

Bearbeitungsbeginn: 01.09.2019

Vorgelegt am: 29.05.2020

# **Thesis**

zur Erlangung des Grades

**Master of Science**

im Studiengang Medieninformatik

an der Fakultät Digitale Medien

***Markus Walter Weiß***

***Matrikelnummer: 259149***

**Deep Reinforcement Learning in Games**  
**Integration und Anwendung von ML-Agents in einem bestehenden**  
**Prototyp in der Unity-Engine**

*Erstbetreuerin:* Prof. Dr. Ruxandra Lasowski

*Zweitbetreuer:* Prof. Jirka Dell'Oro-Friedl



# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Master-Thesis selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben.

Furtwangen, den

---

Ort, Datum

---

Markus Walter Weiß



# Abstract

This Thesis deals with the development of an *artificial intelligence* for a given prototype based on *reinforcement learning* using the *ML-Agents* plug-in for the *Unity Engine*. To this end, this thesis first introduces to the basics of *ML-Agents* and gives an overview of the theoretical background. An *example-environment* developed by *ML-Agents* is used to integrate *ML-Agents* into the prototype. For that purpose, initially the configuration is explained, followed by a description of application and evaluation. Within this thesis several experiments were carried out, which served to adapt the configuration. Further experiments with the prototype proved that after training its artificial intelligence was almost as powerful as that of the *example-environment*. Finally this thesis presents the prospect of a usable *artificial intelligence* for the implementation of the prototype.



# Zusammenfassung

Inhalt dieser Thesis ist die Entwicklung einer *Artificial Intelligence* für einen vorgegebenen Prototyp auf Basis des *Reinforcement Learning* unter Anwendung des Plug-ins *ML-Agents* für die *Unity-Engine*. Dabei führt diese Arbeit zunächst in die Grundlagen von *ML-Agents* ein und gibt einen Überblick über den theoretischen Hintergrund. Auf Basis eines von *ML-Agents* erstellten *Example-Environment* wird *ML-Agents* in den Prototyp integriert. Zu diesem Zweck wird zunächst die Konfiguration erklärt, gefolgt von einer Beschreibung der Anwendung und Auswertung. Innerhalb der Thesis wurden Experimente dazu genutzt, die Konfiguration zu verbessern. Weitere Experimente konnten, verglichen mit dem *Example-Environment*, ein ähnlich leistungsfähige *Artificial Intelligence* belegen. Die Thesis stellt für eine spätere Umsetzung des Prototyps eine leistungsfähige *Artificial intelligence* in Aussicht.



# Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Anfertigung dieser Masterarbeit unterstützt und motiviert haben. Zuerst möchte ich den Kollegen und Mitgründern des Entwicklerstudios *Couch in the Woods Interactive* danken, für die Chance, die Abschlussarbeit für das gemeinsame Unternehmen zu verfassen. Anschließend möchte ich mich bei allen Korrekturlesern bedanken, die mir in der Endphase der Thesis geholfen haben.



# Vorwort



# Inhaltsverzeichnis

<b>Abstract</b>	<b>iii</b>
<b>Zusammenfassung</b>	<b>v</b>
<b>Abkürzungsverzeichnis</b>	<b>xix</b>
<b>I. Einleitung, Grundlagen</b>	<b>1</b>
<b>1. Einleitung</b>	<b>3</b>
1.1. Kontext und Motivation . . . . .	3
1.2. Verwandte Arbeiten . . . . .	3
1.3. Forschungsfrage . . . . .	4
1.4. Struktur der Thesis . . . . .	5
<b>2. Grundlagen zu ML-Agents</b>	<b>7</b>
2.1. Reinforcement Learning in ML-Agents . . . . .	7
2.2. Der Markov Decision Process . . . . .	7
2.3. Der Markov Decision Process in Unity . . . . .	8
2.4. Das Unity-Environment von ML-Agents . . . . .	10
2.4.1. Das Example-Environment SoccerTwos . . . . .	10
2.4.2. Observations and Actions . . . . .	10
2.5. Das Python-Environment von ML-Agents . . . . .	14
<b>3. Konfiguration von ML-Agents</b>	<b>17</b>
3.1. Trainingsalgorithmen . . . . .	18
3.1.1. Proximal Policy Optimisation . . . . .	18
3.1.2. Soft Actor-Critic . . . . .	24
3.2. Reward Signals . . . . .	24
3.2.1. Extrinsic Reward . . . . .	24
3.2.2. Intrinsic Curiosity Module . . . . .	24
3.2.3. Generative Adversarial Imitation Learning . . . . .	27
3.3. Imitation Learning . . . . .	27

3.4. Curriculum Learning und Self-Play . . . . .	28
3.4.1. Curriculum Learning . . . . .	28
3.4.2. Adversarial Self-Play . . . . .	28
3.5. Recurrent Neural Networks . . . . .	29
<b>II. Forschungsdesign, Evaluation und Diskussion</b>	<b>31</b>
<b>4. Integration von ML-Agents</b>	<b>33</b>
4.1. Anpassung eines Prototyps für ML-Agents . . . . .	33
4.2. Implementierung eines Prototyps für ML-Agents . . . . .	35
4.2.1. Agent Soccer . . . . .	36
4.2.2. Behaviour Parameter . . . . .	37
4.2.3. Decision Requester . . . . .	38
4.2.4. Raycast Sensor Component . . . . .	38
<b>5. Anwendung von ML-Agents</b>	<b>41</b>
5.1. Konfiguration von ML-Agents in einem Prototyp . . . . .	41
5.1.1. Konfiguration des Computers . . . . .	41
5.1.2. Konfiguration des Python-Environment . . . . .	41
5.1.3. Komplexität des Prototyps . . . . .	42
5.2. Experiment 1: Training des SoccerTwos-Environment . . . . .	43
5.2.1. Wahl der Hyperparameter . . . . .	43
5.2.2. Training zur Reproduzierbarkeit . . . . .	43
5.3. Experiment 2: Training des Prototyp-Environment . . . . .	44
5.3.1. Wahl der Hyperparameter . . . . .	45
5.4. Experiment 3: Training des Prototyp-Environment mit Erweiterung . . . . .	45
5.4.1. Wahl der Hyperparameter . . . . .	47
<b>6. Auswertung von ML-Agents</b>	<b>51</b>
6.1. Auswertung der Proximal Policy Optimisation . . . . .	53
6.2. Auswertung des Self-Play . . . . .	55
6.3. Auswertung von Curiosity Learning . . . . .	56
6.4. Auswertung von Behaviour Cloning und Generative Adversarial Imitation Learning . . . . .	58
6.4.1. Auswertung von Behaviour Cloning . . . . .	58
6.4.2. Auswertung von Generative Adversarial Imitation Learning . . . . .	58
6.5. Auswertung mit der Unity-Engine . . . . .	58

<b>7. Resultate und Diskussion</b>	<b>61</b>
7.1. Experiment 1: Auswertung des SoccerTwos-Environment . . . . .	61
7.1.1. Auswertung des Trainings für die Buffergröße . . . . .	61
7.1.2. Auswertung zur Reproduzierbarkeit . . . . .	64
7.1.3. Auswertung unter Verwendung des Unity-Editors . . . . .	66
7.2. Experiment 2: Auswertung der Prototyp-Umgebung . . . . .	66
7.2.1. Vorbereitendes Training . . . . .	66
7.2.2. Vollständiges Training . . . . .	69
7.2.3. Auswertung unter Verwendung des Unity-Editors . . . . .	72
7.3. Experiment 3: Auswertung der Prototyp-Umgebung mit Erweiterung . . . . .	72
7.3.1. Auswertung unter Verwendung des Unity-Editors . . . . .	73
<b>III. Zusammenfassung und Ausblick</b>	<b>77</b>
<b>8. Fazit und Ausblick</b>	<b>79</b>
8.1. Fazit . . . . .	79
8.2. Ausblick . . . . .	80
<b>Literatur- Quellenverzeichnis</b>	<b>85</b>
<b>Anhang</b>	<b>87</b>
1. Experiment 3: Weitere Trainingsdaten . . . . .	87
2. SoccerTwos Parameter Screenshots . . . . .	89
3. Trainingsinstanzen in SoccerTwos . . . . .	91
4. Screenshot der max steps des Crawler Static-Environment . . . . .	91



# Abbildungsverzeichnis

2.1.	Markov Decision Process . . . . .	8
2.2.	Markov Decision Process im Unity-Environment . . . . .	9
2.3.	Screenshot des SoccerTwos-Environment . . . . .	10
2.4.	Raycast Sensor Component . . . . .	13
2.5.	Markov Decision Process in Unity mit Python . . . . .	15
3.1.	Die Konfigurationsdatei von ML-Agents . . . . .	17
3.2.	Policy Gradient Verfahren . . . . .	20
3.3.	Policy Gradient Verfahren mit Kullback-Leiber-Divergenz . . . . .	21
3.4.	Proximal Policy Optimisation. Visualisierung des Clip Verfahrens . . . . .	22
3.5.	Die Proximal Policy Optimisation mit Kullback-Leiber-Divergenz . . . . .	23
3.6.	Intrinsic Curiosity Module . . . . .	26
3.7.	Aufbau des Behaviour Cloning in Unity . . . . .	27
3.8.	Aufbau des Self-Play (SP) in Unity . . . . .	29
4.1.	Screenshot der SoccerTwos Szene in der Topdown Ansicht . . . . .	34
4.2.	Screenshot der NEON SHIFTER Szene in der Topdown Ansicht . . . . .	34
4.3.	Steuerungssachsen von NEON SHIFTER . . . . .	36
5.1.	Reinforcement Learning in Kombination mit Behavior Cloning, Generative Adversarial Imitation Learning und Curriculum Learning . . . . .	47
6.1.	Screenshot von Tensorboard . . . . .	52
6.2.	Proximal Policy Optimisation mit Tensorboard . . . . .	53
6.3.	Self-Play mit Tensorboard . . . . .	55
6.4.	Curiosity Learning mit Tensorboard . . . . .	56
7.1.	Experiment 1 mit 1.0 und 1.1 . . . . .	63
7.2.	Experiment 1 mit 1.0 und 1.2 . . . . .	65
7.3.	Experiment 2 mit 2.0 und 2.1 . . . . .	68
7.4.	Experiment 2 mit 1.0 und 2.2 . . . . .	71
7.5.	Experiment 3 mit 1.0, 2.2 und 3.0 . . . . .	75
1.	Weitere Ergebnisse des Trainings 3.0 für Imitation Learning, Behaviour Cloning, Generative Adversarial Imitation Learning und Reinforcement Learning	88

## Abbildungsverzeichnis

---

2.	Screenshot der Agent Soccer-Parameter von <i>SoccerTwos</i> im Unity Editor . . . . .	89
3.	Screenshot der Behaviour Parameter-Parameter von <i>SoccerTwos</i> im Unity Editor . . . . .	89
4.	Screenshot der Decision Requester-Parameter von <i>SoccerTwos</i> im Unity Editor	90
5.	Screenshot der Raycast Sensor Component-Parameter von <i>SoccerTwos</i> im Unity Editor . . . . .	90
6.	Screenshot der SoccerTwos-Instanzen . . . . .	91
7.	Screenshot der max steps des Crawler-Environment . . . . .	91

# Tabellenverzeichnis

4.1.	Actions von SoccerTwos und NEON SHIFTER . . . . .	37
4.2.	Actions von SoccerTwos und NEON SHIFTER im Unity-Editor mit Summe	38
4.3.	Die Raycast Sensoren von SoccerTwos und NEON SHIFTER . . . . .	38
4.4.	Tags von SoccerTwos und NEON SHIFTER . . . . .	39
5.1.	Aufbau des Computers . . . . .	41
5.2.	Observations von SoccerTwos und NEON SHIFTER . . . . .	42
5.3.	Komplexität von SoccerTwos und NEON SHIFTER . . . . .	43
5.4.	Hyperparameter für SoccerTwos . . . . .	44
5.5.	Hyperparameter für Proximal Policy Optimisation . . . . .	46
5.6.	Hyperparameter für Proximal Policy Optimisation mit Generative Adversarial Imitation Learning und Behaviour Cloning . . . . .	49
6.1.	Tensorboard Auswertungstabelle des Proximal Policy Optimisation . . . . .	54
6.2.	Tensorboard Auswertungstabelle für Curiosity Learning . . . . .	57
6.3.	Tensorboard Auswertungstabelle für Generative Adversarial Imitation Learning . . . . .	58
7.1.	Experiment 1 Bewertungstabelle 1.0 und 1.1 . . . . .	62
7.2.	Experiment 2 Bewertungstabelle 2.0 und 2.1 . . . . .	67
7.3.	Bewertungstabelle mit Experiment 1.0 und 2.2 . . . . .	70
7.4.	Experiment 3 Ergebnistabelle mit 1.0, 2.2 und 2.0 . . . . .	74



# Abkürzungsverzeichnis

**AI** Artificial Intelligence. 3–5, 79, 80

**BC** Behaviour Cloning. 27, 28, 46, 58, 73, 88

**CL** Curriculum Learning. 28, 46, 56, 57, 73, 80

**GAIL** Generativ Adversarial Imitation Learning. 27, 28, 46, 58, 73, 88

**GAN** Generative Adversarial Network. 28

**ICM** Intrinsic Curiosity Modul. 24–26, 57

**IL** Imitation Learning. 27, 45, 88

**KL** Kullback-Leiber. 19, 21–23

**LSTM** Long-Short-Term-Memory. 29

**MDP** Markov Decision Process. 5, 7–9, 26

**PGV** Policy-Gradient-Verfahren. 18, 19

**PPO** Proximal Policy Optimization. 18, 22–24, 45, 53, 54, 62, 67

**RL** Reinforcement Learning. 3–5, 7, 18, 46, 73, 79, 80, 88

**RNN** Recurrent Neural Network. 29, 80

**SAC** Soft-Actor-Critic. 18, 24, 80

**SP** Self-Play. xv, 12, 28, 29, 55, 62, 67, 69

**TRPO** Trust Region Policy Optimisation. 19, 21, 22



# Teil I.

## Einleitung, Grundlagen



# 1. Einleitung

## 1.1. Kontext und Motivation

*NEON SHIFTER* ist ein Weltraum- und Sportspiel, mit bis zu vier Spielern. Unter Verwendung von Physikelementen, setzt es einen, inhaltlich an Fußball oder Hockey erinnernden, Spielaufbau um. Auf Basis der *Unity-Engine* wurde für *NEON SHIFTER* ein Prototyp entwickelt, der die Kernelemente des Spiels, also das Spiel mit einem Ball unter Anwendung von Gravitation sowie den Vier-Spieler Ansatz ermöglicht.[50]

Für das fertige Spiel ist geplant, dass es über eine Artificial Intelligence (AI) verfügt. Das Entwicklerstudio *Couch in the Woods Interactive* befindet sich zum Abgabetermin dieser Thesis Ende Mai 2020 vor der Entwicklungsphase für das Computerspiel *NEON SHIFTER*.

*Unity-Technologies* bietet mit der Entwicklung des Plug-ins *ML-Agents* die Möglichkeit, aktuelle Erkenntnisse der Wissenschaft aus dem Bereich des Reinforcement Learning (RL) in der *Unity-Engine* anzuwenden [40, S.8].

Im Kontext der Entwicklung von *NEON SHIFTER* stellt sich für das Entwicklerteam von *Couch in the Woods Interactive* die Frage, ob die Entwicklung einer leistungsfähigen AI durch Anwendung von *ML-Agents* für *NEON SHIFTER* möglich ist. Der Begriff Leistungsfähig wird hier synonym zu den *Example-Environments* von *ML-Agents* definiert. *ML-Agents* soll deshalb im Zuge dieser Thesis, auf Basis von *Example-Environments*, in den Prototyp integriert und angewendet werden. Die Ergebnisse sollen *Couch in the Woods Interactive* die Entscheidung erleichtern, ob *ML-Agents* in den Entwicklungsprozess integriert wird.

## 1.2. Verwandte Arbeiten

*ML-Agents* verfügt über eine Dokumentation zur Anwendung und Integration von *ML-Agents* und der implementierten Algorithmen [22]. Darüber hinaus bietet *ML-Agents* eine Reihe von Beispielumgebungen bzw. *Example-Environments* [11]. Diese werden dazu verwendet, die Integration von *ML-Agents* in den *NEON SHIFTER*-Prototyp zu vereinfachen.

Mit dem *Example-Environment SoccerTwos* bietet *ML-Agents* eine Umgebung [11, SoccerTwos], die auf Grund seiner Ähnlichkeit zum Prototyp, zur Integration genutzt werden kann. *SoccerTwos* bildet eine an Fußball erinnernde Szene mit vier Akteuren. Diese kön-

## *1. Einleitung*

---

nen von einer Artificial Intelligence (AI) oder menschlichen Spielern gesteuert werden [5, Agents].

### **1.3. Forschungsfrage**

Aus der Motivation dieser Arbeit, wird folgende Forschungsfrage abgeleitet:

Ist es mit *ML-Agents* möglich, eine AI für das Game *NEON SHIFTER* zu erstellen, die vergleichbar leistungsfähige Ergebnisse wie ein *Example-Environment* von *ML-Agents* liefert?

Diese Thesis dient nicht dem Zweck, eine leistungsfähige AI für *NEON SHIFTER* zu erstellen. Die Ergebnisse sollen auf Basis des Prototypen in Aussicht stellen, ob dies unter Anwendung von *ML-Agents* möglich ist. Zur Beantwortung der Forschungsfrage wurden die unten aufgelisteten Teilfragen formuliert:

- Was ist *ML-Agents* und wie integriert es Reinforcement Learning (RL)?
- Wie wird *ML-Agents* konfiguriert und lässt sich dabei ein Einblick in den theoretischen Hintergrund geben?
- Wie kann *ML-Agents* integriert werden?
- Wie kann *ML-Agents* angewendet werden?
- Wie kann *ML-Agents* ausgewertet werden?
- Wie muss ein *Example-Environment* konfiguriert werden und lassen sich diese Ergebnisse reproduzieren?
- Erzeugt die Anwendung von *ML-Agents* im Prototyp eine leistungsfähige AI?
- Lässt sich die Leistungsfähigkeit durch Anpassungen von Trainingsergebnissen verbessern?
- Wie schlägt sich die AI gegenüber eines menschlichen Spieler?
- Stellen die Trainingsergebnisse im Vergleich zu *SoccerTwos* eine verwertbare AI in Aussicht?

## 1.4. Struktur der Thesis

In Kapitel 2 wird auf Basis des Markov Decision Process (MDP) die Integration von RL in *ML-Agents* erläutert. Anschließend werden anhand eines von *ML-Agents* bereitgestellten *Example-Environment* die Umsetzung im *Unity*-Editor und die notwendigen Schritte zur Konfiguration und Anwendung aufgeführt.

Zur Darstellung, wie *ML-Agents* eine Schnittstelle zwischen RL und *Unity* umsetzt, wird als Basis die Konfigurationsdatei von *ML-Agents* verwendet. Kapitel 3 zeigt dabei über die Dokumentation hinaus einen Einblick in den theoretischen Hintergrund von RL.

Der Prototyp *NEON SHIFTER* wird in Kapitel 4 unter Anwendung der gesammelten Grundlagen mit *ML-Agents* kombiniert. Dabei dient das *Example-Environment SoccerTwos* als Vergleich zum Prototyp, auf dessen Basis die Integration umgesetzt wird.

Kapitel 5 führt in die Anwendung von *ML-Agents* ein. Angefangen mit dem ersten Experiment in Abschnitt 5.2 für das *Example-Environment SoccerTwos*, um für den späteren Prototyp Vergleichsdaten zu erhalten. Weiter wird mit diesem Experiment untersucht, wie die *Buffer*-Größe des neuronalen Netzes zu wählen ist. Darüber hinaus versucht die Wiederholung des vergleichsweise besseren Trainings eine Reproduzierbarkeit zu belegen.

Das zweite Experiment in Abschnitt 5.3 liefert Auskunft darüber, wie das neuronale Netz von *NEON SHIFTER* im Vergleich zu *SoccerTwos* im Hinblick auf die größere Komplexität des *Environments* angepasst werden muss. Anschließend werden zu dieser Konfiguration mehrere Trainings ausgeführt.

Das dritte Experiment in Abschnitt 5.4 untersucht unter Anwendung von Recherchergebnissen [49, Abb. 1], ob die Trainingsergebnisse von *NEON SHIFTER* verbessert werden können.

Die Auswertung von Trainingsdaten findet in *ML-Agents* unter Anwendung von *Tensorboard* statt. *ML-Agents* hat einerseits *Tensorboard* zur Auswertung vorkonfiguriert, andererseits liefert es Empfehlungen zur Anwendung und Auswertung mit.[17] Kapitel 6 zeigt dabei das Auswertungsverfahren auf Grundlage der Dokumentation.

In Kapitel 7 und 8 werden die Experimente diskutiert und zusammengefasst. Weiterhin wird ein Ausblick gegeben wie die Forschung fortgesetzt werden könnte. Dabei beantwortet das Fazit, ob *ML-Agents* in den Entwicklungsprozess von *NEON SHIFTER* integriert werden soll, um eine verwertbare AI zu erzeugen.



## 2. Grundlagen zu ML-Agents

*ML-Agents* ist ein von *Unity-Technologies* entwickeltes Plug-in für die *Game-Engine Unity* [22, ML-Agents Toolkit Overview]. Dieses bietet Forschern und Spieleentwicklern eine Plattform, um wissenschaftliche Erkenntnisse aus dem Bereich des RL in einer Entwicklungs-umgebung für Computerspiele anzuwenden [40, S.8]. *Unity-Technologies* veröffentlicht einmal monatlich einen neuen Stand von *ML-Agents* [4, Releases & Documentation]. Im Rahmen der Thesis wurde mit Version *Beta 0.14.1* auf dem Stand vom 26.02.2020 gearbeitet.

Da noch zum Zeitpunkt der Thesis, am 01.05.2020 das erste offizielle Release von *ML-Agents* stattfand [4, ML-Agents Release 1], sind Teile der zitierten Onlinequellen nicht mehr abrufbar. Daher wurden Quellen aus der Dokumentation im Literaturverzeichnis um einen Pfad in den digitalen Anhang, welcher der Thesis beiliegt, erweitert.

Im folgenden Kapitel wird erklärt wie *ML-Agents* RL integriert.

### 2.1. Reinforcement Learning in ML-Agents

Im RL führt ein *Agent* (z. Dt. Akteur) in einem zeitdiskreten *Environment* (z. Dt. Umgebung), in einem Zustand oder *Status s*, eine Aktion bzw. *Action a* aus. Für diese *Action* erhält er einen *Reward r* (z. Dt. Belohnung). Die Aufgabe eines *Agents* ist in erster Linie, den *Reward* zu maximieren. Ein *Agent* muss durch selbstständiges Probieren herausfinden, durch welche *Action* er den größten *Reward* erhält.[48, S.48]

Um den *Reward* zu maximieren, muss ein *Agent* eine *Policy* (z. Dt. Strategie) entwickeln [43, 3.5 Optimal Policy]. Die *Policy* ist dabei vereinfacht gesagt: “[...]a strategy of the agent to select certain actions depending on the current state s.”[43, 3.3 Policies].

In *ML-Agents* übernehmen die *Agents* die Erhebung der Daten. Diese befinden sich innerhalb eines *Unity-Environment*. Sie sammeln *Observations* (z. Dt. Beobachtungen) und führen *Actions* in einem *Environment* anhand der *Policy* aus. Das Auswerten der Daten, also die Berechnung der *Policy*, wird dabei an ein *Python-Environment* weitergeleitet.[22, Key Components]

### 2.2. Der Markov Decision Process

Der Markov Decision Process (MDP), visualisiert in Abbildung 2.1, übernommen aus [48, S.48], zeigt das Zusammenspiel zwischen *Agent* und Umgebung.

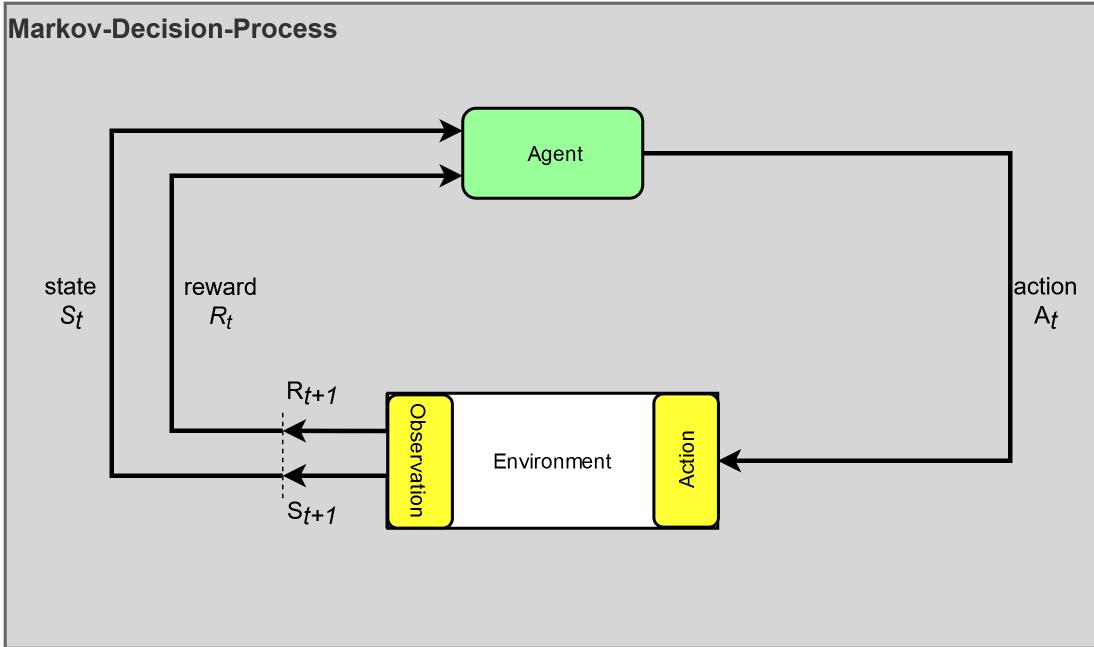


Abbildung 2.1.: Markov Decision Process: Die Abbildung zeigt den MDP (Eigene Darstellung)[48, Abb 3.1]. Eine Veranschaulichung der Interaktion zwischen *Agent* und *Environment*. Dabei zeigt die Abbildung unverändert den Ablauf des MDP nach [48, Abb 3.1], diese wurde aber neu erstellt um besser in die Grafiken der Thesis zu integrieren.

In einem *Environment E*, zum Zeitpunkt  $t$ , im Status  $S_t$ , führt, auf Basis des *Reward R<sub>t</sub>*, ein *Agent* zum Zeitpunkt  $t$  die Aktion  $A_t$  aus. Diese Aktion führt im *Environment E* zu einer Änderung und erzeugt den neuen *Status S<sub>t+1</sub>* und gibt den *Reward R<sub>t+1</sub>* zurück. Mit dem neuen *Status S<sub>t+1</sub>* und dem erhaltenen *Reward R<sub>t+1</sub>* beginnt dieser Prozess erneut. Zusammengefasst werden durch *Actions* Statusänderungen erzeugt. Diese liefern *Rewards* und *Observations*. Dieses von MDP gelieferte Feedback führt durch die Berechnung eines Algorithmus zu einer neuen, angepassten *Policy* bzw. einer Sammlung vom angepassten *Actions*. [48, S.48]

## 2.3. Der Markov Decision Process in Unity

*ML-Agents* integriert den MDP in das *Unity-Environment*. Die Abbildung 2.2 zeigt eine vereinfachte Darstellung der Integration des MDP in ein *Unity-ML-Agents-Environment*.

