



UNIVERSITY OF APPLIED SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

Master Thesis

Anwendung von Reinforcement Learning zur Entwicklung authentischer Spielwelten mithilfe bedürfnisbasierter Agenten

**Ein alternativer Ansatz zur Entwicklung von Non-Player-Characters
anhand einer Dorfsimulation mit Unity**

Eingereicht am:

26 Februar 2020

Eingereicht von:

Tobias Jansing
Billwerder Neuer Deich 6
20539 Hamburg
Tel.: (+49 157) 52 56 65 78
E-mail: tob.jansing@gmail.com

Referent:

Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-41
E-mail: uh@fh-wedel.de

Koreferent:

Prof. Dr. Michael Predeschly
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-45
E-mail: mpr@fh-wedel.de

Kurzfassung

Die vorliegende Arbeit stellt einen neuen Ansatz für die Erzeugung glaubwürdiger *Nicht-Spieler-Charaktere* (NPCs) in Videospielen vor. Der Fokus liegt auf dem Entscheidungsprozess von intelligenten Agenten für die Darstellung eines authentischen Verhaltens. Anstelle von klassischen Methoden wie *State Machines* basiert der entwickelte Ansatz auf *Reinforcement Learning* und *Utility Based Agents*. Dabei handelt es sich um bedürfnisorientierte Agenten, die über menschliche Attribute verfügen. Sie lernen ihre Entscheidungen so zu wählen, dass ihre Bedürfnisfunktion maximiert wird.

Als Voraussetzung für das Verfahren werden Grundlagen zu intelligenten Agenten sowie im Machine Learning-Bereich mit starkem Fokus auf Reinforcement Learning geschaffen. In der Spieleindustrie etablierte Technologien für die Entwicklung von NPCs werden diskutiert, um einen direkten Vergleich mit dem eigenen Verfahren herstellen zu können. Der Hauptteil der Arbeit besteht aus einer Implementierung des Verfahrens anhand eines bewohnten Dorfsystems als Umgebung für lernende Agenten mithilfe der Game Engine Unity. Die entstandene Architektur sowie ihre einzelnen Komponenten werden ausführlich diskutiert.

Ferner werden darauf aufbauend zwei zusätzliche Ansätze entwickelt, um verschiedene Methoden gegeneinander abzuwägen und Vergleiche zu ziehen. Genauer werden eine regelbasierte Vorgehensweise sowie *Goal Oriented Action Plannings* (GOAPs) verwendet.

Schlussendlich folgt ein Überblick über mögliche weiterführende Entwicklungen und ein Fazit zu den Implementierungen mit einem Fokus auf die Beurteilung der Glaubwürdigkeit der NPCs und die Schwierigkeiten der jeweiligen Technologien. Dabei werden der Erfolg des entwickelten Ansatzes auch im Hinblick auf sowohl vergleichbare als auch unterschiedliche relevante Verfahren bewertet und seine Schwierigkeiten sowie Vorteile herausgestellt. Zuletzt liefert ein Ausblick Informationen über mögliche zukünftige Forschung.

Abstract

This study introduces a new approach towards the creation of believable *non-playable-characters* (npcs) in videogames. Its focus is set on improving the internal decision process utilized in intelligent agents to induce a more authentic environment for the player. A Machine Learning approach is applied using *Reinforcement Learning* rather than classical methods like *Finite State Machines*. Furthermore the agent is encouraged to display humanlike behavior by maximizing need-based utility functions.

As a preliminary requirement for the proposed method, fundamental knowledge is provided by illustrating underlying concepts such as intelligent agents and Reinforcement Learning. Well-established technologies for the definition of NPC behavior is discussed in order to create a direct comparison to the designed process. The main part of the thesis comprises the implementation of said method by developing a village simulation as an environment for learning agents using the game engine *Unity*. A general outline of the architecture as well as a detailed description of the various components employed in this system is then conducted.

Moreover, two additional methods are applied to create a direct comparison to draw conclusions and weigh the benefits and tracks of the different approaches. Specifically this includes a rule-based approach as well as *Goal Oriented Action Planning* (GOAP). Finally an overview about further improvements and a conclusion about the generated approaches is given. The latter includes an assessment of the believability of the respective methods and their difficulties as well as advantages of the underlying technology determining the success of the method. An outlook then provides an overview about possible further research.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
List of Listings	IV
1. Einleitung	1
2. Intelligente Agenten in Videospielen	6
2.1. Finite State Machines	7
2.2. Decision Trees	11
2.3. Behavior Trees	12
2.4. Goal Oriented Action Planning (GOAP)	15
2.5. Utility Based Agents	16
3. Reinforcement Learning	18
3.1. Einordnung	18
3.1.1. Überwachtes Lernen	19
3.1.2. Unüberwachtes Lernen	20
3.1.3. Reinforcement Learning	20
3.2. Markov Decision Problems	22
3.2.1. Beispiel	22
3.2.2. Formale Beschreibung	24
3.2.3. Markov-Eigenschaft	25
3.3. Modellfreie und modellbasierte Verfahren	26
3.3.1. Modellbasierte Verfahren	26
3.3.2. Modellfreie Verfahren	29
4. Anwendung von Machine Learning in Videospielen	36
4.1. Lernende Agenten	36
4.1.1. Lernen zur Laufzeit	36
4.1.2. Nutzung vortrainierter Modelle	37
4.2. Anwendungsfälle außerhalb der intelligenten Agenten	38
4.3. Aktuelle Entwicklungen	40
4.3.1. DeepMind	40
4.3.2. OpenAI Five	40
4.4. Potential	42
5. ML-Agents Toolkit	43
5.1. Unity	43
5.1.1. Machine Learning in Unity	45
5.2. ML-Agents-Architektur	46
5.2.1. Grundlegende Funktionsweise	46
5.2.2. Academy	47
5.2.3. Agent	47
5.2.4. Brain	49
5.2.5. Model	49
5.2.6. Training	49
5.3. ML-Agents-Beispiel	51
5.3.1. Aufgabe	51

Inhaltsverzeichnis

5.3.2. Implementierung des Agenten	51
5.3.3. Training	54
6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning	56
6.1. Anforderungen	56
6.1.1. Trennung von Modell, Darstellung und Logik	57
6.2. Modellierung	57
6.3. Softwaredesign- und Architektur	59
6.4. Implementierung	61
6.4.1. Village	61
6.4.2. GameAcademy	62
6.4.3. LocationHandler	64
6.4.4. ResourceHandler	67
6.4.5. LightHandler	70
6.4.6. Implementierung des Agenten und dessen Subkomponenten	73
6.4.7. VillageAgent	93
6.4.8. UI	94
6.4.9. Logging	96
6.4.10. Fake Inference	96
6.4.11. Player	99
6.5. Start und Bedienung der Dorfsimulation	100
7. Training und Ergebnisse	101
7.1. Optimierung der Trainingsumgebung	101
7.2. Ablauf des Trainings	102
7.3. Verschiedene Ansätze für Ressourcensystem und VillageAgent	104
7.4. Wahl der initialen Hyperparameter	108
7.5. Güteüberprüfung	110
7.6. Trainingsergebnisse	111
7.6.1. Basistraining	111
7.6.2. Optimierung der Ergebnisse durch Anpassung der Hyperparameter	113
7.6.3. Finaler Trainingsvorgang	115
7.6.4. Verhalten der Agenten	116
8. Weitere entwickelte Ansätze	124
8.1. Regelbasierter Ansatz	124
8.2. Goal Oriented Action Planning	126
8.2.1. Verwendete GOAP-Bibliothek	126
8.2.2. Architektur	127
8.3. Ähnliche Ansätze	130
8.4. Vergleich der Ansätze	130
8.4.1. Benötigter Entwicklungsaufwand	130
8.4.2. Verhalten der Agenten	132
9. Fazit	135
9.1. Zusammenfassung	135
9.2. Bewertung	136
9.2.1. Probleme mit Machine Learning und Reinforcement Learning	137
9.2.2. Beschaffenheit des Aktionsraumes	137
9.2.3. Dauer der Trainingsvorgänge	138
9.2.4. Nichtdeterminismus als Hindernis	138

Inhaltsverzeichnis

9.2.5. Probleme mit ML-Agents	139
9.3. Ausblick	139
A. Anhang A - Wahl der optimalen Hyperparameter	141
Literaturverzeichnis	160

Abbildungsverzeichnis

1.1. Grafik in <i>The Witcher 3: Wild Hunt</i> , Sonnenuntergang [Wal15]	3
1.2. Grafik in <i>The Witcher 3: Wild Hunt</i> , Nacht [Wal15]	3
2.1. Beispielhafte Finite State Machine [LESM19]	7
2.2. Hinzufügen eines neuen Zustands, normale State Machine	10
2.3. Hinzufügen eines neuen Zustands, hierarchische State Machine	11
2.4. Beispielhafter Decision Tree [DM18]	12
2.5. Beispielhafter Behavior Tree [DM18]	12
2.6. Sequenzielle Ausführung in Behavior Tree [DM18]	13
2.7. Ausgewählte Ausführung in Behavior Tree [DM18]	14
2.8. Kombination sequenzieller und ausgewählter Ausführung in Behavior Tree [DM18] . .	14
2.9. GOAP-Sequenz nach Orkin [Ork]	15
2.10. Utility-Funktion für Benutzung eines Heiltranks	17
3.1. Grundsätzlicher Ablauf von <i>Reinforcement Learning</i> (RiL) [CR16]	21
3.2. Deterministisches MDP [CS03]	23
3.3. Lösung des deterministischen MDP [CS03]	23
3.4. Nichtdeterministisches MDP [CS03]	23
3.5. Lösung des nichtdeterministischen MDP [CS03]	24
4.1. Trainingsverlauf OpenAI Five [CB19]	41
5.1. Unity-Echtzeitgrafik in <i>The Heretic</i> [Tec19a]	43
5.2. Oberfläche des Unity Editors	45
5.3. ML-Agents Architektur [Tec17d]	46
5.4. Tensorboard-Beispiel	50
5.5. ML-Agents-Beispiel, Aufbau der Szene	51
5.6. ML-Agents-Beispiel, Konfiguration des Brains	52
5.7. ML-Agents-Beispiel, aktualisierter Aufbau der Szene mit Multi-Agent-Setup	54
5.8. ML-Agents-Beispiel, kumulative Belohnung des Trainingsverlaufs	55
6.1. Beispiele für modulare Komponenten zur Modellierung des Dorfes	57
6.2. Modelliertes modulares Haus	58
6.3. Modellierte Stadt, äußere Ansicht	58
6.4. Modellierte Stadt, innere Ansicht	59
6.5. Verwendete Notation für die Beschreibung von Klassen	59
6.6. Grundsätzliche Architektur des Dorfsystems	60
6.7. Methoden der Klasse <code>Village</code>	61
6.8. Methoden der Klasse <code>GameAcademy</code>	62
6.9. Architektur, Ladevorgang mehrerer Dörfer	64
6.10. Methoden der Klasse <code>LocationHandler</code>	65
6.11. Definition von <code>BuildingNavPoints</code> und <code>BuildingDistanceNavPoint</code>	66
6.12. Methoden der Klasse <code>ResourceHandler</code>	68
6.13. Methoden der Klasse <code>TimeController</code>	68
6.14. Methoden der Klasse <code>LightHandler</code>	71
6.15. Modellierte Stadt, äußere Ansicht, nächtliche Beleuchtung	72
6.16. Modellierte Stadt, innere Ansicht, nächtliche Beleuchtung	72
6.17. Methoden der Klasse <code>NpcAgent</code>	73

Abbildungsverzeichnis

6.18. Methoden der Klasse <code>BaseActionPlan</code>	75
6.19. Methoden der Klasse <code>ActionHandler</code>	78
6.20. Methoden der Klasse <code>MoveHandler</code>	79
6.21. Methoden der Klasse <code>WaitHandler</code>	79
6.22. Methoden der Klasse <code>WaitHandler</code>	82
6.23. Hungerfunktion des Agenten	85
6.24. Bedürfnisfunktionen des Agenten für Arbeit und Schlaf	86
6.25. Beispielhafter Ausschnitt des generierten NavMesh	88
6.26. Agenten navigieren über das NavMesh	89
6.27. Methoden der Klasse <code>ActionMaskingHandler</code>	90
6.28. Architektur der Agenten	91
6.29. UI aus Sicht eines Agenten	95
7.1. Occlusion Culling, Ignorieren von Objekten außerhalb des Sichtfeldes	103
7.2. Occlusion Culling, Ignorieren von Objekten durch Verdeckung	103
7.3. Trainingsverläufe für <code>NpcAgents</code> , verschiedene Ansätze für das Ressourcensystem	104
7.4. Trainingsverlauf des <code>VillageAgent</code>	105
7.5. Erweiterter Trainingsverlauf des <code>VillageAgent</code>	106
7.6. Akkumulierte Belohnungen im Basistraining des <code>NpcAgent</code>	111
7.7. Value Loss im Basistraining des <code>NpcAgent</code>	112
7.8. Entropieverlauf im Basistraining des <code>NpcAgent</code>	112
7.9. Dauern der unterschiedlichen Trainingsvorgänge	113
7.10. Belohnungsfunktion durch Kombination der optimalen Einzelparameter	114
7.11. Akkumulierte Belohnungen im finalen Training des <code>NpcAgent</code>	115
7.12. Value Loss im finalen Training des <code>NpcAgent</code>	116
7.13. Entropieverlauf im finalen Training des <code>NpcAgent</code>	116
7.14. Verteilung der Attribute und Berufe der Agenten	118
7.15. Verteilung der Attribute und Berufe der Agenten	118
7.16. RIL-Ansatz, Verlauf der durchschnittlichen Bedürfniswerte	119
7.17. Durchschnittliche Dauer der Aktionen	121
7.18. Verhältnis der Menge der ausgeführten Aktionen	122
8.1. Methoden der Klasse <code>Manual Decision Maker</code>	124
8.2. Regelbasierter Ansatz, Verlauf der durchschnittlichen Bedürfniswerte	132
8.3. Regelbasierter Ansatz, Verlauf der durchschnittlichen Bedürfniswerte	133
8.4. Häufigkeiten der Aktionen, regelbasierter Ansatz und ML-Agents-Ansatz	133
A.1. Belohnungsverlauf, Anpassung von <i>learning_rate</i>	141
A.2. Verlauf des Value Loss, Anpassung von <i>learning_rate</i>	142
A.3. Entropieverlauf, Anpassung von <i>learning_rate</i>	142
A.4. Belohnungsverlauf, Anpassung von <i>num_layers</i>	143
A.5. Verlauf des Value Loss, Anpassung von <i>num_layers</i>	144
A.6. Entropieverlauf, Anpassung von <i>num_layers</i>	144
A.7. Belohnungsverlauf, Anpassung von <i>hidden_units</i>	145
A.8. Verlauf des Value Loss, Anpassung von <i>hidden_units</i>	145
A.9. Entropieverlauf, Anpassung von <i>hidden_units</i>	146
A.10. Erweiterter Belohnungsverlauf, Anpassung von <i>hidden_units</i>	147
A.11. Belohnungsverlauf, Anpassung von <i>use_recurrent</i>	148
A.12. Verlauf des Value Loss, Anpassung von <i>use_recurrent</i>	148
A.13. Entropieverlauf, Anpassung von <i>use_recurrent</i>	149
A.14. Belohnungsverlauf, Anpassung von <i>gamma</i>	149
A.15. Entropieverlauf, Anpassung von <i>gamma</i>	150

Abbildungsverzeichnis

A.16.Verlauf des Value Loss, Anpassung von <i>gamma</i>	150
A.17.Belohnungsverlauf, Anpassung von <i>batch_size</i>	151
A.18.Verlauf des Value Loss, Anpassung von <i>batch_size</i>	151
A.19.Entropieverlauf, Anpassung von <i>batch_size</i>	152
A.20.Belohnungsverlauf, Anpassung von <i>beta</i>	152
A.21.Verlauf des Value Loss, Anpassung von <i>beta</i>	153
A.22.Entropieverlauf, Anpassung von <i>beta</i>	153
A.23.Belohnungsverlauf, Anpassung von <i>lambd</i>	154
A.24.Verlauf des Value Loss, Anpassung von <i>lambd</i>	154
A.25.Entropieverlauf, Anpassung von <i>lambd</i>	155
A.26.Belohnungsverlauf, Anpassung von <i>num_epochs</i>	155
A.27.Verlauf des Value Loss, Anpassung von <i>num_epochs</i>	156
A.28.Entropieverlauf, Anpassung von <i>num_epochs</i>	156
A.29.Belohnungsverlauf, Anpassung von <i>epsilon</i>	157
A.30.Verlauf des Value Loss, Anpassung von <i>epsilon</i>	157
A.31.Entropieverlauf, Anpassung von <i>epsilon</i>	158
A.32.Belohnungsverlauf, Anpassung von <i>buffer_size</i>	158
A.33.Verlauf des Value Loss, Anpassung von <i>buffer_size</i>	159
A.34.Entropieverlauf, Anpassung von <i>buffer_size</i>	159

List of Listings

2.1. State Class	8
2.2. Transition Class	8
2.3. Beispiel-State	8
2.4. Beispiel-Transition	8
2.5. State Machine Controller	9
5.1. ML-Agents-Beispiel, CollectObservations	52
5.2. ML-Agents-Beispiel, AgentAction	53
5.3. ML-Agents-Beispiel, AgentReset	53
6.1. Initialisierung von Agenten	62
6.2. Auswahl von zufälligen Gebäuden in der Klasse <code>LocationHandler</code>	67
6.3. Funktion <code>advanceTime</code> im <code>TimeController</code>	69
6.4. Funktion <code>_setCurrentTime</code> im <code>TimeController</code>	69
6.5. Funktion <code>_checkIfDaysPassed</code> im <code>TimeController</code>	69
6.6. Funktion <code>updateTime</code> im <code>TimeController</code>	70
6.7. Berechnung der Dauer einer Sekunde in Simulationszeit	70
6.8. Umrechnung von Simulationszeit und Realzeit	70
6.9. Anpassung der Sonne durch den <code>LightHandler</code>	71
6.10. Beobachtungen des Agenten	74
6.11. Funktion <code>AgentAction</code> des Agenten	74
6.12. ActionPlan-Callback	76
6.13. Starten eines ActionPlans	76
6.14. Beispielhafter ActionPlan <code>ReligionChurchActionPlan</code>	77
6.15. Implementierung von <code>BaseAction</code>	78
6.16. Funktion <code>updateState</code> in <code>NpcState</code>	83
6.17. Grundfunktion für die Abnahme der Grundbedürfnisse	83
6.18. Aufruf der Grundfunktion bei einer Anpassung der Bedürfnisse	84
6.19. Aufruf der Grundfunktion bei einer Anpassung der Bedürfnisse	84
6.20. Grundfunktion für Bedürfnisse (Beispiel Hungerbedürfnis)	84
6.21. Funktion für das Schlafbedürfnis	85
6.22. Berechnung des Offset für das Arbeitsbedürfnis	86
6.23. Berechnung des Recreationwerts	87
6.24. Erzeugung der <code>ActionMask</code> für Aktionen	89
6.25. Aktionen des <code>VillageAgent</code>	94
6.26. Berechnung der Pfadlänge durch <code>calculatePathLength</code>	97
6.27. Behandlung der Fake Inference in der Klasse <code>MoveAction</code>	98
6.28. Implementierung des <code>PlayerController</code>	99
8.1. Funktion <code>makeDecision</code> im <code>ManualDecisionMaker</code>	125
8.2. Funktion <code>makeNeedDecision</code> im <code>ManualDecisionMaker</code>	126
8.3. Erzeugung GOAP-basierter Agenten <code>NpcGoapAgent</code>	128
8.4. Implementierung der Funktion <code>Move</code> in der Klasse <code>NpcGoapAgent</code>	128
8.5. Implementierung der Getter-Funktion <code>ActiveActionInRange</code> in der Klasse <code>NpcGoapAgent</code>	129

1

Einleitung

In der heutigen Zeit gewinnen Videospiele immer mehr an Bedeutung und verdrängen dabei zunehmend klassische Medien der Entertainment-Branche. So hat der bisher finanziell erfolgreichste Film *Avengers: Endgame* insgesamt einen Umsatz von etwa \$2,8 Milliarden erzielt [Box20]; das Spiel *Grand Theft Auto V* hingegen konnte mehr als \$6 Milliarden [Don18] und *World of Warcraft* sogar Einnahmen von \$9,23 Milliarden bereits im Jahr 2016 [Lea17] erzielen. Mit der steigenden Relevanz des Marktes sowie dem verfügbaren Kapital in der Branche steigen ebenfalls die Erwartungen der Spieler an das Spielerlebnis. Auf dem visuellen Level äußert sich dies in einer stetig verbesserten Grafikqualität und damit verbundenen höheren Hardwareanforderungen.

Gleichzeitig ist ein ähnlich großes Wachstum in der Machine Learning-Branche zu verzeichnen. Gestiegene Möglichkeiten und kontinuierliche Entwicklungen besonders im Bereich des *Deep Learning* haben dazu geführt, dass immer mehr Firmen vermehrt auf *Künstliche Intelligenz* bauen und KI-gestützte Prozesse für intelligentere Verfahren verwenden. Kürzliche Erfolge haben gezeigt, dass mit Technologien wie *Reinforcement Learning* Systeme entwickelt werden können, deren Intelligenz in ihrem Expertenbereich über die kognitiven Fähigkeiten von menschlichen Anwendern und selbst professionellen Esports-Spielern hinausgeht [Sta19]. Somit wurden auch in der Spieleindustrie Durchbrüche erzielt.

Im Vergleich zu älteren Spielen haben heutige NPCs hinsichtlich ihrer Glaubwürdigkeit eine enorme Steigerung erfahren. Generell konnte die Qualität und Überzeugungsfähigkeit virtueller Spielwelten seit der Popularisierung von Videospielen kontinuierlich gesteigert werden. Der hohe grafische Standard erzeugt dabei gerade in Verbindung mit neuen Technologien wie *Virtual Reality* aber auch einen hohen Erwartungslevel an die Authentizität von NPCs [MG17]. Der Kontrast zwischen visueller Qualität und der Authenzität der Umgebung des Spielers steht allerdings im direkten Konflikt zu den Erwartungen der Spieler. Bei näherer Betrachtung zeigen diese häufig trotz der Verbesserungen einen sehr niedrigen Level von Dynamik und Lebendigkeit. Für sich betrachtet wirken einzelne NPCs häufig berechenbar und wenig eigenständig, obwohl die Atmosphäre eines Spiels maßgeblich davon abhängt, wie lebendig diese wirken. Die Glaubwürdigkeit der Umgebung des Spielers wird direkt durch die gewählten Aktionen eines NPCs beeinflusst und erhöht sich mit einem möglichst menschlich wirkenden Verhalten. Dazu zählen nicht nur interaktionsfähige NPCs sondern ebenfalls Hintergrundcharaktere die häufig verwendet werden, um Orte zu bevölkern und diese lebendig wirken zu lassen.

Der erreichte Immersionslevel steht also im Kontrast zu der weit fortgeschrittenen visuellen Qualität aktueller Spiele, die dem Spieler eine höchst realistische Welt vorgeben. Die erreichte Authentizität unterscheidet sich dabei enorm vom betrachteten Spiel und dem für die Entwicklung von NPCs betriebenen Aufwand. Idealerweise versucht man, die Illusion zu schaffen, jeder NPC hätte eine eigene Persönlichkeit und käme durch die Umstände seines persönlichen Lebens verschiedenen Pflichten und Aktivitäten nach.

In der Spieleindustrie typische Methoden basieren dabei verstärkt auf *Scripting*, sodass das Verhalten von Hintergrund-NPCs oft durch „hardcoding“ definiert wird [Fal13]. Die resultierenden Entscheidungsbäume bestimmen dem Agenten, mit welchen Aktionen er auf äußere Einflüsse zu

1. Einleitung

reagieren hat. Dabei kann es sich um einen sehr zeitaufwändigen Prozess handeln. Eine Ursache für statisches Verhalten ist oft unzureichendes *Scripting*, was sich durch repetitive Tagesabläufe, statisch festgelegte Zeitpunkte und Orte für Aktionen oder sogar das vollständige Fehlen von Änderungen der Aktionen der Agenten äußert. In diesem Fall muss viel Aufwand betrieben werden, um möglichst viele vorstellbare Szenarien abzudecken, sodass ein dynamisch wirkendes Ergebnis entstehen kann [Shi17].

Das 2018 erschienene Spiel *Red Dead Redemption 2* [Stu18] gilt weithin als eines der Spiele mit den immersivsten und lebendigsten virtuellen Welten. Für jeden NPC wurde hier eine 80-seitige Hintergrundgeschichte entwickelt, um diesem eine Hintergrundgeschichte zu vermitteln. Dadurch wird ein grober Eindruck über den nötigen Scripting-Aufwand für die Umsetzung solch einer Persönlichkeit vermittelt [Thu18].

Die Leistung eines derart hohen Aufwands mit den verbundenen nötigen Kapazitäten ist in der Regel jedoch nicht realistisch. Fehlende Ressourcen und Zeit oder der Fokus auf andere Bereiche des Spiels resultiert in der sehr kompetitiven Spielebranche daher häufig in einem statischen Verhalten der Agenten und einem limitierten Entscheidungsprozess für die Ausführung von Aktionen.

Im Rollenspiel *The Elder Scrolls V: Skyrim* [Sof11] existieren eine Anzahl von Dörfern und kleineren Städten in einer offenen Welt. Es gibt eine begrenzte Anzahl von Charakteren, jeder von ihnen hat einen Namen. Die meisten Charaktere sind an ein Dorf gebunden, in dem sie sich dauerhaft aufhalten. Dem durchschnittlichen Spieler fällt zunächst nicht auf, dass der Großteil der NPCs ein deutlich statisches Verhalten aufweist. Kehrt man jedoch zu unterschiedlichen Tageszeiten an den selben Ort zurück, so fallen repetitive Muster auf - beispielsweise halten sich in der Dorftaverne dieselben Personen an den gleichen Orten auf wie bereits zuvor. Dies gilt nicht für alle NPCs - einige Wichtigere, mit dem der Spieler oft interagiert, verfügen über deutlich größere Entscheidungsbäume als andere - es raubt den Übrigen allerdings einen großen Teil der Glaubwürdigkeit. Zahlreiche Community-Mods sollen die statischen NPCs verbessern und ihnen ein lebendigeres Verhalten verleihen. Das zeigt den Wunsch von Spielern nach glaubwürdig agierenden NPCs.

In dem Spiel *The Witcher 3: Wild Hunt* [RED15] wird die virtuelle Welt grafisch auf eine äußerst realistische Art dargestellt. Neben einer hohen visuellen Qualität durch Beleuchtungseffekte und Shader werden ebenfalls die Tageszeiten mit korrekter Beleuchtung sowie Wetterbedingungen mit Einflüssen auf die Flora glaubhaft simuliert, wie in den Abbildungen 1.1 und 1.2 zu sehen. Obwohl das Spiel bereits 2015 veröffentlicht wurde, zeigt sich auf den ersten Blick eine sehr glaubhafte, realistische Darstellung der Spielwelt. Diese Darstellung steht im direkten Kontrast zu den NPCs, die die Spielwelt befüllen.

1. Einleitung



Abbildung 1.1.: Grafik in *The Witcher 3: Wild Hunt*, Sonnenuntergang [Wal15]



Abbildung 1.2.: Grafik in *The Witcher 3: Wild Hunt*, Nacht [Wal15]

Aufgrund der Größe der Städte im Vergleich zu Skyrim muss eine deutlich größere Menge von NPCs verwendet werden. Wegen der höheren Anzahl handelt es sich bei dem Großteil dieser NPCs um namenlose, unbekannte Charaktere. Eine festgelegte Anzahl bekannter NPCs würde hier nicht ausreichen, um die deutlich größere Spielwelt zu füllen. Begibt sich der Spieler an einen Ort, so werden diese Hintergrundcharaktere hinzugefügt und nach dem Verlassen des Ortes wieder entfernt. Diese dienen weniger dem Spielfortschritt sondern der Bevölkerung der Spielwelt. Besonders wegen der Anzahl der Charaktere fällt nicht sofort auf, dass diese stets die gleichen Aktivitäten ausüben. Verfolgt man hingegen einen NPC, so wird schnell ersichtlich, dass sehr einfache Handlungsmuster vorliegen, die sich stets wiederholen. Beispielsweise gehen NPCs endlos zufällig durch die Stadt oder stehen kontinuierlich auf dem Marktplatz, anstatt dass sie ihre Aktivitäten zu einem gewissen Zeitpunkt abbrechen, auf ihre Umwelt reagieren und eine neue, sinnvolle Tätigkeit suchen. Dadurch wird die Immersion der Spielwelt verringert. Die Agenten verfügen über keinen sinnvollen Tagesablauf, der mit ihren persönlichen Präferenzen oder Eigenschaften wie ihrem Alter, ihrem Geschlecht oder

1. Einleitung

ihrem Beruf übereinstimmt. Um eine überzeugende Welt darzustellen verfügen Spieler über eine bestimmte Erwartungshaltung, die nicht vollständig erfüllt wird. Beispielsweise sollte ein NPC dem gefolgt wird nach einer gewissen Zeit seinen Wohnort erreichen, sich dann zur Abendzeit schlafen legen und anschließend morgens zur Arbeit gehen.

Die oben beschriebenen statischen Verhaltensmuster lassen sich trotz einer Zunahme der Immersion in anderen Bereichen in den meisten Spielen feststellen. Die vorliegende Arbeit fokussiert sich auf einen Lösungsansatz für diese zugrunde liegende Problematik. Dabei wird der Fokus nicht auf den visuellen Aspekt gelegt, sondern auf das Finden sinnvoller NPC-Aktionen und dem damit verbundenen Entscheidungsprozess. Ein möglichst menschlich wirkendes Verhalten soll durch eine sinnvolle Auswahl von Tätigkeiten vermittelt werden. Dabei wird ein Machine Learning-Ansatz gewählt, um statt aufwändigem Scripting einen Lernprozess der Agenten anzuregen, sodass weniger manuelle Arbeit nötig ist. Konkret sollen NPCs entwickelt werden, die sich als Hintergrundcharaktere in einer entwickelten Dorfsimulation, um die Spielwelt lebendig wirken zu lassen.

