicht selbst definierte Agenten zu erstellen. Die 5 meist genutzten Funktionen welche überschrieben werden sollten sind:

- **void Initialize()**
  - Diese Funktion funktioniert wie die 'Start()' Funktion der 'MonoBehaviour'-Klasse und wird aufgerufen, wenn der Agent initialisiert wird.
- **void CollectObservations(VectorSensor sensor)**

  - Diese Funktion wird einmal pro Step aufgerufen und sammelt Informationen in einem ′VectorSensor′, welcher an die Akademie weiter gegeben wird um Entscheidungen zu fällen. Mithilfe der ′AddObservation()′ Funktion können dem ′Sensor′ Int32, Singel, Vector3, Vector2, Quaternion, Boolean und IEnumerable<Singel> Informationen hinzugefügt werden.

- **void OnActionReceived(ActionBuffers action)**

  - Diese Funktion wird einmal pro Step aufgerufen. Der ′ActionBuffer′ enthält die Aktionen welche vom Neuronalen Netzwerk anhand der Informationen (welche in ′CollectObservations′ gesammelt wurden) errechnet wurden. Mithilfe dieser Aktionen kann das Verhalten des Agenten implementiert werden.

- **void Heuristic(in ActionBuffers action)**

  - Diese Funktion wird, wenn es von Nutzer gewünscht ist oder wenn keine Akademie gefunden werden kann und der Agent kein Gehirn-Modell besitzt genutzt. Diese ersetzt die Ergebnisse der Akademie. D.h. hier kann der Output des Neuronalen Netzwerkes simuliert werden in dem der ′ActionBuffer′ mit Information gefüllt wird. Mithilfe dieser Funktion kann der Agent z.B. zum Testen der Umgebung direkt mit Nutzereingaben gesteuert werden.

- **void OnEpisodeBegin()**

  - Diese Funktion wird am Anfang einer Episode einmal aufgerufen und wird meist zum zurücksetzen der Agenten genutzt, wenn der Agent einen terminalen Zustand erreichen.

Das überschreiben dieser Funktionen ermöglicht es uns den Agenten auf unser Umfeld einzustellen. D.h. Die von uns erstellten Implementationen müssen so gewählt sein, dass der Agent z.B. alle benötigten Informationen in den ′CollectObservations′ Funktion dem ′sensor′ hinzufügt oder die in der ′OnActionsReceived′ Funktion erhaltenen Informationen nutzt um sich selbst zu steuern.
In unserem Fallbeispiel wird dies mithilfe der folgenden Code-Stücken genauer erklärt. Hierbei überschreiben wir die folgenden Funktionen der ′Agent′-Klasse auf folgende Art und Weise:

′Initialize′: Hier wird die Ausgangsstellung der Umgebung hergestellt. D.h. der Ball und die Fläche werden beide auf Ausgangsstellung zurück gesetzt

```
public override void Initialize()
{
    //Reset the Ball
    ballToBalance.transform.position = transform.position + (Vector3.up *
        3f);
    ballToBalance.transform.rotation = Quaternion.identity;
    ballRB.velocity = Vector3.zero;

    //Reset the Plane
    transform.rotation = Quaternion.identity;
}
```

'CollectObservations': Hier wird die Geschwindigkeit des Balls in den 'sensor' geladen um diesen an die Akademie weiter zu geben

```
public override void CollectObservations(VectorSensor sensor)
{
    base.CollectObservations(sensor);
    sensor.AddObservation(ballToBalanceRigidbody.velocity);
}
```

'OnActionReceived': Hier werden die Kontinuierlichen Aktionen (siehe 'Behaviour Parameters') welches der Agent von der Akademie erhält genutzt, um die Rotation der Fläche zu steuern. Wir limitieren hierbei die maximale Rotation der Fläche von -25 Grad bis 25 Grad in der x/z-Achse. Hierbei darf der Agent die Fläche von -2 Grad bis 2 Grad in jeweils die x- und z-Achse pro Step rotieren. Zudem, wenn der Agent den Ball fallen gelassen hat, enden wir die momentane Episode und starten einen neuen Trainings versuch.

```
public override void OnActionReceived(ActionBuffers action)
{
    //Errechnung einer zahl zwischen -2 Grad und 2 Grad
    var zAngle = 2f * Mathf.Clamp(action.ContinuousActions[0], -1f, 1f);
    var xAngle = 2f * Mathf.Clamp(action.ContinuousActions[1], -1f, 1f);

    //Limit Rotation von -25 Grad bis 25 Grad (z-Achse)
    if ((gameObject.transform.rotation.z < 0.25f && zAngle > 0f) ||
        (gameObject.transform.rotation.z > -0.25f && zAngle < 0f))
    {
        //Rotiere die Fläche um zAngle auf der z-Achse
        gameObject.transform.Rotate(Vector3.forward, zAngle);
    }

    //Limit Rotation von -25 Grad bis 25 Grad (x-Achse)
```

```
    if ((gameObject.transform.rotation.x < 0.25f && xAngle > 0f) ||
        (gameObject.transform.rotation.x > -0.25f && xAngle < 0f))
    {
        //Rotiere die Fläche um xAngle auf der x-Achse
        gameObject.transform.Rotate(Vector3.right, xAngle);
    }

    //Check if the Agent let the ball fall
    if (ballRB.transform.position.y - transform.position.y < -3f)
    {
        //If so End Episode and reset
        EndEpisode();
    }
}
```

'Heuristic': Da unser Problem so simpel ist, ist die Steuerung durch den Nutzer nicht nötig. Deswegen gibt es hier keine Implementation im Fallbeispiel.

'OnEpisodeBegin': Hier werden wie in 'Initialize' der Ball und die Fläche auf die Ausgansstellung zurück gesetzt.

```
public override void OnEpisodeBegin()
{
    //Reset the Ball
    ballToBalance.transform.position = transform.position + (Vector3.up *
        3f);
    ballToBalance.transform.rotation = Quaternion.identity;
    ballRB.velocity = Vector3.zero;

    //Reset the Plane
    transform.rotation = Quaternion.identity;
}
```

Dies ist vorerst alles, was innerhalb des selbst erstellten Skriptes Implementiert werden muss. Um das Agentenobjekt zu vollenden fehlen jedoch noch 2 weitere Komponenten. Diese Komponenten werden genutzt um diverse Einstellungen an unserem Agenten vorzunehmen.

Das erste der beiden ist der sogenannte 'Decision Requester'. Hierbei handelt es sich um eine 'Quality-of-Life' Komponente. In dieser Komponente könne wir einstellen in welchen Zeitintervallen der Agent Aktionen von der Akademie anfordert. Dies kann mit Hilfe des Sliders 'Decision Periode' eingestellt werden. Der Slider reicht hierbei von 1 bis 20, was die Anzahl der Steps darstellt, welche vergehen müssen bis der Agent erneut Aktionen von der Akademie anfordert. D.h.

eine Einstellung des Sliders auf 10 Steps bedeutet im Klartext, der Agent erhält einmal alle 10 Steps (Standardmäßig 20ms * 10 = 200ms) einen neuen ActionBuffer mit neuen Aktionen. Die Check-Box ′Take Actions Between Decisions′, welche sich direkt unter dem Slider befindet, sagt hierbei aus, ob der Agent nur dann Aktionen tätigen soll, wenn ein neuer ActionBuffer aus der Akademie erhalten wird, oder ob er jeden Step mit dem momentanen ActionBuffer Aktionen ausführen soll. Einfach gesagt heißt dies, wenn die Checkbox angehagt ist, der Agent führt die ′OnActionsRecieved′ Funktion jeden Step aus. Wenn die Checkbox nicht angehagt ist, führ der Agent nur dann die ′OnActionRecieved′ Funktion aus, wenn ein neuer ActionBuffer von der Akademie erhalten wurde. Diese Komponente ist jedoch rein Optional. Man kann manuell mit der ′RequestDecision′-Funktion einen neuen ActionBuffer von der Akademie Anfordern.

Die zweite wichtige Komponente ist die ′Behaviours Parameter′ Komponente. Hier werden diverse Einstellungen getätigt, welche Informationen über den Agenten enthalten. Folgend ist eine Liste mit den Einstellungsmöglichkeiten und einer kurzen Beschreibung dieser zu finden:

- **Behaviour Name**
  - Der Name des Verhaltens. Agenten welche den gleichen Verhaltensnamen besitzen, teilen ihren Lernfortschritt in einem gemeinsamen ′Gehirn′.
- **Space Size**
  - Die Anzahl der zu erwartenden Inputs. Wichtig hierbei, es wird die Gesamtsumme einzelner Zahlen erwartet. D.h. Das einlesen eines Vector3 benötigt eine Space Size von 3 (2 Vector3 benötigen 6, usw.). Wenn dieser Wert nicht mit der Anzahl der eingelesenen Informationen übereinstimmt liefert MLAgents eine Fehlermeldung.
- **Stacked Vectors**
  - Diese Zahl gibt an, ob es sich bei den einzulesenden Informationen um überlagerte Vektoren handelt um z.B. die Geschwindigkeit eines Objektes über mehrere Steps besser zu verfolgen. Wenn diese Zahl > 1 ist, werden ältere einzulesende Informationen gespeichert und in zukünftigen Steps der Akademie zur Verfügung gestellt. (Einfach gesagt, simuliert diese Einstellung Erinnerungsvermögen).
- **Continous Actions**

  - Eine ′Continous Action′ beschreibt eine rationale Zahl zwischen -1 und 1 (beide inklusive). Diese Einstellung definiert wie viele dieser ′Continous Actions′ der Agent von der Akademie erwartet. Diese werden dann später zum steuern des Agenten genutzt.

- **Discrete Branches**

  - Eine ′Discrete Action′ beschreibt eine Ganze Zahl zwischen 0 und dem eingestellten Wert des Branches. Wenn diese Einstellung > 0 ist, kann man mithilfe der erscheinenden Liste eine Reihe von ganzen Zahlen definieren, welcher der Agent von der Akademie im ActionBuffer erhält. Der Wert der einzelnen ′Discrete Actions′ kann innerhalb der Liste definiert werden. D.h. wenn ′Branch 0 Size′ auf z.B. 5 gesetzt ist, erhält er Agent eine ′Discrete Action′ (Ganz zahl) zwischen 0 (inklusive) und 5 (exklusive)

- **Model**

  - Das Modell, welches der Agent nutzten soll um aus Informationen Aktionen zu berechnen. Einfach Formuliert wird hier das ′Gehirnmodell′ definiert (falls vorhanden), welches aus vorherigen Trainingsläufen erstellt wurde. Wenn kein Modell definiert ist, erstellt der Agent ein neues.

- **Inference Device**

  - Hier wird definiert welches Gerät zur Verarbeitung des oben definierten Modells genutzt wird. Generell gilt hier dass CPU schneller funktioniert als GPU. Die GPU sollte hier nur verwendet werden wenn ein ′ResNet Visual encoder′ oder eine große Anzahl von Agenten mit visuellen Beobachtungen genutzt wird.

- **Behaviour Type**

  - Hier wird angegeben, ob der Agent mithilfe des Neuronalen Netzwerkes, oder der implementierten ′Heuristic′ Funktion gesteuert werden soll. Der Standardwert versucht zuerst eine Verbindung zu einem Neuronalen Netzwerk herzustellen (über Tensorflow oder das oben definierte Modell). Wenn dies nicht gelingt greift der Agent auf die ′Heuristic′ Funktion zurück.

- **Team ID**

  - Die ID des Teams welcher der Agent angehört. Mithilfe dieser können verschiedene Teams innerhalb einer Szene simuliert werden.

- **Use Child Sensors**
  - Besage ob der Agent die Informationen von Sensor-Komponenten mit einbeziehen soll, welche im Agentenobjekt enthalten sind. Die Anzahl der Informationen die diese Sensoren liefern muss nicht in der ´Space Size´ Einstellung berücksichtigt werden.
- **Observable Attribute Handling**
  - MLAgents bietet die Möglichkeit Attribute als ´Observable´ zu markieren. D.h. diese Attribute können automatisch als zu verarbeitende Informationen eingelesen werden. Dieses Attribut definiert ob und welche ´Observable´ Attribute mit einbezogen werden und welche nicht. Die ´Ignore´ Einstellung ist für unsere Fallbeispiele ausreichend.

Nun, da die einzelnen Bestandteile der Behaviour Parameter erklärt wurden, kommen wir auf unser Fallbeispiel des BallBalancer zurück. Die folgende Grafik zeigt die Behaviour Parameter Komponente des Agenten im Inspektor.

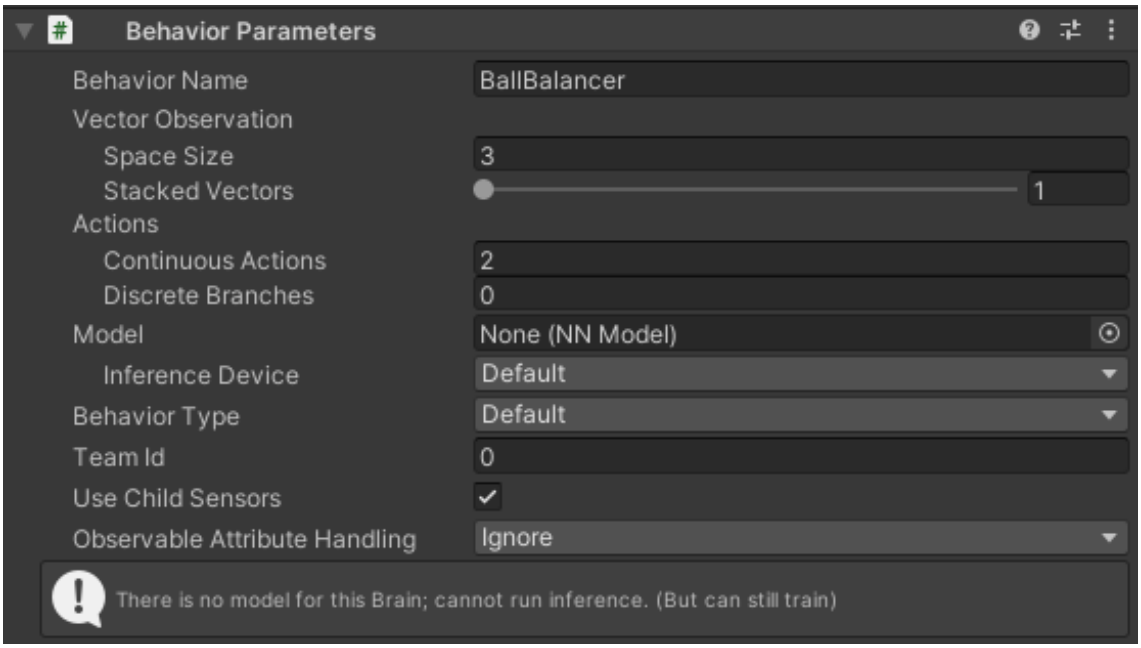

Abbildung 4.1: Behaviour Parameters des BallBalancers

Wie beschrieben benötigt unser BallBalancer 3 Eingaben (die Beschleunigung des Balles, welche aus einem Vector3 besteht) und 2 ausgaben. Diese Ausgaben sind im Continous Actionspace". Wie im ÖnActionReceived"beschrieben nutzt der Agent diese Ausgaben, um maximal von -2 Grad bis 2 Grad pro Step zu rotieren.

Diskrete Actions benötigen wir hier nicht. Der einfachheitshalber (und da dies nicht wirklich von Nöten ist um einen guten Agenten zu erzeugen) stellen wir die SStacked Vectorsäuf 1. Dies bedeutet, dass der Agent keinerlei Eingaben aus vorherigen Steps erhält, sondern immer nur die Aktuelle Umgebung beachtet. Das Modell ist nicht definiert, da wir momentan keines Trainiert haben. Und die restlichen Einstellungen sind bei unserem Fallbeispiel zu vernachlässigen.

Nachdem wir die Behaviour Parameter eingestellt haben sind wird mit dem Erstellen des Agenten fertig.

4.4 Zuweisen von Belohnungen

Nun kommt einer der wichtigsten Schritte welche zur erfolgreichen Erstellung eines Agenten von Nöten ist. Die Erstellung/Zuweisung von Belohnungen (Rewards) welcher der Agent während des Trainings erhält.

Wie in den Grundlagen erklärt, erkennt der Agent anhand dieser Belohnungen während des Trainings, ob sein momentanes vorgehen positiv oder negativ ist. Der Agent versucht hier am Ende jeder Episode eine höchstmögliche Belohnungssumme zu erreichen.

Es sei von Anfang an gesagt, dass hier viel ´Trial&Error´ betrieben werden muss, um sehr gute bis perfekte Reward- Funktionen zu erzeugen. Wenn man die Belohnungen nur sehr sparsam vergibt, kann es sein dass der Agent Vorgehensweisen entwickelt, welche für uns Menschen als unkonventionell wirken mögen. Dies kann natürlich erwünscht sein, um z.B. neue Wege zu finden um Aufgaben zu lösen, jedoch sind solche Verhaltensweisen meist unerwünscht. Generell gilt bei den Rewards, dass hier nicht die perfekte Funktion gefunden werden muss. Es gibt hier viele verschiedene Wege die alle zu ähnlichen Zielen führen können. Wenn man während des Training auf Komplikationen stößt, bietet sich die Belohnungsfunktion meist als guter Startpunkt an.

Bei der Umsetzung der Rewards-Systeme wurden nach viel ´Trail&Error´ 2 Vorgehensweisen gefunden, welche sich als zielführender erwiesen haben als andere. Diese sind wie folgt:

- Sehr häufiges (z.B. Step-Intervall) kleine gleichbleibende positive Belohnungen bei richtigen"Verhalten dem Agenten zuweisen. Vergleichsweise selten, gleichbleibende große negative "Belohnungen"bei falschen"Verhalten dem Agenten zuweisen.

- Sehr häufig (z.B. Step-Intervall) kleine, gleichbleibende negative "Belohnungen"bei falschen"Verhalten dem Agenten zuweisen. Vergleichsweise selten, wachsende große positive Belohnungen bei richtigen"Verhalten dem Agenten zuweisen. (Z.B. Erste positive Belohnung = 1, Zweite positive Belohnung = 2, Dritte positive Belohnung = 4, ...)

Welches dieser beiden Verfahren genutzt werden sollte hängt von der Umgebung ab, in dem der Agent trainiert werden soll. Hierbei eignet sich das erste Verfahren meist dann, wenn der Agent ein theoretisch zeitlich unbegrenztes Ziel hat (wie z.B. im ′BallBalancer′). Das zweite Vorgehen eignet sich dann wenn der Agent ein zeitkritisches Ziel erreichen soll. D.h. wenn er etwas so schnell wie möglich machen sollte. Hierbei kann man den Agenten dann pro Step (also zeitlich) negativ Belohnen und vergrößernde positive Belohnungen zuweisen wenn er ′Checkpoints′ zum finalen Ziel erreicht. Dies ist genauer im Fallbeispiel ′RaceCar′ erklärt.