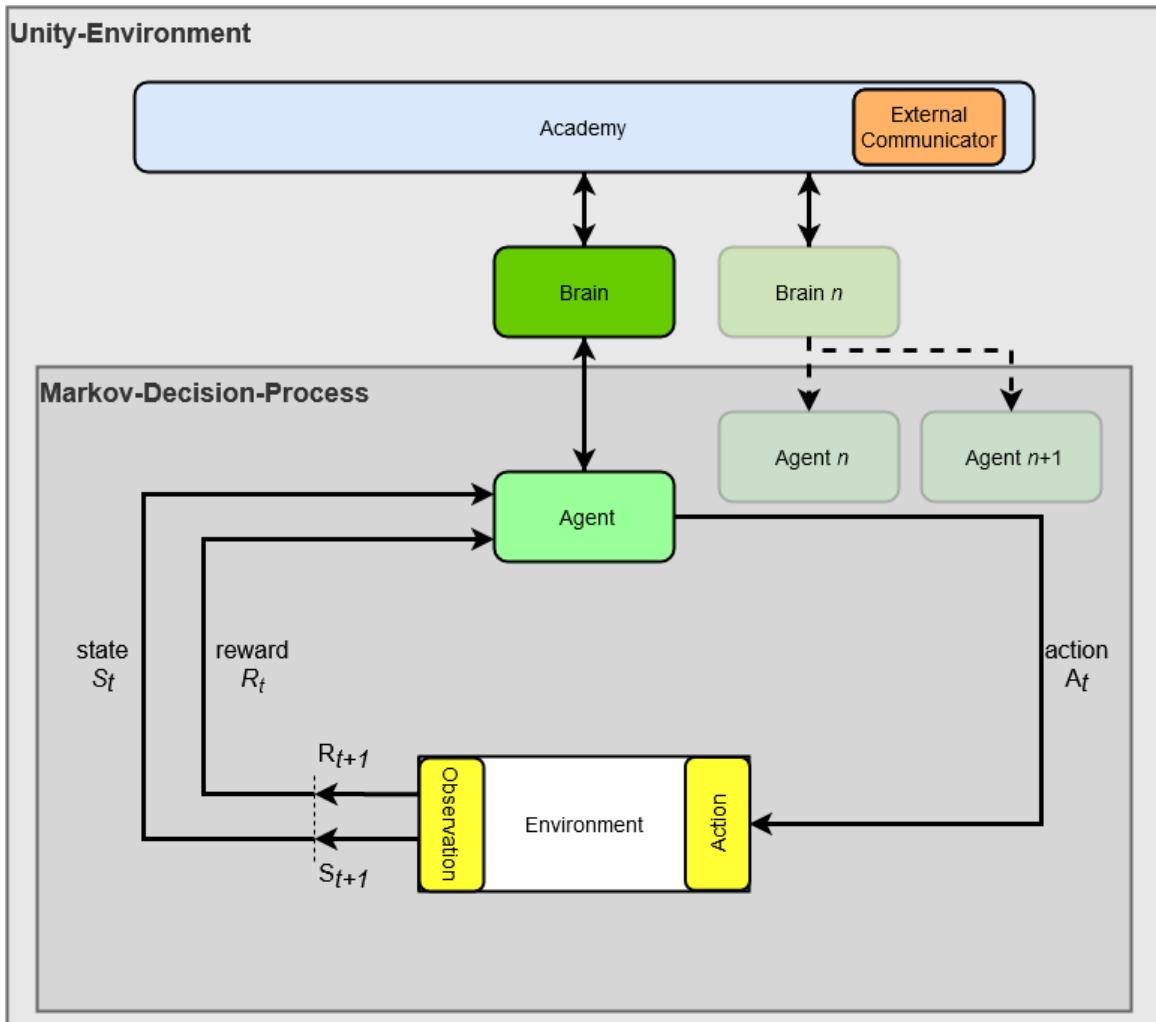


Abbildung 2.2.: *Markov Decision Process im Unity-Environment*: Die Abbildung zeigt den MDP von [48, Abb 3.1], wie er sinngemäß in ein *Unity-Environment* auf Basis der Abbildung (Eigene Darstellung)[40, Figure 2] integriert ist. Diese Abbildungen wurden neu erstellt, um sie besser kombinieren zu können.

Der im *Unity-Environment* befindliche *Agent* erhält einen *Reward* und leitet diesen über ein sogenanntes *Brain* zur *Academy* weiter. Die *Academy* sammelt die Daten aller *Agents* und übermittelt sie an das *Python-Environment*. Dieses berechnet eine Anpassung der *Policy* und liefert die Ergebnisse zurück an die *Academy* und weiter an die *Agents*, welche daraufhin eine neue Handlung ausführen. [22, Key Components]

Dabei sammelt ein *Brain* alle Daten von *Agents*, welche den Namen des *Brains* in den *Behaviour-Parameters*, erklärt in Abschnitt 2.4.2, konfiguriert haben. D.h. wenn verschiedene Verhaltensweisen trainiert werden sollen, werden in einem *Unity-Environment* verschiedene *Brains* eingefügt. Jedes *Brain* steht dabei für ein eigenes Verhalten bzw. eine eigene *Policy*, die einerseits vom *Python-Environment* oder von einem menschlichen Spieler gesteuert werden kann. [42, S.160]

## 2.4. Das Unity-Environment von ML-Agents

An dieser Stelle wird unter Anwendung des *SoccerTwos-Examples* erläutert wie *Actions* und *Observations* verwendet werden.

### 2.4.1. Das Example-Environment SoccerTwos

ML-Agents bietet für Unity eine Reihe an *Example-Environments* [11]. Im Folgenden ist anhand des *SoccerTwos-Environment* die weitere Integration von ML-Agents in Unity erläutert [2, SoccerTwos]. Das *SoccerTwos-Environment* verfügt über die Implementierung eines *Multi-Agent-Environment* und wird im weiteren Verlauf der Thesis als Vergleichsbeispiel dienen [42, S.284]. Die Abbildung 2.3 zeigt eine Screenshot des *SoccerTwos-Environment*.

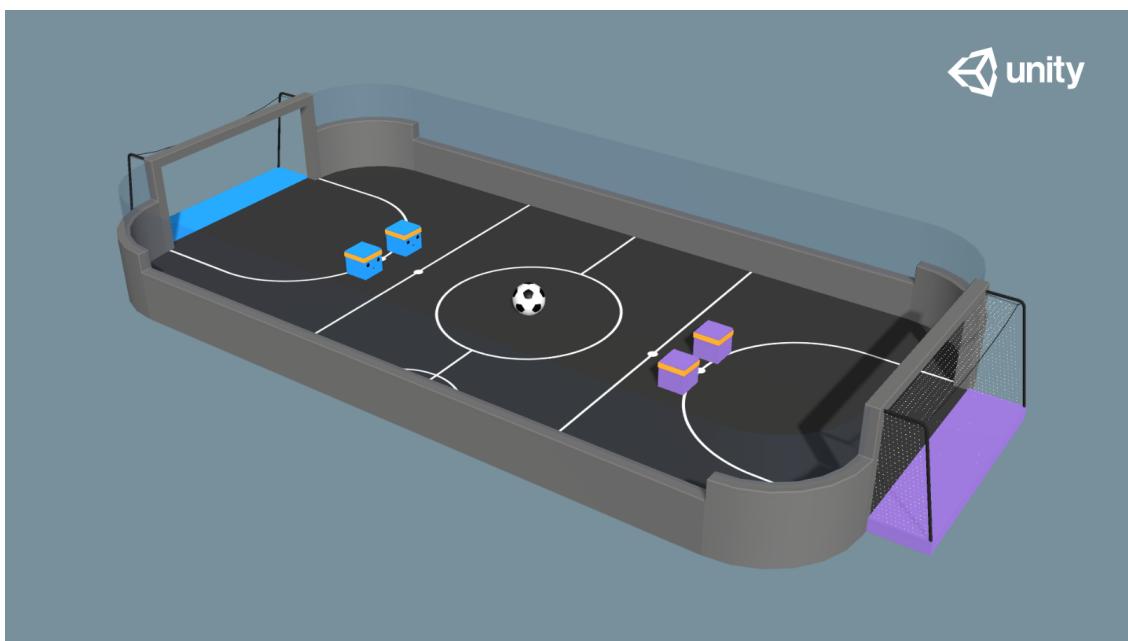


Abbildung 2.3.: Screenshot des SoccerTwos-Environment: Die Abbildung zeigt einen Screenshot des *SoccerTwos-Environments*, welches über ein Spielfeld, Außenwände, vier Spieler, zwei Tore und einen Ball verfügt (Unity-Editor Screenshot)[2].

### 2.4.2. Observations and Actions

Um mit ML-Agents *Observations* an das *Python-Environment* zu senden und von diesem aus die *Policy* zu empfangen, verfügt ein *Agent* im *SoccerTwos-Environment* über die unten aufgeführten Skripte. Diese sind dabei aus dem *SoccerTwos-Example* [2] übernommen worden und werden auf Basis der Dokumentation [5] beschrieben und erklärt.

- *Agent*-Skript: Implementiert die grundsätzliche Verwaltung des *Agents* [5, Agent] [30].
- *Behaviour Parameter*-Skript: Verwaltet *Actions* und *Observations* [5, Agent].

- *Decision Requester*-Skript: Konfigurierbarkeit des Intervalls der Entscheidungen, wenn ein *Agent* selbstständig darüber entscheiden soll [5, Decisions].
- (Optional: *Ray Perception Sensor*-Skript): *Observations* auf Basis von *Raycasts* [5, Raycast Observations].

Im Folgenden befindet sich eine Liste der Funktionen dieser Skripte. Diese wurde auf Basis der Dokumentation [5] erstellt. Im Anschluss werden diese eingehend erklärt.

- *Agent*-Skript, wurde hier in *AgentSoccer* umbenannt: [30, Zeile 5]
  - *Max Step*: Die maximale Zeit, die eine Episode dauern darf [5, Agent Properties].
- *Behaviour Parameters*:
  - *Behaviour Name*: Der Name des *Brains*.[42, S. 160]
  - *Vector Observation*
    - \* *Space Size*: Wie viele Vektoren ein *Agent* in einem *Environment* beobachtet [5, Agent Properties].
    - \* *Stacked Vectors*: Wie viele Vektor-*Observations* gespeichert werden sollen, bevor sie an das *Brain* weitergegeben werden [5, Agent Properties].
  - *Vector Action*:
    - \* *Space Type*: Diskret - dabei werden Vektoren mit *int*-Werten konfiguriert [5, Agent Properties].
    - \* *Space Size*: Kontinuierlich - dabei werden Vektoren mit *float*-Werten konfiguriert [5, Agent Properties].
    - \* *Branches Size*: Für diskrete Vektoren - Die Anzahl der möglichen *Actions* [5, Agent Properties].
  - *Model*: Ein *Model*, dass von dem *Python-Environment* erstellt wird, kann hier eingesetzt und angewendet werden [5, Agent Properties].
    - \* *Inference Device*: Bietet die Möglichkeit, ein zuvor trainiertes *Model* zu verwenden und die Berechnungseinheit zu wählen [5, Agent Properties].
      - *CPU*: wird für die meisten Beispiele schneller sein [6, Using the Unity Inference Engine].
      - *GPU*: wird im Umkehrschluss bei den meisten Beispielen langsamer sein.
  - *Behaviour Type*:
    - \* *Default*: Wird mit der *Policy* von Python trainiert [5, Agent Properties].

## 2. Grundlagen zu ML-Agents

---

- \* *Heuristic Only*: Verwendet die *Heuristic-Method*. Die *Agents* können von menschlichen Spielern trainiert werden [5, Agent Properties].
  - \* *Inference Only*: Die *Agents* werden nur von der *Python-Environment* trainiert [5, Agent Properties].
  - *Team ID*: Für *Multiagents-Teams* bei der Anwendung von SP [5, Agent Properties].
  - *Use Child Sensors*: Ob Sensoren auf *Child-Objects* mitbenutzt werden sollen, [5, Agent Properties].
- *Decision Requester*:
  - *Decision Period*: In welchem Intervall ein *Agent* Entscheidungen erhält [5, Decision].
- *Raycast Sensor Component*:
  - *SensorName*: Der Name des Sensors [5, Raycast Observations].
  - *Detectable Tags*: Eine Liste an *Gameobject-Tags* die sich in der *Unity-Umgebung* befinden, und erkannt werden sollen [5, Raycast Observations].
  - *Layer*: Liste an *Layern*, die erkannt werden sollen [5, Raycast Observations].
  - Einstellung der *Raycasts*[5, Raycast Observations].

Im Folgenden sind die Begriffe, die einer genaueren Definition bedürfen, erklärt.

**Agent Script - Max Step** Gibt die maximale Dauer einer Episode eines *Unity-Environments* an [5, Agent Properties]. Entspricht dem Zeitraum, der im Fall von *SoccerTwos* den *Agents* zur Verfügung steht, ein Tor zu schießen, bis die Szene zurück gesetzt wird. Die *Max Step* entspricht dabei in *ML-Agents* den *Unity Fixed-Update Steps* [38, Zweiter Post].

**Behaviour Parameter - Brainname** Der *Brainname* entscheidet, welches *Brain*, welchem *Agent* zugewiesen ist, d.h. welches Verhalten der *Agent* ausführt und trainiert[18, Observing the Environment]. Dies wird im Abschnitt 3 erläutert.

**Behaviour Parameter - Vector-Observation Size** Die *Behaviour Parameter* geben die Anzahl der *Observations*-Vektoren im *Environment* an[5, Agent Properties]. Im Beispiel von *SoccerTwos* wird die *Observation* über den *Raycast Sensor Component* gesteuert und muss hier nicht angegeben werden [41, Abschnitt 3]. (siehe auch Abschnitt:2.4.2)

**Behaviour Parameter - Vector-Action** Gibt die Steuerungsachsen des *Agent* an [5, Agent Properties].

**Behaviour Parameter - Model** Hier kann ein trainiertes Modell eines *Brains* eingesetzt werden [5, Agent Properties].

**Raycast Sensor Component** Der *Raycast Sensor Component* dient dem Erkennen von *Game-Objects* in Unity. Über verschiedene Parameter können *Raycasts* eingestellt werden, die in entsprechende Richtungen gestreut werden. Über eine Tag-Liste können die relevanten *Game-Objects* eingetragen werden. [5, Raycast Observations]

Die Abbildung 2.4 zeigt dessen Aufbau.

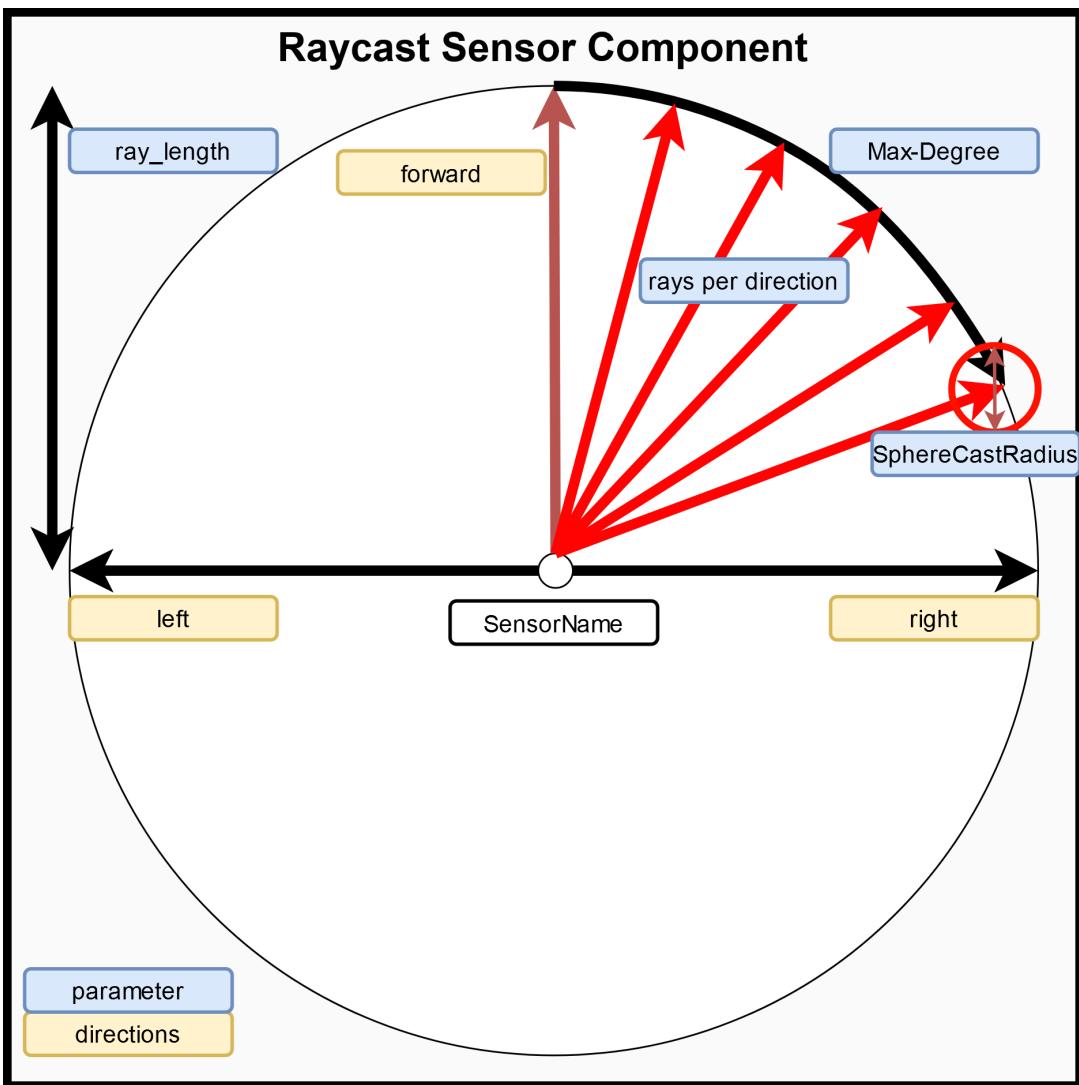


Abbildung 2.4.: Raycast Sensor Component: Zeigt den Aufbau des *Raycast Sensor Component*. Diese Grafik wurde erstellt unter Anwendung der gesammelten Erkenntnisse aus [41] und [5, Raycast Observations].

Ausgehend vom Nullpunkt des Sensors ist der *forward*-Vektor der erste *Raycast*. Von diesem aus werden unter anderem folgende Parameter für *left* und *right* gleichmäßig konfiguriert (Eigene Darstellung)[41], [5, Raycast Observations].

- *ray-length*: Welche Länge der *Raycast* beträgt [41].

- *max-degree*: Der Winkel des äußersten *Raycasts* ausgehend vom *forward*-Vektor [41].
- *rays per direction*: Wie viele *Raycasts* zwischen äußerstem Winkel *max-degree* und dem *forward-Raycast* liegen [41].
- *Observations Stacks*: Wie viele vergangene *Observations* gespeichert werden [41].
- *Sphere Cast Radius*: Was um einen *Raycast* beobachtet werden kann [41].

## 2.5. Das Python-Environment von ML-Agents

Im folgenden Abschnitt wird die Anbindung des *Python-Environment* an das *Unity-Environment* erläutert.

Die Grafik 2.5 zeigt, wie das *Python-Environment* über den *External-Communicator* mit der *Academy* verbunden ist [22].

Beim Start des Trainings lädt das *Python-Environment* die Konfigurationsdatei für das Training [14, Training with mlagents-learn]. Die von der *Academy* gelieferten Daten werden dann im *Python-Environment* ausgewertet und die Ergebnisse zurück an die *Academy* gesendet [14, Training with mlagents-learn]. Zusätzlich werden die Daten in *Tensorboard* zur optischen Auswertung veranschaulicht [17]. Die Auswertung mit *Tensorboard* ist im Abschnitt 6 tiefergehend erklärt.

Über die Kommandozeile lassen sich unter anderem auch die *Trainings-ID* und der Pfad zur Konfigurationsdatei angeben. Darüber hinaus kann auch die Anzahl der zu startenden Instanzen von *Unity-Environment* eingegeben werden.[14, Command Line Training Options]

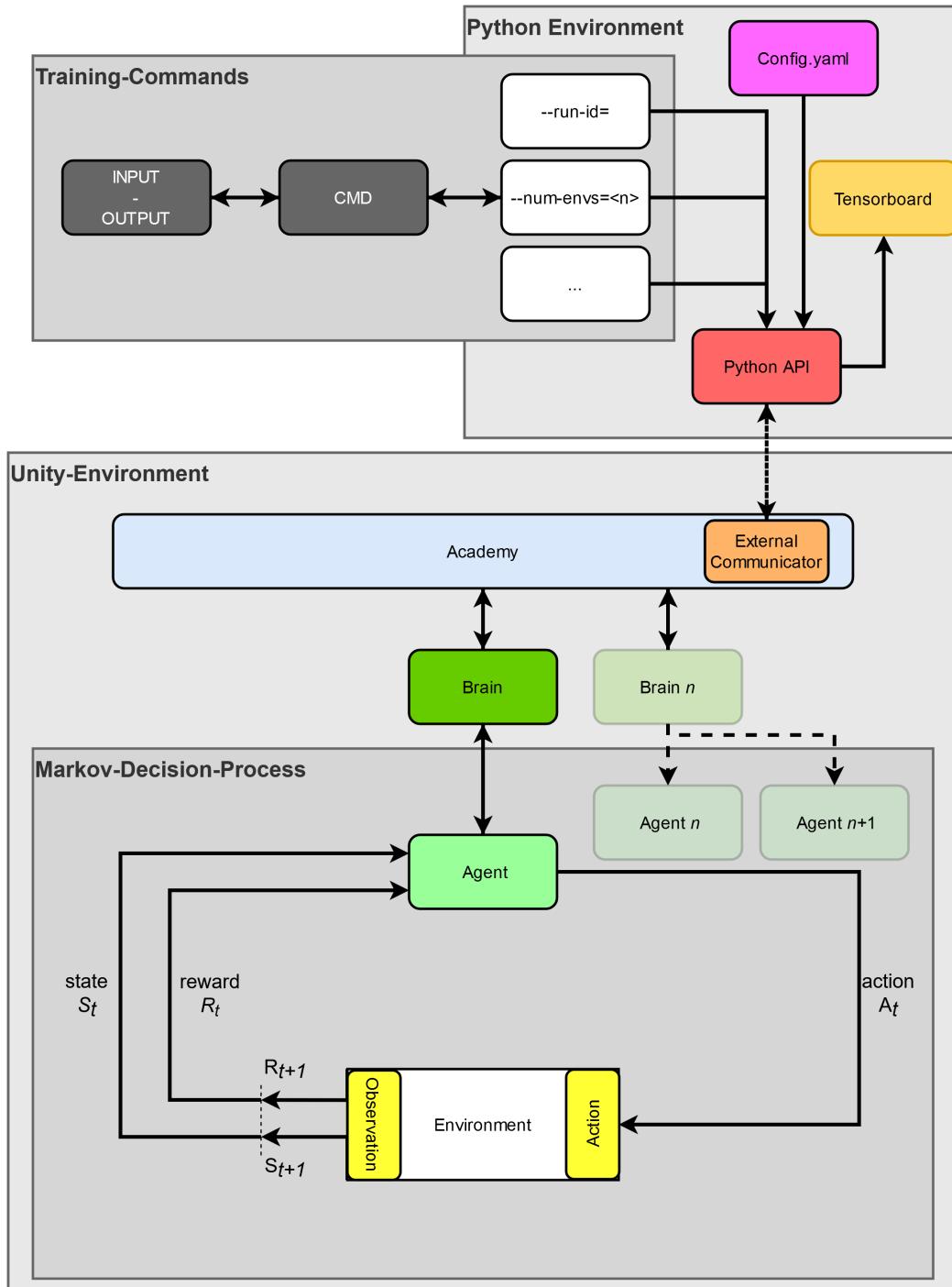


Abbildung 2.5.: Markov Decision Process in Unity mit Python: Die Darstellung erweitert auf Basis der Grafik von (Eigene Darstellung)[48, Abb 3.1] und der Abbildung (Eigene Darstellung)[40, Figure 2] die in 2.5 Abbildung um das *Python-Environment* von ML-Agents. Dabei wird darüber hinaus das *Command Line Window* als Darstellung für *Input/Output* genutzt. Das *Python-Environment* zeigt darüber hinaus die *Config.yaml* und die Verbindung zu *Tensorboard*.



### 3. Konfiguration von ML-Agents

Die in Abschnitt 2.2 aufgeführte *Config.yaml* wird vom *Python-Environment* aufgerufen. Sie dient zur Konfiguration der Hyperparameter.[14, Training with mlagents-learn].

Eine Übersicht der *Config.yaml* zeigt die Abbildung 3.1. Diese wurde aus den Informationen der Dokumentation zusammengetragen [15, 16, 33, 49, 23] und unter Anwendung von [42, S.239, S.194] erweitert.

Die folgenden Abschnitte erläutern die Abbildung 3.1 im Einzelnen.

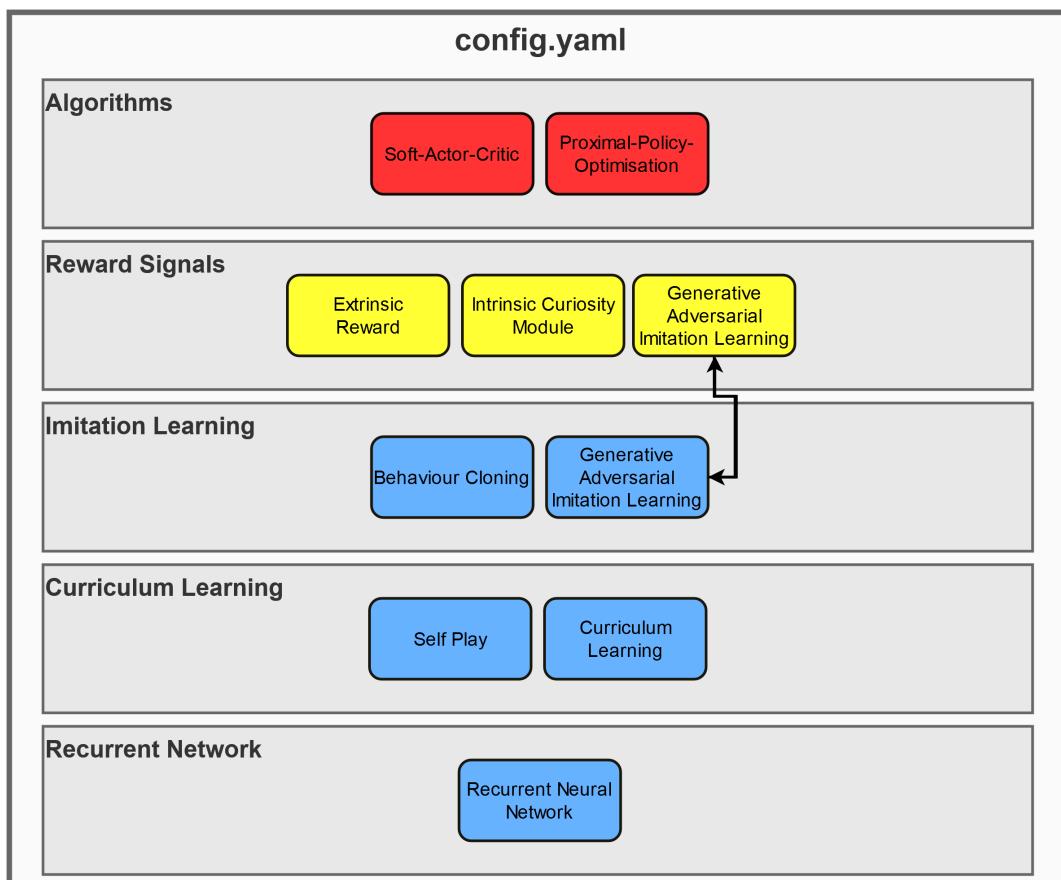


Abbildung 3.1.: Die Konfigurationsdatei von ML-Agents: Übersicht der Einstellungsmöglichkeiten von *ML-Agents* auf einen Blick. Einstellbar über die *Config.yaml* (Eigene Darstellung)[15], [16], [33], [49], [23], [42, S.239, S.194].