Das Verhalten der Agenten soll einen sinnvollen Tagesablauf abbilden, der von ihren Attributen, Bedürfnissen und z.B. dem Beruf des NPC abhängt. Sie erhalten eine Auswahl aus verschiedenen Aktionen, mit denen sie ihre Bedürfnisse befriedigen können, die auf grundsätzlichen menschlichen Bedürfnissen basieren. Dieses Vorgehen wird gewählt, weil gezeigt wurde, dass die Verwendung von menschlichen Eigenschaften wie Bedürfnissen zur Glaubwürdigkeit von Charakteren beitragen können [Fal13]. Konkret handelt es sich bei den verwendeten Bedürfnissen um interne Funktionen für Eigenschaften wie Hunger, Durst oder Kommunikation. Man spricht von *Utility Based Agents* (siehe Abschnitt 2.5). Weiterhin verfügen sie über statische Attribute wie Sportlichkeit, Kommunikationsfreudigkeit oder Intelligenz, die ihre Persönlichkeit definieren sollen. Generell soll dadurch ein möglichst menschliches Verhalten angeregt werden.

Im Folgenden wird kurz die jeweilige Thematik vorgestellt, mit der sich die einzelnen Kapitel der Thesis befassen.

Kapitel 2 - Intelligente Agenten in Videospielen: Als erstes behandelt dieses Grundlagenkapitel die Funktionsweise intelligenter Agenten. Dabei werden unterschiedliche typische Verfahren beleuchtet, die für die Entwicklung intelligenter Agenten relevant sind.

Kapitel 3 - Reinforcement Learning: *Reinforcement Learning* wird als ein Kernkonzept für die Entwicklung lernender Agenten vorgestellt. Zunächst wird dafür auf die generelle Verwendung maschinellen Lernens in Spielen eingegangen. Wichtige Grundlagen wie *Markov Decision Problems* werden näher erläutert. Die generelle Funktionsweise sowie der mathematische Hintergrund von modellfreien und modellbasierten Verfahren werden vorgestellt. Weiterhin werden ausgewählte relevante Algorithmen aus dem Bereich, insbesondere der in dieser Arbeit verwendete Algorithmus *Proximal Policy Optimization* (PPO), erläutert. Zuletzt werden Beispiele zum State-of-the-Art von Reinforcement Learning beschrieben.

Kapitel 4 - Anwendung von Machine Learning in Videospielen: In diesem Kapitel werden verschiedene Anwendungsfälle für Machine Learning in Videospielen beschrieben und in Kategorien eingeteilt. Dabei wird der Fokus nicht nur auf intelligente Agenten, sondern auf generelle Lösungen für die Verbesserung der Authentizität von Spielwelten gelegt. Es soll gezeigt werden, dass es Anwendungsbereiche gibt, in der die Verwendung von Machine Learning Verbesserungen hinsichtlich des Immersionslevels von Spielwelten erzielen kann.

1. Einleitung

Kapitel 5 - ML-Agents Toolkit: Das Kapitel geht auf die Entwicklung von lernenden intelligenten Agenten innerhalb der Game Engine *Unity* ein. Das Hauptthema ist hier die Einführung in das *ML-Agents Toolkit*-Framework inklusive der Erläuterung der wichtigsten Konzepte. Zur Veranschaulichung wird ein beispielhaftes Projekt entwickelt und vorgestellt, um wesentliche Aspekte sowie die wichtigsten Komponenten herauszustellen.

Kapitel 6 - Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning: Der entwickelte Ansatz einer Dorfsimulation mit lernenden, bedürfnisbasierten Agenten wird in diesem Kapitel dokumentiert. Die Anforderungen und generellen Eigenschaften der Simulation werden zunächst beschrieben. Die Architektur des Verfahrens wird anschließend erst generell und dann im Detail vorgestellt. Das Resultat in Form des modellierten Dorfes mit der zugrunde liegenden Architektur sowie den entwickelten Agenten, die sich in der Dorfsimulation bewegen und Aktionen ausführen, werden vorgestellt.

Kapitel 7 - Training und Ergebnisse: In diesem Kapitel wird der Trainingsvorgang beschrieben, der für den Lernvorgang der Agenten nötig gewesen ist. Dabei wird der Ablauf besprochen, zugrunde liegende Theorien erläutert und Probleme sowie ihre Lösungen gezeigt. Weiterhin werden die Güteüberprüfung des Trainings diskutiert und die Ergebnisse des Trainings vorgestellt. Dabei spielt die Wahl der Hyperparameter sowie die Wahl der Belohnungen der Agenten eine kritische Rolle.

Kapitel 8 - Weitere entwickelte Ansätze: Zusätzlich zu dem auf *Reinforcement Learning* basierenden Ansatz wurden zwei weitere Verfahren für die Entwicklung intelligenter Agenten genutzt. Zunächst wird die Implementierung dieser Ansätze besprochen. Anschließend werden Unterschiede, Vorteile und Nachteile der drei entwickelten Verfahren untersucht.

Kapitel 9 - Fazit: Schlussendlich wird ein Fazit zur vorliegenden Arbeit getätigt. Es beginnt mit einer rückblickenden Zusammenfassung aller Kapitel. Es wird eine Bewertung der entwickelten Dorfsimulation und den lernenden Agenten vorgenommen. Generelle Probleme mit *Machine Learning* und *Reinforcement Learning* sowie spezifische Schwierigkeiten, die in der Thesis entstanden sind, werden diskutiert. Ein Ausblick soll schließlich Vorschläge zu Verbesserungsmöglichkeiten und weiterer Forschung geben.

2

Intelligente Agenten in Videospielen

Intelligente Agenten sind autonom handelnde Entitäten, deren Handlungen von bestimmten Zielen abhängen. Sie sammeln Beobachtungen aus einer simulierten oder realen Umgebung, auf deren Basis sie Entscheidungen treffen und die unter Umständen für einen Lernvorgang genutzt werden können. Oft wird dabei auch von *Artificial Intelligence* (AI) bzw. *Künstliche Intelligenz* (KI) gesprochen. Intelligente Agenten werden häufig in Spielen verwendet, um unterschiedliche Arten von *Nicht-Spieler-Charakteren* (NPCs) zu simulieren. In der Spielwelt haben diese virtuellen Charaktere in der Regel eine visuelle Repräsentation, beispielsweise anhand einer Person oder eines Tieres, mit der der Spieler interagieren kann. Sie können sowohl eine neutrale Rolle einnehmen (Händler, Dorfbewohner), als auch eine dem Spieler gegenüber feindliche oder freundliche Gesinnung aufweisen (Gegner, Gefährte).

Es gibt eine Vielzahl von Technologien die für die Entscheidungsfindung intelligenter Agenten genutzt werden können. Sowohl die Wahl des Verfahrens als auch der aufgebrachte Aufwand spielen eine kritische Rolle für die Authentizität und die erreichte Intelligenz des Agenten. Im Laufe der Jahre haben diese fortwährend an Komplexität gewonnen. Dabei wird der Versuch unternommen, biologische Verhaltensmuster nachzuahmen, um eine Art von pseudo-intelligentem Verhalten zu erschaffen. Die Definition von *Intelligenz* oder *künstlicher Intelligenz* ist dabei allerdings unpräzise, da es keine allgemeingültige Definition gibt.

Nach Russel und Norvig [SR03] könnte bereits eine simple Maschine wie ein Thermostat als intelligenter Agent bezeichnet werden, da er eigenständig auf äußere Einflüsse reagiert. Laut dieser Definition könnten auch Software-Agenten wie Mining-Bots, die scheinbar eigenständig Daten sammeln, als intelligente Agenten bezeichnet werden. Nikola Kasabov [NK98] bietet hingegen eine vollkommen andere Definition intelligenter Agenten anhand einer Menge von Anforderungen an den Agenten:

- Regeln für Problemlösungen müssen inkrementell angepasst werden.
- Die Anpassung muss in Echtzeit stattfinden.
- Eine Selbstanalyse bezüglich des Verhaltens und des Erfolgs muss stattfinden.
- Ein Lernvorgang durch Interaktion mit der Umgebung muss gegeben sein.
- Der Lernvorgang muss mit einer ausreichenden Datenmenge schnell ablaufen.
- Es muss ein Gedächtnis vorhanden sein.
- Parameter für die Realisierung von Kurz- und Langzeitgedächtnis sowie das Vergessen von Erinnerungen müssen gegeben sein.

Kein intelligenter Agent in einem Spiel ist bisher in der Lage, diesen Anforderungen gerecht zu werden. Alternative Definitionen gehen in der Regel nicht über die Anforderung hinaus, dass der Agent selbst handeln muss.

2. Intelligente Agenten in Videospielen

Der Begriff *Intelligenter Agent* wird daher im Folgenden stets als Software-Agent in Umfeld einer simulierten Realität anhand von Videospielen definiert, der selbst Entscheidungen trifft. Ziel dieses Agenten ist eine sinnvolle Entscheidungsfindung zwecks einer Interaktion mit dem Spieler. Dabei soll so weit wie möglich eine scheinbare Imitation menschlichen Verhaltens umgesetzt werden, es werden allerdings keine weiteren Anforderungen an Lernvorgänge oder ein Gedächtnis gestellt. Dafür kann eine analytische, formale Vorgehensweise der Verfahren für die Entwicklung von Algorithmen für Agenten verwendet werden. Alternativ wird der Versuch unternommen, biologische oder kognitive Prozesse zu imitieren, um intelligentes Verhalten zu simulieren. Unabhängig von der gewählten Technologie soll lediglich der Eindruck von Intelligenz erweckt werden, ohne dass dabei tatsächliche Intelligenz erreicht werden muss. Die von Kasabov definierten Kriterien bezüglich eines Lernvorgangs oder einem Gedächtnisvermögen etc. sind daher nur optionale Eigenschaften. Im folgenden Abschnitt werden einige etablierte Methoden für die Entscheidungsfindung intelligenter Agenten vorgestellt.

2.1 Finite State Machines

Bei *Finite State Machines* oder *Endlichen Automaten* handelt es sich um abstrakte Zustandsmaschinen, die sich in jeweils einem von mehreren Zuständen befinden können [DMB04] [Ric] [DM18] [AFP19] [Buc04]. Das Ziel des Verfahrens ist eine reduzierte Abbildung der Wirklichkeit anhand eines Modells, welches durch die State Machine beschrieben wird. Die State Machine stellt also nur den Ansatz einer Simulation der Wirklichkeit dar, sowie auch alle weiteren vorgestellten Technologien für intelligente Agenten. Dafür müssen Zustände und Zustandsübergänge bzw. *States* und *Transitions* definiert werden. Die Zustandsübergänge schaffen Verbindungen in andere Zustände und werden durch bestimmte Konditionen gestartet, sogenannte *Trigger*. Ein populäres Beispiel für Spiele, in denen State Machines zum Einsatz kommen, ist das 1992 veröffentlichte *Wolfenstein 3D* [iS92].

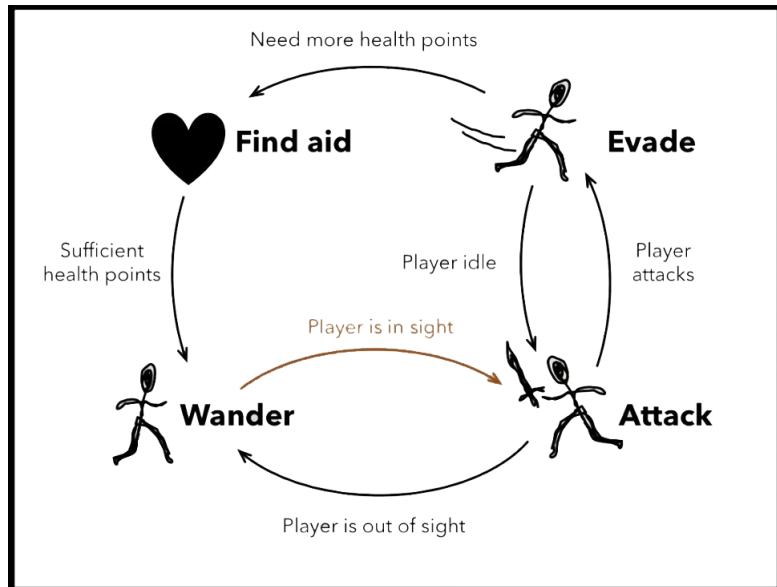


Abbildung 2.1.: Beispielhafte Finite State Machine [LESM19]

Die Abbildung 2.1 zeigt eine Finite State Machine anhand eines simplifizierten Beispiels. Es werden die Zustände „Find aid“, „Wander“, „Attack“ und „Evade“ für einen NPC definiert. Wenn der NPC sich im *Wander*-Zustand befindet, so kann er lediglich in den *Attack*-Zustand wechseln. Der Trigger dafür ist die Sichtbarkeit des Spielers. Sobald die Triggerkondition eintrifft, findet ein Zustandsübergang in den *Attack*-Zustand statt. Für diesen Zustand gibt es nun zwei mögliche

2. Intelligente Agenten in Videospielen

Zustandsübergänge: Wenn der Spieler ebenfalls angreift, so wechselt der NPC in den *Evade*-Zustand und versucht, vor dem Spieler zu flüchten. Wenn der Spieler außer Sicht ist, wechselt er zurück in den *Wander*-Zustand. Aus dem *Evade*-Zustand heraus kann der NPC zurück in den *Attack*-Zustand wechseln, wenn der Spieler sich im *Idle*-Modus befindet, also nichts tut. Weiterhin wechselt der NPC in den *Find Aid*-Zustand, sobald seine Lebenspunkte unter einem bestimmten Wert liegen. Wenn seine Lebenspunkte aufgefüllt wurden findet ein Zustandsübergang in den *Wander*-Zustand statt.

Die folgenden Codebeispiele (siehe Listings 2.1 bis 2.5) zeigen eine mögliche minimalistische Implementierung der State Machine:

```
1 class State {
2     var action;
3     var entryAction;
4     var exitAction;
5     var transitions;
6 }
```

Listing 2.1: State Class

```
1 class Transition {
2     var state;
3     function isActive();
4 }
```

Listing 2.2: Transition Class

```
1 class AttackState extends State {
2     var action = new AttackAction();
3     var entryAction = new MoveAndDrawSwordAction();
4     var exitAction = new SheathSwordAction();
5     var transitions = [new WanderAttackTransition(),
6         new EvadeAttackTransition()];
7 }
```

Listing 2.3: Beispiel-State

```
1 class WanderAttackTransition extends Transition {
2     var state = new AttackState();
3     function isActive() {
4         return isPlayerClose();
5     }
6 }
```

Listing 2.4: Beispiel-Transition

2. Intelligente Agenten in Videospielen

```

1  class StateController {
2      [...]
3
4      function getActions() {
5          activeTransition = null;
6          foreach (transition in currentState.transitions) {
7              if (transition.isActive()) {
8                  activeTransition = transition;
9                  break;
10             }
11         }
12         if (activeTransition != null) {
13             newState = activeTransition.state;
14             actions += currentState.exitAction;
15             actions += newState.entryAction;
16             actions += newState.action;
17             currentState = newState;
18             return actions;
19         } else return [currentState.action];
20     }
21 }

```

Listing 2.5: State Machine Controller

Zunächst werden die Klassen `State` und `Transition` definiert. `State` benötigt die ausführbare Hauptaktion `action`, sowie eine Eingangs- und Ausgangsaktion, wie beispielsweise eine bestimmte Animation, die für einen flüssigen Übergang der Zustände ausgeführt wird. Für den Zustand `AttackState` werden daher entsprechende Aktionen für das Aufnehmen sowie Verstauen eines Schwertes definiert. Die Hauptaktion `AttackAction` sorgt dafür, dass eine Attacke ausgeführt wird. Weiterhin werden zwei Zustandsübergänge `WanderAttackTransition` und `EvadeAttackTransition` definiert, da der NPC entweder aus dem *Wander*- oder aus dem *Evade*-Zustand in den *Attack*-Zustand wechseln kann. Jede Transition verfügt dabei über den Zielzustand und über eine Funktion `isActive`, die angibt, ob die Transition aktiv ist. Die `WanderAttackTransition` definiert daher `AttackState` als Zielzustand und prüft innerhalb von `isActive`, ob ein Spieler in der Nähe ist. Der `StateController` prüft nun mittels der Funktion `getActions` in regelmäßigen Abständen oder nach dem Ausführen von Einzelaktionen durch das Aufrufen der `isActive`-Funktionen für alle Transitions, ob neue Zustandsübergänge verfügbar sind. Sobald eine solche gefunden wird, wird der neue Zustand gesetzt und sowohl die Ausgangsaktion des aktuellen Zustands, sowie die Eingangs- und die Hauptaktion des neuen Zustands werden in die Liste der auszuführenden Aktionen aufgenommen. Ansonsten wird stets die Hauptaktion des aktuellen Zustands weiterhin ausgeführt.

Nach diesem einfachen Schema werden Zustände und ihre Übergänge definiert. Der `StateController` stellt anschließend sicher, dass Zustandsübergänge automatisch eingeleitet werden. In diesem Beispiel wird immer der erste mögliche Zustandsübergang genommen, theoretisch könnte man den Übergängen nun aber Prioritäten zuweisen. Die Architektur und das Anlegen neuer Zustände sind simpel und effizient. Bei der Darstellung aller möglichen Zustände eines echten, komplexen Spiels werden jedoch schnell Grenzen erreicht, da eine hohe Menge von Zuständen und Übergängen erreicht wird. Aufgrund des hohen Determinismus von State Machines in ihrer einfachen Form eignen sie sich weiterhin nicht für den Einsatz in jedem Spiel. Man stelle sich ein Strategiespiel vor, in dem State Machines für NPC-Gegner verwendet werden. Diese Gegner würden jedes Mal auf die gleiche Art und Weise auf bestimmte Situationen reagieren. Das ist zum einen problematisch, weil dadurch statt einem authentischen, dynamischen Verhalten lediglich eine repetitive Spielerfahrung

2. Intelligente Agenten in Videospielen

zustandekommt und zum anderen, weil derartiges Verhalten vorhersehbar ist und daher leicht durch den Spieler ausgenutzt werden kann.

Hierarchical State Machines State Machines haben zusätzlich das Problem, dass die Anzahl der Zustände schnell große und daher unübersichtliche Ausmaße annehmen kann. Außerdem ist es nur möglich, einen einzelnen Zustand zur Zeit einzunehmen, wodurch nur sehr eingeschränktes Verhalten realisiert werden kann. Möchte man beispielsweise modellieren, dass ein NPC zur selben Zeit verwundet ist und gleichzeitig den Spieler angreift, muss dafür ein eigener neuer Zustand *IsAttackingWhileWounded* entworfen werden. Wenn man erreichen will, dass der Spieler in jedem bereits entworfenen Zustand ebenfalls verwundet sein kann, muss dann eine Vielzahl neuer Zustände entworfen werden. So resultiert die Kombination aus allen möglichen Zuständen für die Erstellung neuer Sub-Zustände und Sub-Transitions oft in vollkommen unbrauchbaren Zustandsmengen, die nicht mehr verwaltet werden können.

Eine mögliche Lösung für diese Probleme ist die Entwicklung von State Machines in einer hierarchischen Struktur. Dafür gibt es eine Anzahl von *High-Level*-Zuständen, die einen allgemeinen, übergeordneten Kontext für das Charakterverhalten darstellen. In jedem dieser High-Level-Zuständen gibt es dann mehrere *Sub-Level*-Zustände, die für spezifischeres Verhalten sorgen. Solche Systeme werden als *Hierarchical State Machines* bezeichnet [DM18]. Die Struktur des Verfahrens lässt gewisse Gemeinsamkeiten zu UML-Diagrammen erkennen, in denen ebenfalls hierarchisch geschachtelte Gruppen von Aktionen gebildet werden.

Als weiteres Beispiel wird ein Agent angeführt, der die Zustände „On Guard“, „Fight“ und „Run Away“ mit Transitions untereinander besitzt. Jetzt soll ein neuer Zustand „Get Food“ eingeführt werden, der zu jeder Situation eintreten kann, sobald der Agent hungrig ist. Von jedem möglichen Zustand müsste dann ein Übergang zu dem neuen Zustand geschaffen werden (siehe Abbildung 2.2).

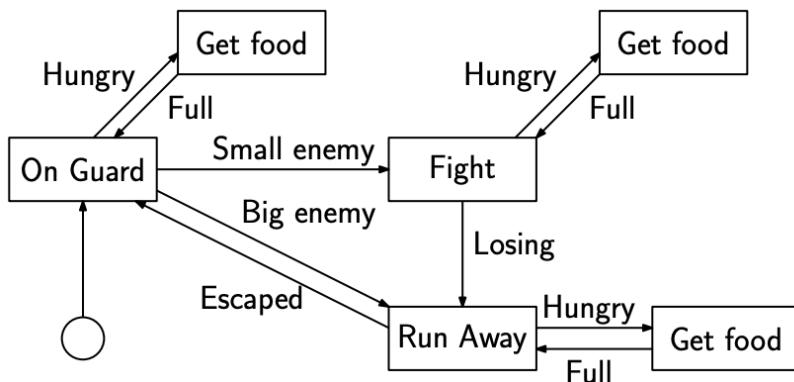


Abbildung 2.2.: Hinzufügen eines neuen Zustands, normale State Machine

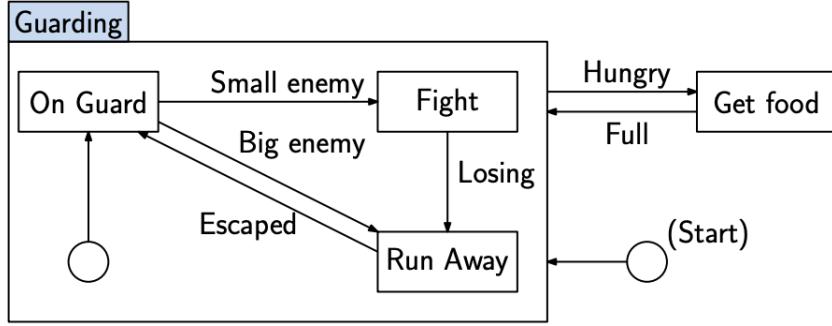


Abbildung 2.3.: Hinzufügen eines neuen Zustands, hierarchische State Machine

Abbildung 2.3 zeigt den gleichen Vorgang des Ergänzens um einen weiteren Zustand mit einer hierarchischen State Machine, statt einer herkömmlichen. Es wird ein neuer High-Level-State *Guarding* gebildet, der die bereits vorhandenen Zustände enthält. Dieser neue „Super-State“ verfügt nun über einen Startpunkt, der nur dann angesprochen wird, wenn man das erste mal den Super-State betritt. Die Zustandsübergänge von jedem Sub-Zustand des Super-States sind implizit gegeben. Das könnte beispielsweise funktionieren, indem ein Stack verwendet wird, der die aktiven Zustände nach ihren Level sortiert, sodass der Zustand mit dem höchsten Level immer an der ersten Stelle des Stacks steht. Um mögliche Transitions zu prüfen, würde dann vom letzten bis zum ersten Element geprüft werden, ob ein Zustand für die bestimmte Aktion verfügbar ist [DM18].

Die Architektur der State Machine wird dadurch simplifiziert und modularisiert, da insgesamt deutlich weniger Zustände verwaltet und erzeugt werden müssen. Weiterhin können insgesamt mehr Zustände verarbeitet werden, weil die Transitions vieler Zustände automatisch implizit definiert werden.

2.2 Decision Trees

Ein klassisches Verfahren für die Entscheidungsfindung intelligenter Agenten beruht auf der Verwendung von *Decision Trees*. Dabei wird eine einfache Baumstruktur aus möglichen Entscheidungen gebildet. Genauer handelt es sich um einen gewurzelten Baum. Häufig findet man binäre Bäume vor. Das Verfahren ist besonders sinnvoll, wenn der Agent aus vielen möglichen Entscheidungen zur selben Zeit auswählen muss. Je nachdem welche Entscheidung gewählt wurde, öffnet sich ein neuer Zweig des Baumes, unter Umständen mit weiteren Entscheidungen und möglichen Abstiegen. Die Knoten des Baumes werden als „Decision Point“, also *Entscheidungsknoten*, bezeichnet. Jeder Entscheidungsknoten prüft eine bestimmte binäre Bedingung, beispielsweise: *Ist der Spieler sichtbar?*. Damit wird eine Bedingung für den Abstieg des Baumes aufgestellt. Die Antwort auf die Bedingung ist dafür verantwortlich, ob ein Abstieg in den linken oder rechten Zweig des Entscheidungsknotens erfolgt. Bei einem binären Baum handelt es sich um „Ja-oder-Nein“-Fragen. Abbildung 2.4 zeigt ein Beispiel für einen typischen, stark simplifizierten Entscheidungsbaum für einen kämpfenden NPC.

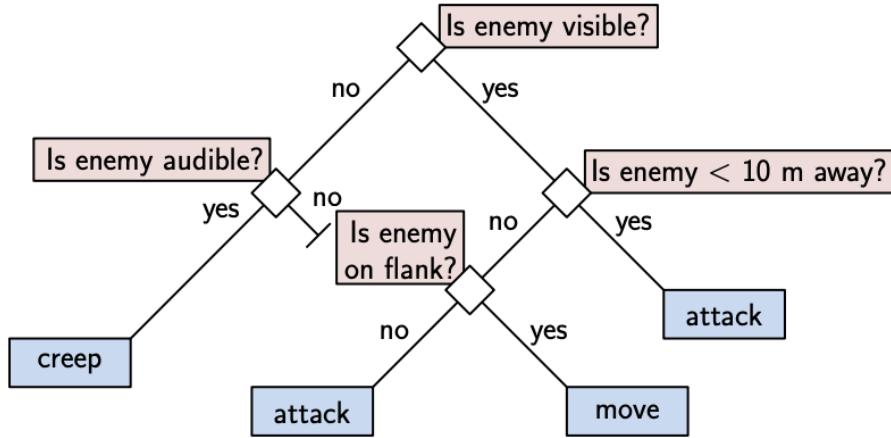


Abbildung 2.4.: Beispielhafter Decision Tree [DM18]

Decision Trees sind einfach zu verstehen und können leicht implementiert werden. Zusätzlich bieten sie eine hohe Performanz. Neben der Entscheidungsfindung von NPCs haben sie noch weitere Anwendungsbereiche in Videospielen. Beispielsweise werden Bäume für Questsysteme eingesetzt. Das Spiel *Witcher 3* [RED15] verwendet beispielsweise Decision Trees, um je nach den Handlungen des Spielers im Spiel den Ablauf vom Ende des Spiels festzulegen.

2.3 Behavior Trees

Behavior Trees sind eine Weiterentwicklung von Finite State Machines, bei der Baumstrukturen genutzt werden. Sie gehen von einem modularen komponentenbasierten System für das Verhalten bzw. *Behavior* von Agenten aus und verwendet dafür gewurzelte Bäume [DM18] [DMB04] [Sim14]. Der erste Einsatz von Behavior Trees erfolgte im Jahr 2004 im erfolgreichen Spiel *Halo 2* [Isl05] [Stu04]. Ein beispielhafter Behavior Tree ist in Abbildung 2.5 zu sehen.

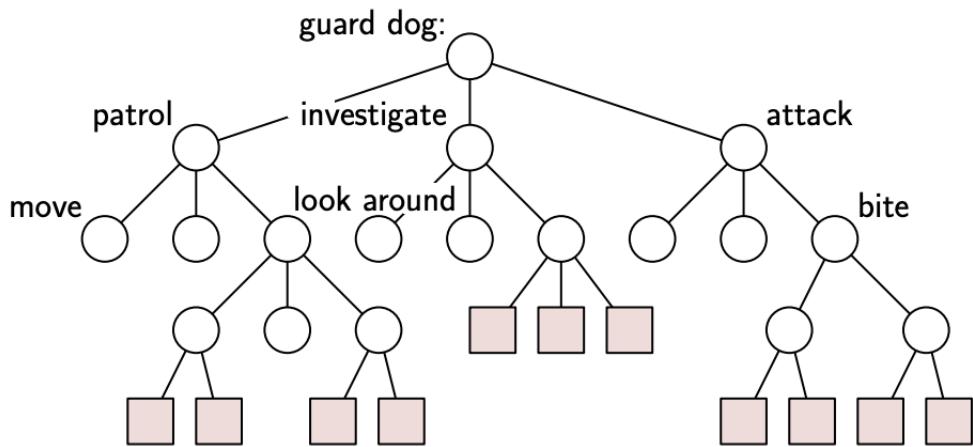


Abbildung 2.5.: Beispielhafter Behavior Tree [DM18]

Die Knotenpunkte des Baumes enthalten sogenannte *Tasks* - spezifische Aufgaben, die ein explizites Ziel verfolgen. Tasks besteht aus einer Menge von Konditionen die bestimmen, welcher Task aktiv

2. Intelligente Agenten in Videospielen

ist, sowie Aktionen, die die Ausführung des Tasks enkodieren. Da Tasks fehlschlagen können, geben diese bei der Ausführung einen von drei Statusberichten an: *Running*, *Success* oder *Failure*. Alle im Baum enthaltenen Tasks sind Akteuren des Spiels zugewiesen. Die Granularität der Tasks erhöht sich mit steigender Tiefe des Baumes. Jeder Task beschreibt dabei ein Sub-Verhalten, das eine bestimmte Aufgabe im Kontext des übergeordneten Knotens übernimmt. Die Wurzel des Baumes beschreibt das generelle Verhalten für eine bestimmte Entität. Sie könnte beispielsweise das Verhalten *Gegner* oder *Wachhund* darstellen. Einzelne Top-Level-Tasks (Knoten direkt unter der Wurzel) beschreiben allgemeines Verhalten, wie beispielsweise „Angreifen“, „Umschauen“ oder „Patrouillieren“. Die Sub-Tasks von *Angreifen* könnten zum Beispiel *Schlagen* oder *Schießen* sein.

Die Blätter des Baumes beschreiben die tatsächliche Interaktion der Entität mit dem Spiel und die Ausführung von Aktionen. Sie beinhalten Konditionen, die erfüllt sein müssen, und Aktionen, die dann ausgeführt werden. Für das Beispiel des Wachhundes könnten die Konditionen den Zustand des Hundes (*Ist der Hund hungrig?* *Ist der Hund verletzt?*) oder den allgemeinen Spielzustand betreffen (*Ist ein Gegner in der Nähe?*). Die mit den Konditionen verbundenen Aktionen verändern dann dessen Zustand - der Hund könnte beispielsweise den Spieler beißen und seine Lebenspunkte verringern oder einfach eine Animation abspielen. Die Konditionen können auch als Filter betrachtet werden, die bestimmen, welche Aktionen ausgeführt werden [DM18].

Task Composition Behavior Trees bieten verschiedene Möglichkeiten für die Komposition von Tasks. Typischerweise werden Tasks entweder sequentiell oder ausgewählt abgearbeitet.

