Philipp Thoms

Validierung des Einsatzpotenzials von ML-Agents für kompetitive Multiplayer-Spiele

FAKULTÄT FÜR
INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Validierung des Einsatzpotenzials von ML-Agents für kompetitive Multiplayer-Spiele

Bachelor Thesis

Philipp Thoms

13. November 2019

Betreuender Professor: Prof. Dr.-Ing. habil. Sanaz Mostaghim

Betreuer: Enrico Gebert

Philipp Thoms: *Validierung des Einsatzpotenzials von ML-Agents für kompetitive Multiplayer-Spiele*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2019.

Abstrakt

Nortivag ist ein von Silver Seed Games entwickeltes, firmeneigenes, kompetitives, physikbasiertes, lokales Mehrspieler-Actionspiel, dass eine Künstliche Intelligenz (KI) benötigt, gegen die der Spieler antreten kann. Das Ziel dieser Arbeit war es herauszufinden, wie gut sich das ML-Agents Toolkit eignet, diese KI zu implementieren oder ob, wie ursprünglich geplant, eine klassiche Spiele-KI auf Basis von State-Machines entworfen und implementiert werden muss. Um das zu erreichen, soll einem Agenten die Wegfindung innerhalb eines Nortivag-Levels beigebracht werden. Dazu werden zunächst einige Konfigurationstests auf einer einfachen eigens dafür entwickelten Nortivag-Kopie "NLite" durchgeführt, um herauszufinden, welche Einstellungen für den Test in Nortivag sinnvoll sind, um dort eine KI mit den optimalen Voraussetzungen zu lernen. Mit den Experimenten in NLite wird auch getestet, ob sich ML-Agents für das Spielprinzip von Nortivag eignet. Nach den Konfigurationstests werden diese Ergebnisse innerhalb einiger Leveltests auf verschiedenen Karten validiert, um anschließend eine geeignete Konfiguration für den finalen Nortivag-Test zu finden. In diesem letzten Test wird entschieden, wie gut die AI innerhalb der aktuellen Version des Hauptspiels lernen kann und inwiefern es Sinn macht, den lernenden KI-Ansatz weiter für Nortivag zu verfolgen. Das Ergebnis der NLite-Experimente war sehr vielversprechend für die Funktionstüchtigkeit des Plugin. Es wurde eine sehr gute KI erschaffen, die sich gut durch zwei der drei vordefinierten Level bewegen konnte, wobei der schweste Level diente, die Grenzen der Wegfindung des gelernten Agenten zu testen. Diese Ergebnisse konnten aber nicht im Nortivag-Test bestätigt werden, denn dort konnte der Agent innerhalb des Testrahmens nicht ansatzweise die Performance der NLite-Experimente erreichen. Demnach kann die Ausgangsfrage mit "Ja" beantwortet werden. Innerhalb Nortivags ließ sich dieses Ergebnis nicht bestätigen, aufgrund des Entwicklungsstandes mit vielen Bugs und Performanceproblemen, deren Auswirkung auf den Lernvorgang unbekannt sind. In der Theorie ist der Reinforcement-Learning-Algorithmus, der in ML-Agents benutzt wird, aber in der Lage, Spiele in dieser Komplexität lernen zu können.

Inhaltsverzeichnis

Abbildungsverzeichnis **V**

Tabellenverzeichnis **VII**

Glossar **IX**

1 Einführung **1**
1.1 Motivation 2
1.2 Ziele 2
1.3 Struktur 3

2 Grundlagen **5**
2.1 Künstliche Intelligenz in Spielen 5
2.2 Stand der Wissenschaft 7
2.2.1 Regelbasierte Architekturen 8
2.2.2 Reinforcement Learning 9
2.2.3 Proximal Policy Optimization 11
2.2.4 Beispiele 15
2.3 ML-Agents 19

3 Nortivag **25**
3.1 Das Spiel 25
3.1.1 Spielablauf 25
3.1.2 Features 26
3.1.3 Steuerung 28
3.2 Interface 28
3.3 Die Simulation Nortivag Lite 29

4 ML-Agents in Nortivag 33
4.1 Ausgangszustand und Vorbereitung 33
4.2 Implementierung des Plugins 34
4.3 KI entwickeln für Nortivag und NLite 35
4.3.1 Ansätze 35
4.3.2 Vergleich und Hypothesen 40

5 Evaluierung 45
5.1 Konfigurationstests in Nortivag Lite 45
5.1.1 Vorbereitung 45
5.1.2 Durchführung 46
5.1.3 Auswertung 47
5.2 Leveltests in Nortivag Lite 58
5.2.1 Vorbereitung 58
5.2.2 Durchführung 58
5.2.3 Auswertung 60
5.3 Finaler Test in Nortivag 68
5.3.1 Vorbereitung 68
5.3.2 Durchführung 69
5.3.3 Auswertung 69
5.4 Gesamtauswertung 72

6 Zusammenfassung und Ausblick 75

Literaturverzeichnis 77

Abbildungsverzeichnis

2.1 Weltmarkt Umsatzprognose 5
2.2 PPO in Pseudocode 11
2.3 Clipping in PPO 13
2.4 Vergleich der RL-Algorythmen 14
2.5 Vergleich 1v1 vs 5v5 Bots 16
2.6 MMR-Gewinn der OpenAI-Five KIs 17
2.7 Zusammenfassung der Pac-Man-Geist-KI im Jagen-Modus . . . 18
2.8 Bestandteile in ML-Agents 20
2.9 Anacondapromt beim Lernen 21

3.1 Nortivag Gameplay 26
3.2 Nortivag Gameplay auf der BlackHole-Karte 27

4.1 Ausgangszustand der Implementation von ML-Agents in Nortivag 33
4.2 Implementierung von ML-Agents in Nortivag 34
4.3 NLite-Level 37
4.4 In den Experimenten genutzte Curriculum-Konfiguration 38
4.5 Standard-Konfigurationsdatei mit abgeänderter maximaler Schrittanzahl 40

5.1 Experiment 1 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 48
5.2 Experiment 2 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 49
5.3 Experiment 2, die ersten $5 \cdot 10^5$ Iterationen Result: Cumulative Reward Nr. of Runs 50

5.4 Experiment 2b Ergebnis: Mittlere Reward über die Anzahl der Iterationen 52
5.5 Experiment 2b, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 53
5.6 Experiment 3 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 54
5.7 Experiment 3, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 55
5.8 Experiment 4 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 56
5.9 Experiment 4, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 57
5.10 Konfigurationen der einzelnen Levelzustände 59
5.11 Experiment 5 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 61
5.12 Experiment 5, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 62
5.13 Experiment 6 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 63
5.14 Experiment 6, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 64
5.15 Experiment 7 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 66
5.16 Experiment 7, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 66
5.17 Die einzelnen Curriculum-Konfigurationen des Nortivag-Level, der für das achte Experiment benutzt wurde. 68
5.18 Experiment 8 Ergebnis: Mittlere Reward über die Anzahl der Iterationen 70
5.19 Experiment 8, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen 71

Tabellenverzeichnis

0.1 Abkürzungsübersicht IX
0.2 Begriffserklärung X

2.1 Komplexitätsanforderungen die eine Simulation an eine KI stellen können muss 8

3.1 Unterschied Nortivag zu NLite 31

4.1 Erklärung der Curricula-Datei aus Abbildung 4.4 38
4.2 Konfigurationsexperimente und Hypothesen 42
4.3 Levelexperimente und Hypothesen 43

5.1 Ergebnisse der Konfigurationstests 47
5.2 Ergebnisse der Leveltests 60
5.3 Ergebnisse des Nortivag-Tests 69

Glossar

Tabelle 0.1: Abkürzungsübersicht

Abkürzung	Begriff
KI	Künstliche Intelligenz
AI	Artificial intelligence
RL	Reinforcment Learning (Bestärkendes Lernen)
ML	Maschinelles Lernen
NLite	Nortivag Lite (ein vereinfachter Nortivag Klon)
PPO	Proximal Policy Optimization

Tabelle 0.2: Begriffserklärung

Begriff	Erklärung
ML-Agents	Ein Unity-Plugin, das die Möglichkeit bietet, eine KI in seinem Projekt zu trainieren
Koop-Spiel	Ein Spiel das mit mehreren Personen meist an einem/r PC/Konsole gespielt werden kann
Kompetitiv	Wettbewerbsfähig
Unity	Eine 3D Game-Engine
Tensorflow	Ist eine Open Source Plattform für Maschinelles Lernen
Content-Creation	Das Erstellen von Inhalten
Bot	Durch KI gesteuerte Spielfigur
Gameplay	Ist die Art und Weise, wie der Spieler das Spiel spielt
Statemachine	Gängiger Ansatz für KI-Implementierung
Szene	Der Rahmen, in dem die Umwelt und die Menüs in Unity abgebildet sind
GameObjekt	Ein Objekt in einer Unity-Szene
Prefab	Ein Verbund von GameObjekten und Skripten
Spawning	Das Erscheinen eines Objektes unter festgelegten Regeln

1 Einführung

Schon lange ist künstliche Intelligenz nicht mehr aus Videospielen wegzudenken. Innerhalb des Spiels sorgt sie für große Immersion und eine glaubhafte Welt. Dies erreicht sie, je nach Aufgabe für die sie eingesetzt wird, wenn sie realistisch handelt und dem Spieler menschliches Verhalten vorspielt oder wenn sie innerhalb prozeduraler Content-Creation Level oder Objekte erstellt, die sich logisch in den Rahmen und die Designidee des Spiels einfügen. Einige Unternehmen benutzen KI, um Realweltprobleme zu lösen und testen dazu ihre KI in Simulationen [14]. Darunter fällt auch, ein komplexes Videospiel so gut zu spielen, dass die Bots besser als jeder menschliche Kontrahent sind [7]. Jedoch kann eine schlechte KI ein Spiel sehr stark negativ beeinflussen, wenn sie unlogisch handelt, grobe Fehler macht oder ganz einfach zu leicht auszuhebeln ist [17]. Zusätzlich ist es selbst mit den vorhandenen Mitteln nicht immer einfach, einen voll funktionsfähigen Bot für ein Spiel zu erschaffen. Deswegen verzichten aktuelle kleine und große Koop-Spiele auf KI oder legen einen sehr kleinen Fokus auf sie [33], weil sich der sehr große Aufwand nicht lohnt. Um eine zu leichte KI schwerer zu machen, wird das Spiel künstlich entweder durch schiere Massen an Gegnern oder durch unfaire Vorteile dieser beeinflusst. Silver Seed Games befindet sich in der Entwicklungsphase von Nortivag, einem kompetetiven physikbasierten lokalen Mehrspieler-Actionspiel, für das langfristig eine KI erstellt werden soll, die dem Spieler eine Herausforderung bietet. Das Spiel besitzt eine komplexe Steuerung und Spieltiefe, die mit dem Statemachine-Ansatz zur KI Erstellung nur schwer umzusetzen ist. Deshalb wird nach Möglichkeiten gesucht, einen künstlichen Gegner mit verhältnismäßig geringerem Aufwand zu erschaffen. Diese Arbeit beschäftigt sich damit, ob ML-Agents, ein Unity-Plugin, das dem Nutzer ermöglicht, eine KI in seinem Spiel zu trainieren, geeignet ist, in Nortivag eine KI zu erstellen und dabei die genannte Anforderung zu erfüllen. Dazu wird getestet, wie gut die KI die Wegfindung auf mehreren Spielfeldern mit unterschiedlichen Schwierigkeitsgraden meistern kann. Aus diesen Ergebnissen kann geschlossen werden, ob sich

das Plugin eignet und weiter verwendet werden kann oder die herkömmlichen Methoden der KI-Erstellung benutzt werden müssen.

1.1 Motivation

In Nortivag ist die KI ein wesentlicher Bestandteil der Einzel- und Mehrspielerinhalte. In beiden Formaten tritt sie gegen den Spieler als Feind an und zusätzlich können im letzteren fehlende menschliche Mitspieler durch sie ersetzt werden. Je nach Schwierigkeitsgrad der KI muss sie den Spieler fordern wie es ein menschlicher Gegner tun würde, damit das Gameplay funktionieren kann. Das kann nur von der KI erreicht werden, wenn sie viele Probleme gleichzeitig und in Echtzeit lösen kann, um sich in den unterschiedlichsten Situation, die während eines Spiels auftreten, behaupten zu können. In der aktuellen Vorgehensweise der Videospielindustrie wird KI meist nur in Form von Statemachines implementiert, da es große Risiken in der Anwendung maschinell gelernter KI's gibt [30]. Diese Arbeit soll als Test des ML-Ansatzes dienen, auch wenn die Aufgabe der KI vergleichsweise klein ist.

1.2 Ziele

Langfristig soll für Nortivag eine voll funktionsfähige und spielstarke KI entstehen. Die Ergebnisse dieser Arbeit legen den Grundstein dafür. Es wird getestet, ob ML-Agents ein geeignetes Plugin ist, um gute KI zu erstellen oder ob eine manuelle Lösung mit den älteren Ansätzen präferiert werden sollte. Nach erfolgreicher Implementation soll das Plugin einer KI beibringen, leichte Bewegungsaufgaben zu lösen. Vorher wird getestet, ob Aufgaben dieser Art überhaupt von ML-Agents lösbar sind. Dazu wird in einem neuen Unityprojekt eine Kopie von Nortivag erstellt, die später als Nortivag Lite bezeichnet wird. Diese enthält lediglich das Notwendigste, um das Grundprinzip der Lernaufgaben nachzustellen. Wenn die Tests erfolgreich sein sollten, können andere Entwickler über eine standardisierte Schnittstelle einfach neue KIs für Nortivag erstellen.

1.3 Struktur

Um die gestellten Zielanforderung zu erfüllen, wird als erstes im Grundlagen-Kapitel geklärt, was ML-Agents ist und wie es funktioniert. Vorher gibt es eine Einführung in den aktuellen Stand der KI Entwicklung in Videospielen und welche Ansätze dafür benutzt werden. Daraufhin wird kurz das Spiel Nortivag erklärt, in dem das Plugin zum Einsatz kommen soll. Dabei wird auf die notwendigsten Features und Inhalte eingegangen, die für die KI-Entwicklung relevant sind. Mit dem Wissen aus diesen beiden Kapiteln kann schließlich im vierten Teil dieser Arbeit die Implementation des Plugins erläutert werden. Im gleichen Abschnitt wird auch beschrieben, welche Ansätze es gibt, um eine gute KI zu erstellen und welche Experimente dafür gemacht werden müssen. Zu den Experimenten werden Hypothesen aufgestellt, die im nächsten Teil getestet werden. In diesem Evaluationsteil werden alle drei großen Experimentblöcke vorbereitet, durchgeführt und ausgewertet und in einer Gesamtauswertung zusammengefasst. Im letzten Kapitel kann damit eine Antwort auf die Frage, ob ML-Agents ein geeignetes Plugin für Nortivag ist, gegeben werden. Darüber hinaus wird ein Ausblick gegeben, wie die Ergebnisse dieser Arbeit noch weiter verwendet werden können.

2 Grundlagen

2.1 Künstliche Intelligenz in Spielen

Je nach Interpretation gibt es verschiedene Anwärter auf den Titel "Erstes Videospiel" [21]. Ob Pong, Tennis for Two oder gleich OXO, ein digitales Tic-Tac-Toe-Spiel von 1952, diese Spiele enthielten alle keine KI [21] und konnten nur mit mehreren Spielern gespielt werden. Bis zum Ende der 1990er Jahre spielte mit Ausnahme von Schach und Wirtschaftssimulationen KI keine große Rolle [20]. Mit dem Fortschreiten der Technik änderte sich das. Half-Life und andere Ego-Shooter spielten dabei eine große Rolle und die Industrie erkannte, dass KI eine enorm wichtige Instanz ist, die Welt und seine Charaktere glaubwürdig aussehen zu lassen [20]. Dazu sagte Alan Kertz, Lead Gameplay Designer bei EAs Dice Studio: „Good AI (Artificial Intelligence) pulls the player into the game and improves the experience."

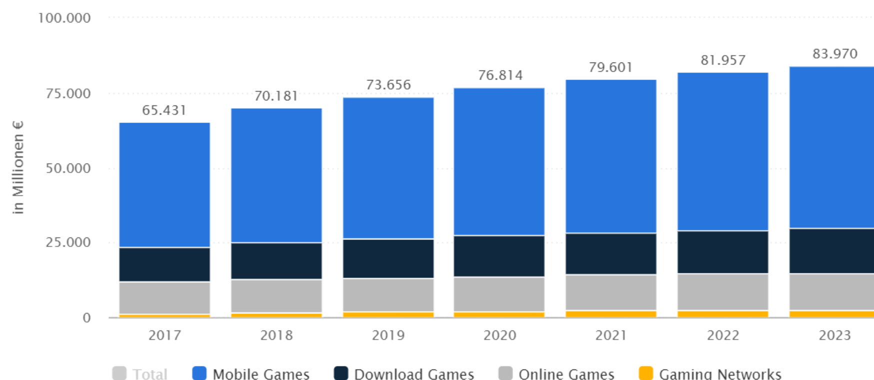

Abbildung 2.1: Weltmarkt Umsatzprognose
In der Grafik ist eine Umsatzprognose des Videospiel-Weltmarkts abgebildet, bei dem ein starker Anstieg innerhalb der nächsten 4 Jahre erkennbar ist.
Quelle: [29]

Damit die KI in ihren ersten Jahren ein herausfordernder Gegner sein konnte, schummelte sie oftmals. So hatte sie gegenüber dem menschlichen Spieler oft unfaire Vorteile, indem sie beispielsweise an unerreichbare Ressourcen herankam [20], obwohl dies durch bessere Technik und Algorithmen teilweise nicht mehr notwendig war. Durch das Wachstum der Industrie, dargestellt in Abbildung 2.1, erweiterte sich das Spielspektrum immer mehr. Zusätzlich zur Unterhaltung gab es Lern-, Therapie-, Trainings-, und Simulationsspiele. Dementsprechend vergrößerte sich das Einsatzgebiet von KI in Spielen stetig. Ein Bot hat in einem Ego-Shooter ganz andere Herausforderungen und Aufgaben als in Pac-Man oder der KI in No Man's Sky, welche ein ganzes Universum generiert [26] hat. Einige der Teilbereiche der KI sind [34][20]:

- Wegfindung: Das Entdecken von sinnvollen Routen für die Navigation von Spielcharakteren in der virutellen Umgebung.
- Steuerung: Die sinnvolle Nutzung der Aktionsmöglichkeiten der Spielfigur
- Verhalten: Den Ablauf zusammenhängender Aktionen planen
- Content-Creation: um automatisiert neue Inhalte zu generieren
- Glaubwürdigkeit: um ein realistisches Verhalten nachzuahmen

Trotz der momentanen Möglichkeiten von ML-Algorithmen wird in der Industrie größtenteils weiterhin der ältere Ansatz der Statemachines benutzt [30]. Das liegt daran, dass die vom Entwickler an die KI gestellten Ziele bereits jetzt zufriedenstellend gelöst werden [30]. Selbst sehr komplexe Ziele sind mit diesen älteren Ansätzen lösbar, wie zum Beispiel der extreme Schwierigkeitsgrad der Endgegner im Action-RPG Dark Souls [30], der riesigen und lebhaften Welt von Red Dead Redemtion [30] oder einfach den prozedural erstellten Leveln im Spiel Dead Cells [10]. Mit einem Konstrukt aus sehr vielen verschiedenen Statemachines lässt sich eine Welt schaffen die glaubhaft ist aber immer noch klare Grenzen besitzt [30]. Diese sind enorm wichtig für Gamedesigner, denn nur so können sie garantiert ein in sich geschlossenes Spiel erschaffen. Bei selbst lernender KI können unvorhersehbare Ergebnisse entstehen, die das Spielgefühl stark negativ beeinträchtigen können [30]. Unschlagbare Gegner, wie am Beispiel von Dota 2 [7] noch erklärt wird, oder Level die nicht den Wünschen und Vorstellungen des Designers entsprechen sind nur einige der Dinge die passieren können, weshalb in der Videospielindustrie noch immer die alten standardisierten Ansätze benutzt werden [30], die zusätzlich viel einfacher zu im-

plementieren sind. Außerdem ist für das Lernen einer KI für größere Spiele eine enorme Menge an Rechenleistung nötig, die nicht immer zur Verfügung steht. Trotzdem versucht diese Arbeit einen Schritt in Richtung KI-Implementierung mithilfe maschinellen Lernens zu gehen, um bessere Leistungen mit weniger Programmieraufwand zu erzielen.

2.2 Stand der Wissenschaft

Videospiele stellen den Menschen vor verschiedenste Herausforderungen und sind deshalb ein idealer Startpunkt, um KI-Algorithmen zu testen, bevor sie in der Realwelt Anwendung finden [14]. Innerhalb der letzten Jahre gab es große Fortschritte im Bereich Deep Reinforcement Learning [14]. Wesentlich verantwortlich dafür sind gute Simulationsplattformen mit denen die Algorithmen getestet werden können [14]. Beispielsweise war die Arcade Learning Environment (ALE) eine Simulationsplattform, essenziell als Benchmark für den Control-from-Pixel-Ansatz im Deep Q-Network [14]. Mit der Zeit wurden die Algorithmen soweit verbessert, bis sie die Herausforderungen der Simulationen mit übermenschlicher Performance lösen konnten und damit alle Umgebungen in ALE als gelöst galten [14][19]. Des wegen wurde ALE als Simulationsbenchmark weniger wertvoll und es mussten neue Simulationen entworfen werden. Dadurch entstand eine Spirale, mit der sich Simulationen und Algorithmen immer gleichzeitig verbessern mussten [14]. Viele dieser Simulationen basieren auf aktuellen Videospielen [28]. Minecraft, Quake 3, Doom sind nur einige dieser Spiele. Jedes von ihnen bietet einen anderen Schwerpunkt, und macht sie in ihrem Bereich für Simulationen bedeutsam [14]. Für die KI-Entwicklung sind Spiele erst bei einer großen Komplexität und Übertragbarkeit in die Realwelt relevant [14]. Deshalb muss eine gute Simulation in der Lage sein, diese verschiedenen Komplexitätsbereiche abzudecken, um den Anforderungen an die Realwelt gerecht zu werden [14]. Diese Komplexität besteht aus vier Kernbereichen, abgebildet in Tabelle 2,1, die notwendig sind, um die Algorithmen zu fordern [14]

Tabelle 2.1: Komplexitätsanforderungen die eine Simulation an eine KI stellen können muss

Teilbereich	Beschreibung
Sensorisch	große Mengen an visuellen, auditiven und textbasierten Daten die der KI übertragen werden
Physisch	großen Kontrollmöglichkeiten des eigenen Agenten
Kognitiv oder Kombinatorisch	große Mengen an Handlungsmöglichkeiten innerhalb eines riesigen Suchraums
Sozial	Agent muss Aufgaben innerhalb einer Gruppe bewältigen, entweder durch Interaktion mit ihr oder durch Lösen kleinerer notwendiger Aufgaben, die sich innerhalb der Agenten zu einer Großen formen.

In der Tabelle werden die Kernbereiche genannt und beschrieben, inwiefern die einzelnen Anteile eine Herausforderungen an die KI stellen können.

2.2.1 Regelbasierte Architekturen

Ein Weg, KI in Spielen zu implementieren, sind regelbasierte Architekturen. Diese einfache Art der KI war eine der ersten Lösungen für Bots und andere Probleme in Videospielen und ist bis heute in weiter entwickelten Formen immer noch die herkömmliche Herangehensweise [30]. Diese Architekturen bestehen immer aus zwei Kernkomponenten [12]. Die Erste ist das Set der Fakten, welches den Momentanzustand des Spiels und seine Situation beschreibt [12]. Aus diesen Daten können verschiedene Informationen wie Geschwindigkeit und Position abgelesen werden. Mit dem Set der Regeln, welches die andere Kernkomponente bildet, wird festgelegt, welche reaktive Handlung als Konsequenz auf den momentanen Zustand ausgeführt werden darf [12]. Wenn dazu die menschliche Erfahrung und das Wissen dazukommen, wird dabei von "Expert Based Rules" gesprochen [12]. Die Vorteile dieses Ansatzes sind die einfache Implementierung, das einfache Management und hohe Stabilität [12]. Das Set der Regeln, welches auf dem Prinzip der If-Then-Strukturen basiert, beschreibt dabei die genaue Handlungsweise der KI [2]. Der Nachteil bei diesen Regeln ist allerdings, dass bei größeren Systemen die Menge an benötigten Regeln extrem ansteigt, um alle Einzelfälle abzudecken [2]. Hinzu kommt, dass selbst kleinste

Änderungen im Szenario eine Überarbeitung der Regeln erfordern, was je nach System ein sehr großer Aufwand sein kann [2].