## 3.1. Trainingsalgorithmen

*ML-Agents* verfügt über zwei verschiedene Trainings-Algorithmen aus dem Bereich des RL [9, Features]. Die Proximal Policy Optimization (PPO)[46] und die Soft-Actor-Critic (SAC)[35]. In den folgenden zwei Abschnitten werden diese Algorithmen erklärt.

### 3.1.1. Proximal Policy Optimisation

Zur Verwendung der PPO werden in *ML-Agents* über die Konfigurationsdatei die Hyperparameter eingestellt [15, Hyperparameters]. Dabei geben (*hidden\_layers*) und (*hidden\_units*) die Größe des Netzwerks an [15, 24, Hidden Units, Number of Layers]. Zur Ausführung des PPO-Algorithmus, werden Beobachtungen (Observations) eines in Unity ausgeführten Trainings, in regelmäßigen Zeitabständen (*time\_horizon*) zu einer Sammlung an Daten in den sogenannten *Buffer* hinzugefügt [15, Time horizon, Buffer Size]. Ist die festgelegte Größe des *Buffers* (*buffer\_size*) erreicht, wird aus dem *Buffer* eine Anzahl an Stichproben (*batch\_size*) entnommen [15, Batch Size].

Mit dem Gradienten-Verfahren werden diese Stichproben mehrmals (*num\_of\_Epochs*) in zuvor festgelegten Schritten (*learning\_rate*) dazu verwendet, einen Erwartungswert  $E$  auf Basis der (*batch\_size*) zu schätzen  $\hat{E}$ . Dazu wird der gesamte kumulierte, diskontierte<sup>1</sup> *Reward*  $R$  für den Status  $S$  geschätzt  $\hat{V}_s$  [45, S.1]. Dabei entscheidet der Hyperparameter (lambda  $\lambda$ ), wie sehr die *Reward*-Schätzung von der vorangegangen Schätzung beeinflusst wird [15, Lambda]. Anschließend wird für den *Status*  $S$  die logarithmisch verteilte Wahrscheinlichkeit für die möglichen *Actions*  $\log\pi(a|s)$  berechnet [45, S.1].

Daraufhin wird die *Policy* von den *Agents* im *Unity-Environment* durch *Actions* ausgeführt. Diese Ausführung liefert im *Status* für die entsprechende *Action* einen kumulierten, diskontierten *Reward* zurück  $Q(a|s)$ . Um zu berechnen, welche *Action* den größten *Reward* liefert, wird für alle *Policys*  $\pi(a|s)$  der Vorteil  $\hat{A}$  berechnet. Um den Vorteil, wie in Formel 3.1 angegeben, zu errechnen, wird die vor dem Ausführen einer *Action* geschätzte Summe des diskontierten, kumulierten *Reward* für einen Status  $\hat{V}(s)$  herangezogen. Anschließend wird nach dem Ausführen einer Episode, der tatsächlich erhaltenen kumulierten, diskontierten *Rewards*  $Q(s,a)$  für eine *Action* davon subtrahiert. Dieser Prozess wird *Policy-Gradient-Verfahren* genannt.[46, S.2-3]

Die Formel 3.1 zeigt dabei die Formel zur Berechnung des Vorteils [47, Introducing the Advantage function to stabilize learning]. Die Formel 3.2 die Berechnung das Policy-Gradient-Verfahren (PGV) [46, S.3].

$$\hat{A} = \hat{V}_s - Q(s,a) \quad (3.1)$$

---

<sup>1</sup>Der discount-Faktor ist hier der Anteil wie sehr ein *Agent* sich eher auf eine aktuelle *Action*  $a$  zum Zeitpunkt  $t$  oder eher auf eine zukünftige *Action*  $a$  zum Zeitpunkt  $t + 1$  konzentriert. Mit anderen Worten wie kurz- bzw. weitsichtig ein *Agent* handelt. [48, S.55]

$$L^{PG}(\theta) = \hat{E}[\log \pi_{\theta}(a_t | s_t) \hat{A}_t] \quad (3.2)$$

Die Abbildung 3.2 zeigt den Ablauf des *Policy-Gradient-Verfahren*. Das neuronale Netz schätzt den zu erwartenden *Reward* im Status  $S$  unter Einbezug der vorangegangenen Schätzung, konfiguriert durch den Hyperparameter  $\lambda$ . Daraus ergibt sich ( $V(s)$ ) und eine *Policy* mit der höchsten Wahrscheinlichkeit. Diese *Policy* wird angewandt und erzeugt einen neuen *Reward*. Der zuvor geschätzte *Reward* und der tatsächlich erhaltene *Reward* werden anschließend miteinander verglichen und die ausgeführte *Policy* wird anhand dessen durch das neuronale Netz neu gewichtet. [46, S.2-3]

Die Abbildung 3.2 veranschaulicht diesen Prozess sinngemäß auf Basis von [46, S.2-3] mit den in rot gefärbten Hyperparametern, den lila gefärbten Berechnungen und den grün gefärbten *Policies*. Das neuronale Netz ist dabei blau hinterlegt. Die (*length\_of\_episode*) ist ausgeraubt, da diese nicht als Hyperparameter festgelegt wird, sondern im *Unity-Environment* selbst [5, Agent Properties].

Bei der Anpassung einer *Policy* durch das PGV kann eine neu gewählte *Policy* stark zu einer vergangenen abweichen. Um diese Abweichung einzuschränken, wird zusätzlich das Verhältnis *ratio* der Wahrscheinlichkeitsverteilungen  $\frac{\pi_{\theta}}{\pi_{\theta old}}$  unter Anwendung der Kullback-Leiber (KL)-Divergenz berechnet und entsprechend angepasst und diese Abweichung eingeschränkt. Dieses Verfahren wird Trust Region Policy Optimisation (TRPO) genannt. Die Formel 3.3 zeigt die TRPO. [46, S.2-3]

$$\underset{\theta}{\text{maximize}} \hat{E}_t \left[ \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta old}(a_t | s_t)} \hat{A}_t - \beta KL[\pi_{\theta old}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)] \right] \quad (3.3)$$

Die Abbildung 3.3 zeigt eine sinngemäße Darstellung der TRPO auf Basis von [46, S.2-3].

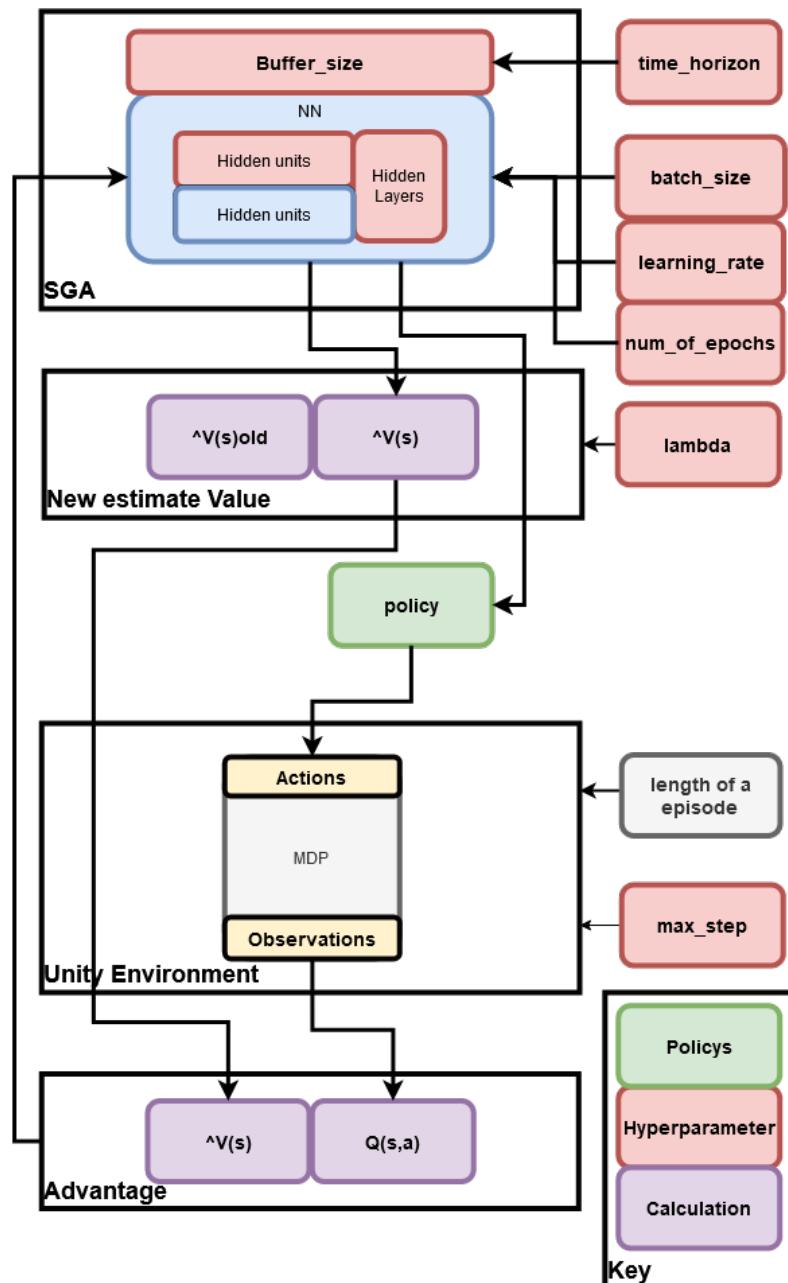


Abbildung 3.2.: Policy Gradient Verfahren: Das *Policy Gradient Verfahren* mit entsprechenden Hyperparametern (Eigene Darstellung)[46, S.2-3].

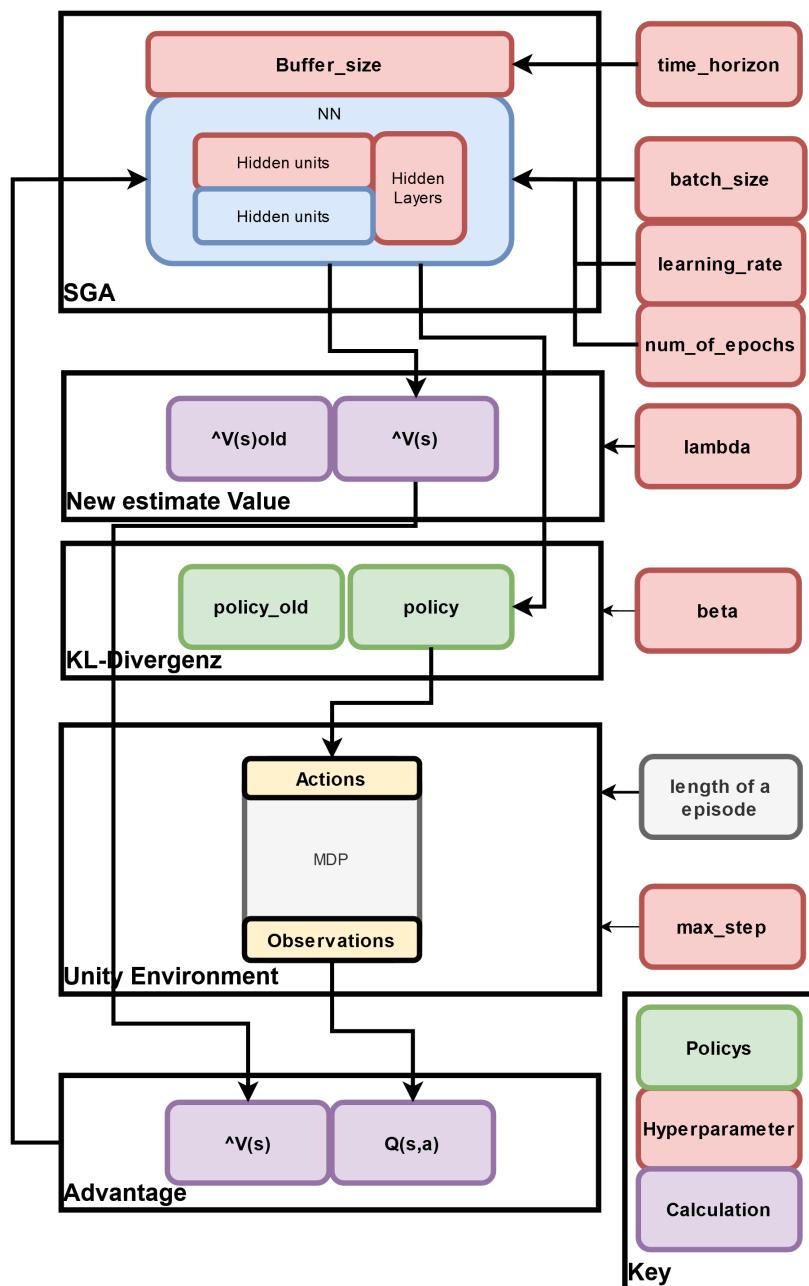


Abbildung 3.3.: Policy Gradient Verfahren mit Kullback-Leiber-Divergenz: Das *Policy Gradient Verfahren* mit der Erweiterung der KL-Divergenz TRPO mit entsprechenden Hyperparametern (Eigene Darstellung)[46, S.2-3].

### 3. Konfiguration von ML-Agents

---

Die Anpassung durch die KL-Divergenz der *Policy* kann zu einem zu schnellen Konvergieren der *Policy* führen. Dadurch können später erlernte *Policies*, die eventuell besser geeignet sind, verpasst werden.[46, S.2-3]

Um dieses Problem zu umgehen wird die PPO eingesetzt. Dabei ist die PPO ein Algorithmus, “[...]that makes policy updates using a surrogate loss function to avoid catastrophic drops in performance.”[24, Absatz 2] Die Formel 3.4 zeigt die mathematische Definition der PPO.[46, S.3] Hier ohne die Anwendung der KL-Divergenz.

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (3.4)$$

Im hinteren Teil davon befindet sich die *Surrogate* (z. Dt. Stellvertreter)-Funktion: ( $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t$ ). In dieser befindet sich der Hyperparameter ( $\epsilon$ ). Dieser gibt an, ab welchem Schwellwert die *Policy* angepasst wird, wenn diese zu sehr abweicht. Der vordere Teil der Formel ist wie bisher, der Anteil der TRPO. Dabei wird  $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$  als  $r_t(\theta)$  notiert. Ist dieser dann größer als die *Surrogate*-Funktion, wird die *Surrogate*-Funktion angewendet und schneidet den *Policy*-Anstieg ab, (*clipping*, z. Dt. abschneiden).[46, S.3]

Eine aus [46, S.3] entnommene grafische Darstellung des *clip*-Verfahrens zeigt die Abbildung 3.4.

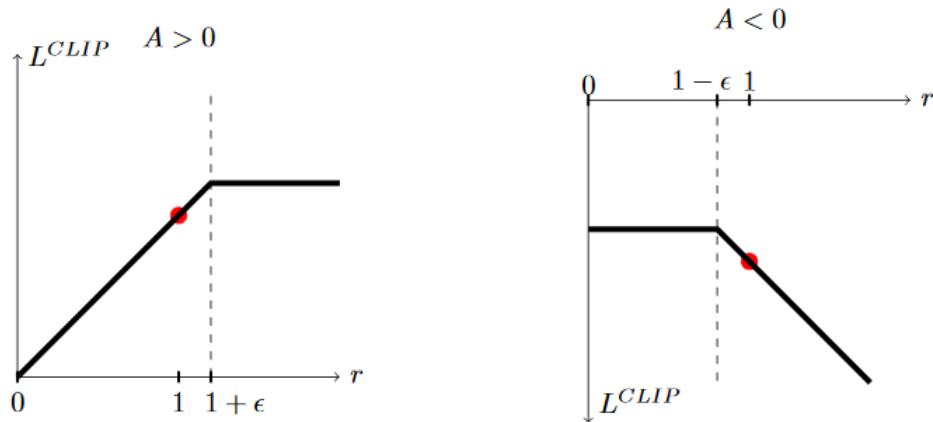


Abbildung 3.4.: Proximal Policy Optimisation. Visualisierung des Clip Verfahrens: Dabei wird die Wahrscheinlichkeitsverteilung der alten *Policy* und der neuen verglichen. Liegt dieser Unterschied dabei über oder unter dem Schwellwert  $\epsilon$ , wird die Veränderung der *Policy* beschränkt.[46, S.3, Figure 1]

Die Abbildung 3.5 zeigt dabei das Policy-Gradient-Verfahren mit der Erweiterung der KL-Divergenz und dem *CLIP*-Verfahren der PPO sowie den entsprechenden Hyperparametern.

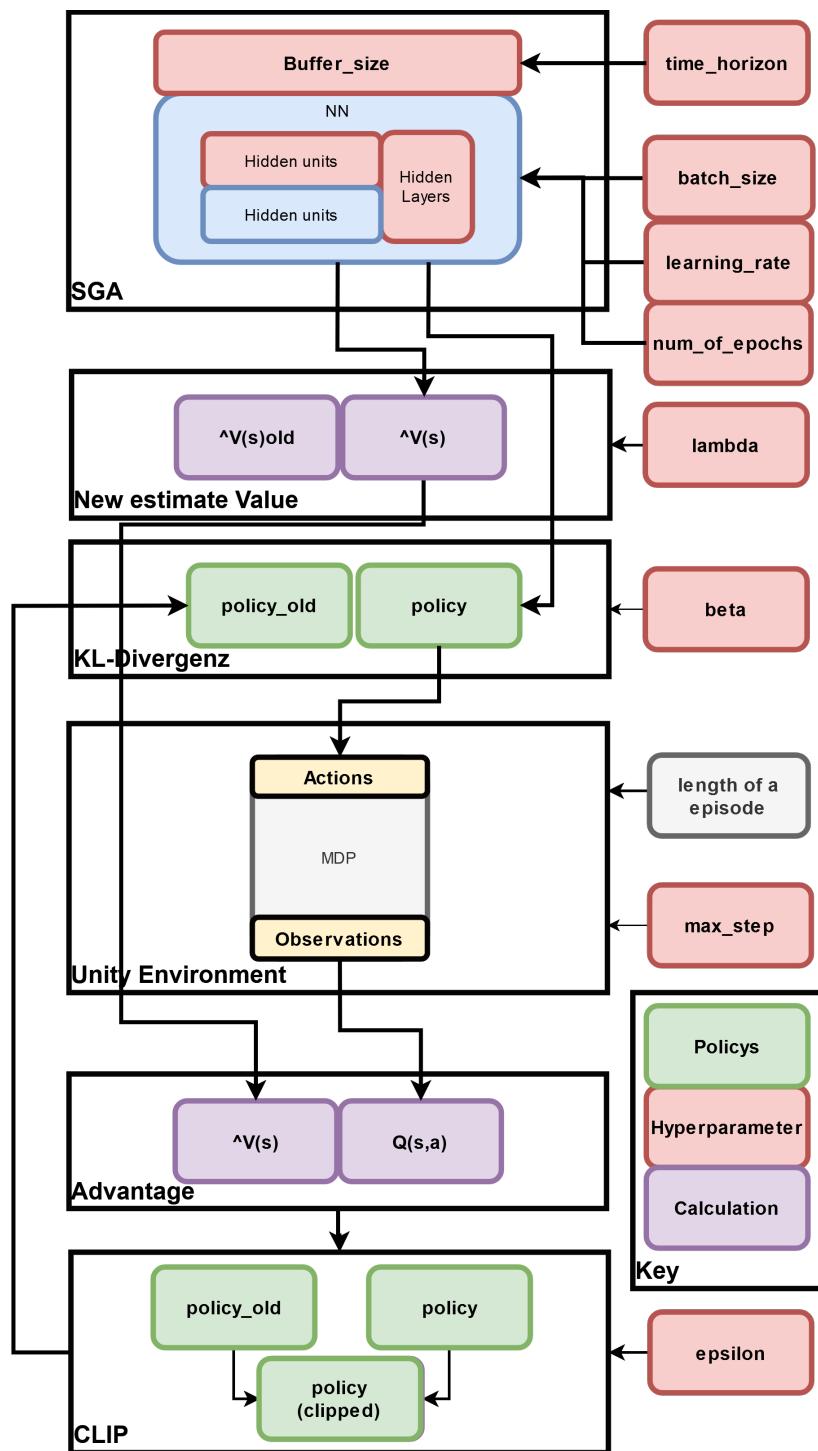


Abbildung 3.5.: Die Proximal Policy Optimisation mit KL-Divergenz: Die PPO erweitert, um die der KL-Divergenz (Eigene Darstellung)[46, S.2-3].

### 3.1.2. Soft Actor-Critic

Der SAC-Algorithmus ist ein “[...]"maximum entropy"reinforcement learning algorithm,[...]" [16, Absatz 3] d.h. die *Agents* sind dazu angehalten, das Problem zu lösen sich aber gleichzeitig so zufällig wie möglich zu verhalten [16, Absatz 2].

Dabei versucht der Algorithmus nicht nur den *Reward* aus einer Sammlung an *Policies* zu optimieren, sondern auch die Neuauswahl der *Policy* so zufällig wie möglich zu gestalten. [20, Soft Actor-Critic: Absatz 2-3] spricht hier von der Entropie-Regulierung. Der Anteil der Entropie wird über den Entropie-Koeffizienten gesteuert. Mit anderen Worten ist die Aufgabe des SAC eine beschränkende Maximierung der Entropie des erwarteten *Rewards*. [20, Soft Actor-Critic: Absatz 2-3]

In gewissen Fällen läuft die *Q*-Funktion  $Q(s, a)$  in *Actor-Critic*-Algorithmen Gefahr, den *Reward* von *Actions* zu überschätzen, wenn *Agents* in einer *Action a* ein lokales Maximum gefunden haben [36, S.1-2].

Aufgrund dessen, wird beim SAC der sogenannte *Double-Q Trick* eingesetzt, indem zwei *Q*-Funktionen erlernt werden [35, 1]. Die Idee von *Double Q-Learning* ist, die Auswahl von der Bewertung zu trennen, um die jeweils andere *Q*-Funktion zu aktualisieren [36, S.1-2].

Da im Rahmen dieser Thesis als Grundlage das *SoccerTwos-Environment* dient und dieses, wie den Hyperparametern in [13, Soccer] zu entnehmen ist, auf dem PPO basiert, wird dieser Algorithmus zum Training nicht verwendet.

## 3.2. Reward Signals

Wie in der Abbildung 3.1 gezeigt, besteht ein Teil der Konfiguration aus den Einstellungen der *Rewards*. Diese können, wie aus [33] hervorgeht, in zwei bzw. drei Teile gegliedert werden. Diese drei Teile werden in den folgenden Abschnitten erläutert.

### 3.2.1. Extrinsic Reward

Der *Extrinsic Reward* (z. Dt. äußere Belohnung) beschreibt in *ML-Agents*, wie der *Reward*, der in einem *Environment* gesammelt ist, verarbeitet wird [33, Extrinsic Reward Signal]. Dabei gibt es in *ML-Agents* zwei Einstellwerte:

- *strength* - Ist der Multiplikator des erhaltenen *Rewards*.[33, Strength]
- *gamma* - Gibt den discount-Faktor an.[33, Gamma]

### 3.2.2. Intrinsic Curiosity Module

Wie in [44, S.1] erwähnt wird, sollen in *Environments* mit schwer zugänglichem *Reward*, *Agents* dazu angeregt werden, neue Status zu entdecken. Dazu werden beim Intrinsic Curio-

sity Modul (ICM) zwei neuronale Netze (NN) trainiert. Ein *forward-NN* und ein *inverse-NN*. Das *forward-NN* schätzt den zukünftig zu erwartenden *Status*, das *inverse* schätzt die aktuelle *Action*. [44, S.1-3]

Die beiden Schätzungen werden mit den tatsächlichen Werten verglichen. Je weiter die Schätzungen von den tatsächlichen Werten abweichen, desto höher ist der *Curiosity Reward*. Der *Agent* wird also dafür belohnt, sich von den *Actions* und *Status* fern zu halten, die vorhergesagt worden sind. [39, Curiosity-driven exploration]

Dabei verfügt *ML-Agents* über folgende Hyperparameter:

- *strength*: Entspricht dem Ausmaß, wie sehr ein *Agent* für Neugier belohnt wird [33, Curiosity Reward Signal].
- *gamma*: Entspricht dem discount-Faktor in Bezug auf den *Curiosity-Reward* [33, Curiosity Reward Signal].

Die Grafik 3.6, die sinngemäß [44, S.3] entnommen ist, zeigt dabei den Aufbau des ICM mit entsprechenden Hyperparametern von *ML-Agents* [33, Curiosity Reward Signal].

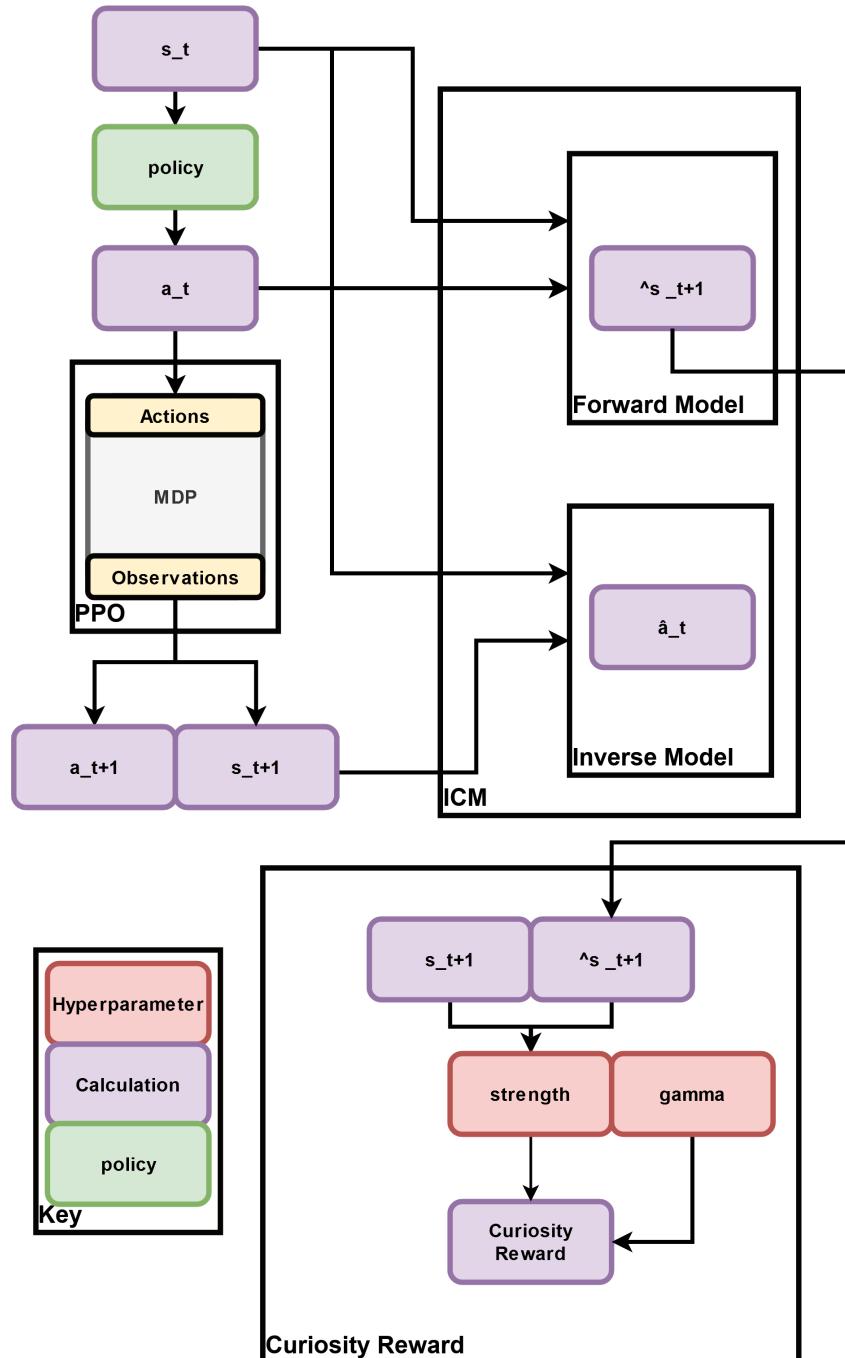


Abbildung 3.6.: Intrinsic Curiosity Module: Darstellung des ICM in Verbindung mit dem MDP (Eigene Darstellung) auf Basis von [44, Abb. 2 S.3] und [33, Curiosity Reward Signal].po

### 3.2.3. Generative Adversarial Imitation Learning

Das Generativ Adversarial Imitation Learning (GAIL) wird zwar über den *Reward*-Teil der Abbildung 3.1 eingestellt, da es aber inhaltlich zum Imitation Learning (IL)<sup>2</sup> eingeordnet werden kann, ist dieser im Abschnitt 3.3 erläutert.