Bei der sequentiellen Ausführung von Tasks wird eine Liste von Tasks nacheinander verarbeitet (*Sequences*). Sobald ein Task fehlschlägt, wird die Sequenz terminiert. Eine beispielhafte Ausführung wird anhand der Abbildung 2.6 verdeutlicht. Der Pfeil in der Grafik steht für die sequenzielle Ausführung der Aktionen.

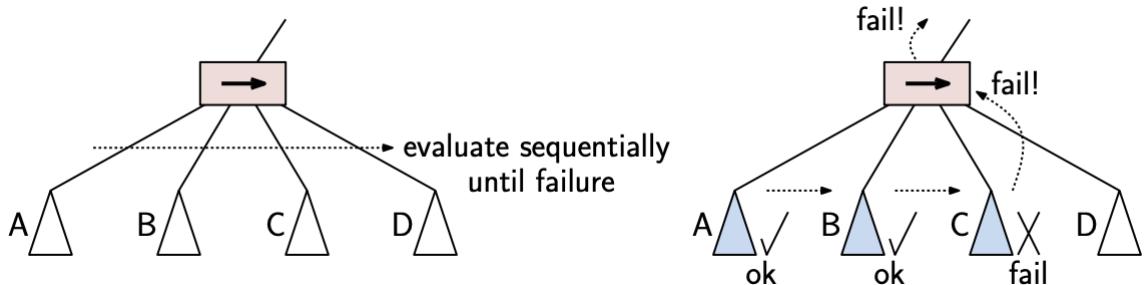


Abbildung 2.6.: Sequenzielle Ausführung in Behavior Tree [DM18]

Bei der ausgewählten Ausführung (*Selector*) wird maximal ein Task aus einer Menge von Tasks ausgewählt. Sobald der erste Task einen erfolgreiche Statusbericht versendet, wird die Ausführung terminiert. Abbildung 2.7 zeigt eine beispielhafte ausgewählte Ausführung. Das Fragezeichen verdeutlicht in der Abbildung den Selector, eine ausgewählte Ausführung bis zum ersten Erfolg.

2. Intelligente Agenten in Videospielen

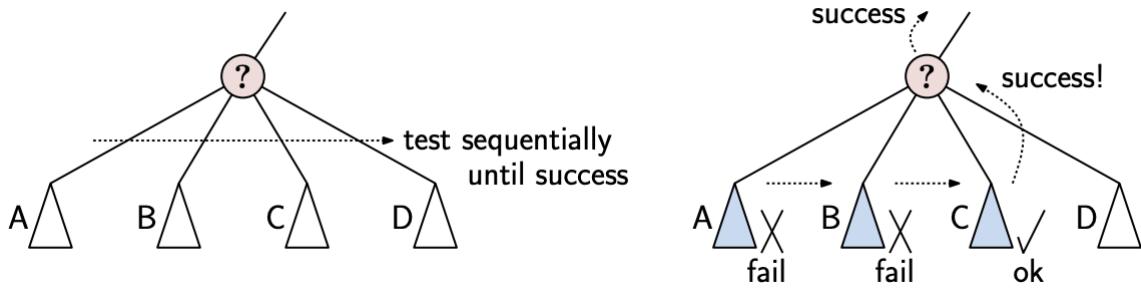


Abbildung 2.7.: Ausgewählte Ausführung in Behavior Tree [DM18]

Mit Sequences und Selector werden den Behavior Trees einige Möglichkeiten hinzugefügt, die für Finite State Machines nicht verfügbar sind. Sequences und Selector können kombiniert werden, um ein komplexeres Verhalten zu realisieren. Einzelne Teilbäume dieses Verhaltens können dann auf Sequences und andere Teilbäume auf Selector basieren. In Abbildung 2.8 ist ein konkretes Beispiel für eine Kombination von Selector und Sequence zu erkennen, in dem ein NPC auf verschiedene Art und Weise je nach Beschaffenheit des Spielzustands versucht, in einen Raum zu gelangen. Anhand der Symbole in der Grafik („!“ und „?“) ist die Kombination von Selector und Sequence erkennbar. Der Agent betritt unmittelbar den Raum, wenn die Tür offen ist. Falls dies nicht gelungen ist, bewegt er sich zunächst zur Tür und versucht anschließend, diese zu öffnen. Wenn auch dies scheitert, wird die Tür aufgebrochen, sodass der Agent schlussendlich den Raum betreten kann.

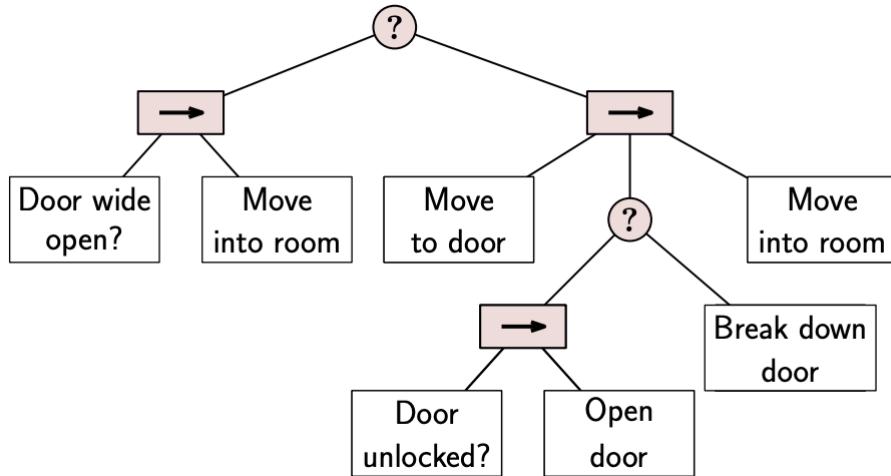


Abbildung 2.8.: Kombination sequenzieller und ausgewählter Ausführung in Behavior Tree [DM18]

Aus der Sicht des Software-Engineering werden dem Programmierer mit Behavior Trees ein besser strukturierter Kontext für die Entwicklung von Verhalten zur Verfügung gestellt [DM18]. Da sie alle Vorteile von Finite State Machines bieten und darüber hinaus über eine hohe Performanz verfügen, sind Behavior Trees zu einer sinnvollen Alternative zu State Machines geworden [QZ18]. Sie werden daher zunehmend in großem Maßstab in erfolgreichen und komplexen Spielen wie *Halo*, *Bioshock* oder *Spore* verwendet [Wik19a].

2.4 Goal Oriented Action Planning (GOAP)

GOAP ist ein System für die Planung und Ausführung von Aktionen für einen Agenten und wurde zunächst für das Spiel *F.E.A.R.* entwickelt [Ork]. Es erlaubt einem Agenten die Planung einer Sequenz von Aktionen zum Erreichen eines bestimmten Ziels. GOAP bietet gegenüber Finite State Machines den Vorteil, dass keine großen Mengen von Zuständen und Zustandsübergängen definiert werden müssen [Owe14] [Buc04]. Anstelle von Zuständen werden hier modulare, voneinander entkoppelte Aktionen verwendet [Nie14].

GOAP verlangt für die Entscheidungsfindung lediglich die Definition von *Goals* und *Aktionen* [DMB04]. Goals sind festgelegte Konditionen, die ein Agent erfüllen möchte. Um ein Goal zu erreichen, werden Aktionen ausgeführt, die den Spielzustand verändern. Eine Kette aus Aktionen wird aufgebaut, die den Spielzustand so verändern, dass die Konditionen des Goals erfüllt sind. Bei einer Aktion kann es sich beispielsweise um das simple Öffnen einer Tür oder das Aufnehmen eines Gegenstands in das Inventar des NPCs handeln. Die vom Agenten ausgeführten Aktionen haben jeweils eine Menge von Bedingungen (*Preconditions*), die für die Ausführung der Aktion vorausgesetzt werden, sowie eine Menge von Effekten als eine Folge der Aktion. Diese sorgen dafür, dass die Bedingung eines Goals schlussendlich erfüllt wird.

Aus den einzelnen Aktionen erstellt der sogenannte *GOAP Planner* eine Sequenz zum Erreichen eines Ziels. Es handelt sich um eine zentrale Verwaltungs- und Planungseinheit für die Ausführung von Aktionen im GOAP-System. Der GOAP Planner kennt die Bedingungen der Goals sowie die Preconditions und Effekte der Aktionen. Auf dieser Grundlage ist er in der Lage zu erkennen, welche Effekte welcher Aktionen benötigt werden, um das Goal zu erreichen. Dadurch kennt der GOAP Planner auch die Preconditions der benötigten Aktionen und kümmert sich ebenfalls darum, dass diese Bedingungen erfüllt werden. Dafür werden unter Umständen weitere Aktionen in die Sequenz eingefügt. Der Planner stellt sicher, dass die benötigten Ketten von Aktionen in der richtigen Reihenfolge ausgeführt werden. Konkret sucht der GOAP Planner zunächst nach allen vorhandenen Goals. Für diese erstellt er rückwärts vom Goal aus gehend eine Baumstruktur. Diese besteht aus allen möglichen Sequenzen von Aktionen, die zum jeweiligen Ziel führen. Dabei wird der jeweils kürzeste Weg gewählt, der zum Ziel führt. Die Pläne des Agenten sind nicht statisch, sondern werden erst zur Laufzeit generiert, sodass dieser sich dynamisch an die jeweiligen Umstände seiner Umgebung anpassen kann. In der Abbildung 2.9 ist eine beispielhafte Sequenz in einem GOAP-System zu erkennen.

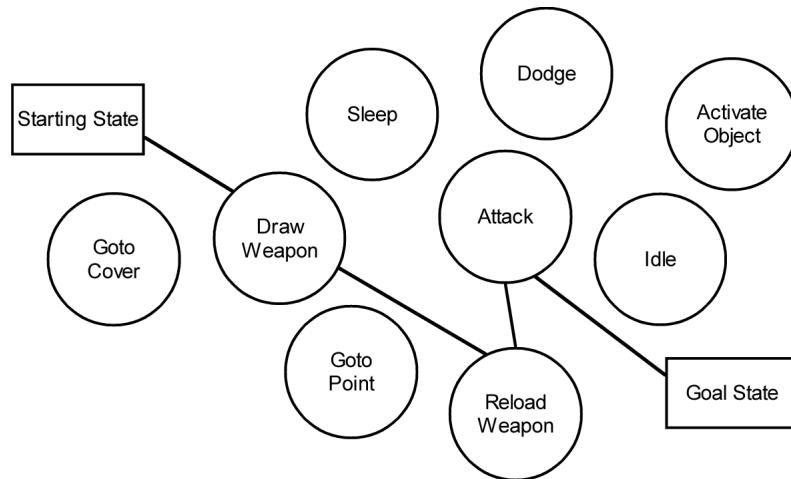


Abbildung 2.9.: GOAP-Sequenz nach Orkin [Ork]

2. Intelligente Agenten in Videospielen

Es sei darauf geachtet, dass keine Übergänge zwischen den Aktionen vorhanden sind. Die Abbildung zeigt das GOAP-Äquivalent einer State Machine mit übergangslosen Aktionen statt Zuständen und Transitions. Intern verwendet GOAP eine minimale State Machine, die über drei Zustände verfügt: „Idle“, „MoveTo“ und „PerformAction“. Das ist notwendig, da sich der Agent für einige Aktionen in der Nähe eines bestimmten Objekts befinden muss. Das Bewegen des Agenten ist nicht Teil des eigentlichen GOAPs, weil sonst viele zusätzliche Aktionen für die Bewegung zu bestimmten Zielen von Aktionen definiert werden müssten. Stattdessen kann jeder Aktion bei Bedarf ein Ziel zugewiesen werden. Im *Idle*-Zustand hat der Agent eine vorherige Sequenz von Aktionen abgeschlossen. Der *GOAP Planner* berechnet nun die nächste Sequenz für das Erreichen eines Ziels. Der Agent wechselt dann in den *PerformAction*-Zustand. Vor dem Ausführen jeder Aktion wird geprüft, ob diese über eine Zielposition verfügt und ob der Agent sich nah genug an dieser Position befindet. Falls dies nicht der Fall ist, wechselt der Agent in den *MoveTo*-Zustand und anschließend wieder in den *PerformAction*-Zustand. In diesem Zustand führt der Agent eine GOAP-Aktion aus.

Die Vorteile von GOAP liegen in der Modularität des Systems. Das Hinzufügen und Austauschen von Aktionen ist ohne weiteres möglich. Der GOAP-Entwickler Orkin nennt ein Beispiel aus der Entwicklung von *F.E.A.R.*: Es wurde zum Ende der Entwicklungsphase die Bedingung gestellt, dass NPCs beim Betreten von Räumen stets das Licht anschalten sollen. Mit GOAP musste lediglich eine Aktion *TurnOnLightsAction* mit einem neuen Effekt *LightsOn* erstellt werden, der auch eine Precondition für die *GoToRoomAction* ist [Ork]. Mit einer State Machine wäre dieser Vorgang komplexer gewesen und hätte deutlich mehr Zeit gekostet. Ein weiterer Vorteil ist, dass bei der Verwendung desselben Goals bei unterschiedlichen Agenten eine völlig unterschiedliche Sequenz von Aktionen erstellt werden kann, da diese vom Zustand des Spiels abhängen, sodass die Agenten dynamischer und realistischer wirken [Owe14].

2.5 Utility Based Agents

Utility Based Agents beruhen auf einem AI-System namens *Utility-Based Action Bias*. Es basiert auf einer simplen Idee: Die Agenten wechseln nicht wie z.B. bei Finite States Machines zwischen einer endlichen Menge von Zuständen, die durch bestimmte Trigger ausgelöst werden, sondern bewerten kontinuierlich die ihnen zur Verfügung stehenden Aktionen basierend auf *Utility-Functions* $U(s_t)$ [SR03] [Ait13]. Diese Funktionen bestimmen, wie groß der daraus hervorgehende Vorteil für den Agenten ist - sie bestimmen also in gewisser Weise die *Präferenzen* eines Agenten in Abhängigkeit von seinem Zustand s_t . Der Agent wählt stets die Aktion aus, die ihn in den Zustand mit dem höchsten Utility-Wert bringt. Die Effekte von Aktionen sind bekannt, sodass der Utility-Wert für einen neuen Zustand leicht berechnet werden kann.

In der *Utility Theory* wird davon ausgegangen, dass es eine Menge von Utility-Functions gibt, die nicht direkt mit Aktionen gekoppelt werden. In diesem Fall müssen alle möglichen neuen Zustände s_{t+1} betrachtet werden, die aus den möglichen Aktionen a aus der Menge aller Aktionen \mathbb{A} hervorgehen. Da der Utility-Wert maximiert werden soll wird die Aktion a ausgewählt, die den höchsten Wert $U(s_{t+1})$ erzielt, wie von Russel und Norvig beschrieben [SR03]:

$$action = \arg \max_a U(s_{t+1}) \quad \forall a \in \mathbb{A}$$

Russell und Norvig unterscheiden weiterhin zwischen deterministischen und nichtdeterministischen Umgebungen des Agenten. Der Einfachheit halber wird diese Unterscheidung an dieser Stelle vernachlässigt, da sie in den meisten Fällen nicht relevant ist. Wenn Utility-Based Agents in Spielen zum Einsatz kommen, werden Aktionen - anders als in der Literatur - meistens direkt mit ihrer eigenen Utility-Function gekoppelt. Jeder Aktion kann dann unmittelbar ein Utility-Wert zugewiesen

2. Intelligente Agenten in Videospielen

werden, ohne dass zukünftige Zustände betrachtet werden. Man stelle sich beispielsweise eine Aktion vor, die die Waffe eines Agenten nachlädt. Die dazugehörige Utility-Funktion „Waffe nachladen“ sinkt mit der Menge der Munition in der Waffe. Je niedriger der aktuelle Utility-Wert ist, desto höher ist der Utility-Gewinn beim Ausführen der Aktion.

Die Utility-Funktionen können vom Entwickler beliebig definiert werden, sodass sie dem Agenten ein dynamisches Verhalten verleihen. Beispielsweise könnte eine Utility-Funktion für die Benutzung eines Heiltranks so definiert werden, dass der Utility-Wert sich mit sinkender Anzahl der Lebenspunkte des Agenten erhöht. Somit steigt die Wahrscheinlichkeit für die Benutzung von Heiltränken mit sinkender Anzahl der Lebenspunkte, wie in Abbildung 2.10 zu erkennen. Dafür wurde die Funktion $f(x) = (1 - (x/100))^4$ verwendet.

Utility-Funktion, Heiltrank

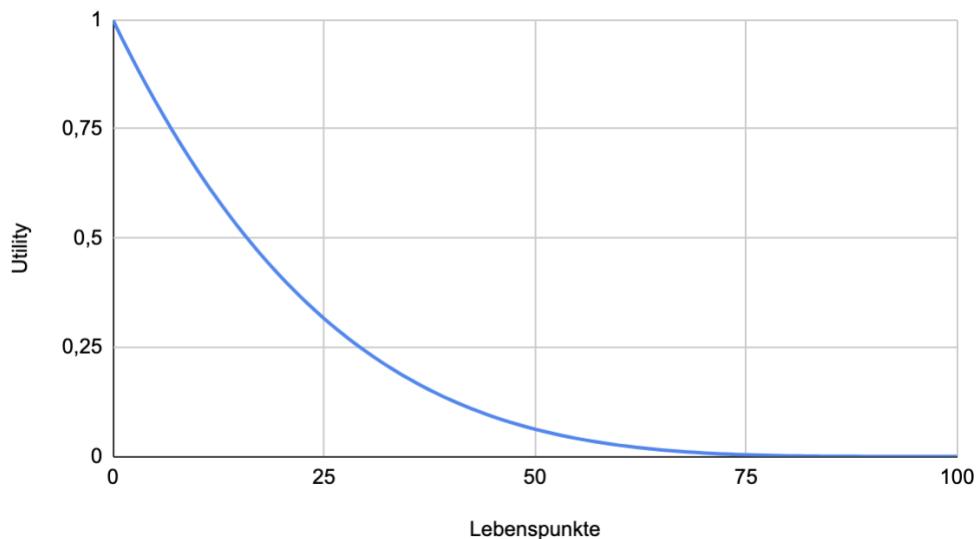


Abbildung 2.10.: Utility-Funktion für Benutzung eines Heiltranks

Das Resultat der Verwendung von Utility-Funktionen für das Auslösen bzw. die Auswahl von Aktionen ist die Abbildung eines deutlich dynamischeren Verhaltens als beispielsweise bei der Verwendung einer Finite State Machine.

Eine Variante von Utility-Based Agents sind *Needs-Based Agents* [Zub]. Die Utility-Funktionen in dieser Variante basieren auf individuellen Motivationen, die menschlichen Bedürfnissen nachempfunden wurden; beispielsweise „Hunger“, „Durst“, „Schlaf“ oder „Kommunikation“ [Zub]. Damit soll ein möglichst menschliches Verhalten simuliert werden.

Mit dem Verfolgen von Bedürfnissen haben Utility-basierte Agenten nicht nur Wurzeln im Design von AI-Systemen, sondern mit der Utility Theory ebenfalls in der Psychologie und Soziologie [KD09]. Durch die beliebige Gestaltung von Bedürfnisfunktionen kann das Verhalten von Agenten verbessert werden. Besonders im Vergleich zu klassischen, statischen Methoden wie Finite State Machines kann daraus ein sinnvollereres, nachvollziehbares Entscheidungsverhalten resultieren. Ein populäres Beispiel für die Verwendung von Utility Based Agents (und in diesem Spezialfall Needs Based Agents) ist das Spiel *Die Sims 4* [TSS14]. Das Spiel stellt eine Simulation von menschlichen Bedürfnissen und Beziehungen sowie des menschlichen Alltags dar. Es verwendet Utility-Funktionen für die Repräsentation des internen Zustands der NPCs, um ein annähernd menschliches Verhalten zu erreichen [Orf14].

3

Reinforcement Learning

Das vorherige Kapitel 2 beschäftigt sich mit der Entwicklung intelligenter Agenten zur Erschaffung von künstlicher Intelligenz für virtuelle Entitäten in Videospielen. Der Fokus dieses Kapitels liegt in der Erläuterung von Reinforcement Learning (RiL) als eine alternative Methode. Die Nutzung von Machine Learning stellt den logischen nächsten Schritt zu den bereits vorgestellten Methoden für die Realisierung *lernender* intelligenter Agenten in Videospielen dar. Im Gegensatz zu klassischen Methoden werden hier also Agenten verwendet, die aufgrund ihrer gesammelten Erfahrungen einen Lernprozess durchlaufen. Das Kapitel dient als vorbereitendes Element für die Implementierung von lernenden intelligenten Agenten in einem Dorfssystem als zentrales Thema der vorliegenden Thesis.

Insgesamt wird ein starker Fokus auf RiL gesetzt. Zunächst wird dafür eine Gegenüberstellung zwischen klassischen Machine Learning-Verfahren und RiL vorgenommen. Es folgt eine Einordnung der Verfahren in einen übergeordneten Kontext sowie die Beschreibung aktueller Entwicklungen in diesem Bereich. Anschließend werden *Markov Decision Problems* (MDPs) als elementare Problemstellung für RiL diskutiert. Ein Menge konkreter modellfreier und modellbasierter Methoden zur Implementierung des Verfahrens werden ausführlich erläutert. Dafür werden nötige mathematische Grundlagen geschaffen.

Der erste Abschnitt beinhaltet eine Rekapitulation zur grundsätzlichen Funktionsweise von *Machine Learning*. Zur Einordnung des Themas in einen übergeordneten Kontext findet anschließend eine Einordnung der klassischen Teilbereiche von Machine Learning und RiL statt.

3.1 Einordnung

Maschinelles Lernen beschäftigt sich mit der Entwicklung eines Systems, das aus einer Menge von beliebigen Inputdaten einen gewünschten Output erzeugt. Dabei wird der Versuch unternommen, durch eine Verbesserung des Outputs in Abhängigkeit vom Input einen Lerneffekt zu erzielen. Diese Fähigkeit wird erlangt, indem das System mit relevanten Informationen gespeist wird, aus denen Assoziationen gebildet und Muster erkannt werden können. Dadurch können schlussendlich zukünftige Outputs verbessert werden, ohne dass dieser Vorgang spezifisch definiert wird. Der Lernvorgang passiert nicht durch die Nutzung eines statischen Algorithmus und wird nicht explizit programmiert, sondern wird durch die automatische Anpassung von Werten und Gewichtungen beschrieben, die dem System zugrunde liegen. Häufig werden hierfür neuronale Netze verwendet. Man entscheidet dabei unter anderem zwischen *Feedforward*-Netzen und *rekurrenten* Netzen. Feedforward-Netze verbinden stets Neuronen aus vorderen Schichten mit Neuronen aus hinteren Schichten. Rekurrente neuronale Netze erlauben auch umgekehrte Verbindungen von vorderen Neuronen zu hinteren Neuronen. So entsteht eine Rückkopplung, die zeitliche Informationen aus zuvor getätigten Entscheidungen mit einbeziehen kann. Ein System, das einen Input manipuliert, wird also so lange verändert, bis es für jeden Output den gewünschten Input erzeugen kann. Anstatt Beispiele auswendig zu lernen wird ein statistisches Modell erzeugt, das auch unbekannte Daten beurteilen kann. Dieses Modell wird durch

3. Reinforcement Learning

eine Menge von Parametern beschrieben, in der Regel handelt es sich dabei um die Gewichtungen der einzelnen Neuronen in einem neuronalen Netz.

Weiterhin sollte geklärt werden, was als Input und als Output für das System genutzt wird. Wir gehen im Folgenden davon aus, dass ein einfaches neuronales Netz verwendet wird, dessen Struktur für die Lösung des folgenden Problems irrelevant ist. Um den Lernvorgang eines Agenten zu ermöglichen, müssen für den Input Werte gewählt werden, die für die Problemstellung relevant sind. Dabei kann es sich um beliebige Informationen aus dem Weltzustand des Spiels handeln. Im Folgenden wird ein Beispiel betrachtet, das indirekt bereits das Vorgehen beim RiL beschreibt, um das generelle Vorgehen von maschinellem Lernen zu verdeutlichen.

Ein Agent soll lernen, auf einer sich bewegenden Plattform zu balancieren. Für diese Aufgabe gibt es bestimmte relevante Parameter - beispielsweise die Orientierung der Plattform sowie die Orientierung des Agenten. Unter der Annahme, dass keine weiteren äußeren Parameter die Problemstellung beeinflussen, werden diese beiden Vektoren als Inputparameter für das neuronale Netz verwendet. Zusätzlich muss definiert werden, was die Outputparameter des neuronalen Netzes steuern sollen. In diesem simplen Szenario ist es offensichtlich, dass für das Balancieren die Orientierung des Agenten angepasst werden muss. Nun findet ein Lernvorgang statt, indem durch eine kontinuierliche Änderung der Strategie des Agenten durch die Anpassung seines Modells Beziehungen zwischen Input und Output hergestellt werden können. Der Agent sammelt *Erfahrungen*, indem er über einen langen Zeitraum versucht, das Problem zu lösen und währenddessen immer wieder von der Plattform fällt. Einige Entscheidungen führen jedoch dazu, dass der Agent sich länger auf der Plattform halten kann als andere. Es werden Assoziationen zwischen der Orientierung der Plattform und des Agenten sowie dem Erfolg der jeweils gesammelten Erfahrungen gebildet. Das Modell des Agenten bzw. dessen Strategie als Reaktion auf die Inputparameter wird kontinuierlich angepasst, bis der Agent das Balancieren beherrscht. Man spricht hier vom *Training* eines neuronalen Netzes. Der Trainingsvorgang unterscheidet sich dabei je nach verwendeter Technologie. Es wird weiterhin zwischen *Training* und *Inference* unterschieden. Die Inference beschreibt die Nutzung eines bereits trainierten Machine Learning-Modells zur Generierung eines Outputs. Dabei werden im Unterschied zum Training keine Gewichtungen mehr angepasst. Das fertige Modell wird nicht verändert, sondern lediglich genutzt. Generell unterscheidet man zwischen einem überwachten Training mithilfe von bereits bekannten Inputdaten, denen ein bekannter Output zugeordnet wurde oder einem unüberwachten Training, für das derartige Daten nicht vorhanden sind. Weiterhin wird in der vorliegenden Arbeit RiL als zusätzliche Kategorie angeführt. Im Folgenden werden zunächst unüberwachte und überwachte Methoden näher betrachtet.

3.1.1 Überwachtes Lernen

Das *überwachte Lernen* (*supervised learning*) basiert auf der Verwendung von Trainingsdaten, die über bestimmte *Label* verfügen. Diese Label sind Werte oder Einordnungen in Klassen, die ein bestimmtes Beispiel aus dem Datensatz beschreiben und über die jedes Beispiel des Datensatzes verfügen muss. Beispielsweise könnte es sich um eine Menge von Bildern von Pflanzen handeln, wobei jedes Bild mit dem Namen der Pflanze versehen ist. Diese Klassifizierungen sind bereits vor dem Trainingsvorgang bekannt. Es ist weiterhin genau bekannt, was der Output des Verfahrens ist, nämlich eine Klassifizierung der Beispiele in Klassen von Pflanzen. Häufig wird diese Methode daher für Klassifizierungsaufgaben verwendet. Dabei müssen allgemeine Muster, Gemeinsamkeiten und Unterschiede zwischen den Beispielen identifiziert werden, sodass das Verfahren aus den gefundenen Merkmalen lernt, Label aus Bildinformationen vorherzusagen. Anders formuliert ist der Input in das neuronale Netz eine Menge von Bildinformationen, der Output ist jeweils ein Label bzw. eine Klasse. Zusätzlich kann überwachtes Lernen für Regressionen verwendet werden. Es gilt, folgende Frage zu beantworten: „Wenn eine Variable x einen bestimmten Wert hat, welchen Wert hat dann

$y?$ “ Ein mögliches Anwendungsszenario wäre ein neuronales Netz, das den Preis einer Wohnung in Hamburg anhand ihrer Lage, Ausstattung und Größe bestimmt.

3.1.1.1 Imitation Learning

Bei *Imitation Learning* (*Imitation Learning*) handelt es sich um eine Kombination aus RiL und überwachtem Lernen, bei der bereits vorhandene Daten verwendet werden, um Sequenzen von zukünftigen Aktionen vorherzusagen [YY18]. Imitation Learning wird häufig in Spielen verwendet um intelligente Agenten zu erschaffen die versuchen, die Aktionen eines Spielers zu imitieren. Man spricht auch von *Behavioral Cloning*. Als Trainingsdaten können beispielsweise Aufzeichnungen des Spielverhaltens eines echten Spielers dienen. Sowohl die Beobachtungen aus dem Spielzustand als auch die daraus resultierenden Entscheidungen des Spielers anhand seines Hardwareinputs sind gegeben, um eine Imitation zu ermöglichen. Gegeben sind also sowohl Beobachtungen, als auch die resultierenden Aktionen; nun muss der Zusammenhang zwischen den Beiden gelernt werden. In der Rennspielreihe *Forza Motorsport* wurde Imitation Learning erfolgreich eingesetzt, um sogenannte „Drivatars“ [Stu05] zu erstellen. Dabei handelt es sich um Agenten, die genau das Fahrverhalten von spezifischen Spielern imitieren sollen.

3.1.2 Unüberwachtes Lernen

Trainingsdatensätze sind nicht für jedes Machine Learning-Problem gegeben. Tatsächlich ist das Fehlen von gelabelten Daten häufig ein Problem, da das Anfertigen solcher Datenmengen eine lange Zeit in Anspruch nehmen würde oder in bestimmten Szenarien gar nicht möglich ist. Beim *unüberwachten Lernen* bekommt ein neuronales Netz während des Trainingsvorgangs eine Menge von nicht gelabelten Daten ohne spezielle Anweisungen. Anhand von Labels ist es hier also nicht möglich, bestimmte Merkmale in den Trainingsdaten zu erkennen. Stattdessen werden alternative Ansätze gewählt, zum Beispiel:

- **Clustering:** Bestimmte Charakteristiken oder *Features* werden aus den Daten extrahiert. Dabei kann es sich im oben genannten Pflanzenbeispiel um die Größe der Blätter, Farbe der Blüten oder die Form handeln. Die Beispiele werden anhand von Ähnlichkeiten angeordnet, also „geclustered“.
- **Anomaly Detection:** Anomalien in den Features werden erkannt, um „Ausreißer“ zu erkennen. Wenn beispielsweise alle Pflanzen im Datensatz gelbe Blüten haben außer Eine, so würde Diese als Anomalie erkannt werden.