2.2.2 Reinforcement Learning

Maschinelles Lernen

Im Gegensatz zu den regelbasierten Architekturen arbeitet ML mit Wahrscheinlichkeiten und einem statistischen Modell [12]. Das Ergebnis wird aus der Kombination verschiedener Eingabeparameter gebildet und muss erst anhand Trainingsdatensätzen gelernt werden [12]. Diese Daten sind jedoch nicht immer vorhanden oder gut genug, da die Datensätze schnell veralten können [25]. Deshalb kann es schwierig sein, das Modell genau auf das gewünschte Problem anzupassen. Nachdem der Lernvorgang abgeschlossen ist, ist es sehr schwer nachzuvollziehen, warum das Modell spezifische Aktionen durchführt [12]. Diese falschen oder unvorhersehbaren Entscheidungen können bei kritischen Aufgaben zu mitunter fatalen Fehlern führen [2]. Des wegen kann es sehr schwer sein herauszufinden, welches Verhalten für die Handlung innerhalb des Systems, das sich in einer Black-Box befindet, verantwortlich war [12] [2]. Mit immer besser werdenden Algorithmen werden diese Blackbox-Systeme zusätzlich immer unverständlicher. Allerdings ist es nicht Teil dieser Arbeit darzustellen, wie genau der Algorithmus innerhalb des Spiels in der Blackbox abläuft. Durch das automatisierte Lernen ist es allerdings möglich, sehr gute Ergebnisse mit relativ wenig Aufwand (innerhalb von ML-Agents) zu erschaffen. Zusätzlich dazu kann das Modell immer weiter gelernt werden, weswegen sich die Ergebnisse immer weiter verbessern können, was im Rulebased-Ansatz nicht möglich ist.

Reinforcement Learning

Eine spezielle Form des maschinellen Lernens ist das Reinforcement Learning. Diese Algorithmen enthalten immer einen Testrahmen, in deren Grenzen der Lernvorgang stattfindet [35]. Innerhalb dieser Umgebung befindet sich mindestens ein Agent, welcher mit ihr interagiert [35]. Er ist die handelnde Instanz, die durch Entscheidungen anhand der Regeln progressiv lernt [35]. Dabei wird beim Reinforcement Learning sehr stark auf die sequenziellen Daten geachtet,

indem bei der aktuellen Eingabe immer eine Abhängigkeit zur vorangegangenen besteht [35]. Das gleiche gilt für die Aktionen des Agenten, denn diese hängen auch von ihrer Handlung im letzten Schritt ab [35]. Da es keinen Supervisor gibt [35] und damit die optimalen Handlungen zunächst unbekannt sind, muss das System sie selbst durch eigenes Ausprobieren lernen. Die gewählten Handlungen hängen von einem neuronalen Netz ab, welches Policy genannt wird und wahrscheinlichkeitsbasiert festlegt, was der sinnvollste nächste Spielzug in Abhängigkeit vom aktuellen Spielzustand ist [35]. Dieses Mapping wird über die Zeit des Lernvorgangs immer weiter verbessert. Damit die Änderungen unter Beachtung des Rulesets zu einem positiven Ergebnis führen, gibt es Belohnungen, später Rewards genannt, welche dem Agenten in Abhängigkeit von seinen Handlungen gegeben werden. Diese hängen davon ab, wie zielführend seine Aktion war und können bei Versagen auch negativ ausfallen [19]. Ziel des Agenten ist es, die erhaltenen Rewards zu maximieren, wodurch ein wünschenswertes Verhalten antrainiert wird.
Im Allgemeinen beeinflusst der Agent im festgelegten Rahmen die Umgebung [35]. Wie dieser Zustand aussieht, hängt von den Aktion ab, die er durchführt. Demnach ist das Geschehen zum Zeitpunkt abhängig vom vorangegangenen Zeitpunkt und allen Zeitpunkten davor. In dem Spezialfall, dass jeder Punkt nur nur noch abhängig von seinem direkten Vorgänger ist, erfüllt das System die Markov Property [35] und unter diesen Umständen kann das Reinforcement Learning als Marcov Decision Process (MDP) bezeichnet werden [35]. Denn hier hängt der Zustand der Umwelt nur vom seinem Vorangegangen und der jeweiligen Aktion ab und ist damit unabhängig von weiter zurückliegenden Zuständen und Aktionen [35]. Mithilfe von berechenbaren Transition Probabilities können dann die verschiedenen Zustände vorhergesagt werden [35]. Um den besten Zustand zu erhalten, können für die Policy sogenannte Action-Value Functions benutzt werden [35]. Hier wird berechnet, welcher Reward mit einer bestimmten Aktion erwartet werden kann [35]. Dadurch kann die Policy bei guten Value Functions schnell die Entscheidungen mit dem positivsten Ergebnis treffen und damit das Lernen erfolgreich beenden. Trotzdem hat auch Reinforcement Learning das Problem, dass die Trainingsdaten abhängig von der aktuellen Policy sind, weil diese Daten generiert werden, indem der Agent basierend auf seiner Policy in der Umwelt agiert [16]. So ändert sich die Datenverteilung über den Observations und Rewards ständig innerhalb des Lernvorgangs, was ihn sehr instabil macht. Des Weiteren sind die hohe Empfindlichkeit und genaues Parameter-Feintuning weitere Probleme [32].

2.2.3 Proximal Policy Optimization

Um die Probleme von Reinforcement-Learning aus 2.2.2 zu umgehen, entwickelte OpenAI den Algorithmus PPO mit den Zielen, eine möglichst einfache Implementierung, hohe Effizienz und einfaches Feintuning zu bieten [13].

Algorithm 1 PPO, Actor-Critic Style

```
for iteration=1,2,... do
    for actor=1,2,...,N do
        Run policy π_θold in environment for T timesteps
        Compute advantage estimates Â_1,...,Â_T
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ NT
    θ_old ← θ
end for
```

Abbildung 2.2: PPO in Pseudocode
Quelle: [28]

Im ersten Teil des Algorithmus aus Abbildung 2.2 wird für eine Zeit T die vorhandene Policy ausgeführt und generiert dabei Sequenzen von Erfahrungen, mit denen die Advantagefunction A_t berechnet wird. Sobald genug Erfahrungen gesammelt wurden, um die Policy zu aktualisieren, wird der zweite Teil des Algorithmus ausgeführt. Dabei wird mit der gesammelten Erfahrung eine Gradient-Descent-Methode auf der Policy ausgeführt. Da PPO eine Policy Gradient Methode ist, die online lernt, kann sie erhaltene Informationen nicht wie andere anerkannte Ansätze, wie DQN [32], für die spätere Verwendung speichern. Innerhalb der vanilla Policy-Gradient-Methoden wird der Erfahrungs-Verlust mit Formel 2.1 dargestellt [28].

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t \mid s_t)\hat{A}_t] \tag{2.1}$$

- A_t: Schätzung des Mehrwerts mit der Zeit t
- θ: Policy Parameter
- $\hat{E}_t$: empirische Erwartung mit der Zeit t
- a_t: Aktion am Zeitpunkt t
- s: State der Umgebung

Die Advantagefunction A_t ist dabei eine Schätzung des Mehrwerts, den die gewählte Aktion zum Zeitpunkt t hat [13]. Um A_t zu berechnen, werden die

Discounted Rewards benötigt [32], eine gewichtete Summe an Rewards, die der Agent in der aktuellen Episode bekommen hat. Die Gewichtung wird so gewählt , dass Rewards, die früh erlangt werden können, besser sind [28]. In PPO wird die Advantagefunction erst berechnet, wenn die Episode durchgeführt wurde, wodurch alle Rewards bekannt sind und diese nicht geschätzt werden müssen [28]. Der zweite notwendige Teil ist die Valuefunction, welche die Discounted Rewards der zukünftigen Episoden schätzt und damit auch das Endergebnis der finalen Episode [28]. Dieser Teil wird ständig durch die Erfahrungen des Agenten aktualisiert [28]. Wenn dieser Teil vom ersten subtrahiert wird, resultiert daraus A_t und auch eine Antwort auf die Frage ob das Ergebnis der Aktion des Agenten besser oder schlechter als erwartet war [32]. Dies führt dann dazu, dass die jeweiligen Wahrscheinlichkeiten innerhalb der Policy, welche die Aktion bestimmen, sich verändern [32]. Wenn diese Berechnung immer auf nur einer Erfahrung basiert kann es passieren, dass die Parameter falsch geupdatet werden und somit die eigene Policy zerstört wird [13]. Damit dies nicht passiert, wird der Trust-Region-Policy-Optimization (TRPO) Ansatz innerhalb von PPO verwendet [28]. Dieser besagt, dass die Policy sich nicht zu stark pro Episode verändern darf. Die hierfür benutzte Gradient Loss Function ist [28]:

$$\underset{\theta}{maximze} \quad \hat{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t \right] \tag{2.2}$$

Im Vergleich zum Standardansatz wird nur der Logarithmus mit der Division durch die alte Policy ausgetauscht. Damit die Policy nicht zu weit von der Alten abweichen kann, wird ein Kullback-Leibler-Constraint hinzugefügt [27] [13]. Dieser erzeugt allerdings zusätzlichen Overhead, was zu unerwünschtem Trainingsverhalten führen kann [15]. PPO löst dieses Problem, indem sie den KL-Constraint in das Optimierungsproblem inkludiert [13]. Um das zu erreichen, wird zunächst ein Verhältnis r_t zwischen der alten und der neuen Policy definiert [28]. Das Ziel ist es, dieses r_t zu maximieren [28]. Bei gegebenen Wahrscheinlichkeiten und Aktionen ist r_t größer Eins, wenn die Wahrscheinlichkeit für eine spezifische Aktion höher ist und zwischen Null und Eins wenn sie kleiner ist als im letzten Schritt [28]. Multipliziert mit der Advantagefunction ist das Ergebnis auch wieder die normale TRPO Funktion. Aus diesem r_t resultiert die PPO Funktion 2.3 [28].

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)\hat{A}_t)] \tag{2.3}$$

- r_t: ist Verhältnis der neuen zur alten Policy
- ε: ist ein Hyperparameter

Die Zielfunktion, die in diesem Teil der finalen PPO Funktion optimiert wird, ist eine Erwartungsfunktion, die sich aus dem Minimum von zwei Teilen bildet [28]. Der erste Teil ist die normale und der zweite eine mittels clipping mit einem kleinen Epsilon abgestumpfte Version der TRPO Function [28]. Je nach Advantagefunction ändert sich das Verhalten der Optimierung [28].

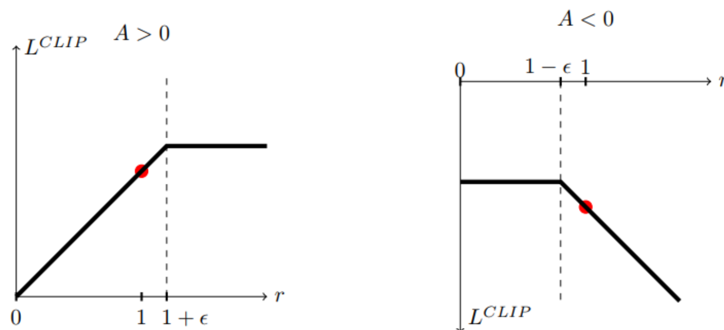

Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

Abbildung 2.3: Clipping in PPO
Quelle: [28]

In Abbildung 2.3 ist der Effekt bei positiver und negativer Advantagefunction abgebildet [28]. Der rote Punkt stellt dabei den Anfang der Optimierung dar. So wird bei zu hohen A die Funktion gedämpft, wenn die Wahrscheinlichkeit für eine Aktion viel höher ist als bei der alten Policy, um eine zu starke Abweichung der neuen von der alten Policy zu verhindern. Das gleiche wird bei zu kleinem A durchgeführt, um zu verhindern, dass die Wahrscheinlichkeiten, dass einzelne Aktionen eintreten, zu schnell sinken. Wenn die durchgeführte Aktion schlecht war und die Policy dieser Aktion trotzdem eine höhere Wahrscheinlichkeit bekommt, wie auf der rechten Seite von Abbildung 2.3 zu sehen ist, dann kann das Update in PPO rückgängig gemacht werden. Dies ist der einzige Fall innerhalb der Funktion 2.3, bei der der linke ungedämpfte Teil kleiner ist als der Rechte und wegen des Minimumoperators wird der Linke zurück gegeben

[28]. Die finale Version zur Berechnung des Erfahrungsverlustes bildet sich wie in Funktion 2.4 dargestellt [28].

$$L_t^{PPO}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) A_t + c_2 S[\pi_\theta](s_t)] \tag{2.4}$$

Der erste Teil ist die bereits erwähnte Clipping Berechnung. Der Zweite ist für die Aktualisierung des Basisnetzes zuständig [28]. Dieses erzeugt die Discounted Rewards, die zur Berechnung der Advantagefunction benötigt werden [28]. Der letzte Part ist der Entropieteil, welcher dafür sorgt, dass der Agent innerhalb des Trainings genug erkundet [28]. Die Parameter c_1 und c_2 dienen hier als Gewichtung, um mit der Zeit das Erkunden weniger werden zu lassen, wenn sich eine stabile Policy entwickelt hat [32].

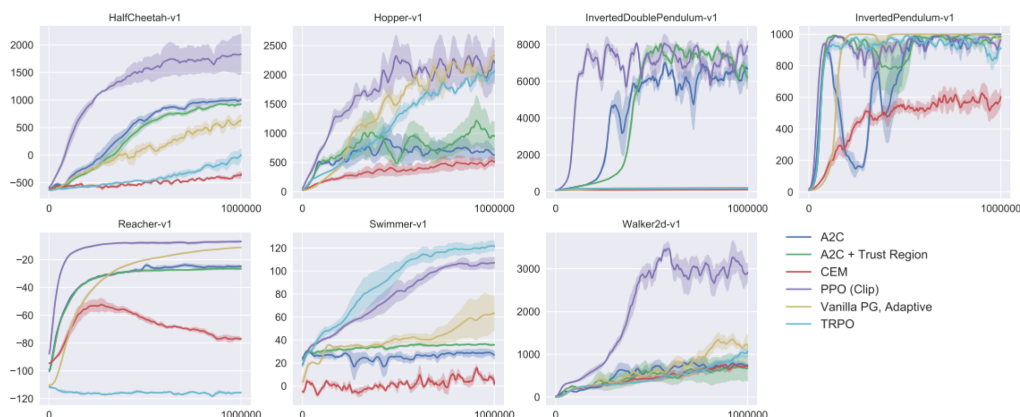

Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.

Abbildung 2.4: Vergleich der RL-Algorythmen
Quelle: [28]

Zusammenfassend ist zu sagen, dass PPO nicht speziell für Sample Effiency entworfen wurde, sondern eine einfache Codestruktur besitzt und die Parameter einfach zu feintunen sind [13]. Da diese beiden Ansätze umgesetzt wurden und die Performance sich trotzdem dem Stand der Wissenschaft annähert und ihn teilweise übertrifft, wurde der Algorithmus zu einem der Benchmarks in Deep RL [28]. Dazu dient der Vergleich in Abbildung 2.4, welcher bekannte RL-Algorithmen wie TRPO, Cross-Entropy Method(CEM), Advantage actor critic (A2C) und Vanilla Policy Gradient mit adaptiver Schrittgröße miteinander vergleicht [28]. Dort ist zu erkennen, dass PPO in den verschiedenen

Umgebungen immer unter den Besten und Schnellsten ist. In HalfCheetah-v1 und Walker2d-v1 ist er der einzige Algorithmus mit guten Ergebnissen.

2.2.4 Beispiele

Dota 2 mit spielexterner AI

Dota 2, ist ein Multyplayer-Online-Battle-Arena-Spiel, kurz MOBA. Das Spiel ist der Nachfolger von Defense of the Ancients, kurz Dota, einer Modifikation von Warcraft 3, einem Strategiespiel von Activion Blizzard [36]. In Dota 2 kämpfen zwei Teams mit jeweils fünf Spielern gegeneinander [18]. Dabei wählt am Start jeder Runde jeder Spieler einen von 117 einzigartigen Helden (Stand 19.6.2019). Diese unterscheiden sich in Aussehen, Fähigkeiten und Eigenschaften.
Das Ziel ist es, mit seinem Team das gegnerische Hauptgebäude, den Ancient, zu zerstören [18]. Innerhalb des Spielverlaufs werden die Spieler stärker, indem sie Gold sammeln und über Erfahrungspunkte ihre Fähigkeiten verstärken [18]. Beides bekommt der Spieler, wenn er gegnerische Einheiten tötet. Diese sind neben feindlichen Helden auch Bot-Einheiten welche auf dem Spielfeld verteilt sind und zusätzlich in der feindlichen Basis erscheinen und auf drei festgelegten Pfaden Richtung eigenem Hauptgebäude laufen [18].
Tote Helden erscheinen nach einer festgelegten Zeit in ihrer Basis neu. Bei jedem Tod verlieren sie allerdings einen Teil ihres Goldes. Dieses wird vom Spieler verwendet, seiner Figur Ausrüstungsgegenstände zu kaufen, die die Charaktereigenschaften stark beeinflussen [18]. Dota 2 ist ein Spiel, bei dem es um Strategie, Ressourcenmanagement, theoretisches und praktisches Verständnis über die Helden und Teamplay geht. Des wegen eignet es sich perfekt für eine lernende KI, welche das Spiel meistern soll.
OpenAI, ein von Elon Musk am 11.12.2015 mitgegründetes Unternehmen [11] beschäftigt sich nach eigenen Angaben mit KI insofern, dass es autonome Systeme schaffen will, die menschliche Fähigkeiten übertreffen aber der Menschheit trotzdem nützen [23].
Dazu wurde am 9.11.2016 der Grundstein für die Dota 2 KI mit dem ersten Commit gelegt [7]. Die Dota AI sollte das Spiel lernen, indem es am Anfang ausschließlich gegen sich selbst spielt [24]. Der dabei verwerndete Algorithmus war eine hochskalierte Version von Proximal Policy Optimization [24], dem in ML-Agents verwendeten RL-Algorithmus. Die benötigte Rechenstärke wurde

in Form von 256 GPUs und 128.000 CPUs bereit gestellt [24]. Mit dieser Rechenkapazität konnte die KI pro Tag etwa 180 Jahre Training absolvieren [24]. Dabei waren die wichtigsten Rewards das Töten von Gegnern und das Gewinnen der Runde [24]. Somit gelang es OpenAI am 11.08.2017, einige der besten eins gegen eins Spieler in der Dota 2 Szene beim The International 7, der Weltmeisterschaft, mit den standardmäßig verwendeten Turnierregeln zu besiegen [7]. Erst am 07.09.2017 gelang es einem menschlichen Spieler den Bot ohne Nutzung von Exploits oder Bugs zu besiegen [7]. Aufbauend auf den Erfolgen im 1v1 wurde angefangen, die Bots im 5v5, dem eigentlichen Spielmodus von Dota 2 zu trainieren. Dafür wurden einige Lernansätze abgeändert, welche in Abbildung 2.5 zusammengefasst werden.

	OPENAI 1V1 BOT	OPENAI FIVE
CPUs	60,000 CPU cores on Azure	128,000 preemptible CPU cores on GCP
GPUs	256 K80 GPUs on Azure	256 P100 GPUs on GCP
Experience collected	~300 years per day	~180 years per day (~900 years per day counting each hero separately)
Size of observation	~3.3 kB	~36.8 kB
Observations per second of gameplay	10	7.5
Batch size	8,388,608 observations	1,048,576 observations
Batches per minute	~20	~60

Abbildung 2.5: Vergleich 1v1 vs 5v5 Bots
In dieser Tabelle werden die 1v1 mit den 5v5 Lernansätzen der OpenAI Dota 2 KI verglichen. Dabei ist ein klarer Leistungsanstieg in den Anforderungen an die 5v5 KI erkennbar. Quelle: [24]

Im Januar 2018 gelang es dem nun unter dem Namen OpenAI Five agierenden KI-Team in einem speziell eingeschränkten Modus mit besonderen Regeln das eigens von OpenAI geskriptete Bot-Team zu besiegen [7]. Im April gewann das KI-Team in einem weniger regelbeschränkten Spiel gegen das eigene OpenAI Team und damit auch gegen ihr erstes menschliches Team. [7] Diesen Ansatz verfolgte man weiter. Die Restriktionen im Spiel wurden weniger und dabei besiegten die OpenAI Five immer stärkere Teams. Am 13.04.2019 gewann das Bot-Team gegen den Weltmeister OG von 2018 zwei zu null. [7] Damit wurde

bewiesen, dass die gelernte AI zu den besten Dota 2 spielenden Teams gehört [7]. In Abbildung 2.6 wurde der Anstieg des spielerischen Könnens der KI im Jahr 2018 erfasst, der in MOBAs anhand der MMR, dem Match Making Rating, gemessen wird [7].

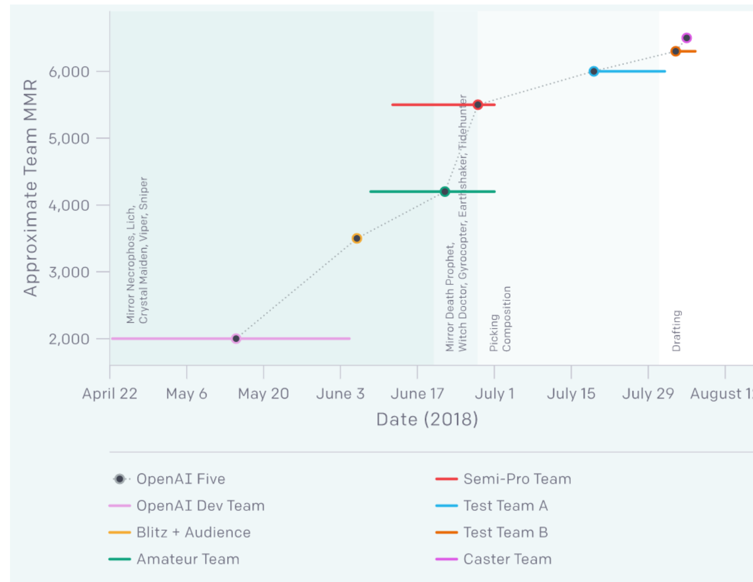

Abbildung 2.6: MMR-Gewinn der OpenAI-Five KIs
Der Graph zeigt den Anstieg der MMR des OpenAI Five Teams im Jahre 2018. Dazu stehen im Vergleich die Werte einiger anderer menschlicher Teams, die teilweise von der KI besiegt wurden. Quelle: [7]

Damit gehört das OpenAI-Five-Projekt zu einem Vorreiter der ML-KI [31] und hat möglicherweise das Potenzial, eine neuen Ära der Videospiel-KI einzuleiten.

Pac-Man mit spielinterner AI

Pac-Man aus dem Hause Namco war 1980 eines der ersten Spiele, das verschiedene erste Ansätze von weitergehender KI enthielt. Hier muss der Spieler die namensgebende Spielfigur durch ein Labyrinth bewegen und dabei die verteilten Punkte einsammeln. Ein Level ist beendet, wenn der Spieler alle Punkte eingesammelt hat. Dabei wird er von vier verschiedenfarbigen Geistern gejagt, welche unterschiedliche Handlungsdirektiven mit verschiedenen Zuständen haben [22]. Diese drei Zustände sind Jagen, Angst und Verteilen [22]. Beim Jagen

versuchen die Gegner, Pac-Man einzuholen, um ihn bei Berührung zu besiegen. Dabei hat jeder der Vier eine andere Strategie [22]. Rot verfolgt aggressiv den Spieler auf dem kürzesten Weg [22]. Pink versucht, ihn auf seinem Weg abzufangen und ihm den Weg abzuschneiden [22]. Türkis patrouilliert in einem Gebiet und ist dabei unabhängig vom Spieler [22]. Der Letzte, Orange, bewegt sich zufällig durch das Spielfeld [22].

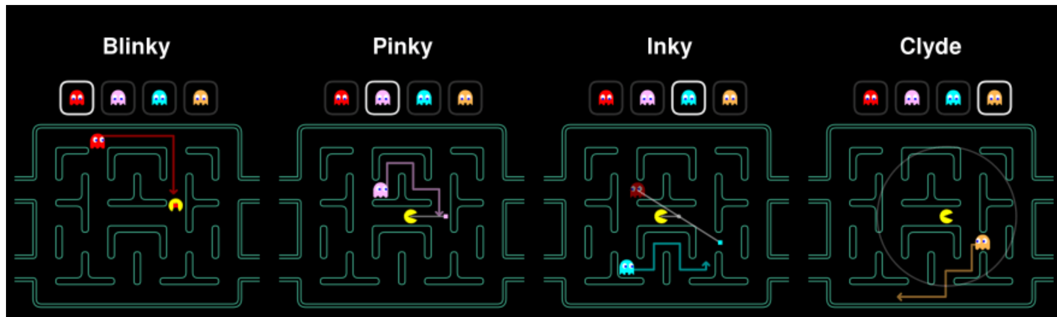

Abbildung 2.7: Zusammenfassung der Pac-Man-Geist-KI im Jagen-Modus
In der Abbildung sind die verschiedenen Jagdansätze der vier Geister dargestellt. Quelle: [22]

Zusammenfassend wird das Jagdverhalten in Abbildung 2.7 bildlich dargestellt. Nach jeweils 20s Angriffszeit gehen die Geister automatisch vom Jagen in den Verteilen Modus [3]. Dabei bewegen sich die Geister in ihre dafür vorgesehene Spielfeldecke und patrouillieren kurz um sie. Dieser Zustand hält anfangs 7 Sekunden und beim dritten Mal nur noch 5 Sekunden, wobei er pro Level maximal vier mal auftreten kann [3]. Anschließend gehen die Bots automatisch wieder in den Jagen Modus. Wenn der Spieler allerdings eine Powerpille einnimmt, ein besonderes Item was im Labyrinth verteilt liegt [3], gehen die Geister in den Angststatus über [3]. Hier fliehen sie vor Pac-Man, da dieser sie nun bei Berührung besiegt [3]. Dieser Zustand hält allerdings nur kurze Zeit und danach gehen die Geister wieder in den Zustand, in dem sie vor der Powerpille waren [3]. Damit gehört Pacman zu den ersten größeren Spielen, welche einen erfolgreichen Statemachine-Ansatz benutzt haben mit einer AI, die verschiedene Verhaltensmuster hat [21].