Warum die erste Vorgehensweise funktioniert möchte Ich anhand des BallBalancers erklären. Der Agent kann, da er in sehr kleinen regelmäßigen Abständen (z.B. Step) positive Belohnung erhält, guten von schlechten Trainings versuche eindeutig unterscheiden. Der Agent erhält momentan 0.1 Punkt pro Step solange er den Ball Balanciert. D.h. wenn der Agent den Ball für 2 Sekunden Balanciert erhält er 10 Punkte, wenn er den Ball jedoch nur 1.5 Sekunden balanciert erhält der Agent nur 7.5 Punkte. Daher realisiert der Agent welcher Versuch besser war. Würde der Agent nur in großen regelmäßigen Abständen (z.B. Sekundentakt) erhalten würde er keinen unterschied zwischen z.B. 1.1 Sekunden und 1.9 Sekunden erkennen, obwohl der 1.9 Sekunden versuch eindeutig besser ist. Deshalb ist es immer wichtig dem Agenten seine Belohnungen in relativ kleinen Zeitabständen zu vermitteln. Der Abzug von einem fixen Betrag am Ende ist nicht unbedingt nötig, jedoch hilf dies dem Agenten gewisse Zustände zu vermeiden. Wenn der Agent verschiedene terminale Zustände erreichen kann, hilft es, den Agenten je nach terminalen Zustand unterschiedliche fix negativ zu Belohnen, sodass eine Tendenz des Agenten besteht gewisse Zustände zu vermeiden.

Um nun auf unser Fallbeispiel des ′BallBalancers′ zurück zu kommen und um den Agenten endlich zu finalisieren müssen wir noch 2 kleine Änderungen im ÄgentenSkript (BallBalancerController) vornehmen. Wir müssen uns hier entscheiden wie wir unseren Agenten Belohnen. Wir nutzen hierzu die erste Vorgehensweise und geben dem Agenten relativ oft kleine Belohnungen (solange dieser den Ball balanciert) und geben ihm eine vergleichsweise große negative Belohnung, wenn dieser den Ball ′fallen′ lässt. Bei uns (und generell) eignet sich hier die ÖnActionReceivedMethode, da diese einmal pro Step ausgeführt wird und die Aktionen der Akademie verarbeitet. Hier werden wir am Ende (nachdem der Agent die Aktionen aus der Akademie verarbeitet hat) der Funktion die Belohnungen zuweisen.

Hier, fügen wir mithilfe der 'SetReward()'-Funktion an 2 stellen Belohnungen hinzu. Einmal an der Stelle, an der der Agent den Ball fallen gelassen hat (Also dort, wo wir momentan den Ball auch zurücksetzen). Hier setzen wir eine negative Belohnung von '-2f'. Die zweite Stelle an der wir positive Belohnung zuweisen ist am Ende der Funktion, wenn der Ball nicht zurück gesetzt wurde. Hie weisen wir eine Belohnung von '0.1f' zu. Wenn wir alle Belohnungen eingefügt haben sieht die komplette Funktion wie folgt aus:

```
public override void OnActionReceived(ActionBuffers action)
{
    //Errechnung einer zahl zwischen -2 Grad und 2 Grad
    var zAngle = 2f * Mathf.Clamp(action.ContinuousActions[0], -1f, 1f);
    var xAngle = 2f * Mathf.Clamp(action.ContinuousActions[1], -1f, 1f);

    //Limit Rotation von -25 Grad bis 25 Grad (z-Achse)
    if ((gameObject.transform.rotation.z < 0.25f && zAngle > 0f) ||
        (gameObject.transform.rotation.z > -0.25f && zAngle < 0f))
    {
        //Rotiere die Fläche um zAngle auf der z-Achse
        gameObject.transform.Rotate(Vector3.forward, zAngle);
    }

    //Limit Rotation von -25 Grad bis 25 Grad (x-Achse)
    if ((gameObject.transform.rotation.x < 0.25f && xAngle > 0f) ||
        (gameObject.transform.rotation.x > -0.25f && xAngle < 0f))
    {
        //Rotiere die Fläche um xAngle auf der x-Achse
        gameObject.transform.Rotate(Vector3.right, xAngle);
    }

    //Check if the Agent let the ball fall
    if (ballRB.transform.position.y - transform.position.y < -3f)
    {
        //If so set negative Reward, End Episode and reset
        SetReward(-2f);
        EndEpisode();
    }
    else
    {
        //If still balancing, set positive Reward
        SetReward(0.1f);
    }
}
```

Wenn wir die 2 Zeilen in das Skript eingefügt haben, sind wir mit dem Erstel-

len/Konfigurieren des Umfeldes und des Agenten fertig und können uns um das Trainieren des Agenten kümmern.

4.5 Hyperparameter und ′Good-Practises′

Bevor wir jedoch den Agenten tatsächlich Trainieren können müssen wir definieren, wie genau unser Agent trainiert werden soll beziehungsweise wie dessen Neuronales Netzwerk aussehen soll. Tensorflow benötigt diese Informationen, sodass der Agent trainiert werden kann. Hierzu müssen wir eine YAML-Datei erzeugen in der wir verschiedene Parameter (Hyperparamter) definieren müssen. Ähnlich wie beider Belohnungsfunktion kann hier etwas ′Trail&Error′ von Nöten sein, bis man Einstellungen gefunden hat die gute Ergebnisse liefern. Die Menge an ′Trial&Error′ hängt hier hauptsächlich von der Komplexität der Problemstellung an.

ML-Agents und Tensorflow bietet uns hier eine Vielzahl an Einstellungsmöglichkeiten. Dabei gehen wir nicht auf alle Möglichkeiten ein, sondern werde nur die wichtigsten aufzählen, beschreiben und ′Good- Practises′ dieser nennen. Anschließend werden wir die YAML-Datei betrachten, welche zum Training des Agenten genutzt wurde.

- **Trainer Typ (trainer$_type$)**

    - Gibt an ob der Agent mithilfe des PPO- oder SAC-Algorithmus trainiert werden soll. In der Ausarbeitung werden wir jedoch nur den PPO-Algorithmus verwenden.

    - Akzeptierte Werte: ′ppo′ oder ′sac′

Gamma (gamma)

- Gibt an wie stark der Agent Belohnungen in der Zukunft werten soll. In Situationen bei denen der Agent eher in der Gegenwart Aktionen tätigen soll um sich auf die entfernte Zukunft vorzubereiten sollte dieser Wert groß sein. In Situationen bei denen eher nur an die Gegenwart gedacht werden soll, sollte dieser klein ausfallen.

- Typischer Wert: 0.8 - 0.995

Lambda (lambd)

- Wird genutzt um den ′Generalized Advantage Estimate′ (GAE) zu berechnen. Dieser gibt an wie Start der Agent auf momentane Umgebungswerte beruht um zukünftige Umgebungswerte zu vorhersagen. Bei niedrigen Werten beruht der Agent mehr auf die momentane Umgebungswerte (kann Tendenzen zu gewissen Vorgehensweisen bilden), bei höheren Werten beruht der Agent mehr auf Belohnungswerte welche er erhält (diese sind sehr unterschiedlich, hohes Chaos"beim Agenten). Dieser Parameter bietet hierbei eine Balance zwischen Tendenzbildung und Chaos.

- Typischer Wert: 0.9 - 0.95

Buffer Size (buffer_size)

- Definiert wie viele Erfahrungen (Agenten Beobachtungen, Aktionen und Belohnungen erhalten) der Agent sammeln soll, bevor Tensorflow Aktualisierung des Agenten-Modell (Gehirn) vornehmen soll. **Dies sollte eine multiplikatives von ′Batch Size′ sein.**. Größere Werte haben generell zu stabileren Training/Modell Aktualisierungen.

- Typischer Wert: 2048 - 409600

Batch Size (batch_size)

- Die Batch Size gibt an wie viele Erfahrungen in einer Iteration der ′gradient decent′ Aktualisierung genutzt werden soll. Diese ′gradient decent′ Aktualisierung versucht die verwendeten Funktionen des Modells/Gehirns zu minimieren. **Diese sollte immer ein Bruchteil der ′Buffer Size′ sein.** In einem ′Continous Actionspace′ sollte hier ein sehr großer Wert (>1000) vorliegen. In einem ′Discrete Actionspace′ sollte kein kleiner Wert (vielfaches von 10) vorliegen.

- Typischer Wert (vorwiegend Continous Actions): 512 - 5120

- Typischer Wert (vorwiegend Discrete Actions): 32 - 512

Number of Epochs (num_epoch)

- Wird während des gradient decent genutzt um zu bestimmen wie oft der Erfahrungs-Buffer genutzt wird. Je größer die ′Batch Size′ desto größer kann diese Einstellung gesetzt werden. Kleine Werte versichern hierbei stabilere Modell-Aktualisierungen auf Kosten der Lerngeschwindigkeit.

- Typischer Wert: 3 - 10

Learning Rate (learning_rate)

- Bezieht auf die Stärke jedes ′gradient decent′ Aktualisierungsschrittes. Dieser Wert sollte typischerweise reduziert werden, wenn das Training instabil verläuft und die Belohnungssummen sich nicht konsistent verbessern.

- Typischer Wert: 1e-5 - 1e-3

Time Horizon (time_horizon)

- Gibt an wie viele Steps von Erfahrungen (pro Agent) gesammelt werden sollen, bevor diese dem Erfahrungsbuffer hinzugefügt werden. Dieser Parameter bietet einen Kompromiss zwischen einer geringen Tendenz, dafür aber sehr unterschiedliches Verhalten (bei großen Werten) oder hohen Tendenzen zu gewissen Aktionen, dafür aber eine sehr gleichbleibendes Verhalten (bei kleinen Werten). Bei Umfeldern in denen der Agent häufig Belohnungen innerhalb einer Episode bekommt oder bei denen Episodenlängen sehr lange ausfallen, erweisen sich kleinere Werte als idealer.

- Typischer Wert: 32 - 2048

Max Steps (max_steps)

- Gibt an wie viele Steps während des Trainings maximal ausgeführt werden. Das Training wird nach erreichen dieser Steps unterbrochen. Dieser Wert solle bei komplexeren Problemstellungen größer ausfallen.

- Typischer Wert: 5e5 - 1e7

Beta (beta)

- Gibt die stärke der ′Entropy Regulation′ an. Diese erstellt die Regeln (Policies) des Modells ′mehr Zufällig′. Dies versichert, dass der Agent genügend Aktionsraum während des Trainings besitzt um eventuell neue oder bessere Wege zu finden das Problem zu lösen. Je größer diese Zahl ausfällt, desto ′zufälliger′ sind die Verhaltensweisen des Agenten. Dieser Wert sollte so eingestellt sein, dass die Entropie (kann in TensorBoard überwacht werden) sich stetig über den Zeitraum des Trainings verkleinern soll. Wenn die Entropie zu schnell sinkt, sollte dieser Wert vergrößert werden. Wenn diese zu langsam sinkt, sollte der Wert vergrößert werden.

- Typischer Wert: 1e-4 - 1e-2

Epsilon (epsilon)

- Beschreibt den akzeptablen Abweichungsgrad zwischen alten und neuen Regeln des Modells während der ′Gradient decent′-Aktualisierung. Kleinere Werte führen zu stabileren Aktualisierungen auf kosten der Trainingsgeschwindigkeit.

- Typischer Wert: 0.1 - 0.3

Normalize (normalize)

- Beschreibt ob Vectoren welche Observiert werden normalisiert werden oder nicht. Normalisierung kann bei komplexen ′continous′ Steuerungsproblemen hilfreich sein und kann sich als Nachteilig bei einfachen ′discrete′ Steuerungsproblemen erweisen.

- Akzeptierte Werte: ′true′ oder ′false′

Number of Layers (num_layers)

- Gibt an aus wie vielen Schichten (Nach dem Input-Layer) das Neuronale Netzwerk des Agenten besteht. Für Umfelder, bei denen die korrekte Aktion eine einfache Kombination der Observation des Agenten ist, ist ein kleiner Wert ausreichend. Für Umfelder bei denen die korrekte Aktion eine komplexe Interaktion der Observation des Agenten ist, sollte dieser Wert größer ausfallen.

- Typischer Wert: 1 - 3

Hidden Units (hidden_units)

- Gibt an aus wie vielen Neuronen eine Schicht des Neuronalen Netzwerkes besteht. Diese Neuronen sind komplett mit allen Neuronen aus vorherigen und nachfolgenden Schichten verbunden. Wie bei den ′Number of Layers′ ist bei einfachen Observation ==> Aktion Abbildungen eine kleiner Wert Empfehlenswert, sowie bei komplexen Abbildungen sollte ein größerer Wert genutzt werden.

- Typischer Wert: 32 - 512

Summary Frequenzy (summary_freq)

- Gibt die Anzahl der Steps an, nach denen eine kurze Zusammenfassung der momentanen Lernergebnisse auf der Konsole wieder gegeben werden soll.

- Typischer Wert: 1e4 - 1e7

Keep Checkpoints (keep_checkpoints)

- Gibt die Anzahl der Zusammenfassungen an, welche längerfristig gespeichert werden sollen. Um z.B. den Agenten von einem früheren Zeitpunkt an erneut trainieren zu lassen ohne das komplette Training von neuen Anfangen zu müssen.

- Typischer Wert: 3 - 10

Mithilfe dieser Parameter kann eine YAML-Datei erstellen werden, welche genutzt wird um unseren Agenten zu trainieren. Im Fallbeispiel des ′BallBalancers′ sieht die YAML-Datei wie folgt aus:

```
behaviors:
  BallBalancer:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 1000
    summary_freq: 12000
```

Der ′behaviours′ Parameter gibt hierbei an, dass es sich in den folgenden Parametern um Verhalten des Agenten handelt. Tensorflow bietet uns die Möglichkeit verschiedene verhalten, verschiedener Agenten innerhalb einer YAML-Datei zu konfigurieren. Da wir jedoch nur unseren ′BallBalancer′ (der Name ist in den ′Behaviour Parameter′ Komponente des Agenten zu finden) trainieren wollen müssen wir nur diesen definieren. Danach definieren wir mit welchem Algorithmus

der Agent trainiert werden soll. Des weiteren ist hier noch wichtig dass wir unsere Parameter in 4 verschiedene Kategorien unterteilen. ′hyperparameter′ in denen generelle Trainingseinstellungen vorgenommen werden, ′network_settings′ in denen das Neuronale Netzwerk definiert wird, ′reward_signals′ welche die Einstellungen zu den Belohnungen enthält und zu Letzt ohne genaue Definition, Einstellungen welche ′Logging′ und Ende des Trainings definieren.

Wenn wir nun diese Datei erstellt haben, können wir den Agenten zum ersten mal trainieren.

4.6 Starten/Fortsetzen des Trainings

Das Starten des Trainingsvorganges ist relativ einfach und schnell erklärt. Hierbei starten wir die Python mlagent-Bibliothek in einer normalen Konsole (z.B. Windows-Command-Prompt). Das starten des Lernprozesses kann hierbei mithilfe des folgenden Befehl getan werden:

```
mlagents-learn <pathToYAMLConfigFile> --run-id=<RunName>
```

Hierbei ist <pathToYAMLConfigFile> der Pfad der zu vorherig definierten YAML-Datei führt und <RunName> der Name, welches die Bibliothek nutzt um das Training später abzuspeichern und/oder fortzuführen falls eine Unterbrechung statt gefunden hat. Wenn bei ′<pathToYAMLConfigFile>′ keine Datei angegeben wurde, nutzt die ′mlagents′ Standardeinstellungen welche nach starten des Trainings in der Konsole ausgegeben werden.

In unserem Fallbeispiel führen wir also den folgenden Befehl aus:

```
mlagents-learn .\configs\BallBalancing\ThesisSettings.yaml
    --run-id='ThesisBallBalancer'
```

Nach kurzem initialisieren wartet die Konsole dann auf eine Unity Instanz, welche mit der Konsole anfängt zu kommunizieren um Trainingsdaten auszutauschen. Hierbei hat man ca. 1 Minute Zeit eine Verbindung herzustellen, bevor die Konsole einen ′Time-out′ Fehler wirft. Diese Verbindung erzeugt man einfach, indem man in Unity in den ′Play-Mode′ wechselt. Die Unity ML-Agents-Bibliothek nimmt dann standardmäßig automatisch über den Port 5004 (kann bei Bedarf geändert werden) mit der Python mlagents Bibliothek Verbindung auf.

Im Anhang ist die Konsolenausgabe (Abbildung 1 und Abbildung 2) zu finden welche während des Trainings des Agenten entstanden ist. Hier sind noch einmal die Einstellungen dargestellt, welche genutzt werden um den Agenten zu trainieren. Zusätzlich können diverse Hinweis-Nachrichten erscheinen, welche aber in den meisten Fällen den Lernprozess nicht stören. Diese können deswegen Ignoriert werden. In der Abbildung 2 sind dann die Zusammenfassungen (Checkpoints) zu sehen welche während des Lernprozesses erstellt wurden. Der ′Step′ beschreibt hier die momentane Step-Anzahl bei welcher die Zusammenfassung erzeugt wurde, ′Time Elapsed′ gibt die vergangene Zeit seid dem starten des Trainingsdurchlaufes an, ′Mean Reward′ gibt die durchschnittliche Belohnung aller Episoden der letzten Aktualisierung an, ′Std of Reward′ gibt die Belohnungssummendifferenz verschiedener Episoden an (d.h. der unterschied zwischen der besten und schlechtesten Episode seid der letzten Zusammenfassung).

Um schnell zu erkennen ob unser Agent positiven Fortschritt zum lösen des Problem macht, sollte der ′Mean Reward′ betrachtet werden. Dieser sollte stetig über den kompletten Trainingsprozess steigen. Wenn dieser nicht mehr steigt, oder gar sinkt, ist der Agent entweder perfekt Trainiert oder das Umfeld des Agenten muss überarbeitet werden (Inputs/Outputs, Belohnungssysteme, Hyperparameter, ...) sodass dieser das Problem lösen kann. Tensorboard (welches nachfolgend erklärt wird) kann hier helfen genauere Anhaltspunkte zu finden welche eventuell überarbeitet werden sollten.