Im Gegensatz zum überwachten Lernen ist der Output des Modells unbekannt. Weiß man beispielsweise beim überwachten Lernen genau, dass als Output eine Klassifizierung der Beispiele in eine bestimmte Anzahl von Pflanzenkategorien vorgenommen wird, so muss beim unüberwachten Lernen der Output genau analysiert werden, um ein Verständnis darüber zu erlangen. Es ist weiterhin vorher nicht immer klar, wie viele Kategorien bzw. Cluster gebildet werden und wie die Verteilung der Beispiele aussieht. Durch das Fehlen von Referenzdaten ist eine Überprüfung der Genauigkeit des Verfahrens beim unüberwachten Lernen oft schwierig.

3.1.3 Reinforcement Learning

Der Begriff *Reinforcement Learning* beschreibt ein Verfahren, in dem ein Agent eigenständig lernt, sinnvolle Entscheidungen zu treffen, indem aus einer Menge von Aktionen diejenigen ausgewählt werden, die eine Belohnungsfunktion maximieren [DMB04] [RSS18] [Dam] [MA17a] [Mat15]. Zur

Anwendung kommt es neben Videospielen ebenfalls in der Robotik oder in der Medizin. Der grundsätzliche Ablauf von RiL wird in Abbildung 3.1 dargestellt.

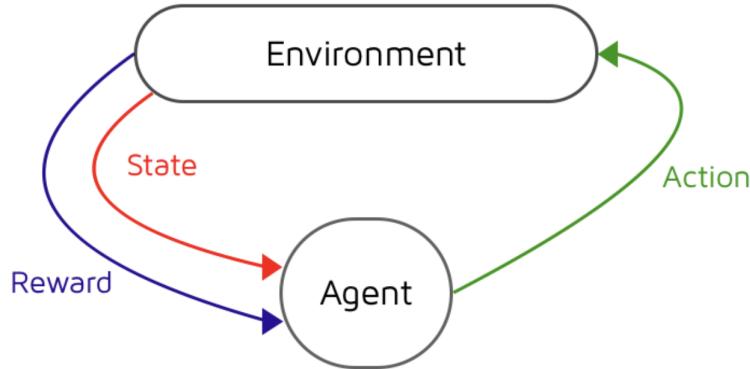


Abbildung 3.1.: Grundsätzlicher Ablauf von RiL [CR16]

Die Ausführung einer Aktion versetzen den Agenten in einen neuen Zustand und resultiert in einer Belohnung (*Reward*) oder einer Bestrafung (*Punishment*). Es handelt sich um einen skalaren Wert, der beschreibt, wie gut der Agent sich verhalten hat. Der Zustand des Agenten wird durch Beobachtungen (*Observations*) aus seiner Umgebung (*Environment*) beschrieben. Der Agent soll aus seinen Erfahrungen lernen, indem er die besten Aktionen wählt, die die zukünftige Summe seiner Belohnungen maximieren. Der Lernvorgang erfolgt dabei durch die kontinuierliche Änderung seiner Entscheidungsstrategie in Abhängigkeit von seinem Zustand, indem beispielsweise Gewichtungen in einem zugrunde liegenden neuronalen Netz angepasst werden. Dabei wird der Versuch unternommen, die Belohnungsfunktion zu maximieren, indem die Beziehungen zwischen Beobachtungen und gewählten Aktionen, sowie den jeweils resultierenden Belohnungen modelliert werden. Für die Suche der besten Entscheidung werden zusätzlich zukünftige Belohnungen berechnet und mit einbezogen (siehe *Markov Decision Problems* in Abschnitt 3.2). Generell muss nicht explizit ein klares Ziel definiert werden, stattdessen führt das eigenständige Lernen dazu, dass sich das optimale Verhalten eines Agenten automatisiert ergibt.

Eine Kategorisierung von RiL in *überwachtes Lernen* oder unüberwachtes Lernen (siehe Abschnitte 3.1.1 und 3.1.2) ist schwierig. Im Gegensatz zum unüberwachten Lernen können den möglichen Aktionen konkrete Werte zugewiesen werden, die das Verhalten des Agenten bewerten. Anstelle des Aufspüren von Anomalien oder Clustern wird also versucht, dem Verhalten des Agenten einen Wert bzw. ein *Label* zuzuweisen. Im Unterschied zum überwachten Lernen werden hier allerdings keine gelabelten Trainingsdatensätze verwendet, sondern eine Menge von Belohnungswerten die sich erst zur Zeit des Trainingsvorgangs ergeben. Bei der Entscheidungsfindung sind beim Reinforcement Learning zudem nicht nur einzelne Vorhersagen relevant, sondern ganze Sequenzen. Man kann also weder von überwachtem, noch von unüberwachtem Lernen sprechen. RiL wird daher als eine zusätzliche Kategorie betrachtet, die sich von den genannten klassischen Verfahren abgrenzt und eine Art Hybrid-Verfahren darstellt.

Für die Entwicklung intelligenter Agenten ist unüberwachtes Lernen im klassischen Sinne nicht geeignet, da dort mit der Extrahierung markanter Features und Zusammenhänge in Datensätzen ein gänzlich anderes Ziel verfolgt wird. Wegen fehlender gelabelter Datensätze eignet sich auch überwachtes Lernen nicht. Man stelle sich ein Szenario vor, in dem ein Agent das Schachspielen lernen soll: Für eine große Menge von denkbaren Situationen würden dann Daten mit dem jeweiligen Label für den korrekten auszuwählenden Zug benötigt. Diese große Datenmenge müsste mühselig angefertigt bzw. gesammelt werden. Im deterministischen Falle von Schach wäre das Sammeln dieser Daten beispielsweise durch die Aufzeichnung von Spielen in einer Schachapp möglich. Diese

Möglichkeit ist jedoch nicht mehr gegeben, sobald das Zielverhalten des Agenten unbekannt ist oder die Umgebung nicht-deterministische Strukturen aufweist. RiL bietet hingegen ein leicht zu handhabendes Framework, in dem Probleme auf eine simple Art und Weise beschrieben werden können. In der Theorie müssen hier lediglich gute Züge positiv und schlechte Züge negativ bewertet werden, sodass der Agent aus den Belohnungen in Kombination mit seinen Beobachtungen Schlüsse ziehen kann. Weiterhin fällt die Notwendigkeit von großen (gelabelten) Datensätzen weg. Aktuelle Projekte (siehe Kapitel 4.3) zeigen dabei vielversprechende Ergebnisse und beweisen sowohl die Skalierbarkeit als auch das hohe Lernpotential von RiL-basierten Algorithmen. Mit Projekten wie *OpenAI Gym* [Ope16] und *ML-Agents* [AJ18] (siehe Abschnitt 5) stehen darüber hinaus umfangreiche Umgebungen und Bibliotheken für die Einbindung in Spiele bereit. Weiterhin finden sich keine vergleichbaren Ansätze in der Fachliteratur, die auf RiL zur Definition des Verhaltens von NPCs setzen, wodurch ein Gegenpol zu klassischen Verfahren gesetzt und eine neuartige Methode getestet werden kann.

RiL bietet aus den genannten Gründen den sinnvollsten und möglichweise einzigen realistischen Lösungsansatz für das gebotene Problem und wird daher in der vorliegenden Arbeit verwendet.

3.2 Markov Decision Problems

Dieses Kapitel beschäftigt sich mit der Erläuterung von MDPs als Grundlage für das *Reinforcement Learning*, welches die Lösung von *Markov-Problemen* anstrebt. Daher bilden sie eine wichtige Basis für das folgende Kapitel. Bei Markov-Problemen (bzw. *Markov-Entscheidungsproblemen*) handelt es sich um ein Modell für Entscheidungsprobleme, bei denen ein Agent versucht, eine Belohnungsfunktion zu maximieren [DMB04] [RSS18] [Dee16] [SR03] [Dam] [MA17b] [Ash18]. Zu jeder Entscheidung befindet sich der Agent in einem bestimmten Zustand und verfügt über eine Menge von Aktionen, aus denen er wählen kann. Durch die Ausführung von Aktionen bewirkt der Agent Übergänge in andere Zustände. Es muss einen oder mehrere Endzustände geben, die die Ausführung abbrechen, sobald sie erreicht werden. Die Wahrscheinlichkeit für die Auswahl einer Aktion liegt jeweils bei einem Wert von $p < 1$. Wenn ein Agent stets die beste Entscheidung trifft, dann spricht man von einer optimalen *Policy* für die Lösung des Problems. Eine Policy ist also eine Strategie des Agenten in Abhängigkeit von seinem Zustand, welcher durch Beobachtungen aus seiner Umgebung festgelegt wird. Diese werden auch *Observations* genannt. Die Policy kann weiterhin definiert werden als eine Funktion vom Zustand eines Agenten zu einer Wahrscheinlichkeitsverteilung über die möglichen Aktionen [CB19].

3.2.1 Beispiel

Im folgenden Beispiel wird ein Roboter betrachtet [CS03], der sich jeweils um ein Feld in vier Richtungen bewegen kann. Wenn dieser die Wände des in Abbildung 3.2 sichtbaren Spielfelds berührt, dreht er automatisch um und bewegt sich in die gegenüberliegende Richtung. Nur die weißen Felder sind passierbar. Felder, die mit einer Zahl versehen sind, geben Belohnungen oder Bestrafungen. Hier wird ein skalarer Wert +1 oder -1 genutzt. Das Feld mit einer Bestrafung heißt *Bestrafungsfeld*. Weitere Felder resultieren jeweils in einer Bestrafung von -0.04. Das Ziel ist es, den besten Weg zum Ziel (dem Feld mit der +1) mit der höchsten Belohnung für den Roboter zu finden.

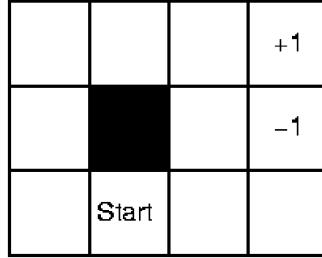


Abbildung 3.2.: Deterministisches MDP [CS03]

Betrachtet wird nun ein deterministischer Ansatz. Hier gehen wir davon aus, dass der Roboter die Aktionen, die er auswählt, immer korrekt ausführt. Als bester Weg ergibt sich dabei mathematisch der kürzeste Weg, wie in Abbildung 3.3 zu sehen. Da das Szenario leicht in einen Graphen mit Kosten transformiert werden kann, ist offensichtlich, dass Suchverfahren wie A* verwendet werden können, um den optimalen Pfad zu finden.

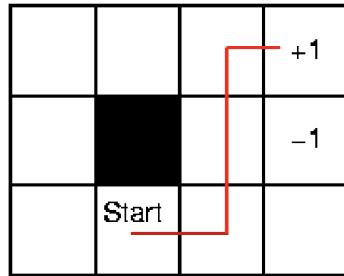


Abbildung 3.3.: Lösung des deterministischen MDP [CS03]

Nun wird ein nichtdeterministischer bzw. stochastischer Ansatz gewählt. Bei solchen Problemen spricht man von einem *Partially Observable Markov Decision Process* (POMDP). Die Zustandsübergänge bekommen dann einen stochastischen Wert zugewiesen. Das bedeutet, dass der Roboter Aktionen nun mit einer gewissen Wahrscheinlichkeit korrekt oder inkorrekt ausführen kann. Die Umgebung des Agenten ist also nichtdeterministisch.

Die Wahrscheinlichkeiten für das Roboter-Beispiel sind in Abbildung 3.4 erkennbar. Der Agent hat in seiner Laufrichtung eine Chance von $p = 0,8$ und zu den Seiten hin jeweils eine Wahrscheinlichkeit von $p = 0,1$. Das bedeutet, dass der Agent nur in 80% der Fälle die gewünschte Aktion und in 20% der Fälle die falsche Aktionen ausführt.

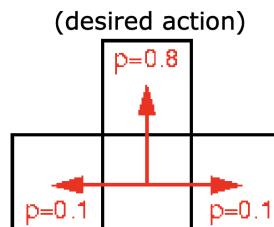


Abbildung 3.4.: Nichtdeterministisches MDP [CS03]

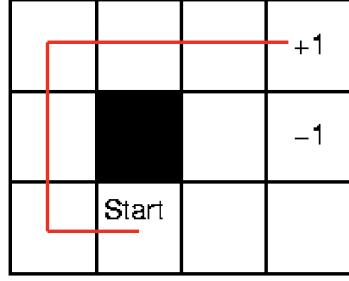


Abbildung 3.5.: Lösung des nichtdeterministischen MDP [CS03]

Abbildung 3.5 zeigt den resultierenden besten Weg. Anstatt den kürzesten Weg zu wählen wird nun ein Weg gewählt, bei dem die Wahrscheinlichkeit geringer ist, durch eine falsch ausgeführte Aktion versehentlich das Bestrafungsfeld zu betreten. Statistisch gesehen ergibt sich daraus die höchste Belohnung für den Agenten. In diesem Fall kann das Problem also nicht ohne Weiteres durch einen Suchalgorithmus gelöst werden, da es nichtdeterministisch ist.

3.2.2 Formale Beschreibung

Nachdem im vorherigen Kapitel zum Verständnis je ein Beispiel für ein deterministisches MDP und ein nichtdeterministisches MDP geliefert wurde, folgt in diesem Abschnitt eine formale Beschreibung von Markov-Problemen.

Sei N ein Agent in einer Umgebung ϵ mit einer Anzahl diskreter Zeitschritte t . Die Umgebung kann beispielsweise durch eine virtuelle Welt in einem Spiel dargestellt werden. N verfügt über endlich viele Zuständen s_t aus der Menge aller möglichen Zustände \mathbb{S} mit einer jeweils dazugehörigen Aktion a_t . Die jeweils nächsten Zustände heißen $s_t + 1$ und $a_t + 1$. Zu jedem Zeitschritt t wählt N durch π eine neue Aktion a_t aus einer endlichen Menge von Aktionen \mathbb{A} . Der Agent N erhält nach a_t einen neuen Zustand $s_t + 1$ und eine Belohnung r_t . Dieser Prozess wiederholt sich, bis ein Terminalzustand s_t erreicht wurde.

N verfügt über eine Policy π . Diese wurde bereits definiert als eine Funktion, die den aktuellen Zustand s_t aus \mathbb{S} nimmt und eine Aktion a_t aus \mathbb{A} zurückgibt. Formal wird π also wie folgt beschrieben:

$$\pi(s_t) : \mathbb{S} \rightarrow \mathbb{A}$$

Aus den einzelnen Belohnungen ergibt sich dann eine akkumulierte Gesamtbelohnung R_t , wobei gilt:

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

Die Summe von 0 bis ∞ wird gebildet, weil davon ausgegangen wird, dass es kein Zeit- bzw. Zeitschrittlimit gibt. In der oben sichtbaren Definition beschreibt γ den sogenannten *Discount-Faktor*. Er wird wie folgt definiert:

$$\gamma \in]0, 1]$$

Nach jeder ausgeführten Aktion werden alle Belohnungen, die der Agent theoretisch noch erreichen kann, mit γ multipliziert. Dies führt dazu, dass weiter in der Zukunft liegende Belohnungen eine geringere Gewichtung bekommen. Eine weit entfernte hohe Belohnung kann durch die Gewichtung des Discount-Faktors eine deutlich kleinere tatsächliche Belohnung darstellen und damit für den Agenten „uninteressant“ werden. Der optimale Lösungsweg wird durch einen Discount-Faktor

$\gamma < 1$ so verändert, dass der Agent im Allgemeinen kürzere Wege bevorzugt. Zu kleine Werte führen dazu, dass der Agent keine langfristigen Belohnungen berücksichtigen kann und kurzsichtige Entscheidungen trifft.

Das Ziel des Agenten ist also die Maximierung der gesamten zukünftigen Belohnung, also der Gesamtbelohnung R_t für jeden Zustand s_t . Die potentielle Belohnung wird durch die gewichtete Summe der erwarteten Belohnungswerte für alle zukünftigen Aktionen gebildet, wobei alle möglichen Aktionen in Betracht gezogen werden. Dadurch wird effektiv die aktuell auszuwählende Aktion durch zukünftige Belohnungen beeinflusst.

Es wurde ein Beispiel für Markov-Probleme demonstriert und eine formale Beschreibung aufgestellt. Der nächste Schritt besteht nun in der Lösung von Markov-Problemen. Als weitere Voraussetzung wird im Folgenden kurz die *Markov-Eigenschaft* erläutert.

3.2.3 Markov-Eigenschaft

Es wird ein Szenario betrachtet, in dem ein Agent die Zustände A , B , C und D annehmen kann [Lem00]. Die Wahrscheinlichkeit für das sequentielle Auftreten der genannten Zustände hintereinander kann wie folgt faktorisiert werden:

$$\begin{aligned} p(A, B, C, D) &= p(A)p(B|A)p(C|A, B)p(D|A, B, C) \\ &= p(B)p(A|B)p(C|A, B)p(D|A, B, C) \dots \\ &= p(C)p(A|C)p(B|A, C)p(D|A, B, C) \dots \\ &\quad \dots \end{aligned}$$

Die *Markov-Eigenschaft* gilt, wenn es eine ausgezeichnete (z.B. zeitliche) Anordnung gibt, für die sich die Darstellung wie folgt vereinfachen lässt [Lem00]:

$$p(A, B, C, D) = p(A)p(B|A)p(C|B)p(D|C)$$

Das bedeutet, dass die Wahrscheinlichkeit für den nächsten Zustand nur unmittelbar von dem vorherigen Zustand abhängt. Die Wahrscheinlichkeit für C hängt also nur vom vorherigen B und nicht von dem ursprünglichen A ab.

3.3 Modellfreie und modellbasierte Verfahren

Konkret werden beim Reinforcement Learning Markov-Entscheidungsprobleme gelöst. Es gibt also, wie in Kapitel 3.2 definiert, einen Agenten mit einer Menge von Zuständen, einer Menge von Aktionen und einer Policy, die optimiert werden soll, indem Belohnungen von Aktionen in Abhängigkeit von Beobachtungen bzw. *Observations* in Betracht gezogen werden.

Man unterscheidet zwischen *modellbasierten* und *modellfreien* Ansätzen (*model-based* und *model-free*) [Pon18]. Dabei geht es nicht um die Verwendung eines Modells im Sinne von einer Belegung/Gewichtung der Neuronen eines neuronalen Netzes. Dieser Begriff wird häufig beim überwachten Lernen verwendet, beschreibt aber nicht die selbe Sache.

Beim modellbasierten Lernen verfügt der Agent über Informationen zu seinen möglichen Zuständen und Aktionen. Ihm sind sowohl alle möglichen Zustandsübergänge als auch die nötigen Aktionen bekannt, die diese auslösen. Er verfügt also über ein Modell, das seine Umgebung repräsentiert. Dies gibt ihm die Möglichkeit, vorauszuplanen und zukünftige Belohnungen direkt zu berechnen.

Man spricht vom modellfreien Lernen, wenn ein Agent lediglich aus gesammelten Erfahrungen Rückschlüsse über seine Zustände und Aktionen treffen kann. Für das Treffen von Entscheidungen verlassen sich derartige Verfahren auf vergangene Beispiele aus ihrer Umgebung, ohne direkte Vorhersagen über den nächsten Zustand zu treffen. Der Agent verwendet also eine Art „Trial-And-Error“-Verfahren für die Berechnung einer möglichst guten Policy.

3.3.1 Modellbasierte Verfahren

Im Folgenden werden einige relevante modellbasierte Verfahren konkret vorgestellt. Als erstes werden dafür weitere nötige Grundlagen geschaffen, indem zugrunde liegende Sachverhalte wie die *Bellman-Gleichung* erläutert werden.

3.3.1.1 Grundlagen modellbasierter Verfahren

Das im Kapitel 3.2 vorgestellte Beispiel eignet sich gut für modellbasierte Verfahren. Der Agent kennt das MDP-Modell der Welt mit Zustandsübergängen, Aktionen und Wahrscheinlichkeitsverteilungen.

Zunächst wird eine sogenannte *Value Function* aufgestellt, die beschreibt, wie gut ein Zustand für einen Agenten ist. Sie entspricht der erwarteten Belohnung ausgehend von einem Startzustand. Die Funktion hängt von der Policy π des Agenten ab [MA17b] [Alp10].

$$V^\pi(s) = \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k * r_{t+k} \right] \quad \forall s \in \mathbb{S}$$

Das Symbol \mathbb{E} steht dabei für einen Erwartungswert. Die Value Function für einen bestimmten Zustand und eine bestimmte Policy berechnet also für alle Zustände s aus \mathbb{S} die erwartete Gesamtbelohnung, welche sich aus den einzelnen Belohnungen der einzelnen Zustände bzw. Aktionen auf dem Weg bis zum Zielzustand befindet.

Es existiert eine Optimale Value Function V^* , die von allen Funktionen den höchsten Reward liefert.

$$V^*(s) = \max_{\pi} V^\pi(s) \quad \forall s \in \mathbb{S}$$

3. Reinforcement Learning

Weiterhin existiert eine optimale Policy π^* , welche in der optimale Value Function resultiert.

$$\pi^* = \arg \max_{\pi} V^\pi(s) \quad \forall s \in \mathbb{S}$$

Weiterhin wichtig für sowohl modellbasierte als auch modellfreie Verfahren sind sogenannte *Q-Functions*. Diese nutzen einen Zustand s und eine Aktion a . Dabei wird ein Mapping von Zuständen zu möglichen Aktionen verwendet. Eine Q-Function gibt einen reellen Wert zurück, der die zu erwartende Belohnung widerspiegelt, wenn im Zustand s die Aktion a ausgeführt wird.

$$Q : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R}$$

Die optimale Q-Funktion lautet $Q^*(s, a)$. Ein Agent startet dann in Zustand s und führt bis zum Terminalzustand nur optimale Aktionen aus. Sie gibt Angaben darüber, wie gut ein Zustand für einen Agenten ist, wie die optimale Value Function V^* . Da V^* die maximal zu erwartende Belohnung beim Start in Zustand s ist, so wäre das Q-Äquivalent das Maximum von $Q^*(s, a)$ über alle möglichen Aktionen. Also gilt:

$$V^*(s) = \max_a Q^*(s, a) \quad \forall s \in \mathbb{S}$$

$$\pi^* = \arg \max_a Q^*(s, a) \quad \forall s \in \mathbb{S}$$

Bellman-Gleichung Die beiden modellbehafteten Verfahren *Value Iteration und Policy Iteration* und weitere Ansätze nutzen die *Bellman-Gleichung* [Bel57]. Die Gleichung stellt ein Optimierungsverfahren dar. Die Q-Funktion $Q^*(s, a)$ entspricht laut Bellman der Summe der unmittelbaren Belohnungen der Aktion a aus Zustand s heraus und der mit dem Discountfaktor verrechneten zukünftigen Belohnung nach der Zustandsänderung in einen neuen Zustand s' [MA17b] [Alp10] [Jul17] [MA17b].

$$Q^*(s, a) = R(s, a) + \gamma \mathbb{E}_{s'} [V^*(s')]$$

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathbb{S}} p'(s|s, a) V^*(s')$$

$$V^*(s) = \max_a \left[R(s, a) + \gamma \sum_{s' \in \mathbb{S}} p'(s|s, a) V^*(s') \right]$$

3.3.1.2 Value Iteration

Value Iteration nutzt die Bellman-Gleichung zur iterativen Berechnung der optimalen Value Function $V(s)$. Es garantiert immer nach n Schritten eine Konvergenz zur optimalen Value Function. Der Ablauf von Value Iteration wird anhand des folgenden Pseudocodes verdeutlicht [Alp10]:

Algorithm 1: Value Iteration

Result: Optimal Value Function

Initialize $V(s)$ to arbitrary values;

while $V(s)$ has not converged **do**

```

for all  $s \in \mathbb{S}$  do
    for all  $a \in \mathbb{A}$  do
        |  $Q(s, a) \leftarrow \mathbb{E}[r|s, a] + \gamma \sum_{s' \in \mathbb{S}} p'(s|s, a) V^*(s')$ 
    end
     $V(s) \leftarrow \max_a Q(s, a)$ 
end

```

end

Bei diesem Vorgehen wird eine relativ willkürliche Endbedingung festgelegt. Es sollte ein sinnvoller Wert für die erreichte Konvergenz der Value Function als Abbruchbedingung gefunden werden.

Die Funktion beinhaltet einen rekursiven Abstieg. $V(s)$ wird so lange mit dem jeweils nächsten Zustand aufgerufen, bis das Ende erreicht wird, wobei für jeden Zwischenzustand alle möglichen Aktionen ausgewertet und jeweils die mit der maximalen Belohnung R gewählt wird. Dies wird für jeden möglichen Startzustand aus \mathbb{S} und jede mögliche Aktion wiederholt.

3.3.1.3 Policy Iteration

Das Vorgehen bei *Policy Iteration* verfolgt den Ansatz, statt der optimalen Value Function die optimale Policy zu finden. Bei jedem Schritt wird hier also die Policy angepasst, bis diese konvergiert. Ebenso wie bei Value Iteration wird hier eine Konvergenz für ein optimales Ergebnis nach n Schritten garantiert. Im folgenden Pseudocode wird die Funktionsweise der Policy Iteration gezeigt [Alp10]:

Algorithm 2: Policy Iteration

Result: Optimal Policy

Initialize a policy π arbitrarily;

while $\pi \neq \pi'$ **do**

```

 $\pi \leftarrow \pi'$  ;
Compute Value Functions for  $\pi$  by solving linear equations;
 $V^\pi(s) = \mathbb{E}[r|s, \pi(s)] + \gamma \sum_{s' \in \mathbb{S}} P(s'|s, \pi(s))V^\pi(s')$  ;
Improve the policy at each state;
 $\pi'(s) \leftarrow \arg \max_a (\mathbb{E}[r|s, a] + \gamma \sum_{s' \in \mathbb{S}} P(s'|s, a) V^\pi(s'))$  ;

```

end

Das Verfahren läuft so lange, wie die jeweils neu berechnete Policy eine Änderung erfährt. Sobald π' gleich π gilt, terminiert der Algorithmus. In jedem Schritt werden zunächst die Value Functions für die aktuelle Policy π berechnet. Diese bestehen jeweils aus dem erwarteten Reward, der sich aus dem aktuellen State mit der aktuellen Policy ergibt sowie der mit γ verrechneten Summe aller nächsten Zustände s' je nach der durch die Policy bestimmten Wahrscheinlichkeit. Dabei gibt es erneut einen rekursiven Abstieg durch das Aufrufen von $V^\pi(s')$ mit allen weiteren Zuständen s' bis hin zum Erreichen des Endzustands.

Nach der Berechnung aller Value Functions für die aktuelle Policy π wird die neue Policy π' berechnet. Für jeden möglichen Zustand s werden mittels der berechneten Value Functions alle möglichen Wege zum Ziel berechnet und jeweils der mit der höchsten Belohnung gewählt.

3.3.2 Modellfreie Verfahren

In diesem Abschnitt werden modellfreie Verfahren als Gegensatz zu modellbasierten Verfahren behandelt. Es werden einige der konkrete Beispiele genannt. Dabei sollen die jeweils erfolgreichsten Verfahren hinsichtlich ihrer Funktionsweise und mathematischen Abläufen betrachtet werden. Ein besonderer Fokus wird dabei auf Gradientenverfahren gesetzt. Im Folgenden werden konkrete modellfreie Verfahren vorgestellt.

3.3.2.1 Q-Learning

Value Iteration und Policy Iteration garantieren stets ein Ergebnis mit einer optimalen Lösung für ein Problem. Dies ist nur möglich, weil es sich um modellbehaftete Vorgehen handelt, bei denen alle Zustandsübergänge und deren Effekte bekannt sind.

Beim *Q-Learning* ist dies nicht der Fall. Q-Learning gehört zu der Kategorie des *Temporal Difference Learning*. Beim Temporal Difference Learning passt der Agent nach jeder ausgeführten Aktion seine Policy an, indem die Erwartungen für jede mögliche Aktion für den aktuellen Zustand des Agenten geschätzt werden. Der Unterschied zu Policy Iteration ist, dass es sich um ein modellfreies Verfahren handelt. Dabei werden beim Lernvorgang keine vorausschauenden Aktionen getätig - der Agent lernt lediglich aus seinen eigenen Erfahrungen. Anstelle der mathematischen Erkundung aller möglichen Wege bis zum Zielzustand werden also viele Samples aus der Erfahrung des Agenten gemacht, um statistisch festzuhalten, welche Aktion in welchem Zustand die sinnvollste ist.

3.3.2.2 Q-Tables

Beim Q-Learning werden Tupel aus Zuständen, Aktionen und Belohnungen gebildet. Diese Tupel füllen dann eine Lookup-Table aus Wertepaaren. Jede Kombination aus für den Agenten nötigen Beobachtungen n ergeben dabei einen neuen State. Die gefüllte Tabelle enthält pro Zustand Werte für jede mögliche Aktion, die Auskunft darüber geben, wie gut die Aktion mit diesem Zustand ist. Typischerweise wird diese *Q-Table* mit Zufallswerten gefüllt. Unter Anwendung des Temporal Difference Learnings werden die optimalen Zustand-Aktion-Tupel approximiert. Wenn in einem Zustand s eine Aktion a mit einer registrierten Belohnung $r(s, a)$ ausgeführt wird, so funktioniert die Berechnung des neuen Tupels $Q(s, a)$ wie folgt:

$$Q(s, a) = (1 - \alpha) Q(s, a) + \alpha Q_{obs}(s, a)$$

$$Q_{obs}(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$$

Bei α handelt es sich um die Lernrate. Der neue Wert von $Q(s, a)$ ergibt sich aus der Addition des alten Werts mit dem neuen observierten Wert Q_{obs} , jeweils verrechnet mit α . Q_{obs} wird aus dem registrierten Belohnungswert und dem maximalen Q-Wert für den neuen Zustand s' errechnet.

Abhängig von der Anzahl der relevanten Variablen und damit den Zuständen ergeben sich daraus unter Umständen sehr große Lookup-Tabellen. Man stelle sich ein Spielfeld von vier mal vier Feldern vor. Jedes Feld kann entweder ein Hindernis, ein Ziel oder nichts beinhalten. Jedes der insgesamt 16 Felder kann also drei verschiedene Feldzustände annehmen. Selbst aus diesem minimalistischen Szenario ergeben sich bereits $n_s = 3^{16} \approx 43$ Millionen maximal mögliche Zustände. Diese Zahl müsste zusätzlich noch mit der Anzahl der Aktionen multipliziert werden, um die Anzahl der Q-Werte zu erhalten. Beim Q-Learning muss der Agent in der Lage sein, über möglichst alle möglichen Zustände zu abstrahieren, indem er aus seinen gesammelten Erfahrungen lernen kann, um die Q-Tabelle

sinnvoll zu füllen. Das setzt auch voraus, dass genügend Erfahrungen gesammelt werden. Ab einer zu hohen Menge von Zuständen ist dies nicht mehr garantiert. Bei der Definition der Zustände und ihrer Granularität und bei der Entscheidung, was überhaupt als ein Zustand bewertet wird, sollte dies berücksichtigt werden.