## 3.3. Imitation Learning

Beim IL versuchen *Agents* das Verhalten von zuvor aufgezeichneten Daten zu erlernen [49, Abschnitt 3 BC]. Dabei gibt es im Rahmen von *ML-Agents* zwei Möglichkeiten, welche in den folgenden Abschnitten erklärt werden.

### Behaviour Cloning

Das Behaviour Cloning (BC) versucht, “[...]the Agent’s neural network to exactly mimic the actions shown in a set of demonstrations.”[49, Abschnitt 3 BC] Dabei wird zu einer voreingestellten Gewichtung (*strength*), zu einer bestimmten Dauer (*steps*), die *Policy* mit Daten aus der *Demonstrations*-Datei befüllt [49, Behavioral Cloning Using Demonstrations].

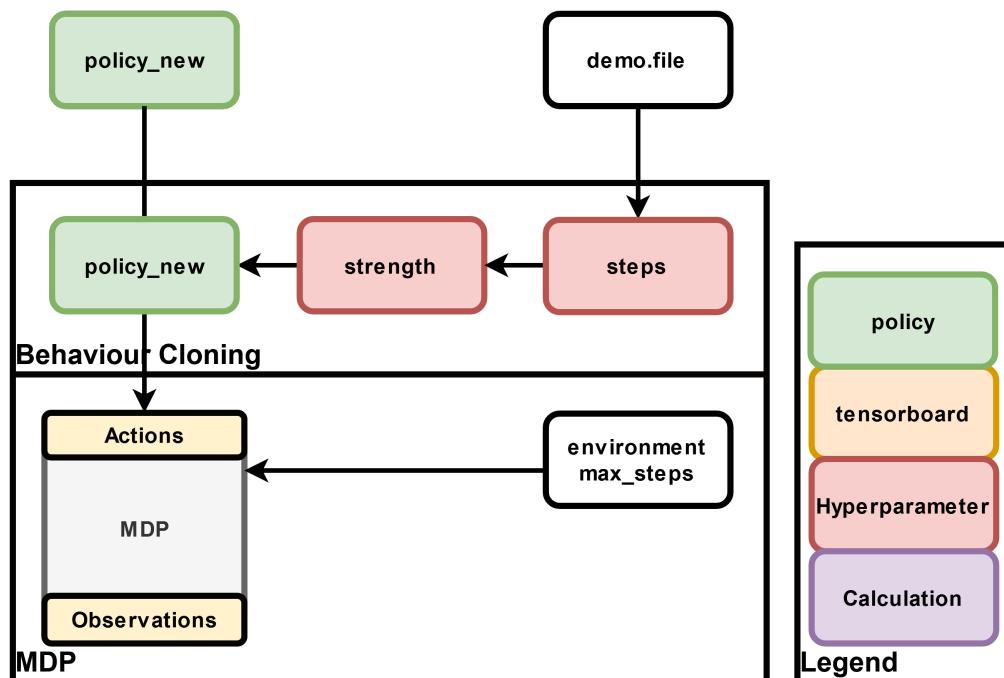


Abbildung 3.7.: Aufbau des Behaviour Cloning in Unity: Aufbau des Behaviour Cloning in Unity auf Grundlage der Dokumentation (Eigene Darstellung)[49, Behavioral Cloning Using Demonstrations].

<sup>2</sup>siehe Abschnitt 3.3

### Generative Adversarial Imitation Learning - GAIL

GAIL verbindet das BC mit dem Ansatz eines Generative Adversarial Network (GAN) [37, S.6]. Hier lernt ein *Discriminator D* zwischen *Actions* von *Demonstrations* und von *Agents* erzeugte *Actions* zu unterscheiden [37, S.6]. Dabei vergibt der *Discriminator* einen *Reward* dafür, wie nah das Verhalten eines *Agents* an dem einer *Demonstration* liegt [33, GAIL Reward Signal].

*ML-Agents* verfügt dabei unter anderem über drei einstellbare Hyperparameter:

- *strength* : Die Stärke, wie sehr ein *Agent* von einer *Demonstration* lernt [33, Strength].
- *gamma* : Entspricht dem discount-Faktor in Bezug auf den GAIL-Reward [33, Gamma].
- *demopath* : Der Pfad der *Demonstation*-Datei [33, Demo Path].

## 3.4. Curriculum Learning und Self-Play

Ist ein *Agent* vor komplexe Aufgaben gestellt, kann mit dem Curriculum Learning (CL) der *Agent* Stück für Stück an ein Problem herangeführt werden. Dabei wird ihm, sobald er eine Aufgabe gelöst hat, eine schwerere Aufgaben zugewiesen.[42, S.241]

### 3.4.1. Curriculum Learning

Mit dem CL ist es möglich, schwierige Aufgaben für *Agents* in Teilaufgaben zu unterteilen. Dabei werden stückweise *Rewards* für Teilaufgaben vergeben. Kommt ein *Agent* häufig genug über eine Schwelle an *Rewards*, wird seine Aufgabe schwerer.[42, S.239]

Dabei kommt CL im Rahmen dieser Thesis nicht zum Einsatz, da *SoccerTwos* es ebenfalls nicht verwendet.

### 3.4.2. Adversarial Self-Play

Das SP bietet über das *Environment* und dem *CL* hinaus die Möglichkeit, dem *Agent* automatisch eine immer schwerere Aufgabe zukommen zu lassen [23, Erster Absatz]. Dies geschieht, indem er gegen ein früheres Selbst spielt [26, S.1]. Diese Art der Aufgabenstellung kann zu unvorhergesehenem, emergentem Verhalten führen [28, Self-play and Soccer Environment].

Dabei verfügt die Implementierung von SP in *ML-Agents* über folgende Parameter:

- *save\_steps*: Wie viele Schritte eines Teams sollen aufgezeichnet werden [23, Save Steps].

- *swap\_steps*: Gibt die Anzahl der Schritte an bis die *Policy* geändert wird [23, Swap Steps].
- *play\_against\_latest\_model\_ratio*: Die Wahrscheinlichkeit, dass ein *Agent* gegen ein früheres selbst spielt [23, Play against current self ratio].
- *window*: Die Anzahl an vergangenen aufgezeichneten Gegnern, die verwendet werden können [23, Window].

Die Abbildung 3.8 zeigt, auf Basis der Dokumentation erstellt, den Aufbau des SP [23].

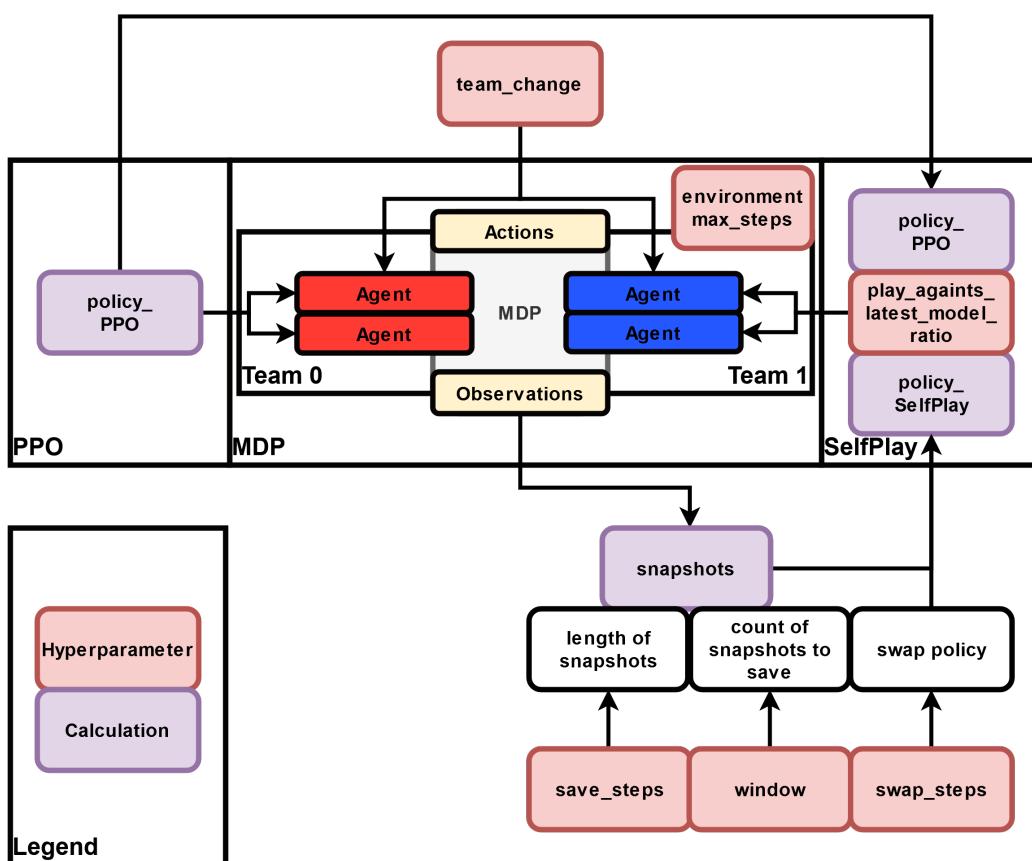


Abbildung 3.8.: Aufbau des SP in Unity: Aufbau des SP von *ML-Agents* auf Basis von (Eigene Darstellung)[23].

### 3.5. Recurrent Neural Networks

Ein Recurrent Neural Network (RNN) ist ein neuronales Netzwerk, das eine Speicherung von Eingangsdaten ermöglicht. Dies bedeutet, aufeinanderfolgende Reihen von Eingangsdaten zu vergleichen und dadurch Zusammenhänge in Daten erkennen[29, Abschnitt: Erster Absatz]. *ML-Agents* verwendet das *LSTM* [21, What are memories used for?]. Ein Long-Short-Term-Memory (LSTM) ist eine Sonderform des RNN, das die Möglichkeit bietet, Zusammenhänge in Daten über Lang- wie auch Kurzzeit skalierbar zu behandeln [34, S.1].



# Teil II.

## Forschungsdesign, Evaluation und Diskussion



# 4. Integration von ML-Agents

*NEON SHIFTER* ist ein von dem Entwicklerstudio *Couch in the Woods*<sup>1</sup> entwickelter Prototyp [50]. Dieser Prototyp soll zur späteren Entwicklung eines Spiels für die Nintendo Switch dienen. *NEON SHIFTER* ist ein kompetitives Spiel mit bis zu vier Teilnehmern, bei dem zwei Teams mit jeweils zwei Spielern gegeneinander antreten [50].

Auf dieser Basis wurde *ML-Agents* in den Prototyp integriert. Dabei wurden die Einstellungen und Skripte aus dem *Unity-Example-Environment SoccerTwos* in das *Unity-Editor-Environment* von *NEON SHIFTER* übernommen und entsprechend angepasst.

In den folgenden Abschnitten befindet sich ein Überblick zu *NEON SHIFTER* im Vergleich zum *ML-Agents-Example-Environment SoccerTwos*.

## 4.1. Anpassung eines Prototyps für ML-Agents

Im Zuge der Recherche konnte eine große Ähnlichkeit von *NEON SHIFTER* und dem *SoccerTwos-Example-Environment* festgestellt werden. Aus diesem Grund ist die Integration von *ML-Agents* in den Prototyp an den Aufbau von *SoccerTwos* angelehnt. Dieses Kapitel soll daher die Ähnlichkeit der beiden Szenen aufzeigen und die Integration darstellen.

Die Grafik 4.1 zeigt einen Screenshot aus der *Topdown*-Ansicht der *SoccerTwos* Szene im *Unity-Editor*. Dabei ist zu erkennen, dass es pro Team ein *Goal* und zwei *Agents* gibt. Die *Wall* bildet die Eingrenzung des Spielfelds.

Die Grafik 4.2 zeigt die *Topdown*-Ansicht von *NEON SHIFTER* mit ihren Inhalten. Dabei unterscheidet sich diese gegenüber *SoccerTwos* darin, dass die Tore versetzt sind. Ein weiterer Unterschied der beiden *Environments* besteht im Aufbau der Tore. Bei *NEON SHIFTER* werden diese zusätzlich von sogenannten *Dyson-Shields* geschützt [50]. Diese schützen das jeweilige *Goal*. Sie verschwinden bei Ballkontakt nacheinander, womit sie den Zugang zum *Goal* ermöglichen, bzw. erleichtern [50].

---

<sup>1</sup>[CouchintheWoods.de](http://CouchintheWoods.de)

#### 4. Integration von ML-Agents

---

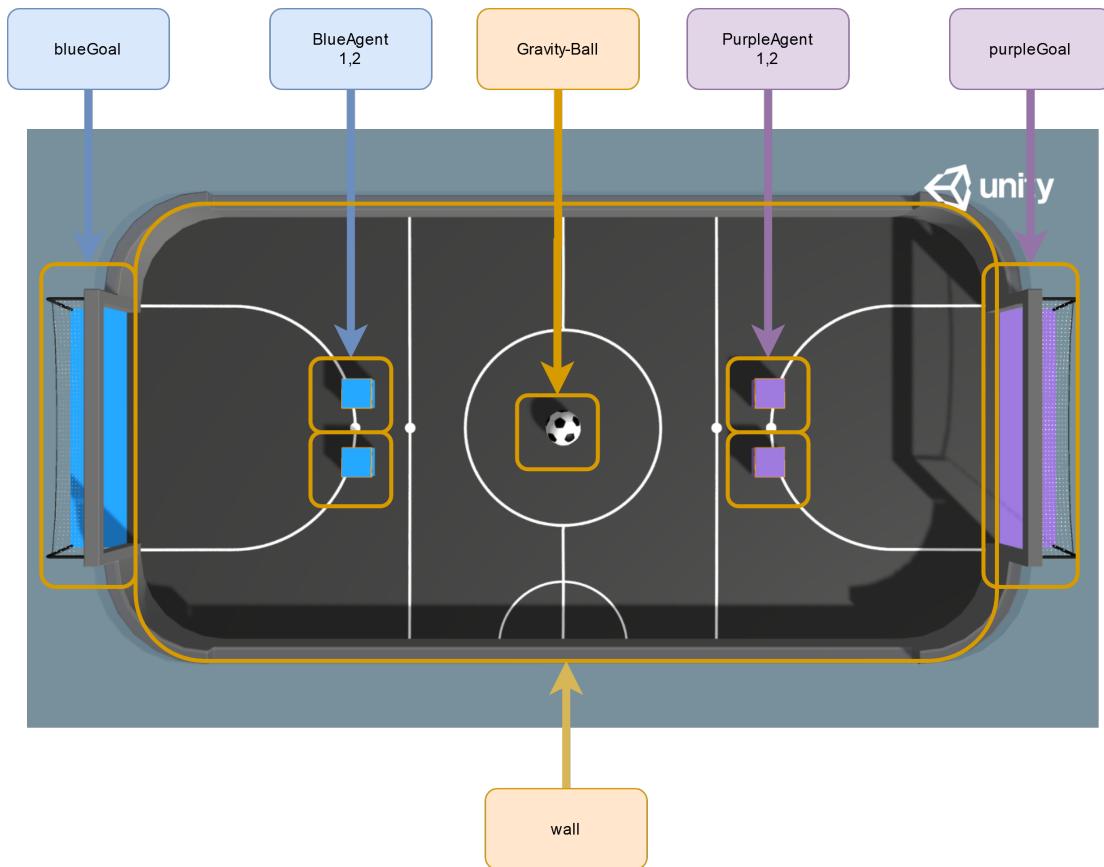


Abbildung 4.1.: Screenshot der SoccerTwos Szene in der Topdown Ansicht: Ein Screenshot des *SoccerTwos-Environment* aus der *Topdown*-Ansicht im *Unity-Editor* (Unity-Editor Screenshot)[2]. Mit zwei Toren, vier Agents, einem Ball und einer Wall als Abgrenzung.

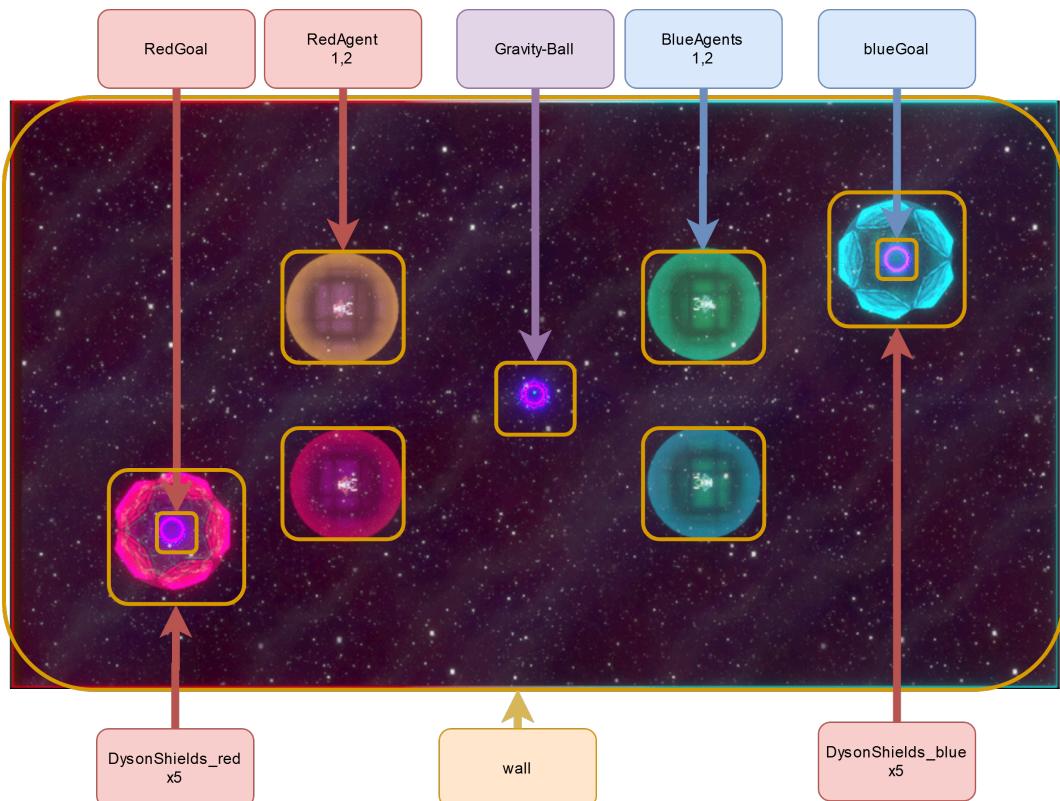


Abbildung 4.2.: Screenshot der NEON SHIFTER Szene in der Topdown Ansicht: Ein Screenshot von *NEON SHIFTER* aus der *Topdown*-Ansicht [50, Unity Editor]. Mit zwei Toren mit jeweils sechs *DysonShields*, vier Agents, ein Ball und eine rechteckige Abgrenzung, die *Wall*.

Weitere Unterschiede der Szenen sind dabei aus dem *Unity-Environment* übernommen:

- Wie aus dem Screenshot hervorgeht, ist die *Wall* im *SoccerTwos-Example* abgerundet.
- *SoccerTwos Agents* besitzen einen *Cube-Collider*, *NEON SHIFTER Agents* einen *Sphere-Collider*, um den Kontakt mit dem Ball festzustellen.
- Die Größe von Spieler und Spielfeld in Unity-Einheiten gemessen im *Unity-Editor*:
  - Spielfeld: *SoccerTwos*: 16x30, *NEON SHIFTER*: 264x460.
  - *Agent-Collider*: *SoccerTwos-Cube*: 1x1x1, *NEON SHIFTER Sphere-radius*: 5.
- *SoccerTwos* verfügt auch über ein Skript, dass die Spielrichtung der Mannschaften zufällig tauscht. Dies wurde in *NEON SHIFTER* ebenfalls implementiert.
- Der Ball in *NEON SHIFTER* setzt den Spieler bei Kontakt kurzzeitig außer Gefecht. [27, Zeile 7]
- Unterschiede in der Steuerung der *Agents* aus dem *Skript entnommen*:
  - *SoccerTwos-Keyboard*: W, S Vorwärts, Rückwärts | Q, E Rotation | S, A Seitwärts [30, Zeile 115-145].
  - *NEON SHIFTER-Controller*: up, down, left, right für die jeweilige Richtung.
- Zusätzliche Interaktionsmöglichkeiten bei der Steuerung von *NEON SHIFTER*:
  - Engine Off [50].
  - Shield [50].
  - Boost [50].
  - Gravity [50].

Die Grafik 4.3 zeigt die zusätzlichen Interaktionsmöglichkeiten als Screenshot aus dem Spiel.

## 4.2. Implementierung eines Prototyps für ML-Agents

Zur Integration von *ML-Agents* zählt auch die Übernahme der Skripte und deren Parameter. Um zu verdeutlichen, was bei der Integration in den Prototyp aus *SoccerTwos* übernommen wurde, sind im folgenden Abschnitt die Funktionen der Skripte erläutert.

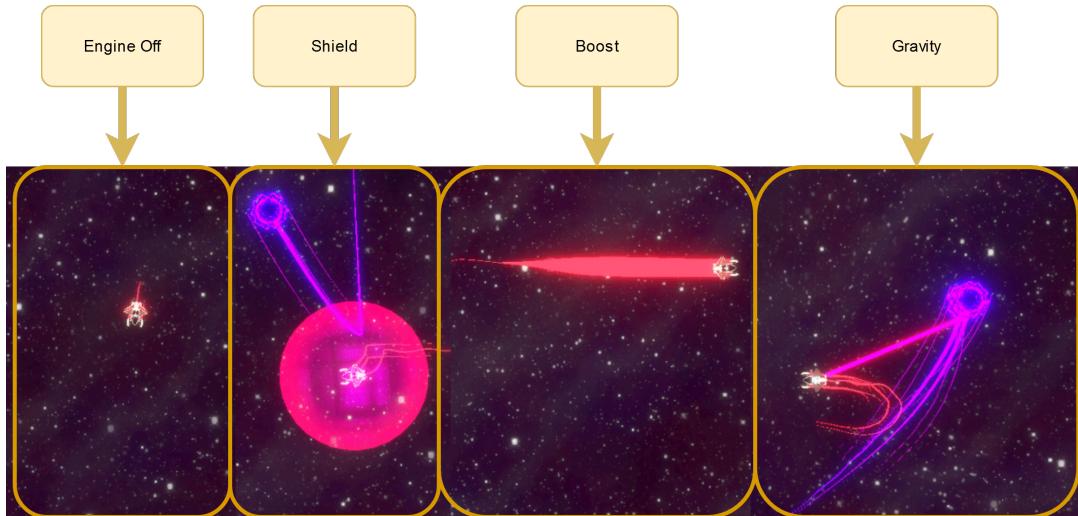


Abbildung 4.3.: Steuerungsachsen von NEON SHIFTER: Eine Übersicht von Screenshots der zusätzlichen Steuerungsachsen von *NEON SHIFTER* (UnityEditor Screenshot)[50].

#### 4.2.1. Agent Soccer

Im Folgenden sind die Anpassungen des *AgentSoccer*-Skripts aufgeführt.

##### Max-Step

*SoccerTwos* hat, wie aus dem Screenshot 7 im Anhang hervorgeht, hier im *Unity-Environment* eine *Episode-length* von 3000. Für *NEON SHIFTER* wurde eine Einstellung von 5000 gewählt. Dies begründet sich in der höheren Komplexität der Aufgabe und ist im Abschnitt 5.1.3 definiert. Dabei erschien 6000 als ein etwas zu hoher Wert, da *Example-Environments* wie *CrawlerStaticTarget* und *Pyramids* jeweils 5000 eingestellt haben.

##### Gestaltung der Rewards

In *SoccerTwos* sind die *Rewards* entsprechend der erfolgreichen Tor-Schüsse gestaltet. Das heißt, für jedes Tor erhält die entsprechende Mannschaft den maximalen *Reward* (1.0), die gegnerische Mannschaft erhält den Reward (-1.0). Anschließend wird die Episode beendet und die Szene zurückgesetzt. [5, Rewards]

Diese Gestaltung der *Rewards* dient dazu, einer Einschränkung durch die Wahl der *Rewards* vorzubeugen, sodass der *Agent* selbstständig eine Strategie zur Lösung des Problems finden kann. [25, S.4]

Diese Gestaltung der *Rewards* wurde für *NEON SHIFTER* entsprechend in das *Agent*-Skript übernommen [31, Zeile 51-70].

Dabei wurde aber auf das Zurücksetzen der *Agents* beim Zurücksetzen einer Episode verzichtet, da diese beim Treffer auf ein Tor in *NEON SHIFTER* nicht zurückgesetzt werden [50].

## Zeitstrafe

*SoccerTwos* verfügt über eine sogenannte *Existential penalty* bzw. *accumulated time penalty* (z. Dt. grundsätzliche bzw. Zeitstrafe) von  $\frac{1}{3000}$ . Diese erhält ein *Agent*, wie aus dem Skript hervorgeht, zu jedem *time step*. [7, Zeile 111]

Diese wurde für *NEON SHIFTER* entsprechend in das *Agent*-Skript übernommen.

### 4.2.2. Behaviour Parameter

Im Folgenden sind die Einstellwerte des *Behaviour Parameter*-Skript, aufgeführt und erläutert.

#### Agent-Observations

Durch die Verwendung des *Raycast Sensor Component* ist in den *Behaviour Parameters* keine *Observations* einzustellen.[41, 3. Absatz]

#### Vector Actions

Die von *NEON SHIFTER* vorgegebene Steuerung besteht aus fünf Steuerungsachsen. Die Tabelle:4.1 stellt dabei die Steuerungsachsen von *SoccerTwos* und *NEON SHIFTER* gegenüber.

Environment	Dimension	Branch	Count of Actions	Behaviour
<i>SoccerTwos</i>	rotation	1	3	aus,links,rechts
	forward	2	3	aus, vorwärts, rückwärts
	sideward	3	3	aus,links,rechts
<i>NEON SHIFTER</i>	rotation	1	3	aus,links,rechts
	Shield	2	2	Aus,An
	Boost	3	2	Aus,An
	Gravity	3	2	Aus,An
	Engine	4	2	Aus,An

Tabelle 4.1.: Actions von *SoccerTwos* und *NEON SHIFTER*: Gegenüberstellung der Anzahl der möglichen Steuerungsachsen eines *Agents* von *SoccerTwos* und *NEON SHIFTER* - entnommen aus den Skripten im *Unity-Editor*.

Wie aus der Tabelle 4.1 hervorgeht, sind im *SoccerTwos* Beispiel alle Steuerungsachsen diskret bzw. boolesche Werte. Da bei *NEON SHIFTER* vorgegeben ist, dass ein *Agent* sich ohne Interaktionseingabe nach vorne bewegt [50], ist die erste der drei Handlungssachsen der Rotation die Geradeausfahrt. Diese entspricht somit dem "nichts tun". Dies wurde deshalb so angepasst, damit der *Agent* nicht ständig zwischen links- und rechtsrotieren wechselt. Die Tabelle 4.2 zeigt dabei die direkte Einstellung der Parameter im *Unity-Editor*.

Environment	Branch	count of Actions	Summe
SoccerTwos	1	3	
	2	3	
	3	3	
Summe			= 9
NEON SHIFTER	1	3	
	2	2	
	3	2	
	4	2	
	5	2	
Summe			=11

Tabelle 4.2.: Actions von SoccerTwos und NEON SHIFTER im Unity-Editor mit Summe: Vergleich und Berechnung der Summe der *Actions* der Szenen *SoccerTwos* und *NEON SHIFTER*, entnommen aus dem *Unity-Editor*.