Es kann während des Trainings vorkommen dass der ′Mean Reward′ kurzzeitig sinkt und dann wieder stark ansteigt. Der Agent hat hier wahrscheinlich eine Vorgehensweise gefunden welche das Problem langsam löst. Durch weiteres ′Trial&Error′ des Agenten könnte dieser einen neuen Weg finden wie er das Problem auf andere Art und weiße lösen kann, jedoch benötigt er hier viele Versuche um einen alternativen weg so zu konstruieren sodass dieser ein besseres Ergebnis liefert. Sollte der ′Mean Rewards′ jedoch über einen vergleichsweise langen Zeitraum konsistent gleich bleiben oder sinken, muss man sich genauer Fragen warum kein Lernfortschritt erzieht werden kann. Typische Fragen welche sich hier dann gestellt werden sollten sind: Ist der Agent perfekt gelehrt? Fehlen dem Agenten wichtige Informationen zum Lösen des Problems? Bekommt der Agent eventuell zu viele Informationen und ist ′verwirrt′? Muss Ich meine Hyperparameter anpassen?

Dieser Kreislauf von Trainieren, Überwachung des Fortschritts und Überarbeiten der Umgebungen/Einstellungen ist bei komplexeren Problemstellungen sehr üblich und beinhaltet viel ′Trail&Error′ um gute Ergebnisse zu erzielen.

Falls der Agent während des Trainings unterbrochen werden sollte kann der Trainingsprozess mithilfe der ′–resume′-Flag fortgesetzt werden. (Also am Fallbeispiel mit folgendem Befehl:

```
mlagents-learn .\\configs\\BallBalancing\\ThesisSettings.yaml --resume
    --run-id='ThesisBallBalancer'
```

Des weiteren, wenn der Trainings versuch von Anfang an (unter gleichem Namen) gestartet werden soll kann die '–force'-Flag genutzt werden.

```
mlagents-learn .\\configs\\BallBalancing\\ThesisSettings.yaml --force
    --run-id='ThesisBallBalancer'
```

Dieses überschreibt jedoch den Vorherigen Trainings versuch und dessen Aufzeichnungen. Deswegen sollte dies mit Vorsicht eingesetzt werden um keinen Fortschritt zu verlieren.

4.7 Überwachen/Analysieren des Trainings (TensorBoard)

Bei TensorBoard handelt es sich um ein Werkzeug, welches Messwerte und Visualisierungen bietet welche während des Maschine Learning Workflows von nutzen sind. TensorBoard kann hierbei während oder nach des Trainings eines Agenten genutzt werden um dessen Fortschritt zu überwachen und zu analysieren. Bevor wir jedoch die erstellten Grafen von TensorBoard betrachten, müssen wir den Service zuerst starten. Hierzu öffnen wir einen neue Konsole und führen den Folgenden Befehlt aus:

```
tensorboard --logdir <DirectoryOfResults>
```

Hierbei steht <DirectoryOfResults> für den Pfad an welchem die Ergebnisse unserer Trainingsdurchläufe gespeichert werden. Diese sind standardmäßig in einem 'results'-Ordner welcher sich im gleichen Pfad befinden, in dem das Training gestartet wurde.

Das starten von TensorBoard kann einige Sekunden dauern. Das erfolgreiche starten, sowie die Addresse unter der TensorBoard zu finden ist, ist hierbei in der Konsole zu sehen. Bei unserem Beispiel, da TensorBoard lokal auf der Maschine läuft ist diese Addresse wie folgt: 'http://localhost:6006'.

Abbildungen zu TensorBoard können im Anhang (Abbildung 3+4 und Abbildung 11-15) gefunden werden.

Wenn wir nun auf die TensorBoard Seite navigiert haben, können wir die von TensorBoard erstellten Grafiken betrachten. Diese sind unter dem Reiter 'SCALARS' zu finden. Die anderen Reiter sind für uns eher unwichtig und wir Fokussieren uns auf diesen 'SCALARS' Reiter. Auf der Linken Seite sehen wir verschiedene Einstellungsmöglichkeiten, sowie die Trainingsdurchläufe ('Runs') welche wir bisher durchgeführt haben. Falls hier mehrere Runs dargestellt werden wählen wir denjenigen aus, welchen wir betrachten möchten. Falls wir 2 'Runs' miteinander vergleichen wollen, können wir hier auch mehrere auswählen. TensorBoard stellt diese dann in unterschiedlichen Farben innerhalb des selben Grafen dar.

Nachdem wir einen (oder mehrere) Run(s) ausgewählt haben, können wir 10 verschiedene Grafen betrachten, welche uns unterschiedliche Informationen über den Trainingsverlauf des Agenten liefert. Der dabei wichtigste Graf ist der 'Cumulative Reward'-Graf. Dieser gibt die Durchschnittsbelohnung aller Episoden (aller Agenten) zum entsprechenden Step-Zeitpunkt an. Hier sollte eine generelle Steigung der Kurve erkannt werden. Gelegentliches sinken mit anschließendem anstieg sind hier zu erwarten. Es kann hier, je nach Komplexität der Problemstellung, auch vorkommen dass die initiale starke Steigung der Kurve erst nach mehreren Millionen Steps erkannt werden kann. Wenn die Kurve über einen gewissen Zeitraum keinen sichtbaren Anstieg mehr erkennen lässt, kann davon ausgegangen werden dass der Agent keine weiteren Lernfortschritt mehr erzielen kann. Wenn der Agent nach diesem Zeitpunkt das Problem noch nicht befriedigend gelöst haben sollte, ist es nötig Änderungen am Umfeld/Einstellungen/Hyperparametern/... vorzunehmen.

Ein weiterer wichtiger Graf ist der 'Entropy'-Graf in der 'Policy'-Kategorie. Dieser gibt an, wie zufällig das Verhalten des Agenten und die Entscheidungen dessen 'Gehirn' sind. Diese Kurve sollte während des Trainings konsistent sinken. Falls diese Kurve zu schnell oder kaum bis überhaupt nicht sinkt, sollte der 'beta'-Wert in den Hyperparametern justiert werden.

Der letzte wichtige Graf, der betrachtet werden sollte ist der 'Learning Rate'-Graf in der 'Policy'-Kategorie. Dieser gibt an wie groß der Unterschied ist, welcher der Algorithmus nutzt um optimale Regeln (Policy) zu finden. Dieser wert sollte über die Zeit sinken.

Die restlichen Grafen können bei gewissen Problemstellungen sehr hilfreich sein. Desswegen hier eine kurze Beschreibung der restlichen Grafen und dessen Funktion:

- **Episode Length**
  - Die durchschnittliche Länge der Episoden in Steps. Je nach Problemstellung sieht hier der Graf unterschiedlich aus.

- Policy Loss
  - Der durchschnittliche Ausmaß der ′Policy Loss′-Funktion. Dieser schwankt generell sehr stark.
- Value Loss
  - Der durchschnittliche Verlust des Werts aus der Aktualisierungsfunktion. Dieser Graf sollte erst steigen (während der Agent ′lernt′) und dann sinken (während sich der Agent auf sein finales Verhalten stabilisiert).
- Beta
  - Gibt die stärke des ′Zufalls′ an, welcher der Agent während des Trainings nutzt um verschiedene Verhaltensweisen zu entwickeln. Die Kurve sollte während des Trainings stetig (idealerweise linear) sinken.
- Epsilon
  - Hilfsvariable für die ′gradient decent′ Aktualisierung. Kurve sollte stetig während des Trainings sinken.
- Extrinsic Reward
  - Gibt den Durchschnittswert der Belohnung pro Episode aller Agenten an. Kurve sollte stetig steigen.
- Extrinsic Value Estimate
  - Gib den vorhergesagten Durchschnittswert der Belohnung pro Episode aller Agenten an. Dieser Wert sollte mit Zunahme der Durchschnittsbehohnung steigen.

Wenn der Agent nach abschließen eines Trainigsvorganges die Problemstellung nicht löst, können diese Grafen hinzugezogen werden um Anhaltspunkte für den Start des ′Troubleshooting′ zu finden. In unserem Fallbeispiel sehen alle Grafen optimal aus und wir können davon ausgehen, dass der Agent unser Problem teilweise bis sehr gut lösen kann.

4.8 Finalisieren des Agenten

Wir haben nun eine Problemstellung definiert, ein Umfeld und Agenten geschaffen, diesen Agenten dazu trainiert die Problemstellung zu lösen und das Ergebnis des Trainings überwacht und analysiert. Hierbei hat uns Tensorflow und ml-agents ein (Gehirn)-Modell erzeugt, welches unser Agent nutzen kann um das Problem nun autonom (ohne weitere externe Tool) zu lösen.

Diese Modell ist hierbei in dem ′results′-Ordner in einem Ordner welcher den vorherigen definierten ′Behaviour′ Namen trägt zu finden. In diesem Ordner befinden sich verschiedene Dateien. Für uns sind die ′ONNX′-Dateien interesannt. Diese beinhalten das (Gehirn)-Modell. Die Zahl welche in den Dateinamen beinhaltet ist, gibt die Anzahl der Steps an, an denen diese Dateien während des Trainings erzeugt wurden. Generell möchten wir hier die Datei mit der größten Zahl für unseren Agenten nutzen (da diese am längsten Trainiert wurde).

In unserem Fallbeispiel kopieren wir uns die ′BallBalancer-500081.onnx′ an einen in Unity erreichbaren Ordner. Anschließend wählen wir in den ′Behaviour Parameters′ des Agenten (welchen wir Trainiert haben) unter der Variable ′Model′ das oben kopierte Modell aus. Wenn wir dies getan haben können wir die Simulation in Unity starten und sehen, dass der Agent komplett autark und autonom den Ball balanciert.

Somit haben wir erfolgreich einen Agent trainiert, welcher einen Ball auf einer Fläche balanciert.

Dieser ist zwar noch nicht bei weitem perfekt, kann jedoch mit mehr Training und etwas ′Trial&Error′ bei der Belohnungsfunktion/Hyperparametern bis zur Perfektion gebracht werden. Für den Zweck dieser Ausarbeitung reicht der momentane Agent jedoch vollens aus.

5 Weitere Vorgehensweisen und Funktionen zum Umsetzen eines Agenten mit ML-Agents

In dem vorherigen Kapitel wurde das generelle Vorgehen behandelt, um ein Problem zu definieren, ein Umfeld zu schaffen und einen Agenten und dessen Einstellungen zu erzeugen. Der Agent nutzt hierbei mlagents und Tensorflow um sich eigenständig anhand der Umgebung zu trainieren, sodass das gegebene Problem automatisiert oder gelöst wird.
Dies sind jedoch nur die Grundlagen, welche beherrscht werden sollten um die nachfolgenden Funktionen einfach und effizient nutzen zu können. Es gibt hierbei verschiedene Vorgehensweisen um einen Agenten auf eine gezielte Problemstellung zu trainieren, diese Vorgehensweisen sind allgemein gültig und gelten auch außerhalb Unity und MLAgents. Jedoch bietet MLAgents viel Hilfestellung welche beim Implementieren von anderen Vorgehensweisen sehr nützlich sind.
In dem Folgenden Kapitel werden wir auf weitere Vorgehensweisen eingehen und diese Anhand von Fallbeispielen (in Unity mithilfe von MLAgents) genauer erläutern.

5.1 Camera/Image-basierender Input am Beispiel 'Race Car'

In diesem Fallbeispiel soll erklärt werden wie Bild-Basierendes lernen generell funktioniert und welche Probleme mit dieser Methodik einfacher gelöst werden können. Dies wird am folgenden Fallbeispiel des 'RaceCars' genauer erklärt.

Die Problemstellung wir wie folgt beschrieben: Es soll ein Agent trainiert werden, welcher einen Rundkurs so weit wie möglich in einer gewissen Zeit entlang fährt. Das Auto bewegt sich mit konstanter Geschwindigkeit nach vorne. Der Rundkurs ist von Mauern eingegrenzt. Wenn der Agent den Rand des Kurses berührt, soll die Umgebung zurückgesetzt werden.

Dies wird mit Bild/Textur-Basierenden Input umgesetzt. D.h. Der Input des Agenten besteht ausschließlich aus einem Bild/Textur welches eine Kamera in der Umgebung einmal pro Step aufnimmt und dem Agenten zur Verfügung stellt. ML-Agent bietet uns hier eine sehr Hilfreiche Komponente namens ′Camera Sensor′ (Abbildung 5). Diese Komponenten nutzt eine Kamera (welche sich in der Szene befindet) und verarbeitet das Bild. Dieses wird dann so weiter verarbeitet, sodass der Agent dieses als Input nutzen kann. Hierbei können die Höhe und Breite (in Pixel) so wie Farb- oder Grau-Stufen-Ausgabe eingestellt werden. Eine Option zum ′Stacken′ des Sensors, sodass mehrere Bilder (aus vorherigen Steps) ′übereinander gelegt′ werden besteht ebenfalls.

Der Kamera Sensor funktioniert relativ einfach. Dieser skaliert dass Bild der definierten Kamera auf eine eingestellte Höhe und Breite. Anschließend gibt er jedes Pixel des Bildes an den Agenten als Input weiter. Hierbei, wenn ′Grayscale′ nicht ausgewählt ist, werden die Farbwerte des Pixel in RGB, d.h. als ein Vector3 dem Agenten weiter gegeben. Ein Pixel entspricht hier also 3 Werten welcher der Agent aufnehmen muss. Bei ausgewählten ′Grayscale′ werden die Pixel mit einer einzigen Zahl dem Agenten weitergegeben (der Grau-stufe des Pixel). Hier ist dann ein Pixel eine Eingabe. Wichtig ist hier, wenn eine Image-Basierende Lösung umgesetzt werden soll, sich zu fragen, ob der Agent Farben erkennen muss oder nicht, da dies ein Faktor 3 bei den Inputs des Agenten ausmacht. Die nächste Frage die sich gestellt werden muss, ist welche Auflösung eingestellt werden soll. Diese sollte so gering wie möglich ausfallen, da diese einen großen Einfluss auf die Anzahl der zu verarbeitenden Inputs hat. Eine gute Faustregel hierbei ist, diese so niedrig einzustellen, sodass eine Person mit dieser Auflösung das Problem konsistent lösen kann. Hierzu ist die ′Heuristic′ Methode des Agenten sehr hilfreich, da diese genutzt werden kann um den Agenten zu steuern. Dann kann Stück für Stück die Auflösung der Kamera reduziert und nach jedem reduzieren getestet werden ob man selbst das Problem noch konsistent lösen kann. Wenn das Problem nicht mehr gelöst werden kann, wird die nächst größere Auflösung genutzt.

Um die Wichtigkeit einer geringen Auflösung noch einmal zu verdeutlichen, eine kleine Rechnung. Man hat 2 Kameras, eine mit einer Auflösung von 16x16 und die andere mit einer Auflösung von 32x32. Die 16x16-Kamera enthält hierbei 256 Pixel zum Verarbeiten, d.h. entweder 256 (Grau-stufen) oder 768 (Farbig) Inputs. Bei der 32x32-Kamera haben wir hierbei 1024 Pixel, d.h. entweder 1024 (Grau-stufen) oder 3072 (Farbig) Inputs. Da wir wissen, dass mehr Inputs mehr Lernzeit und erschwertes Lernen bedeutet kann man hier schnell und einfach erkennen, dass die Auflösung der Kameras so gering wie möglich gehalten werden soll, da die Auflösung und Anzahl der zu verarbeitenden Inputs exponentiell in Zusammenhang stehen.

Nun da wir wissen wie diese Komponente genau Funktioniert, können wir die Umgebung umsetzen. Hierbei bauen wir einen kleinen Rundkurs. Um es dem Agenten zu erleichtert die Wände zu erkennen färben wir die äußeren Wände blau und die inneren Wände rot. Als nächstes erstellen ein ′Auto′ welches der Agent steuern soll. Dieses besitzt eine Kamera welche so platziert ist, um die kommende Strecke gut beobachten zu können. Wir steuern das Auto mithilfe einer einzigen ′Continous Action′. Das Auto kann sich mit einer gewissen Grad-zahl pro Sekunde um die Y-Achse drehen. Die ′Continous Action′ gibt den Ausschlag der Rotation pro Sekunde an.

Das Auto hat 60 Sekunden zeit den Rundkurs soweit wie möglich entlang zu fahren. Wenn der Agent eine Wand berührt oder die Zeit abgelaufen ist wird die Umgebung zurückgesetzt. Um sicher zu stellen, dass der Agent in die korrekte Richtung fährt, erstellen wir in regelmäßigen Abständen ′Checkpoints′ welcher der Agent abfahren soll.
Die Belohnung des Agenten ist wie folgt:

- Der Agent besitzt einen Multiplikator
- Der Agent erhält pro Step eine relativ kleine fixe Belohnung, welche (bevor der Agent die Belohnung erhält) mit dem definierten Multiplikator multipliziert wird.
- Wenn der Agent den nächsten korrekten ′Checkpoint′ erreicht, wir der Multiplikator um einen festen Wert erhöht
- Der Agent erhält eine relative große negative Belohnung, wenn dieser in eine Wand fährt.

Punkt 1 - 3 ermutigen hier den Agenten so schnell wie möglich den nächsten Checkpoint zu erreichen, da diese den Multiplikator (und somit) die Belohnung stark erhöhen. Punkt 4 trainiert den Agenten in keine Wand zu fahren und die 60 Sekunden voll auszuschöpfen. Der Aufbau der Umgebung, die Einstellungen des Agenten und die überschriebenen Agent-Methoden können im Anhang in den Abbildungen 5-8 gefunden werden.
Eine wichtige Anmerkung ist, dass in den ′Behaviour Parameters′ die ′Space Size′ auf 0 gesetzt ist und die Option ′Use Child Sensor′ ausgewählt ist. Wenn Sensor-Komponenten der MLAgents Bibliothek genutzt werden müssen wir diese nicht in der ′Space Size′-Einstellung berücksichtigen. Hier werden also nur die Anzahl der manuell gesammelten Inputs benötigt. Die ′Use Child Sensor′-Option sagt dem Agenten, dass dieser nach Sensor-Komponenten in seinem ′GameObject′ suchen und diese nutzen soll.

Wenn die Umgebung und der Agent erstellt wurde müssen noch die Hyperparameter des Trainings eingestellt werden. Hierbei, da diese Problemstellung relativ einfach ist, können die gleichen Hyperparameter wie die des ′BallBalancers′ genutzt werden. Anschließend können wir unseren Agenten trainieren und unser Ergebnis begutachten.

Es kann sein dass der Agent mehr als 500000 Steps benötigt um gute Ergebnisse zu erzeugen. Falls dies der Fall sein sollte, können wir einfach die ′max_steps′ in den Hyperparametern erhöhen und den Agenten weiter Trainieren bis die gewünschten Ergebnisse erkennbar sind.