Q-Learning mit neuronalen Netzen Aufgrund der schlechten Skalierbarkeit von Q-Tables muss ein alternativer Weg für die Beschreibung der Zustände gefunden werden. Neuronale Netze können dabei als Approximator dienen, um mit einer hohen Anzahl von Zuständen zu arbeiten. Dabei kann ein Vektor mit einem *One-Hot-Encoding* als Input für das neuronale Netz verwendet werden. Die Größe des Vektors hängt dabei von der Anzahl der für die Definition der Zustände relevanten Variablen ab. Jede Variable im Vektor wird entweder mit einer 0 oder einer 1 markiert, um die Belegung der Variablen zu definieren. Beim oben genannten Beispiel ergibt sich dann eine Vektorlänge von $n_{vec} = 16 * 3 = 48$. Der Output des neuronalen Netzes liefert dann jeweils einen Wert pro ausführbarer Aktion. Der Lerneffekt ergibt sich dann durch die Anpassung der Gewichte des neuronalen Netzes. Statt der Aktualisierung der Q-Table wird dann eine Loss-Funktion sowie Verfahren wie *Backpropagation* [HN89] angewendet.

3.3.2.3 Policy Gradient

Eine wichtige Oberklasse von Methoden für die Anwendung von Reinforcement Learning basiert auf *Policy Gradient*-Algorithmen. Dabei wird ein Gradientenaufstieg verwendet. Allgemein werden Gradientenverfahren in der Numerik genutzt, um Optimierungsprobleme zu lösen. Hier handelt es sich um ein Optimierungsproblem bezüglich der erwarteten Belohnung einer Policy. Gradientenbasierte Verfahren werden in POMDP-Umgebungen verwendet.

Alle bisher vorgestellten Algorithmen basieren auf *Action-Value*-Methoden (Value Functions) [RSS18]. Die Policies in diesen Verfahren lernen direkt die Belohnungswerte von Aktionen und beinhalten Schätzungen dieser Belohnungen basierend auf der Aktion und dem Zustand eines Agenten. Statt dessen werden nun parametrisierte Policies verwendet. Um die Parameter einer Policy zu erlernen, können hier Value Functions verwendet werden, für die Auswahl von Aktionen sind diese allerdings nicht notwendig.

Grundlagen Es wird davon ausgegangen, dass eine Policy parametrisiert ist. Die Menge dieser Parameter, die die Beschaffenheit der Policy ausmachen, heißt θ . Das Ziel von Reinforcement Learning ist stets die Anpassung dieser Parameter der Policy, sodass die erwartete Gesamtbelohnung J einer Policy π maximal ist. Man spricht auch von *Return of Policy*. Die Parameter dieser Policy sollen mithilfe eines Gradientenverfahren gelernt werden, sodass die gesamte zu erwartenden Belohnung maximiert wird [RSS18] [Pog19]. Bei den Parametern handelt es sich üblicherweise um die Gewichtungen der einzelnen Neuronen in einem neuronalen Netz.

$$\theta^* = \arg \max_{\theta} J(\theta)$$

Dies gilt universell für alle Methoden des Reinforcement Learnings. Die Parameter θ könnten sich beispielsweise auch auf eine Value Function oder auf Q-Functions beziehen, die optimiert werden sollen. Im Folgenden wird der Fokus auf das Policy Gradient-Verfahren gelegt.

Die dem Verfahren zugrunde liegenden Markov-Entscheidungsprobleme sind stets von stochastischer Natur (siehe Kapitel 3.2). Es ist nicht exakt vorhersehbar, in welchem Zustand der Agent sich

im nächsten Schritt befinden wird. Es ergeben sich Wahrscheinlichkeiten für die Übergänge von Zuständen:

$$p(s_{t+1}|s_t, a_t)$$

Durch die wiederholte Ausführung von Aktionen und resultierenden Zustandsübergängen folgt schließlich das Erreichen eines Endzustandes. Jeder Weg vom Startzustand s_0 zum Endzustand s_n , den der Agent beschritten hat, wird als *Trajectory* τ beschrieben und beinhaltet die einzelnen Zustände, Aktionen und Zustandsübergänge. Ein Trajectory τ hat einen Belohnungswert R , oder *Return of Trajectory*, der sich aus der Summe der jeweiligen Belohnungen r für die einzelnen Aktionen von τ ergibt:

$$R(\tau) = \sum_{i=0}^N r(s_i, a_i)$$

R hängt weiterhin vom bereits definierten Discount-Faktor γ ab. Im Folgenden wird dieser Faktor der Einfachheit halber ignoriert.

Weil es sich um eine stochastische Umgebung handelt, sind Trajectories nicht unbedingt vorher festgelegt. Aus diesem Grund ist je nach Problem unter Umständen eine unendlich hohe Anzahl von Trajectories möglich. τ wird daher zu einer zufälligen Variablen.

Die Gesamtbelohnung $J(\theta)$ in Abhängigkeit der Parametrisierung von π ergibt sich aus der Summe der gewichteten Einzelbelohnungen R von τ . Sie beschreibt, wie gut eine Policy ist. Für die Gewichtung wird jede registrierte Belohnung mit dem Erwartungswert für die Aktion a_i multipliziert. Der Erwartungswert ist die Wahrscheinlichkeiten $p(\tau|\theta)$ für das Vorkommen von τ in Abhängigkeit von θ . Da τ eine Zufallsvariable darstellt, wird ein Integral gebildet. J hängt also von den jeweiligen Belohnungswerten R aller τ ab:

$$J(\theta) = \int p(\tau|\theta) R(\tau) d\tau$$

Die Wahrscheinlichkeit $p(\tau)$ für die Wahl eines Trajectory τ mit einer Parametrisierung θ ist die Wahrscheinlichkeit für den Startzustand s_0 multipliziert mit dem Produkt der einzelnen Wahrscheinlichkeiten aller Zustandsübergänge in τ . Wegen der geltenden Markov-Eigenschaft (siehe Abschnitt 3.2) in diesem stochastischen Prozess ist diese Formulierung gültig:

$$p(\tau|\theta) = p(s_0) * \prod_{i=0}^N p(s_{i+1}|s_i, a_i) \pi(a_i|s_i; \theta)$$

Berechnung des Gradienten Ein Gradient zeigt bei einem vorhandenen Gefälle stets in die Richtung des größten Anstiegs. Klassischerweise handelt es sich bei einem Gradienten um einen Vektor. Man stelle sich ein skalares Feld mit einer ortsabhängigen Temperaturverteilung vor. Der Gradient zeigt dann in jedem Punkt des Feldes in die Richtung des höchsten lokalen Temperaturanstiegs. Im Falle von Policy Gradient wird ein Gradientenverfahren verwendet, um sich entlang von Steigungen bezüglich der Gütfunktion $J(\theta)$ einer Policy π zu bewegen und so ein Maximum zu finden [Pog19] [RSS18].

3. Reinforcement Learning

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \int p(\tau|\theta) R(\tau) d\tau$$

Beziehungsweise:

$$\nabla_{\theta} J(\theta) = \int \nabla_{\theta} p(\tau|\theta) R(\tau) d\tau$$

J hängt damit indirekt von θ und direkt von den jeweiligen Trajectories ab. Durch Einsetzen der Wahrscheinlichkeiten erhält man:

$$\nabla_{\theta} J(\theta) = \int \nabla_{\theta} p(s_0) * \prod_{i=0}^N p(s_{i+1}|s_i, a_i) \pi(a_i|s_i; \theta) R(\tau) d\tau$$

Es werden weitere mathematische Umformungen vorgenommen, um den Term zu reduzieren.

$$\nabla_{\theta} J(\theta) = \int p(\tau|\theta) \nabla_{\theta} \log p(\tau|\theta) R(\tau) d\tau$$

$$\nabla_{\theta} J(\theta) = \int p(\tau|\theta) \nabla_{\theta} \log \left[p(s_0) \prod_{i=0}^N \pi(a_i|s_i; \theta) \right] R(\tau) d\tau$$

$$\nabla_{\theta} J(\theta) = \int p(\tau|\theta) \sum_{i=0}^N \nabla_{\theta} \log \pi(a_i|s_i; \theta) R(\tau) d\tau$$

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi} \left[\sum_{i=0}^N \nabla_{\theta} \log \pi(a_i|s_i; \theta) R(\tau) \right]$$

Bei dem Ergebnis handelt es sich um einen Erwartungswert über Trajectories für die Policy π . Aufgrund der möglicherweise enormen Anzahl der Trajectories kann diese Funktion nicht direkt berechnet, aber beispielsweise durch Monte Carlo Sampling approximiert werden.

Aus der Policy Gradient-Funktion kann weiterhin die Updatefunktion für die Parameter θ abgeleitet werden. Die Änderung der Parameter entspricht der Lernrate α multipliziert mit dem Gradientenwert:

$$\Delta\theta = \alpha * \nabla_{\theta} \log \pi(a_i|s_i; \theta) R(\tau)$$

Diese Funktion bestimmt, wie die Parameter während des Gradientenaufstiegs angepasst werden. Die Güte der Policy J wird dann durch die graduelle Anpassung der Wahrscheinlichkeiten der Aktionen für τ angepasst. Wenn R ein hohes Ergebnis erzielt hat, so wird die Wahrscheinlichkeit für die ausgeführten Aktionen innerhalb von τ erhöht, andernfalls werden sie verringert.

Loss Functions Statt der Nutzung von Value Functions oder Policy Gradient-Funktionen können sogenannte *Loss Functions* oder *Verlustfunktionen* angegeben werden. Konkret beschreibt eine Verlustfunktion den Unterschied zwischen tatsächlichen Parametern und den Parametern, die für eine getroffene Entscheidung des Modells vorhergesagt wurden. Dadurch kann bestimmt werden, wie groß der aus einem Zustandsübergang resultierende Fehler der vorhergesagten Parameter ist, bzw. wie sehr sich ein neuer Zustand von dem vorhergesagten Zustand unterscheidet. Allgemein wird dadurch beschrieben, wie gut die Anpassung einer Value Function bzw. die Anpassung der Parameter θ einer Policy gewesen ist. Bei gegebenen Parametern θ gibt eine Loss Function also den Verlust an, der durch die Wahl der neuen Parameter θ_{t+1} entsteht.

Vorteile und Nachteile Gegenüber anderen Verfahren bieten Gradienverfahren den Vorteil einer schnelleren Konvergenz. Wertebasierte Verfahren wie Q-Learning neigen zu einer höheren Oszillation, weil bereits kleine Änderungen der Einschätzung der Q-Werte drastische Veränderungen für das Verhalten und den Trajectory des Agenten zur Folge haben können. Policy Gradient stellt darüber hinaus sicher, dass jeder Schritt eine Verbesserung der Belohnung der Policy zur Folge hat. In jedem Fall wird dabei mindestens ein lokales, im besten Falle ein globales Maximum gefunden.

Weiterhin wird die Effektivität in hochdimensionalen Aktionsräumen oder unter der Verwendung von *Continuous Actions* erhöht. Für Continuous Actions wird stets ein skalarer Wert geliefert, anstatt dass eine Entscheidung für diese Aktion getroffen werden muss. Bei Verfahren wie Q-Learning wird jeder möglichen Aktion zu jedem Zustand eine Art *Score*, nämlich der zu erwartende zukünftige Reward, zugewiesen. Bei einer hohen Anzahl von Aktionen und Zuständen wird die Anzahl der Q-Werte zu hoch.

Weiterhin kann in Gradientenverfahren eine stochastische Policy gelernt werden, was mit Value Functions nicht möglich ist. Weil eine stochastische Policy eine Wahrscheinlichkeitsverteilung über Aktionen definiert, kann der Agent den Zustandsraum zu erkunden, ohne stets die gleiche Aktion auszuführen, da die Aktionen stochastisch ausgewählt werden. Die Erkundung muss also nicht expliziert definiert werden wie in anderen Verfahren.

Aufgrund dieser Vorteile sind Policy Gradient-Verfahren für große Fortschritte im Bereich des Reinforcement Learnings verantwortlich. Ein großer Nachteil besteht allerdings zum einen darin, dass häufig statt globalen Maxima nur lokale Maxima gefunden werden und zum anderen in der Schwierigkeit der Festlegung der Lernrate [JS17]. Verschiedene Verfahren die auf Policy Gradient basieren versuchen, dieses Problem zu lösen.

3.3.2.4 Actor Critic

Ein Beispielverfahren für *Policy Gradient* ist *Actor Critic* [RSS18] [Sim18]. *Critic* repräsentiert die approximierte Value Function und berechnet die erwartete Belohnung für eine Aktion a_t in einem bestimmten Zustand s_t . Der *Actor* ist die Policy, welche eine Aktion a_t für einen gegebenen Zustand s_t generiert und damit das Verhalten des Agenten bestimmt.

Policy Gradient-Verfahren weisen ein großes Problem hinsichtlich der Bewertung der Güte von Trajectories auf. Der Reward R für die gesamte Trajectory τ wird als Basis für die Bewertung verwendet. Wenn dieser Wert hoch ist, wird τ insgesamt als gut bewertet. Das bedeutet im Umkehrschluss allerdings nicht, dass alle Zustandsübergänge innerhalb von τ gut gewesen sein müssen - tatsächlich könnten einzelne sehr schlechte Aktionen getroffen worden sein.

Mit einer sehr hohen Anzahl von Samples kann diesem Problem entgegengewirkt werden, allerdings auf Kosten der Konvergenzdauer. Mit Actor Critic wird stattdessen ein neues Modell mit einer

3. Reinforcement Learning

verbesserten Bewertung der Trajectories verwendet. Folgende aktualisierte Updatefunktion wird verwendet:

Ursprüngliche Updatefunktion:

$$\Delta\theta = \alpha * \nabla_\theta \log \pi(a_i|s_i; \theta) R(\tau)$$

Neue Updatefunktion:

$$\Delta\theta = \alpha * \nabla_\theta \log \pi(a_i|s_i; \theta) Q(s_t, a_t)$$

Die Policy wird also nicht wie gewohnt zum Ende einer Episode bzw. mit jedem Trajectory τ , sondern mit jedem Schritt aktualisiert, Actor Critic gehört also zum Temporal Difference Learning. In diesem Fall wird nicht mehr die Gesamtbelohnung $R(\tau)$ verwendet. Stattdessen wird ein Modell gefunden, das die Value Function approximiert. Actor Critic verwendet dafür zwei separate neurale Netze. Diese werden zu Beginn zufällig initialisiert. Der Actor führt dann eine Aktion aus, woraufhin der Critic beurteilt, wie gut die Aktion war. Er „kritisiert“ also den Actor, woraufhin der Actor seine Policy anpasst. Anschließend passt auch der Critic seine Gewichtungen an, um zukünftig besseres Feedback geben zu können. Es ergeben sich ein Actor mit einer Policyfunktion ($\pi(s, a, \theta)$) und ein Critic mit Q-Werten ($q(s, a, w)$), die parallel arbeiten. Da nun zwei Netze verwendet werden, müssen auch zwei Optimisierungsverfahren mit unterschiedlichen Updatefunktionen parallel laufen.

Actor Policy Update:

$$\Delta\theta = \alpha * \nabla_\theta \log \pi(a_i|s_i; \theta) q_w(s_t, a_t)$$

Critic Value Function Update:

$$\Delta w = \beta * (R(s_t, a_t) + \gamma q_w(s_{t+1}, a_{t+1}) - q_w(s_t, a_t)) \nabla_w q_w(s_t, a_t)$$

Für den Critic wird hier eine zusätzliche Lernrate β verwendet. Das Update hängt ab von dem Error zwischen dem erwarteten Q-Wert und dem tatsächlich erfahrenen Belohnungswert. Diesen inneren Teil der Gleichung (siehe Term innerhalb der großen Klammern) wird als *TD Error* bezeichnet. Das Ausmaß TD Error sowie die Lernrate bestimmen das Update Value Function mithilfe des Gradientenverfahrens.

Zusammenfassend wird also bei jedem Zeitschritt t der aktuelle State s_t sowohl dem Actor als auch dem Critic übergeben. Die Policy generiert daraus eine Aktion a_t mit einem Reward $R(s_t, a_t)$. Der Critic berechnet damit die aktualisierte Value Function für die generierte Aktion in diesem Zustand und der Actor nutzt diese dann, um seine Policy zu aktualisieren.

A2C Eine Verbesserung von Actor Critic ist *A2C*. Value Functions haben das Problem einer hohen Variabilität. Man kann sie schnell zum Oszillieren bringen, wie in Abschnitt 3.3.2.3 beschrieben. Um diesem Problem entgegenzuwirken, wird für A2C die sogenannte *Advantage Function* $A(s_t, a_t)$ anstelle der Value Function verwendet. Diese Funktion gibt Angaben über die *Verbesserung*, die bei der Wahl der jeweiligen Aktion erzielt werden kann, verglichen mit der durchschnittlichen Belohnung der Aktionen dieses Zustands. In anderen Worten gibt sie Auskunft darüber, wie viel besser die Aktion im Vergleich zu den anderen ist.

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

wobei $V(s)$ die Value Function für den Zustand beschreibt und der durchschnittlichen erwarteten Belohnung für alle Aktionen in diesem Zustand entspricht.

3. Reinforcement Learning

Wenn $A(s_t, a_t) > 0$, dann sollte der Gradient in diese Richtung bewegt werden. Wenn $A(s_t, a_t) < 0$, so sollte der Gradient in die entgegengesetzte Richtung bewegt werden.

Für die Berechnung werden nun jeweils sowohl Q-Werte als auch Value Functions benötigt. Bis jetzt haben wir mit dem Critic die Value Function approximiert, indem Q-Werte aktualisiert wurden. Man könnte nun ein weiteres neuronales Netz mit einzelnen Value Functions einführen, sodass sowohl Q-Values als auch Value Functions zur Verfügung stehen. Das wäre jedoch höchst ineffizient. Stattdessen kann die Beziehung von Q-Values und V-Values aus der Bellman-Gleichung verwendet werden, um $A(s_t, a_t)$ umzuschreiben:

$$A(s_t, a_t) = R(s_{t+1}, a_{t+1}) + \gamma V(s_{t+1}) - V(s_t)$$

3.3.2.5 Proximal Policy Optimization

PPO ist ein Algorithmus aus dem Bereich der Policy Gradient-Verfahren, der auf Actor Critic bzw. A2C basiert. Alternative Verfahren aus dem Bereich sind dafür bekannt, sehr sensibel auf eine Änderung der Lernrate bzw. der *Step Size* zu reagieren, was zu einer deutlichen Verschlechterung des Ergebnisses führen kann. Der Hauptfokus des Verfahrens liegt daher in der Vermeidung von zu großen Policy-Updates [SWD⁺17] [JS17] [Ope18b] [Hui18]. PPO erzielt so gute Ergebnisse, dass die im Reinforcement Learning-Bereich erfolgreiche Firma OpenAI sich dazu entschieden hat, in allen Projekten ausschließlich den PPO-Algorithmus zu nutzen [JS17]. Dabei wird zwischen zwei Varianten von PPO unterschieden: *PPO-Penalty* und *PPO-Clip*.

PPO-Penalty verwendet sogenannte *Trust Regions* und weist daher große Ähnlichkeiten zu dem Verfahren *Trust Region Policy Optimization* (TRPO) [JS15] auf. Das Vorgehen verbessert Policies, indem jeweils der größte mögliche Schritt zur Anpassung unternommen wird, während gleichzeitig bestimmte Einschränkungen für die Unterschiedlichkeit der alten und der neuen Policy eingehalten werden müssen. Für diese Einschränkung wird die *Kullback-Leibler-Divergenz* (KLD) verwendet [KS51] [Joy11] - dabei handelt es sich um ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen. Der Begriff *Trust Region* beschreibt also das Aktualisieren der Policy in sinnvollen Abständen und Regionen. Bei TRPO wird die KLD als harte Beschränkung verwendet. PPO-Penalty bestraft hohe KLD-Werte direkt in der Updatefunktion als negativen Reward.

PPO-Clip beinhaltet keine Beschränkungen wie die Kullback-Leibler-Divergenz der Policies. Stattdessen findet ein weniger aufwendigeres aber effektiveres Verfahren, genannt *Clipping*, statt. Folgende Updatefunktion wird definiert:

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

Dabei wird die Variable L anstelle von $\Delta\theta$ verwendet. Die Definition L wird in der Fachliteratur verwendet und steht für *Local Approximation*, hat aber die selbe Bedeutung, nämlich die Aktualisierung der Policy.

Beim Clipping wird die Veränderung der Policy direkt beschnitten. Der Term $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$ sorgt dafür, dass der Wert $r_t(\theta)$ stets im Wertebereich zwischen den Grenzen $1 - \epsilon$ und $1 + \epsilon$ bleibt. Das Resultat wird mit dem Advantage-Wert multipliziert. Anschließend wird das Minimum aus dem geclippten Advantage-Wert und dem ungeclippten Advantage-Wert gebildet. Zusätzlich kann über den Hyperparameter ϵ bestimmt werden, wie sehr sich die neue Policy von der Alten unterscheiden darf.

4

Anwendung von Machine Learning in Videospielen

Obwohl klassische Machine Learning-Verfahren bereits seit Jahrzehnten und damit schon vor der Popularisierung von Videospielen existierten, gab es bis vor kurzem nur wenige Spiele, die tatsächlich Gebrauch von Machine Learning gemacht haben. Im folgenden Kapitel werden Beispiele für die Anwendung von Machine Learning in Videospielen aufgezeigt um das Potential, das in dem Bereich besteht, hervorzuheben. Die genannten Anwendungsfälle beschränken sich nicht nur auf die Entwicklung intelligenter Agenten, sondern umfassen verschiedene Lösungen für die Erzeugung einer authentischeren Spielwelt.

4.1 Lernende Agenten

Hier kann zwischen zwei Arten unterschieden werden, um das zugrundeliegende Modell, das die Entscheidungen der Agenten bestimmt, zu trainieren. Zum einen gibt es Agenten, deren Modelle zur Laufzeit trainiert werden, also während des Spielens. Zum anderen kann schon zur Entwicklungszeit des Produkts ein Modell trainiert werden, das anschließend nicht mehr verändert wird.

4.1.1 Lernen zur Laufzeit

Lernende Agenten können zur Laufzeit trainiert werden, um die KI so anzupassen, dass sie sinnvoll auf die Spielweise des Spielers reagieren kann. Man könnte sich ein typisches *Stealth Game* vorstellen, in dem der Spieler über verschiedene mögliche Wege in ein Haus eindringen kann und sich dabei möglichst unauffällig verhalten muss. Das Haus wird von Wachen beschützt, die in bestimmten Intervallen um das Haus patrouillieren. Diese Agenten der Wachen könnten nun das durch das Verhalten des Spielers trainiert werden, sodass sie beispielsweise häufige Laufwege oder weitere Spieldynamiken kontern können. In derartigen Spielen verhalten sich NPCs normalerweise statisch, sodass ein Level sozusagen „auswendig gelernt“ werden kann. Sobald eine gute Lösung für ein Level gefunden wurde, kann dieser Vorgang wiederholt werden, um das Level immer auf die gleiche Art und Weise zu lösen. Einige Spiele versuchen dann, das statische Verhalten von NPCs zu überdecken, indem Zufallsvariablen verwendet werden. Wenn die Wachen allerdings zur Laufzeit lernen, den Spieler zu kontern, entsteht ein deutlich dynamischeres System und eine höhere Wiederspielbarkeit, da der selbe Ansatz nicht immer zum selben Ergebnis führt. Der Spieler muss dann kreativ werden und sich neue Lösungsansätze überlegen.

4.1.1.1 Echo

Im Spiel *Echo* [Ult17] hingegen werden tatsächlich lernende Agenten benutzt, um ein ähnliches Szenario zu realisieren. Der Spieler tritt gegen eine Menge von Entitäten an, die das Verhalten

4. Anwendung von Machine Learning in Videospielen

des Spielers imitieren. Es werden sozusagen *Klone* des Spielers erzeugt. Wenn alle Klone besiegt wurden verdunkelt sich nach einer gewissen Zeit die Spielwelt. Dem Spieler ist bewusst, dass dies ein Hinweis darauf ist, dass während des Zeitraums der Verdunklung „verbesserte Versionen“ der Klone hergestellt werden. Tatsächlich wird ein Machine Learning-Verfahren genutzt, um in dieser Zeit Agenten zu trainieren, die das Verhalten des Spielers erlernt haben [JR17]. Je nachdem wie starr die Spielweise des Spielers ist, muss dieser anschließend in gewissem Maße „gegen sich selbst“ kämpfen. Ein möglichst diverser Spielstil wird damit also gefördert.

4.1.1.2 Black & White

Bereits im Jahr 2001 wurde im Spiel *Black & White* [Stu01] eine Kombination aus *Reinforcement Learning* (siehe Abschnitt 3) und *Imitation Learning* (siehe Kapitel 3.1.1.1) verwendet. Dabei handelt es sich um ein Strategiespiel, in dem der Spieler die Rolle eines Gottes einnimmt, der über ein Volk herrscht. Dieser kann sich entscheiden, ein guter oder ein böser Herrscher zu sein. Weiterhin befehligt der Spieler eine Kreatur, die seinen Stellvertreter in der Welt darstellt. Mithilfe eines neuronalen Netzes imitiert diese Kreatur die Aktionen des Gottes und tendiert je nach Spielweise dazu, gut oder böse zu agieren. Weiterhin hat der Spieler die Möglichkeit, das Training direkt zu beeinflussen, indem er sie für bestimmte Aktionen bestraft oder belohnt, was sich auf die Moral der Kreatur auswirkt. Konkret wurde der Output des neuronalen Netzes, das für das Reinforcement Learning verwendet wurde, für die Anpassung eines Decision Trees verwendet, der die weitere Entscheidungsstrategie der Kreatur beeinflusst [Wex01].

4.1.1.3 Vorteile

Die genannten Beispiele zeigen, dass durch das Anlernen von Agenten zur Laufzeit die Möglichkeit geschaffen wird, sehr interessante und dynamische Spielerlebnisse zu erschaffen. Agenten können auf Aktionen des Spielers mit einer Änderung ihres eigenen Verhaltens reagieren, was mit einem vortrainierten Modell nicht möglich ist. Das Trainieren des Modells kostet allerdings zusätzliche Leistung während der Ausführung des Spiels.

4.1.2 Nutzung vortrainierter Modelle

Eine weitere Möglichkeit ist das Trainieren von lernenden Agenten vor der Laufzeit, zum Beispiel schon während der Entwicklungsphase eines Produkts. Das kann dann sinnvoll sein, wenn man dem Agenten objektive „richtige“ und allgemeingültige Entscheidungen in Abhängigkeit seiner Inputparameter beibringen möchte. Das Verhalten des Agenten verändert sich zur Laufzeit nicht mehr. Daher eignet sich dies beispielsweise für die Realisierung von NPC-Gegnern in Strategiespielen, in denen durch das verwendete, erweiterte „Schere-Stein-Papier“-Prinzip Entscheidungen oft offensichtlich sind und nicht an das Verhalten des Spielers angepasst werden müssen.

4.1.2.1 Forza Motorsport

Ein weiteres Anwendungsbeispiel für Agenten, die vom Spieler selbst lernen, ist die bereits erwähnte Rennspielreihe *Forza Motorsport* [Stu05]. Hier wird das Verhalten des Spielers genutzt, um einen Agenten mit Imitation Learning zu trainieren, mit dem sich der Spieler selbst messen kann [Sta14]. Er fährt also Rennen gegen sich selbst, um sich zu verbessern und seine Fehler zu erkennen. Es ist auch möglich, Rennen gegen die „virtuellen Versionen“ anderer Personen, genannt „Drivatar“ von Freunden zu fahren, da die Drivatars in einer Cloud gespeichert werden [DGCFM15].

4.1.2.2 Supreme Commander 2

Im Spiel *Supreme Commander 2* [Gam10] werden gegnerische NPC-Agenten durch ein neuronales Netz gesteuert [CJ12]. Genauer befasst sich die AI mit der Auswahl sinnvoller Aktionen für das Kampfsystem. Es sollte daher betont werden, dass nicht das gesamte NPC-System auf Machine Learning basiert, sondern nur ein Teilaspekt davon, der allerdings für dieses spezifische Spiel essentiell ist. Für die Kampfsysteme der verschiedenen Typen von Einheiten wurde dafür jeweils ein eigenes neuronales Netz trainiert. Es handelt sich um Land-, Luft-, Marine-, und Bombereinheiten. Jedes der neuronalen Netze verfügt über 34 Eingangsneuronen, die mit Inputdaten wie der Geschwindigkeit oder der Lebenspunkte der Einheit gespeist werden. Wenn sich zwei Gruppen gegenerischer Einheiten gegenüberstehen, werden hierfür Differenzen aus den jeweiligen Werten gebildet. Dieser Vorgang wurde in einer Präsentation des Lead Developers Mike Robbins auf der *Game Developers Conference* (GDC) [PLC14] im Jahre 2012 näher erläutert. Es wird eine einzelne *Hidden Layer* verwendet, die 98 Neuronen beinhaltet. Die Outputschicht verfügt über 15 Neuronen, die jeweils eine mögliche Aktion für die Einheit darstellen. Dabei handelt es sich um taktische Entscheidungen, wie beispielsweise „Attack Closest“ oder „Attack Weakest“. Damit die Agenten dazulernen, wird beim Training die sogenannte „Fitness“ der involvierten Einheiten betrachtet. Diese Metrik wird analysiert, indem alle Inputvariablen des Netzes nach der durchgeführten Aktion erneut betrachtet und Differenzen zu den vorherigen Werten gebildet werden. So kann entschieden werden, welcher Agent einen Kampf gewonnen hat. Durch *Backpropagation* oder auch *Fehlerrückführung* werden anschließend Gewichtungen innerhalb des neuronalen Netzes vorgenommen. Laut Robbins werden die Netze anschließend für jeweils eine Stunde trainiert.

Auch im komplexeren inoffiziellen Nachfolgerspiel von Supreme Commander 2, *Planetary Annihilation* [Ent13], wurde ein solches Verfahren verwendet [MR14]. Die Dimensionen dieses Spiels übersteigen die des Vorgängers erheblich. Der Spieler kann auf einer Mehrzahl unabhängiger Planeten gleichzeitig agieren und tausende Einheiten gleichzeitig kommandieren. Damit wird gezeigt, dass Machine Learning-Verfahren eine hohe Skalierbarkeit aufweisen können.