#### 4.2.3. Decision Requester

Der *Decision Requester* definiert, in welchem zeitlichen Abstand ein *Agent* eine *Policy* anfordert [5]. Dieser Intervall ist bei *SoccerTwos* und *NEON SHIFTER* auf 5 eingestellt.

#### 4.2.4. Raycast Sensor Component

Das *Raycast Sensor Component*-Skript kann in zwei Abschnitte gegliedert werden. Dazu befindet sich unter 5 auch ein Screenshot im Anhang. Zuerst die *Tags*, die vom Sensor im *Unity-Environment* erfasst werden, danach die Einstellungen der *Raycasts* selbst.

##### Tags

Die *Tags* wurden von dem *SoccerTwos-Environment* soweit wie möglich übernommen. *NEON SHIFTER* wurde um die *DysonShields* erweitert, da diese ein wichtiges Spielelement beinhalten [50].

SoccerTwos	NEON SHIFTER
Ball	Ball
Eigenes <i>Goal</i>	Eigenes <i>Goal</i>
<i>Goal</i> des Gegners	<i>Goal</i> des Gegners
Wall	Wall
<i>Agent</i> im Team	<i>Agent</i> im Team
<i>Agent</i> im gegnerische Team	<i>Agent</i> im gegnerische Team <i>DysonShields</i> des eigenen Teams <i>DysonShields</i> des gegnerischen Teams

Tabelle 4.3.: Die Raycast Sensoren von SoccerTwos und NEON SHIFTER: Gegenüberstellung der *Tags* von *SoccerTwos* und *NEON SHIFTER*. Die Tags der Sensoren sind zum besseren Verständnis ausgeschrieben. Diese Daten sind den entsprechenden *Unity-Szenen* entnommen.

Parameter	SoccerTwos	NEON SHIFTER
Sensorname	Forward	NeonShifterSensor
Rays per Direction	5	18
Max Ray Degrees	60	180
Sphere Cast Radius	0,5	10
Ray length	20	20
Ray layer mask	Mixed	RaySensors3D
Observation Stack	3	3
Sensorname	Reverse	
Rays per Direction	1	
Max Ray Degrees	45	
Sphere Cast Radius	0,5	
Ray length	20	
Ray layer mask	Mixed	
Observation Stack	3	

Tabelle 4.4.: Tags von SoccerTwos und NEON SHIFTER: Eine Gegenüberstellung der *Tags* des *Raycast Sensor Component* für die *Environments NEON SHIFTER* und *SoccerTwo*. Diese Daten sind den entsprechenden *Unity-Szenen* entnommen (siehe dazu den Screenshot der Raycast Parameter aus dem *SoccerTwos-Environment* im *Unity-Editor 5*).

## Raycast

Um die *Observations* des *NEON SHIFTER-Environment* so nah wie möglich an der Erfahrung eines menschlichen Spielers anzuknüpfen, wurde die Zahl der *Raycasts* so erhöht und eingestellt, dass der Agent möglichst jederzeit, jedes Element auf dem Spielfeld erkennen kann. Die Anzahl der Sensoren wurde auf eins reduziert, da der *Reverse* Sensor in *SoccerTwos* der *Observation* der Rückseite dient und dies bei *NEON SHIFTER* mit einem Sensor umgesetzt wurde.



# 5. Anwendung von ML-Agents

Dieses Kapitel bildet den Hauptteil der Thesis. Darin lassen sich die Konfiguration der Experimente und die Auswertungsmetrik finden.

## 5.1. Konfiguration von ML-Agents in einem Prototyp

Für das Durchführen eines Trainings mit *ML-Agents* ist in den folgenden Abschnitten zunächst der Aufbau des verwendeten Computers, die Konfiguration des *Python-Environments* und die anschließende Festlegung der Hyperparameter aufgeführt.

### 5.1.1. Konfiguration des Computers

Der Computer, auf dem das Training ausgeführt wurde, ist wie in der folgenden Tabelle 5.1 aufgebaut.

Bauteil	Daten
RAM	32GB(2x16384MB)DDR4 – 3200MHz
CPU	AMDRyzen71700X8x3.40GHz
GPU	8GBKFA2GeForceGTX1070EX
MB	MSIB350GAMING

Tabelle 5.1.: Aufbau des Computer

### 5.1.2. Konfiguration des Python-Environment

*Unity-Technologies* veröffentlichte 2019 Ergebnisse bzgl. der Trainingsgeschwindigkeit mit *ML-Agents*. Dabei wurde in einem Spiel die zum Erreichen eines bestimmte Levels benötigte Zeit gemessen. Dabei ist aufgeführt, dass ein paralleles Training mehrerer *Unity-Environments* zu einer Beschleunigung des Trainings führt. [32, Performance results]

*Unity-Technologys* weist darauf hin, dass “[...]diminishing returns as we scale further.” [32, Performance results], daher wurde die Anzahl der verwendeten *Unity-Environments* wie empfohlen auf 16 eingestellt.

## 5. Anwendung von ML-Agents

---

Dabei verfügt, wie aus der *Unity*-Szene hervorgeht, *SoccerTwos-Example* in einem *Environment* über acht Instanzen, also acht Spielfeldern wie auch aus dem Screenshot 6 im Anhang hervorgeht. Daraus errechnet sich für das Training mit *NEON SHIFTER* folgende Konfiguration:

$$(8 \text{ Instanzen pro Environment}) * (16 \text{ Environments}) = 128 \text{ Trainingsinstanzen} \quad (5.1)$$

Des Weiteren wurde der *no-graphics*-Befehl für die Python-Umgebung angewendet, um die Performance zu erhöhen [14, Command Line Training Options].

Zusätzlich gab es in der Dokumentation den Hinweis, dass bei einer Erhöhung der Instanzen auch die *buffer\_size* angepasst werden kann. “A common practice is to multiply *buffer\_size* by *num-envs*.” [8, Buffer Size]

Dabei war jedoch unklar, auf welche Konfiguration an *Environments* sich diese Erhöhung beim *SoccerTwos-Environment* bezieht und ob diese erhöht werden muss. Um dies festzulegen, wurde Experiment 1 ausgeführt. Die dadurch erhaltenen Erkenntnisse wurden anschließend in Experiment 2 verwendet.

### 5.1.3. Komplexität des Prototyps

Zur Festlegung der Hyperparameter für das Training des Prototyps wurde die Anzahl der *Observations* und *Actions* aus dem *SoccerTwos-Environment* und der *NEON SHIFTER-Environment* verglichen und das Verhältnis berechnet.

Um die Anzahl der *Observations* zu berechnen, gibt *ML-Agents* folgende Formel an [5]:

$$(\text{Observation Stacks}) * (1 + 2 * \text{Rays Per Direction}) * (\text{Num Detectable Tags} + 2) \quad (5.2)$$

In Tabelle 5.2 folgt eine Gegenüberstellung der *Observations* der *SoccerTwos*-Szene und der *NEON SHIFTER*-Szene und deren errechnete Gesamtzahl.

Environment	Stacks	Rays per direction	Tags	Summe
SoccerTwos	3	5	6	264
	3	1	6	72
Summe				336
NEON SHIFTER	3	18	8	1110
Summe				1110

Tabelle 5.2.: Observations von SoccerTwos und NEON SHIFTER: Vergleich der *Observations* von *SoccerTwos* und *NEON SHIFTER*. Errechnet unter Anwendung der Formel 5.2 mit den eingestellten Daten aus den *Unity-Environments* (siehe dazu ).

Dabei zeigt sich, dass die *Observations* für *NEON SHIFTER* um ca. das Dreifache größer

ist. Zu der Anzahl der *Observations* wurde auch die Anzahl der *Actions* vergrößert. Deshalb wurden diese ebenfalls in die Berechnung einbezogen.

Environment	Anzahl der <i>Actions</i>	Anzahl <i>Observations</i>	Mittelwert
SoccerTwos	9	336	
NEON SHIFTER	11	1110	
Verhältnis	1,22	3,3	2,26

Tabelle 5.3.: Komplexität von SoccerTwos und NEON SHIFTER: Eine Gegenüberstellung von Verhältnis, errechnetem Mittelwert und Durchschnitt der beiden Verhältnisse für die *Actions* und *Observations* von *SoccerTwos* und *NEON SHIFTER*. Entnommen aus den *Unity-Environments*.

Wie die Tabelle 5.3 zeigt, hat sich das Verhältnis der *Actions* um den Faktor 1,22 geändert und das Verhältnis der *Observations* um den Faktor 3,3. Anschließend ist daraus ein Mittelwert berechnet worden. Dieser Mittelwert dient als Grundlage für die Konfiguration der Hyperparameter in den Experimenten.

## 5.2. Experiment 1: Training des SoccerTwos-Environment

Zur Erhebung von Vergleichsdaten wurde zunächst das *SoccerTwos-Example* trainiert. Aus der Dokumentation geht, wie zuvor beschrieben, nicht eindeutig hervor, ob beim Training mit *SoccerTwos* unter Verwendung von mehreren Trainings-*Environments*, die *buffer\_size* erhöht werden muss. Deshalb wurden in Experiment 1 zwei Trainings ausgeführt. Einmal mit der voreingestellten *buffer\_size* und einmal mit einer vergrößerten *buffer\_size*.

Dabei wurde die Anzahl der *Environments* durch das *Python-Environment* Absatz 2 von 2.5 auf wie in 16 eingestellt.

Dazu wurde die *buffer\_size* um den Faktor 10 erhöht. Auch wenn dies von der in Absatz 1 2.5 zitierten Empfehlung abweicht, da eine Erhöhung über den Faktor 10 beim Prototyp die empfohlene *buffer\_size* überschritten hätte [15].

### 5.2.1. Wahl der Hyperparameter

Die Tabelle 5.4 zeigt dabei die gewählten Hyperparameter für die ersten beiden Trainings.

### 5.2.2. Training zur Reproduzierbarkeit

Nachdem in der Auswertung von Experiment 1 in Abschnitt 7.1 gezeigt werden konnte, dass die Konfiguration von Training 1.0 mit guten Ergebnissen abgeschlossen werden konnte, wurde das Training erneut ausgeführt, um eine Reproduzierbarkeit zu belegen. Die Hyperparameter sind der Tabelle 5.4 mit der Bezeichnung 1.0 zu entnehmen. Dabei wird erwartet, dass die Trainingsergebnisse wiederholbar sind.

Bezeichnung	SoccerTwos 1.0	SoccerTwos Angepasst 1.1
run-ID	<i>SoccerTwo_0.1_random_Soccer</i>	<i>SoccerTwo_0.2_Soccer</i>
Hyperparameter		
trainer	ppo	ppo
lambd	0.95	0.95
buffer_size	20480	<b>204800</b>
batch_size	2048	2048
num_epochs	3	3
learning_rate	3.0e-4	3.0e-4
learning_rate_schedule	constant	constant
time_horizon	1000	1000
max_steps	5.0e7	5.0e7
beta	5.0e-3	5.0e-3
epsilon	0.2	0.2
normalize	false	false
num_layers	2	2
hidden_units	512	512
<b>reward signals</b>		
<b>extrinsic</b>		
strength	1.0	1.0
gamma	0.99	0.99
<b>self_play</b>		
save_steps	50000	50000
team_change	50000	50000
swap_steps	0.5	0.5
play_against_la-		
test_model_ratio		
window	10	10

Tabelle 5.4.: Hyperparameter für SoccerTwos: SoccerTwos Hyperparameter für Training 1.0 und 1.1. Die Hyperparameter von 1.0 wurden aus der Konfigurationsdatei für *SoccerTwos* übernommen [13, Soccer]. Die Hyperparameter des Trainings 1.1 zeigt dabei die angepassten Werte

### 5.3. Experiment 2: Training des Prototyp-Environment

Dieses Experiment soll zunächst zeigen, wie die Hyperparameter für den Prototyp konfiguriert werden müssen. In Abschnitt 5.1.3 wurde dafür die Komplexität des Prototyps errechnet und mit dem *SoccerTwos-Example* verglichen. Die Tabelle 5.5 stellt diese gegenüber. Die ersten beiden Trainings dieses Experiments wurden mit einer verringerten Anzahl an (*max\_steps*) ausgeführt bzw. abgebrochen, da schon früh beim Training ersichtlich war, dass zwei Trainings viel Zeit gebraucht hätten.

### 5.3.1. Wahl der Hyperparameter

Die Hyperparameter wurden der Konfigurationsdatei für *SoccerTwos* entnommen und unter Anwendung der in der Dokumentation [15] empfohlenen Einstellwerte für das Training mit PPO auf das Training mit *NEON SHIFTER* angepasst. Dazu wurde auch der errechnete Komplexitätsfaktor aus Abschnitt 5.3 genutzt.

Des Weiteren wurde auch der Einsatz der multiplen *Environments* und der Ergebnisse aus dem ersten Experiment in Abschnitt 7.1 beibehalten und verwendet. Dabei wurden für *NEON SHIFTER* die folgenden Hyperparameter geändert.

- *buffer\_size*: Wurde verdoppelt (Auf Grundlage der Komplexität aus Tabelle 5.3).
- *batch\_size*: Wurde verdoppelt (Auf Grundlage der Komplexität aus Tabelle 5.3).
- *time\_horizon*: Wurde verdoppelt (Auf Grundlage der Komplexität aus Tabelle 5.3).
- Experiment 2.0
  - *num\_layers*: Wurde belassen (Empfehlung Dokumentation) [15, Number of Layers].
  - *hidden\_units*: Wurde belassen (Empfehlung Dokumentation) [15, Hidden Units].
- Experiment 2.1
  - *num\_layers*: Wurde verdoppelt (Auf Grundlage der Komplexität aus Tabelle 5.3).
  - *hidden\_units*: Wurde verdoppelt (Auf Grundlage der Komplexität aus Tabelle 5.3).

Als Resultat aus den Ergebnissen in Abschnitt 7.2 dieses Vortrainings (2.0, 2.1) wurde das Netzwerk für das vollständige Training (Experiment 2.2) wie in Experiment 2.0 konfiguriert und mit fünfzig Millionen *max\_steps* ausgeführt. Die Tabelle 5.5 zeigt die Gesamtübersicht über die eingestellten Hyperparameter für die *Environments SoccerTwos* (Für das Experiment 1.0) und *NEON SHIFTER* (Für das Experiment 2.0, 2.1, 2.2).

## 5.4. Experiment 3: Training des Prototyp-Environment mit Erweiterung

Das Experiment 3 soll der Leistungssteigerung der Trainingsergebnisse dienen. In der *ML-Agents*-Dokumentation sind Trainingsergebnisse in Bezug auf das Training mit IL, aufgeführt [49, Abb. 1], diese werden in Abbildung 5.1 dargestellt. Diese Ergebnisse beziehen sich auf das *Pyramids-Example*<sup>1</sup>. Dabei wird dargestellt, dass IL unter Anwendung einer

---

<sup>1</sup>In Abschnitt *Pyramids-Environment*

## 5. Anwendung von ML-Agents

---

Bezeichnung	SoccerTwos 1.0	NEON SHIFTER 2.0, 2.2	NEON SHIFTER angepasst. 2.1
run-ID	SoccerTwo_0.1_random_Soccer	NeonShifter_PPO_SelfPlay_small_Buffer_1.0_NeonShifter	NeonShifter_PPO_SelfPlay_small_Buffer_1.1_NeonShifter
Hyperparameter			
trainer	ppo	ppo	ppo
lambd	0.95	0.95	0.95
buffersize	20480	<b>40960</b>	40960
batch_size	2048	<b>4096</b>	4096
num_epochs	3	3	3
learning_rate	3.0e-4	3.0e-4	3.0e-4
learning_rate_schedule	constant	constant	constant
time_horizon	1000	<b>2000</b>	2000
max_steps	5.0e7	5.0e7	5.0e7
beta	5.0e-3	5.0e-3	5.0e-3
epsilon	0.2	0.2	0.2
normalize	false	false	false
num_layers	2	2	<b>3</b>
hidden_units	512	512	<b>1024</b>
<b>reward signals</b>			
<b>extrinsic</b>			
strength	1.0	1.0	1.0
gamma	0.99	0.99	0.99
<b>self_play</b>			
save_steps	50000	50000	50000
team_change			
swap_steps	50000	50000	50000
play_against	0.5	0.5	0.5
<sub>-la-</sub>			
test_model_ratio			
window	10	10	10

Tabelle 5.5.: Hyperparameter für Proximal Policy Optimisation: Eine Gegenüberstellung der Hyperparameter der Experimente 1.0 und 1.1. 1.1 ist hier nicht aufgeführt, da es nicht weiter Verwendet wurde. Diese Daten der Hyperparameter für *SoccerTwos* sind ebenfalls der Konfigurationsdatei entnommen [13, Soccer].

Kombination aus BC, GAIL, CL, und RL zu einer Beschleunigung des Trainings führt. Daraus ist eine Verbesserung der Trainingsergebnisse für das Experiment 3 zu erwarten.

## Reinforcement Learning using Demonstrations on Pyramids

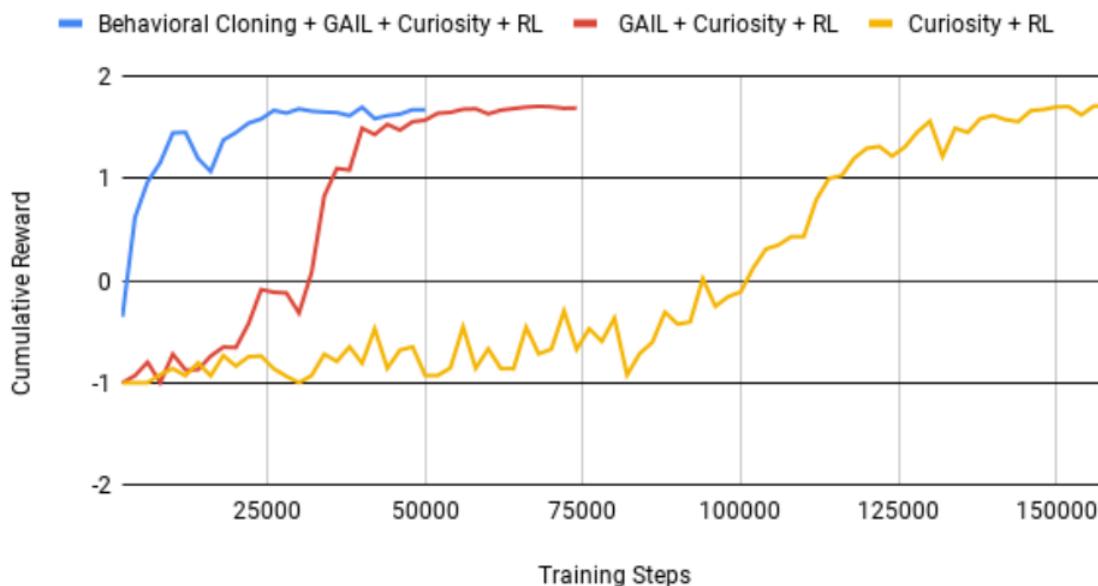


Abbildung 5.1.: Reinforcement Learning in Kombination mit Behavior Cloning, Generative Adversarial Imitation Learning und Curriculum Learning: Die Trainingsergebnisse für das *Pyramids-Example* unter Anwendung von *Demonstrations* aus der Dokumentation entnommen [49, Abb.1].

### 5.4.1. Wahl der Hyperparameter

Um das Experiment 3 durchzuführen, wurde die Konfiguration der Hyperparameter aus Experiment 2.2 mit den Hyperparametern aus dem *Pyramids-Environment* [3] kombiniert. Die Hyperparameter wurden dabei der Konfigurationsdatei entnommen. Die folgende Tabelle zeigt die Übersicht der Hyperparameter für Experiment 3.0.



Bezeichnung	<i>SoccerTwos</i>	<i>Pyramids</i>	<i>NEON SHIFTER</i>
run-ID	1.0	-	2.2, 3.0
Hyperparameter			
trainer	ppo		ppo
lambd	0.95		0.95
buffersize	20480		40960
batch_size	2048		4096
num_epochs	3		3
learning_rate	3.0e-4		3.0e-4
learning_rate_schedule	constant		constant
time_horizon	1000		2000
max_steps	5.0e7		5.0e7
beta	5.0e-3		5.0e-3
epsilon	0.2		0.2
normalize	false		false
num_layers	2		2
hidden_units	512		512
<b>reward signals</b>			
<b>extrinsic</b>			
strength	1.0		1.0
gamma	0.99		0.99
<b>curiosity</b>			
strength		0.02	0.02
gamma		0.99	0.99
encoding_size		256	256
learning_rate			
<b>gail</b>			
demo_path			
strength		0.01	0.01
gamma		0.99	0.99
encoding_size		128	128
learning_rate			
use_actions			
use_vail			
<b>behaviour Cloning</b>			
demo_path			
strength		0.5	0.5
steps		150000	150000
<b>self_play</b>			
save_steps	50000		50000
swap_steps	50000		50000
play_against-_latest_model_ratio	0.5		0.5
window	10		10

Tabelle 5.6.: Hyperparameter für Proximal Policy Optimisation mit Generative Adversarial Imitation Learning und Behaviour Cloning: Gegenüberstellung der Hyperparameter aus dem *SoccerTwos-Example* [13, Soccer], *Pyramids-Example* [12, Pyramids] und deren Kombination für die Konfiguration von *NEON SHIFTER*. Teilsweise übernommen aus dem Training 2.2



# 6. Auswertung von ML-Agents

Um Trainingsprozesse auszuwerten, wurde von *Unity-Technologys* für *ML-Agents Tensorboard* integriert und vorkonfiguriert [17]. In diesem Kapitel wird auf Basis der Dokumentation [17] dargestellt, wie die Trainingsprozesse evaluiert werden können. Der Screenshot in Abbildung 6.1 zeigt eine Übersicht von *Tensorboard*, um dessen Anwendung zu verdeutlichen. Der Tab *Scalars* zeigt die Trainingsergebnisse. Unter *Text* lassen sich die Hyperparameter des Trainings anzeigen. Darüber hinaus können durch *Tensorboard* Trainingsdaten wie folgt dargestellt werden:

- Darstellung nach ausgeführten Schritten im *Environment Steps*.
- Darstellung nach benötigter Zeit relativ zueinander *Relativ*.
- Darstellung nach gebrauchter Zeit nacheinander *Wall*.

Dabei lässt sich jede Anzeige zusätzlich in den Vollbildmodus schalten, als logarithmische Darstellung anzeigen und in das angezeigte Fenster einpassen.

## 6. Auswertung von ML-Agents



Abbildung 6.1.: Screenshot von Tensorboard: Screenshot der *Tensorboard*-Oberfläche mit zusätzlichen Hinweisen. *Tensorboard* kann wie dargestellt durch Aufrufen in der Kommandozeile geöffnet werden [17, 3. Absatz].

## 6.1. Auswertung der Proximal Policy Optimisation

Die Abbildung 6.2 zeigt den Aufbau des PPO-Trainings, wie in 3.5 dargestellt, mit den zugehörigen Hyperparametern und den entsprechenden Ausgaben in *Tensorboard*.

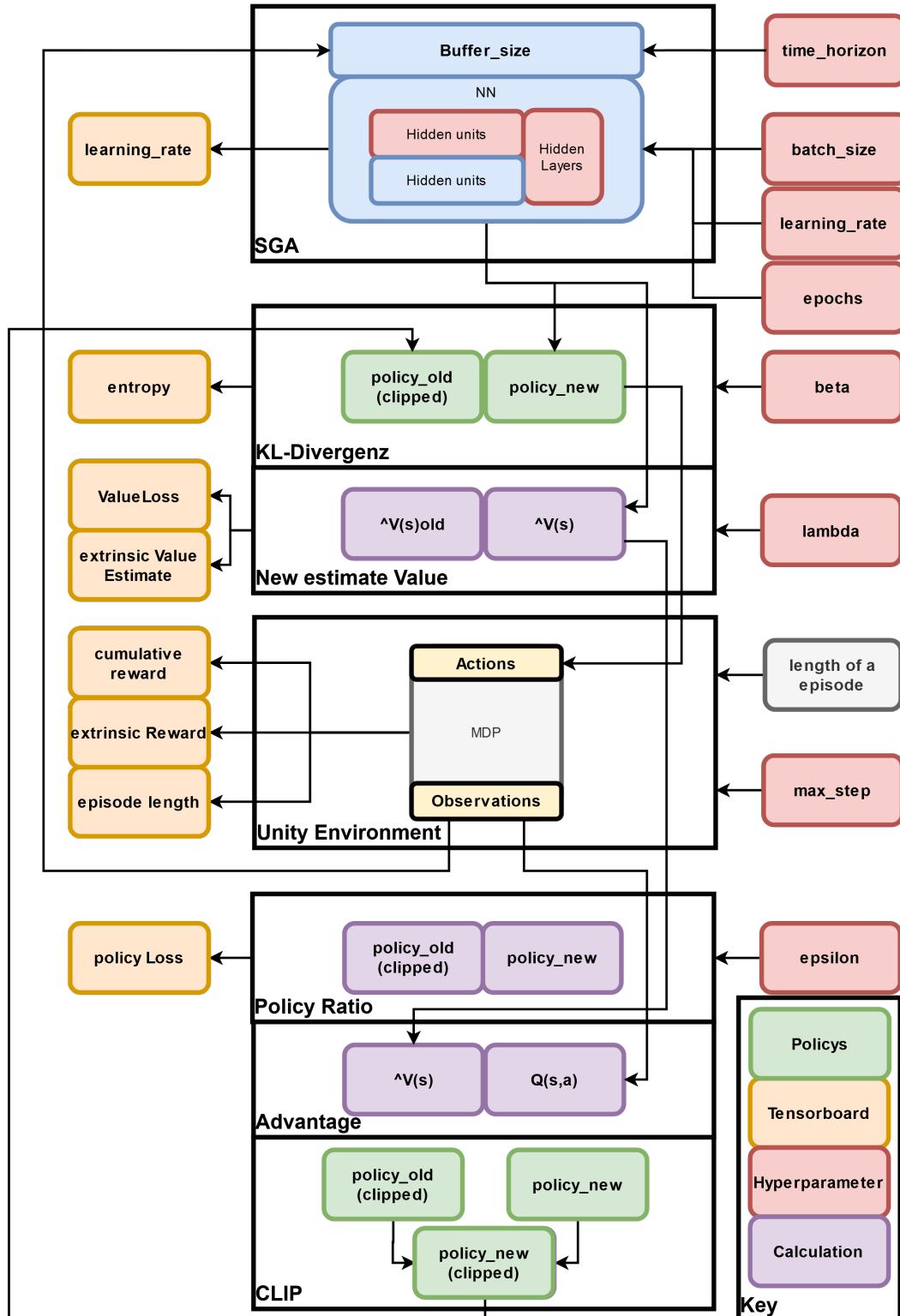


Abbildung 6.2.: Die Proximal Policy Optimisation mit Tensorboard: Proximal Policy Optimisation erweitert um die Ausgaben in *Tensorboard* (Eigene Darstellung)[46, S.2-3]

## 6. Auswertung von ML-Agents

---

Dabei zeigt die Tabelle 6.1 auf Basis der Dokumentation [15, Training Statistics], welcher Wert sich wie zu verhalten hat.