2.3 ML-Agents

"The ML-Agents toolkit is an open source project which enables researchers and developers to create simulation environments using the Unity Editor and interact with them using a Python API. The toolkit is built to take full advantage of the properties of the Unity Engine described above which make it a potentially strong research platform" ist ein Zitat aus dem ML-Agents-Paper [14], dass das Plugin gut zusammenfasst.

Wie in Kapitel 2.2 angesprochen, werden an die Simulationen einige Anforderungen gestellt. Neben diesen kommen drei weitere Eigenschaften hinzu, die von der Simulation erfüllt werden müssen, damit der Lernvorgang optimal durchgeführt werden kann: Die Simulation muss schnell sein, da ML-Algorithmen oft sehr große Mengen Daten verarbeiten müssen, um eine optimale Lösung zu erhalten [14]. Dazu gehört auch die zweite Eigenschaft, dass mehrere Durchläufe parallelisiert und damit die Berechnungen unabhängig voneinander gemacht werden können. Die letzte Eigenschaft ist flexible Kontrolle [14]. Innerhalb der Simulation muss dem Benutzer die Möglichkeit gegeben werden, einen festen Rahmen an Kontrolle über die Konfiguration sowohl in der Entwicklungsphase als auch in Laufzeit zu haben [14].
Unity besitzt als Engine große graphische Rendering-Fähigkeiten zum Beispiel eigene Shader oder Echtzeitbeleuchtung [14]. Dadurch kann der KI schnell ein fotorealistisches Bild zur Verfügung gestellt werden, um seine sensorische Komplexität abzufragen [14]. Durch die eigene Unity-Physik können verschiedenste Körper realistisch und physikalisch korrekt miteinander reagieren, wodurch die KI beispielsweise Bewegungen lernen kann, die dem Verhalten in der realen Welt sehr ähnlich sind [14]. Durch die in Unity enthaltene Möglichkeit, Skripte mittels C# oder JavaScript zu erstellen und zu benutzen, können unterschiedlichste Gameplay- und Simulationsfeatures implementiert, verändert und eingefügt werden. Dadurch können komplexe Aufgaben an die KI gestellt werden [14] und die kognitive Komplexität wird erfüllt. Die soziale Komplexität wird mit dem gleichen Skript-System erreicht, wenn mehrere Agenten in unterschiedlichen Szenarien verschiedene Aufgaben erhalten [14]. Da in Unity das Rendern des Bildes und die Physik-Engine unabhängig voneinander sind, kann man die Physikkomponenten erheblich beschleunigen, ohne die Bildrate des Renderprozesses zu verändern oder komplett ohne Rendering zu arbeiten [14]. Falls das Rendern relevant sein sollte, können die Geschwindigkeit der Spiellogik und die Bildrate separat eingestellt werden. Solche Konfigura-

tionsmöglichkeiten sind auch für die meisten anderen Aspekte der Simulation vorhanden [14]. So ist es möglich, fortgeschrittene Methoden, wie das Definieren der Curricula, einem Lernansatz bei dem der Level über Laufzeit verändert wird, innerhalb des Trainings zu verwenden, um den Trainingsprozess in Echtzeit zu beeinflussen [14]. Die benötigten Lernumgebungen können frei in Unity erstellt werden und mit der Integration des ML-Agents Plugins kann eine KI trainiert werden, an die unterschiedliche Herrausforderungen aus den vier Komplexitätsanteilen gleichzeitig gestellt werden.

ML-Agents Komponenten

ML-Agents besteht aus drei großen Komponenten [6], die in Abbildung 2.8 abgebildet und deren Verhältnis dargestellt ist. Die Python-API ist für das Lernen zuständig [6]. Sie bildet die Black-Box, auf die der User in der Regel keinen Zugriff braucht, wenn er das Plugin im Rahmen seiner Funktion verwendet. Hier werden die Reinforcement-Learning-Algorithmen welche ML-Agents benutzt, im Hintergrund ausgeführt [6].

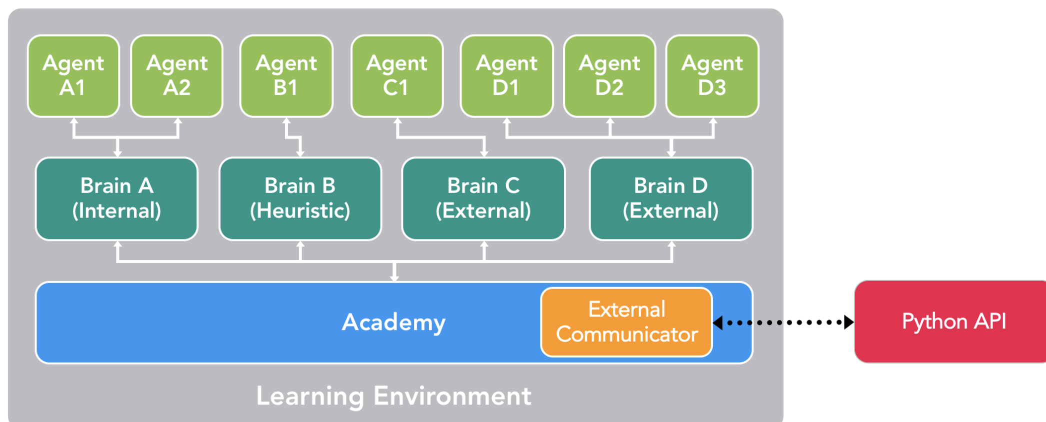

Abbildung 2.8: Bestandteile in ML-Agents
Die Abbildung zeigt eine Zusammenfassung der Bestandteile ML-Agents. Dabei werden die 3 Kernbestandteile Python-API, Learning Enviroment und External Communicator abgebildet. Innerhalb der Learning Enviroment werden zusätzlich die Zusammenhänge der Unity-spezifischen Komponenten dargestellt. Quelle: [6]

Damit die Ergebnisse übertragen werden können, braucht es den externen Kommunikator [6]. Dieser bildet die Verbindung zwischen Unity und der Python-API. Er initialisiert den Lernprozess mithilfe vom User angepasster Konfigurationsdateien [6]. Die genaue Bedeutung der einzelnen Bestandteile dieser Dateien können auf der Githubseite [8] nachgelesen werden. Wie in Abbildung 2.9 zu sehen, gibt der Kommunikator zusätzlich eine allgemeine Anzeige über den Lernfortschritt.

```
 Step: 5000. Mean Reward: -0.680. Std of Reward: 0.893. Training.
onfiguration -> 0.0

 Step: 10000. Mean Reward: -0.453. Std of Reward: 1.055. Training.
 Step: 15000. Mean Reward: -0.349. Std of Reward: 1.026. Training.
 Step: 20000. Mean Reward: 0.058. Std of Reward: 0.871. Training.
 Step: 25000. Mean Reward: -0.012. Std of Reward: 0.974. Training.
 Step: 30000. Mean Reward: 0.499. Std of Reward: 0.706. Training.
 Step: 35000. Mean Reward: 0.578. Std of Reward: 0.754. Training.
 Step: 40000. Mean Reward: 0.775. Std of Reward: 0.568. Training.
 Step: 45000. Mean Reward: 0.769. Std of Reward: 0.564. Training.

 Step: 50000. Mean Reward: 0.861. Std of Reward: 0.413. Training.
 Step: 55000. Mean Reward: 0.914. Std of Reward: 0.320. Training.
 Step: 60000. Mean Reward: 0.921. Std of Reward: 0.310. Training.
 Step: 65000. Mean Reward: 0.967. Std of Reward: 0.148. Training.
 Step: 70000. Mean Reward: 0.960. Std of Reward: 0.198. Training.
 Step: 75000. Mean Reward: 0.981. Std of Reward: 0.014. Training.
 Step: 80000. Mean Reward: 0.980. Std of Reward: 0.090. Training.
 Step: 85000. Mean Reward: 0.985. Std of Reward: 0.008. Training.
 Step: 90000. Mean Reward: 0.982. Std of Reward: 0.084. Training.
 Step: 95000. Mean Reward: 0.986. Std of Reward: 0.007. Training.

 Step: 100000. Mean Reward: 0.983. Std of Reward: 0.084. Training.
```

Abbildung 2.9: Anacondapromt beim Lernen
In dieser Abbildung ist ein Lernprozess dargestellt. Alle 5000 Schritte ist der Anstieg der Meanreward zu erkennen.

Als letzte Komponente befindet sich die Lernumwelt in Unity. Hier werden die Regeln, das Ziel und der Rahmen für das Lernen festgelegt [6]. Um eine Unity-Szene mit ML-Agents einzurichten, muss zunächst das Toolkit in das Projekt importiert werden [5]. Zur Lernumgebung gehören die die drei Bestandteile Agent, Brain (Gehirn) und Academy, die in der Unity-Szene benötigt werden, um lernen zu können [5].
In einer vorher erstellten Spielfläche, die in der Komplexität von einer einfachen ebenen Fläche bis zu einem komplizierten mehrstufigen Labyrinth reichen kann, ist der Agent die zu steuernde Spielfigur, welcher ein Gehirn zugewiesen wird [6]. Die Anzahl der Gehirne und Agenten ist nicht begrenzt, allerdings muss es immer mindestens jeweils eins geben [5]. Gehirne werden dann inner-

halb der Academy verwaltet [4]. In dieser wird auch festgelegt, welche Gehirne vom externen Kommunikator gesteuert werden können [5]. Dies ist aber nicht mit allen möglich, denn es gibt drei verschiedene Arten von Gehirnen: Learning-, Player- und Heuristic-Gehirne. Je nach Gehirn wird der Agent von einer anderen Instanz gesteuert. Das Learningbrain ist das Einzige, was von der Python-API beim Lernvorgang benutzt werden kann. Das Playerbrain ist eine Schnittstelle für den Benutzer, damit er manuell die Lernumgebung testen kann [4]. Das Heuristic-Gehirn ist ein freier Ansatz, bei dem der Agent mit anderen selbst erstellten Skripten mit eigener Logik gesteuert wird. Diese eigenen Skripte haben nichts mit dem Lernansatz zu tun [4]. In allen Gehirnen wird aber festgelegt, welche Eingaben es dem Agent weitergeben kann und wie groß der Observationspace ist [4]. Dieser ist eine Liste von Gleitkommazahlen (Floats), welche Positionen, Geschwindigkeiten und andere Informationen enthalten können [6]. Damit wird festgelegt, was er alles sehen und wahrnehmen kann.

Um mit dem Lernen beginnen zu können, fehlen nur noch die Academy- und Agentskripte. Im Academyskript können generelle Einstellungen zur Umgebung getätigt und im Agentskript vorgegebene Funktionen implementiert werden [5]. Die erste Methode ist AgentReset() [5]. Dabei wird festgelegt, was bei einem Reset mit der Lernumgebung passiert, ob beispielsweise neue Objekte erscheinen oder alle auf alte Positionen zurück gehen [5]. Am Ende der Methode wird auch der momentane Reward an das Gehirn weitergeleitet und auf 0 gesetzt [6].

Falls die Lernumgebung mit Curriculum Learning arbeitet, kann innerhalb des Resets auch der Level verändert werden. Des Weiteren muss in der CollectObservations() Methode implementiert werden, was der Agent sehen und wahrnehmen kann [5]. In der AgentAction(float[] vectorAction, string textAction) ist die vectorAction-Liste und der textAction-String die Eingabe, die vom Gehirn getätigt wird [1]. Diese Anweisungen müssen in die Spielwelt übersetzt werden. Wenn zum Beispiel der Designer einen Körper mithilfe der festgelegten Ausgabe des Gehirns bewegen will, muss er in AgentAction die Bewegung implementieren, die die in vectorAction gegebenen Float-Ausgaben benutzt. Zusätzlich werden in der Methode die Abbruchbedingungen implementiert [5]. Wenn der Agent beispielsweise von der Plattform fällt, muss das Spiel mit AgentReset zurückgesetzt werden und der Agent kann in dem Schritt auch eine Belohnung, positiv oder negativ, bekommen [5].

Als Letztes muss entschieden werden, ob der Agent mit Curriculum-Lernen

oder normalem Lernen arbeiten soll. Für Curriculum-Lernen muss innerhalb des Agent-Skriptes festlegt werden, was die verschiedenen Levelkonfigurationen sind, was in ihnen geschieht, unter welchen Umständen sich die Konfiguration ändert. Zusätzlich zur von beiden Ansätzen benötigten Konfigurationsdatei muss eine geeignete Curriculum-Konfigurationsdatei entworfen werden, auf die der externe Kommunikator zugreifen kann [9]. Wie sich diese zusammensetzt, wird unter [9] erklärt. Sind diese Vorkehrungen getroffen, kann mit dem Lernen angefangen werden.

3 Nortivag

3.1 Das Spiel

Nortivag ist ein kompetitives, physikbasiertes, lokales Mehrspieler-Actionspiel, das alleine gegen die KI oder mit menschlichen Mitspielern gespielt werden kann. Jeder Teilnehmer steuert eine Figur, die je nach Auswahl verschiedene Fähigkeiten und Eigenschaften besitzt.
In verschiedenen Spielmodi und unterschiedlichen Karten kämpft der Spieler gegen seine Gegner. Dabei versuchen die Teilnehmer sich gegenseitig auszuschalten, indem sie sich gegenseitig in Hindernisse drängen. Um das zu erreichen, bietet Nortivag eine Reihe an Möglichkeiten, um die Kräfte erzeugen, die die Bewegung beeinflussen und getroffene Mitspieler mit Hindernissen kollidieren lassen.
In den nachfolgenden Kapiteln werden ausführlich die Steuerung, die Features und der genaue Spielablauf von Nortivag erläutert.

3.1.1 Spielablauf

Zu Beginn des Spiels muss der Spieler einige Einstellungen vornehmen. Jeder wählt, mit welchem Eingabegerät er spielen will. Danach muss eine Karte und ein Spielmodus ausgewählt werden und jeder Spieler entscheidet sich für eines der vielen Elemente, welche verschiedene Fähigkeiten und Figurattribute besitzen. Nach erfolgereicher Konfiguration beginnt das eigentliche Spiel. Je nach Auswahl müssen die Spieler eine bestimme Anzahl an Punkten erzielen, die sie für das Erfüllen des Ziels, den Gegner in Hindernisse zu drängen, bekommen, um die Runde für sich entscheiden zu können. Ist das Gesamtziel erreicht oder die Zeit abgelaufen, ist die Runde mit einer Siegerehrung zu Ende.

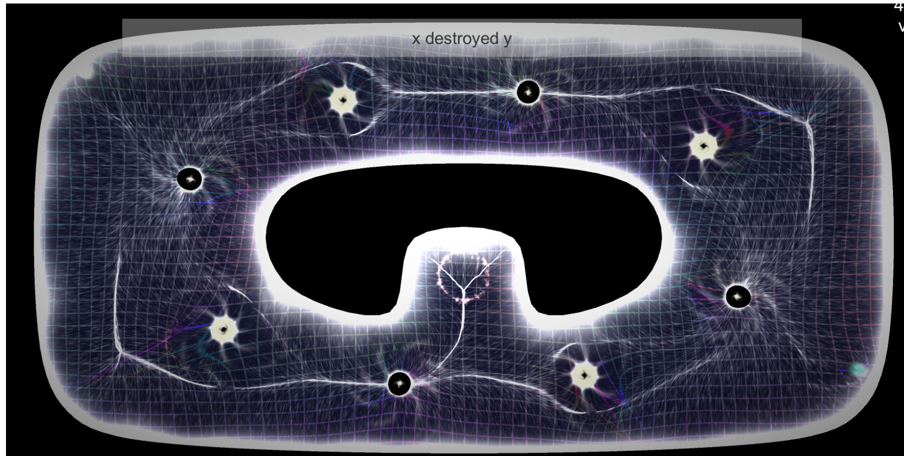

Abbildung 3.1: Nortivag Gameplay
Ein Ausschnitt aus einer Runde im Deathmatch-Spielmodus auf der Karte Daisychain. Dabei bekämpfen sich der der türkise (unten rechts) und der graue Spieler (oben links) im Eins gegen Eins. Dabei werden sie von Attraktoren-Hindernissen und der großen mittleren Wand behindert. Der Ring in der Mitte ist ein Schalter der die Polung der Attraktoren umkehrt.
Quelle: Nortivag

Um eine Vielzahl von unterschiedlichen Einstellungen zu ermöglichen und damit auch eine größere Abwechslung im Spielgeschehen zu bieten, besitzt Nortivag eine Reihe an Features.

3.1.2 Features

Nortivag besitzt sowohl Einzel- als auch Mehrspielerinhalte, wobei die KI-Erstellung für ersteres der relevante Teil ist.
Allerdings befindet sich das Spiel noch in der Entwicklungsphase, weshalb viele Inhalte und Features nicht implementiert oder ausgereift sind. Des wegen kann zu diesem Zeitpunkt nicht mit genauen Zahlen gearbeitet werden.
Der Spieler kann sich eine aus verschiedenen Spielfiguren aussuchen, die im Spiel "Elemente" genannt werden, die dann für die ganze Spielpartie festgelegt sind. Jedes Element besitzt zwei Fähigkeiten und eine Gewichtsklasse. Die Fähigkeiten unterscheiden sich stark von Element zu Element, allerdings sind sie alle im Grunde zwei Projektilgeschosse mit unterschiedlichen Wirkungen, zum Beispiel einer Explosion oder der Fähigkeit, an Wänden abzuprallen. Die drei

verschiedenen Gewichtsklassen leicht, mittel und schwer bestimmen die Basisattribute der Spielfigur. Beschleunigung, Maximalgeschwindigkeit und Masse sind gewichtsabhängig und unterscheiden damit die einzelnen Elemente weiter in ihren taktischen Möglichkeiten.

Neben den Spielfiguren soll es eine große Anzahl an Karten geben, auf denen

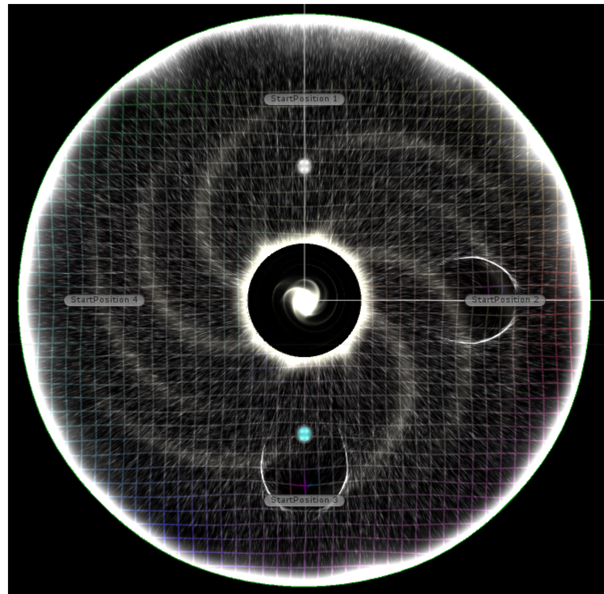

Abbildung 3.2: Nortivag Gameplay auf der BlackHole-Karte
Ein Ausschnitt aus einer Eins gegen Eins Runde im Deathmatch-Spielmodus
Quelle: Nortivag

gespielt werden kann. Jede hat dabei eine Besonderheit, zum Beispiel ein großes schwarzes Loch in der Mitte. In der fertigen Version von Nortivag soll es eine Reihe an Fallen und Featueres geben. Darunter befinden sich Attraktoren, sich bewegende Wände, Todes- und Giftzonen und Schalter, welche die Umgebung innerhalb einer Runde immer wieder verändern. Da vorerst nur getestet werden soll, inwieweit ML-Agents die Wegfindung des Agenten übernehmen kann, werden die meisten dieser Features zunächst nicht zum Einsatz kommen.

3.1.3 Steuerung

Nortivag kann mit der Tastatur und dem Controller gespielt werden. Vier Richtungstasten oder ein Analogstick bewegen dabei die Spielfigur. Zwei weitere Tasten sind für die Fähigkeiten vorgesehen.
Die Bewegungssteuerung erfolgt aufgeteilt kraft- und richtungsbasiert. Wenn der Spieler seine momentane Bewegungsrichtung ändern will, muss er zusätzlich zu seiner eigenen Bewegungskraft den Kräften an seiner Position entgegenwirken. Die Kraft, die er dafür aufwenden kann und damit auch wie agil er sich durch die Karte bewegen kann, hängt von seiner Gewichtsklasse ab.
Mit zunehmender Geschwindigkeit lässt sich die Spielfigur immer schwerer steuern, was bei den engen Passagen der Karte und ihren Fallen oft zum Crash führt.
Der richtungsbasierte Teil dient zur Vereinfachung der Steuerung, indem die Figur sich immer sofort ein wenig mit in die Richtungsanweisung vom Spieler bewegt. Dadurch wird die ganze Steuerung um einiges flüssiger. Bei hoher Geschwindigkeit ist es aber immer noch sehr schwierig, die Spielfigur genau zu navigieren.
Andere Fähigkeiten und Fallen üben zusätzlich Kräfte auf die Spielfigur aus, denen der Spieler entgegenwirken muss, um erfolgreich auf der Karte agieren zu können und zu überleben.
Falls der Spieler direkt von einem Projektil getroffen oder zu stark von einem Gegner gerammt wird, wird er kurz betäubt, wodurch er für eine gewisse Zeit die Möglichkeit verliert, seinen Charakter zu steuern. Wenn keine vier menschlichen Spieler anwesend sind, können die restlichen Plätze durch die KI ersetzt werden. Wie die ungelernten künstlichen Gegner implementiert werden können, wird im folgendem Teil beschrieben.

3.2 Interface

Innerhalb von Nortivag gibt es zwei Interfaces. Zum einen gibt es das Hauseigene, welches eine komplett freie Implementation jeglicher KIs ermöglicht und dem Benutzer lediglich eine Reihe an Informationen bereitstellt und im Gegensatz dazu gibt es die Integration des ML-Agents-Interfaces, was in Kapitel 4 genauer erklärt wird.
Das normale Interface ermöglicht es, sehr einfach eine eigene KI in das Spiel

einzubinden. Sie muss als DLL-Datei, die die Vorgaben erfüllt, in einem vorgesehenen Ordner abgelegt werden. Dafür müssen nur die vorgegebenen Methoden implementiert werden, welche folgende sind:

- OnLevelStart(): Hier können Voreinstellungen getroffen werden, die relevant werden bevor das eigentliche Spiel startet.
- EndOfFrame(): Diese Funktion wird am Ende jedes Frames aufgerufen und ist größtenteils relevant für Debug-Zwecke.
- UseSkill(): Hier kann bestimmt werden, ob und welcher Fähigkeit in diesem Frame benutzt wird.
- Init(): Wird einmal beim Spielstart ausgeführt.
- NewForce(): hier findet die ganze Berechnung und Logik für die KI statt. Am Ende wird ein Ergebnisvector als Richtungsanweisung der KI an das Spiel zurückgegeben.

In jedem Frame bekommt die KI wichtige Informationen übergeben, damit der Programmierer seiner KI je nach Situation andere Handlungsanweisungen geben kann. Die Informationen sind unter anderem die eigene Position und Geschwindigkeit und die der Gegner, die Zeit, wie lange die Runde geht und andere spielrelevante Informationen. Diese werden der KI mit einer speziellen Info-Klasse übergeben, sodass sie in Echtzeit alle Werte abrufen kann.

3.3 Die Simulation Nortivag Lite

Bevor ML-Agents in Nortivag implementiert werden kann muss getestet werden, ob das eigentliche Gameplay mit dem Tool erlernbar ist. Dazu wird eine Gameplay-Kopie entworfen, die Nortivag nachempfunden ist und Nortivag Lite heißt. Diese Kopie besitzt allerdings einen viel einfacheren Aufbau, der nur notwendige Features besitzt, um den Agenten die Bewegungssteuerung inklusive der Wegfindung lernen zu lassen. Fähigkeiten und Kampfverhalten spielen in der Simulation keine Rolle.
Des Weiteren kann innerhalb der Kopie, dank leichterem Aufbau viel leichter Fehleranalyse betrieben werden und es können eigene Debug-Tests durchgeführt werden, die in Nortivag aufgrund der großen Codestruktur nur sehr schwer umsetzbar sind. In den zwei nächsten Unterpunkten wird kurz beschrieben, wie NLite aufgebaut ist und welche Unterschiede zu Nortivag bestehen.