4.1.2.3 Vorteile

Im Vergleich zur Nutzung von Agenten, die erst während des tatsächlichen Spielvorgangs trainiert werden, liegt laut Robbins ein Performancevorteil vor. Durch das Wegfallen dieses Trainingsvorgangs zur Laufzeit können mehr Ressourcen für die Darstellung des Spiels verwendet werden. Abhängig vom Spiel seien auch ausreichend vortrainierte Modelle in der Lage, allgemeingültige und objektiv richtige Entscheidungen unabhängig vom Spieler zu fällen, sodass ein Trainingsvorgang während der Laufzeit nicht zwangsläufig ein besseres Ergebnis erzielle.

4.2 Anwendungsfälle außerhalb der intelligenten Agenten

Dieser Abschnitt befasst sich zusätzlich mit weiteren Anwendungsfällen, die außerhalb des Bereichs der Entwicklung von intelligenten Agenten liegen. Im Allgemeinen wird maschinelles Lernen eingesetzt, um überlegende Technologien und Inhalte zu entwickeln. Häufig dient es aber auch als ein Mittel für das Einsparen von Kosten durch die prozedurale Generierung von Inhalten. Die Nutzung solcher Methoden bietet nicht nur das Potential, verbesserte Spielerlebnisse zu erzeugen, sondern ebenfalls die Möglichkeit einer erheblichen Kostensenkung durch die nicht länger bestehende Notwendigkeit zur Beschäftigung großer Mengen von beispielsweise Level- oder Game Designern.

4. Anwendung von Machine Learning in Videospielen

Prozedurale Generierung Ein klassischer Anwendungsfall für prozedurale Generierung ist die Erzeugung von *Height Maps* und Terrain, etwa durch Rauschfunktionen wie *Perlin Noise* [MAP11] [SH17]. Ein bekanntes Beispiel ist das erfolgreiche Spiel Minecraft [AB09]. Kürzlich konnte gezeigt werden, dass ebenfalls mithilfe von *Generative Adversarial Networks* (GANs) eine parametrisierbare prozedurale Generierung ganzer Level umgesetzt werden kann [TU18]. Es werden bereits vorhandene Level genutzt, um neue Level zu erzeugen und anschließend aus der resultierenden latenten Menge ideale Level auszusuchen, die bestimmte Kriterien erfüllen.

Game Design Verschiedene Arbeiten demonstrieren darüber hinaus, dass auch der Teilbereich des Game Design, wie das *Balancing* von Machine Learning übernommen werden kann [AN19] [FGG18] [GA06]. Mithilfe von Reinforcement Learning werden dabei bestimmte Parameter für gegnerische Agenten oder ganz direkt der Schwierigkeitsgrad des Spiels bestimmt.

Bewegungen und Animationen Ein weiterer Bereich für potentielle Kostenreduzierungen ist die Erzeugung von *Character Content*. 2017 wurde von Holden et al. gezeigt, dass mithilfe von *Phase-Functioned Neural Networks* äußerst glaubwürdige Echtzeit-Charakteranimationen erzeugt werden können [DH17b] [DH17a]. Frei verfügbare Motion Capturing-Datenbanken wie *MoCap* von der Carnegie Mellon University [Uni01] können als Trainingsdaten dienen und erleichtern derartige Verfahren zusätzlich. NVidia und Remedy Entertainment haben ebenfalls 2017 erfolgreich gezeigt, dass mit Trainingsdaten, die lediglich aus zwei zehnminütigen Videosequenzen bestehen, in Echtzeit realistisch wirkende Gesichtsanimationen erzeugt werden können, die ausschließlich auf Audiodaten basieren [TK17] [Cor17]. Dabei wurde das Ziel verfolgt, den Sprechstil einer einzelnen Person zu modellieren. Dieses wurde deutlich überschritten, da mit den verwendeten *Deep Learning*-Methoden selbst Geschlechter, Sprachen und Akzente abgebildet werden können. Square Enix hat weiterhin bereits 2013 gezeigt, dass Reinforcement Learning verwendet werden kann, um einen Charaktercontroller zu entwerfen [Eni13]. Dieser wurde im Rahmen des Spiels *Hitman: Absolution* [Int14] entwickelt und verwendet. Damit wird gezeigt, dass Machine Learning auch in anderen Bereichen zur Verbesserung intelligenter Agenten beitragen kann. Neben den Entscheidungen, die ein solcher Agent trifft, ist auch die audiovisuelle Wirkung von hoher Bedeutung für die Authentizität und den erreichten Grad von Realismus und Immersion.

Testing und Analytics Machine Learning kann weiterhin für Test- oder Analysezwecke verwendet. In der Regel werden dann im Hintergrund oder im Backend Dienste genutzt, die dem Spieler nicht bekannt sind. Häufig stellen sich hier Klassifizierungsaufgaben, bei denen Aussagen über das Spielerverhalten getroffen werden sollen. Für eine beispielhafte Firma, die Onlinespiele vermarktet, kann es etwa vorteilhaft sein, Aussagen über das zukünftige Spielerverhalten treffen zu können - beispielsweise wann ein Spieler mit dem Spielen aufhört. Dieser kann dann explizit durch Marketingmaßnahmen anvisiert werden. Ein weiteres Anwendungsfeld ist das Durchführen von Usertests, die typischerweise in der Entwicklungsphase von Spielen durchgeführt werden, um potentielle Defizite in verschiedenen Bereichen zu identifizieren. So können etwa anhand der Dauer, die Spieler an bestimmten Orten verbringen, Heatmaps erstellt werden, um Probleme im Level- oder Game Design aufzudecken. Derartige Tests können durch Machine Learning-gestützte Verfahren verbessert werden, indem beispielsweise die Gesichtsausdrücke eines Spielers identifiziert werden, um Informationen zu seiner Stimmung zu erhalten. Derartige Vorhersagen sind auch für die Verwendung im eigentlichen Spiel selbst denkbar. Das Spielverhalten könnte analysiert werden, um beispielsweise den Schwierigkeitsgrad automatisch anzupassen oder Vorlieben des Spielers in die Generierung von prozeduralen Inhalten mit einzubeziehen.

4.3 Aktuelle Entwicklungen

In diesem Abschnitt werden aktuelle Entwicklungen im Bereich von RiL ausgeführt. Heutzutage ist vermehrt zu beobachten, dass das Verfahren für die Realisierung kompetitiver Agenten in bereits fertigen Spielen verwendet wird, die gegen echte Spieler antreten. Neueste Projekte wie *AlphaZero* [DT10] und *OpenAI Five* [CB19] beweisen, dass RiL über ein großes Potential verfügt. Die folgenden Beispiele zeigen, dass komplexe Verhaltensmuster entstehen und übermenschliche Leistungen erreicht werden können.

4.3.1 DeepMind

Der Firma DeepMind Technologies, einer Tochter von *Alphabet Inc.*, ist es mit seiner KI *AlphaZero* [DS17] [DT10] gelungen, traditionelle Brettspiele wie Schach, Jogi und Go zu meistern und selbst Weltmeister darin zu schlagen. Der seit 2013 amtierende Schachweltmeister Magnus Carlsen selbst behauptet, *AlphaZero* sei zu gut, um emuliert zu werden oder ihren Stil nachzuahmen [MC19].

Weiterhin hat DeepMind einen Agenten entwickelt, der mit Deep Reinforcement Learning einen Großteil aller Spiele der Atari-Konsole meistert [VM13]. Dafür wurde ein Convolutional Neural Network verwendet, das mit einer Variante von Q-Learning arbeitet (siehe Abschnitt 3.3.2.1). Der Input des Netzes besteht lediglich aus den auf dem Bildschirm zu sehenden Pixeln. Der Output stellt die Steuerung des Charakters dar, so wie sie auch über einen Controller möglich wäre. Die Besonderheit ist dabei, dass der Agent ohne eine Anpassung der Architektur oder des Lernalgorithmus in der Lage ist, eine Vielzahl von Spielen zu beherrschen. Konkret wurde das Verfahren mit sieben Spielen getestet, wobei in drei Fällen eine übermenschliche Performance des Agenten erreicht wurde, obwohl lediglich Pixel als Beobachtungen verwendet wurden.

4.3.2 OpenAI Five

Die Firma OpenAI LP arbeitet an vergleichbaren Projekten, allerdings auch im Bereich von Online-Games und E-Sports. Das Spiel *Dota 2* [Cor13] wird von OpenAI seit 2016 als Plattform für KI-Forschung verwendet. In dem komplexen Spiel treten zwei Teams aus jeweils fünf Helden mit einer Vielzahl von Fähigkeiten und möglichen Aktionen gegeneinander an. In diesem Rahmen wurde eine ebenfalls auf RiL basierende KI mit dem Namen *OpenAI Five* entwickelt [Ope18a] [CB19]. OpenAI Five übernimmt dabei die Kontrolle über alle fünf Charaktere eines Teams. Im Jahr 2018 schaffte es die KI nur fast, zwei professionelle E-Sports Teams zu schlagen. Im Dezember 2019 hingegen gelang ihr ein eindeutiger Sieg über das aktuelle Weltmeisterteam. Dieser Fortschritt zeigt, dass die KI selbst nach jahrelangem Training noch deutlich verbessert werden konnte. Das Training der Agenten findet in einer beschleunigten virtuellen Dota2-Umgebung statt. Dabei spielt die KI stets gegen sich selbst und hat umgerechnet bereits mehr als 10.000 Jahre Spielerfahrung gesammelt. Die Observations der Agenten entsprechen den Informationen, die auch ein menschlicher Spieler auf seinem Bildschirm sehen würde. Ebenso verhält es sich mit den möglichen auswählbaren Aktionen.

Folgende Umstände waren bei der Entwicklung der KI problematisch:

- **Langfristiger Zeithorizont:** Bei einer angenommenen Serverframerate von 30 Hz und einer Länge von durchschnittlich 45 Minuten mit Entscheidungen, die jeden vierten Frame getroffen werden ergeben sich insgesamt etwa 20.000 Aktionen pro Spiel. Zum Vergleich liegt diese Zahl beim Schach bei etwa 80 und in dem Brettspiel *GO* bei etwa 150.

4. Anwendung von Machine Learning in Videospielen

- **Eingeschränkt sichtbarer Spielzustand:** Spieler und Agent können nur in einem begrenzten Bereich um den Charakter Objekte wie Feinde, etc. wahrnehmen.
- **Hochdimensionaler kontinuierlicher Observations- und Handlungsraum:** Mit dutzenden Spielern, NPCs, Gebäuden und Fähigkeiten stehen dem Agenten zu jedem Zeitpunkt etwa 1.000 von rund 170.000 möglicher verschiedener Aktionen zur Verfügung. Zusätzlich werden etwa 20.000 Werte aus der Umgebung als relevante Beobachtungen für Entscheidungen verwendet.

Diese Hürden wurden durch ein Deep Learning-Verfahren gemeistert, dessen zugrunde liegendes rekurrentes neuronales Netz über 159 Millionen Neuronen verfügt. Dabei wurde der PPO-Algorithmus verwendet (siehe Kapitel 3.3.2.5), der auch in dieser Thesis verwendet wird. Zusätzlich wird ein *Long short-term memory* [Wik19b] mit einer Schicht und 4096 Einheiten verwendet, mit dem den Agenten ein Gedächtnis verliehen wird. Die Policy, die zum Sieg über den Dota 2-Welmeister geführt hatte, ergab sich nach einem zehn Monate andauernden Trainingsvorgang unter Nutzung von Proximal Policy Optimization bei einer durchschnittlichen neuronalen Rechenleistung von 770 PFlops/s-days, oder *pfs-day* [DA18]. Ein pfs-day entspricht einer Rechenleistung von 10^{15} neuronalen Rechenoperationen pro Sekunde für eine Dauer von einem Tag. Verglichen mit AlphaGO wurden 20 mal so viele Neuronen verwendet und 25 mal so viel Trainingszeit aufgebracht. Dadurch hat sich der in Abbildung 4.1 zu sehende Trainingsverlauf ergeben.

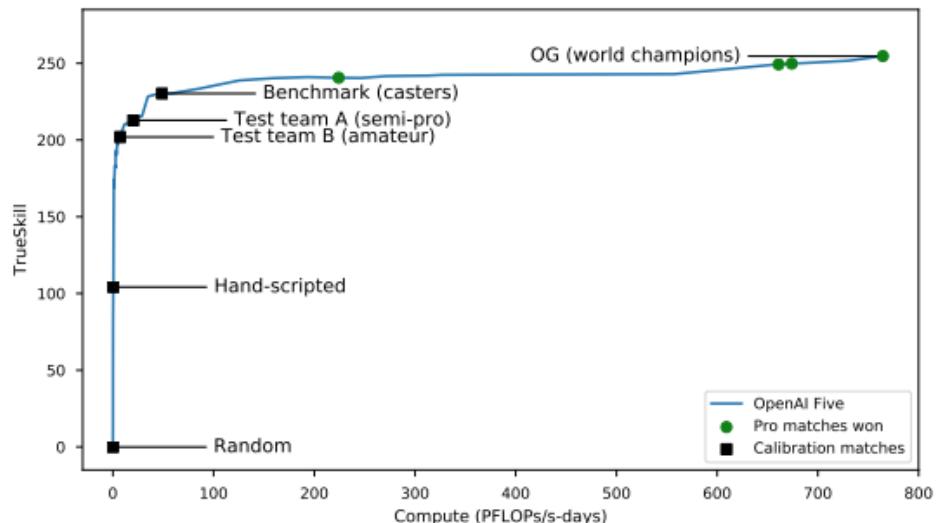


Abbildung 4.1.: Trainingsverlauf OpenAI Five [CB19]

Bei dem auf der y-Achse verzeichneten *TrueSkill*-Wert handelt es sich um eine im Jahr 2007 vorgestellte Metrik für die Einschätzung des Skill-Levels in kompetitiven Spielen [HMG07]. Die Grafik zeigt einen zunächst drastischen Anstieg aufgrund der exponentiellen Natur des Problems, sodass bereits nach sehr kurzer Zeit semiprofessionelle Teams geschlagen werden konnten. Das ist bereits eine sehr aussagekräftige Metrik, da auch semiprofessionelle menschliche Spieler jahrelanges intensives Training absolvieren müssen, um dieses Level zu erreichen. Der anschließende Verlauf ist nur sehr leicht steigend. Verbesserungen zum Ende des Verlaufs wurden mit der Einführung weiterer Fähigkeiten, die der KI zuvor nicht verwenden konnte, begründet, sodass schließlich auch das aktuell beste menschliche Team geschlagen werden konnte.

Es zeigt sich, dass selbst überaus komplexe Spiele und damit schwierige Probleme mit RL gemeistert werden können. Sie beweisen zum Einen die Skalierbarkeit klassischer Machine Learning-Methoden und zum Anderen die Möglichkeit, Agenten in Verbindung mit einer großen Menge von Rechenleistung

übermenschliche Fähigkeiten zu verleihen. Dabei dienen Videospiele als eine Art „Spielplatz“ für das Experimentieren und Forschen mit Machine Learning-Verfahren, da diese eine optimale Umgebung dafür bieten.

4.4 Potential

Die oben genannten Beispiele zeigen, dass es eine Vielzahl von Anwendungsfällen für Machine Learning in Videospielen gibt. Neben der Verbesserung intelligenter Agenten sind auch andere Bereiche relevant, die zur Erschaffung von Welten mit einer erhöhten Glaubwürdigkeit und Authentizität beitragen können. Die genannten Spiele zeigen, wie kreativ lernende Agenten in sehr unterschiedlichen Bereichen eingesetzt werden können und welche interessante Szenarien damit vorstellbar sind. Die Nutzung in Strategiespielen beweist darüber hinaus die Skalierbarkeit der Verfahren. Sie ermöglichen eine Interaktion mit dem Spieler, die mit klassischen Verfahren nicht realisierbar ist und erzeugen so authentische Spielerlebnisse.

Der Entwickler von *Supreme Commander 2* und *Planetary Annihilation*, nennt eine Menge von Vorteilen, die sich direkt aus der Nutzung eines neuronalen Netzes ergeben:

- **Gewichtung der Inputs:** Eine Gewichtung der Inputs fällt weg.
- **Wichtigkeit der Inputs irrelevant:** Wenn irrelevante Inputs vorhanden sind, sollte das System davon nicht beeinflusst werden.
- **Vergleichbarkeit der Zustandsvariablen:** Es ist nicht nötig, die einzelnen Werte miteinander zu vergleichen. Wäre dies nicht der Fall, so müsste ein Weg gefunden werden, um beispielsweise Inputvariablen wie *Geschwindigkeit* und *Schaden* vergleichen zu können.
- **Kein Algorithmus nötig:** Es muss kein Algorithmus entwickelt werden, der das genaue Verhalten der Einheiten beschreibt.

Laut Robbins bestehe ein großer Vorteil also darin, dass fast beliebige Werte als Inputvariablen für das neuronale Netz verwendet werden können. Das neuronale Netz kümmere sich dann anstelle des Entwicklers von selbst um die Normalisierung und den Vergleich der Variablen. Weiterhin sei es nicht nötig das Verhalten der Agenten direkt zu beschreiben, sodass insgesamt viel Arbeitszeit gespart wird.

Im Allgemeinen können außerdem Kosten gespart werden, indem Inhalte generiert werden, anstatt dafür einen hohen Arbeitsaufwand aufzubringen. Besonders bei der Nutzung intelligenter Agenten besteht das Potential, den massiven Zeitaufwand des Scriptings zu reduzieren, indem beispielsweise Reinforcement Learning genutzt wird. Das Verhalten der Agenten muss dann nicht explizit definiert werden, weil es *erlernt* wird. Die Menge der Inhalte, die für ein erfolgreiches Spiel nötig sind, sollten dabei nicht unterschätzt werden. Im 2018 erschienenen Spiel *Red Dead Redemption 2* [Stu18] wurden allein etwa 300.000 Charakter- und Tieranimationen angefertigt bzw. aufgenommen [Thu18]. Dadurch kann ein guter Eindruck darüber gewonnen werden, wie viel Zeit und Geld durch eine prozedurale Generierung von Animationen gespart werden könnte. Aktuelle Entwicklungen beweisen ein vielversprechendes Potential, da selbst in komplexen Szenarien menschenähnliches oder sogar übermenschliches Verhalten erlernt werden kann. Reinforcement Learning zeigt, dass auch mit altbewährten Algorithmen erstaunliche Ergebnisse erreicht werden können, die in hohem Maße und durch den erhöhten Einsatz von leistungsfähiger Hardware skalierbar sind. Das lässt für die Entwicklung von NPCs vermuten, dass ebenfalls menschenähnliches Verhalten nachgeahmt werden kann, um die Immersion virtueller Welten zu erhöhen.

5

ML-Agents Toolkit

In diesem Kapitel wird das Unity-Framework *ML-Agents* für die Entwicklung intelligenter Agenten auf der Grundlage von Reinforcement Learning vorgestellt. Der erste Abschnitt befasst sich mit einer Einleitung in die Unity-Engine. Anschließend werden der Aufbau und die Funktionsweise von ML-Agents skizziert und anhand einer Beispielimplementierung verdeutlicht.

5.1 Unity

Unity ist eine State-of-the-Art Echtzeit-Entwicklungsplattform der Firma *Unity Technologies* für 3D-Simulationen und wird neben Teilbereichen wie Architektur und Industrie hauptsächlich für den Games-Bereich verwendet [Tec05]. Zahlreiche Zielplattformen wie *Microsoft Windows* und diverse Spielekonsolen werden unterstützt. Unity liefert Features wie eine weit entwickelte Grafik- und Physik-Engine sowie einen modifizierbaren Editor für die Realisierung komplexer Projekte. Mit dem Editor können sämtliche gängigen Prozesse aus der Spieleentwicklung genutzt werden, beispielsweise aus den Bereichen „Networking“ oder „Asset-Management“. Unity's eingebaute Mechanismen werden dabei durch Scripte ergänzt, um die Logik von Spielen zu implementieren. Abbildung 5.1 zeigt eine Momentaufnahme aus einer in Echtzeit berechneten Unity-Techdemo, um die Möglichkeiten der Engine zu verdeutlichen.



Abbildung 5.1.: Unity-Echtzeitgrafik in *The Heretic* [Tec19a]

Unity wendet sich sowohl an semiprofessionelle als auch hauptberufliche Spieleentwickler und große Entwicklerstudios. Mit einem Marktanteil von etwa 48% im Vergleich zu den lediglich 13% der *Unreal Engine* stellt sie die mit Abstand verbreiteteste Game Engine dar [VG20].

Grundlagen Unity verwendet die Programmiersprache C# und erweitert diese um zusätzliche Konzepte, die speziell für die Entwicklung von Spielen vorgesehen sind. Alle in der Spielszene verwendeten Entitäten - unabhängig davon, ob sie über eine visuelle Repräsentation verfügen - basieren auf sogenannten *GameObjects*. Sie dienen als fundamentale Container, denen beliebig viele Komponenten hinzugefügt werden können. Scripts, die direkt an ein GameObject gebunden werden, heißen *MonoBehavior* und unterstützen eingebaute Funktionen der Engine wie beispielsweise die Funktion *Update*, die mit jedem neuen Frame aufgerufen wird. Komplexe hierarchische Konstrukte von *GameObjects* können anhand von sogenannten *Prefabs* zu Vorlagen transformiert werden, die auf der Festplatte gespeichert werden. Prefabs erlauben die Änderung von allen Objekten eines Typs durch die simple Anpassung des Basisobjekts. Weiterhin können große Mengen von Objekten desselben Typs leicht instantiiert werden. Ein weiteres wichtiges Konzept sind *Coroutines*, welche in der Lage sind, die Ausführung einer Methode anzuhalten, um auf ein Ergebnis zu warten [Tec19b]. Das geschieht mit dem Stichwort *yield*. Weiterhin können Coroutinen eigene Threads zugewiesen werden, indem *AsyncCoroutines* verwendet werden. Coroutines geben ein Ergebnis vom Typen *IEnumerator* zurück. Dieser Typ ist nicht Unity-basiert, sondern stammt aus dem Microsoft .NET-Framework. Coroutinen können nur von Monobehaviors durch die Funktion *StartCoroutine* und *StartCoroutineAsync* gestartet werden.

Unity Editor Unity liefert einen umfangreichen Editor für die Entwicklung von Spielen und Simulationen. Bis auf die Anfertigung von Assets und der Entwicklung von Programmcode finden alle Vorgänge zur Entwicklung eines Spiels in Unity selbst statt. Scripts werden in einer externen C#-fähigen IDE entwickelt, wie zum Beispiel Visual Studio oder vergleichbaren Anwendungen. Ebenso werden Modelle und Grafiken in externen Anwendungen angefertigt. Der Editor arbeitet szenenbasiert, wobei jede Szene ein eigenständiges Level darstellt. Im Verlauf dieser Arbeit wurde lediglich eine Szene für die Simulation des Dorfes entwickelt. Mit der Anwendung werden die folgenden Aufgaben bearbeitet:

- Verwaltung eines komponentenbasierten Systems, Erzeugung und Manipulation von GameObjects
- Bereitstellung einer Engine für physikalische Vorgänge wie Bewegungen, Kollisionen, etc.
- Rendering der Szene, Möglichkeit der Nutzung von Render-Pipelines
- Gestaltung des Levels mit visuellen Komponenten wie Mesh, Shadern, Texturen, Beleuchtung, Partikel, etc.
- Grundlegende Erzeugung und Manipulation weiterer Assets wie Sounds und Animationen (häufig in externen Programmen gelöst)
- Nutzung von Programmcode durch Scripts
- Verwaltung von Assets wie Sounds, Grafiken und 3D-Modellen
- Bereitstellung von speziellen Features wie Pathfinding und Leistungsoptimierungen wie *Occlusion Culling* [Tec19c]
- Bereitstellung diverser Plugins zur Erweiterung des Editors in beliebigen Bereichen
- Profiling und Debugging der Anwendung
- Ausspielen der Anwendung auf verschiedene Plattformen

5. ML-Agents Toolkit

Die Oberfläche des Editors wird in der Abbildung 5.2 gezeigt. Das Bild zeigt die fundamentalen Komponenten des Editors.



Abbildung 5.2.: Oberfläche des Unity Editors

Auf der rechten Seite ist der *Inspector* zu erkennen, der die Inspektion sowie das Hinzufügen und die Manipulation von Komponenten eines GameObjects erlaubt. Der Inspector zeigt die Komponenten des in der Szene ausgewählten GameObjects, die auf der linken Seite zu sehen ist und in der navigiert werden kann. In der Mitte der Abbildung ist die Szenenhierarchie zu erkennen, die den Aufbau der Szene beschreibt und Abhängigkeiten von GameObjects definiert. Der untere Bereich zeigt die Ordnerstruktur des Projekts mit Assets, Scripts, Bibliotheken etc. Ein großer Vorteil von Unity ist weiterhin die Möglichkeit, entwickelte Spiele direkt im Editor auszuführen, ohne dass Build-Vorgänge für spezifische Zielplattformen notwendig sind. Änderungen in der Szene oder im Programmcode können daher schnell getestet werden.

5.1.1 Machine Learning in Unity

Vor der Einführung von ML-Agents musste für die Nutzung von Machine Learning in Unity ein deutlich höherer Aufwand betrieben werden, da ausschließlich Community-Plugins verfügbar waren. Diese wurden sowohl genutzt, um eine Verbindung zu externen Schnittstellen wie Tensorflow aufzubauen, oder aber, um selbst neuronale Netzwerke zu implementieren. Da diese jedoch lediglich von Nutzern aus der Unity-Gemeinschaft entwickelt wurden, bestand aufgrund offensichtlicher Nachteile kein großes Interesse an der Verwendung von Machine Learning in Unity. Mit der Entwicklung des *ML-Agents Toolkit* wird der Einstieg in die Nutzung von Machine Learning in Unity extrem vereinfacht. Das Framework wird im folgenden Abschnitt näher skizziert.

5.2 ML-Agents-Architektur

ML-Agents ist ein von Unity Technologies entwickeltes Open-Source Framework, das Spielen und Simulationen erlaubt, als Entwicklungs- und Testumgebung für lernende Agenten zu fungieren [AJ18] [Tec17d] [Tec17c]. Es befindet sich aktuell in der Beta-Phase, verfügt aber bereits über alle grundlegenden Features und ist öffentlich nutzbar. Mit ML-Agents wurde eine Plattform geschaffen, die die Entwicklung von lernenden Agenten einer größeren Menge von Entwicklern zugänglich machen, da sie vorhandene Hürden abbaut und den nötigen Aufwand reduziert.

In der Regel wird ML-Agents in der Kombination mit Imitation Learning oder Reinforcement Learning verwendet (siehe Abschnitte 3.1.1.1 und 3.1.3). Andere Verfahren wie genetische Algorithmen (z.B. *NeuroEvolution*), können ebenfalls verwendet werden, sind aber weniger gut dokumentiert und weniger üblich. Für den Trainingsvorgang stellt das Framework eine Verbindung zu einer Python-API her. Verschiedene State-Of-The-Art-Algorithmen wie PPO und *Soft Actor Critic* (SAC) werden dem Nutzer zur Verfügung gestellt. Das genaue Anwendungsszenario des Frameworks bleibt dabei dem Entwickler überlassen; es kann sowohl für die Verhaltenssimulation von NPCs als auch für automatisierte Tests oder der Evaluation des Game Designs verwendet werden [Tec17c]. ML-Agents bietet eine intuitive Architektur sowie eine große Menge von veränderbaren Parametern, um eine flexible Entwicklung von Agenten zu gewährleisten. Im Folgenden wird das Framework ausschließlich in der Kombination mit Reinforcement Learning verwendet.

5.2.1 Grundlegende Funktionsweise

Zur Installation wird ein `git clone` des ML-Agent GitHub-Archivs [Tec17c] in ein leeres Unity-Projekt durchgeführt. Zusätzlich wird das `mlagents` Python Package benötigt. Diverse Beispiele sind bereits im Projekt enthalten und können zum Verständnis verwendet werden. Alternativ wird die Verwendung eines Docker-Image angeboten.

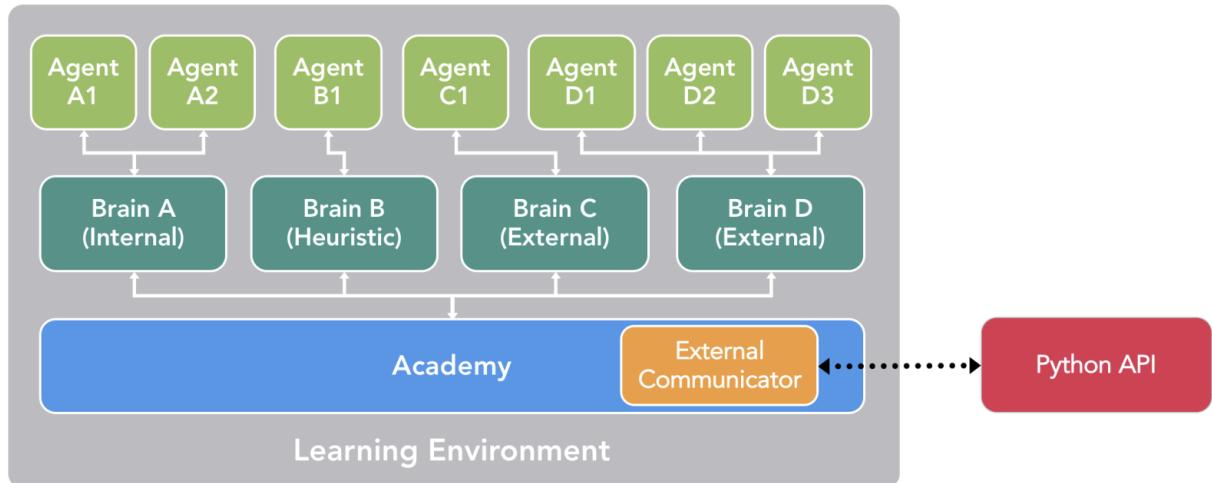


Abbildung 5.3.: ML-Agents Architektur [Tec17d]

Vier grundlegende Klassen müssen in einem ML-Agents-Projekt verwendet werden, um einen lernenden intelligenten Agenten zu entwickeln. Im Einzelnen handelt es sich dabei um die folgenden zu implementierenden Konzepte, zu denen eine kurze Übersicht geliefert wird.

5. ML-Agents Toolkit

- **Agent:** Sammelt Observations und führt Aktionen aus.
- **Brain:** Das Gehirn der AI, trifft Entscheidungen.
- **Academy:** Zentrale Verwaltung aller Brains, externe Verbindung zu Python.
- **Model:** Resultat des Lernvorgangs, enthält Policy für ein Brain.

Die Abbildung 5.3 zeigt die zugrunde liegende Architektur, die die erwähnten Komponenten beinhaltet. Im Folgenden wird die Funktionsweise der einzelnen Konzepte beschrieben. Weiterhin wird näher darauf eingegangen, was bei einer Implementierung zu beachten ist.