Name	Art	Verhalten
<i>Cumulative Reward</i>	Der kumulative <i>Reward</i> , den ein <i>Environment</i> über alle <i>Agents</i> insgesamt generiert	Sollte steigen - kann aber Millionen von Steps dauern
<i>Episode Length</i>	Die durchschnittliche Länge einer Episode für alle <i>Agents</i>	Sollte sinken
<i>Policy Loss</i>	Ausmaß der Strategieänderung	Sollte kleiner werdende Schwankungen zeigen
<i>Value Loss</i>	Die Loss der <i>Value</i> -Funktion	Sollte ansteigen - aber mit Stabilisierung des <i>Rewards</i> sich ebenfalls stabilisieren
<i>Entropy</i>	Wie zufällig eine Entscheidung ist	Sollte sinken
<i>Extrinsic Reward</i>	Der kumulative <i>Reward</i> , den ein <i>Agent</i> generiert	Sollte steigen - kann aber Millionen Schritten dauern
<i>Extrinsic Value Estimate</i>	Der geschätzte <i>Reward</i> für alle Status, die ein <i>Agent</i> vollzogen hat	Sollte ansteigen
<i>learning_rate</i>	Die Schrittgröße des <i>Stochastic Gradient Ascent</i>	Ist durch die Einstellung konstant

Tabelle 6.1.: Tensorboard Auswertungstabelle des Proximal Policy Optimisation: Aufstellung der Verhaltensweisen der Messwerte in *Tensorboard* bei einem Training mit PPO, auf Basis der von *ML-Agents* gegebenen Dokumentation [15, Training Statistics]

## 6.2. Auswertung des Self-Play

Das SP lässt sich in *Tensorboard* über die *ELO* messen [23, ELO]. Die *ELO* ist ein relativer Messwert zur Einstufung des Könnens eines Spielers im Vergleich zu anderen in einem Null-Summen-Spiel [28, Implementation and usage details]. Dieser Wert sollte beim Training ansteigen, um zu beurteilen, ob ein Spieler besser gegen ein vergangenes Selbst agiert [23, ELO].

Dabei ist beim Training mit SP zu beachten, dass der *Cumulative Reward* den *Reward* aller *Agents* der gesamten Szene misst [17, Environment/Cumulative Reward]. Dies gilt im Falle des SP mit *SoccerTwos* für das Team von *Agentes* das trainiert, wie auch das Team, das als vergangenes Selbst des Teams agiert. Bei einem optimal ablaufenden Null-Summen-Spiel sollte sich dieser bei Null einpendeln [19].

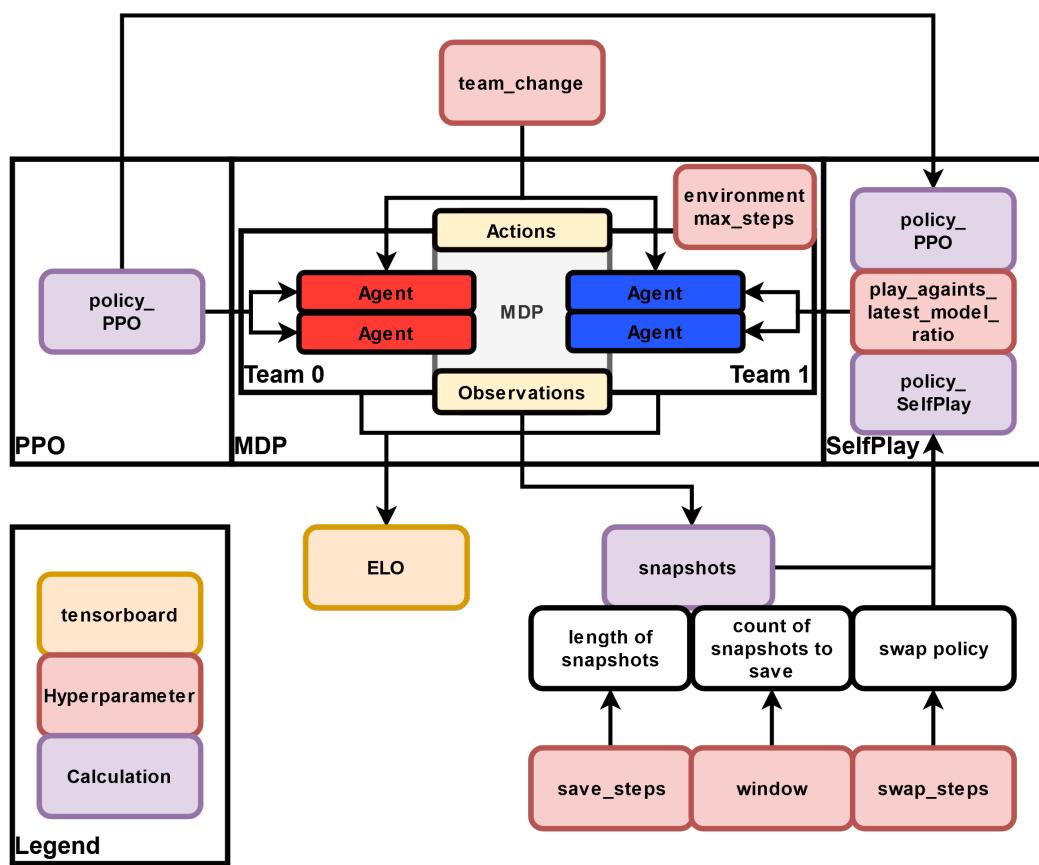


Abbildung 6.3.: Self-Play mit Tensorboard: Die Darstellung des SP und die Ausgaben in *Tensorboard* (Eigene Darstellung)[23]

### 6.3. Auswertung von Curiosity Learning

Die auf der Basis der Veröffentlichung von [44, S.3] erstellten Grafik 3.6, wird in Abbildung 6.4 um die Ausgaben aus *Tensorboard* erweitert.

*ML-Agents* liefert zum jetzigen Stand zwar eine Definition der Ausgabewerte, aber keine Empfehlung für das Evaluieren von CL [17]. Daher ist kein Verhalten eingetragen.

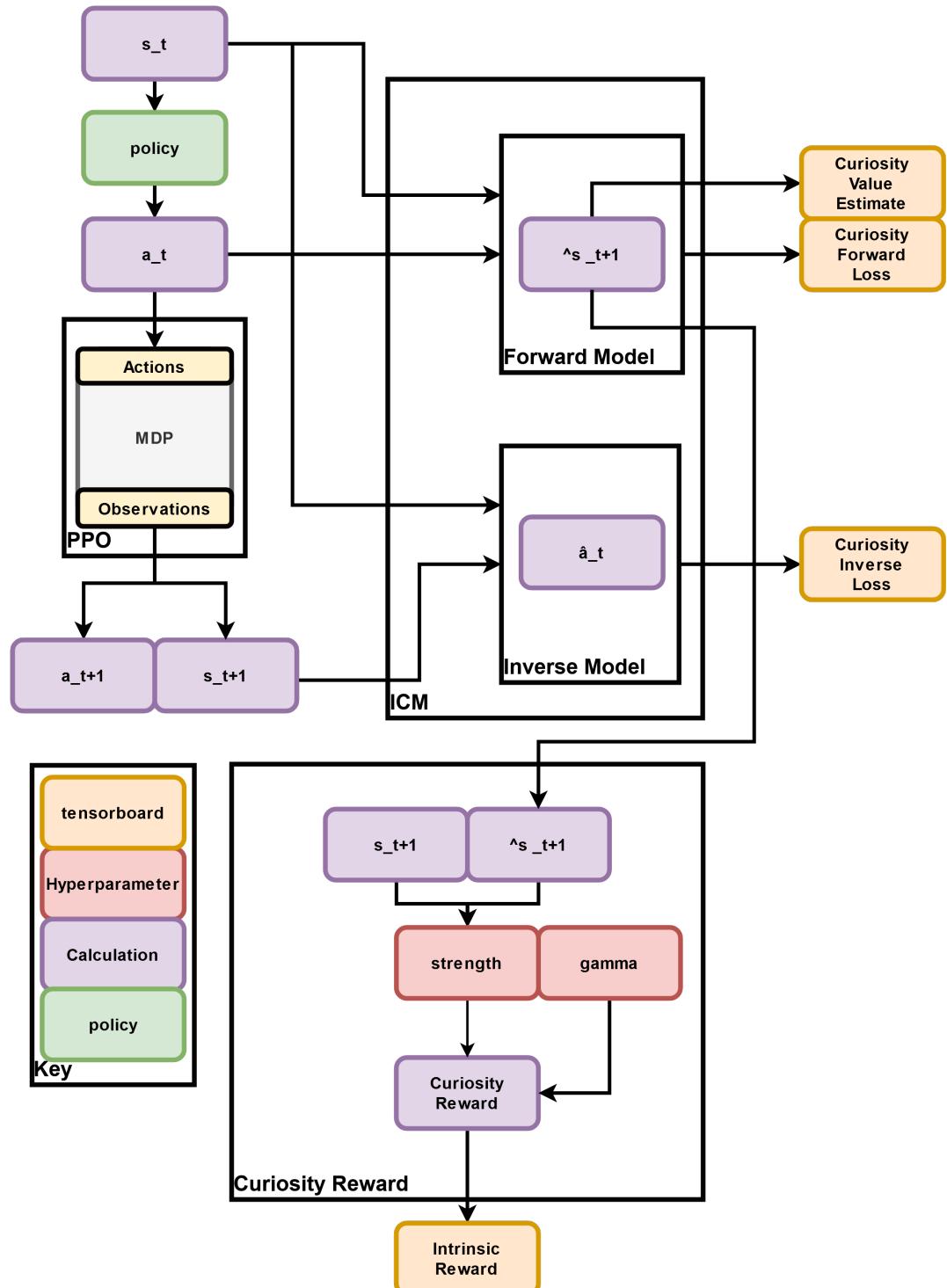


Abbildung 6.4.: Curiosity Learning mit Tensorboard: Das CL mit den Ausgabewerten von Tensorboard(Eigene Darstellung)[44, Abb. 2 S.3]

Name	Art	Verhalten
<i>Curiosity Value Estimate</i>	Der geschätzte Curiosity Value $\hat{V}(s)$	-
<i>Curiosity Forward Loss</i>	Wie gut das ICM in der Lage ist, <i>Observation</i> vorherzusagen	-
<i>Curiosity Inverse Loss</i>	Wie gut das ICM in der Lage ist, eine <i>Action</i> vorherzusagen	-
<i>Intrinsic Rewards</i>	Der intrinsische bzw. <i>Curiosity-Reward</i> $Q(a,s)$	-

Tabelle 6.2.: Tensorboard Auswertungstabelle für Curiosity Learning: Die Ausgabewerte des CL in Tensorboard auf Basis der Dokumentation in [17, Policy Statistics, Learning Loss Functions] von *ML-Agents* ohne Verhaltenswerte, da diese nicht angegeben sind.

## 6.4. Auswertung von Behaviour Cloning und Generative Adversarial Imitation Learning

Die folgenden beiden Abschnitte zeigen die Möglichkeiten zur Auswertung von BC und GAIL auf.

### 6.4.1. Auswertung von Behaviour Cloning

Die *Pretraining Loss* gibt Auskunft darüber, wie gut die *Agents* in der Lage sind, die *Demonstration*-Daten nachzuahmen - gibt dabei aber keine Auskunft über das Verhalten an [10].

### 6.4.2. Auswertung von Generative Adversarial Imitation Learning

Das Verhalten der Parameter zum Training mit GAIL ist in der *ML-Agents* Dokumentation nicht genau definiert [10, 17]. Daher zeigt die Tabelle 6.3, eine Auflistung der in der Dokumentation auf geführten Informationen zum Training mit GAIL - wobei das Verhalten der Kurven nicht aufgeführt ist.

Tensorboard	Art	Verhalten
<i>GAIL Loss</i>	<i>Discriminator Loss</i>	-
<i>GAIL Expert Estimate</i>	Die Schätzungen des <i>Discriminator</i> für $V(s)$ und $Q(a,s)$ die aus der Demo abgeleitet wurden	-
<i>GAIL Policy Estimate</i>	Die Schätzung des <i>Discriminator</i> für die Werte der angewandten <i>Policy</i>	-
<i>GAIL Value Estimate</i>	Die Schätzung für den GAIL Reward	-
<i>GAIL Reward</i>	Der <i>Discriminator</i> basierte Reward in einer Episode	-

Tabelle 6.3.: Tensorboard Auswertungstabelle für Generative Adversarial Imitation Learning: Tensorboard Parameter Training mit GAIL ohne Verhaltenweisen, da diese in der Dokumentation nicht definiert sind.

## 6.5. Auswertung mit der Unity-Engine

Über die Auswertung mit *Tensorboard* hinaus ist es auch möglich, ein Training subjektiv zu beurteilen, in dem das Verhalten der *Agents* im *Unity-Environment* per Beobachtung beurteilt wird. Zu dieser Auswertung befinden sich im digitalen Anhang *Builds* der Szenen.

Darüber hinaus wurde ein Video für das Spiel gegen einen menschlichen Spieler aufgezeichnet.



# 7. Resultate und Diskussion

Das folgende Kapitel zeigt die Ergebnisse der Experimente und deren anschließende Diskussion.

## 7.1. Experiment 1: Auswertung des SoccerTwos-Environment

Dieses Training diente der Feststellung, welche *buffer\_size* unter Anwendung mehrerer *Unity-Environments* zu wählen ist. Darüber hinaus wurde dieses Training auch zum Nachweis für die Reproduzierbarkeit eines Trainings genutzt.

### 7.1.1. Auswertung des Trainings für die Buffergröße

Die Abbildung 7.1 zeigt die Messwerte in *Tensorboard* für Experiment 1 5.2. Hier ist zu erwähnen, dass alle Werte der Abbildung, wie in Abschnitt 6 erläutert, mit der *Step*-Option dargestellt sind. Eine Ausnahme bildet hier die *Policy Loss*. Bei dieser wurde die Option *Relative* gewählt, da sich die Daten sonst verdeckt hätten.

Die Tabelle 7.1 stellt die Messwerte gegenüber. Die Ergebnisse zeigen, dass bei einer Trainingsdauer (*time*) von über neun Stunden durch die verzehnfachte *buffer\_size* ein Unterschied von *00h : 37min : 59sec* bzgl. der Trainingszeit eingetreten ist. Daraus lässt sich ableiten, dass eine Änderung der *buffer\_size* nicht zu einer erheblichen Änderung in der Trainingsdauer führt. Die Abbildung 7.1 zeigt den Unterschied der *ELO*. Diese zeigt für *SoccerTwos 1.0* in Orange einen deutlichen Anstieg gegenüber der Daten von *SoccerTwos 1.1* in blau. Es kann nicht eindeutig bestimmt werden, ob dieses Ergebnis auf die Änderung der *buffer\_size* zurückzuführen oder zufallsbedingt ist. Bei gleichbleibender Stichprobengröße und einer erhöhten Zahl an Trainingsdaten kann aber vermutet werden, dass die Wahrscheinlichkeit sinkt, ein gleichbleibendes Ergebnis zu erhalten.

Die Abbildung 7.1 zeigt die unterschiedlichen *Value-Loss* Ergebnisse. Hier ist zu erkennen, dass die blaue Kurve des *SoccerTwos 1.0* mit einer kleineren *buffer\_size* deutlich mehr Schwankungen unterliegt. Daraus kann abgeleitet werden, dass eine niedrigere *buffer\_Size* zwar eine instabile *Value-Loss* zufolge hat, aber zu einer besseren *ELO* führt. Das gleiche gilt umgekehrt für die *Policy-Loss*, d.h. eine instabile *Policy Loss* führt zu einem erfolgreicherem Training.

## 7. Resultate und Diskussion

---

*ML-Agents* beschreibt, dass sich dieser Wert in einem späteren Training stabilisieren sollte [17, Policy Loss]. Für *SoccerTwos 1.0* ist dies nicht der Fall, obwohl es das Training mit den insgesamt besseren Werten ist. Daraus kann gefolgert werden, dass eine hohe Schwankung zu besseren Ergebnissen führt.

Die Werte von *Cumulative Reward*, *Episode Length* waren fast identisch bei beiden Trainings. Der kumulative *Reward* geht wie zu erwarten<sup>1</sup> gegen Null. Der leichte Hang in den negativen Bereich ist durch die im Skript implementierte Zeitstrafe 4.2.1 erklärbar [7, Zeile 111]. Daraus ist abzuleiten, dass die *buffer\_size* keine Auswirkungen auf diese Werte hat.

Als Resultat aus Experiment 1 kann abgeleitet werden, dass die von *SoccerTwos* in den Hyperparameter vorkonfigurierte *buffer\_size* ausreichend groß für ein Training mit mehreren *Environments* ist. Für eine belastbarere Beurteilung der Auswirkungen der *buffer\_size* würden mehr Trainings benötigt werden. Bedingt durch die Trainingsdauer wurden keine weiteren Trainingsdaten erhoben.

Für Experiment 2 wurden als Resultat aus Experiment 1 die *buffer\_size* wie in Abschnitt 5.1.3 bzgl. der Komplexität nur verdoppelt, und nicht in Abhängigkeit der durch das *Python-Environments* zusätzliche gestarteten *Unity-Environments* angepasst.

Algorithmus		PPO, Self-Play	PPO, Self-Play
Bezeichnung		1.0	1.1
run-ID		SoccerTwo_0.1_-random_Soccer	SoccerTwo_-0.2_Soccer
<i>max_step</i>		5e7	5e7
<i>time</i>		09:08:22h	09:46:21h
<b>buffer_size</b>		20480	204800
	Erwartung	Ergebnis	Ergebnis
<i>ELO</i>	steigen	stark gestiegen	Kaum angestiegen
<i>Cumulative Reward</i>	Bei Null einpendeln	bei Null eingependelt	bei Null eingependelt
<i>Episode Length</i>	sinken	über Null eingependelt	über Null eingependelt
<i>Policy Loss</i>	weniger ausschlagen	weniger schwankend	stärker schwankend
<i>Value Loss</i>	steigen und später stabilisieren	relativ instabil	stabil
<i>Entropy</i>	sinken	schnell gesunken	langsamer gesunken
<i>Extrinsic Reward</i>	steigen	gestiegen	schwankend um null
<i>Extrinsic Value Estimate</i>	steigen	gestiegen	konstant wenig
<i>learning_rate</i>	konstant	konstant	konstant

Tabelle 7.1.: Experiment 1 Bewertungstabelle 1.0 und 1.1: Experiment 1 Bewertungstabelle auf Basis der Bewertungsgrundlage in Kapitel 6 für das Training von PPO in Kombination mit SP. Erstellt auf der Basis der in Kapitel 6 erläuterten Bewertungsmöglichkeiten durch Tensorboard [17].

---

<sup>1</sup>siehe 6.2

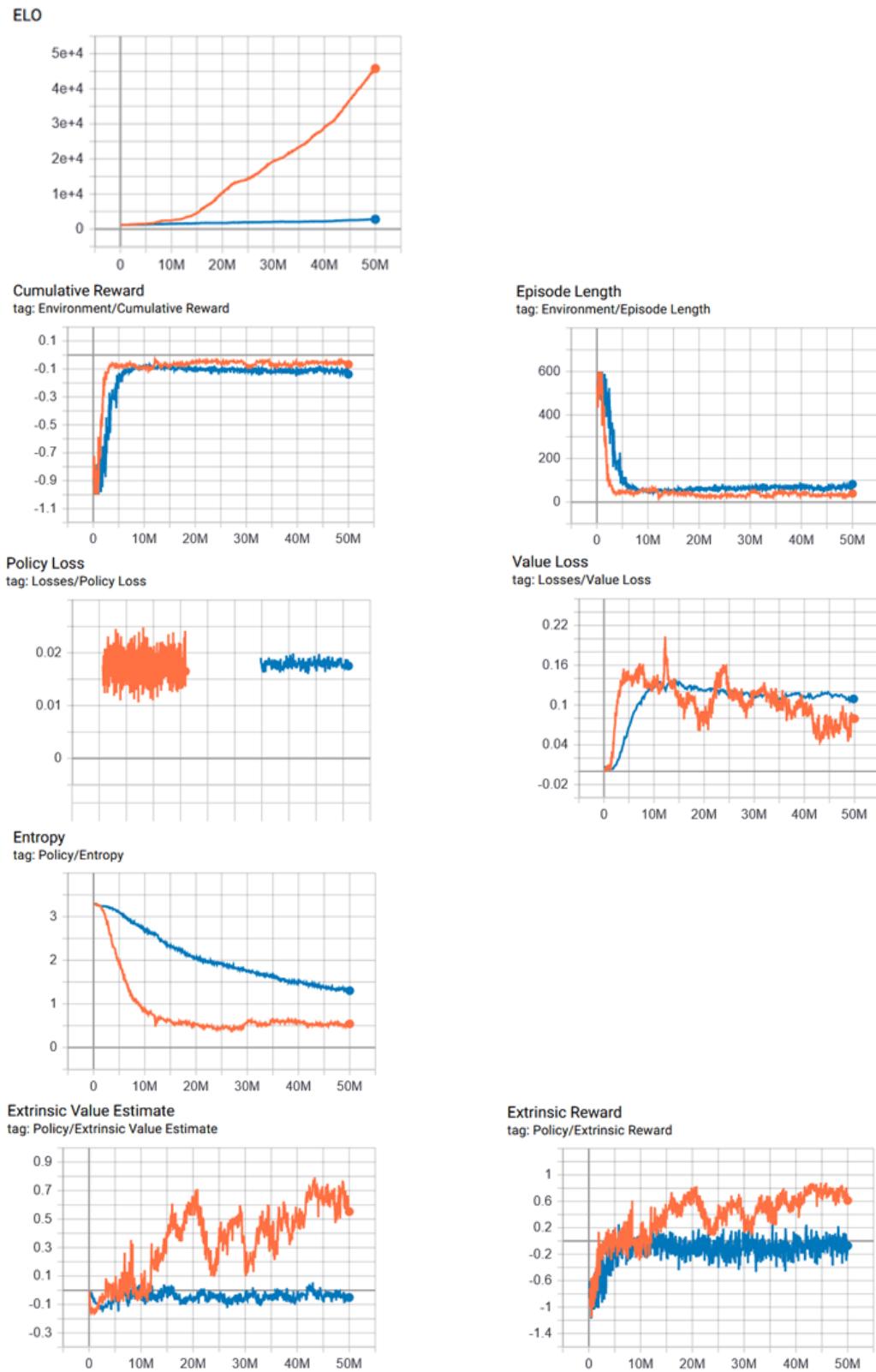


Abbildung 7.1.: Experiment 1 mit 1.0 und 1.1: Screenshots der Ergebnisse des Experiment 1 in *Tensorboard*. Hier im Vergleich aufgeführt, das Training SoccerTwos1.0 in orange und das Training SoccerTwos1.0 in blau.

### 7.1.2. Auswertung zur Reproduzierbarkeit

Die Abbildung 7.2 zeigt die Screenshots aus *Tensorboard* für die Ergebnisse des Trainings zur Reproduzierbarkeit, mit den Trainingsdaten von *SoccerTwos 1.0* in orange und *SoccerTwos 1.1* in grau.

Dabei war die Dauer ca. gleich auf mit *9h:8m:22s* für *SoccerTwos 1.0* und *9h:46m:47s* für *SoccerTwos 1.1*. Die *ELO* von *SoccerTwos 1.0* in Grau bricht dabei bei ca. fünfzehn Millionen *Steps* ein, verbessert sich aber ab ca. vierzig Millionen *Steps* wieder.

*Cumulativ Reward*, *Episode Length*, *Policy Loss*, *Value Loss* und *Entropy* sind dabei fast identisch.

Der *Extrinsic Value Estimate* und der *Extrinsic Reward* unterscheiden sich dabei zwar in ihren absoluten Werten, aber sind sich insgesamt relativ ähnlich. Der Einbruch dieser beiden Werte auf den Nullpunkt lässt sich mit der *ELO* bei ca. fünfzehn Millionen *Steps* in Zusammenhang bringen.

Die Werte sind nicht exakt reproduziert und die *Elo* bricht mit dem *Extrinsic Value Estimate* und dem *Extrinsic Reward* gemeinsam an bestimmten Stellen ein. Dies könnte auf eine wahrscheinlichkeitbasierte Fehlentscheidung des neuronalen Netzes zurückzuführen sein. Daher werden die Ergebnisse des Trainings als bedingt reproduzierbar eingestuft.

Auf eine Darstellung in einer Tabelle wurde in diesem Fall verzichtet, da die Hyperparameter identisch und die Kurven wie auch die Trainingsdauer ähnlich waren.

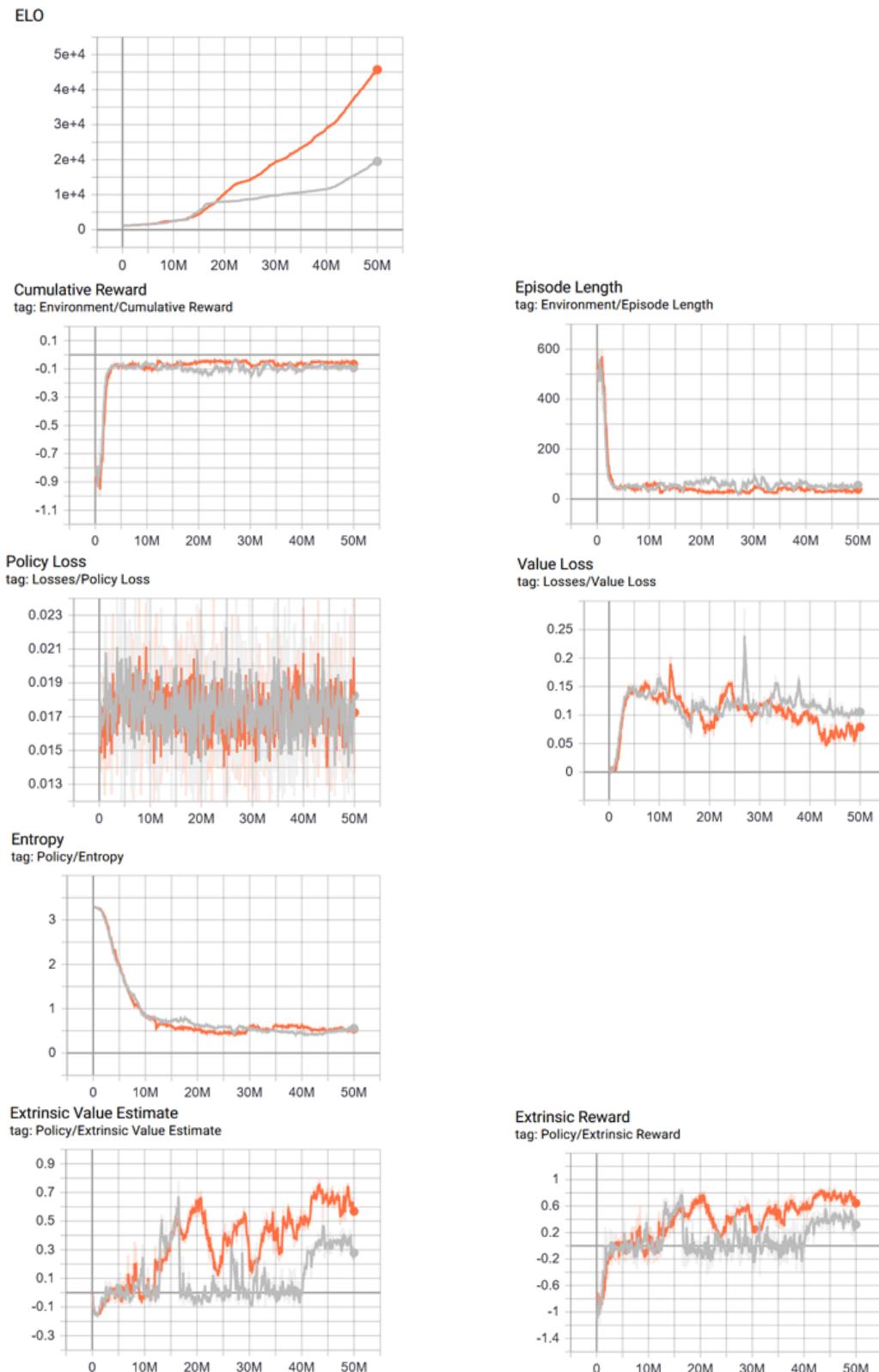


Abbildung 7.2.: Experiment 1 mit 1.0 und 1.2: Screenshots der Ergebnisse des Experiment 1 in Tensorboard. Hier im Vergleich aufgeführt, das Training SoccerTwo1.0 in orange und einer Wiederholung mit identischen Parametern in grau.