Aufbau

NLite entsteht genauso wie Nortivag in der Unity Engine. Als Spielfeld wird eine einfache Ebene benutzt und als Wände dienen einfache Würfel-Objekte, welche in Form und Größe angepasst wurden. Eine grundlegende Academy, ein Brain und ein Agent sowie sein Zielobjekt sind direkt in der Szene integriert. Das Agentskript ist hierbei das gleiche, das später in Nortivag benutzt werden soll. Die wichtigen Informationen, die die AI in Nortivag über seine Info-Klasse bekommt, werden in NLite direkt dem Agenten einzeln übergeben.
In den Ansätzen der AI in Kapitel 4.3 wird erklärt, welche Level in der Kopie erstellt werden und wozu diese in den Experimenten benutzt werden.

Unterschiede

In folgender Tabelle 3.1 werden kurz Unterschiede zwischen NLite und Nortivag aufgezeigt. Dabei werden auch einige Vorteile der Simulation deutlich.
Das Lernen müsste in der Simulation bedeutend schneller gehen, weil es keinerlei Effekte und keine weiteren Hintergrundberechnungen außer der reinen Physik gibt, wie zum Beispiel Punktestände oder ähnliches, was die Performance der Simulation deutlich steigern sollte. Das NLite als eine 3D Umgebung implementiert wurde, macht für das Lernen keinen Unterschied im Vergleich zum späteren Nortivag-Lernen, da das Gameplay auch nur auf 2D Bewegung basiert. Der Unterschied in der Steuerung hat auch keinen Einfluss auf das Lernen, da die reine Bewegungssteuerung in Nortivag und NLite die gleiche ist und sich nur die Effektstärke der Eingabe unterscheidet. Mit der fertigen Simulation kann daher effizient getestet werden, welche Ansätze, Lern- und Konfigurationseinstellungen die besten sind, um das Wegfindungsproblem gut zu lösen. Mit der funktionierenden Simulation muss nur noch ML-Agents in Nortivag implementiert und ein Konzept für die Experimente entworfen werden.

Tabelle 3.1: Unterschied Nortivag zu NLite

Merkmal	Nortivag	NLite
Umgebung	2D	3D
Steuerung	Force- und Direktsteuerung	Nur Forcesteuerung
Spielstart	eigener Game-Manager startet Spiel und initialisiert alle relevanten Objekte	direkter Spielstart in Szene
Max. Anzahl der Felder	1 Feld	Limitiert durch PC-Leistung
Physik	eigens geschriebene Physik	Unity-Physik
Wände	eigene Boundary Shapes	Default Cubes
Effekte	so weit minimiert wie möglich	keine
Sonstiges	Performanceprobleme und Bugs	Zugeschnitten auf die Lerntests eigene Debug-Features

In dieser Tabelle ist der Unterschied von Nortivag zu seiner Kopie NLite abgebildet. Dabei sollen die jeweiligen Unterschiede so wenig Einfluss auf die Lernversuche haben wie möglich. Die Benutzung von NLite soll das Lernen beschleunigen und effizienter machen.

4 ML-Agents in Nortivag

In diesem Kapitel wird anhand der Abbildungen 4.1 und 4.2 der Ausgangszustand von Nortivag und die Implementation des ML-Agents Plugins in Nortivag erklärt. Daraufhin wird die KI-Entwicklung mit den Ansätzen und den dabei entstehenden Hypothesen für Nortivag und NLite erläutert.

4.1 Ausgangszustand und Vorbereitung

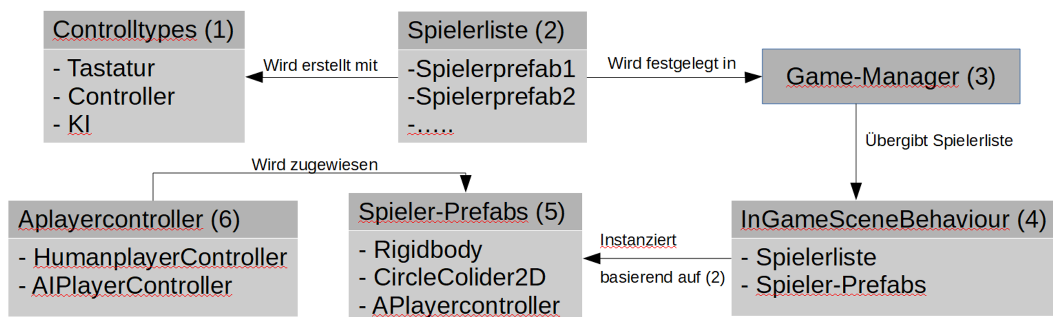

Abbildung 4.1: Ausgangszustand der Implementation von ML-Agents in Nortivag
In der Abbildungs ist der ML-Agents-relevante Teil der Nortivagimplementation schematisch dargestellt

Die Steuerung in Nortivag ist so geregelt, dass es einen virtuellen Controller gibt, der als Schnittstelle für die verschiedenen Eingabemöglichkeiten dient. Von diesem abstrakten APlayerController(6) leiten sich die einzelnen speziellen Eingaben ab. HumanPlayerController und AIPlayerController bilden dann die Grundlage, damit die Spieler oder die KI ihre Spielfigur steuern können. Zusätzlich muss speziell für jeden dieser Controller ein Prefab(5) einer eigenen Spielsteuerung erstellt werden. Das Zutreffende wird dann im InGameSceneBehaviour(4) basierend auf dem in den Spielereinstellungen festgelegten Controlltype(1) und der im Game-Manager(3) festgelegten Spielerliste(2) instanziiert.

Diese bestehenden Skripte müssen um die ML-Agents-Komponenten erweitert und fehlende ML-Agents-spezifische neu implementiert werden.
Wenn die reine Funktionsweise des ML-Agents-Toolkits sichergestellt ist, müssen weitere spezielle Features und Funktionen implementiert werden, um das Toolkit mit Informationen über das Spiel zu versorgen. Dazu gehören neben den Standardinformationen, die auch die KI bekommt, Möglichkeiten, Rewards festzulegen und spezielle Spielzustände abzufragen.
Zusätzlich müssen verschiedene spezielle Karten erstellt werden, die ihre eigene Logik brauchen, auf der der Agent in einem geregelteren Umfeld lernen kann. Im nächsten Schritt wird erläutert, wie diese Implementierung umgesetzt wurde, welche Features und Funktionen zusätzlich eingebaut und von der Standard-KI übernommen wurden.

4.2 Implementierung des Plugins

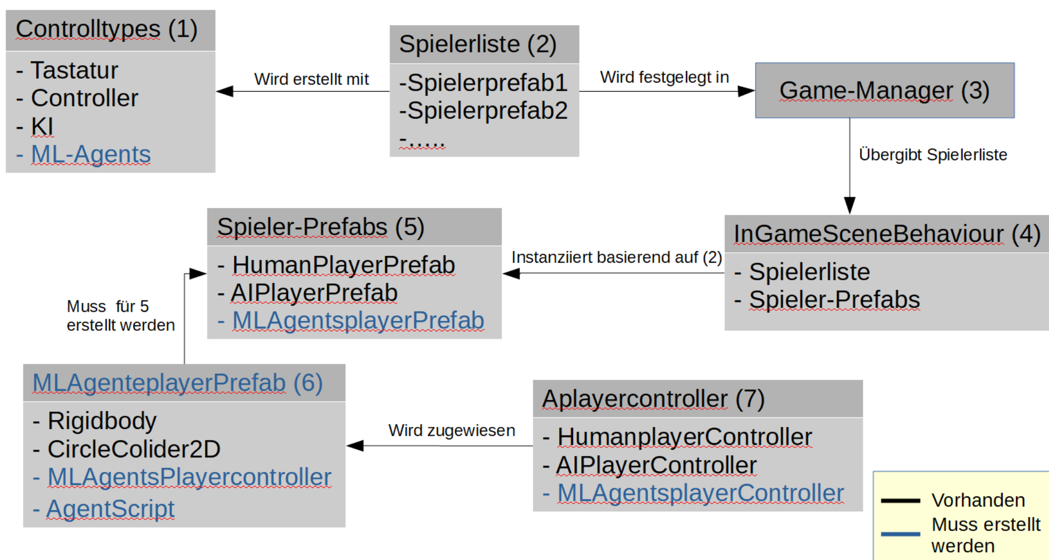

Abbildung 4.2: Implementierung von ML-Agents in Nortivag
In der Abbildung sind die Zusätze und Implementierungen schematisch dargestellt die nötig sind, um ML-Agents in Nortivag benutzen zu können.

Um die Steuerung der Spielfiguren durch das ML-Agents-Toolkit zu gewährleisten, müssen die Steuerungsoptionen erweitert werden, die momentan nur Tastatur, Controller und AI enthalten. Dafür wird bei den bestehen-

den Controlltypes MLAgents hinzugefügt (1). Mit diesen Types wird daraufhin im Game-Manager(3) eine Spielerliste(2) erstellt. Wenn dieser im Game-Manager(3) ausgwählt wird, wird innerhalb des InGameSceneBehaviour(4) ein MLAgentsPlayerPrefab-GameObject(6), das vorher erstellt wurde, instanziiert. An dem Objekt hängen, wie bei allen anderen Spieler-Prefabs(5), die Standard Spielerkomponenten Rigidbody2D, Circlecollider2D und der PlayerController. Für ML-Agents wurde der MLAgentsPlayerController implementiert, der nur die abgeleitete Grundversion des APlayerController(7) ist, in der die Spielinformationen mit einem nInfo-Struct an den Agenten weitergegeben werden. Zusätzlich enthält das MLAgentsPlayerPrefab(6) noch das Agent-Skript. Die Academy wird an der gleichen Stelle erstellt, an der sich der InGameSzeneBehavoir befindet und das Gehirn wird nur in den Assets erstellt und muss sich nicht in der Unity-Szene befinden. Nortivag ist damit vorbereitet, um AIs mit ML-Agents lernen zu lassen. Was fehlt, sind noch die Standard-Skripte Agent, Academy und Brain und eine geeignete Karte. Letzteres wird ein einfaches Spielfeld mit ein paar Wänden ohne besondere Nortivag-Features sein. Falls der Bot erfolgreich durch die Karte navigieren kann, wird diese mit weiteren Wänden versehen, um die Navigation zu erschweren. Falls dies durch die KI auch gelöst werden sollte, können Fallen und andere Features hinzugefügt werden. Wie oben beschrieben, wird der Agent innerhalb des Spielstarts vom IngameSceneBehaviour instanziiert.

4.3 KI entwickeln für Nortivag und NLite

Im Folgenden werden die Ideen und Ansätze erläutert, die getestet werden. Dabei wird zwischen NLite und Nortivag unterschieden und die daraus resultierenden Experimente werden geordnet. Die Ziele und Erwartungen an den jeweiligen Test werden in Ergebnishypothesen begründet und zusammengefasst.

4.3.1 Ansätze

Um mit dem Lernen anfangen zu können, müssen noch die notwendigen, in Kapitel 2.3 beschriebenen ML-Agents-Bestandteile entworfen und implementiert werden. In den Beispielumgebungen des ML-Agents-Tools, befindet sich eine

Beispiellernumgebung, die der Wegfindungsaufgabe von Nortivag sehr ähnlich ist. In dieser wurde die Standard-Academy benutzt, weshalb bei der Academy auch für Nortivag und NLite zunächst die Grundversion genommen wird, die keine weiteren Features implementiert.
Beim Agent-Skript wird zwischen NLite und Nortivag unterschieden, da es klare Abweichungen zwischen der Kopie und dem Hauptspiel gibt (Tabelle 3.1) und der Aufbau beider Unity-Projekte sehr verschieden ist. In der Kopie wird die Bewegung des Agenten innerhalb der AgentAction()-Methode ausgeführt, wohingegen beim Original eine Variable gesetzt wird, die innerhalb vom ML-AgentsPlayerController abgefragt und dort in die Movement-Uptdate-Methode weitergegeben wird. In der Methode werden auch die Abbruchbedingungen implementiert. Diese sind immer erfüllt, wenn der Agent mit etwas zusammen stößt oder in NLite von der Plattform fällt, weil es keine äußere Wandbegrenzung gibt.
Innerhalb der AgentReset wird der Agent bei beiden Simulationen auf eine neue zufällige Position innerhalb der Karte, die nicht in einer Wand liegt, gesetzt und sein Ziel liegt innerhalb eines vorher festgelegten Maximalradius um den Agenten mit den gleichen Bedingungen wie er selbst.
NLite biete sich als effizientere Lernumgebung an, da der Aufbau der Kopie extra für diese Lerntests zugeschnitten ist und bessere Debug-Optionen implementiert wurden, die in Nortivag nicht realisierbar waren. Mit den auf der Kopie durchgeführten Tests wird herausgefunden, welche der gewählten Ansätze am besten funktionieren. Mit diesen Ergebnissen kann die Frage geklärt werden, ob ML-Agents sich für Nortivag eignet.
Da innerhalb des ML-Agents-Plugins ein Beispiel existiert, bei dem ein Agent die Wegfindung zu einem Ziel lernen muss[5], muss der allgemeine Funktionstest nicht wiederholt werden. So kann direkt innerhalb der Konfigurationstests mit einigen Leistungstests auf einer speziellen Karte, die in Abbildung 4.3 links abgebildet ist, angefangen werden. Auf diesem Bild ist die Spielfigur des Agenten die schwarz-weiße Kugel und das Ziel die gelbe Kugel. Fragen die mit diesen Tests beantwortet werden sollen sind: welchen Unterschied mehrere gleichzeitig lernende Agenten auf jeweils einer Karte (Feld) machen, wie viel effektiver Curriculum-Lernen im Gegensatz zum normalen Lernen ist, welche Observationansätze die besten sind und ob es einen Unterschied zwischen positiven und negativen Reward-Strukturen gibt. [1].

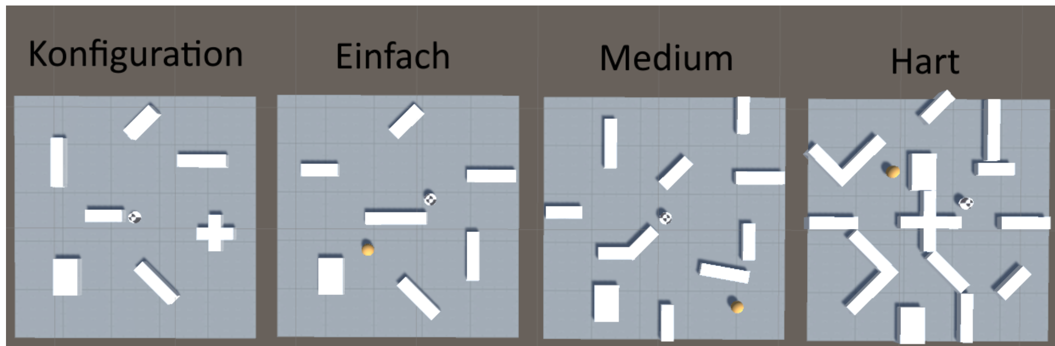

Abbildung 4.3: NLite-Level
Hier sind die 4 Level abgebildet die in NLite benutzt werden, um die Experimente durchzuführen.

Für das erste Experiment mit unterschiedlicher Anzahl an Agenten wird getestet, wie lange das Modell braucht, um zu lernen, den Agenten zum Ziel zu steuern. Im ersten Durchgang mit nur einem Agenten auf einem Feld und im Zweiten mit neun parallelisierten Agenten und auf je einem eigenen Feld gleichzeitig. Die Agenten werden zeitgleich auch mit den Funktionen implementiert, die für das Curriculum-Lernen benötigt werden. Die Logik, dass der Level sich mit dem Lernfortschritt verändert, ist im Agenten implementiert. Dort werden verschiedene Levelkonfigurationen programmiert, die später das Lernen der Experimente vereinfachen sollen. Beispielsweise ist die erste Konfiguration eine leere Karte ohne Wände, damit für das Experiment mit der verschiedenen Anzahl von Agenten keine extra Karte angefertigt werden muss. In den folgenden Konfigurationen erscheinen immer mehr Wände in der Umgebung und der maximale Abstand, in der das Ziel vom Agent spawnen kann, wird vergrößert. Die Änderung des Levels geschieht dabei nach verschiedenen Zeitabschnitten innerhalb des Lernens und wird in Abbildung 4.4 und Tabelle 4.1 zusammengefasst.
Bei der Agentobservation gibt es zwei Ansätze. Der Erste ist, dass der Agent lediglich seine eigene Position, Geschwindigkeit und die Position des Ziels kennt und dafür die Map mithilfe von Sichtstrahlen wahrnehmen kann. Der Zweite ist, dass er zusätzlich zu seiner Position, die des Ziels und seiner Geschwindigkeit die Position der Mittelpunkte aller Wände bekommt. Nur mit dem Mittelpunkt besitzt der Agent zunächst keine Informationen über die Außmaße der jeweiligen Wand und muss jede Begrenzung einzeln lernen.

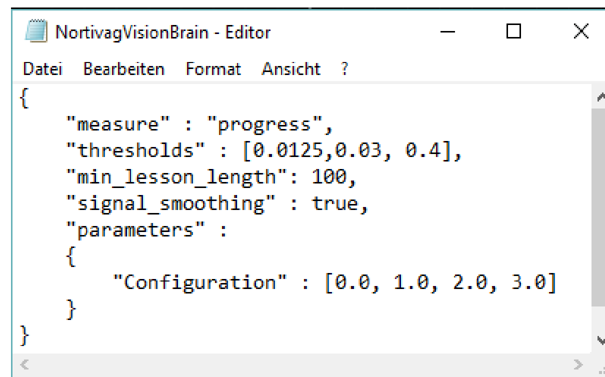

Abbildung 4.4: In den Experimenten genutzte Curriculum-Konfiguration

Tabelle 4.1: Erklärung der Curricula-Datei aus Abbildung 4.4

Konfiguration	Übergang bei (in %)	Übergang bei (in Iterationen)	Leveländerungen
0	0	0	nur ein leeren Feld mit 10x10 Ausmaß
1	1,25	100.000	maximal Spawn-Abstand zwischen Ziel und Agent wird erhöht
2	3	240.000	erste Wände erscheinen
3	40	3.200.000	vollständiger Level

Die Tabelle liefert eine Zusammenfassung, welche Leveländerungen beim Wechsel einer Konfiguration auftreten.

Der Sichtstrahlenansatz wird dabei als Grundlage für die vorherigen Experimente genommen und der Agent kann mit seinen neun Strahlen mit jeweils 40 Grad Abstand die Karte wahrnehmen. Der letzte Einstellungstest ist dann, ob es einen Unterschied gibt zwischen einem positiven und einem negativen Reward-Ansatz. Beim positiven bekommt der Agent Punkte dafür, wie nah er am Ziel ist. Je näher er dran ist, desto mehr Punkte bekommt er. Beim negativen Ansatz bekommt er gleichbleibend negative Punkte solange er sein Ziel nicht erreicht hat. Wenn die KI eine Wand berührt, beendet sie bei beiden Ansätzen den Durchlauf mit einer Reward von -1 und +1, falls sie das Ziel erreicht. Damit liegt die Reward auch immer in den empfohlenen Grenzen der Entwickler zwischen 1 und -1, da es sonst zu unerwünschten Lerneffekten

kommen kann
Wenn diese Tests abgeschlossen werden, entsteht daraus die finale Konfiguration, mit der zunächst die drei NLite-Level aus Abbildung 4.3, die alle einen anderen Schwierigkeitsgrad (Leicht, Mittel, Schwer) haben, gelernt werden sollen. Mit zunehmender Schwierigkeit werden bei den Karten die Bewegungsmöglichkeiten des Agenten kleiner. Mit mehr Wänden werden die Gassen immer kleiner und oftmals ist der Weg zum Ziel versperrt. Des wegen ist das schwere Feld nur noch ein kleines Labyrinth, in dem der Agent sich zurecht finden soll. Nortivag soll am Ende verschiedene Level besitzen, welche in ihrer Schwierigkeit zwischen dem mittleren und dem schweren Feld liegen.
Mit dem Leveltest soll die Lernfähigkeit von ML-Agents in der Wegfindung ausgereizt werden. Zeitgleich soll herausgefunden werden, wie schwer Level in Nortivag sein können damit die spätere KI sie immer noch beherrschen kann. Nach diesen NLite-Tests wird die finale Konfiguration zu Nortivag übertragen und getestet. Als erfolgreich gelten die Versuche, wenn der Agent das Ziel im gegebenen Zeitrahmen von 30s erreicht und eine geringe Fehlerquote aufweist, die unter der in den Hypothesen genannten liegt. Die Fehlerquote wird gemessen, indem das fertige Modell 100.000 mal das Ziel erreichen muss und gezählt wird, wie oft es dies nicht schafft. Trotz dieser Menge an Versuchen unterliegt der Test einer gewissen statistischen Schwankung die später in Betracht gezogen werden muss.
Damit die Experimente durchgeführt werden können, muss noch eine Konfigurationsdatei für ML-Agents erstellt werden. Dafür wird die in Abbildung 4.5 mitgelieferte trainer_config.yaml genutzt. Das Einzige was vorerst verändert wird, sind die max_step. Außer dem Versuch mit unterschiedlicher Anzahl an Feldern wird jeder Lernversuch 8.000.000 Iterationen durchlaufen. Bei sinnvoller Konfiguration sollte die KI weniger Zeit brauchen, um ihre Aufgabe zu lernen.
Im nächsten Abschnitt werden diese Ansätze geordnet, damit in sinnvoller Reihenfolge gelernt werden kann. Gleichzeitig werden einige Hypothesen ausgearbeitet.

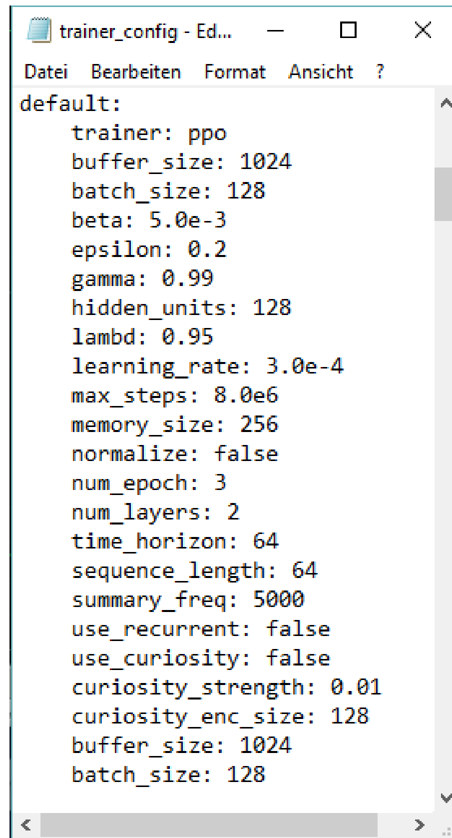

Abbildung 4.5: Standard-Konfigurationsdatei mit abgeänderter maximaler Schrittanzahl

4.3.2 Vergleich und Hypothesen

In der Tabelle 4.2 werden die Konfigurationsexperimente zusammengefasst und Ergebnishypothesen aufgestellt, welche beim Lernen herauskommen können.

Mit erfolgreicher Konfiguration werden in NLite die drei verschiedenen Level gelernt. Da angenommen wird, dass Sichtstrahlen sich besser als Observation eignen als die Mittelpunktposition der Wände, werden die Level jeweils drei Mal mit einer verschiedenen Anzahl an Strahlen gelernt. Dabei ist wichtig, inwiefern sich die Menge der Strahlen auf das Ergebnis auswirken. Hat die Anzahl nur Auswirkungen auf die mechanische Steuerung des Agenten oder verbessert sie auch die Wegfindung, die besonders im schweren Level wichtig ist. Dadurch kann ermittelt werden, wieviele Strahlen am Ende für den Agent

in Nortivag sinnvoll sind. In Tabelle 4.2 wird die Lernreihenfolge mit ihren Hypothesen dargestellt. Allerdings werden für diese Experimente nur Annahmen über die Fehlerquoten getroffen, ob der Agent in der Lage ist, das Level zu lernen und welche Besonderheiten am Ende auftreten können.

Mit den Ergebnissen aus den NLite-Tests kann das gewonnene Wissen in Nortivag übertragen werden. Die Hypothese dafür ist allerdings, dass der Nortivag-Test schlechter abschneiden wird als die Tests in NLite. Das hängt davon ab, wie gut die Steuerung in NLite nachempfunden wurde und wie gut die Steuerung der Spielfigur durch ein externes Model in Nortivag funktioniert. Auf den zweiten Punkt kann kein Einfluss genommen werden und er könnte deshalb das größte Problem dieser Tests werden.
Im nächsten Kapitel werden die Experimente durchgeführt und die Ergebnisse ausgewertet, aus denen in Nortivag eine Lernumgebung gebaut werden und ein passender Agent entwickelt werden kann, der dann das finale Ergebnis dieser Arbeit lernt und die Frage beantwortet, ob ML-Agents ein geeignetes Tool ist, um eine funktionierende AI in Nortivag zu implementieren.