5.2 Bekämpfen von ′Overfitting′

Nun da wir ein gutes Verständnis besitzen wie wir selbst lernende Agenten auf einfache Problemstellungen trainieren können, ist es an der Zeit ein wichtiges Problem anzusprechen, welches während der Erstellung dieser Agenten auftreten kann. Dieses Problem kann häufig auftreten, wenn man dies nicht von anfangen mit einbezieht und berücksichtigt.

Das angesprochene Problem nennt sich ′Overfitting′ und beschreibt das ′Übertrainieren′ eines Agenten auf ein gewisses Problem. Im genauen beschreibt dieses Problem, dass der Agent so gut im Lösen des beschriebenen Problems geworden ist, dass dieser genau dieses und nur genau dieses Lösen kann. D.h. wenn sich Kleinigkeiten im Umfeld ändern, ist der Agent verwirrt, da dieser nie gelernt hat mit kleineren Ungewissheiten umzugehen. In unserer Lernumgebung können wir sicherstellen, dass es zu keinen Ungewissheiten während des Trainings kommt. In der Realität ist dies jedoch nicht der Fall. Hier kommt es bei so gut wie jeder Problemstellung vor, dass wir Ungewissheiten oder zufällige Variablen haben mit welcher der Agent später im Einsatz zurecht kommen muss.
Glücklicherweise lässt sich ′Overfitting′ relativ einfach beseitigen, wenn man dies von Anfang an (vor der Erstellung des Umfeldes) mit in die Planung einbezieht. Hierbei wird die Lernumgebung so geplant und implementiert, dass diese Ungewissheiten und zufällige Variablen während des Trainings besitzt und/oder simuliert. Hierbei muss man sich Fragen, welche Ungewissheiten später im realen Einsatz bestehen und diese dann so gut wie möglich in der Trainingsumgebung mit einbinden. Je präziser die reale Umgebung nachgebildet wird, in welcher der Agent später eingesetzt werden soll, desto besser kann dieser auf diese Trainiert werden.
Hierbei ist jedoch zu beachten dass diese Ungewissheiten die Trainings-zeit des Agenten leicht (bei wenig und kleinen) und extrem stark (bei hohen Änderungen oder vielen verschiedenen Variablen) erhöhen können. Deshalb muss eine gute Balance zwischen sauberer und einheitlicher Trainingsumgebung und ′realer′

chaotischen Umgebung gefunden werden.
Um dies etwas einfacher zu gestalten kann hier eine Risikoanalyse betrieben werden. Hierbei sollten die verschiedenen Ziele des Agenten analysiert werden und betrachtet werden wie kritisch Auswirkungen währen, wenn gewisse Ziele nicht oder nur teilweise erreicht werden. Dann sollten die kritischsten Risiken genauer analysiert werden und es sollte ermittelt werden welche Informationen der Agent benötigt um präzise Vorhersagen zum Lösen dieser Risiken treffen zu können. Wenn man dann die kritischsten Informationen gesammelt hat, muss sich gefragt werden ob und wie stark diese Informationen in der realen Umgebung schwanken können.
Dies bietet eine sehr gute Grundlage zum entwickeln einer Trainingsumgebung, in der die wichtigsten Ungewissheiten implementiert sind ohne die Trainings-zeit des Agenten in extreme Größen schießen zu lassen.

Diese Implementierung der Ungewissheiten kann auf viele verschiedene Arten umgesetzt werden. Glücklicherweise besitzt MLAgents und Tensorflow Werkzeuge, um dieses Problem zu bekämpfen. Hierbei können in der Hyperparameter-Datei, welche genutzt wird um den Agenten zu trainieren, sogenannte Umgebungsvariablen definiert werden. Diese besitzen entweder einen festen Wert oder einen Werte-spanne. Der feste Wert ist nützlich um Variablen zu definieren welche wir eventuell ändern möchten, ohne direkt im Quellcode Änderungen vornehmen zu müssen. Dies ist generell Nützlich, hilft uns jedoch gegen das ′Overfitting′-Problem nur bedingt. Interessanter für uns ist die Möglichkeit Werte-spannen für eine Variable zu definieren. Hierbei definieren wir einen Minimal- und Maximalwert für die Variable. Während des Trainings wird dann in jeder Episode ein zufälliger Wert von Tensorflow definiert (welcher in der Werte-spanne beinhaltet ist) und dann in diese Umgebungsvariable gespeichert.

Hierbei bietet uns Tensorflow und MLAgents folgende Einstellungsmöglichkeiten zum individualisieren der Umgebungsvariablen:

- **sampler_type: uniform**
  Ermittelt eine zufällige Fließkommazahl zwischen einem Minimal- und Maximalwert (beide Inklusive)
  - min_value - der Minimalwert (inklusive)
  - max_value - der Maximalwert (inklusive)
- **sampler_type: multirangeuniform**
  Ermittelt eine zufällige Fließkommazahl welche sich innerhalb mehrerer verschiedenen Werte-spannen zwischen den Minimal- und Maximalwert befindet (beide Inklusive)

  - intervals - Die Intervalle in denen sich die Zufallszahl befinden kann.

  - Beispiel: [[1, 5], [8 ,90], [3 , 30]]

- **sampler_type: gaussian**
  Ermittelt eine zufällige Fließkommazahl welche mithilfe der Gaußischen Normalverteilung errechnet wird.

  - mean - Der Durchschnittswert aller gezogenen Zufallszahlen

  - st_dev - Die maximale Abweichung vom Durchschnittswert der gezogenen Zufallszahlen.

Um nun Umgebungsvariablen zu definieren, erweitern wir einfach die Hyperparameter-Datei mit allen Variablen und wie diese ermittelt werden sollen. Nachfolgend ist ein kurzer Abschnitt mit einem Beispiel zum definieren solcher Variablen zu sehen:

```
environment_parameters:        # Start of the Environment Variables

   mass:                       # Variable Name
      sampler_type: uniform    # Variable calculation type
      sampler_parameters:      # Start of Variable Parameters
         min_value: 0.5
         max_value: 10
   length:
      sampler_type: multirangeuniform
      sampler_parameters:
         intervals: [[7, 10], [15, 20]]
   scale:
      sampler_type: gaussian
      sampler_parameters:
         mean: 2
         st_dev: .3
```

In diesem Beispiel werden 3 verschiedene Variablen mit den verschiedenen Berechnungsmethoden definiert. Um diese dann in unserem Quellcode nutzen zu können, nutzen wir folgende Funktion der MLAgents-Bibliothek:

```
Academy.Instance.EnvironmentParameters.GetWithDefault(<NameOfEnvVariable>,
   <defautValue>)
```

Hierbei gibt '<NameOfEnvVariable>' den Namen der in den Hyperparametern definierten Umgebungsvariablen an. '<defaultValue>' gibt den Wert an auf den

zurückgefallen wird, wenn diese Umgebungsvariable nicht gefunden werden konnte.
Während des implementieren des Umfeldes nutzen wir diese Methode mit den gewünschten Variablen an allen stellen an dem wir die verschiedenen zufälligen Variablen benötigen.
Man könnte natürlich auch an all diesen Stellen einfach zufällige Zahlenwerte manuell im Quellcode selbst ziehen, jedoch ist hiervon abzuraten, da es wesentlich einfacher und übersichtlicher ist, wenn man alle ungewissen Variablen in einer Datei definieren und überblicken kann.
Wenn dann die Umgebung implementiert und der Agent erstellt wurde, kann man mit dem Trainieren anfangen. Anschließend sollte auf jeden Fall getestet werden ob der Agent korrekt in der realen Umgebung funktioniert. Falls nicht sollte man zurück zur Risikoanalyse gehend und überprüfen, ob alle wichtigen Risiken und deren benötigten Informationen aufgelistet und umgesetzt worden sind. Hier ist im gewissen fällen 'Trial&Error' unabdinglich.

5.3 Curriculum Learning am Fallbeispiel 'HuntingGame'

Die bisherigen betrachteten Problemstellungen waren alle sehr simpel. Jedoch ist im Normalfall eine Problemstellung in der Realität sehr komplex und benötigt mehrere Schritte, welche Nacheinander erfolgreich ausgeführt werden müssen um ein Problem zu lösen. Hier kann 'Curriculum Learning' sehr hilfreich sein.

Wie der Name sagt wird der Agent hier mithilfe eines 'Lehrplanes' trainiert. Konkret heißt dies dass sich das Umfeld während des Trainings langsam erweitert. Die Erweiterungen des Umfeldes geschehen immer dann, wenn der Agent vordefinierte Ziele erreicht hat. Hierbei startet der Agent in einem einfach zu lösenden Umfeld, welches stetig komplexer und komplizierter wird. Der Agent kann dann bei den komplexeren Problemstellungen auf die Erfahrungen zurückgreifen welche er in den einfacheren Umfeldern zu früheren Zeiten des Trainings gemacht hat. Ziel des ganzen ist es den Agent solange in immer schwerer werdenden Umfeldern zu trainieren, bis das Umfeld so komplex geworden ist, dass es unseres initiales Problem nachbildet und vom Agenten gelöst werden kann.

In diesem Kapitel möchte Ich die Verschiedenen Aspekte und Änderungen anhand eines Fallbeispiels namens 'HuntingGame' erläutern. Bevor wir mit den Erläuterungen anfangen, definieren wir dieses 'Hunting'Game':

Die Simulation besteht aus einer Fläche auf der sich 3 Jäger und 1 Gejagter befinden. Der Gejagte ist hierbei in der Mitte der Fläche platziert und die Jäger sind

ringsum den Gejagten in fixen Abständen platziert. Ziel der Jäger ist es, alle Jäger gleichzeitig in einen gewissen Abstand zu dem Gejagten zu bewegen. Hierbei ist jeder Jäger ein eigener Agent. Die Jäger teilen sich ein ′Gehirn′ während des Training. Ziel des Gejagten ist es, sich eine bestimmte Distanz von der Mitte der Fläche zu entfernen. Der Gejagte ist hierbei eine simple KI welche nach fester Logik handelt (kein RL-Agent). Der Gejagte bewegt sich hierbei in einem errechenbaren Muster. Dieser errechnet den Mittelpunkt aller Jäger. Dann errechnet das Muster den Vektor des Mittelpunktes aller Jäger und seiner eigenen Position. Dieser Vektor wird normalisiert und gibt die Bewegungsrichtung des gejagten an (in kurz, der Gejagte entfernt sich immer optimal von den Jägern). Die Agenten der Jäger erhalten als Input die x/z-Position aller Jäger und die des Gejagten. Zusätzlich ′Stacken′ wir diese Vektoren 3-mal (d.h. die Agenten erhalten die Informationen aus den letzten 3 Steps plus den Aktuellen Step), sodass die Agenten die Richtung ihrer Verbündeten und des Gejagten erkennen können.

Die Agenten erhalten hierbei folgende Belohnungen:

- kleine negative Belohnung pro Step pro Step (z.B. -0.001)
- Verdoppelung der negativen Belohnung pro Step solange sich ein Agent während der Simulation weiter weg von Mittelpunkt der Fläche befindet als zu beginn der Episode
- einmalige (pro Episode) kleine positive Belohnung wenn der Agent das äußere drittel des Fläche (zwischen Startpunkt der Jäger und des Gejagten) betritt (z.B. +0.25)
- einmalige (pro Episode) ′mittlere′ positive Belohnung wenn der Agent das äußere drittel des Fläche (zwischen Startpunkt der Jäger und des Gejagten) betritt (z.B. +0.75)
- große positive Belohnung, wenn die Agenten den Gejagten Fangen (z.B. +5)
- ′mittelgroße′ negative Belohnung, wenn der Gejagte entkommt (z.B. -3)

Der erste Punkt ermutigt die Agenten so schnell wie möglich den Gejagten zu fangen. Punkt 2 - 4 hilft den Agenten initial die richtige Richtung zu finden um das Training etwas zu beschleunigen.

Wenn wir dieses Umfeld umsetzen würden und die Agenten direkt anfangen würden zu trainieren, könnte es sehr lange dauern, bis die Agenten erkennbaren Fortschritt zeigen (Dies wird am Ende des Kapitels beim betrachteten der Ergebnissgrafen in TensorBoard deutlicher). Die Agenten wissen Initial nicht wie diese sich selbst bewegen, wer Freund und Feind ist, was das eigentliche Ziel ist und wie dieses erreicht werden kann. Zudem haben wir ein sehr zeitkritisches Ziel, da

wenn der Gejagte entkommt, die Umgebung zurückgesetzt wird und kein Lernerfolg der Agenten erzielt wird. Dass sich die Agenten alle zufälligerweise schnell und zeitgleich sich am Gejagten befinden ist sehr unwahrscheinlich. Wir könnten natürlich versuchen eine extrem komplexe und sensible Belohungsfunktion zu entwickeln, welche die Agenten versucht auf diese komplexe Problemstellung zu trainieren, dies erfordert jedoch sehr viel ′Trial&Error′ und ist deswegen sehr Zeitintensiv zum entwickeln.

Die bessere Alternative ist es hier einen geeigneten Lernplan für das ′Curriculum Learning′ zu erstellen. Hierzu müssen wir nun unsere komplexe Problemstellung Stück für Stück vereinfachten, bis wir an einen Punkt gelangt sind, welcher eine relative einfach zu erlernende Problemstellung beschreibt. Hier baut man seine komplexe Problemstellung am besten in einzelne kleinere Problemstellungen auseinander, welche Stück für Stück gelöst werden müssen um das gesamt Problem zu lösen. In unserem Fallbeispiel besteht unser komplexes Problem aus 4 kleineren Problemen:

- Bewegung: Wie bewege Ich mich?
- Zielerkennung: Wer ist und wo befindet sich mein Ziel gerade?
- Verfolgung: Wie bewege Ich mich auf mein Ziel zu?
- Team-Koordination: Wie stelle Ich sicher dass alle Verbündeten (Jäger) gleichzeitig sich am Ziel (Gejagten) befinden.

Als nächstes müssen wir uns Fragen wie man die erkannten simplem Problemstellungen aus der komplexen Problemstellung für das Training extrahieren kann. Also wie und wo muss man das Umfeld abändern um die simpleren Umfelder zu erzeugen. Anschließend muss man sich fragen, wie man das simple Umfeld erweitert, um wieder auf das komplexe Umfeld zurück zu kehren. Während dieses Prozesses (von simplen Umfeld zu komplexen) muss zudem sichergestellt werden, dass alle erkannten simplen Problemstellungen schrittweise behandelt, umgesetzt und gelöst werden.

In unserem Fallbeispiel sieht unser Lehrplan wie folgt aus.

i. Zuerst bringen wir den Agenten bei sich zu bewegen und wie diese das Ziel erkennen. Diese beiden Problemstellungen können perfekt kombiniert werden. Der Gejagte ist hier in dieser kompletten Phase bewegungslos und hat einen sehr großen Radius in welchem er gefangen werden kann. Dies gibt den Agenten eine gute Hilfestellung schnell zu erlernen wie diese sich bewegen und das Ziel ausfindig machen können, da es viel positive Belohnung und relativ wenig negative Belohnung für die Agenten gibt.

ii. Wenn die Agenten konsistent den Gejagten mit großen ′Fangradius′ fangen können, verkleinern wir den Radius schrittweise, bis wir auf der gewünschten Größe angekommen sind. Dies gibt den Agenten zeit ihre Bewegungsroutine und Zielverfolgung zu verbessern.

iii. Wenn die Agenten dann konsistent den Gejagten mit kleinem ′Fangradius′ fangen können, erhöhen wir schrittweise die Geschwindigkeit des Gejagten. Wir fangen hierbei sehr klein an und werden stetig schneller. Dies erhöhen wir entweder endlos weiter oder bis ein gewisses Ziel erreicht ist.

iv. Wenn die Geschwindigkeit des Gejagten dann die der Jäger erreicht und überschreitet, müssen die Jäger zusammen arbeiten um das Ziel konsistent fangen zu können. Diese Verhalten wird automatisch in unserer Umgebung erzielt.

Nun da wir wissen wie unser Lehrplan aussieht, müssen wir uns Fragen welche variablen sich während des Training ändern. Diese Variablem werden zur Implementierung des Lehrplans benötigen. In unserem Fallbeispiel benötigen wir 2 Variablen, die sich während des Trainingslaufes verändern. Der ′Fangradius′ und die Geschwindigkeit des Gejagten. Wenn diese komplett ermittelt wurden, können wir mit der Implementierung des Lehrplans in MLAgents anfangen.

Der Lernplan wird hier in der Hyperparameter-YAML-Datei hinzugefügt. Wie wir schon im Kapitel ′Overfitting′ gelernt haben, können wir Umgebungsvariablen definieren auf welche wir während des Training zugreifen können. Beim Curriculum Learning möchten wir hierbei jedoch keine feste oder zufälligen Werte, sondern wir möchten Werte die sich nach erfüllen von gewissen Voraussetzungen ändern. Hierbei können wir diese Umgebungsvariablen mit dem Attribut ′curriculum′ erweitern. Mit den folgenden Einstellungen können wir genau definieren wie sich die Umgebungsvariable während des Trainings verhalten soll.

- **name**
  Der Name der momentanen Lektion (im Lernplan)

- **completion_criteria** - Hier werden weitere Attribute des Kriteriums definiert. Wenn diese erfüllt sind, geht die Akademie in die nächste Lektion über.

  - **measure**
    Was soll als Erfolgskriterium gemessen werden?

    * reward - Die erhaltene Belohnung

    * progress - Der Fortschritt in Verhältnis zu ′current step′′max steps′

  - **behaviour**
    Welches verhalten (welches in den Hyperparametern definiert wurde) soll für die aktuelle Lektion genutzt werden?

  - **min_lession_length**
    Die minimale Anzahl an Steps welche vergehen müssen um in die nächste Lektion übergehen zu können. Der Durchschnittswert der Belohnung in diesem Zeitraum wird dann verwendet um zu prüfen ob das Erfolgskriterium erreicht wurde. **Wichtig: Der Durchschnittswert der Belohnung kann hierbei unterschiedlich zu dem Durchschnittswert der Belohnung sein, welcher in der Konsole ausgegeben wird, da dieser 'summary_freq' zum errechnen benutzt.**

  - **threshold**
    Der Wert welcher zum erfüllen des Kriteriums überschritten werden muss.

  - **signal_smoothing**
    Wenn 'true' dann wir der momentane Belohnungswert mit 75% mit dem Belohnungswert aus dem Step davor mit 25% zusammen gerechnet um zu verhindern das Ausreißer eventuell ein Fortschritt im Lehrplan zur folge haben.