### 7.1.3. Auswertung unter Verwendung des Unity-Editors

Die subjektive Auswertung unter Verwendung des *Unity-Editors* mit dem *Model* von Experiment 1.0, konnte zeigen das die *Agents* dazu in der Lage waren, ein Spiel zu absolvieren. Dabei wurden abwechselnd von beidem Mannschaften Tore erzielt, und auch vereinzelt ein Eigentor geschossen. Insgesamt verhielten sich die *Agents* aber so wie es zu erwarten war, dass sie ihr eigenes Tor verteidigen, und in Richtung des gegnerischen Tor spielen.

## 7.2. Experiment 2: Auswertung der Prototyp-Umgebung

Das zweite Experiment diente der Festlegung der Hyperparameter *hidden\_units*, *num\_layers* für das neuronale Netz und einer anschließenden Ausführung eines Trainings mit diesen Hyperparametern.

### 7.2.1. Vorbereitendes Training

Die Tabelle 7.3 stellt die Ergebnisse für das Training 2.0 und 2.1 dar. Dabei kann daraus geschlossen werden, dass eine Vergrößerung der Hyperparameter für das neuronale Netz von *hidden\_units*, *num\_layers* zu einem deutlichen Anstieg der Trainingszeit führt. Als Resultat aus der angestiegenen Trainingszeit wurden diese vorbereitenden Trainings mit ca. 2.35 Millionen *steps* und nicht wie Experiment 1 mit fünfzig Millionen *steps* ausgeführt.

Dabei ist hier zu erwähnen, dass bei einem vollständigen Training die Ergebnisse vermutlich aussagekräftiger wären. Um dennoch einen Vergleichswert mit Experiment 1.0 zu erhalten, ist das Training mit der Konfiguration von Experiment 2.2 vollständig ausgeführt.

Die Grafik 7.3 zeigt im oberen Bereich die Ergebnisse der *ELO*. Das Training 2.0 wird in rot, das Training 2.1 in grün dargestellt. Beim Training 2.0 ist zu erkennen, dass Ergebnisse ca. um den Mittelwert  $1.2e + 3$  schwanken. Beim Training 2.1 ist im ersten Drittel ein Abfall des Werts zu sehen. Bei einer Million vierhundertfünftausend *steps* ist jedoch ein Trend nach oben festzustellen. Es ist nach dem Anstieg am Ende des zweiten Drittels zu vermuten, dass beim Training mit dem größeren Netzwerk ein Trend nach oben zu erwarten ist. Dies kann jedoch auch zufallsbedingt sein. Bei der weiteren Analyse der Trainingsdaten in der Abbildung liefern alle Daten eher schlecht einzuordnende Ergebnisse. Als Erkenntnis aus diesem Vorexperiment wurde aufgrund der Trainingsdauer die kleinere Konfiguration aus *num\_layers* und *hidden\_units* gewählt. Es ist darüber hinaus zu erwähnen, dass sich vermutlich durch Zufall, die Messwerte genau am Trainingsende getroffen haben.

Des Weiteren wird aus den Ergebnissen abgeleitet, dass das Training in diesem kurzen Zeitraum instabil ist und deshalb vermutlich noch Anpassung benötigt. In Anbetracht der weniger benötigten Zeit dient das Training 2.0 als Basis für das Training 2.2 und 3.0.

Algorithmus		PPO und Self-Play	PPO und Self-Play
Bezeichnung		2.0	2.1
run-ID		NeonShifter_PPO-SelfPlay-small_Buffer-1.0_NeonShifter	NeonShifter_PPO-SelfPlay-small_Buffer-1.1_NeonShifter
max_step		2.35e6	2.34e6
time		01h:51min:32sec	04h:47min:02sec
buffer_size		40960	40960
num_layers		2	3
hidden_units		512	1024
	Erwartung	Ergebnis	Ergebnis
<i>ELO</i>	steigen	flach	sinkt zeitweise ab
<i>Cumulative Reward</i>	Bei Null einpendeln	schwankt um -1	schwankt um Null
<i>Episode Length</i>	sinken	steigt stark an und schwankt stark	steigt stark an und schwankt stark
<i>Policy Loss</i>	weniger ausschlagen	schwankt schwach	schwankt schwach
<i>Value Loss</i>	steigen und später stabilisieren	fällt stark ab und schwankt leicht	fällt stark ab und schwankt leicht
<i>Entropy</i>	sinken	fällt sehr leicht ab	fällt sehr leicht ab
<i>Extrinsic Reward</i>	steigen	schwankt um -1	schwankt um -1
<i>Extrinsic Value Estimate</i>	steigen	fällt ab und steigt nicht an	fällt ab und steigt nicht an
<i>learning_rate</i>	konstant	konstant	konstant

Tabelle 7.2.: Experiment 2 Bewertungstabelle 2.0 und 2.1: Die Bewertungstabelle des Experiments 2 für das Training von PPO in Kombination mit dem SP für die Festlegung der Hyperparameter des neuronalen Netzes mit dem *NEON SHIFTER-Example*. Erstellt auf der Basis der in Kapitel 6 erläuterten Bewertungsmöglichkeiten durch Tensorboard [17].

## 7. Resultate und Diskussion

---

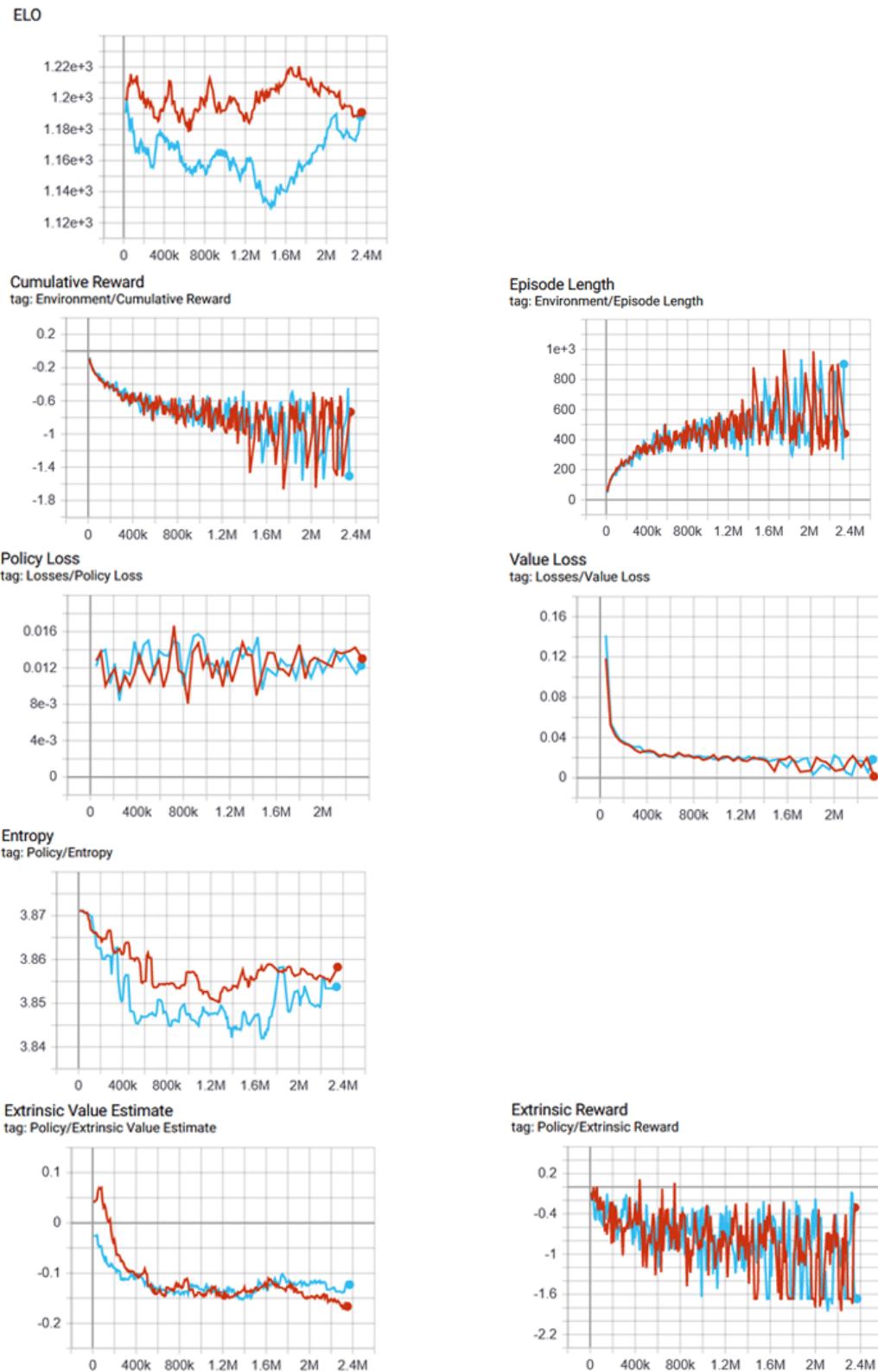


Abbildung 7.3.: Experiment 2 mit 2.0 und 2.1: Screenshots der Ergebnisse des Experiment 2.0 in rot und 2.1 in blau aus *Tensorboard*. Hier im Vergleich aufgeführt, Training 2.0 und 2.1 zur Festlegung der Hyperparameter.

### 7.2.2. Vollständiges Training

Das Experiment 2.2 ist mit fünfzig Millionen *max\_steps* ausgeführt worden. Durch die Ergebnisse von 2.0 und 2.1 wurden die Hyperparameter von 2.0 übernommen. Dieses Training dient dem Vergleich der Leistungsfähigkeit des Prototyps mit *SoccerTwos*. Die Abbildung 7.4 und die Tabelle 7.3 zeigen dabei die Gegenüberstellung der Messwerte.

Wie aus der Tabelle hervorgeht, hat sich die Trainingsdauer von ca. 9h auf ca. 2d verfünfacht. Bei der Abbildung ist zunächst die *ELO* zu betrachten. Diese zeigt eindeutig, dass beim Prototyp ein kaum merklicher Anstieg zu verzeichnen ist. Werden dabei zusätzlich die Werte vom Experiment 1.0 betrachtet, liegt das *ELO*-Ergebnis gleich auf. Daraus kann abgeleitet werden, dass die Größe des Netzwerks angepasst werden muss. Des Weiteren zeigt sich beim *CumulativeReward* eine starke Schwankung und eine große Entfernung vom Nullpunkt. Dies lässt annehmen, dass bei der Implementierung des SP Fehler vorliegen, da dieser wert ca. bei Null liegen sollte. Es kann aber auch daraus geschlossen werden, dass kein *Agent* dazu in der Lage ist ein Tor zu schießen und daher eine hohe Zeitstrafe sammelten. Dies lässt sich auch aus der *Episode Length* schließen, da diese schwankend ist und ansteigt. Die *Policy Loss* ist weniger stark schwankend und liegt im Mittel unter dem *SoccerTwos*. Die *Value Loss* ist während des gesamten Trainings auf einem niedrigen Wert in Vergleich zu den Ergebnissen des *SoccerTwos*-Training, was im Vergleich zum *SoccerTwos* ein wenig erfolgreiches Training vermuten lässt. Dies lässt sich auch aus der *Entropy* schlussfolgern, da diese kaum bemerkbar sinkt. Aus diesen Trainingsdaten lässt sich vermuten, dass noch einige Anpassungen des Prototyps notwendig sind.

Algorithmus		PPO, Self-Play	PPO, Self-Play
Bezeichnung:		1.0	2.2
run-ID		SoccerTwo_0.1_random_Soccer	NeonShifter_-PPO_SelfPlay_-small_Buffer-_1.2_NeonShifter
<i>max_step</i>		5e7	5e7
<i>time</i>		09h:08min:22sec	2d:2h:34min:15sec
<i>buffer_size</i>		20480	40960
<i>num_layers</i>		2	2
<i>hidden_units</i>		512	512
	Erwartung	Ergebnis	Ergebnis
<i>ELO</i>	steigen	stark gestiegen	ganz langsam gestiegen
<i>Cumulative Reward</i>	Bei Null eingependeln	bei Null eingependelt	bei -1 eingependelt
<i>Episode Length</i>	sinken	über Null eingependelt	schwankend und ansteigend
<i>Policy Loss</i>	Schwankung sollte sinken	schwankt durchgehend	schwankt durchgehend aber nicht so stark wie SoccerTwos
<i>Value Loss</i>	steigen und später stabilisieren	relativ instabil	durchgehend konstant
<i>Entropy</i>	sinken	schnell gesunken	sinkt kaum
<i>Extrinsic Reward</i>	steigen	gestiegen	konstant unter Null
<i>Extrinsic Value Estimate</i>	steigen	gestiegen	konstant
<i>learning_rate</i>	konstant	konstant	konstant

Tabelle 7.3.: Bewertungstabelle mit Experiment 1.0 und 2.2: Die Bewertungstabelle für eine Gegenüberstellung von Experiment 1.0 und 2.2 zum Vergleich der Leistungsfähigkeit. Erstellt auf der Basis der in Kapitel 6 erläuterten Bewertungsmöglichkeiten durch Tensorboard [17].

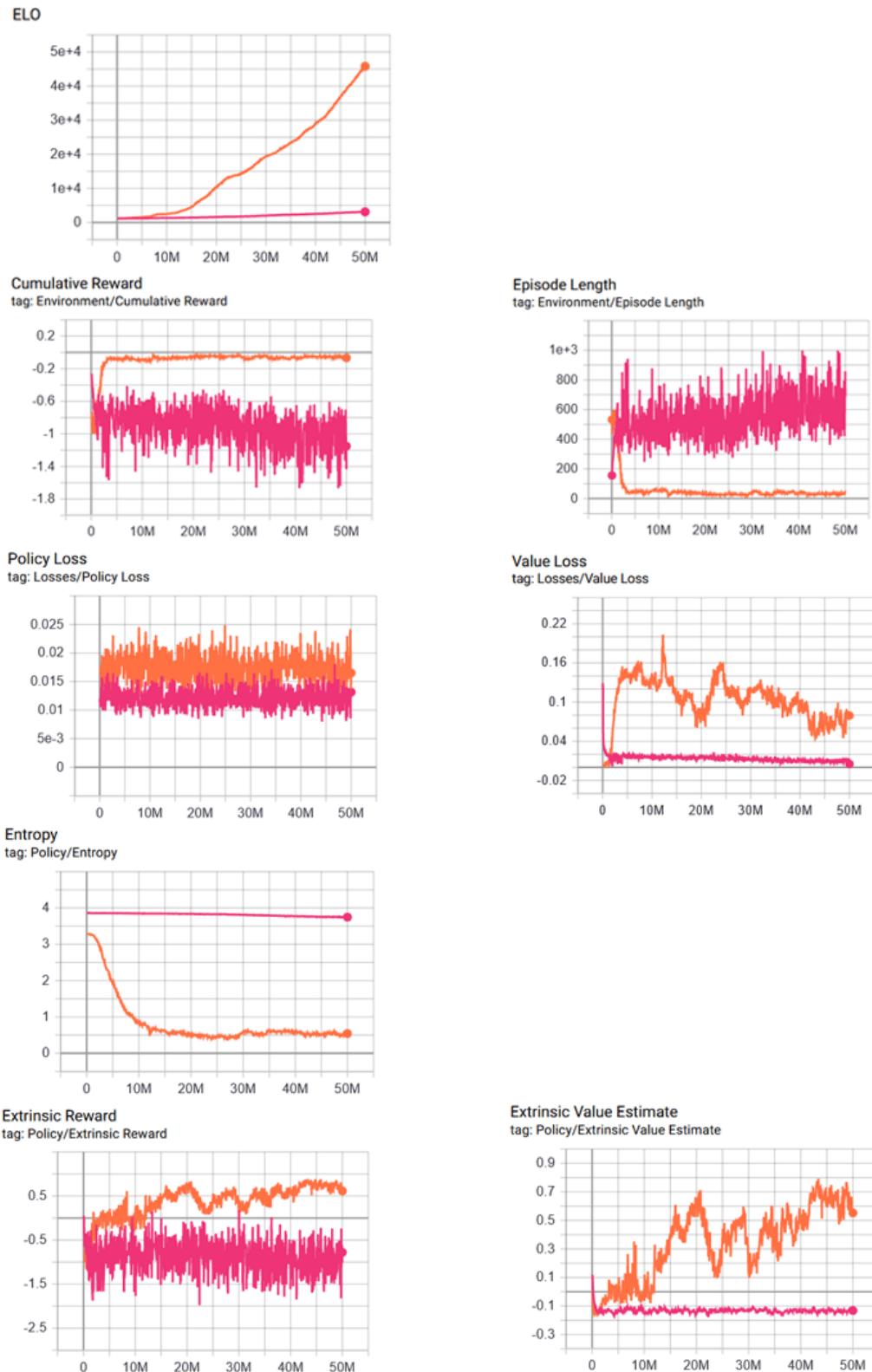


Abbildung 7.4.: Experiment 2 mit 1.0 und 2.2: Screenshots der Ergebnisse aus *Tensorboard* des Experiment 1.0 in orange im Vergleich zum Training 2.2 in rot.

### 7.2.3. Auswertung unter Verwendung des Unity-Editors

Durch die subjektive Auswertung in der *Unity-Szene* konnte festgestellt werden, dass die *Agents* keine erkennbare *Policy* ausführten. Es fielen zwar Tore, diese machten aber einen eher zufälligen Eindruck. Die Positionierung der *Agents* auf dem Feld schien im Hinblick auf den Spielpartner bewusst, doch der Ball wurde ignoriert.<sup>2</sup>

## 7.3. Experiment 3: Auswertung der Prototyp-Umgebung mit Erweiterung

Dieser Abschnitt wertet das dritte Experiment aus. Das Experiment 3 diente der Anwendung von Rechercheergebnissen, um die Leistungsfähigkeit zu steigern. Die Tabelle 7.4 stellt die Trainings von Experiment 1.0, 2.2 und 3 gegenüber. Dabei zeigt sich, dass das Experiment 3 im Vergleich zu Experiment 2.2 eine deutliche Steigerung der Trainingszeit ergibt. Von *2d:2h:34min:15sec* für 2.2 zu *9d:20h:8min:21sec* für 3.0. Eine Vergrößerung um über 6 Tage.

Des Weiteren kann der Tabelle auch entnommen werden, dass das Verhalten der visualisierten Werte von *Tensorboard* deutlich besser ausgefallen ist. Die Werte sind in 7.5 aufgeführt. Dabei zeigt die *ELO* des Experiment 3 in grün eine deutliche Anhebung zum Experiment 2 in rot. Sie verläuft ab ca. zwanzig Millionen *Steps* fast parallel zu den Messwerten des *SoccerTwos* Beispiel in orange.

Der *Cumulative Reward* zeigt eine Stabilisierung des Messwerts. Hier kann vermutet werden, dass bei einer längeren Trainingszeit der *Cumulative Reward* weiter stabilisieren wird. Die *Episoden Length* weist eine klare, stabilisierende Tendenz gegen Null auf. Dabei sind die Werte klar besser einzuordnen als bei Experiment 2.2, liegen jedoch noch klar unter dem von Experiment 1.0.

Die *Policy Loss*-Werte von Experiment 2.2 und 3.0 überlappen sich und sind fast identisch, wohingegen die Werte von Experiment 1.0 leicht erhöht sind und etwas instabiler erscheinen. An der *Value Loss* lässt sich klar erkennen, dass Experiment 2.2 und 3.0 fast aufeinander liegen und sich, verglichen mit Experiment 1.0, kaum geändert haben. Die *Entropy* bewegt sich tendenziell von Experiment 2.2 weg, ist aber noch weit entfernt von Experiment 1.0. Aus diesen Werten lässt sich keine klare Verbesserung ableiten.

Der *Extrinsic Value Estimate* von Experiment 2.2 und 3.0 befindet sich bis ca. fünfzehn Millionen *Steps* gleichauf. Danach zeigt der Wert einen deutlichen Anstieg über den Nullpunkt, jedoch um einiges stabiler als der *Extrinsic Value Estimate*. Der *Extrinsic Reward* von 3.0 hat sich im Verlauf des Trainings klar über den Nullpunkt und die Ergebnisse von 2.2 abgesetzt. Dabei lässt sich aus diesen Werten schließen, dass ein Training mit einer hö-

---

<sup>2</sup>Siehe dazu im digitalen Anhang den *Build* zur Szene im Ordner:  
AI\_BUILDS\NEON\_SHIFTER\_AI\_BUILD\_2.2

heren *max-Step* vermutlich bessere Ergebnisse liefern würde - das mehr Zeit in Anspruch nehmen würde.

Darüber hinaus erstellte das Training auch Daten zum Training mit BC, RL, GAIL und CL. Die Daten werden hier nicht weiter ausgewertet, da aufgrund der Trainingsdauer keine Vergleichsdaten erhoben werden konnten. Die Daten sind in Abschnitt 1 im Anhang einzusehen.

### 7.3.1. Auswertung unter Verwendung des Unity-Editors

Nach Auswertung der Trainingsdaten ist das trainierte *Model* zusätzlich noch im *Unity-Environment* mit den *Agents* live getestet worden. Dazu wurde das erstellte *Model* vom Experiment 3.0 verwendet.

Zunächst wurde in der Szene der *Goal Change* abgeschaltet, um den *Agents* das finden des Tors zu erleichtern. Die *Agents* des blauen Teams wiesen keine Reaktion auf, weswegen sie durch eine Kopie der *Agents* des roten Teams mit angepassten Tags ersetzt wurden. Diese Anpassung konnten die inaktiven *Agents* beheben.

Des Weiteren konnte festgestellt werden, dass die *Agents* auf das falsche *Goal* konzentrieren, also das rote Team verteidigt das blaue Tor und schießt auf das rote Tor. Daher wurden die Tags entsprechend getauscht.

Nach den oben genannten Änderungen konnte festgestellt werden, dass die *Agents* ein Spiel mit klar erkennbarer *Policy* ablieferten. Mehrheitlich positionierten sich die *Agents*, abgestimmt auf ihren Spielpartner, sinnvoll und schossen Tore für ihr Team. Darüber hinaus ist klar erkennbar, dass die *Agents* meist zielstrebig auf den Ball hinzu bewegten, je nachdem wo sich der Ball befand. Darüber hinaus ist wie in [28, Self-play and Soccer Environment] erwähnt, die Strategie zu erkennen, dass sich einer der zwei Spieler bei Bedarf eher auf die defensive im Spiel konzentriert, während ein anderer Spieler eher eine offensive *Policy* einsetzt. Dabei wechselte sich die *Agents* mit der Spielposition sich Lückenlos ab.<sup>3</sup>

Zusätzlich wurde in diesem Anwendungsfall ein Agent durch einen menschlichen Spieler ersetzt. Dabei konnte die Erkenntnis gewonnen werden, dass die *Agents* zwar durchaus zu besiegen waren, aber das Spiel sich durch ihre Anwesenheit als durchaus komplexer und anspruchsvoller gestaltete. Die *Shields* wurden von den *Agents* häufig aktiviert, unabhängig davon, ob ein Gegner oder Ball in der Nähe war. Dies führte zu durchaus erfolgreichen Strategien. Es ist zu vermuten, dass die *Agents* diese *Policy* gelernt haben, um sich zu jeder Zeit vor dem Ball schützen zu können, der sie für kurze Zeit betäuben würde.<sup>4</sup>

---

<sup>3</sup>Siehe dazu im digitalen Anhang den *Build* zur Szene im Ordner:  
AI\_BUILDS\NEON\_SHIFTER\_AI\_BUILD\_3.0

<sup>4</sup>Siehe dazu im digitalen Anhang das Video zur Szene mit dem Dateinamen:  
Experiment\_3.0\_3\_Agents\_1\_Human.mp4

## 7. Resultate und Diskussion

---

Algorithmus		PPO, Self-Play	PPO, Self-Play	PPO, Self-Play, IL, GAIL, BC
Bezeichnung:		1.0	2.2	3.0
run-ID		SoccerTwo_0.1_random_Soccer	NeonShifter_PPO_SelfPlay_small_Buffer_1.2_NeonShifter	NeonShifter_PPO_SelfPlay_BC_small_Buffer_3.0_NeonShifter
<i>max_step</i>		5e7	5e7	5e7
<i>time</i>		09h:08min:22sec	2d:2h:34min:15sec	9d:20h:8min:21sec
<b>buffer_size</b>		20480	40960	40960
<b>num_layers</b>		2	2	2
<b>hidden_units</b>		512	512	512
	Erwartung	Ergebnis	Ergebnis	Ergebnis
<i>ELO</i>	steigen	stark gestiegen	ganz langsam gestiegen	Ca. ab der Mitte stark gestiegen
<i>Cumulative Reward</i>	bei Null eingependeln	bei Null eingependelt	bei -1 eingependelt	Stabilisiert sich und geht gehen Null
<i>Episode Length</i>	sinken	über Null eingependelt	schwankt und ansteigend	Stabilisiert sich und geht gehen Null
<i>Policy Loss</i>	Schwankung sollte sinken	schwankt durchgehend	schwankt durchgehend aber nicht so stark wie SoccerTwos	Ähnlich zum Training 2.2
<i>Value Loss</i>	steigen und später stabilisieren	instabil	durchgehend konstant	Fast Identisch zum Training 2.2
<i>Entropy</i>	sinken	schnell gesunken	sinkt kaum	Fällt etwas mehr als 2.2 ab
<i>Extrinsic Reward</i>	steigen	gestiegen	konstant unter Null	Steigt über Null
<i>Extrinsic Value Estimate</i>	steigen	gestiegen	konstant	Steigt über Null
<i>learning_rate</i>	konstant	konstant	konstant	konstant

Tabelle 7.4.: Experiment 3 Ergebnistabelle mit 1.0, 2.2 und 2.0: Die Ergebnisse von Experiment 1.0, 2.2 und 3.0. Dabei sind die Trainings 1.0 links, 2.2 in der Mitte und 3.0 rechts dargestellt. Erstellt auf der Basis der in Kapitel 6 erläuterten Bewertungsmöglichkeiten durch Tensorboard [17].