Tabelle 4.2: Konfigurationsexperimente und Hypothesen

	Experiment	Hypothese
1	Felderanzahl	Das gewünschte Lernergebnis, dass der Agent sein Ziel sicher erreicht, sollte bei neun gleichzeitig lernenden Feldern in weniger Durchläufen erreicht werden als bei nur einem Feld. Durch die Parallelisierung entsteht mehr Rechenaufwand, der die Simulation in diesem Experiment nicht entscheidend verlangsamen sollte. So könnte theoretisch der Neun-Feldversuch seine Aufgabe um ein vielfaches schneller lernen als der Ein-Feldversuch.
2	Curriculum oder Normal	Laut den ML-Agents-Entwicklern kann sinnvolles Nutzen von Curriculum-Lernen den Lernprozess stark beschleunigen [9]. Am Start des Experiments sollte der Curriculumagent seine Aufgabe innerhalb der ersten 20.000-40.000 Iterationen lernen und jedes Mal bei einer Leveländerung wird die KI einen Reward-Verlust haben, der nicht so stark ist, um auf das Reward-Niveau des normal lernenden Agenten zu fallen. Beim Konfigurations-Level verläuft die Lernkurve des Normal-Agenten viel flacher, als die des Curriculum-Agenten. Bis zur 8.000.000 Iteration sollten beide Agenten beim Konfigurationslevel trotzdem ungefähr die gleichen Reward- und Performancewerte haben.
3	Rays- oder Positionobservation	Die zwei gängigen Ansätze der ML-Agents-Beispiele werden innerhalb von NLite getestet. Wahrscheinlich ist mithilfe von Rays das Lernen effektiver, da es Speichereffizienter ist und praktischer, da die Menge an zu überwachenden Variablen bei den Rays konstant ist und bei den Positionen von der Menge der im Spiel befindlichen Körper abhängt. Des Weiteren muss der Agent bei den Positionen zusätzlich die nicht gegebenen Maße der Wände lernen, was zusätzlich Zeit in Anspruch nimmt.
4	Positive und Negative Rewards	Die ML-Agents-Beispiele arbeiten außerhalb der Ziel-Reward oft nur mit negativen Rewards. RL benutzt normalerweise aber positive Rewards, weshalb der Unterschied verglichen werden muss. Dieser dürfte aber nicht sehr groß ausfallen.

Tabelle 4.3: Levelexperimente und Hypothesen

	Level	Nr. der Strahlen	Erwartete Fehlerquote	Hypothese
5	Leicht	5	<5%	Da es ein sehr einfacher Level ist, wird der Agent ein ähnlich gutes Ergebnis wie die Neun- und Achtzehnstrahlagenten haben
	Leicht	9	<5%	Wird ungefähr die gleiche Quote haben wie der Achtzehner-Agent aufgrund des einfachen Levels.
	Leicht	18	<5%	Bestes Ergebnis aber keine signifikante Verbesserung zum Neuner-Agent
6	Mittel	5	5-10%	Fünf Strahlen sind nicht ausreichend, um ein Ergebnis wie im ersten Level zu erhalten
	Mittel	9	<5%	Der mittlere Level wird nicht schwer genug sein, damit sich die Quote der Neuner- und Achtzehner-Agenten groß unterscheidet.
	Mittel	18	<5%	Bestes Ergebnis und hat diesmal einen größeren Unterschied vom Neuner-Agents als beim einfachen Level
7	Schwer	5	>90%	Bei der Levelschwierigkeit kann das Model seine Aufgabe nicht lernen. Der Agent schafft seine Aufgabe nur in seltenen Fällen, wenn das Ziel in seine Nähe spawned
	Schwer	9	80-90%	Ähnlich wie beim Fünf-Strahlversuch allerdings schafft der Agent längere Wege als der Fünfer-Agent.
	Schwer	18	70-80%	Der Level wird auch zu schwer für den Achtzehner-Agent sein, allerdings wird er sich zu weiter entfernt spawnenden Zielen bewegen können

5 Evaluierung

Im Evaluierungskapitel werden die oben genannten Experimente umgesetzt. Innerhalb der Vorbereitung wird erklärt, welche Einstellungen gemacht werden müssen und was alles nötig ist, um die Experimente durchzuführen. In der Durchführung wird der genaue Ablauf der Lernversuche beschrieben, welche Programme benutzt wurden und auf welchem PC gelernt wurde. Die Ergebnisse werden darauf in der Auswertung analysiert und mit den Hypothesen aus 4.3.2 verglichen.

5.1 Konfigurationstests in Nortivag Lite

5.1.1 Vorbereitung

Innerhalb des ersten Testsets soll die optimale Konfiguration für den Leveltest und später den finalen Nortivag-Versuch herausgefunden werden. Dafür werden zwei verschiedene Agenten benötigt, welche beide Curriculum-Lernen unterstützen und die zwei verschiedenen Reward-Ansätze verwenden. Ein Agent bekommt als Observation neun Sichtstrahlen, die im 40-Grad-Winkel um ihn angeordnet sind. Der andere arbeitet mit den Mittelpunktpositionen der Wandobjekte. Wegen dieses Unterschieds besitzen die Agenten jeweils ihr eigenes Gehirn. Der erste der beiden Agenten ist dabei der, der am besten abschneiden sollte, und kann deshalb als Benchmark für die anderen Tests dienen. So kann der Lernaufwand weiter reduziert werden.
Die Konfigurationstests werden auf einer speziellen Karte, welche bereits in Abbildung 4.3 links im Bild gezeigt wurde, durchgeführt, die für paralleles Lernen neunmal dupliziert vorliegt und die mit Curriculum-Lernen die speziellen Bedingungen der einzelnen Experimente erfüllt.
Das Curriculum ist so aufgebaut, dass innerhalb der ersten zwei Konfigurationen keine Wände im Level existieren, und so der Agent zuerst lernt, nicht von

der Plattform zu fallen und seinen Zielkörper zu erreichen. Innerhalb dieser zwei Konfigurationen ändert sich nur der Maximalabstand, den das Ziel nach dem Spawn vom Agenten haben kann. In der dritten Konfiguration werden die ersten Wände hinzugefügt, wodurch der Agent sein Ziel nicht mehr so einfach erreichen kann. In Konfiguration Vier erscheinen die letzten Wände, und der Level ist komplett. Die jeweiligen Veränderungen des Levels finden nach den ersten 10^5, $2,4 \cdot 10^5$ und $3,2 \cdot 10^6$ Iterationen statt, da der Agent am Anfang sehr wenig Zeit braucht, um den ersten Teil zu lernen und mit dem ersten Erscheinen der Wände bei $2,4 \cdot 10^5$ sehr viel mehr Zeit braucht, um seine nun viel anspruchsvollere Aufgabe zu bewältigen.
Angefangen mit dem ersten Experiment wird getestet, wie stark der Zeitunterschied zwischen einem und neun gleichzeitig lernenden Agenten ist. Das findet auf dem Konfigurationslevel ohne Wände statt. Die Einstellungen für das Lernen werden so getroffen, dass sich die Levelkonfiguration, und damit der Level, nicht über den Lernvorgang ändert. Innerhalb von 10^5 Schritten lernen beide Agenten ihre Aufgabe und werden danach verglichen.
Im zweiten Experiment durchläuft der erste Agent das normale Curriculum, wobei für den zweiten Agent die Konfiguration (Tabelle 4.1) mit dem Beginn des Lernens auf drei gestellt wird, damit die Karte sofort in ihrer Endfassung vorliegt. Ab hier durchlaufen die Experimente auch die vorher besprochenen $8 \cdot 10^6$ Iterationen. Für Experiment 3 wird dann der Positions-Agent entweder mit dem Curriculum- oder dem normalen Lernansatz gelernt, was von dem Ergebnis des Vorangegangen abhängt. Der bessere Agent wird unter den gleichen Einstellungen im letzten Test mit verschiedenen Rewards verglichen.
Nach der Vorbereitung kann mit der Durchführung und anschließend der Auswertung begonnen werden. Die dabei entstehende Konfiguration, die für die Leveltests benutzt wird, und mögliche weitere Änderungen werden dort zusätzlich erklärt.

5.1.2 Durchführung

Die Experimente werden auf einem MSI GE60 Laptop mit einem Intel i7-4700MQ Prozessor, einer Nvidia Geforce GTX 765M Grafikkarte mit 2GB GDDR5-Speicher und 8GB RAM wie in 5.1.1 erklärt durchgeführt. Parallel zum Lernen werden andere Aufgaben bezüglich dieser Arbeit erledigt, weshalb es starken Einfluss auf die Zeiten haben kann, die der PC braucht, um einen Trainingsvorgang abzuschließen, weshalb die Zeit als Vergleichsmetrik nicht

sehr relevant ist. Nachdem die Modelle fertig gelernt sind, absolvieren sie noch bei den Experimenten 2, 3 und 4 den in 4.3.1 beschriebenen Performancetest, um die Fehlerquote zu ermitteln. In der Auswertung werden dazu die Ergebnisse mit einer Tabelle und jeweils einem Verlaufsdiagramm dargestellt und evaluiert. Bei der Durchführung von Experiment Zwei ist aufgefallen, dass, nachdem der Agent sein Ziel nach den ersten $2 \cdot 10^4$ Iterationen sehr gut finden und erreichen konnte, er damit aufgehört hat und sich eine Zeit lang neben dem Ziel im Kreis gedreht hat. Des wegen wird der Versuch wiederholt und in dem neuen Experiment 2b mit dem ersten verglichen und ausgewertet.

5.1.3 Auswertung

Allgemein

Die fertig gelernten Modelle werden nun im Tensorboard und mit dem in 4.3.1 beschriebenen Performancetest ausgewertet. Tabelle 5.1 dient dabei als kurze Zusammenfassung über die Ergebnisse der einzelnen Konfigurationsexperimente.

Tabelle 5.1: Ergebnisse der Konfigurationstests

Exp.	Versuch	Dauer	End-Reward	Fehlerrate
1	1 Feld	7m 24s	0,9831	
	9 Felder	12m 25s	0,9915	
2	Curriculum	19h 14m 52s	0,9899	0,03118
	Normal	20h 17m 2s	0,9936	0,0307
2b	Curriculum	19h 14m 52	0,9899	0,03118
	2.Versuch	20h 50m 52s	0,9874	0,0133
3	Position	17h 51m 18s	-0,0216	0,3118
	Rayperception	19h 14m 52s	0,9899	0,54014
4	Negativ	20h 2m 38s	0,9415	0,0166
	Positiv	20h 50m 52s	0,9874	0,0133

Für die Auswertung werden ab Experiment 2 jeweils 2 Graphen Diagramme benutzt. Das Erste ist die kommulative Reward über den gesammten Lernverlauf und das Zweite nur die ersten $5 \cdot 10^5$ Iterationen zur besseren Erkennbarkeit des Anfangs. Mithilfe des Tensorboards werden diese Diagramme erzeugt, die

aber um den Faktor 0,6 geglättet sind, um das Ergebnis besser darstellen zu können. Die ungeglätteten Versionen sind in transparenter Form in schwächerer Farbe zusätzlich im Diagramm sichtbar. Für die Auswertung werden die ungeglätteten Werte benutzt. Da es sich beim Lernen um einen Prozess handelt, bei dem der Zufall, vor allem am Anfang, eine große Rolle spielt, kann es zu starken statistischen Schwankungen kommen. Zusätzlich wurde der Laptop während der Experimente weiterhin für andere Aufgaben genutzt, weshalb die Lernzeiten stark variieren können, je nachdem wann der PC den Großteil der Zeit gelernt hat.

Experiment 1: Feldanzahl

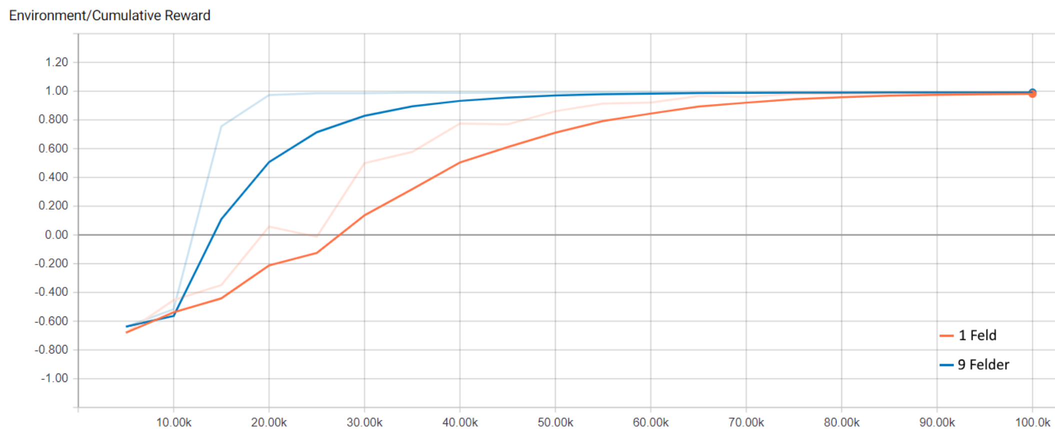

Abbildung 5.1: Experiment 1 Ergebnis: Mittlere Reward über die Anzahl der Iterationen

Der Neun-Feldagent hat seine Aufgabe mehr als dreimal so schnell gelernt.

In Abbildung 5.1 ist Erkennbar, dass das Modell mit neun Agenten auf neun Feldern bedeutend schneller lernt und schon bei etwa $2 \cdot 10^4$ Schritten eine durchschnittliche Reward von 1 hat und damit mit der simplen Aufgabe, sein Ziel zu erreichen, fertig ist. Der Agent, der nur auf einem Feld lernt, hat im Gegensatz dazu erst bei circa $7,5 \cdot 10^4$ Schritten die Aufgabe gelernt. Da die Berechnung von mehreren gleichzeitig lernenden Agenten mehr Ressourcen benötigt, hat der Versuch doppelt so lange gedauert, was zu großen Teilen an der benutzten Hardware liegt. Da die Experimente recht kurz waren, wurde nebenbei der PC nicht weiter benutzt, weshalb die Zeiten hier als relevante Metrik benutzt werden kann. Wenn dieser Zeitunterschied zwischen dem Lernen der

Aufgaben der beiden Agenten und die maximal benötigten Iterationen zurückgerechnet werden, braucht der Agent mit 9 Feldern lediglich 2m 29s, um seine Aufgabe zu lernen, und der mit einem Feld 5m 33s. Das erfüllt die Hypothese, dass der Agent ein vielfachen schneller sein kann, weshalb die Agenten in den nachfolgenden Experimenten auf jeweils 9 Feldern gleichzeitig gelernt werden.

Experiment 2: Curriculum vs. Normal

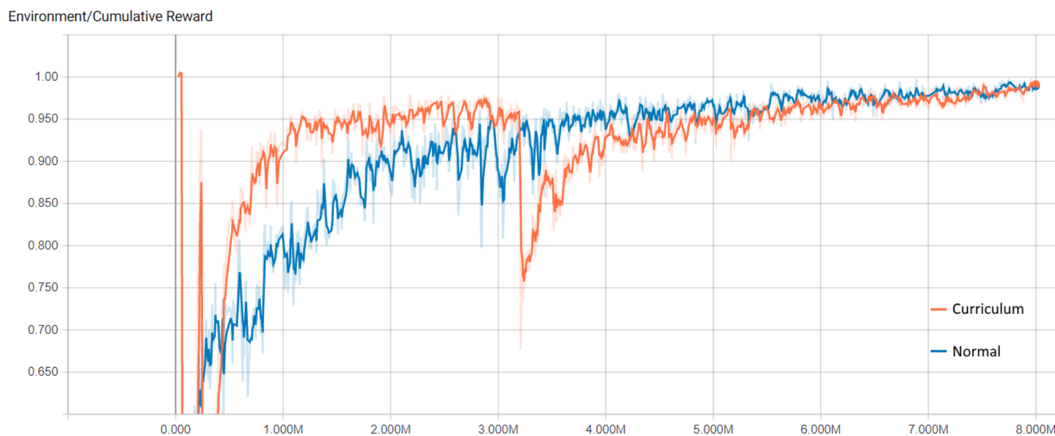

Abbildung 5.2: Experiment 2 Ergebnis: Mittlere Reward über die Anzahl der Iterationen

Beide Ansätze gelangen zu vergleichbaren Ergebnissen, wobei einzelne Lernfortschritte besser innerhalb des Curriculum-Agenten erkennbar waren.

Im Experiment 2 fällt in Abbildung 5.3 auf, dass am Anfang der Curriculum-Agent innerhalb von $2 \cdot 10^4$ Iterationen seine Aufgabe bereits fertig gelernt hat, wobei die Reward des Normalagenten bei -0,83 liegt. Das liegt daran, dass er sofort im fertig aufgebauten Level lernen muss und seine Chancen, das Ziel zufällig zu erreichen, deshalb sehr gering sind. Da der Curriculum-Agent, der in der ersten Levelkonfiguration ist, direkt neben seinem Ziel spawned, sind seine Chancen extrem hoch, das Ziel zu erreichen. Dadurch erreicht er sehr früh die Maximal-Reward und kann seine Policy zielführend aktualisieren und so seine Aufgabe schnell lernen. Trotzdem hat der Curriculum-Agent seinen ersten großen Reward-Einbruch bei $5,5 \cdot 10^4$ Iterationen.

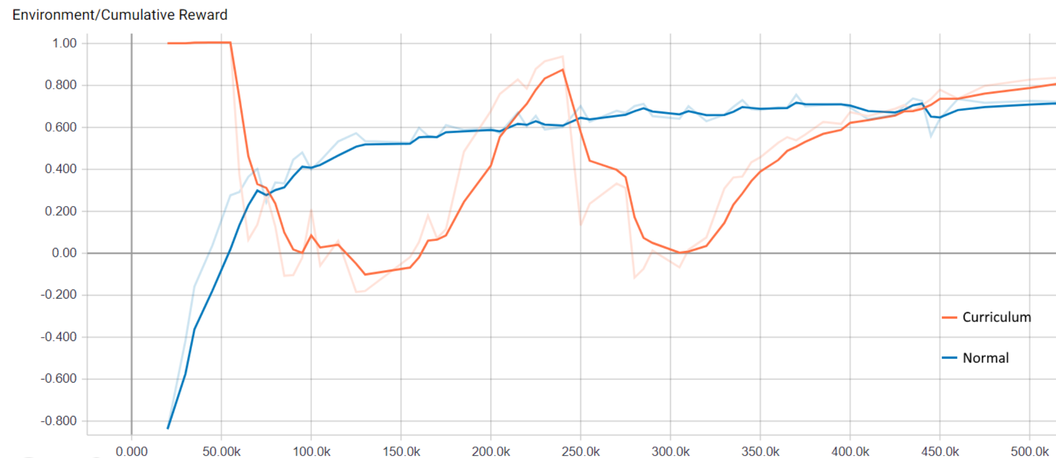

Abbildung 5.3: Experiment 2, die ersten $5 \cdot 10^5$ Iterationen Result: Cumulative Reward Nr. of Runs
Durch die vielen Levelveränderungen schwankt die Reward des Curriculum-Agenten sehr stark, erreicht aber nach kurzer Zeit wieder einen höheren Wert als die des Agenten der ohne Curriculum trainiert, die wie erwartet, stetig steigt.

Innerhalb des Experiments stellte sich heraus, dass der Agent zu dem genannten Zeitpunkt nicht weiter auf sein Ziel zusteuert sondern kurz vor ihm stehen bleibt, was nicht passieren sollte, da seine mittlere Reward bei 1 lag und sie dadurch nur schlechter wurde. Des wegen wird der Lernversuch "Curriculum-Lernen" mit positiven Rewards und 9 Sichtstrahlen als Experiment 2b wiederholt, um herauszufinden, ob dieser Reward-Verlust immer auftritt. Dieser angesprochene Reward-Verlust aus Experiment 2, der zusätzlich durch die Leveländerung bei 10^5 Schritten verstärkt wird, führt zu einem Tiefstand von -0,17 bei $1,3 \cdot 10^5$ wobei der Normalagent bei $8 \cdot 10^4$ bereits das erste Mal den durchschnittlichen Reward seines Konterparts mit 0,33 zu 0,14 übersteigt. In den nächsten Iterationen steigt seine Reward wieder an, bis diese bei $2,4 \cdot 10^5$ wieder stark einbricht, da hier die ersten Wände im Level erscheinen. Durch diese einzelnen Leveländerungen schwankt der Curriculum-Agent innerhalb der ersten $3,5 \cdot 10^5$ Durchläufe auch sehr stark.
Von diesen Konfigurationsänderungen ist der Normalagent nicht betroffen und er erreicht bei $2,4 \cdot 10^5$ Durchläufen bereits 0,6. Ab seinem zweiten Tiefpunkt bei $3,5 \cdot 10^5$, mit einer Reward von -0,06, steigt diese beim Curriculum-Agent bis $3,2 \cdot 10^6$ Iterationen wieder stark an, sodass sie bei $4,4 \cdot 10^5$ Schritten den Normal-Agenten wieder überholt und beim letzten Konfigurationswechsel

bei $3,2 \cdot 10^6$ einen Reward-Höhepunkt von 0,96 hat. Bei diesem Konfigurationswechsel wird der finale Level hergestellt, wodurch abermals ein Reward-Einbruch des Curriculum-Agenten eintritt. Dieser ist diesmal mit einem Tiefpunkt von 0,75 bei $3,24 \cdot 10^6$ aber nicht annähernd so stark wie beim Hinzufügen der ersten Wände. Zu diesem Zeitpunkt hat der Normalagent eine durchschnittliche Reward von 0,9349. Ab diesen Punkt steigen beide Graphen weiter bis sie bei $8 \cdot 10^6$ Durchläufen das Ende des Versuchs erreichen mit 0,9913 für Curriculum-Lernen und 0,9936 für das normale Lernen.
Dabei ist erkennbar, dass die Reward-Steigung des Curriculum-Agenten von circa $5 \cdot 10^6$ bis $8 \cdot 10^6$ nach wie vor stärker ist. Laut den Hypothesen sollte das auch immer der Fall sein, dass der Reward-Anstieg bedeutend höher sein soll als beim Normal-Agenten. Deshalb besteht die Möglichkeit, dass die Rewards sich innerhalb der nächsten Iterationen weiter annähern oder die Curriculum-Reward die Normal-Reward sogar übertreffen kann.
Innerhalb des Performancetests ist dieser Unterschied zwar erkennbar aber für das generelle Problem zu vernachlässigen, da diese Werte beide innerhalb der statistischen Schwankungen liegen, die innerhalb des Performancetests auftreten können. Der Normal-Agent hat bei den angesetzten 10^6 Durchläufen eine Fehlerquote von 3,07% und Curriculum von 3,11%. Damit hat Curriculum innerhalb von $2 \cdot 10^4$ Versuchen einen Fehler mehr gemacht als Normal. Dadurch wird auch diese Hypothese erfüllt, dass beide Ansätze die gleiche Fehlerquote und End-Reward haben.
Für die Konfigurationstest ist dieser Unterschied nicht relevant, weswegen für die weiteren Tests Curriculum-Lernen genutzt wird, da bereits viel früher Lernergebnisse erkennbar sind und Fehler und Probleme innerhalb des Lernens besser eingegrenzt werden können. Ein langsamer Levelaufbau ist wahrscheinlich zusätzlich für die später stattfindenden Leveltests geeigneter, da bei schwereren Leveln die Lernkurve des Normal-Agenten nur weiter abflachen würde. Bei einem zu schweren Level würde der Agent sein Ziel gar nicht mehr erreichen und wäre so nie in der Lage, seine Aufgabe zu lernen.