5.2.2 Academy

Die Academy ist für die globale Koordinierung der Simulation zuständig. Sie dient als eine zentrale Verwaltungseinheit für alle Agenten und Brains. Durch die Herstellung der Verbindung zu einer externen Schnittstelle in Form eines Tensorflow-Backends stellt sie zusätzlich die Trainingsumgebung dar. Jegliche Machine Learning-Vorgänge sind vom eigentlichen Unity-Projekt vollständig entkoppelt, da der Entscheidungsprozess in Python abläuft.

Um Entscheidungen für Agenten zu treffen, findet eine kontinuierliche Kommunikation zwischen der Academy und der Python-API über den *External Communicator* statt (siehe Abbildung 5.3), indem die Observations der Agenten an Tensorflow weitergeleitet werden. Tensorflow verfügt dabei über alle für das Training verwendeten Machine Learning-Algorithmen. Ein neuronales Netz innerhalb des Backends erzeugt einen Output, der die Entscheidung des Agenten in Abhängigkeit von den zuvor gesammelten Observations darstellt. Dieser Vorgang gilt sowohl für den Trainings- als auch für den Inference-Vorgang.

Zum Beginn der Ausführung ruft die Academy die Funktion `AgentReset` für alle Agenten auf. Anschließend werden alternierend die Funktionen `CollectObservations` und `AgentAction` aufgerufen. Jeder Vorgang beinhaltet die Erzeugung einer Entscheidung durch das neuronale Netz und die Ausführung der entsprechenden Aktion durch den Agenten. Dies wird solange wiederholt, bis der Agent eine Episode abgeschlossen hat oder die maximale Schrittweite erreicht wurde. Anschließend wird der Prozess mit dem Aufruf von `AgentReset` erneut gestartet.

5.2.3 Agent

Um einen neuen Agenten zu erzeugen, muss eine neue Klasse angelegt werden, die von der Oberklasse `Agent` erbt. Diese beschreibt die grundsätzlichen Aufgaben und nötigen Vorgänge eines lernenden Agenten. `Agent` erbt von der Unity-Basisklasse `Monobehavior` und verfügt daher über automatisch verwaltete Funktionen wie `Update` (siehe Abschnitt 5.1). Jeder Agent wird daher an ein `GameObject` angehängt. Folgende Funktionen müssen implementiert werden:

- **InitializeAgent**: Beinhaltet die einmalige Initialisierung des Agenten.
- **CollectObservations**: Vor jeder Aktion müssen Observations aus der Umgebung des Agenten gesammelt werden. Das können beispielsweise die Position des Agenten und eines Ziels oder dessen Lebenspunkte sein. Beobachtungen werden gesammelt, indem **AddVectorObs** aufgerufen wird. Diese Funktion unterstützt Zahlenwerte, Vektoren, Strings und Boolesche Werte. Weiterhin besteht die Möglichkeit, visuelle Beobachtungen zu verwenden. In dieser Arbeit wird diese Option nicht weiter behandelt.
- **AgentAction**: Die Parameter der Funktion beschreiben den Output des neuronalen Netzes. Der Agent muss entsprechend darauf reagieren und die jeweiligen Aktionen ausführen.
- **AgentReset**: Damit wird der Agent inklusive etwaiger Abhängigkeiten zurückgesetzt. Soll der Agent beispielsweise lernen, auf einer Plattform zu balancieren, so würde er beim Fallen von der Plattform wieder auf dieser platziert werden. Zusätzlich könnte die Plattform wieder horizontal ausgerichtet werden. **AgentReset** wird automatisch aufgerufen, wenn eine Episode endet, indem **Done** aufgerufen wird.

Essentiell sind die Funktionen **CollectObservations** für das Sammeln von Beobachtungen und **AgentAction**, um zu bestimmen, welche Aktion gewählt werden soll. Bei der Entscheidung der Aktionen kann zwischen zwei Szenarien gewählt werden.

Bei den *diskreten* Entscheidungen wird anfangs eine Menge von möglichen Entscheidungen festgelegt, denen ein Index zugewiesen wird. In den Parametern von **AgentAction** wird dann ein einzelner Wert übermittelt, der eine ausgewählte spezifische Aktion referenziert, die ausgeführt werden soll. Der Output des neuronalen Netz bestimmt also die Ausführung einer bestimmten Aktion, indem ein ausgewählter Index für eine Aktion übermittelt wird. Diese Indizes werden durch Integer dargestellt, die bei 0 aufsteigend beginnen. Bei den *kontinuierlichen* Entscheidungen hingegen wird in jedem Fall für alle festgelegten Aktionen je ein Wert geliefert. Dabei werden keine Indizes, sondern skalare Werte anhand von Floats übermittelt, die die Amplitude für die jeweiligen Aktionen bestimmen. Diese haben einen Zahlenbereich von -1 bis $+1$. Ein mögliches Anwendungsszenario ist die Bewegung eines virtuellen Charakters. Es werden drei Aktionen festgelegt, um den Agenten erstens vor- und rückwärts und zweitens seitwärts zu bewegen. Die dritte Aktion bestimmt die Rotation des Agenten. Die Amplitude bestimmt nun, wie stark der Agent in eine Richtung bewegt oder gedreht werden soll, bzw. in welche Richtung er dabei vorgehen soll. Anstatt sich für eine Aktion zu entscheiden wird jeder möglichen Aktion also ein skalarer Wert zugewiesen, die den kontinuierlichen Ausschlag für die damit assoziierte Aktion bestimmt.

Eine essentielle Aufgabe des Agenten ist darüber hinaus die Erzeugung von Belohnungen. Die Funktion **AddReward** nimmt einen positiven oder negativen Float-Wert, der eine Belohnung oder eine Bestrafung darstellt. Es handelt sich also um einen skalaren Wert der beschreibt, wie gut der Agent agiert (siehe Kapitel 3). Das mehrfache Aufrufen von **AddReward** führt zu einer Akkumulierung der Belohnungen. Außerdem besteht die Möglichkeit, den Reward direkt über **SetReward** festzulegen, wodurch bisherige Belohnungen für die ausgeführte Aktion ignoriert werden. Beim Training werden diese Belohnungen genutzt, um Beziehungen zwischen dem Spielzustand und den möglichen Aktionen herzustellen.

Zu Testzwecken können Entscheidungen vom Spieler selbst bestimmt werden. Dafür muss die Funktion **Heuristic** in der Agentenklasse überschrieben werden. Die Aktionen des Agenten werden dann nicht durch das Brain, sondern vom Spieler selbst bestimmt, indem den Aktionen Tasturbefehle zugewiesen werden. Vor dem Training können so leicht Fehler entdeckt und die Funktionsweise eines Agenten getestet werden.

5.2.4 Brain

Zwischen der Academy und den Agenten existiert eine weitere zwischengeschaltete Entität namens *Brain*. Ein Brain kann beliebig viele Agenten verwalten (sieht Abbildung 5.3). Jeder dieser Agenten verfügt über eine Referenz auf das Brain. Die Aufgabe des Brains ist die Bereitstellung einer Policy oder *Model* für alle mit ihm assoziierten Agenten und die Weiterleitung von Observations und Entscheidungen zwischen den Agenten und der Academy. Für jedes Brain wird im Tensorflow-Backend ein eigenes neuronales Netz angelegt. Im Trainingsmodus werden für jedes Brain separate Hyperparameter anhand einer Konfigurationsdatei zur Verfügung gestellt, die die Beschaffenheit des zugrunde liegenden neuronalen Netzes bestimmen. Jedes Brain wird außerdem über Parameter konfiguriert, die beispielsweise die Anzahl der Beobachtungen oder die Anzahl der möglichen Aktionen festlegen. Auch der Ablauf des Entscheidungsprozesses kann durch Parameter verändert werden: Standardmäßig wird nach einer festlegbaren Anzahl von Zeitschritten automatisch eine neue Entscheidung angefragt. Alternativ können sogenannte *On Demand Decisions* verwendet werden. Der Agent beantragt dann selbst eine neue Entscheidung, wenn seine vorherige Entscheidung abgeschlossen ist. Dies ist zum Beispiel nützlich, wenn die Dauer der Aktionen nicht absehbar ist.

5.2.5 Model

Beim *Model* handelt es sich um eine Policy bzw. ein Modell, das aus dem Trainingsvorgang eines Agenten hervorgegangen ist. Eine Policy ist nichts weiteres, als die Beschreibung der Parameter θ eines neuronalen Netzes, also der Gewichtungen der Neuronen. Für den Inference-Modus muss jedes verwendete Brain über ein *Model* verfügen. Ein Agent führt dann Aktionen aus, die das fertige Model aufgrund der Beobachtungen bereitstellt, ohne dass ein Trainings- oder Lernvorgang stattfindet.

Da die Parameter des Brains wie beispielsweise die Anzahl der Beobachtungen und möglichen Aktionen die Struktur des zugrunde liegenden neuronalen Netzes beeinflusst, ist das Model von ihnen abhängig. Sobald Parameter eines Brains verändert werden, kann ein trainiertes Model nicht mehr verwendet werden. In diesem Fall muss ein neuer Trainingsvorgang erfolgen.

5.2.6 Training

Während des Vorgangs versenden die Brains Observations der mit ihnen verknüpften Agenten über den External Communicator an die Python API und erhalten Aktionen als Antwort zurück. Innerhalb von Tensorflow findet dann der Lernprozess statt, in welchem die beste Policy für das jeweilige Brain gefunden wird. Nach dem Training wird das gelernte Model anhand einer Tensorflow Model-Datei mit der Endung *.nn* gespeichert. Das Training kann sowohl mit nur einem als auch mit mehreren Agenten und Brains ablaufen. Die Verwendung mehrere Agenten kann die Geschwindigkeit und Stabilität des Trainings erhöhen [Tec17d]. Für das Training werden die Framerate des Spiels und die Qualität der visuellen Darstellungen reduziert. Weiterhin kann die sogenannte *Time Scale* verändert werden, um schnellere Aktionen von den Agenten zu provozieren. Die in Unity eingebaute Time Scale simuliert eine Beschleunigung der Zeit. Sämtliche physikalische Aktionen laufen dann schneller ab. Update-Funktionen werden noch immer im selben Abstand aufgerufen, da sie entweder von der Framerate oder einem festen Intervall abhängen. Die scheinbar vergangene Zeit, die in Unity mithilfe von `Time.deltaTime` aufgerufen werden kann, erhöht sich dann proportional zum Time Scale-Faktor.

Um einen Trainingsvorgang zu starten, können Befehle des *mlagents* Python Package verwendet werden. Diese sind unter `ml-agents/mlagents/trainers/learn.py` zu finden. Wichtig ist außer-

5. ML-Agents Toolkit

dem, dass in der Academy die zu trainierenden Brains referenziert werden und jeweils die Flag *Control* gesetzt wird. Folgende Anweisung wird verwendet, um das Training zu starten:

```
mlagents-learn <config-file> -env=<env_name> -run-id=<some-id> -train
```

Der Parameter `env_name` bezieht sich auf den Pfad der kompilierten ausführbaren Unity-Simulation. Wenn der Parameter nicht gesetzt wird, erfolgt das Training im Unity Editor. Nach dem Training kann mit dem Befehl `tensorboard -logdir=summaries -port <port-id>` ein *Tensorboard* verwendet werden. Ein lokaler Webserver wird dann gestartet, über den man in Echtzeit eine Visualisierung des Trainingsverlaufs betrachten kann. Über das Tensorboard können außerdem bereits vorhandene Trainingsdurchläufe evaluiert werden. In Abbildung 5.4 ist ein Tensorboard-Beispiel zu sehen. Zahlreiche Informationen zum Ablauf der jeweiligen Vorgänge stehen zur Verfügung. Dabei können anhand von Diagrammen Metriken wie die Belohnung, die Anzahl der Aktionen pro Episode, die Entropie der Policy und Weitere eingesehen werden.

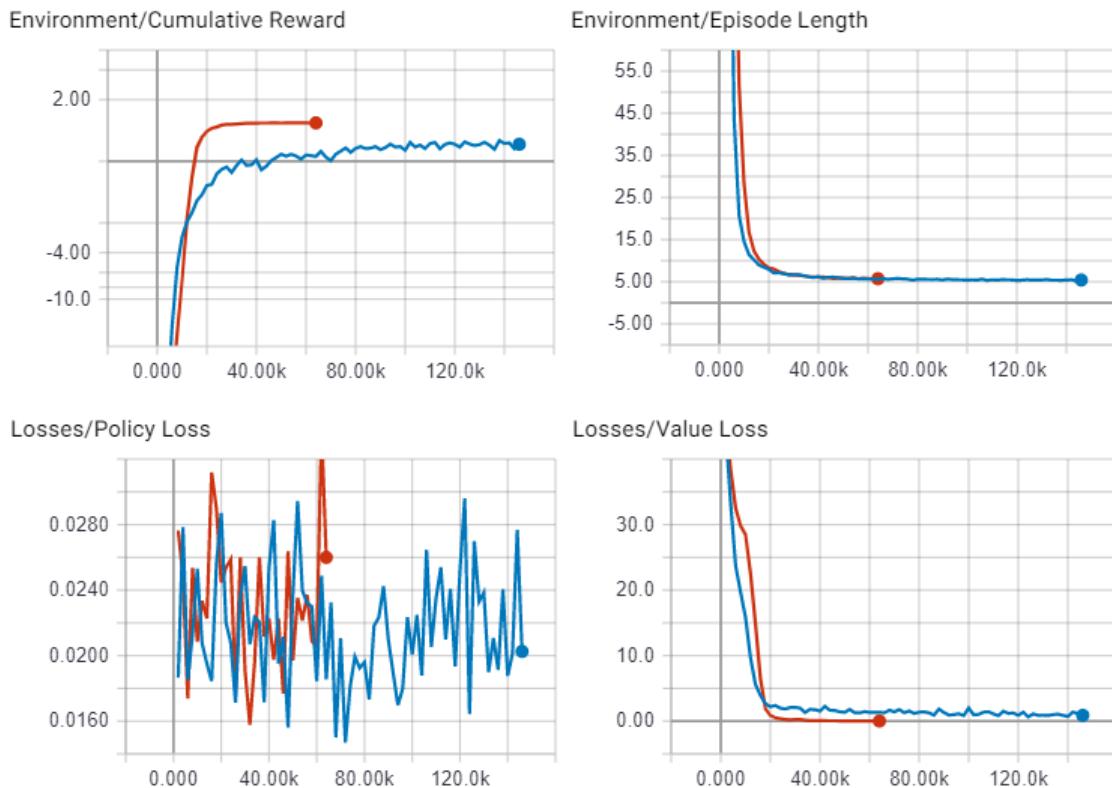


Abbildung 5.4.: Tensorboard-Beispiel

Weiterhin ist eine beispielhafte Konfigurationsdatei unter `config/trainer_config.yaml` vorhanden. In dieser Konfiguration werden die Hyperparameter für das Training festgelegt. Dadurch kann unter anderem beschrieben werden, wie viele *Hidden Units*, also versteckte Neuronen, das neuronale Netz in wie vielen Schichten jeweils enthalten soll, oder auf welchen Wert die Lernrate festgelegt wird. Im Kapitel 7 werden die einzelnen Hyperparameter näher beschrieben. Die Tabelle 7.1 demonstriert eine Übersicht über die relevanten Parameter.

5.3 ML-Agents-Beispiel

Bei der Einarbeitung in das ML-Agents Toolkit wurde aus Gründen des Verständnisses und der Interesse ein Beispielprojekt angefertigt. Anhand dieses Beispiels soll ein erweitertes Verständnis über die Funktionsweise des Frameworks entstehen. Das Projekt wird im Folgenden vorgestellt. Der Quellcode für das Projekt kann im dazugehörigen GitHub-Repository betrachtet werden [Jan20].

5.3.1 Aufgabe

Das Projekt beinhaltet einen einzelnen Agenten in Form eines Balls. Dieser soll lernen, sich zu einem Ziel (*Target*) zu bewegen. Sobald der Agent das Ziel erreicht, erhält der Agent eine Belohnung $r = 1$ und das Ziel wird zu einem anderen Ort bewegt. Sowohl Agent als auch Target befinden sich auf einer Plattform, von der der Agent herunterfallen kann. In diesem Fall bekommt der Agent eine hohe Bestrafung $r = -1$ und die Episode wird zurückgesetzt. Eine weitere Bestrafung $r = -0.1$ wird jedes Mal vergeben, wenn der Agent sich in die falsche Richtung bewegt. Die Abbildung 5.5 zeigt den simplen Aufbau der Szene.

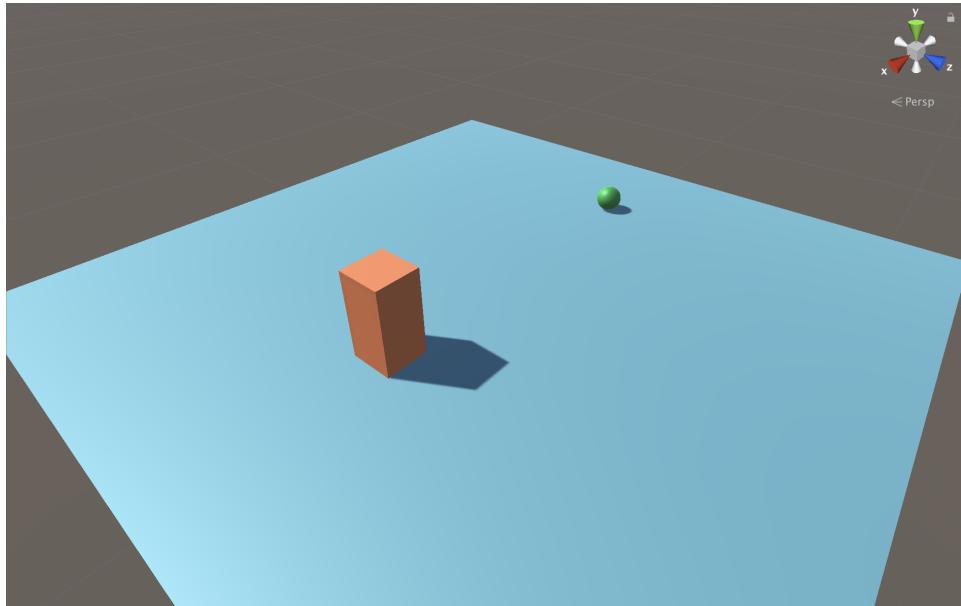


Abbildung 5.5.: ML-Agents-Beispiel, Aufbau der Szene

5.3.2 Implementierung des Agenten

Dieser Abschnitt zeigt, wie die einzelnen wichtigen Funktionen des Agenten implementiert wurden.

CollectObservations Das folgende Listing 5.1 zeigt die Implementierung der Funktion `CollectObservations`. Die Wahl der Observations ist essentiell, damit sinnvolle Beziehungen zwischen Spielzustand und Belohnungswerten hergestellt werden können. Für die Lösung der Aufgabe sind vor allem die Position des Agenten und des Targets relevant, da der Agent sich darüber bewusst sein muss, in welche Richtung er sich bewegen muss (siehe Zeilen 3, 4, 8 und 9). Zusätzlich wird dafür die aktuelle Geschwindigkeit des Agenten verwendet. Diese wird von der *RigidBody*-Komponente bezogen, die *GameObjects* Interaktionen mit der Physik-Engine ermöglicht. Um Informationen darüber zu liefern, ob der Agent sich im Fallen befindet, wird weiterhin ein Boolescher Wert als

5. ML-Agents Toolkit

Beobachtung hinzugenommen. Dadurch soll die Policy leichter eine Verbindung zu dem Grund der hohen erfahrenen Bestrafung erkennen können.

```
1  public override void CollectObservations() {  
2      AddVectorObs(this.transform.position.x);  
3      AddVectorObs(this.transform.position.z);  
4      AddVectorObs(_isAgentFalling());  
5      AddVectorObs(_agentRigidBody.velocity.x);  
6      AddVectorObs(_agentRigidBody.velocity.z);  
7      AddVectorObs(target.transform.x);  
8      AddVectorObs(target.transform.z);  
9  }
```

Listing 5.1: ML-Agents-Beispiel, CollectObservations

AgentAction Zunächst wird ein Brain für den Agenten erstellt und konfiguriert, wie in Abbildung 5.6 erkennbar. Es soll ein kontinuierlicher Aktionsraum mit zwei Achsen verwendet werden, um den Agenten auf zwei Achsen zu bewegen. Für jede Aktion erhält der Agent dann zwei Werte, die seine Geschwindigkeit in X- und Z-Richtung beeinflussen. Weiterhin werden die in der Funktion CollectObservation getätigten Beobachtungen gezählt und eingetragen.

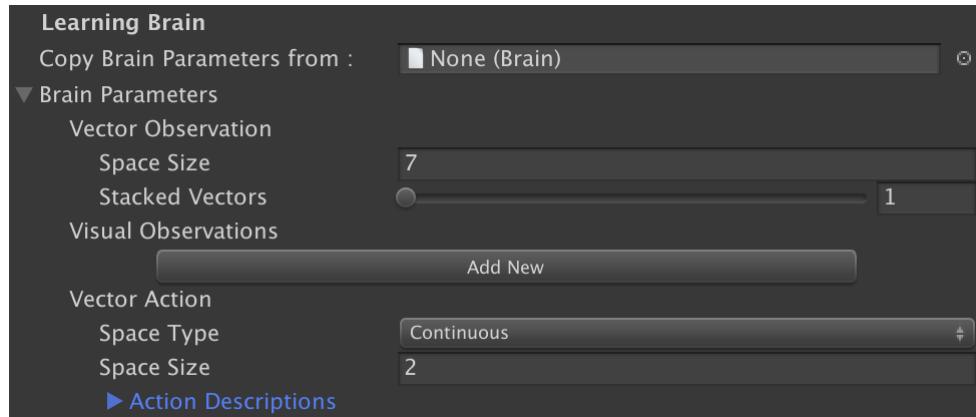


Abbildung 5.6.: ML-Agents-Beispiel, Konfiguration des Brains

Das folgende Listing 5.2 zeigt die Aktionen, die der Agent ausführt, sowie die Verteilung der Belohnungen. Der Output des neuronalen Netzes wird in Form eines Float-Array zur Verfügung gestellt.

```

1  public override void AgentAction(float[] vectorAction, string
        textAction) {
2      // The agent fell off the edge. Finish episode with a punishment.
3      if (_isAgentFalling()) {
4          AddReward(-1f);
5          Done();
6          return;
7      }
8      // The agent has reached the target. Finish the episode with a
     reward.
9      float currentDistance = Vector3.Distance(transform.position,
     target.transform.position);
10     if (currentDistance < CLOSE_DISTANCE) {
11         AddReward(1f);
12         Done();
13         return;
14     }
15     // Set the agents velocity to what the model thinks is a good
     value.
16     float moveY = vectorAction[0] * SPEED_MULT;
17     float moveX = vectorAction[1] * SPEED_MULT;
18     _agentRigidbody.velocity = new Vector3(moveX, 0f, moveY);
19
20     // In case the agent moved in the wrong direction, punish it.
21     if (currentDistance > _lastDistance) AddReward(-0.1f);
22     _lastDistance = currentDistance;
23 }
```

Listing 5.2: ML-Agents-Beispiel, AgentAction

AgentReset Weiterhin muss die Funktion `AgentReset` implementiert werden, um den Agenten und seine Umgebung zurückzusetzen (siehe Listing 5.3). Die Geschwindigkeit des Agenten wird zurückgesetzt und die Position des Targets auf der Plattform wird zufällig ausgewählt. Falls der Agent von der Plattform gefallen ist, wird auch dessen Position zufällig zurückgesetzt. Das Zurücksetzen der Positionen ist simpel, da die Plattform ihr Zentrum auf dem Koordinatennullpunkt besitzt. Dabei wird darauf geachtet, dass Agent und Target in einem gewissen Mindestabstand zueinander platziert werden.

```

1  public override void AgentReset() {
2      _agentRigidbody.velocity = _agentRigidbody.angularVelocity =
     Vector3.zero;
3      if (_isAgentFalling()) _setRandomAgentPosition();
4      _setRandomTargetPosition();
5      _lastDistance = Vector3.Distance(transform.position, target.
     transform.position);
6  }
```

Listing 5.3: ML-Agents-Beispiel, AgentReset

Multi Agent Training Um die Laufzeit des Trainings zu verringern, wurde ein *Multi-Agent-Training* verwendet. Mehrere Agenten lösen gleichzeitig die gleiche Aufgabe, sind aber in ihrem eigenen Umfeld gekapselt und interagieren nicht mit anderen Agenten oder Umgebungen. Durch die Anfertigung eines Prefabs, das das Spielfeld und den Agenten umgibt, kann dies sehr schnell realisiert

werden. Geringfügige Änderungen des Codes sind dafür notwendig, die im GitHub-Repository eingesehen werden können. Beispielsweise muss ein Offset bei der Positionierung von Agenten und Targets sowie bei der Sammlung von Observations verwendet werden, das von ihren initialen Positionen abhängt. Der aktualisierte Aufbau der Szene ist in Abbildung 5.7 erkennbar.

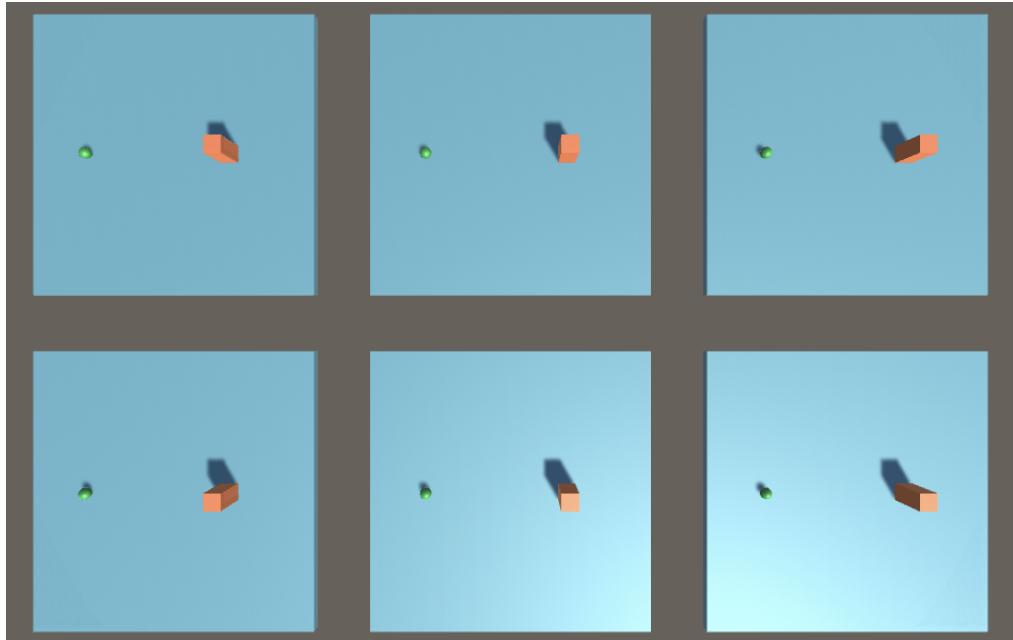


Abbildung 5.7.: ML-Agents-Beispiel, aktualisierter Aufbau der Szene mit Multi-Agent-Setup

5.3.3 Training

Das Training wurde mit PPO und den Default-Hyperparametern durchgeführt, die um grob geschätzte Parameter erweitert wurden, die in der folgenden Auflistung zu finden sind. Die Funktionsweisen der einzelnen Hyperparameter werden im Abschnitt 7.4 ausführlich erläutert.

- epsilon: 0.2, lambda: 0.95, beta: 3e-3, learning_rate: 1e-3,
- num_epoch: 3, max_steps: 2e6
- use_recurrent: false
- batch_size: 1024, buffer_size: 4096, time_horizon: 32
- hidden_units: 64, num_layers: 2
- reward_signals: extrinsic, strength: 1.0, gamma: 0.99

Bei einem maximalen mathematisch möglichen Reward von $r = 1.0$ ergab sich bereits nach ca. 30.000 Schritten eine durchschnittliche Belohnung von $r = 0.94$ pro Episode. Nach etwa 45.000 Schritten wurde ein Wert von $r = 0.98$ erreicht und somit eine schnelle Konvergenz erreicht. Obwohl die Hyperparameter nur grob festgelegt wurden, wurde für diese einfache Aufgabe also nach kurzer Zeit ein gutes Ergebnis erzielt. Bezogen auf die kumulative Belohnung hat sich folgender Trainingsverlauf ergeben:

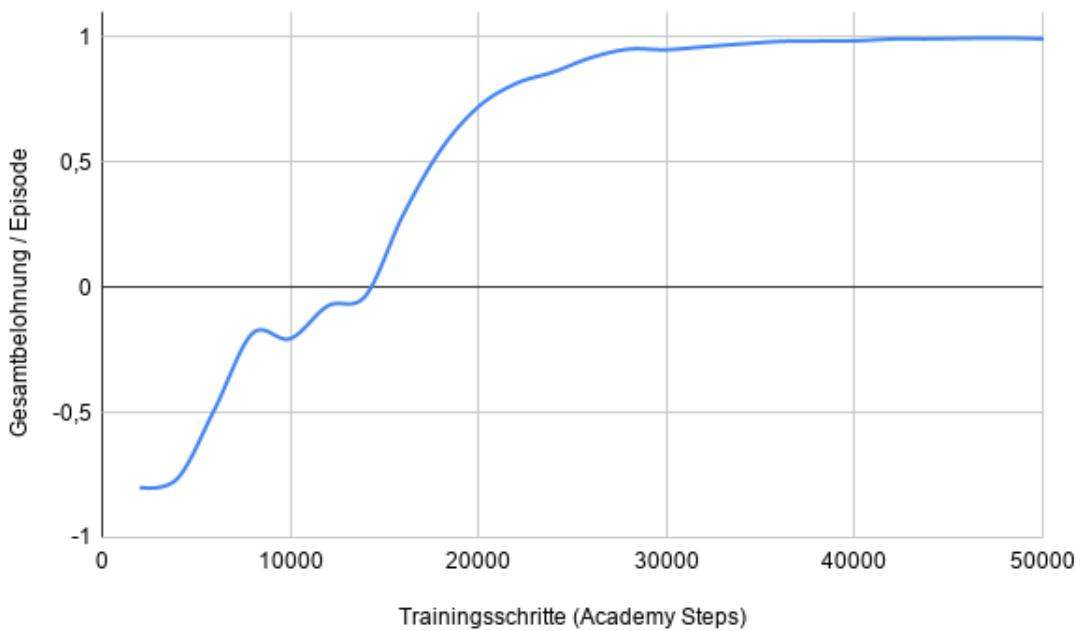


Abbildung 5.8.: ML-Agents-Beispiel, kumulative Belohnung des Trainingsverlaufs

In diesem Kapitel wurde bewiesen, dass einfache Vorgänge innerhalb kürzester Zeit gelernt werden können und dabei hohe Erfolgsquoten erreicht werden. Im Folgenden wird auf dieser Erkenntnis aufgebaut und mit komplexeren Sachverhalten konfrontiert.