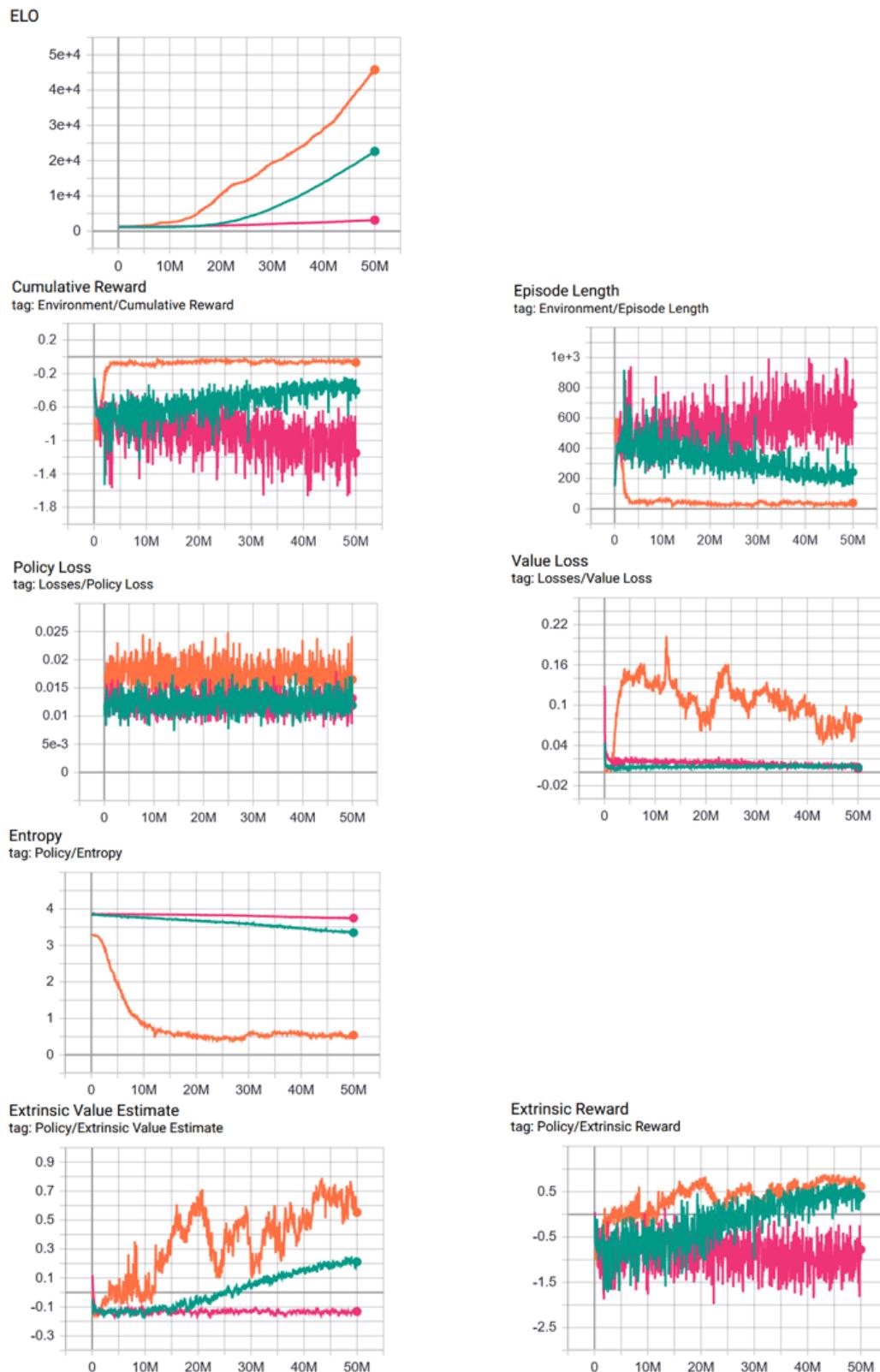


Abbildung 7.5.: Experiment 3 mit 1.0, 2.2 und 3.0: Screenshots aus *Tensorboard* für die Ergebnisse der Trainings 1.0 in orange, 2.2 in rot und 3.0 in grün.



## Teil III.

### Zusammenfassung und Ausblick



# 8. Fazit und Ausblick

In den folgenden Abschnitten befindet sich das Fazit zu dieser Thesis sowie ein Ausblick, der weitere Forschungsansätze zu diesem Thema liefert.

## 8.1. Fazit

*ML-Agents* bietet mit seinem Plug-in einen guten Einstieg in den Bereich des RL. Dabei gibt die Dokumentation Empfehlungen, wie *ML-Agents* konfiguriert werden kann. Die Thesis bietet im Zuge, über die Dokumentation hinaus, einen Einblick in den theoretischen Hintergrund hinsichtlich der verwendeten *Examples*. Dieser beansprucht dabei keine Vollständigkeit.

*ML-Agents* stellt mit seinem Angebot an *Example-Environments* eine umfangreiche Basis zur Integration und Anwendung zur Verfügung. Mit *Tensorboard* bietet *ML-Agents*, über eine subjektive Beurteilung hinaus, die Möglichkeit, die Trainingsdaten auszuwerten. Mit Anwendung der Dokumentation und der *Example-Environments* lässt sich die Auswertung eigener *Examples* vereinfachen.

Das Training des *Example-Environments SoccerTwos* konnte zeigen, dass Trainingsergebnisse bedingt reproduzierbar sind und sich anhand dieser, die Konfiguration für eigene *Environments* ableiten lässt.

Die Übernahme der Konfiguration, deren Anpassung im Hinblick auf die vergrößerte Komplexität und ein kurzes Vortraining konnten zunächst keine leistungsfähigen AI erzeugen. Eine weitere Anpassung durch Anwendung von Rechercheergebnissen konnte darüber hinaus zu einer deutlichen Verbesserung führen.

Nach Beurteilung der Experimente ist aus den Ergebnissen zu schließen, dass *ML-Agents*, verglichen mit dem *Example-Environment*, dazu in der Lage ist, eine leistungsfähige AI zu erzeugen. Darüber hinaus ist auch bei einem Vergleich mit einem menschlichen Spieler die AI dazu fähig, einen Gegner zu bieten. Für eine spätere Verwendung im fertigen Game reicht diese vermutlich noch nicht. Die Ergebnisse sprechen aber dafür, dass dies nach einer weiteren Anpassung möglich ist.

Hier sei zu erwähnen, dass einige Aspekte von *ML-Agents* auf Grund der Bearbeitungsdauer der Thesis nicht behandelt werden konnten, da RL und auch *ML-Agents* ein sehr umfangreiches Thema bieten. Darüber hinaus konnten die Grundlagen im Rahmen der Thesis nicht zu einer Verbesserung der Ergebnisse führen.

## 8.2. Ausblick

Bereiche von *ML-Agents* und RL konnten in dieser Thesis nicht bearbeitet werden. Dies soll im Rahmen der Entwicklung von *NEON SHIFTER* nachgeholt werden. *ML-Agents* wird eine leistungsfähige AI in *NEON SHIFTER* ermöglichen können.

Dabei sind über die Anwendung des *Example-Environemnts* hinaus, weitere Experimente geplant. Diese beinhalten unter anderem die Verwendung eines leistungsfähigen Servers, um die Trainingsgeschwindigkeit zu beschleunigen. Eine Verwendung des CL zum Zerlegen des Trainings in Teilaufgaben, die Verwendung des SAC-Algorithmus und die Nutzung eines RNN wären eine Option, das Training zu optimieren.

Darüber hinaus sind weitere Anpassungsversuche der Hyperparameter geplant, um die Ergebnisse zu verbessern. Dabei ist eine gezielte Recherche zum theoretischen Hintergrund der Teilbereiche des RL angedacht. Eine Implementierung ohne Verwendung von *Raycast*s wird in Betracht gezogen.

Ein zusätzlicher Ansatz wäre die Anwendung von *ML-Agents* zur Auswertung pixelbasierter Datensätze. Dabei stellt sich die Frage, ob diese auch für Realbilddaten genutzt werden können, um so Trainingsergebnisse für reale Anwendungen zu liefern. Eine weitere Frage wäre, ob diese Daten bspw. von einem Arduino über W-LAN geliefert werden können. Interessant wäre es auch, das Potenzial für *ML-Agents* in Hinblick auf das *Balancing* eines Games, sei es bzgl. der Spielmechanik oder des Leveldesigns. Eine automatische Anpassung der Leistungsfähigkeit der AI wäre für eine spätere Umsetzung des Games nützlich.

# Literatur- Quellenverzeichnis

- [1] Josh Achiam. *Soft Actor-Critic*. Zugriffsdatum: 12.04.2020, Copyright 2018, OpenAI. Revision 038665d6., 2018. URL: <https://spinningup.openai.com/en/latest/algorithms/sac.html#documentation>.
- [2] Anupam Bhatnagar et al. *Example Scenes ML-Agents*. Zugriffsdatum: 14.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/tree/release-0.14.1/Project/Assets/ML-Agents/Examples/Soccer>.
- [3] Anupam Bhatnagar et al. *Example Scenes ML-Agents*. Zugriffsdatum: 14.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/tree/release-0.14.1/Project/Assets/ML-Agents/Examples/Pyramids>.
- [4] Anupam Bhatnagar et al. *ML-Agents Beta 0.14.1*. Zugriffsdatum: 16.03.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/releases>.
- [5] Chris Elion et al. *ml-agents/Learning-Environment-Design-Agents*. Pfad in der Dokumentation: ml-agents/docs/Learning-Environment-Design.md, Zugriffsdatum: 01.05.2020, März 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Learning-Environment-Design-Agents.md>.
- [6] Chris Elion et al. *ml-agents/Unity-Inference-Engine*. Pfad in der Dokumentation: ml-agents/docs/Unity-Inference-Engine.md, Zugriffsdatum: 06.04.2020, März 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Unity-Inference-Engine.md>.
- [7] Chris Elion et al. *trainer\_config.yaml*. Zugriffsdatum: 14.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/dd3fd8059446684a2e2e5974c5004c87c3da6777/Project/Assets/ML-Agents/Examples/Soccer/Scripts/AgentSoccer.cs#L111>.
- [8] Chris Elion et al. *Training Using Concurrent Unity Instances*. Pfad in der Dokumentation: ml-agents/docs/Training-Using-Concurrent-Unity-Instances.md, Zugriffsdatum: 19.04.2020, Okt. 2019. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Training-Using-Concurrent-Unity-Instances.md>.

- [9] Chris Elion et al. *Unity ML-Agents Toolkit (Beta)*. Zugriffsdatum: 20.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/tree/release-0.14.1>.
- [10] Chris Elion et al. *Using TensorBoard to Observe Training*. Zugriffsdatum: 03.05.2020, Apr. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Using-Tensorboard.md>.
- [11] Ervin T. et al. *Example Learning Environments*. Pfad in der Dokumentation: ml-agents/docs/Learning-Environment-Examples.md, Zugriffsdatum: 14.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Learning-Environment-Examples.md>.
- [12] Ervin T. et al. *gail\_config.yaml*. Zugriffsdatum: 14.05.2020, Feb. 2020. URL: [https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/config/gail\\_config.yaml](https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/config/gail_config.yaml).
- [13] Ervin T. et al. *trainer\_config.yaml*. Zugriffsdatum: 14.05.2020, Feb. 2020. URL: [https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/config/trainer\\_config.yaml](https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/config/trainer_config.yaml).
- [14] Ervin T. et al. *Training ML-Agents*. Pfad in der Dokumentation: ml-agents/docs/Training-ML-Agents.md, Zugriffsdatum: 01.05.2020, Jan. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Training-ML-Agents.md>.
- [15] Ervin T. et al. *Training with Proximal Policy Optimization*. Pfad in der Dokumentation: ml-agents/docs/Training-PPO.md, Zugriffsdatum: 01.05.2020. Apr. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Training-PPO.md>.
- [16] Ervin T. et al. *Training with Soft-Actor Critic*. Pfad in der Dokumentation: ml-agents/docs/Training-SAC.md, Zugriffsdatum: 15.04.2020, Apr. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Training-SAC.md>.
- [17] Ervin T. et al. *Using TensorBoard to Observe Training*. Pfad in der Dokumentation: ml-agents/docs/Using-Tensorboard.md Zugriffsdatum: 16.03.2020, März 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Using-Tensorboard.md>.
- [18] Marwan Mattar et al. *Making a New Learning Environment*. Pfad in der Dokumentation: ml-agents/docs/Learning-Environment-Examples.md, Zugriffsdatum: 29.04.2020, Apr. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Learning-Environment-Create-New.md>.

- 
- [19] Nancy Iskander et al. *Issues with Self-Play*. Zugriffsdatum: 06.05.2020, März 2020. URL: <https://github.com/Unity-Technologies/ml-agents/issues/3563>.
  - [20] Tuomas Haarnoja et al. *Soft Actor Critic—Deep Reinforcement Learning with Real-World Robots*. Zugriffsdatum: 02.05.2020, Dez. 2018. URL: <https://bair.berkeley.edu/blog/2018/12/14/sac/>.
  - [21] Vincent-Pierre BERGES et al. *Memory-enhanced agents using Recurrent Neural Networks*. Zugriffsdatum: 14.05.2020, Okt. 2019. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Feature-Memory.md>.
  - [22] Vincent-Pierre Berges et al. *ML-Agents Toolkit Overview*. Pfad in der Dokumentation: ml-agents/docs/ML-Agents-Overview.md, Zugriffsdatum: 21.04.2020, Apr. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/ML-Agents-Overview.md>.
  - [23] andrewcoh. *Training with Self-Play*. Pfad in der Dokumentation: ml-agents/docs/Training-Self-Play.md, Zugriffsdatum: 13.04.2020, 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Self-Play.md>.
  - [24] AurelianTactics. *PPO Hyperparameters and Ranges*. Zugriffsdatum: 01.05.2020, Juli 2018. URL: <https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe>.
  - [25] Bowen Baker u. a. „Emergent Tool Use From Multi-Agent Autocurricula“. In: (Sep. 2019). URL: <http://arxiv.org/pdf/1909.07528v1.pdf>.
  - [26] Trapit Bansal u. a. „Emergent Complexity via Multi-Agent Competition“. In: *CoRR* abs/1710.03748 (2017). arXiv: 1710.03748. URL: <http://arxiv.org/abs/1710.03748>.
  - [27] Andrew Cohen. *AgentSoccer.cs*. Zugriffsdatum: 20.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/release-0.14.1/Project/Assets/ML-Agents/Examples/Soccer/Scripts/SoccerSettings.cs>.
  - [28] Andrew Cohen. *Training intelligent adversaries using self-play with ML-Agents*. Zugriffsdatum: 13.04.2020, 2020. URL: <https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/>.
  - [29] developer.nvidia.com. *Memory-enhanced agents using Recurrent Neural Networks*. Zugriffsdatum: 24.04.2020, URL: <https://developer.nvidia.com/discover/recurrent-neural-network>.

- [30] Chris Elion. *AgentSoccer.cs*. Zugriffsdatum: 20.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/release-0.14.1/Project/Assets/ML-Agents/Examples/Soccer/Scripts/AgentSoccer.cs>.
- [31] Chris Elion. *SoccerFieldArea.cs*. Zugriffsdatum: 25.05.2020, Feb. 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/release-0.14.1/Project/Assets/ML-Agents/Examples/Soccer/Scripts/SoccerFieldArea.cs>.
- [32] Jonathan Harper Ervin Teng Esh Vckay und Yuan Gao. *Unity ML-Agents Toolkit v0.8: Faster training on real games*. Zugriffsdatum: 16.04.2020, Apr. 2019. URL: <https://blogs.unity3d.com/2019/04/15/unity-ml-agents-toolkit-v0-8-faster-training-on-real-games/>.
- [33] Jeremy Graziani. *Reward Signals*. Zugriffsdatum 13.04.2020, Pfad in der Dokumentation: ml-agents/docs/Reward-Signals.md, 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Reward-Signals.md>.
- [34] Klaus Greff u. a. „LSTM: A Search Space Odyssey“. In: *CoRR* abs/1503.04069 (2015). arXiv: 1503.04069. URL: <http://arxiv.org/abs/1503.04069>.
- [35] Tuomas Haarnoja u. a. „Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor“. In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: <http://arxiv.org/abs/1801.01290>.
- [36] Hado van Hasselt, Arthur Guez und David Silver. „Deep Reinforcement Learning with Double Q-learning“. In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: <http://arxiv.org/abs/1509.06461>.
- [37] Jonathan Ho und Stefano Ermon. „Generative Adversarial Imitation Learning“. In: *CoRR* abs/1606.03476 (2016). arXiv: 1606.03476. URL: <http://arxiv.org/abs/1606.03476>.
- [38] Arthur Juliani. *Calculating-Good-Max-Steps-Issue237*. Zugriffsdatum: 06.04.2020, Jan. 2018. URL: <https://github.com/Unity-Technologies/ml-agents/issues/237>.
- [39] Arthur Juliani. *Solving sparse-reward tasks with Curiosity*. Zugriffsdatum: 13.04.2020, 2018. URL: <https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/>.
- [40] Arthur Juliani u. a. „Unity: A General Platform for Intelligent Agents“. In: *CoRR* abs/1809.02627 (2018). arXiv: 1809.02627. URL: <http://arxiv.org/abs/1809.02627>.

- 
- [41] Adam Kelly. *Ray Perception Sensor Component Tutorial*. Zugriffsdatum: 29.04.2020, Dez. 2019. URL: <https://www.immersivelimit.com/tutorials/rayperceptionsensorcomponent-tutorial>.
  - [42] Michael Lanham. *Hands-on deep learning for games: Leverage the power of neural networks and reinforcement learning to build intelligent games*. Hrsg. von test. Birmingham, UK: Packt Publishing, 2019. ISBN: 9781788994071.
  - [43] Artem Oppermann. *Self Learning AI-Agents Part I: Markov Decision Processes*. Zugriffsdatum: 05.04.2020, Okt. 2018. URL: <https://towardsdatascience.com/self-learning-ai-agents-part-i-markov-decision-processes-baf6b8fc4c5f>.
  - [44] Deepak Pathak u. a. „Curiosity-driven Exploration by Self-supervised Prediction“. In: *CoRR* abs/1705.05363 (2017). arXiv: 1705.05363. URL: <http://arxiv.org/abs/1705.05363>.
  - [45] Sebastian Ruder. „An overview of gradient descent optimization algorithms“. In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: <http://arxiv.org/abs/1609.04747>.
  - [46] John Schulman u. a. „Proximal Policy Optimization Algorithms“. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: <http://arxiv.org/abs/1707.06347>.
  - [47] Thomas Simonini. *An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog!* Zugriffsdatum: 24.05.2020, Juli 2018. URL: <https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/>.
  - [48] Richard S. Sutton und Andrew Barto. *Reinforcement learning: An introduction*. Second edition. Adaptive computation and machine learning. Cambridge, MA und London: The MIT Press, 2018. ISBN: 978-0262039246.
  - [49] Ervin T. *Training with Imitation Learning*. Pfad in der Dokumentation: ml-agents/docs/Training-Imitation-Learning.md, Zugriffsdatum: 13.04.2020, 2020. URL: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.1/docs/Training-Imitation-Learning.md>.
  - [50] Couch in the Woods Interactive. *NEON SHIFTER*. Alpha Release: 0.35, Betriebssystem: Microsoft. 2019. URL: <https://couchinthewoods.de/>.



# Anhang

## 1. Experiment 3: Weitere Trainingsdaten

Die folgende Tabelle zeigt die zusätzlichen Trainingsdaten von Experiment 3, die nicht weiter ausgewertet wurden, da keine Vergleichswerte erhoben wurden.

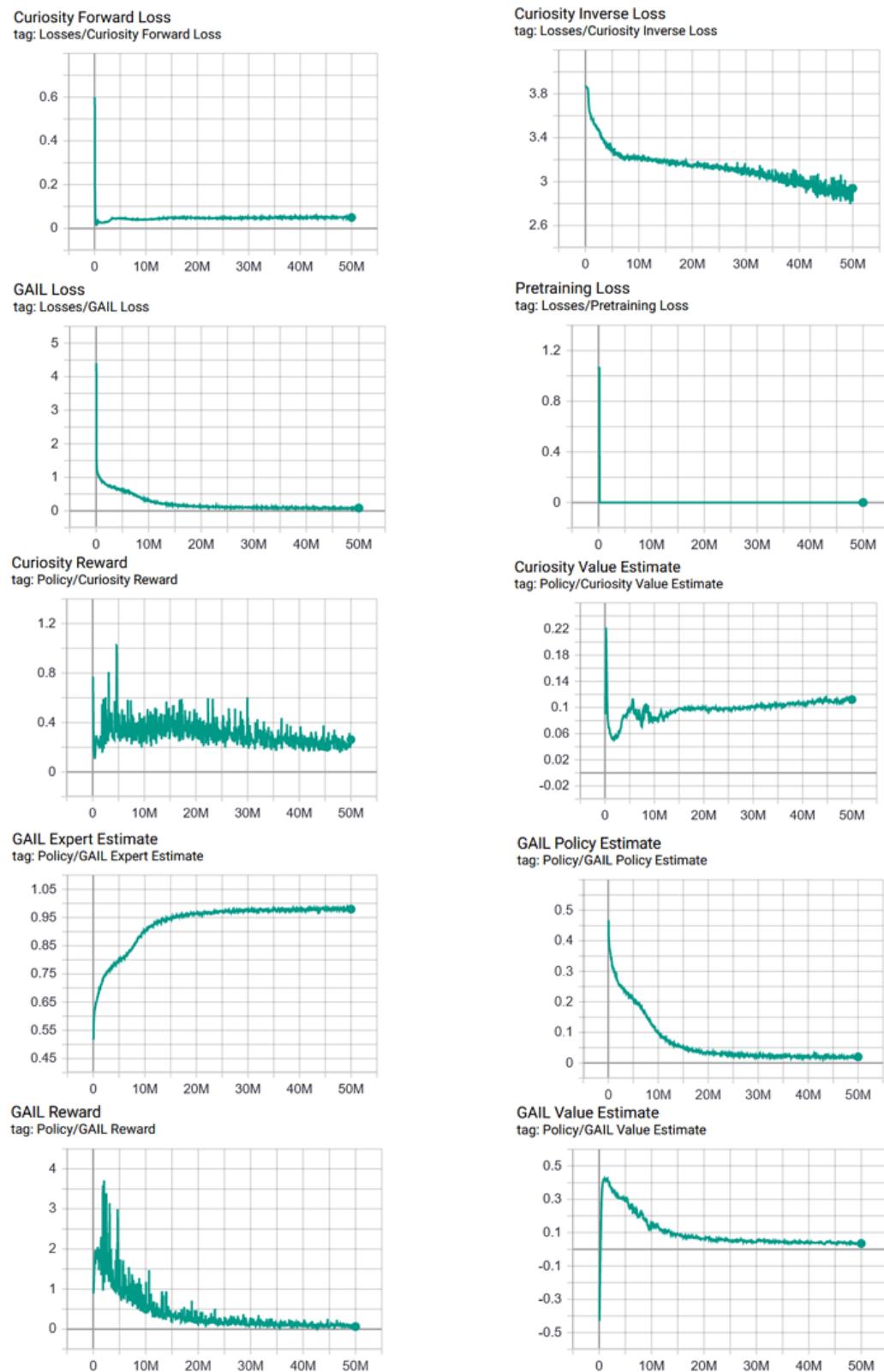


Abbildung 1.: Weitere Ergebnisse des Trainings 3.0 für IL, BC, GAIL und RL.

## 2. SoccerTwos Parameter Screenshots

Die folgenden Abbildungen zeigen Screenshots der Parameter des *SoccerTwos-Environments* im *Unity-Editor*.



Abbildung 2.: Screenshot der *Agent Soccer*-Parameter *SoccerTwos* im Unity Editor

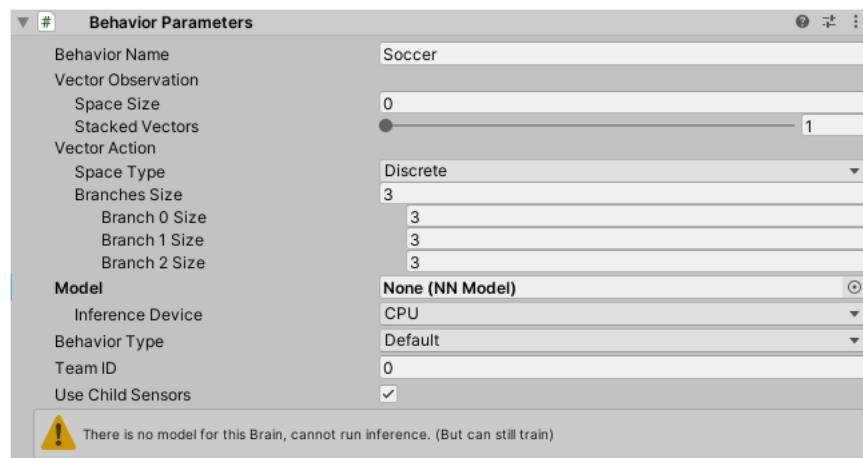


Abbildung 3.: Screenshot der *Behaviour Parameter*-Parameter von *SoccerTwos* im Unity Editor

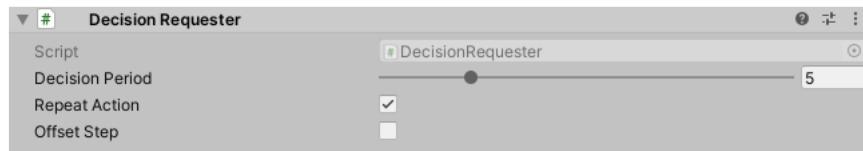


Abbildung 4.: Screenshot der Decision Requester-Parameter von *SoccerTwos* im *Unity Editor*

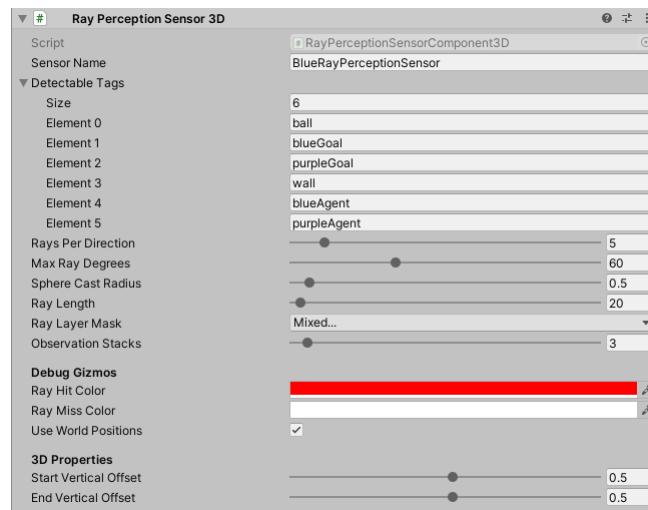


Abbildung 5.: Screenshot der Raycast Sensor Component-Parameter von *SoccerTwos* im *Unity Editor*

### 3. Trainingsinstanzen in SoccerTwos

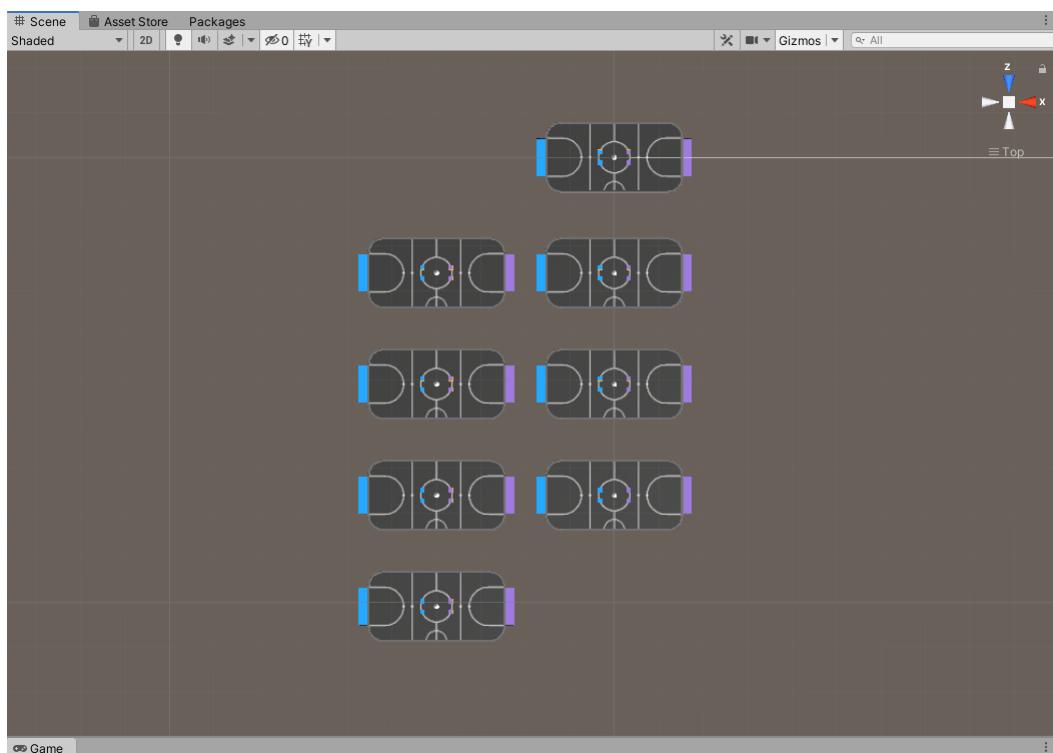


Abbildung 6.: Screenshot der Instanzen der *SoccerTwos-Environment* im Unity Editor

### 4. Screenshot der max steps des Crawler Static-Environment



Abbildung 7.: Screenshot der max\_step-Parameter im Unity Editor des *Crawler-Environment*