Experiment 2b: Wiederholung Curriculum

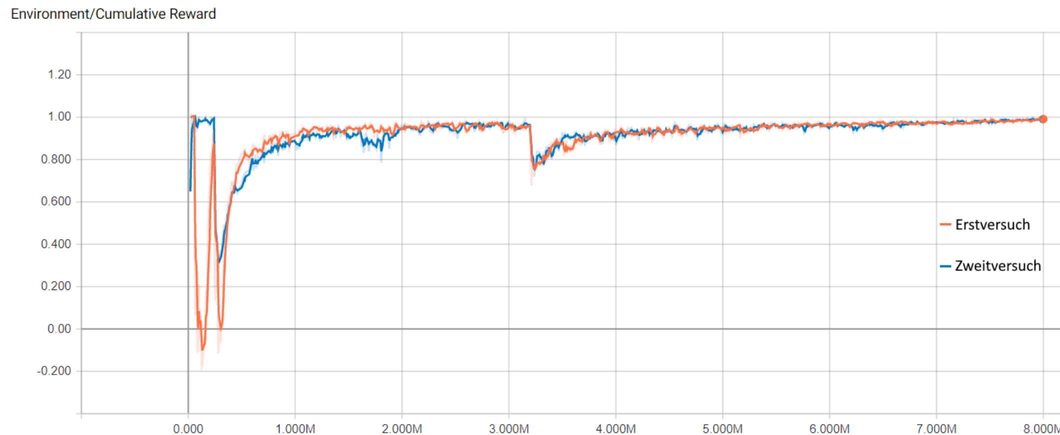

Abbildung 5.4: Experiment 2b Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Ab etwa $2 \cdot 10^6$ Iterationen liegen zwei nahezu identische Graphen vor

Nach der Wiederholung des Curriclum-Lernens in Experiment 2 sind die Ergebnisse eindeutiger. Im Anfang der Lernphase der beiden Versuche in Abbildung 5.5 ist ein klarer Unterschied zu erkennen. Der zweite Versuch hat wie erwartet keinen sehr hohen scheinbar zufälligen Reward-Verlust bei $5,5 \cdot 10^4$, ist allerdings auch etwas langsamer beim Lernen der ersten Konfiguration und braucht anstatt den $2 \cdot 10^4$ Schritten des ersten Versuchs $3 \cdot 10^4$ Iterationen. Trotzdem liegt die Reward des Zweitversuchs bei $2,4 \cdot 10^5$ bedeutend höher als beim Erstversuch mit 0,999 zu 0,9382. Beim anschließenden Konfigurationswechsel mit dem Spawn der Wände fällt die Reward des Erstversuchs tiefer als die des Zweitversuchs mit 0,1331 zu 0,2002.
Diese starke Schwankung der beiden Modelle innerhalb der ersten $2,5 \cdot 10^5$ Schritte liegt am generellen in den Grundlagen besprochenen Problem, dass die Policy falsch aktualisiert werden kann. Zwar hat PPO einige Mechanismen, dieses Problem zu minimieren, aber offensichtlich funktioniert dies nicht immer. Außerdem, soll verhindert werden, dass das Modell sich in ein lokales Maximum trainiert. Deshalb werden andere Steuerungsansätze verfolgt, weshalb die hohe Reward abnimmt. Diese Funktion wird im PPO- oder ML-Agents-Paper nie erwähnt, weshalb dieser Grund sehr unwahrscheinlich ist. Ein weiteres Argument könnte sein, dass das Lernen durch eine ungeeignete Lern-Konfigurationsdatei an dieser Stelle instabil wird. In dieser kann die gewählte Lernrate zu hoch sein was zu diesem Verhalten führen kann[8].

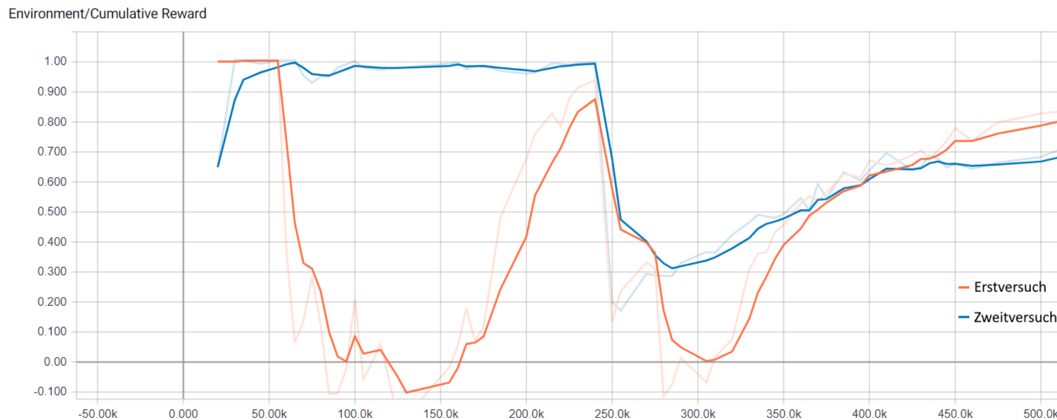

Abbildung 5.5: Experiment 2b, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Der stark unterschiedliche Anfang kommt durch die zufällige Exploration mit fehlerhaften Policy Updates und instabilem Lernen zustande.

Deswegen wird in den nächsten Lernversuchen die Lernrate auf die in [8] empfohlenen $1e^{-5}$ gesetzt. Im Verlauf der Iterationen kann der Reward-Einbruch vom Algorithmus allerdings wieder entfernt werden. Des wegen ist in Abbildung 5.4 zu erkennen, dass die beiden Graphen nach dieser Anfangsphase schnell nahezu identisch sind bis auf einen Ausreißer bei $1,7 \cdot 10^6$, wo die Reward des Zweitversuchs kurzzeitig fällt.
Das Endergebnis bei $8 \cdot 10^6$ Iterationen ist 0,9899 für den Erstversuch und 0,9874 für den Zweitversuch, was an den natürlichen Schwankungen des Lernens liegt.
Trotz des sehr ähnlichen Reward-Werts am Ende, fallen die Fehlerraten mit ca 3% zu 1,3% sehr unterschiedlich aus. Zwar liegt dies teilweise an den statistischen Schwankungen des Performancetests, allerdings sollten so starke Unterschiede innerhalb des Versuchs nicht entstehen, da es sich um den selben Versuch mit den gleichen Einstellungen und Vorraussetzungen handelt. Diese Werte blieben bei bei der Wiederholung des Performancetests allerdings sehr ähnlich mit 2,8% für den Erstversuch und 1,5% für den Zweitversuch. Für die generellen Konfigurationsergebnisse ist allerdings unerheblich, ob die Fehlerrate leicht schwankt, da ein genaues Ergebnis erst beim Nortivag relevant ist, solange die Unterschiede in den noch anstehenden Konfigurations- und Leveltests ausreichend zu erkennen sind. Trotzdem können auch Lernversuche, die am Anfang negative Updates machen, am Ende gute Ergebnisse erzielen. In-

nerhalb der nächsten Experimente sind diese unterschiedlichen anfänglichen Lernverläufe immer wieder zu beobachten.

Experiment 3: Sichtstrahlen vs. Position

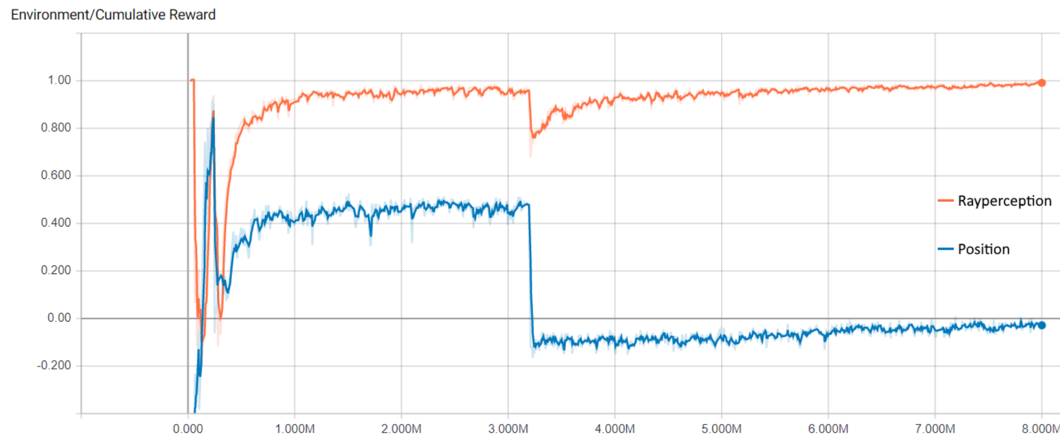

Abbildung 5.6: Experiment 3 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Der Positionsansatz schneidet klar schlechter ab als die Sichtstrahlenwahrnehmung.

Wie schon in den letzten Versuchen und in Abbildung 5.6 und genauer in 5.7 erkennbar, ist der Strahlagent (vorher Curriculum-Agent) bereits bei $2 \cdot 10^4$ Iterationen mit seiner Aufgabe fertig, wobei die Reward des Positions-Ansatzes bei -0,57 ist. Das könnte daran liegen, dass noch keine weiteren Objekte auf der Karte erschienen sind, der Strahlagent demnach nur sein Ziel sehen kann, der Positionsagent allerdings bereits hier ein Array mit einer Menge an Positionen bekommt. Bei $5,5 \cdot 10^4$ Iterationen kommt wieder der erste Einbruch, dessen Ursache in Experiment 2b geklärt wurde, bei dem die Reward des Strahlagenten bis $1,3 \cdot 10^5$ auf -0,18 sinkt. Jedoch hat die Lernratenänderung nicht den gewünschten Effekt. Die des anderen Ansatzes steigt kontinuierlich, außer einem kleinen Verlust bei 10^5, wo wieder der Maximalabstand erhöht wird, und überholt ihn an seinem Tiefpunkt mit einer eigenen Reward von 0,1. Trotzdem übersteigt der Strahlenansatz bei $2,2 \cdot 10^5$ seinen Kontrahenten wieder und beide haben bis $2,4 \cdot 10^5$ ihren nächsten Höhepunkt mit 0,9 für den Positionsagenten und 0,94 für den Strahlagenten. Durch die nun hinzugefügten Mauern sinken die Graphen ab $2,4 \cdot 10^5$ wieder.

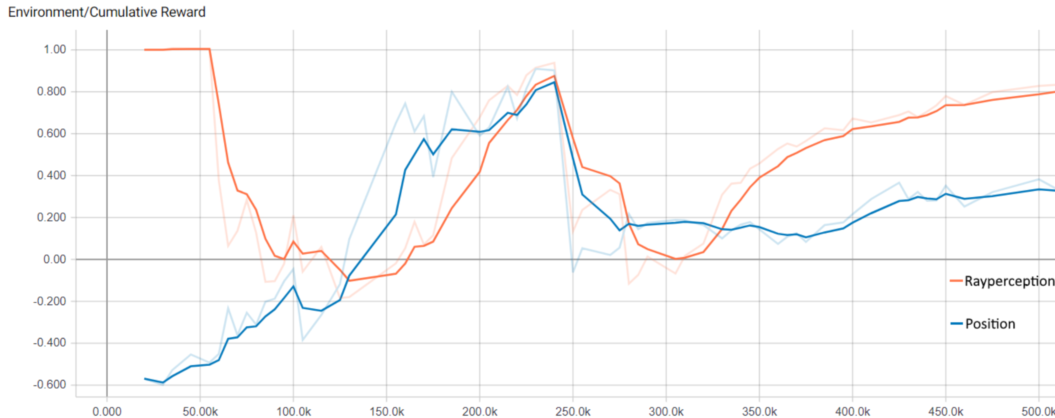

Abbildung 5.7: Experiment 3, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Der Positionsansatz schneidet klar schlechter ab als die Sichtstrahlenobservation.

Die Reward des Strahlenansatzes geht dabei mit -0,07 weiter runter als der des Positionsansatzens mit 0,1, allerdings steigt er auch ziemlich schnell wieder an und erreicht bei 10^6 Iterationen wieder die 0,9 Reward, wobei der Anstieg des Positionsansatzes bei 10^6 mit 0,45 gegen null geht.
Diese Reward-Stagnation liegt daran, dass der Agent nur mithilfe der Position nichts über die Ausmaße der Wände weiß und er so für jede Wand die Ausmaße lernen müsste. Bis zur letzten Levelveränderung bei $3,2 \cdot 10^6$ erreicht der Strahlenansatz 0,96 und der Positionsansatz 0,47. Daraufhin sinkt die Reward beim Strahlenansatz um 0,29 auf 0,67 und die des Postionsansatzes um 0,63 auf -0,16, was über das Doppelte des Verlusts vom anderen Agenten ist. Innerhalb der nächsten $4,8 \cdot 10^6$ Schritte gelingt es dem Positionsansatz auch nicht mehr, über die Null-Reward zu kommen und er endet mit -0,02, wohingegen der Strahlenansatz bei 0,9913 endet. Das spiegelt sich auch an den Fehlerquoten wieder, denn der Positionsansatz schafft nicht jeden zweiten Durchlauf richtig zu lösen mit 54% und der Strahlansatz hat seine schon aus den letzten Experimenten bekannten 3%.
Aus dem Experiment geht hervor, dass $8 \cdot 10^6$ Iterationen nicht ausreichend sind, um dem Positionsagenten vernünftiges Wandverhalten beizubringen, wenn ihm nur die Positionsdaten gegeben sind. Zumal die Implementation viel schwieriger ist, da jedes Mal wenn sich die Menge der Wände ändert, die Menge der Observationen des Gehirns geändert werden muss. Zusätzlich wird die zu observierende Menge an Daten extrem schnell sehr hoch, denn im Vergleich

zu 45 Variablen des 9 Sichtstrahlenansatzes bekam der Positionsansatz pro Wand drei Variablen für die jeweilige Axenposition. Das in der Testumgebung vorhandene Kreuz besteht aus zwei einzelnen Objekten, was letztendlich zu 32 übegebenen Variablen führt. Mit diesem sehr eindeutigen Ergebnis wird in den folgenden Experimenten immer der Sichtstrahlenansatz genutzt, was auch schon in den Hypothesen vermutet wurde. Innerhalb der Leveltests werden deshalb auch die Unterschiede zwischen 5, 9 und 18 Strahlen verglichen.

Experiment 4: Positiv vs. Negativ

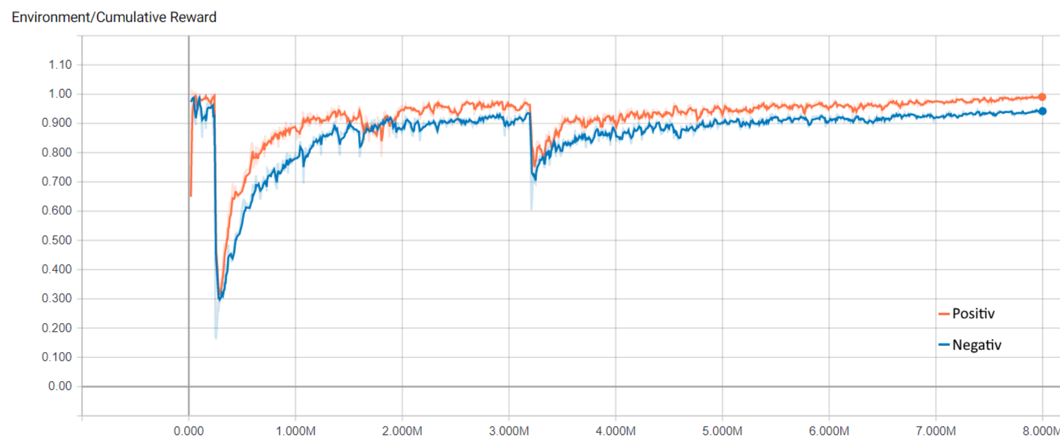

Abbildung 5.8: Experiment 4 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Obwohl beide Graphen sich sehr ähnlich sind, ist die Reward vom positiven Ansatz mit einer Ausnahme bei $1,7 \cdot 10^6$ Iterationen konstant höher.

Da der negative Ansatz hier den aus Experiment 2 bekannten Einbruch nicht besitzt, wird als Vergleichswert der neugelernte und performancetechnisch bessere Ansatz benutzt. Dadurch ist der Anfang in Abbildung 5.9 auch sehr ähnlich. Es ist aber zu sehen, dass der negative Ansatz bedeutend schneller mit dem ersten Lernen in der ersten Konfiguration ist, wofür der Positiv-Agent länger braucht. Allerdings überholt der Positiv-Agent auch die Reward-Werte des Negativen bei $5 \cdot 10^4$. Der Konfigurationswechsel bei 10^5 hat auf den orangefarbenen Graphen nur wenig Einfluss, wobei der blaue auf 0,85 fällt. Trotzdem sind beide Reward-Graphen sehr stabil bis zur Leveländerung mit den ersten Wänden bei $2,4 \cdot 10^5$. Hier fällt der Positiv-Agent von 0,999 auf 0,1712 und sein Konterpart von 0,9431 auf 0,1621.

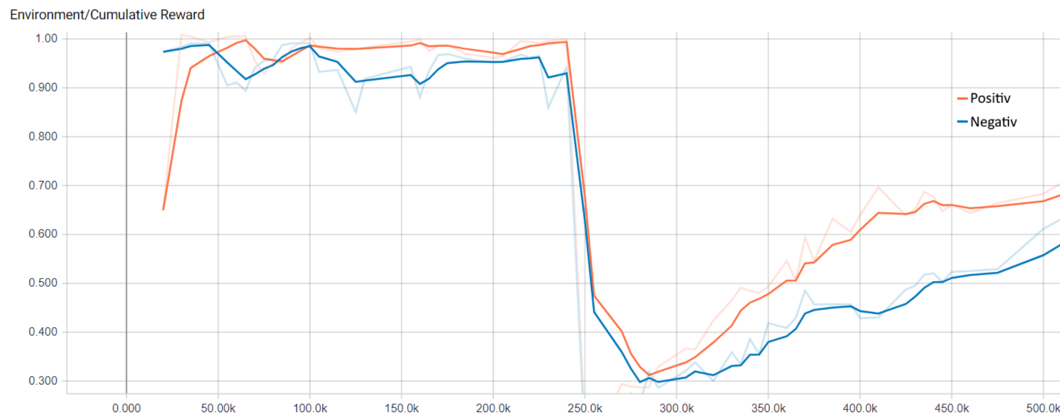

Abbildung 5.9: Experiment 4, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Der Negativ-Agent ist bereits bei $2 \cdot 10^4$ mit dem Lernen seiner ersten Aufgabe fertig, wogegen der Positiv-Agent $3 \cdot 10^4$ Schritte braucht. Allerdings ist Letzterer schneller mit dem Lernen nach dem Spawnen der Wände bei $2,4 \cdot 10^5$.

Jedoch hat der Erste daraufhin einen viel stärkeren Anstieg und erreicht schon bei $1,05 \cdot 10^6$ Iterationen wieder die 0,9-Marke, die der Negativ-Agent erst bei $1,785 \cdot 10^6$ erreicht. Im Gegensatz zum positiven hat der negative Agent bei $1,7 \cdot 10^6$ keinen Reward-Einbruch, bleibt aber trotzdem konstant mit seiner Reward unter dem positiven Ansatz. Bei $3,2 \cdot 10^6$ fällt der Reward-Wert beider Graphen von 0,9624 auf 0,7525 für Orange und von 0,9325 auf 0,6222 für Blau, wovon sich der Erste wieder bedeutend besser erholt. Ab diesem Punkt erreicht die Reward des Negativ-Agenten auch nie wieder die des positiven. Die Reward am Ende von 0,9874 für den Positiv-Agenten und 0,9415 für den Negativ-Agenten spiegeln sich allerdings nicht in ihren Fehlerquoten wider. Mit 0,0133 beim positiven Ansatz zu 0,0166 beim negativen sind beide nahezu identisch, wenn die statistischen Schwankungen in Betracht gezogen werden.
Diese Quote war auch zu erwarten und bestätigt die Hypothese, auch wenn die Graphen dies nicht zu 100% widerspiegeln. Für die Levelexperimente wird deshalb der positive Ansatz benutzt.

Zusammenfassung

Innerhalb der Experimente ist für die Leveltests eine Konfiguration für den Agent herausgekommen. Die sieht vor, dass er:

- falls in Nortivag möglich, auf mehreren Feldern gleichzeitig lernt
- Curriculum-Lernen benutzt
- 5, 9 oder 18 Sichtstrahlen benutzt
- den positiven Reward-Ansatz benutzt.

Grundsätzlich waren die Ergebnisse sehr zufriedenstellend, weshalb für die Leveltests auch weiterhin die leicht abgeänderte Standard-Konfigurationsdatei und Standard-Academy benutzt werden.

5.2 Leveltests in Nortivag Lite

5.2.1 Vorbereitung

Mit der Ergebniskonfiguration aus den Konfigurationstests 5.1 können die Leveltests durchgeführt werden. Die genutzte Konfiguration ist, dass sich ein Agent mit einem positiven Reward-Ansatz in einem Level bewegt, der 9 mal dupliziert vorliegt. Die Karte nimmt er mit 5, 9 oder 18 Sichtstrahlen wahr und diese wird mit Curriculum-Lernen über den Lernverlauf an den Stellen 10^5, $2,4 \cdot 10^5$ und $3,2 \cdot 10^6$ verändert. In Vorbereitung dazu wurden die passenden Level erstellt welche in Abbildung 4.3 zu sehen sind, und ihre Curricula so angepasst, dass sie die Konfigurationen aus Abbildung 5.10 ablaufen. Die Reihenfolge der Experimente lautet wie folgt: Es wird zuerst der einfache Level dreimal mit den festgelegten verschiedenen Sichtstrahlen trainiert wird, danach Mittel und zum Schluss der schwere Level.

5.2.2 Durchführung

Die Durchführung der Levelexperimente geschieht auf dem gleichen Laptop, wie er schon in 5.1 benutzt wurde. Im Anschluss an alle Tests wird an den neun gelernten Modellen noch der gleiche Performancetest durchgeführt, um die Fehlerquote zu ermitteln.

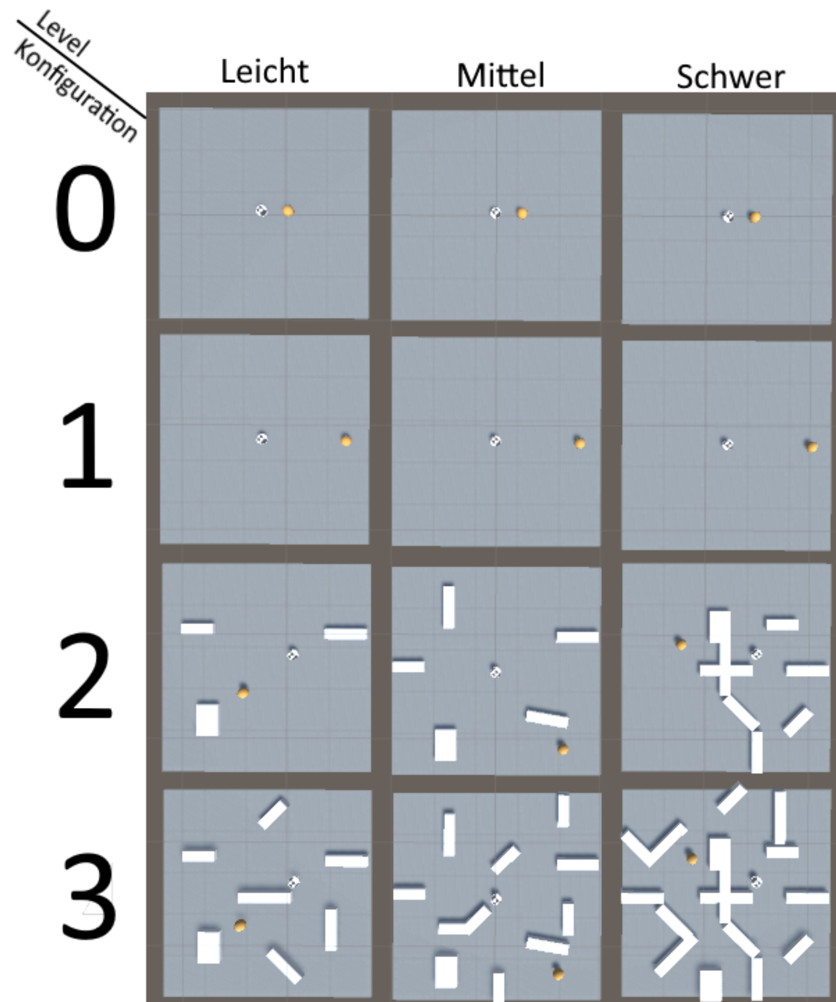

Abbildung 5.10: Konfigurationen der einzelnen Levelzustände
In der Abbildung sind die verschiedene Levelzustände basierend auf ihren Curriculum-Konfigurationen aus Tabelle 4.1

5.2.3 Auswertung

Allgemein

Wie schon in Kapitel 5.1 werden die fertigen Modelle in Tensorboard ausgewertet und zusätzlich die Ergebnistabelle 5.2 aufgestellt. Für die Auswertung der Graphen wurden sie, wie auch in den vorherigen Tests, mit einem Faktor von 0,6 geglättet, um eine bessere Lesbarkeit zu gewährleisten. Die Reward-Werte aus ihnen werden immer in Tripeln (5;9;18) der Strahlenanzahl geschrieben. Da die Level in den ersten $2,4 \cdot 10^5$ Iterationen gleich sind, sollte der Graphenanfang der drei Agenten auch gleich sein. Deshalb wird der Anfang bist dorthin nur einmal im einfachen Level genau beschrieben und bei den anderen Experimenten wird nur noch auf die Besonderheiten eingegangen, die in dieser Zeit aufgetreten sind.