  - **requre_reset**
    Wenn 'true' dann wird die Umgebung bei Fortschreiten des Lehrplans zurückgesetzt. (Standardwert ist 'false')

- **value**
  Der Wert der Umgebungsvariable in dieser Lektion

Im Anhang unter Abbildung 9 und 10 sind die Hyperparameter, sowie ein ein Teil des implementierten Lehrplan zu sehen. Der komplette Lehrplan besteht aus ca. 200 Zeilen und wird deshalb nicht komplett abgebildet. Das Vorgehen kann jedoch in den Abbildungen gut erkannt werden.

Wichtig zu erwähnen ist, dass die einzelnen Umgebungsvariablen unabhängig von einander sind. Hierbei besitzt jede Umgebungsvariable ihren eigenen Lehrplan welchem diese folgt. Diese können Zeitgleich oder auch unterschiedlich voranschreiten. Je nach Implementierung kann dies hier differenzieren.

Nachdem wir den Lehrplan implementiert haben nutzen wir die im vorherigen Kapitel beschriebene MLAgents-Methode um auf diese Umgebungsvariablen zuzugreifen:

```
Academy.Instance.EnvironmentParameters.GetWithDefault(<NameOfEnvVariable>,
    <defautValue>)
```

Wenn wir dann an allen benötigten stellen die Umgebungsvariablen in unsere Umgebung implementiert haben können wir mit dem Training der Agenten anfangen.
Hierbei benutzen wir den vorherig gelernten Befehlt und wählen wie gewohnt die Hyperparameter-Datei für das Training aus. Man kann erkennen, dass die Akademie ′Curriculum Learning′ anwenden, indem man die Konsolenausgaben der Python ′mlagent′ Bibliothek betrachtet. Hier sollte gut zu erkennen sein, welche Lektionen momentan genutzt werden und wenn Lektionen voranschreiten.

Das Training kann dann mithilfe von TensorBoard überwacht und analysiert werden. Im Anhang unter den Abbildungen 11 - 12 können die Grafen des Agenten unseres Fallbeispiels ′HuntingGame′ betrachtet werden. Hierbei unterscheiden sich die Grafen teilweise sehr stark von den Grafen, die wir bisher kennen gelernt haben.
Pro Lektion (pro Umgebungsvariable) sehen wir nun einen zusätzlichen Grafen, welcher darstellt wann (Step-Anzahl) der Agent sich in welcher Lektion (Nummer) befunden hat. Des weiteren wenn wir den ′Cumulative Reward′ betrachten fällt schnell auf da dieser sehr untypisch zu den bisherig betrachteten Grafen aussieht. Hier ist jedoch ein stetiges hoch und tief zu erwarten. Da jedes mal, wenn die Akademie die nächste Lektion startet der Agent ein komplexeres, schwereres Umfeld präsentiert bekommt. Hier muss dieser erneut lernen wie er mit den neuen Problematiken umgehen muss, was ein starkes absinken der Belohnung an jedem Start einer neuen Lektion hervorruft. Dann über Zeit wenn der Agent lernt mit der Situation umzugehen steigt dessen Belohnung wieder an. Dies geschieht solange bis das Kriterium erreicht wurde und die nächste Lektion gestartet wird. Dieses auf und ab ist also ein gutes Zeichen wenn wir einen Agenten mit ′Curriculum Learning′ trainieren.

Am Ende können wir die von Tensorflow erstellten Gehirne in die Agenten innerhalb unserer Unity Szene nutzen und können betrachten wie diese autonom den Gejagten versuchen zu fangen.

6 Erstellen von Agenten im kompetitiven Umfeld am Fallbeispiel ′BeanShooter′

6.1 Umsetzung

Bisher habe wir nur Agenten betrachtet und trainiert, welche entweder alleine in der Umgebung oder in Kooperation mit anderen Agenten trainieren. Dabei haben die Agenten immer das selbe ′Gehirn′ genutzt. Dies ermöglicht den Agenten untereinander Informationen und somit Lernfortschritt auszutauschen. Konkret gesagt bedeutet dass der Fortschritt eines Agenten gleichzeitig der Fortschritt für alle anderen Agenten darstellt. Dies ist in vielen Problemstellungen eine sichere und stabile Vorgehensweise, jedoch gibt es Problemstellungen bei denen der Agent gegen einen unbekannte Entität antritt.
Diesen Fall möchte Ich etwas genauer am Fallbeispiel des ′Bean Shooters′ demonstrieren, da hier interessante Verhaltensweisen erkannt werden können, welche bei kooperativen Training nicht erkennbar sind. Bevor wir jedoch zu diesen Verhaltensweisen kommen möchte Ich kurz das Umfeld definieren in dem die Agenten trainiert wurden.

In der Umgebung treten 2 Agenten in einer Arena gegeneinander an. Die Agenten starten hierbei auf unterschiedlichen Seiten der Arena. Diese besteht aus Wänden und verschiedenen Hindernissen (Blöcke) welche die Sicht der Agenten einschränkt. Die Agenten können sich in horizontaler Ebene bewegen und ihre Sicht um die Y-Achse rotieren. Die Agenten nehmen Ihre Umgebung mithilfe von einem ′Raycast′-Array wahr. Die Raycasts werden vom Agenten in dessen Sicht nach vorn in verschiedenen Winkeln (in einem vordefinierten Sichtfeld-weite und -breite) geschossen. Der Agent kann anhand dieser RayCasts erkennen was sich vor ihm befindet und wie weit dieser davon entfernt ist. Der Agent kann zusätzlich mit einer Abkühlphase von 1 Sekunde zentral (von seiner Sicht aus) nach Vorne schießen. Ziel der Simulation ist es, den anderen Agenten zu finden und diesen zuerst abzuschießen. Wenn ein Agent abgeschossen wurde, wird die Umgebung zurückgesetzt. Die Agenten erhalten hierbei folgende Belohnungen:

- Wenn der andere Agent erfolgreich abgeschossen wurde erhält der schießende Agent eine große positive Belohnung

- Wenn der Agent abgeschossen wurde erhält dieser eine negative Belohnung

- Wenn der Agent den anderen Agenten im Sichtfeld hat, bekommt dieser pro Step eine kleine positive Belohnung

- Wenn der Agent sich näher wie eine bestimme Distanz zum anderen Agenten aufhält, erhält dieser pro Step eine kleine positive Belohnung

- Wenn der Agent sich bewegt erhält er pro Step eine kleine positive Belohnung

Diese Belohnungen sollen den Agenten dazu motivieren sich zu bewegen, den anderen Agenten zu finden und diesen abzuschießen.

Um das ganze sehr simpel zu halten, sind beide Agenten mit der gleichen Logik implementiert worden. D.h. diese nutzen beide die gleichen Komponenten und Skripte im Hintergrund. Der große unterschied hier im Vergleich zu den anderen Fallbeispielen ist, dass jeder der Agenten sein eigenes "Gehirn"besitzt. Wenn nun also Agent A etwas lernt, bekommt Agent B nichts davon mit. Dieser kleine aber auswirkungsvolle Änderung zieht gewisse Verhaltensänderung mit sich, welche später Mithilfe der Trainingsgraphen genauere erläutert werden.
Umgesetzt wird das ganze in der ′Behaviour Parameter′-Komponente und in der Hyperparameter-Datei. In den ′Behaviour Parametern′ müssen wir sicherstellen dass der ′Behaviour Name′ der beiden Agenten unterschiedlich ist. Alle Agenten welche den gleichen ′Behaviour Namen′ besitzen lernen in das selbe ′Gehirn′. Deswegen muss dieser hier unterschiedlich sein. Des weiteren (rein aus besserer organisatorischer Sicht) ändern wir die ′Team Id′, sodass die Agenten hier auch unterschiedliche Team IDs besitzen. Dies ist zwar nicht Zwingend notwendig, hilft aber dabei die Konsolenausgabe von MLAgents besser zu überblicken.
Als nächstes müssen wir die Hyperparameter beider ′Behaviors′ in der Hyperparameter-Datei definieren. Hier würde die Möglichkeit bestehen verschiedene Hyperparameter für die Verschiedenen ′Behaviors′ zu nutzen, jedoch um dieses Beispiel simpel zu halten benutzen wir die selben Hyperparametern bei beiden ′Behaviors′. Wenn diese dann Implementiert worden sind, kann man mit dem Training beginnen. In der Konsole sehen wir nun pro Zusammenfassung die Werte beider Agenten.

Die bessere Überwachung und Analyse kann jedoch wieder in TensorBoard vorgenommen werden.

6.2 Bemerkungen und Interessante Verhaltensweisen

Wenn wir uns nach dem Training die Grafen in TensorBoard genauer betrachten können wir relativ schnell starke Unterschiede zwischen den bisher betrachteten Vorgehensweisen erkennen. In Abbildung 13 - 15 sind die ′Cumulative Reward′ Grafen der beiden Agenten, sowie ein Graf welcher beide Grafen überlappt zu erkennen.

Als Ich mit dem Training begonnen hatte, bin Ich davon ausgegangen, dass ein Agent früher oder später eine starke Strategie entwickelt und den anderen Agenten von dort ab an, bis zum Rest des Trainings dominiert. Dieses verhalten ist auch initial (in den ersten Millionen Steps) zu erkennen. Hierbei zeigt Agent B einen wesentlich größeren Reward als Agent A.
Nach den ersten Millionen Steps ist ein starker Abfall der Episodenbehlohnung bei beiden Agenten zu betrachten. Dies ist nicht ungewöhnlich, da die Agenten lernen wie diese effizient das Ziel erreichen. Wenn wir unsere Simulation noch einmal betrachten fällt auf, dass die Agenten mehr Episodenbehlohnung erreichen können wenn die Episode mehr Zeit in Anspruch nimmt. Da die Agenten jedoch effizienter im Ziel finden und abschießen werden wird die Episoden dauern geringer. Geringere Episodendauer bedeutet sogleich auch geringere Episodenbelohnungen.

Das interessante hierbei ist jedoch dass Agent B während dieses Abfalls der Episodenbelohnung gleich auf mit Agent A ist. Hier muss also Agent A von dem verhalten von Agent B gelernt haben. Die Agenten treiben sich hierbei deren ′Ineffektivität′ gegenseitig aus und trainieren sich so gegenseitig.
Dies geschieht dadurch, da wenn Agent A eine bessere Strategie als Agent B entwickelt hat gewinnt Agent A nur dann, wenn Agent B′s Strategie dessen von Agent A unterlegen ist. Hierbei lernt dann Agent B, dass die genutzt Strategie nicht mehr effizient genug ist um Agent A′s zu schlagen und muss eine neue Strategie entwickeln. Agent A ändert seine Strategie nur bedingt, da diese funktioniert und gute Ergebnisse liefert. Wenn Agent B dann eine bessere Strategie wie Agent A entwickelt hat dreht sich der Spieß um. Dies geht endlos hin und her weiter.
Im Grafen ist dies mehrfach gut zu erkennen. Das erste mal direkt beim ersten Abfall der Episodenbelohnung. Hier wechselt die ′starke′ Seite sehr oft, da die Simulation noch erkundet wird. Hierbei sind beide Agenten gleich auf. Diese werden jedoch in der Simulation langsam ′besser′ (da stetiger Belohnungsabfall (beider Seiten) zu erkennen ist). Dann nach circa 3.5 Millionen Steps, dominiert

Agent B mit einer Strategie für einen längeren Zeitraum. Gefolgt von einem Schlagartigen wechsel. Diesen extremen wechsel kann Ich mir an dieser stelle jedoch nicht wirklich erklären. Ich konnte diesen auch nicht in erneuten Testläufen replizieren und dieser sollte als Ausreißer betrachtet werden. Zu guter Letzt dominiert Agent B zum Schluss des Trainings wieder.

Eines der Verhaltensweisen, welche sich gut erkennen lässt ist dass beide Agenten über einen langen Zeitraum auf eine gleiche Fähigkeitsebene gelangen. Wenn man das initiale kennen lernen der Simulation (hier die ersten 2 Millionen Steps) betrachtet, erkennt man das beide Agenten immer näher an ein Mittelwert gelangen. Wenn man die Agenten in dieser Simulation weiter trainieren würde, wird man feststellen, dass die Agenten früher oder später die gleichen Erlebnisse liefern würden.
Dieses Verhalten ist komplett meiner initialen Vermutung, dass ein Agent den anderen über den kompletten Verlauf des Trainings dominiert, entgegen gesetzt.

6.3 Verbesserungsvorschläge

Das erstellen eines kompetitiven Agenten ist in der Regel sehr komplex mit vielen unbekannten Variablen. Deshalb möchte Ich kurz auf verschiedene Vorschläge eingehen welche überarbeitet werden können um das Ergebnis des darüber liegenden Fallbeispiels zu verbessern.

Der erste Vorschlag, der mir während der Verfassung der Ausarbeitung direkt ins Auge gesprungen war ist die Belohnungsfunktion. Diese gibt dem Agenten sehr viele positive Belohnungen und nur relativ wenig und nur kleine negative Belohnungen. Im nach hinein betrachtet wäre hier eine Belohnungsfunktion besser geeignet, welch dem Agenten pro Step eine kleine negative Belohnung zuweist und dann wenn der Agent das Ziel erfüllt hat erhält dieser eine große positive Belohnung. Hierbei kann es sein, dass sehr viele Variablen im Umfeld vorhanden sind, sodass der Agent niemals oder nur sehr selten den anderen Agenten sieht/abschießt und so mit keine Verbindungen zwischen Abschuss und positiver Belohnung erstellen kann.
Dieses Problem kann auf 2 weitere Arten gelöst werden. Entweder überlegen wir uns kleine Belohnungen welche den Agenten ermutigen den anderen zu finden (so wie momentan im Fallbeispiel vorhanden) oder wir trainieren die Agenten mit Hilfe von ′Curriculum Learning′ und starten in einem Umfeld indem die Agenten sehr nah mit langsamer Bewegungsgeschwindigkeit beieinander starten. Dann

wenn die Agenten besser werden vergrößern wir den Startabstand und die Geschwindigkeit der Agenten bis wir an das ursprüngliche Umfeld gelangt sind.
Da in den Belohnungs-Grafen diverse Ungewöhnlichkeiten aufkamen (siehe Ausreißer im Fallbeispiel) wäre es hier Verbesserungswert dieses Szenario mehrmals durch zu führen und zu überprüfen ob dieser Ausreißer häufiger vorkommt. Ich persönlich habe den Agenten 3x in diesem Szenario trainiert und konnte das stetige wechseln der Dominanz zwar erkennen, jedoch gab es nur einmal diesen extremen Fall. Da jedoch ein Training um die 10-14h dauert ist dies ein sehr zeitaufwendiger Prozess.
Ein weiterer Vorschlag ist es die Agenten mit verschiedener Logik zu implementieren um hier zu überprüfen ob ähnliche Verhaltensmuster erkannt werden können oder ob sich hier eigene Muster bilden.
Es könnten zudem Änderungen an den Hyperparametern der Agenten vorgenommen werden, da Ich mir sicher bin dass die von mir gewählten Hyperparameter nicht perfekt eingestellt sind. Da (aber wie schon angesprochen) ein Testversuch sehr viel Zeit in Anspruch nimmt, fällt es hier schwer viel ′Trial&Error′ zu betreiben. Interessant wäre es jedoch einmal genauer in die Unterschiedlichkeiten der Hyperparameter einzusteigen und zu testen welche Auswirkungen diese in einem kompetitiven Umfeld auf den Agenten und die Ergebnisse dessen haben.
Zu guter Letzt muss auch noch Overfitting in meinem Szenario bekämpft werden. Hierbei könnten wir die Agenten an unterschiedlichen Standorten starten lassen und die Hindernisse in der Szene zufällig platzieren. Da dies aber eine anstieg an Trainingszeit bedeuten würde und Ich nicht genau wusste wie lange meine Agenten trainieren müssen, habe Ich dies jedoch bei der Planung und Implementierung ignoriert.

Man sieht dieses Fallbeispiel hat noch einige stellen an denen Verbesserungen vorgenommen werden können. Dies gilt jedoch generell für Machine Learning Anwendungen, da in den meisten Fällen die KI, dessen Umgebung oder Hyperparameter optimiert werden können.
Der genannte Fallbeispiel gibt trotz vieler verbesserungswürdiger Stellen einen guten Einstieg in das Nutzen von Agenten in kompetitiven Umfeldern und zeigt dass hier die Verhaltensweisen der Agenten sehr unterschiedlich im Vergleich zu kooperativen Umfeldern ausfällt.

7 Zukunftsaussichten

Bevor wir zum Abschluss der Ausarbeitung kommen möchte Ich noch ein paar Ausblicke in die Zukunft von Maschine Learning geben, da Ich persönlich sehr großes Potential in dieser Technologie sehe welche uns das Leben in sehr vielen Feldern wesentlich erleichtern kann. Die beiden größten Potentiale sehe Ich hier in Anwendungsgebieten, welche menschliches eingreifen minimieren oder sogar komplett eliminieren und in Gebieten, in denen menschliche Ressourcen durch autonome Ressourcen ersetzt werden können. Hierzu betrachten wir die Folgenden Bereiche etwas genauer:

Eine der wohl größten Anwendungsgebiet ist die moderne Medizin, welche diese Technologie nutzen könnte um Diagnosen voll automatisch anhand von nicht invasiven Tests an Patienten zu tätigen. Dies würde es uns ermöglichen Heimgeräte zu entwickeln, welche selbst von Laien genutzt werden können um Diagnosen effektiv und präzise zu tätigen. Dies würde das Gesundheitswesen entlasten, da Patienten nur Fachpersonal beanspruchen müssten, wenn diese Tatsächlich schwerere Krankheiten haben. Ein weiterer großer Aspekt hier wäre in der Chirurgie. Hier könnten KIs entwickelt werden welche Operationen entweder unterstützen oder sogar selbstständig autonom durchführen könnten.

Ein weiteres sehr großes Thema wäre autonome Fahren. Hier könnten wir komplett autonome Fahrzeuge entwickeln (welche momentan schon in der Entwicklung sind). Dies ermöglicht uns effizientere Straßenvehrkesmodelle zu Entwickeln welche die Autonomie dieser Fahrzeuge nutzt (Technologie kann präziser zusammenarbeiten als der Mensch). Zudem reduziert diese Autonomie in Fahrzeugen Verkehrsunfälle, da rund 90% aller Verkehrsunfälle in Deutschland (2019) auf menschliches Versagen [25, 26] zurück zu führen sind. Beides würde zur Entlastung der Straßen und zu einem besseren Verkehrsfluss auf den Straßen führen.