Tabelle 5.2: Ergebnisse der Leveltests

Exp.	Anzahl der Strahlen	Dauer	End-Reward	Fehlerrate
Leichter	5	18h 24m 32s	0,933	0,02729
Level	9	19h 23m 20s	0,9712	0,0179
	18	20h 39m 53s	0,9733	0,0148
Mittlerer	5	20h 21m 49s	0,9524	0,02925
Level	9	20h 16m 48s	0,9823	0,02876
	18	22h 15m 38s	0,997	0,01879
Schwerer	5	21h 12m 47s	0,4119	0,8504
Level	9	21h 18m 19s	0,5464	0,6943
	18	24h 25m 2s	0,7263	0,66308

Experiment 5: Einfach

In den Hypothesen aus Tabelle 4.3 wurde davon ausgegangen, dass alle Agenten ähnlich gut abschneiden und ihre Fehlerquote mit kleiner als 5% sehr gering im Vergleich zu den noch kommenden Experimenten ist. Alle drei Modelle haben diese Hypothesen erfüllt, indem sie klar im Rahmen der Erwartung lagen. Der Fünfstrahl-Agent schnitt erwarteter Weise als mit der schlechtesten Fehlerquote von 2,7% ab. Wie zu erwarten, ist der Verlauf der Graphen aus Abbildung 5.11 insgesamt sehr ähnlich.

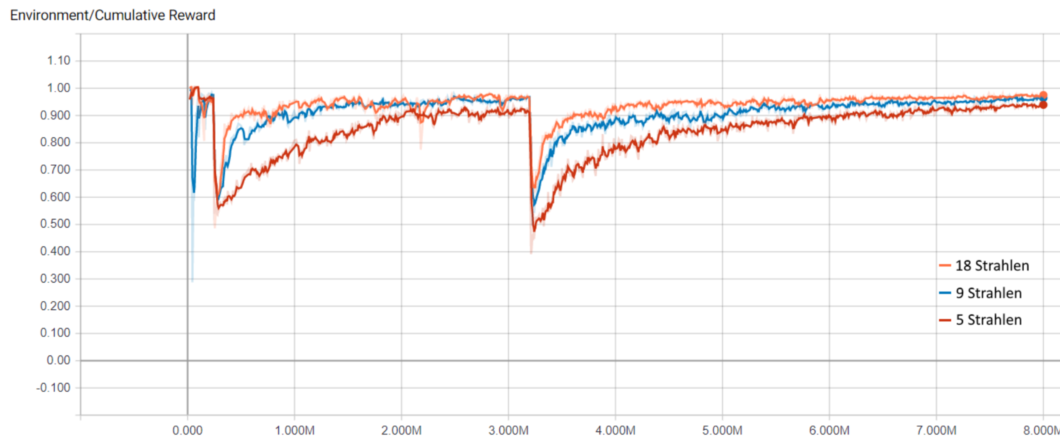

Abbildung 5.11: Experiment 5 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
In der Abbildung ist erkennbar, dass Agenten mit mehr Sichtstrahlen einen stärkeren Anstieg hatben als Agenten mit weniger. Trotzdem befinden sich der Neuner- und Achtzehner-Agent auf dem gleichen Niveau.

Unterschiedlich sind nur die Reward-Höhen und der Anfang. In Abbildung 5.12 ist der bereits aus den Konfigrationsexperimenten bekannte Reward-Verlust beim Neunstrahlagenten (Blau) klar zu erkennen. Zeitgleich verlaufen die Graphen der anderen beiden Agenten ohne diese Schwankung viel stabiler, bis alle drei bei 10^5 einen Reward-Verlust haben. Durch den Zufallsverlust vom Neuner-Ansatz fällt der Graph auch am stärksten, und es sind folgende Minimalwerte zu verzeichnen (0,9317; 0,8198; 0,8937). Dabei fällt auf, dass der Fünfer-Agent (Rot) den kleinsten Verlust hat, was daran liegen könnte, dass er mit weniger Observationsvariablen nicht so viel neu lernen muss, wie die anderen, aber der Verlust kann auch in den Lernschwankungen liegen. Bis $2,4 \cdot 10^5$ Iterationen erreichen alle drei Graphen aber wieder gemeinsam ihr Maximum mit den Werten (0,9619; 0,9747; 0,9760). Nachdem die Wände hinzugefügt wurden, sinken die Rewards wieder. Es fällt auf, dass je mehr Strahlen der Agent hat, desto stärker fällt sie auf jeweilige Minima von (0,5543; 0,5315; 0,486) bei $2,5 \cdot 10^5$.
Der Achtzehner-Agent (Orange) hat eine Reward von mindestens 0,9 bei $4,6 \cdot 10^5$ erreicht, der Neuneragent bei $9,1 \cdot 10^5$ und der Fünfer-Agent erst bei $1,83 \cdot 10^6$. Bei diesen Zahlen wird klar, dass wenn der Agent mehr Strahlen hat, er sich um so besser in der Welt zurechtfinden und bewegen kann. Mit dem Wechsel der Konfiguration zum finalen Level erreichen die Agenten die Werte (0,9016; 0,97; 0,9629).

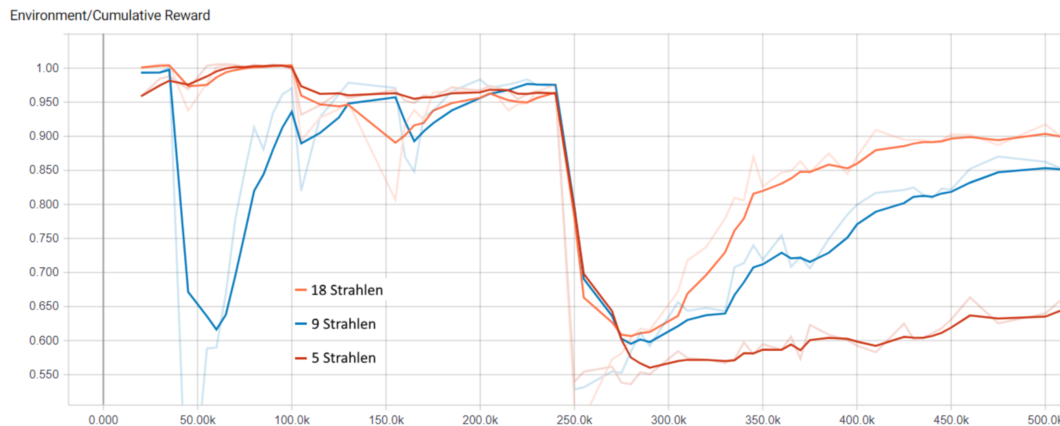

Abbildung 5.12: Experiment 5, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Obwohl der Neuner-Agent am Anfang instabil lernt, kann er das gleiche Reward-Niveau wie die anderen beiden Agenten erreichen. Der große Unterschied der Drei ist der Anstieg nach der $2,4 \cdot 10^5$ Iteration.

Bei dem folgenden Reward-Verlust geschieht das Gleiche wie beim Hinzufügen der Wände und die Rewards sind auf ihrem Tiefpunkt (0,3911; 0,5245; 0,5389) bei $3,21 \cdot 10^6$ Durchläufen, wobei sich der Achtzehner-Agent wieder am schnellsten erholt und schon wieder bei $3,61 \cdot 10^6$ über der 0,9 Reward-Marke liegt. Der Agent mit den 9 Strahlen braucht dafür bis $4,3 \cdot 10^6$ und der Fünfer-Agent noch bis $6 \cdot 10^6$. Im weiteren Verlauf kreuzen sich die Graphen auch nicht mehr und am Ende liegen die Reward-Werte bei (0,933; 0,9712; 0,9733) wobei die letzten beiden Agenten gleichauf liegen und lediglich der Fünfer-Agent merklich schlechter abschneidet.
Der Unterschied zwischen 9 und 18 Strahlen ist dagegen nicht signifikant, obwohl die Anzahl der Strahlen und den dadurch entstehenden Observationspace-Unterschied so verschieden ist. Die Graphen unterscheiden sich, bis auf den Anfang, nur in ihrem Anstieg nach den jeweiligen Konfigurationswechseln.
Das liegt daran, dass die Agenten mit dem Hinzufügen der Wände für jeden einzelnen Strahl erst die Bedeutung der Abstandsmessungen zu den Wänden lernen müssen und auch, dass es ein Fehler ist, wenn sie die Wand berühren. Trotzdem steigt die Reward der Agenten wieder schneller an, wenn sie mehr Strahlen haben.
Am Ende ist die große Abweichung innerhalb der einzelnen Performancetests entstanden, denn die Fehlerquoten des Neuner- und Achtzehner-Strahlagenten sind gleich und unterscheiden sich nur sehr wenig vom Fünfer-Agenten. Das

hat den Grund, dass beim einfachen Level nicht so viele Wände existieren, und diese nur kleine Quader mit viel Freiraum dazwischen sind. Deshalb ist eine so genaue Abtastung des Levels mit 18-Sichstrahlen nicht notwendig und der zusätzliche Gewinn an Rauminformationen gegenüber den neun Strahlen nicht entscheidend. Allerdings hatten alle drei bis zur $8 \cdot 10^6$ Iteration einen positiven Anstieg, weshalb es wahrscheinlich ist, dass sie sich mit noch mehr Lernzeit weiter verbessern könnten. Des wegen geht der Neuner-Agent auch als beste Alternative aus dem Experiment hervor.

Experiment 6: Medium

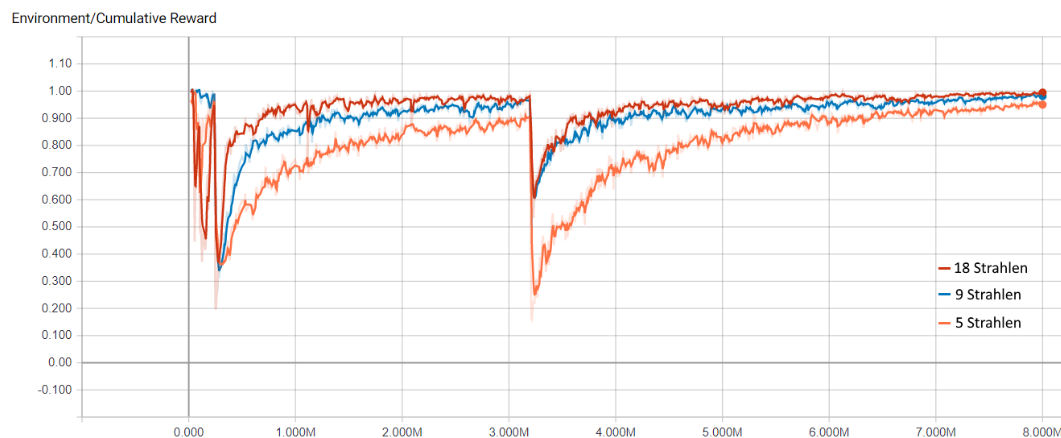

Abbildung 5.13: Experiment 6 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Die Schwierigkeit des Levels wirkt sich auf die Rewardschwankung aus, indem dieser bei den Konfigurationsänderungen stärker fält als beim einfachen Level 5.11.

Der Verlauf der Graphen im Mediumlevel aus Abbildung 5.13 ist sehr ähnlich zu dem aus dem einfachen Level. Allerdings fällt direkt am Start des Experiments, der in Abbildung 5.14 dargestellt ist, auf, dass diesmal im Vergleich zum einfachen Leveltest die beiden Fünf (Orange)- und Achtzehn (Rot)-Strahlagenten den Reward-Verlust haben. Bei der Levelveränderung bei 10^5 tritt diesmal, im Gegensatz zum Neuner-Agent (Blau), auch ein extremer Reward-Verlust der beiden ein, was zu einem Wert von (0,4752; 0,9656; 0,4919) führt, wobei der Achtzehner-Agent danach noch weiter bis 0,3709 fällt.

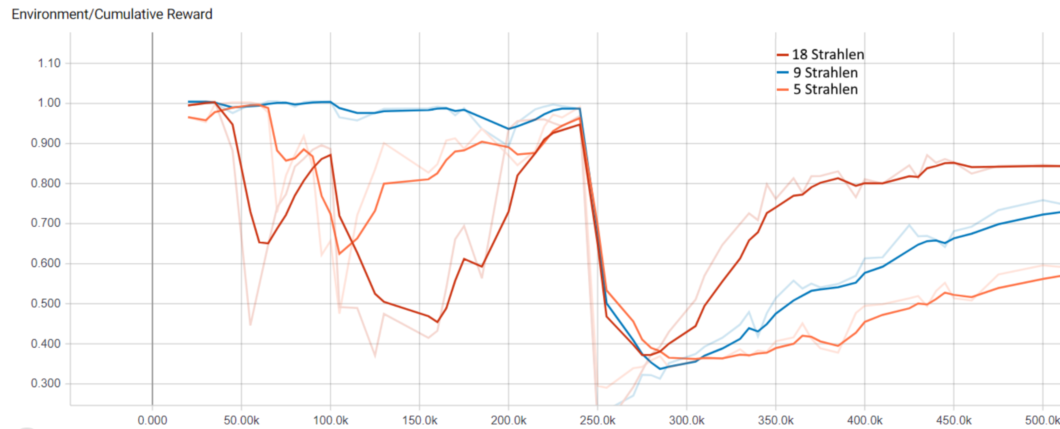

Abbildung 5.14: Experiment 6, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Trotz der beiden instabilen Lernverläufe vom Achtzehner- und Fünfer-Agenten sind die gleichen Ansätze wie in 5.12 erkennbar.

Dieser Verlust ist diesmal bedeutend stärker als beim einfachen Level, wobei bis zum Erscheinen der ersten Wände bei $2,4 \cdot 10^5$ das Lernen der einzelenen Strahlansätze identisch verlaufen sollte. Bis $2,4 \cdot 10^5$ steigen die Rewards der beiden Agenten wieder an und erreichen die Werte von (0,9913; 0,987; 0,9689), der nach dem Hinzufügen der Mauern auf (0,295; 0,2169; 0,202) sinkt. Ab dem Punkt verlaufen alle Graphen wieder identisch wie beim einfachen Level. Der Achtzehner-Agent erreicht bei $6,55 \cdot 10^5$ Schritten das erste Mal wieder die 0,9-Marke, der Neuner-Agent bei $1,195 \cdot 10^6$ und der Fünfer-Agent erst bei $3,12 \cdot 10^6$. Damit erreicht der Fünfer-Agent diese Marke erst $1,3 \cdot 10^6$ Iterationen später im Vergleich zum einfachen Level, wobei er dort auch keinen anfänglichen zufälligen Reward-Verlust hatte.
Bei $3,2 \cdot 10^6$ fallen die Werte von (0,908; 0,9614; 0,9899) auf (0,1632; 0,4954; 0,5331). Der Neuner-Agent und Achtzehner-Agent liegen an diesem Punkt gleichauf und haben in den darauffolgenden Iterationen einen ähnlichen Anstieg. Nur der Fünfer-Agent befindet sich weit unterhalb der beiden. Die Graphen erreichen bei den Iterationen ($5,83 \cdot 10^6$; $4,11 \cdot 10^6$; $3,7 \cdot 10^6$) ihre 0,9-Marke. Bis zum Ende bleiben der rote und blaue Graph gleich auf und der orange bleibt weit unten.
Die Endwerte (0,9524; 0,9823; 0,997) liegen sogar über den End-Rewards des einfachen Levels, was an den statistischen Schwankungen der Lernversuche liegen wird. Jedoch wurden mit dem Peformancetest höhere Fehlerquoten festgestellt. Die größte Abwärtsentwicklung hatte dabei der Neuner-Agent, welcher

seine Fehlerquote von 0,179 auf 0,2876 verschlechterte. Die beiden anderen veränderten ihre Raten um weniger als 1%. Der Unterschied in den Raten und in den Graphen kommt durch den schwereren Level zustande. In diesem ist die Navigation anspruchsvoller als im einfachen Level weshalb mehr Informationen zur optimalen Bewegung benötigt werden, die der Agent nur über seine Sichtstrahlen erhält. Die starke Verschlechterung des Neunstrahl-Agenten von 1,8% auf 2,9% kann daran liegen, dass die Qualität der unterschiedlichen Anfänge beim mittleren und einfachen Level sehr verschieden war. Beim einfachen Level wurde im Optimum und beim Medium-Level nicht im Optimum gelernt. Zusätzlich trägt eine statistische Schwankung beim Performancetest weiter zu diesem Ergebnis bei. Von dieser Schwankung ist auch der Achtzehner-Agent betroffen, dessen Lernergebnisse allerdings wie erwartet ausgefallen sind. Bis auf die Hypothese des Fünf-Strahlagenten werden alle in diesem Experiment erfüllt. Denn dieser Agent liegt klar unter seiner erwarteten Fehlerquote, was daran liegt, dass sein Mangel an Positionsinformationen noch keinen zu großen Nachteil im mittleren Level darstellt. Die Fehlerquoten der 3 Agenten liegen noch weit unter der 5% Marke, was die Erwartungen an den Fünfer-Agenten übertrifft. Mit Einbeziehung der oben beschrieben Schwankungen geht der Achtzehner-Agent trotzdem als bester Agent aus dem Experiment hervor.

Experiment 7: Schwer

Wie zu erwarten war, ist der Verlauf der Graphen aus Abbildung 5.15, im letzten Level ähnlich zu den vorheringen Experimenten. Innerhalb der ersten $5 \cdot 10^5$ Iterationen (Abbildung 5.16) ist beim letzten Experiment kein so großer zufälliger Reward-Verlust zwischen 0 und 10^5 wie bei den letzten beiden Leveltests. Allerdings fällt auf, dass der Fünfer-Agent (Rot) sehr lange braucht, um seine erste Aufgabe zu lösen und erst bei $3,5 \cdot 10^4$ auf gleicher Ebene wie die anderen Graphen liegt. Er und der Neuner-Agent (Blau) sind aber bis 10^5 in einer sehr instabilen Phase und schwanken zwischen 1,0 und 0,6, während der dritte Agent(Orange) sehr stabil verläuft und auch bei der ersten Leveländerung nur von 1,0 auf 0,95 fällt, während der Fünfer-Agent von 0,957 auf 0,042 und der Neuner-Agent, der sich momentan in einem Tief befindet, von 0,537 auf 0,3576 fällt. Trotzdem haben beide unteren Agenten einen starken Reward-Gewinn und des wegen erreichen die Agenten bis $2,4 \cdot 10^5$ einen Wert von (0,9909; 0,8733; 0,9865). Ab hier fallen alle drei auf (-0,08685; -0,1824; -0.1804), was der größte Reward-Verlust aller Leveltests bis jetzt ist.

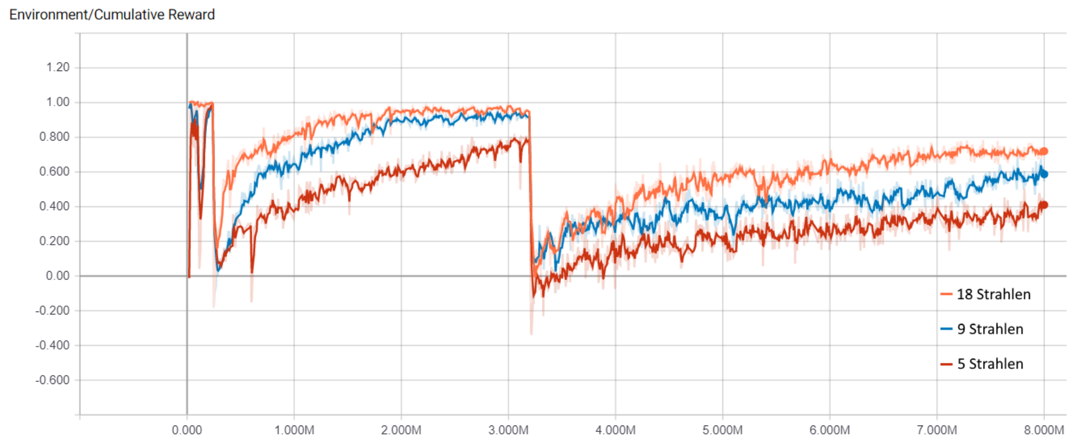

Abbildung 5.15: Experiment 7 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Der schwere Level hat einen noch stärkeren Effekt auf den Anstieg als beim einfachen und mittleren Level. Allerdings ist diesmal der Anstieg der Agenten nicht hoch genug um innerhalb des Lernvorgangs ein sinnvolles Ergebnis zu lernen.

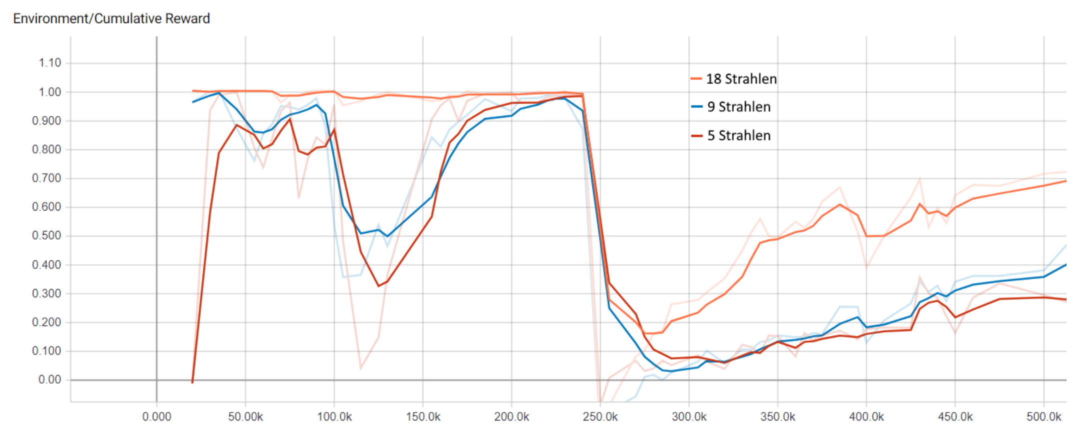

Abbildung 5.16: Experiment 7, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Es ist ein ähnlicher Effekt wie in Abbildung 5.12 und 5.14 zu erkennen. Diesmal ist der Level so schwer, dass der Achtzehner-Agent der einzige ist, der einen normalen Anstieg nach dem Konfigurationswechsel bei $2,4\dot{1}0^5$ aufweist.

Durch die nun hinzugefügten Wände wird das Level halbiert in einen schweren und einen sehr leichten Teil und innerhalb des schweren Teils werden die Agenten ihr Ziel an diesem Iterationspunkt sehr wahrscheinlich nicht erreichen können. Des wegen braucht der Achtzehner-Agent doppelt so lange ($1,14 \cdot 10^6$) um die 0,9 Marke zu erreichen. Mit $1,86 \cdot 10^6$ Iterationen ist der Neuner-Agent Versuch 50% langsamer als im letzten Level und der Fünfer-Agent erreicht diese Marke nicht mehr. Ihre Maxima haben die Graphen beim Punkt $3,2 \cdot 10^6$ mit (0,8011; 0,9094; 0,9354) und diese liegen damit teilweise bis zu 0,1 unter dem Wert des Medium-Levels. Beim darauf folgenden Reward-Verlust erreichen die drei Agenten mit (-0,3156; 0,1987; -0,1565) ihre tiefsten Werte des ganzen Lernprozesses. Das nun fertiggestellte Level gleicht einem Labyrinth, und die Agenten müssen erst die Karte kennenlernen, bevor sie durch sie komplett navigieren können. Für diese Aufgabe ist der Agent mit den meisten Sichstrahlen am effektivsten. Er hat den höchsten Anstieg der drei. Allerdings erreichen alle bis zur $8 \cdot 10^6$ Iteration nicht mehr die 0,9 Marke. Bereits bei etwa $5 \cdot 10^6$ Iterationen stagniert der Anstieg aller Agenten sehr stark, weshalb sie am Ende nur auf die Werte (0,4119; 0,5464; 0,7263) kommen.
Das Ergebnis wird auch in den Fehlerquoten widergespiegelt. Alle Agenten übertreffen ihre in 4.3.2 beschriebenen Hypothesen. Der Achtzehner-Agent hat mit 66% das beste Ergebnis, ist aber nicht viel besser als der Neuner-Agent mit 69%, der im Vergleich zu den anderen die geringste Verschlechterung seiner Quote aufwies. Trotzdem sind alle 3 gelernten Modelle definitiv nicht ausreichend für einen Videospielbot. Der schwere Level hat gezeigt, dass das Lernen der Wegfindung mit ML-Agents seine Grenzen hat. So wird Nortivags AI vorerst keine Labyrintlevels unterstützen können und der Level für den finalen Nortivagtest im nächsten Abschnitt kann darauf angepasst werden.

Zusammenfassung

Obwohl der Achtzehner-Agent die besten Lernergebnisse hatte, war der Effektivitätsunterschied zum Neuner-Agent nicht so groß, wie er vermutet wurde. Trotzdem wird der finale Nortivagtest mit 18-Strahlen durchgeführt und die Schwierigkeit der Karte wird auf dem Level der leichten Umgebung sein, weil es bis jetzt nicht bekannt ist, wie gut der Lernprozess überhaupt in Nortivag funktioniert. Dazu wird auch weiterhin die durch Experiment 2b abgeänderte Standard-Konfigurationsdatei aus Abbildung 4.5 mit geringerer Lernrate benutzt, die für alle Tests bis jetzt verwendet wurde. Da es innerhalb der

NLite-Tests immer wieder am Anfang des Lernversuchs zu instabillem Lernen gekommen ist, wird sicherheitshalber das Nortivag-Experiment drei mal durchgeführt, da innerhalb der Leveltests immer mindestens ein Agent keinen Einbruch aufwies. Natürlich wäre es trotzdem möglich, dass im folgenden Experiment alle drei Agenten "Pech" haben.