Ein weiteres größeres Feld welches momentan im ’Falschen’ Bereich genutzt wird ist die personalisieren von Informationen. Dies wird heutzutage meist in der Werbeindustrie und in Onlinemarketing genutzt. Hier werden Informationen über den

Kunden auf verschiedenste Art und Weise gewonnen. Eine KI errechnet dann weitere Werbungen in welche der Kunde interessiert seinen könnte. Diese Technologie könnte im Bildungswesen mit großen Erfolg eingesetzt werden. Hierbei könnten Informationen über Schüler/Studenten gesammelt werden um individuelle Lehrpläne für diese zu erstellen. Diese Lehrpläne berücksichtigen die individuellen stärken und schwächen, sowie die Lernpreferenzen des Schülers/Studenten und könnten somit die Lerneffetivität dieser steigern.

Es gibt noch unzählige weitere Anwendungsbereiche in denen das nutzen von Machine Learning einen großen Nutzen finden würde. Mit diesen Einblicken möchte ich weiter die Vorstellungskraft aller anregen, welche sich in diesem Umfeld interessieren, da hier der Fantasie nur wenig Grenzen gesetzt sind.

8 Fazit

Wir haben gelernt wie man für verschiedene Problemstellungen einen selbstlernenden Agenten zum lösen dieser erstellt. Hierbei gibt es viele verschiedene Aspekte zu betrachten; viele verschiedene Fragen welche man sich vor und während der Implementierung stellen sollte um eine passende Vorgehensweise zu finden. Diese Wahl der Vorgehensweisen ist hierbei ausschlaggebend auf die Effektivität und Ergebnisse unserer Umgebung, des Agenten und dessen Training. Je nach Vorgehensweise gibt es weitere verschiedene Problematiken, auf welche geachtet werden müssen, um deren negative Auswirkungen auf unser Endergebnisse zu vermeiden. Dieser Prozess der Planung, Implementierung, Umsetzung und Training des Umfeldes und des Agenten ist hierbei ein sehr zeitaufwendiger, komplexer Prozess.
Wann man diese Schritte beherrscht und gut Einsetzen kann, besitzt man ein sehr mächtiges Werkzeug zum automatisierten Lösen von Problemstellungen aller Art.
Hierbei fällt es schwer (wie auch in anderen Kategorien der Softwareentwicklung) den Überblick über alles zu behalten und alle Problematiken während der Design-Phase zu erkennen. Es ist hier sehr Üblich dass Problematiken erst während der Implementierung erkannt werden, welch ein neues Planen des Umfeldes, des Agenten oder des Trainings mit sich ziehen. Dieses 'Trial&Error' ist hier ein unvermeidlicher Schritt in der Erstellung dieser Agenten und kann mit Erfahrung und Planungsziel nur reduziert und nicht komplett terminiert werden.

Maschine Learning und Reinforcement Learning stellt sich als sehr mächtiges Werkzeug dar, welches zum automatisieren und lösen gewissen Problemstellungen in sehr vielen Bereich der Arbeitswelt eingesetzt werden kann. Diese Fähigkeit und Flexibilität von Maschine Learning rechtfertigt hierbei den hohen initialen Aufwand der zum erlernen und implementieren dessen betrieben werden muss.

Danksagung

An dieser Stelle möchte ich mich bei allen denjenigen bedanken, die mich während der Anfertigung dieser Bachelorthesis unterstützt und motiviert haben.

Zuerst möchte Ich mich bei der Unity- und Google-Community bedanken, welche die Bibliotheken entwickelt haben, die Ich in dieser Thesis nutze. Des weiteren möchte Ich mich bei den unzähligen online Nutzern bedanken, welche Zeit aus ihrem Tag nehmen um Fragen der Community auf Plattformen wie ′StackOverflow′ und ′answers.unity.com′ zu beantworten.

Ich bedanke mich bei meiner Freundin und meinen Freunden, welche mir Unterstützung und Motivation in verschiedenen Phasen dieser Ausarbeitung gegeben haben.

Ein besonderer Dank gilt meinem Erstkorrektor Herr Prof. Dr. Benjamin Himpel, welcher mir regelmäßig während des Erstellung der Thesis viele Ratschläge und gutes konstruktives Feedback geliefert hat.

Abschließend möchte ich hier meinen Eltern danken, welche mir die Möglichkeit gegeben haben ohne große Sorgen mein Studium durchzuführen. Zudem möchte Ich ihnen für die Motivation danken, welche Sie mir während des Verlaufes meines Studiums zukommen ließen.

Glossar

PPO - Proximal Policy Optimization
SAC - Soft Actor-Critic
GALI - Generative Adversarial Imitation Learning
BC - Hebavioral Cloning
RL - Reinforcement Learning
ML - Maschine Learning
KI - Künstliche Intelligenz
Agent - Entität welche mithilfe von ML Trainiert werden soll

Literaturverzeichnis

[1] ML-Agent Documeantation - https://github.com/Unity-Technologies/ml-agents/tree/main/docs - Unity-Technologies

[2] Teaching a machine learning Agent to survive in a 2D Topdown environment. - Teemu Ropilo - 2019 - https://urn.fi/URN:NBN:fi:amk-2019120925566

[3] Validierung des Einsatzpotenzials von ML-Agents für kompetitive Multiplayer-Spiele - Phillipp Thomas - 2019

[4] Efficient Training Techniques for Multi-Agent Reinforcement Learning in Combat Tasks - GUANYU ZHANG, YUAN LI, XINHAI XU, HUADONG DAI - 2019 - https://ieeexplore.ieee.org/abstract/document/8789448

[5] Multiagent Cooperation and Competition with Deep Reinforcement Learning - Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aruy, Jaan Aru, Raul VicenteB - https://arxiv.org/abs/1511.08779

[6] A Markov Game Approach for Multi-Agent Competitive Bidding Strategies in Electricity Market - Navid Rashedi, Mohammad Amin Tajeddini, Hamed Kebriaei - 2016 - DOI: 10.1049/iet-gtd.2016.0075

[7] Steuerung eines Agenten zum Lösen eines Videospiels anhand eines evolutionären trainieten neuralen Netzwerk - Alice Grigorjan - 2019

[8] Emergent Tool Use From Multi-Agent Autocurricula - Bowen Baker, Ingmar Kanitscheide, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, Ifor Mordatch - 2020

[9] Deep Reinforcement Learning - Yuxi Li - 2018 - arXiv:1810.06339v1

[10] N. Bell, X. Fang, R. Hughes, G. Kendall, E. O'Reilly and S. Qiu, "Ghost direction detection and other innovations for Ms. Pac-Man,Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, 2010, pp. 465-472, doi: 10.1109/ITW.2010.5593320.

[11] Ou, S., He, X., Ji, W. et al. Machine learning model to project the impact of COVID-19 on US motor gasoline demand. Nat Energy 5, 666–673 (2020). https://doi.org/10.1038/s41560-020-0662-1

[12] Zhou, Z., Chen, K., Li, X. et al. Sign-to-speech translation using machine-learning-assisted stretchable sensor arrays. Nat Electron 3, 571–578 (2020). https://doi.org/10.1038/s41928-020-0428-6

[13] Parul, Dr. Basant Sah, Brain Tumor Detection using Clustering Maschine Learning Algorithm: An Overview (2021) - International Journal of Science, Technology and Management (IJSTM) ISSN (online): 2321-774X Volume 8, Issue 2, 2021

[14] Ou, S., He, X., Ji, W. et al. Machine learning model to project the impact of COVID-19 on US motor gasoline demand. Nat Energy 5, 666–673 (2020). https://doi.org/10.1038/s41560-020-0662-1

[15] Mohri Mehryar, Rostamizadeh Afshin, Talwakler Ameet - Foundation of maschine learning (2018) - LCCN 2018022812 - ISBN 9780262039406 - https://lccn.loc.gov/2018022812

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov - Proximal Policy Optimization Algorithms (2017) - https://arxiv.org/pdf /1707.06347.pdf

[17] https://gitee.com/mirrors/Unity-ML-Agents/blob/master/docs

[18] Training with Curriculum Learning - https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-Curriculum-Learning.md

[19] Funktionsweise und UseCases von Maschine Learning - Luca Rommler - 2022

[20] Serious Games: What Are They? What Do They Do? Why Should We Play Them? - Fran C. Blumberg, Debby E. Almonte, Jared S. Anthony, and Naoko Hashimoto - 2013 - https://www.oxfordhandbooks.com/view/10.1093/oxfordhb /9780195398809.001.0001/oxfordhb-9780195398809-e-19

[21] Hybrid Machine Learning Approach in Data Mining - Jyothi Bellary, Bhargavi Peyakunta, Sekhar Konetigari - 2010 - https://ieeexplore.ieee.org/document /5460721

[22] Generate use case from the requirements written in a natural language using machine learning - Mohamed S. Osman, Nour Zeyad Alabwaini, Tamara Baker Jaber, Thamer Alrawashdeh - 2019 - https://ieeexplore.ieee.org/document/8717428

[23] https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg

[24] How neural Nets Work - Alan Lapedes, Robert Farber - Los Alamos National Laboratory - 1988

[25] Menschliches Versagen verursacht die meisten Verkehrsunfälle - Marie Maier - 2021 - https://www.fahrschule-online.de/nachrichten/menschliches-versagen-verursacht-die-meisten-verkehrsunfaelle-2911045#: :text=Laut%20Statistik%20ist%20jeder%20vierte,Fahrrad%2D%20oder%20sonstigen%20Fahrzeugf%C3%BChrern%20verursacht.

[26] Wer weniger Unfalltote will, muss den Menschen einbremsen - Sueddeutsche Zeitung - 2019 - https://www.sueddeutsche.de/auto/eu-fahrerassistenzysteme-pflicht-2022-1.4386816

[27] Artificial Intelligenxe, Business and Civilization - Andreas Kaplan - 2022 - https://www.routledge.com/Artificial-Intelligence-Business-and-Civilization-Our-Fate-Made-in-Machines/Kaplan/p/book/9781032155319

[28] A Brief History of Artificial Intelligence - Tanya Lweis - 2014 - https://www.livescience.com/49007-history-of-artificial-intelligence.html

[29] https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg

Anhang

Abbildung .1: Konsolen Ergebnisse des ´BallBalancers´ 1

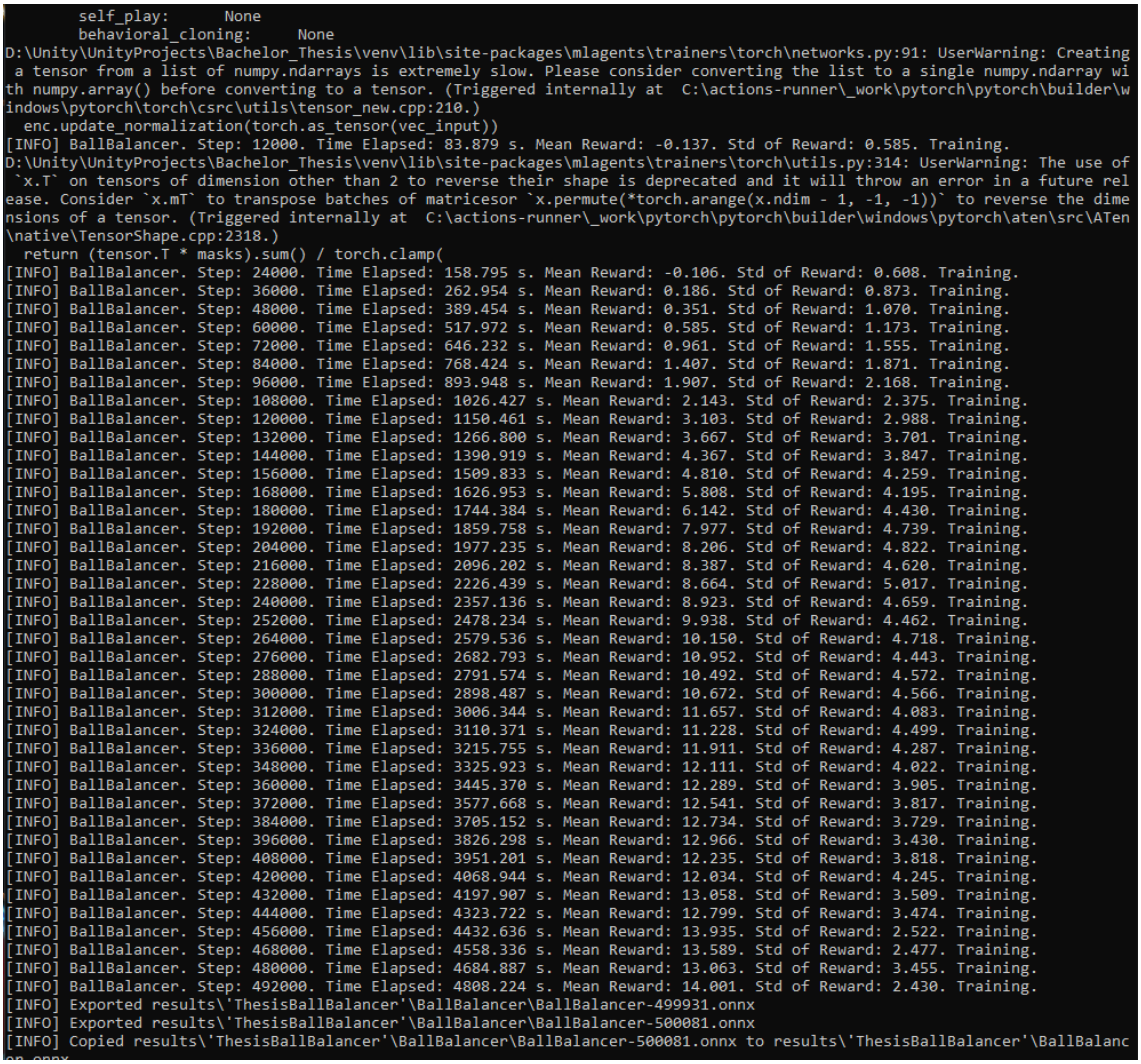

Abbildung .2: Konsolen Ergebnisse des ´BallBalancers´ 2

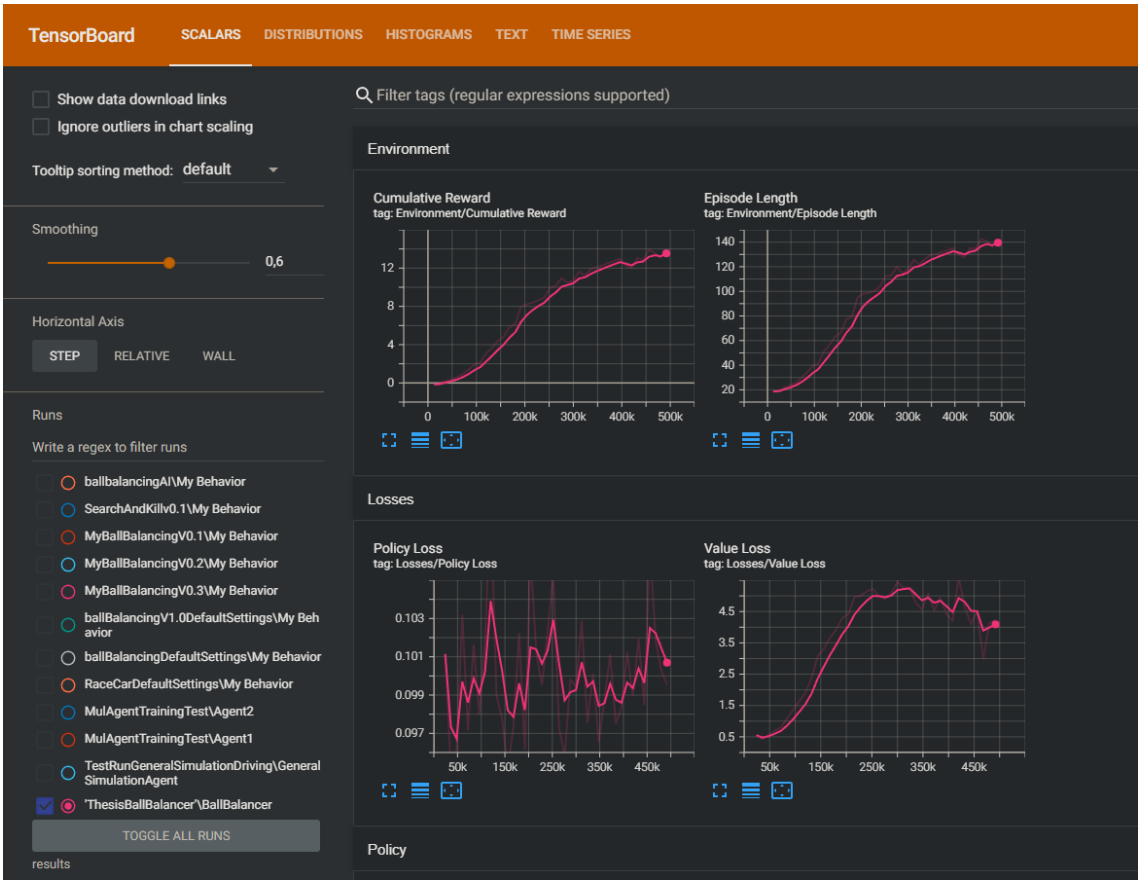

Abbildung .3: TensorBoard Grafen des ’BallBalancers’ 1

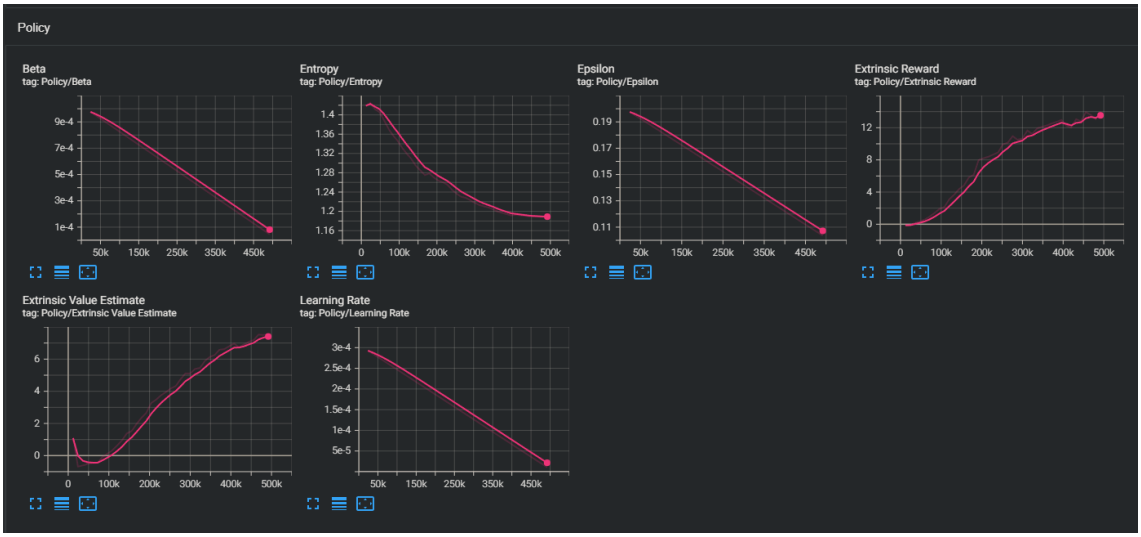

Abbildung .4: TensorBoard Grafen des ’BallBalancers’ 2

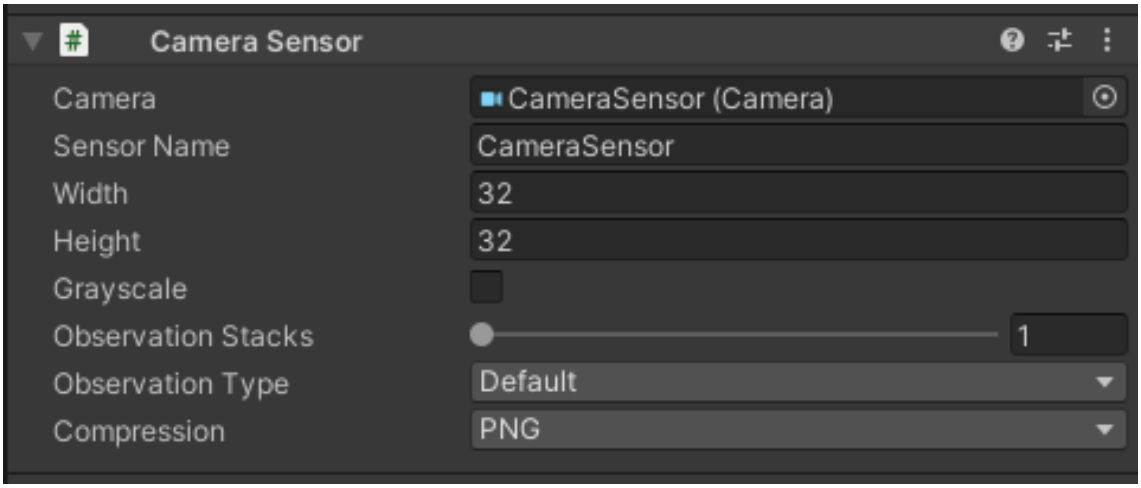

Abbildung .5: Beispiel des ′CameraSensors′

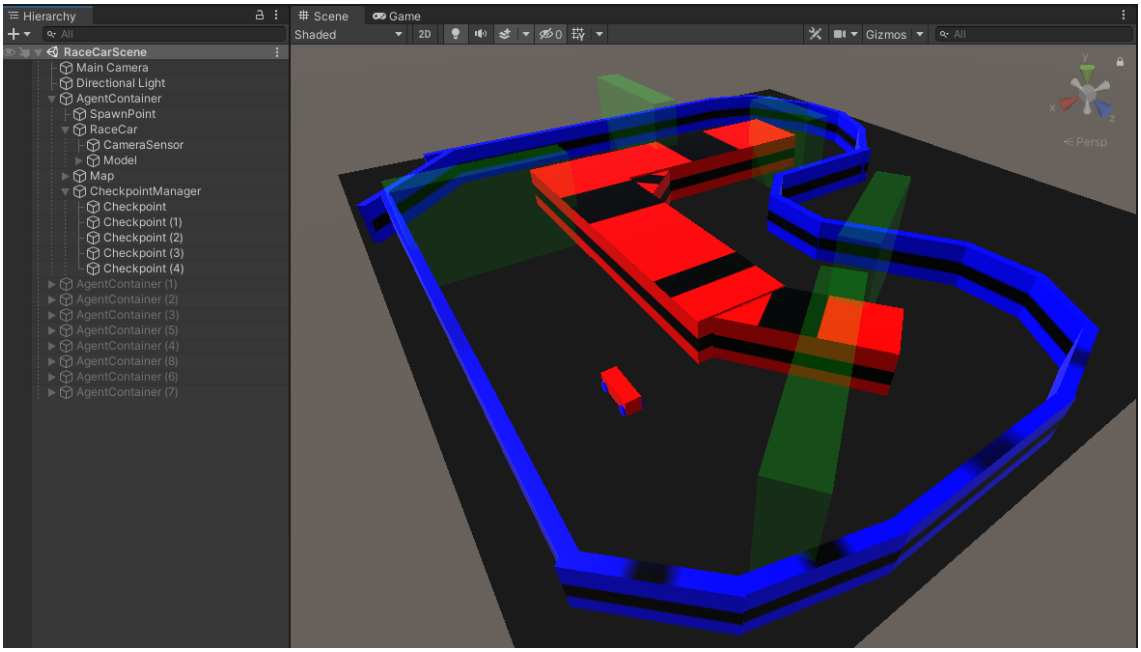

Abbildung .6: ′RaceCar′ Hierarchie und Szene

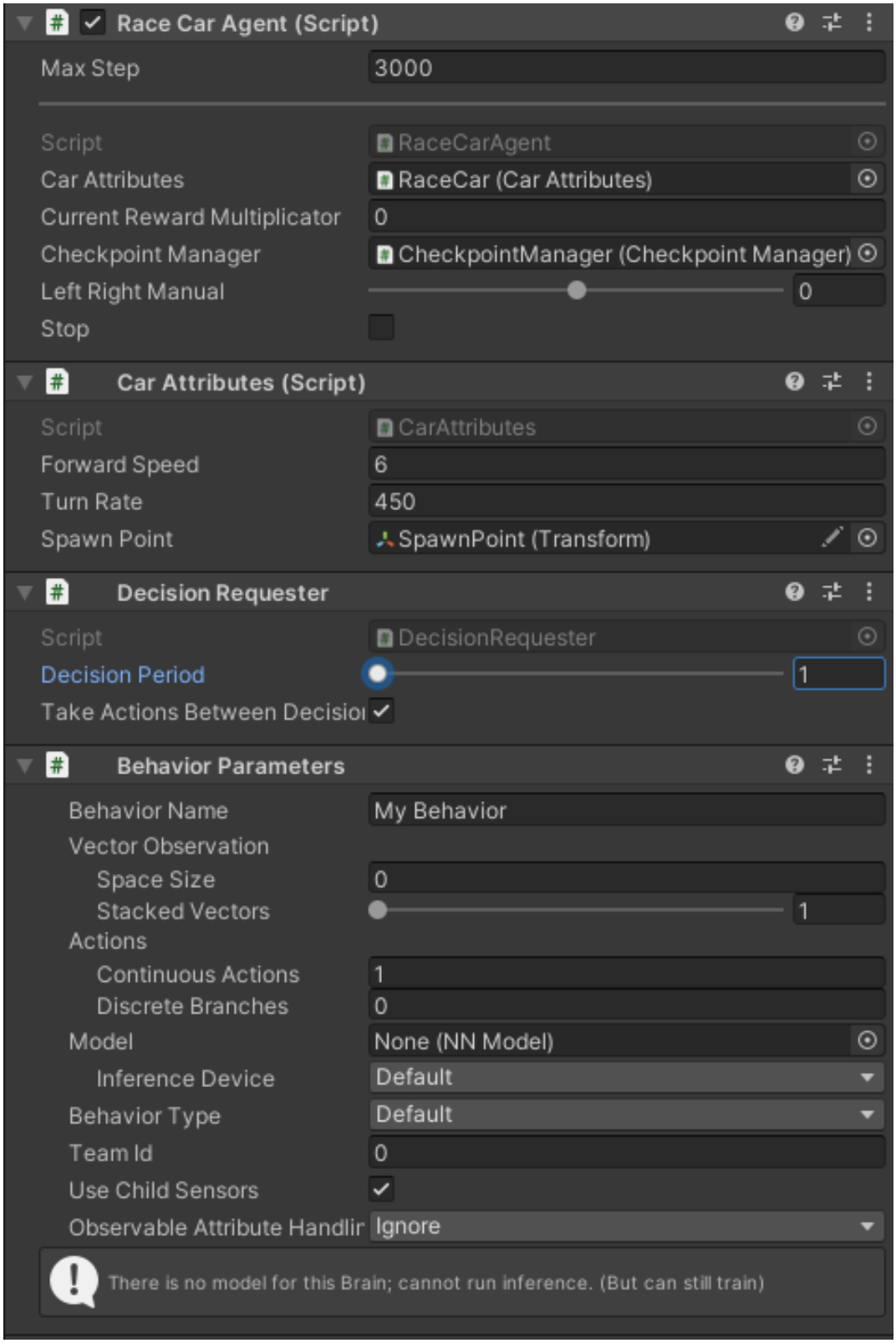

Abbildung .7: 'RaceCar'-Agenten Komponenten Einstellungen

```
public override void Initialize()
{
    carAttributes ??= GetComponent<CarAttributes>();
    rb = GetComponent<Rigidbody>();

    currentRewardMultiplicator = 1f;

    ResetScene();
}

public override void OnActionReceived(ActionBuffers action)
{
    Vector3 forwardDirection = Vector3.Normalize(new
        Vector3(transform.forward.x, 0f, transform.forward.z));
    rb.velocity = forwardDirection * carAttributes.forwardSpeed;

    float leftRight = action.ContinuousActions[0];

    transform.RotateAround(transform.position, Vector3.up, leftRight
        * carAttributes.turnRate * Time.deltaTime);

    SetReward(0.1f * currentRewardMultiplicator);
}

//On EpisodeBegin set the Agent to the start position and Rotation
public override void OnEpisodeBegin()
{
    ResetScene();
}

//Player Controlls for the Agent
public override void Heuristic(in ActionBuffers actionsOut)
{
    transform.RotateAround(transform.position, Vector3.up,
        leftRightManual * carAttributes.turnRate * Time.deltaTime);
}

//Penalise the Agent if it hits a Wall
private void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.CompareTag("Obstacle"))
    {
        SetReward(-3f);
        EndEpisode();
    }
}
```

Abbildung .8: Überschriebene Methoden des ′RaceCar′-Agent-Skrips

```
behaviors:
  HunterAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      num_epoch: 3
      buffer_size: 20580
      learning_rate: 0.0003
      beta: 0.003
      epsilon: 0.2
      lambd: 0.95
      learning_rate_schedule: linear
    network_settings:
      hidden_unity: 256
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
    keep_checkpoints: 5
    summary_freq: 50000
    max_steps: 10000000
    time_horizon: 1024
engine_settings:
  time_scale: 30
  target_frame_rate: -1
```

Abbildung .9: Hyperparameter der Agenten im Fallbeispiel 'HuntingGame'

```
environment_parameters:
  preyMovementSpeed:
    curriculum:
      - name: MovementStop0     # The - is importante because it is a
          list
        completion_criteria:
          measure: reward
          behavior: HunterAgent
          min_lesson_length: 200
          threshold: 4
          signal_smoothing: true
        value: 0
      - name: MovementStop1        # Second Lesson
        completion_criteria:
          measure: reward
          behavior: HunterAgent
          min_lesson_length: 200
          threshold: 4
          signal_smoothing: true
        value: 0
[...]
    - name: MovementVeryHard3.0
        completion_criteria:
          measure: reward
          behavior: HunterAgent
          min_lesson_length: 200
          threshold: 4
          signal_smoothing: true
        value: 3
    - name: MovementImpossible4.0 # Last Lession stays until end of
        training
        completion_criteria:
        value: 4
  preyCatchBoxSize:
    curriculum:
      - name: CatchBoxSize9     # The - is importante because it is a
          list
        completion_criteria:
            [...]
        value: 6                # Last Lession stays until end of
            training
      - name: CatchBoxSize3
        completion_criteria:
        value: 3
```

Abbildung .10: Ausschnitte des Lehrplans in der Hyperparameter-Datei im Fallbeispiel 'HuntingGame'

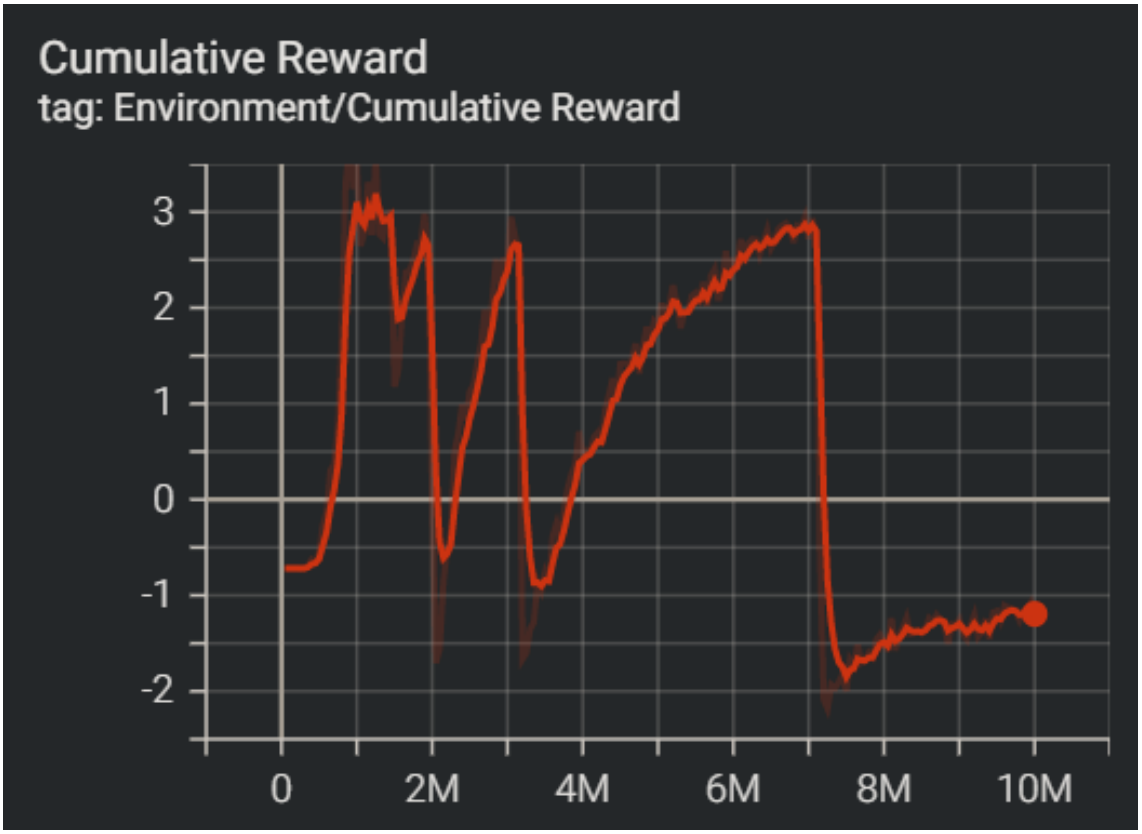

Abbildung .11: ‘Cumulative Reward’ Graf des Fallbeispiels ‘HuntingGame’

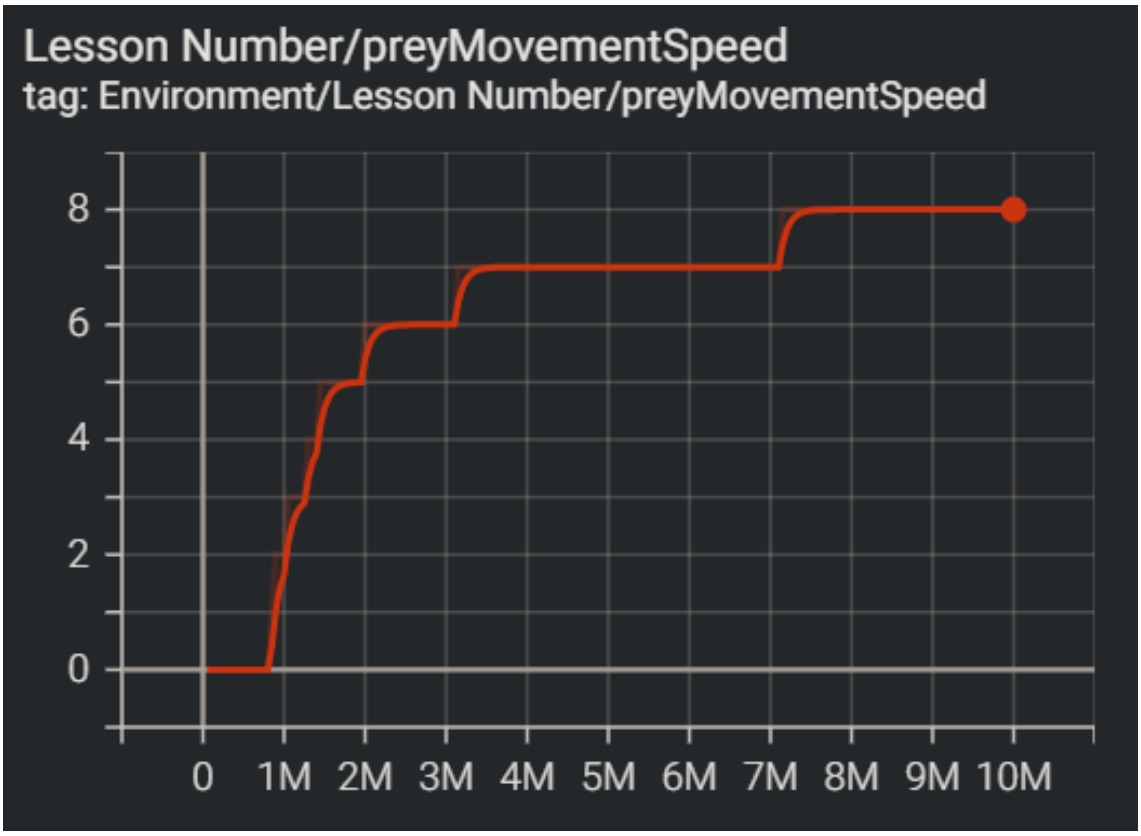

Abbildung .12: ′Lesson Number (PreyMovement)′ Graf des Fallbeispiels ′Hunting-Game′