5.3 Finaler Test in Nortivag

5.3.1 Vorbereitung

Für die Vorbereitung des Experiments muss zuerst der Agent aus den Konfigurationstests und die Curriculum-Konfigurationsdatei in Nortivag übertragen werden. In ihm müssen die AgentReset()- und die CollectObservation()-Methoden auf den Nortivag-eigenen Code angepasst werden. Zusätzlich muss noch ein Level erstellt werden, der dem leichten Schwierigkeitsgrad der Leveltests entspricht. Dazu wurde der Level aus Abbildung 5.17 entworfen, welcher nur wenige kleine Wände und viel Bewegungsfreiheit enthält.

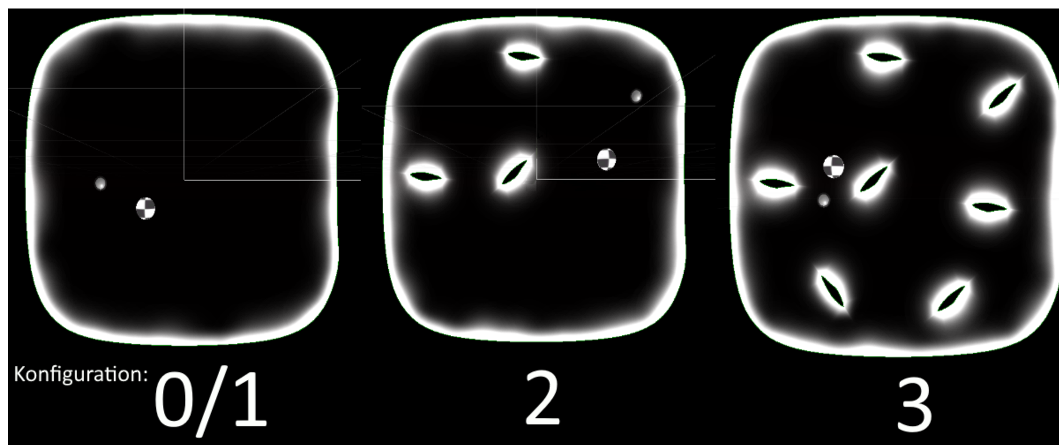

Abbildung 5.17: Die einzelnen Curriculum-Konfigurationen des Nortivag-Level, der für das achte Experiment benutzt wurde.
In der Abbildung sind die einzelnen Levelzustände zu erkennen, die durch die Curriculum-Konfigurationsdatei aus Abbildung 4.4 entstehen.

Auf dieser Karte ist die kleine graue Kugel die Spielfigur des Agenten und der schwarz-weiße Ball das Ziel. Der große Bewegunsraum gleicht die trägere Steuerung des Agenten aus, damit in etwa die gleichen Bewegungsvoraussetzungen

wie in den NLite-Tests vorliegen. Da es in der vorhanden Nortivag-Version nicht möglich ist, mehrere Level gleichzeitig darzustellen und auf ihnen Spielfiguren zu platzieren, wird das Experiment nur auf einer Karte durchgeführt.

5.3.2 Durchführung

Wie auch schon bei den anderen Experimenten wird für den Lernvorgang und den anschließenden Performancetest der gleiche MSI GE 60 Laptop verwendet.

5.3.3 Auswertung

Allgemein

Die Ergebnisse der drei Lerntests wurden in Tabelle 5.3 zusammengefasst. Die variierenden Lernzeiten kommen zustande, da der PC im ersten Versuch nebenbei benutzt wurde, und der zweite und dritte Versuch einen Großteil dieser Zeit über Nacht stattfanden und der Laptop deshalb nicht so stark ausgelastet war.

Tabelle 5.3: Ergebnisse des Nortivag-Tests

Exp.	Dauer	End-Reward	Fehlerrate
1. Versuch	13h 38m 2s	-0,2237	0,63171
2. Versuch	12h 15m 27s	-0,3357	0,64292
3. Versuch	11h 55m 59s	-0,1722	0,55668

Experiment 8: Nortivag

Das Erste, was bei der Auswertung auffällt ist, dass die Graphen viel stärker schwanken als in NLite. Der Glättungsfaktor musste von 0,6 auf 0,9 erhöht werden, um die Graphen innerhalb des Diagramms erkennen zu können. Der ganze Lernversuch war demnach viel instabiler als in NLite. Obwohl das Experiment dreimal wiederholt wurde, weswegen es ähnliche Graphen geben müsste, ist in Abbildung 5.18 ein klarer Unterschied erkennbar. Dieser wird aber erst nach dem Konfigurationswechsel bei $3,2 \cdot 10^6$ sichtbar.

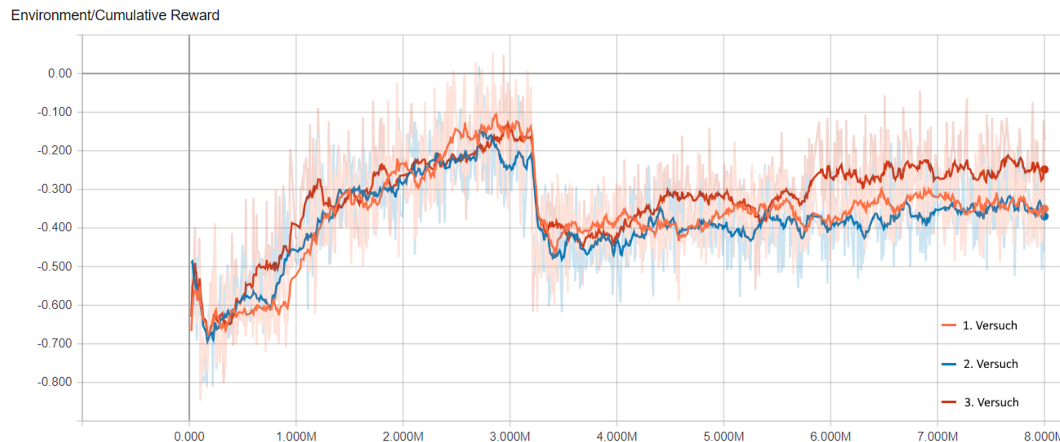

Abbildung 5.18: Experiment 8 Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Durch die enorme Schwankung der Graphen ist selbst bei einer Glättung kaum ein Unterschied in der Reward zu erkennen. Erst bei $5,4 \cdot 10^6$ Iterationen steigt die Reward vom 3. Versuch stärker als bei den anderen beiden.

Innerhalb der ersten $5 \cdot 10^5$ Iterationen sind sich die drei Graphen trotz ihres starken Rauschens sehr ähnlich. Zwar starten die drei Versuche mit vergleichsweise sehr weit entfernten Werten (-0,5644; -0,4908; -0,6301). Jedoch erreichen sie kurz vor der 10^5 Iteration jeweils einen Reward-Hochpunkt von (-0,4734; -0,4686; -0,4258). Der letzte Versuch hat dabei den besten Reward und beim anschließenden Konfigurationswechsel des Curriculums fällt dieser auch nicht so stark wie bei den anderen beiden Experimenten auf eine Reward von (-0,842; -0,8457; -0,7796), die damit auch die niedrigste im ganzen Experiment ist. Das ist auch gleichzeitig ein weiterer Unterschied zu den NLite-Tests. Dort war der niedrigste Punkt immer entweder nach dem Hinzufügen der ersten Wände oder bei Vervollständigung des Levels. Das könnte daran liegen, dass der Agent viel länger braucht um seine Anfangsaufgabe zu lernen, und bei 10^5 bis jetzt nur zufällig das Ziel erreicht hat und in Konfiguration eins diese Chance viel kleiner ist. Bei der Änderung auf Konfiguration Zwei bei $2,4 \cdot 10^5$, sind die Wände kleiner als bei den NLite-Leveln und nur im oberen linken Teil des Spielfelds. Deshalb ist die Wahrscheinlichkeit, dass das Ziel neben dem Agenten erscheint, ohne dass ein Hindernis dazwischen liegt, viel höher. So müssen die Agenten in nur sehr wenigen Fällen dem Hindernis auf dem Weg ausweichen und darum fällt ihre Reward nicht ganz so stark von (-0,3872; -0,3875; -0,6340) auf (-0,7518; -0,7635; -0,7056).

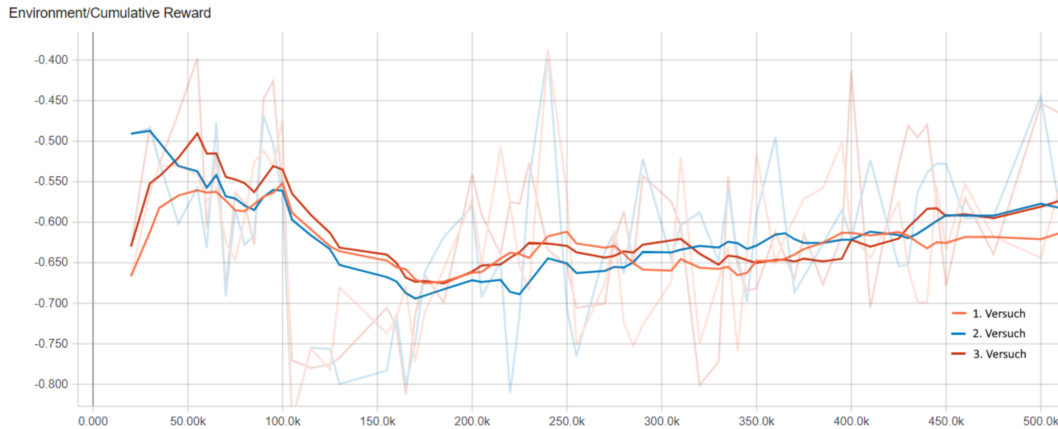

Abbildung 5.19: Experiment 8, die ersten $5 \cdot 10^5$ Iterationen Ergebnis: Mittlere Reward über die Anzahl der Iterationen
Innerhalb der ersten $5 \cdot 10^5$ Iterationen ist kein Unterschied zwischen den Agenten zu erkennen.

Dieser frühe Konfigurationswechsel hätte allerdings nicht so stattfinden sollen. Der Agent hätte erst vollständig lernen müssen, sein Ziel immer zu erreichen und erst danach hätte die Levelveränderung stattfinden dürfen, was im Nachhinein bereits in den Konfigurationstests hätte getestet werden sollen. Der zum Lernen benötigte Zeitunterschied zwischen NLite und Nortivag war allerdings unbekannt, weshalb in zukünftigen Tests das Curriculum angepasst werden muss. Nach dem Reward-Verlust bei etwa $2,5 \cdot 10^5$ Iterationen steigt der Reward wieder an, wobei der Gewinn des Drittversuchs am Anfang der stärkste ist. Allerdings relativiert sich das bis zur $3,2 \cdot 10^6$-Marke wieder. Alle drei Agenten schaffen es innerhalb dieses Zeitraums einige Male über die Null-Reward. Dem Erstversuch gelingt es als erstes bei $2,47 \cdot 10^6$ und wenig später erreicht er bei $2,84 \cdot 10^6$ seinen höchsten Punkt im Lernverlauf mit 0,05397, der gleichzeitig die höchste Reward aller drei Ansätze ist. Der zweite Agent erreicht die Nullmarke bei $2,71 \cdot 10^6$ mit einer Reward von 0,01972 und der letzte Agent bei $2,94 \cdot 10^6$ mit 0,04828. Insgesamt haben diese Punkte allerdings keine zu große Aussagekraft, da die Schwankungen so stark sind. Zu dieser Zeit haben die Agenten eine Reward-Schwankung von 0,4 innerhalb weniger 10^5 Iterationen. Bei $3,2 \cdot 10^6$ Schritten erreichen die Agenten die Werte (-0,03483; -0,1248; -0,1266) und fallen auf (-0,5136; -0,4385; -0,6168). Der darauffolgende Verlauf ist wieder ähnlich zu dem Fall bei 10^5. Bei $5,405 \cdot 10^6$ erreicht der letzte Agent eine viel höheres Reward-Niveau als die anderen beiden Versuche, was auch in der ungeglätteten Version sichtbar ist. Die erreichten End-Rewards liegen bei

(-0,2237; -0,3357; -0,1722), womit der letzte Agent klar als Sieger hervorgeht. Dieses Ergebnis spiegelt sich auch in den Performancetests wider, bei denen er 10% weniger Fehler gemacht hat über die 10^5 Versuche. Bei diesen Tests ist aber auch aufgefallen, dass sich die Agenten nur sehr langsam durch den Level bewegen, was bei den NLite-Experimenten nicht der Fall war. Das könnte auf die Steuerung des Spiels und vieleicht damit verbundene Bugs des Spiels zurückzuführen sein. Mit der reinen Wegfindungs-Performance und der mangelnden Bewegungsgeschwindigkeit ist dieses Ergebnis nicht zufriedenstellend.

5.4 Gesamtauswertung

Aufgrund der Ergebnisse der Leveltests ist bekannt, dass der Wegfindungsanteil des Spielprinzips von Nortivag definitiv mit ML-Agents lösbar ist. Die Auswertung des Nortivag-Tests zeigt jedoch nicht das gewünschte Ergebnis.
Bei einem Level dieses einfachen Schwierigkeitsgrades darf eine KI keine so große Fehlerrate haben, da sie so für den Gebrauch im Spiel nicht geeignet ist. Sie bietet dem Spieler keine Herausforderung und würde das Spielgefühl stark negativ beeinträchtigen. Bei den Performancetests in Nortivag ist zusätzlich aufgefallen, dass die KI sich nur sehr langsam durch den Level bewegt, selbst wenn keine Hindernisse vorhanden sind. Das dürfte nicht passieren und ist in den NLite-Tests auch nie vorgekommen. Wie schon in der Auswertung 5.3.3 beschrieben, muss auch die Curriculum-Konfiguration für Nortivag weiter angepasst werden, da der Agent viel länger braucht, um die Aufgaben zu lernen, die er in NLite innerhalb von $2 \cdot 10^4$ Schritten fertig gelernt hat. Wahrscheinlich könnten viele dieser Probleme gelöst werden, wenn die Lernzeit und Nortivag weiter optimiert werden würden.
Dafür muss Nortivag so angepasst werden, dass der Agent auf mehreren Feldern lernen kann und dass für das Lernen die Hintergrundberechnungen reduziert werden können, zum Beispiel mit dem Entfernen von überflüssigen Effekten. Zusätzlich müssen noch einige Performanceprobleme und Bugs im Spiel behoben werden, deren Auswirkung auf den Lernprozess unbekannt, die aber definitiv vorhanden sind.
Um die Lerngeschwindigkeit zu verbessern, ist die einfachste Verbesserung, einen PC zu benutzen, der eine bessere Leistung hat und der nebenbei nicht für andere Aufgaben benutzt wird. Mit diesem Anstieg kann auch effektiv länger gelernt werden, wodurch eine weitere Verbesserung zu Stande kommen

kann.
Mit allen diesen Optimierungen ist der Lernprozess immer noch qualitativ sehr variabel und instabil, weshalb es auch notwendig sein wird, am Anfang mehrere Anläufe durchzuführen, so wie es im achten Experiment geschehen ist. Von diesen vielen verschiedenen Anläufen kann nun der beste Teil ausgewählt werden, der weiter gelernt wird.

6 Zusammenfassung und Ausblick

Die zentrale Frage dieser Arbeit, herauszufinden ob ML-Agents mit dem firmeneignenen Videospiel Nortivag funktioniert, kann mit dem aktuellen Stand des Spiels mit "Ja" beantwortet werden.
Innerhalb der Experimente wurde herausgefunden, dass der Wegfindungsanteil des Spielprinzips mit ML-Agents umsetzbar ist, allerdings wurde das gewünschte Ergebnis einer funktionierenden Wegfindungs-KI nicht erreicht. Ein Zusammenhang mit dem Plugin (5.4) konnte nicht nachgewiesen werden. Die Ursache liegt an den technischen Voraussetzungen der Spielversion als auch an den technischen Mitteln, die für das Lernen zur Verfügung standen, was beides behebbare Umstände auf Seiten der Entwickler Nortivags sind.
Des Weiteren wurde festgestellt, das sich die Ergebnisse von NLite nicht einfach auf das Hauptspiel übertragen lassen, weswegen sich trotz gleicher Voraussetzungen die besprochenen Resultate sehr stark unterscheiden. Wenn die bekannten Probleme gelöst werden, könnte mithilfe von ML-Agents eine sehr gute KI enstehen, die nicht nur für die Wegfindung benutzt werden kann. Um die Performance der KI zu testen, kann zum verwendeten Test zusätzlich die Zeit gemessen werden die der Agent braucht, um seine Aufgabe zu erfüllen. Diese Zeit ist im Spiel von entscheidener Tragkraft für die Nutzerakzeptanz, da die KI schnell/in Echtzeit reagieren muss. Somit muss dem Agenten zusätzlich zur Wegfindung ein anspruchsvolles Kampfverhalten beigebracht werden, damit er als Spiel-Bot einsatzfähig ist. Innerhalb des Lernens muss beachtet werden, dass die KI auf verschiedenen Karten eingesetzt werden soll und die in 3.1.2 besprochenen Features zu seinem Vorteil benutzen kann. Dazu muss zunächst Nortivag weiter verbessert und entwickelt werden. Sind diese nötigen Bedingungen erfüllt, kann wieder mit dem Trainieren der Agents angefangen werden.
Dazu sollten zu den Ansätzen in dieser Arbeit neue getestet werden. Beispielweise kann geprüft werden, inwiefern ein Agent mit zwei Gehirnen steuerbar

ist, wobei eins das Kampfverhalten steuert, mit all seinenen Fähigkeiten und dem zweiten Gehirn nur Bewegungspositionen übergibt, die es erreichen soll. So könnten beide Gehirne unabhängig voneinander trainiert werden, was den Lernprozess für den Entwickler vereinfacht.
Des Weiteren kann man den von OpenAI genutzten Ansatz übernehmen, bei dem zwei Agenten auf einer Karte gegeneinander lernen und sie ihre Rewards nach den im Spielmodus vorhandenen Punktevergaben bekommen. Dadurch kann der Agent direkt die Spielmodi lernen, mit dem Ziel, die Punkte, die die Agenten am Ende einer Runde bekommen, zu maximieren.
Dazu kann die Anzahl der Strahlen überdacht werden, um auszuschließen, dass es nicht bessere Einstellungen gibt. Für eine bessere Konfiguration muss auch die Standard-Konfigurationsdatei überarbeitet werden. Da eine Senkung der Lernrate nicht den gewünschten Effekt hatte, können noch andere Einstellungen überprüft werden.
Letztendlich hat der genutzte PPO-Algorithmus mit OpenAIs Five in Dota 2 bewiesen, dass er benutzt werden kann, um eine spielstarke KI in Nortivag zu implementieren.

Literaturverzeichnis

[1] Agents. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md, Abrufdatum 07.08.2019.

[2] Ai approaches compared: Rule-based testing vs. learning. https://www.tricentis.com/artificial-intelligence-software-testing/ai-approaches-rule-based-testing-vs-learning/, Abrufdatum: 04.08.2019.

[3] Ghost psychology. https://www.webpacman.com/ghosts.php, Abrufdatum 06.08.2019.

[4] Learning enviroment desing brains. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Brains.md, Abrufdatum 31.07.2019.

[5] Making a new learning enviroment. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md, Abrufdatum 07.08.2019.

[6] Ml-agents toolkit overview. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md, Abrufdatum 31.07.2019.

[7] Openai five. https://openai.com/five/, Abrufdatum 31.07.2019.

[8] Training ppo. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md, Abrufdatum 31.07.2019.

[9] Training with curriculum learning. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Curriculum-Learning.md, Abrufdatum 09.08.2019.

[10] Sebastien Benard. Building the level design of a procedurally generated metroidvania: a hybrid approach., 29.03.2017. https:

//www.gamasutra.com/blogs/SebastienBENARD/20170329/294642/Building_the_Level_Design_of_a_procedurally_generated_Metroidvania_a_hybrid_approach.php, Abrufdatum 20.08.2019.

[11] Greg Brockman, Ilya Sutskever, and OpenAI. Introducing openai, 11.12.2015. https://openai.com/blog/introducing-openai/, Abrufdatum 31.7.2019.

[12] deparkes. Machine learning vs rules systems, 24.11.2017. https://deparkes.co.uk/2017/11/24/machine-learning-vs-rules-systems/, Abrufdatum: 04.08.2019.

[13] Filip Wolski Prafulla Dhariwal Alec Radford John Schulman, Oleg Klimov. Proximal policy optimization. https://openai.com/blog/openai-baselines-ppo/, Abrufdatum: 04.08.2019.

[14] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018.

[15] Sergios Karagiannakos. Trust region and proximal policy optimization, 11.01.2019. https://sergioskar.github.io/TRPO_PPO/, Abrufdatum 21.08.2019.

[16] Sergios Karagiannakos. The secrets behind reinforcement learning, 23.10.2018. https://sergioskar.github.io/Reinforcement_learning/, Abrufdatum 21.08.2019.

[17] Heiko Klinge. Künstliche dummheit statt künstliche intelligenz - warum künstliche intelligenz (ki) in spielen stagniert, 25.01.2008. https://www.tecchannel.de/a/warum-kuenstliche-intelligenz-ki-in-spielen-stagniert,1744817, Abrufdatum 27.08.2019.

[18] Siddhesh V. Kolwankar. Article: Evolutionary artificial intelligence for moba / action-rts games using genetic algorithms. *IJCA Proceedings on International Conference on Recent Trends in Information Technology and Computer Science 2012*, ICRTITCS(10):29–31, February 2013. Full text available.

[19] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. 2017.

[20] Martin Lorber. Künstliche intelligenz in videospielen, 04.04.2016. https://spielkultur.ea.de/allgemein/kuenstliche-intelligenz-in-videospielen/, Abrufdatum 31.07.2019.

[21] Martin Lorber. Geschichte der videospiele: Teil 1 – von pong bis zum atari 2600, 06.06.2011. https://spielkultur.ea.de/themen/gesellschaft-und-kultur/geschichte-der-videospiele-teil-1-von-pong-bis-zum-atari-2600/, Abrufdatum 31.07.2019.

[22] André Maré. Pac-man patterns — ghost movement (strategy pattern), 14.02.2018. https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a, Abrufdatum 31.07.2019.

[23] OpenAI. Aboud openai. https://OpenAI.com/about/, Abrufdatum 31.7.2019.

[24] OpenAI. Openai five, 2018. https://blog.openai.com/openai-five/, Abrufdatum 15.08.2019.

[25] James Ovenden. Bad data is ruining machine learning, here's how to fix it. https://channels.theinnovationenterprise.com/articles/bad-data-is-ruining-machine-learning-here-s-how-to-fix-it, Abrufdatum 19.08.2019.

[26] Clayton Purdom. No man's sky is finally discovering signs of life in its ai universe, 24.07.2018. https://games.avclub.com/no-man-s-sky-is-finally-discovering-signs-of-life-in-it-1827754815, Abrufdatum 31.07.2019.

[27] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[29] Statista. Videospiele weltweit. https://de.statista.com/outlook/203/100/videospiele/weltweit, Abrufdatum 26.08.2019.

[30] Nick Statt. How artificial intelligence will revolutionize the way video games are developed and played, 06.03.2019. https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning, Abrufdatum: 04.08.2019.

[31] Nick Statt. Openai's dota 2 ai steamrolls world champion e-sports team with back-to-back victories, 13.04.2019. https://www.theverge.com/2019/4/13/18309459/openai-five-dota-2-finals-ai-bot-competition-og-e-sports-the-international-champion, Abrufdatum: 19.08.2019.

[32] Xander Steenbrugge. Policy gradient methods and proximal policy optimization (ppo): diving into deep rl!, 01.10.2018. https://www.youtube.com/watch?v=5P7I-xPq8u8, Abrufdatum 26.07.2019.

[33] TheHappyCat. How "smart" ai (basically) works in games (goal oriented action planning), 16.07.2016. https://www.youtube.com/watch?v=nEnNtiumgII, Abrufdatum 14.08.2019.

[34] Prof. Dr. Sanaz Mostaghim und Alexander Dockhorn. Computational intelligence in games: Introduction, folien, 2018.

[35] Prof. Dr. Sanaz Mostaghim und Alexander Dockhorn. Computational intelligence in games: Introduction to reinforcment learning, slides, 2018.

[36] Michael Walbridge, 12.06.2008. https://www.gamasutra.com/php-bin/news_index.php?story=18863, Abrufdatum 02.08.2019.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Philipp Thoms | Magdeburg, 10.09.2019