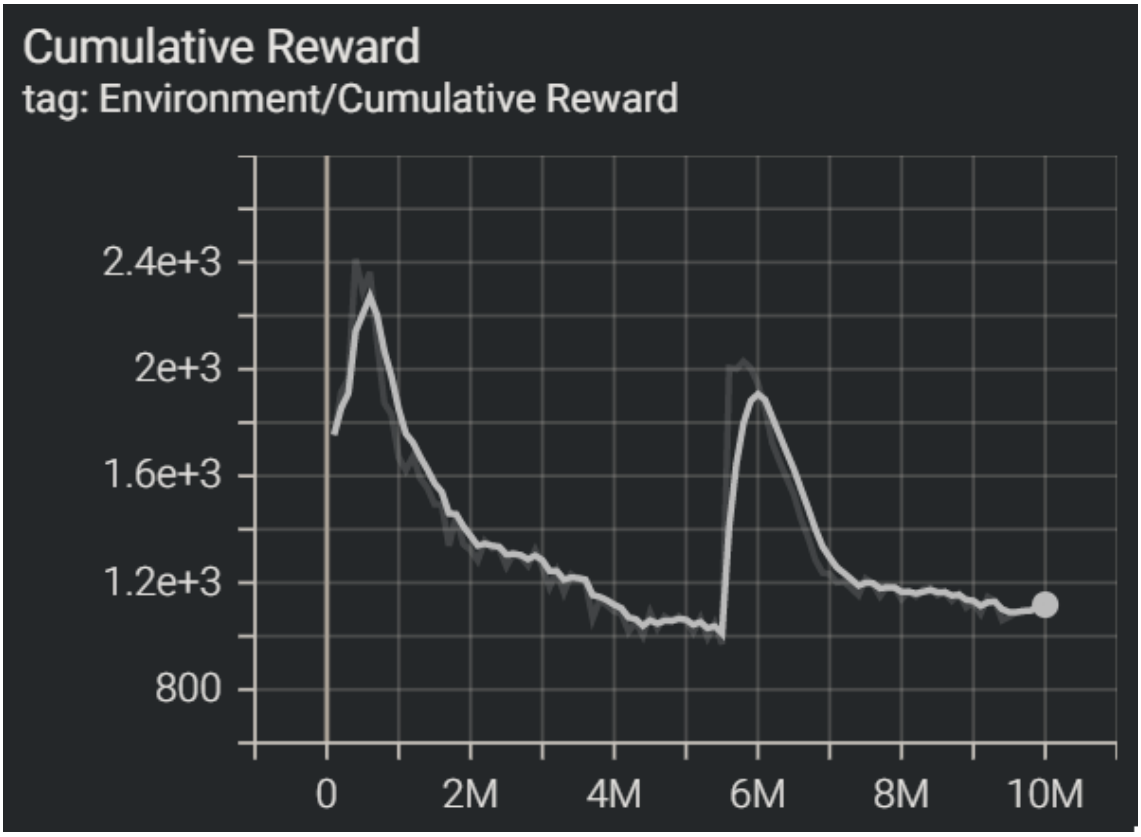

Abbildung .13: ′Cumulative Reward - BeanShooter - Agent A′

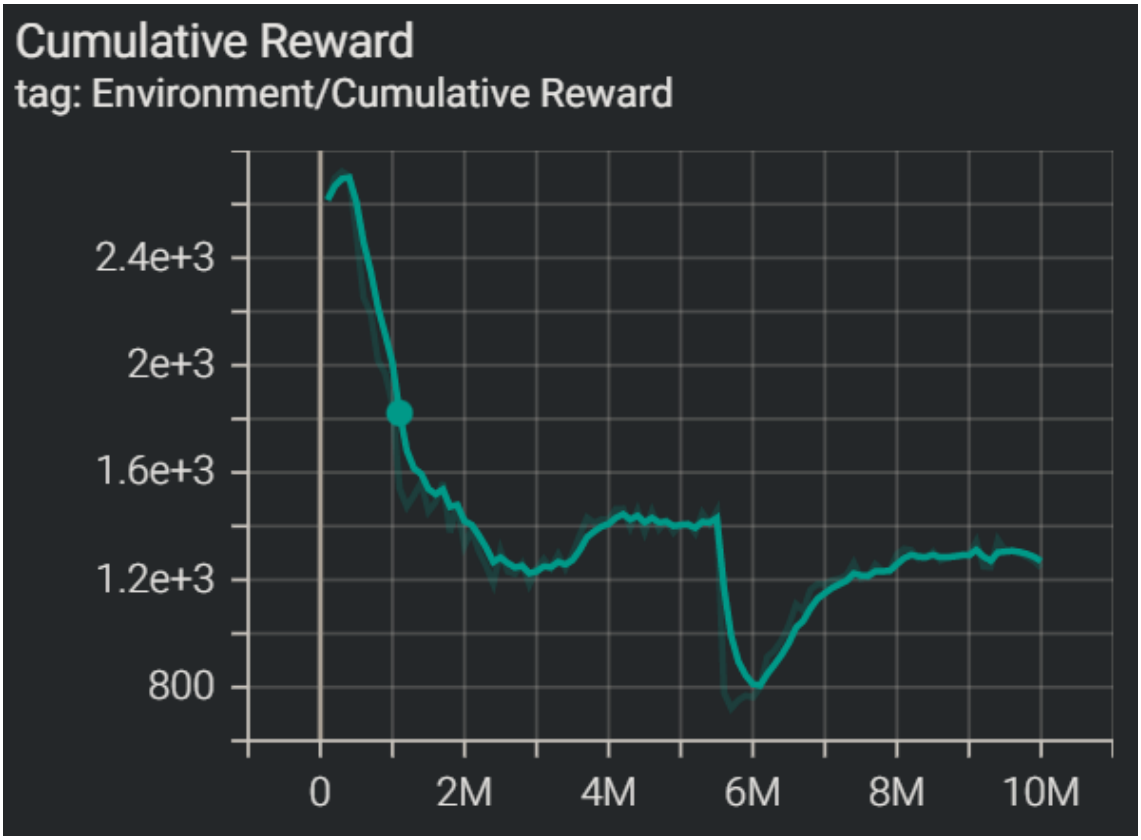

Abbildung .14: ′Cumulative Reward - BeanShooter - Agent B′

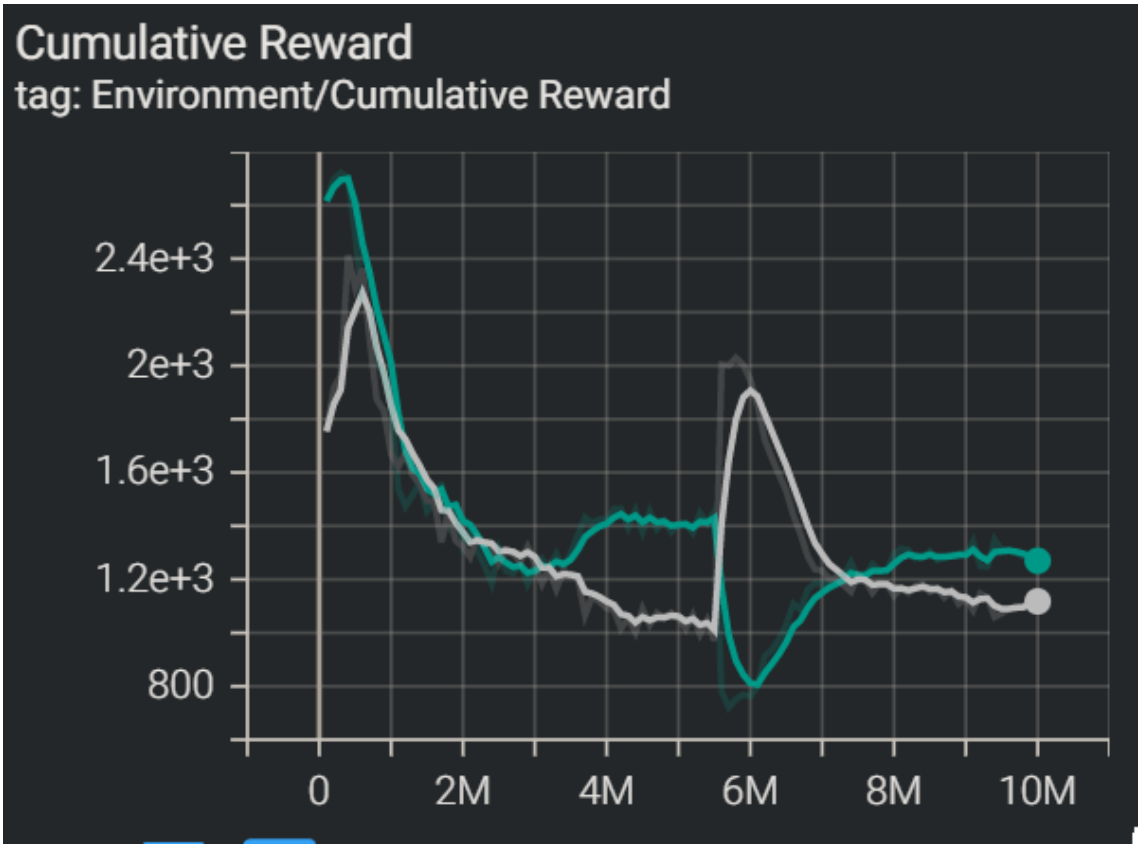

Abbildung .15: ’Cumulative Reward - BeanShooter - Vergleich’

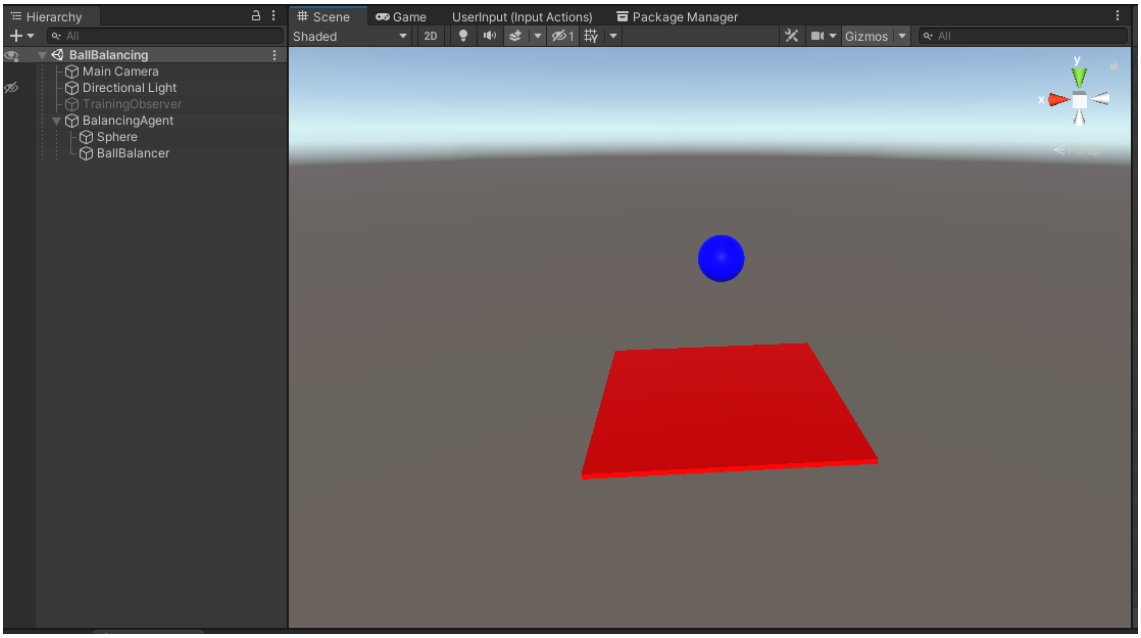

Abbildung .16: ’BallBalancer’ Hierarchie Aufbau

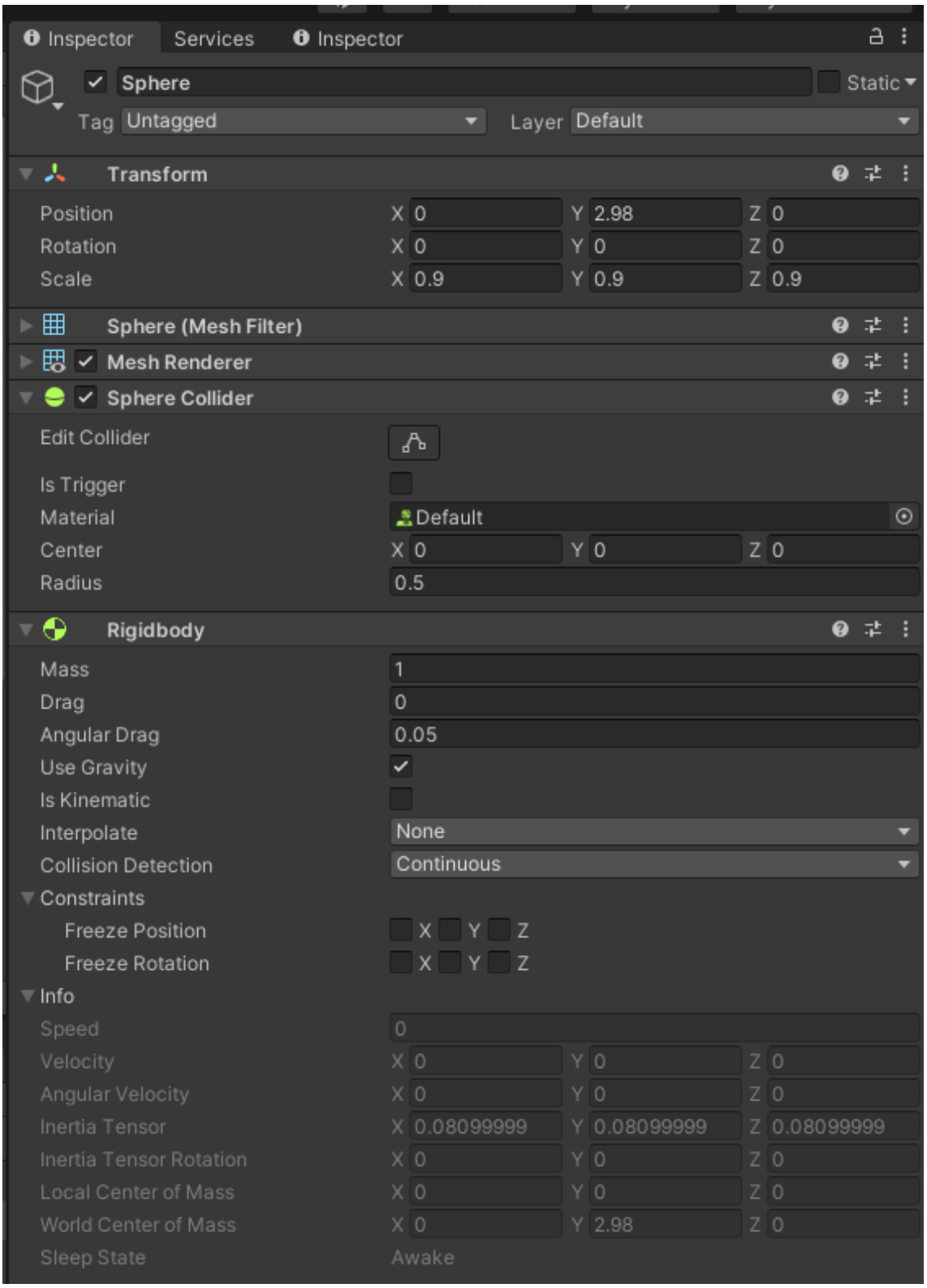

Abbildung .17: ’BallBalancer’ zu balancierende Kugel - Aufbau

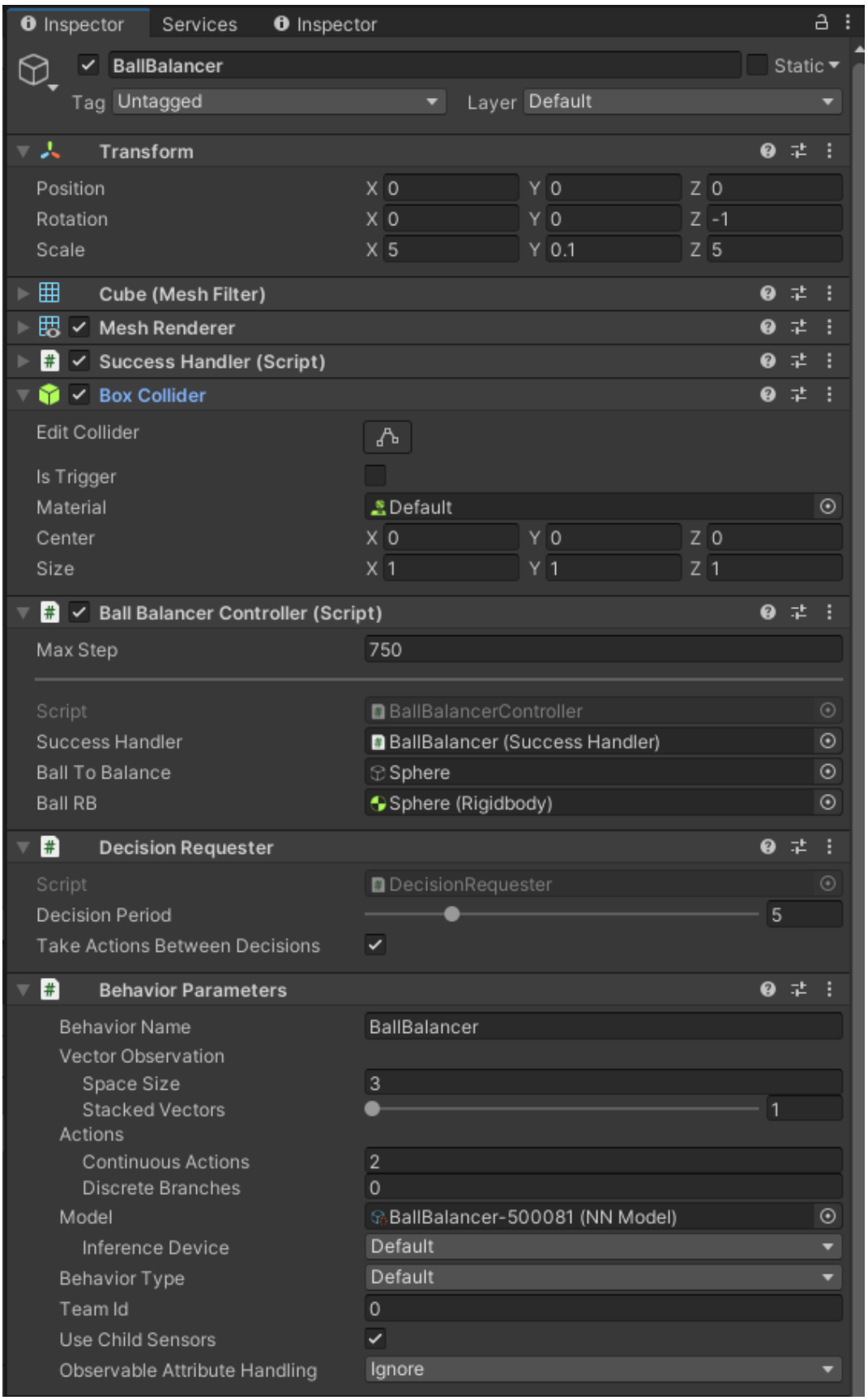

Abbildung .18: ’BallBalancer’ Agent - Aufbau