6

Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Dieses Kapitel bildet den Hauptteil der Thesis und stellt den entwickelten Ansatz einer Dorfsimulation vor. Im Detail wird die Entwicklung eines Verfahrens zur Erschaffung glaubwürdiger Agenten mithilfe von Machine Learning als Alternative zur Nutzung klassischer Methoden wie Decision Trees oder Finite State Machines vorgestellt. Dazu gehört die Entwicklung eines Prototypen für ein in Unity entwickeltes Dorfssystem, in dem sich die Agenten aufhalten. Konkret werden dafür bedürfnisbasierte Utility-Agenten verwendet, die durch Reinforcement Learning ein authentisches Verhalten erlernen sollen. Für die Umsetzung wurde *Unity Pro* in der Version *2019.2.8f1* verwendet. Sämtlicher Code wurde in *Visual Studio* erstellt. Als Basis-Framework dient ML-Agents (siehe Kapitel 5). Für die Nutzung von Multithreading wurde weiterhin die Bibliothek *Thread Ninja* verwendet [Spi14]. Zunächst werden Anforderungen an die Implementierung gestellt.

6.1 Anforderungen

Es soll eine Dorfsimulation angefertigt werden, in der es Gebäude mit unterschiedlichen Zwecken gibt (beispielsweise eine Taverne, einen Marktplatz, einen Bäcker, etc.). Weiterhin soll es intelligente Bewohner geben, die in und außerhalb von Gebäuden Tätigkeiten durchführen können. Folgende Ziele sollen bezüglich des Dorfes erfüllt werden:

- **Dorf:** Es soll ein Dorfssystem entwickelt werden, in dem begehbarer Gebäude mit bestimmter Kapazität und unterschiedlichen Zwecken vorhanden sind (z.B. Bäcker, Taverne, Wohnhäuser, Marktplatz, etc.). Die Simulation soll einen zeitlichen Tagesablauf mit visuellen Hinweisen wie Belichtung, Sonnenverlauf und Lichtquellen haben.
- **NPCs:** Das Dorf soll von NPCs bevölkert werden, die verschiedene Aktionen ausführen können. Dazu gehören Aktionen wie beispielsweise Arbeiten, Essen oder Schlafen. Jede Aktion hat ein oder mehrere Ziele, die einem Gebäudetypen entsprechen. Die Agenten müssen in der Lage sein, zu den Zielen zu navigieren. Die Aktion *Schlafen* hätte beispielsweise das Wohnhaus des Agenten zum Ziel, die Aktion *zur Kirche gehen* dagegen die Kirche. Aktionen können weiterhin Ressourcen produzieren oder verbrauchen.
- **Bedürfnisse:** Die NPCs verfügen über Bedürfniswerte, die für die Auswahl von Aktionen relevant sind. Dazu gehören menschliche Bedürfnisse wie Hunger, Schlaf, Kommunikation, Bildung, etc.
- **Authentisches Verhalten:** Die NPCs sollen selbstständig ein authentisches Verhalten erlernen, das vom internen Zustand der Agenten abhängt. Die Bedürfniswerte bilden die Grundlage für die Auswahl von Aktionen. Die Bedürfnisse sollen so angepasst werden, dass sie in etwa den menschlichen Äquivalenten entsprechen. Wenn ein Agent hungrig ist, soll er automatisch die Aktion *Essen* wählen. Idealerweise soll ein sinnvoller Tagesablauf abgebildet werden.

6.1.1 Trennung von Modell, Darstellung und Logik

Klassischerweise wird in der Softwareentwicklung versucht, die Darstellungsebene von der Logikebene und der Modell- bzw. Datenebene zu trennen. Häufig wird dafür das *Model-View-Controller* (MVC)-Pattern verwendet. Dieses Vorgehen ist grundsätzlich auch in Unity anwendbar. Aufgrund der grafischen Natur einer Game Engine sowie dem grundlegenden Konzept der Verwendung von *GameObjects* (siehe Kapitel 5) findet allerdings eine größere Vermischung von Logik und Darstellung statt. In der entwickelten Anwendung werden keine Pattern wie MVC verwendet, da hierdurch ein zu großer Overhead entstehen würde. Dieser Aufwand wäre für die Implementierung der Dorfsimulation nicht gerechtfertigt. Das Hauptziel ist nicht die Erschaffung einer gut erweiterbaren Anwendung, sondern eine Demonstration für die Möglichkeiten von Reinforcement Learning und bedürfnisbasierten Agenten. Eine Vermengung der Darstellungsebene, Logikebene und Modellebene wird akzeptiert. Es gibt daher Klassen, in denen keine klare Trennung zwischen Visualisierung und Verhalten vorgenommen wird.

6.2 Modellierung

An die in der Stadt verwendeten Gebäude wird die Anforderung gestellt, dass sie begehbar sein müssen. Da keine fertigen Assets für begehbarer Gebäude zur Verfügung standen, wurde für die Modellierung des Dorfes eine externe Sammlung von modularen Assets aus dem Unity Asset Store bezogen [UNTN19]. Diese beinhaltet Texturen, Oberflächenmaterialien, Polygonnetze und Unity-Prefabs. Mithilfe dieser Sammlung wurden eigene Gebäude aus Einzelkomponenten wie Fenstern, Wandelementen und Dekorationen angefertigt. Der Großteil der Gebäude wurde also manuell modelliert. Dadurch wird sowohl die Begehbarkeit der Gebäude sichergestellt, als auch eine höhere Modularität sowie einer Verbesserung der visuellen Attraktivität des Dorfes gewährleistet.



Abbildung 6.1.: Beispiele für modulare Komponenten zur Modellierung des Dorfes

Abbildung 6.1 zeigt einige Beispiele für modulare Komponenten. Weiterhin ist in Abbildung 6.2 ein beispielhaftes Haus zu sehen, das für den Bau des Dorfes aus diesen Komponenten gebildet wurde.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

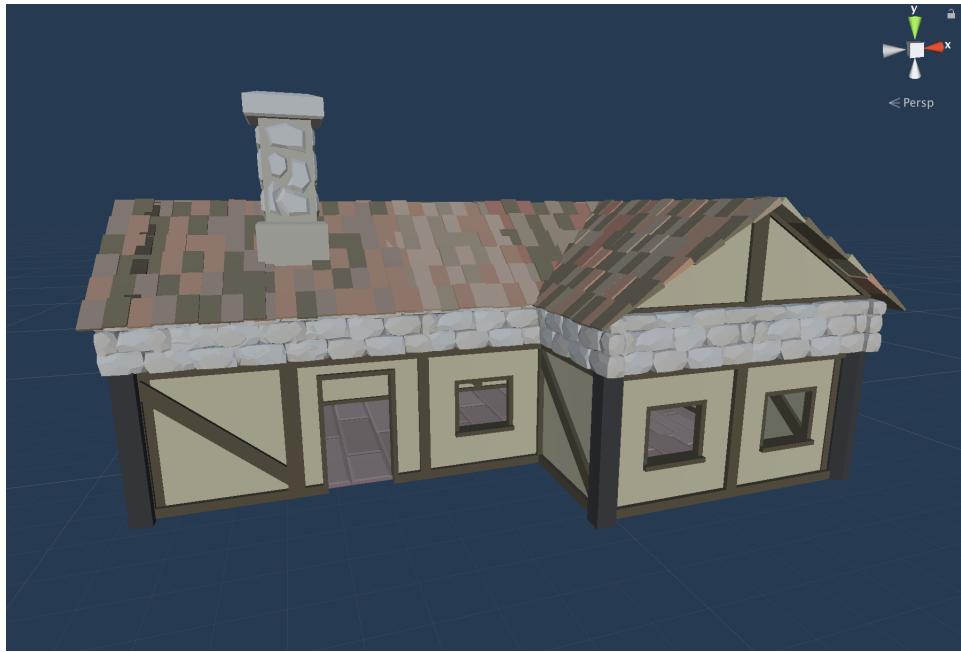


Abbildung 6.2.: Modelliertes modulares Haus

Für Dekorationen, Bäume und Weiteres wurden zusätzlich weitere Sammlungen von Assets verwendet [Stu13] [Dar17] [Pol17] [WC18].

Die Abbildungen 6.3 und 6.4 zeigen die modellierte Stadt mit einer Vielzahl von Gebäuden und Dekorationen wie Bäumen und Zäunen. Auf den Abbildungen sind sowohl eine Außenansicht eines großen Teils der Stadt als auch eine beispielhafte Innenansicht einer Straße zu erkennen. Um eine visuelle Abgrenzung zur Außenwelt vorzunehmen, wurden Felsen um die Stadt herum positioniert, die nicht passierbar sind.



Abbildung 6.3.: Modellierte Stadt, äußere Ansicht

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning



Abbildung 6.4.: Modellierte Stadt, innere Ansicht

6.3 Softwaredesign- und Architektur

Im Verlauf dieses Abschnitts wird die Architektur der Simulation erläutert. Zunächst wird hierfür ein genereller Überblick vermittelt, der die grundsätzliche Struktur beschreibt. Anschließend werden die einzelnen Komponenten anhand der zentralen Klassen näher betrachtet und ihre Funktionsweise erläutert. Generell wurde ein objektorientierter Ansatz gewählt. Im Folgenden wird die in Abbildung 6.5 verwendete Notation verwendet, um den Aufbau von Klassen anhand der Signaturen ihrer Methoden und Felder zu beschreiben.

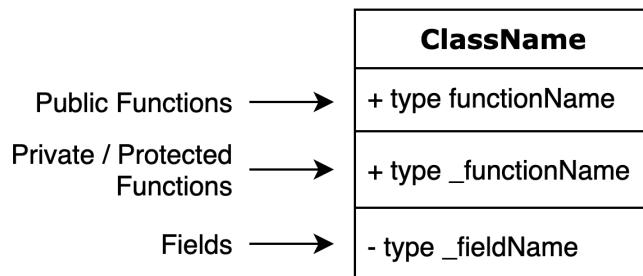


Abbildung 6.5.: Verwendete Notation für die Beschreibung von Klassen

Da das ML-Agents Toolkit verwendet wird, hängt die grundlegende Architektur des Dorfes teilweise von der im Kapitel 5 erläuterten Struktur ab (siehe 5.3). Insbesondere betrifft das die Entwicklung der Agenten und Entscheidungen sowie die Ausführung von Aktionen. Daher wird der in Kapitel 5 vorgestellte Prozess verwendet, der eine `Academy`-Instanz mit mehreren Agenten und bestimmten benötigten Funktionen vorsieht.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

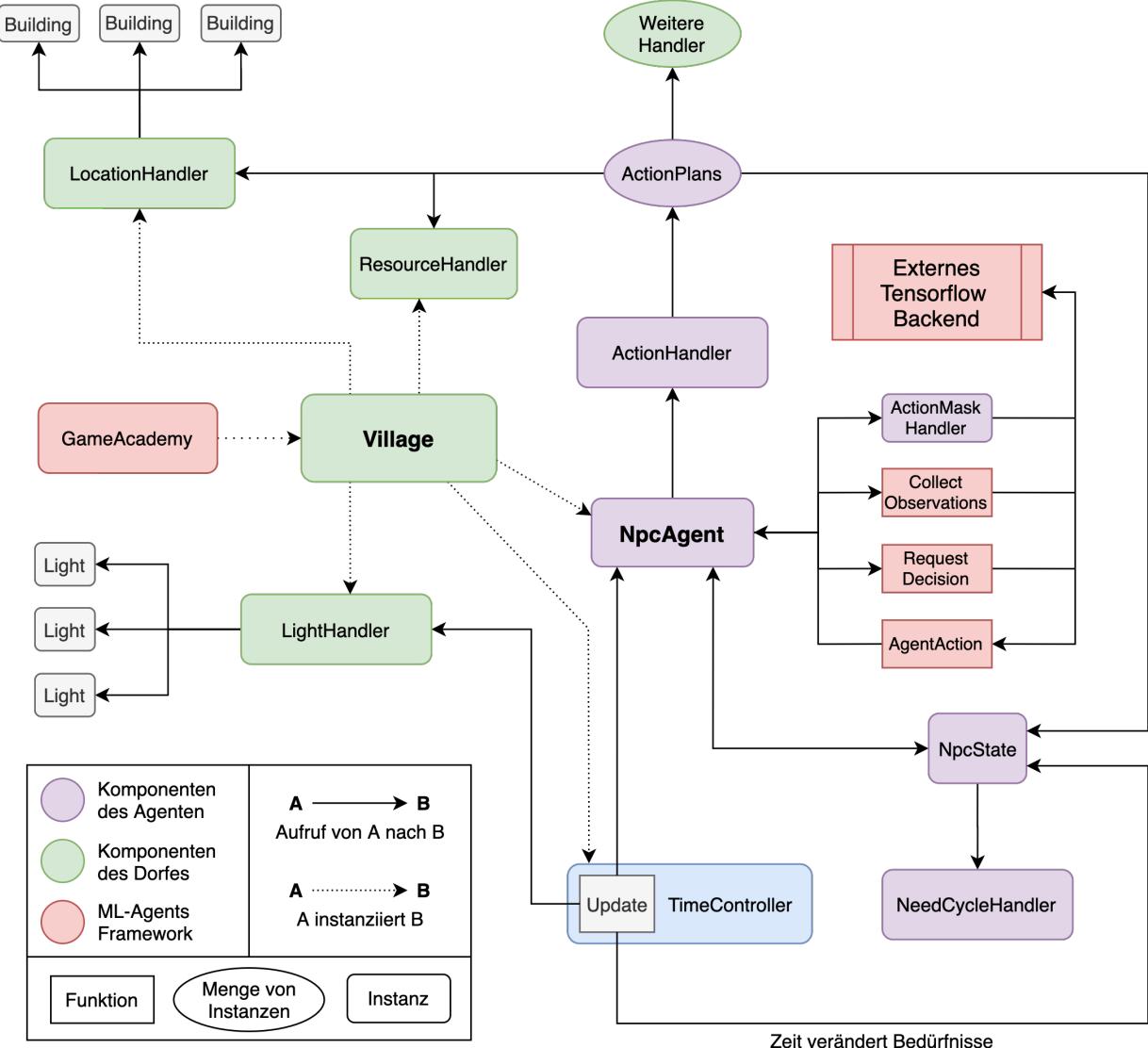


Abbildung 6.6.: Grundsätzliche Architektur des Dorfsystems

Abbildung 6.6 zeigt den grundsätzlichen Aufbau der Implementierung. Die beiden Komponenten **Village** und **NpcAgent** bilden die zentralen Elemente für den Ausführungskreislauf bzw. *Game Loop* der Simulation. Es besteht eine direkte Anbindung zum ML-Agents Toolkit über die **GameAcademy**. Zunächst wird **Village** durch die **GameAcademy** initialisiert. **Village** wiederum erzeugt eine festgelegte Anzahl von Agenten. Es gibt also eine kaskadierte Hierarchie für die Initialisierung der Simulation mit der **GameAcademy** an höchster Stelle, gefolgt von mehreren Instanzen vom Typ **Village** und schließlich jeweils mehreren Instanzen vom Typ **NpcAgent**.

Village verfügt über verschiedene Subkomponenten, die für das Steuern bestimmter Aufgaben verantwortlich sind. Über den **ResourceHandler** wird das Ressourcensystem verwaltet, auf das die NPCs zugreifen können. Zusätzlich verwaltet der **LocationHandler** alle Orte im Spiel, zu denen ein Agent navigieren kann. Dazu gehören sämtliche Häuser inklusive deren Räume und weitere Orte außerhalb von Gebäuden. Zuletzt ist der **LightHandler** dafür zuständig, die Tageszeit anhand des Sonnenstandes und der Kontrolle von Lichtquellen zu simulieren. Der **NpcAgent** erbt von der ML-Agents-Klasse **Agent** und ist somit für das Reinforcement Learning und die Ausführung von Aktionen vorhanden. Das Framework wählt Aktionen für den Agenten aus und stößt diese an. Die Aktionen, die ein Agent ausführen kann, benötigen immer ein Ziel und verwenden daher den **LocationHandler**.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Sie beeinflussen außerdem den Zustand `NpcState` eines Agenten. Dieser beinhaltet beispielsweise die Bedürfnisse eines Agenten, die von Aktionen aufgefüllt werden können. Die Funktionen, die die Abnahme der Bedürfnisse bestimmen, werden im `NeedCycleHandler` definiert und werden über diesen aktualisiert. Aktionen können außerdem Ressourcen aufbrauchen oder produzieren, indem der `ResourceHandler` verwendet wird.

Eine zentrale Zeitverwaltungseinheit wird durch den `TimeController` realisiert. Dieser berechnet in regelmäßigen Abständen aufgrund der verstrichenen Zeit die neue Zeit im Spiel. Anschließend werden alle Komponenten, für die das Voranschreiten der Zeit relevant ist, mit dieser neuen Zeit versorgt. Der `LightHandler` regelt mit dessen Hilfe beispielsweise die Intensität und Position der Sonne wohingegen der `NeedCycleHandler` aufgrund der verstrichenen Zeit die neuen Bedürfnisse des Agenten berechnet und den `NpcState` entsprechend anpasst.

6.4 Implementierung

In den folgenden Abschnitten werden die wichtigsten Klassen der Dorfsimulation im Einzelnen erläutert. Zunächst werden dafür die zentralen, übergeordneten Komponenten `Village` und `GameAcademy` näher betrachtet.

6.4.1 Village

Die Aufgabe der Komponente `Village` ist die Erzeugung von NPCs und die Bereitstellung einer zentralen Verwaltungseinheit für Komponenten. Abbildung 6.7 zeigt den Aufbau der Klasse `Village` anhand ihrer Funktionen:

Village
+ void initVillage
+ void updateTime
+ void resetNpcJobs
+ void _spawnVillageAgent
+ void _spawnNpcs
+ IEnumerator _initNpcs
+ void _initNpcJobs
+ IEnumerator CheckTasksCompleted
+ void _initVillageInfo
+ IEnumerator _initAgent

Abbildung 6.7.: Methoden der Klasse `Village`

`Village` erzeugt und initialisiert zunächst Handler-Klassen wie `TimeController`, `LocationHandler`, `ResourceHandler` und `LightHandler` in der Funktion `initVillage`. Anschließend werden weitere Funktionen aufgerufen, um innerhalb von Coroutinen asynchron Agenten zu erzeugen. Diese werden in `_spawnNpcs` zunächst erstellt und dann in `_initNpcs` initialisiert, wie in Listing 6.1 zu sehen.

```

1  private IEnumerator _initNpcs() {
2      _npcLoadingTasks = new Dictionary<int, bool>();
3      for (int i = 0; i < _villageNpcs.Count; i++) {
4          _npcLoadingTasks[i] = false;
5          _villageNpcs[i].StartCoroutine(_initAgent(i));
6      }
7      yield return StartCoroutine(CheckTasksCompleted());
8      _villageAgent.init(this);
9      GameAcademy.NPC_LOADING_DONE = true;
10 }

```

Listing 6.1: Initialisierung von Agenten

Für jeden Agenten wird eine einzelne Coroutine `_initAgent` gestartet. Anschließend wird mit `yield return` auf die Ausführung einer weiteren Coroutine `CheckTasksCompleted` gewartet, die in bestimmten Abständen prüft, ob die weiteren Coroutinen beendet wurden.

Mit der Funktion `_spawnVillageAgent` wird darüber hinaus die Klasse `VillageAgent` initialisiert (siehe Abschnitt 6.4.7). Sobald alle `NpcAgents` erfolgreich initialisiert wurden, wird auch der `VillageAgent` initialisiert.

`Village` speichert Referenzen auf alle Agenten `NpcAgent` des Dorfes und bietet eine Schnittstelle für weitere Komponenten, um Zugriff auf die Agenten zu erlangen. Zusätzlich verfügt das Dorf über eine Instanz der Klasse `VillageInfo`, die in der Methode `_initVillageInfo` erzeugt wird. Darin werden Informationen über den aktuellen Zustand des Dorfes gespeichert. Dazu gehört beispielsweise die Anzahl der Agenten, die aktuell eine bestimmte Aktion ausführen. Diese Daten werden anschließend im UI der Simulation dargestellt. Aus diesem Grund verfügt `VillageInfo` über eine Vielzahl von Tracking-Funktionen, die diese Statistiken verwalten.

6.4.2 GameAcademy

Die `GameAcademy` erbt von der ML-Agents Basisklasse `Academy` (siehe Kapitel 5) und erweitert diese um verschiedene Einstellungsmöglichkeiten sowie dem Ladevorgang des Dorfes. In Abbildung 6.8 wird der Aufbau der Klasse verdeutlicht:

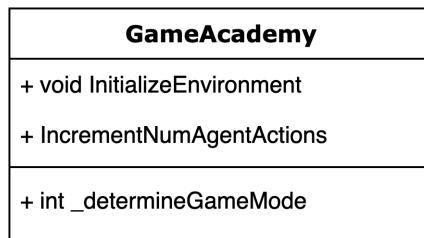


Abbildung 6.8.: Methoden der Klasse `GameAcademy`

Zunächst wird die Funktion `InitializeEnvironment` durch `ML-Agents` aufgerufen. Hier werden zunächst einige Parameter ausgewertet, die über den Unity Editor gesetzt werden können. Neben den Parametern der Oberklasse `Academy` können hier für die Simulation grundlegende Einstellungen vorgenommen werden, wie die Anzahl der zu erzeugenden NPCs oder die Auswahl des Inference-Modus zwischen den unterschiedlichen implementierten Verfahren wie beispielsweise Reinforcement Learning oder GOAP. Für jeden dieser Parameter existiert eine statische Variable, die von allen verwendeten Komponenten innerhalb des Dorfes einfach gelesen werden kann. Aufgrund dieser Parameter sowie

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

dem Rückgabewert von `GetIsInference` wird in der Funktion `_determineGameMode` anschließend eine weitere Variable gesetzt, die den Spielmodus festlegt. Bestimmte Komponenten verhalten sich je nach Spielmodus auf eine andere Art und Weise (siehe Abschnitt 6.4.10).

Alle Komponente vom Typ `Village` werden anschließend durch die Anweisung `FindObjectsofType<Village>()` hinzugefügt. Die `GameAcademy` startet dann den Ladevorgang der Simulation, indem alle Instanzen von `Village` initialisiert werden. Mehrere Dörfer mit jeweils einem `Village`-Modul können gleichzeitig verwendet werden, indem das Prefab für das Spielfeld dupliziert wird.

Die Funktion `IncrementNumAgentActions` ist ausschließlich für den `VillageAgent` relevant und wird im Abschnitt 6.4.7 behandelt.

Die in Tabelle 6.1 zu sehenden Parameter können in der `GameAcademy` angepasst werden. Die Tabelle enthält für jeden Parameter eine kurze Beschreibung, die aus den entsprechenden Kommentaren im Programmcode entnommen wurden.

Parameter	Beschreibung
<code>numAgentsPerVillage25</code>	Define how many agents should be spawned per village. This should be in order of 25s (25, 50, 75...), for the jobs distribution to work properly.
<code>actionsPerAgentUntilReset</code>	Define how many decisions the agents should perform until they are reset.
<code>debugMode</code>	Tracks some additional data useful for debugging when activated.
<code>useFakeInference</code>	Since high timescale levels do not scale well with NavMeshAgents, a kind of fake inference will have to be used, in which agents teleport to their destinations and additionally wait for an amount of time that is equal to the time it would have taken to move there.
<code>verbose</code>	Prints some additional information when activated.
<code>actionsUntilVillageAction</code>	Decides how many steps will be performed per agent between <code>VillageAgent</code> decisions.
<code>villageDecisionsReset</code>	Decides how many decisions will be made by the <code>VillageAgent</code> until Done is called.
<code>useManualNpcDecisions</code>	Uses manual decisions instead of the trained model.
<code>ignoreVillageAgent</code>	Ignores the village agent and sticks with the initial distribution of jobs.
<code>useGoap</code>	Uses GOAP agents instead of a trained model.
<code>agentStoppingDistance</code>	Determines how far from his target an agent will stop when moving to a destination.
<code>agentRythmVariation</code>	Decides the amount of variation between the day/night cycles of the agents.
<code>logActionDurations</code>	Determines whether the duration of actions should be tracked in <code>VillageInfo</code> .
<code>logRewards</code>	Determines whether the rewards should be tracked for each action to gain some statistical information.
<code>logNeeds</code>	Logs the average needs of NPCs at certain times for later analysis.
<code>logAttributes</code>	Logs the average attributes of agents for each action.
<code>logActionMaskAmounts</code>	Determines whether to log the amount of actions that are masked respectively.
<code>ignoreAllResources</code>	Completely ignores all resource handling when set to true.
<code>ignoreJobMask</code>	Ignores the job that has been designated to an agent and instead doesn't mask jobs.

Tabelle 6.1.: Konfigurierbare Parameter der `GameAcademy`

6.4.2.1 Nutzung mehrerer Villages

Die Simulation ist nicht auf die Nutzung eines einzelnen Dorfes limitiert. Tatsächlich kann die Verwendung mehrerer Dörfer für den Trainingsvorgang vorteilhaft sein. Aus diesem Grund wurden sämtliche Komponenten so entwickelt, dass mehrere Dörfer reibungsfrei parallel betrieben werden. Zwar beeinflusst dies massiv die Performance des Testsystems, jedoch können dadurch deutlich mehr Agenten verwendet werden - bei der Verwendung eines einzelnen Dorfes würden die Kapazitäten der Gebäude sonst schnell überschritten werden. Weiterhin können dadurch Erfahrungen gesammelt werden, die diverser sind, da die Zustände der Umgebung in unterschiedlichen Dörfern voneinander abweichen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Zur Verwendung mehrerer Dörfer muss das Prefab des Spielfeldes, das unter anderem das gesamte Mesh beinhaltet, dupliziert werden. Es ist wichtig, dass die Position des Prefabs verändert wird, damit keine Überschneidungen entstehen. Weiterhin sollten alle vorberechneten Daten wie Lichtinformationen oder das NavMesh (siehe Abschnitt 6.4.6.4) erneut berechnet werden. Die **GameAcademy** ist unter anderem für die Initialisierung der Dörfer zuständig. Sie sucht nach allen **Village**-Komponenten in der gesamten Spielszene. Wie in Abbildung 6.9 erkennbar, werden die Dörfer einzeln und unabhängig voneinander geladen. Die Pfeile in der Grafik geben an, welche Klassen durch welche Komponenten instanziert werden. Das gesamte Ökosystem eines Dorfes, inklusive der Ausführung von Aktionen, verläuft vollkommen unabhängig. Jedes Dorf stellt eine in sich gekapselte Einheit dar, die mit der Außenwelt innerhalb der Simulation, also mit anderen Dörfern, nicht kommuniziert. Jedes Dorf verfügt also über eigene Komponenten wie **ResourceHandler**, **LocationHandler** oder **Village**.

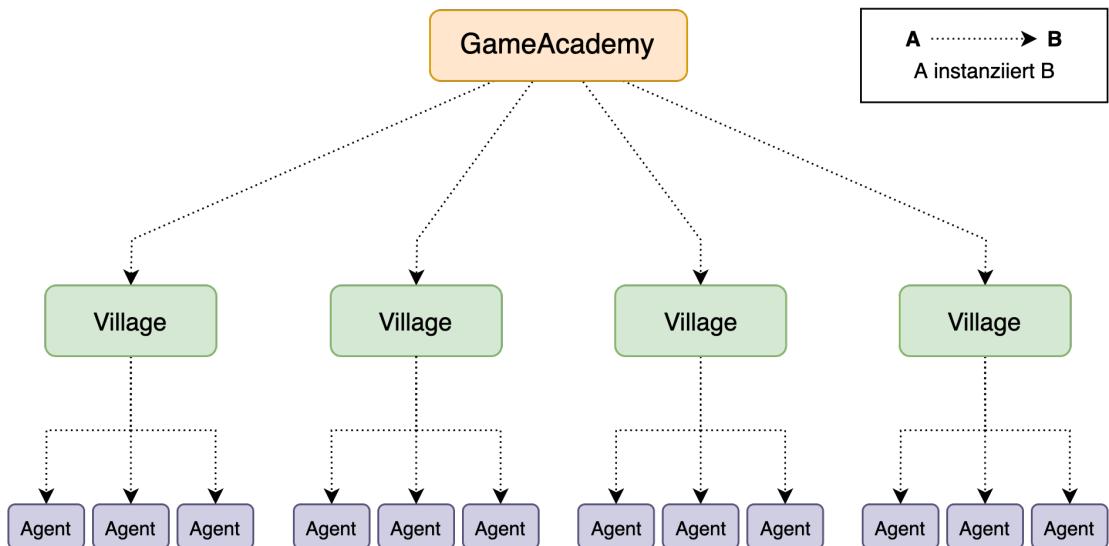


Abbildung 6.9.: Architektur, Ladevorgang mehrerer Dörfer

6.4.3 LocationHandler

Die Klasse **LocationHandler** kennt alle im Spiel verfügbaren Gebäude und Orte und bietet allen Agenten eine Schnittstelle, um mit diesen zu interagieren. Er ermöglicht somit neben der benötigten Navigation ebenfalls das Abfragen von Informationen zu bestimmten Gebäudetypen. Die Gebäude haben jeweils eine begrenzte Kapazität, es kann sich also nur eine bestimmte Anzahl von NPCs gleichzeitig in einem Haus befinden. Daher kümmert der **LocationHandler** sich weiterhin um die Verwaltung dieser verfügbaren **Slots**. Der **LocationHandler** wird ausdrücklich nicht für die eigentliche Navigation der Agenten verwendet, stattdessen werden lediglich Orte verwaltet und Positionen zurückgegeben. Der **LocationHandler** wird durch die Klasse **Village** erzeugt. Abbildung 6.10 beschreibt den Aufbau.

LocationHandler
+ static bool isValidLocation + void init + int getNumSlotsForLocations + void vacateSlot + Transform getCalculatedPosition + IEnumerator occupySlotInFirstBuilding + IEnumerator occupySlotInRandomBuilding + IEnumerator occupySlotInClosestBuilding + IEnumerator occupySlotInNonEmptyBuilding + IEnumerator hasFreeSpots + IEnumerator calculatePathLength + float getCalculatedPathLength + IEnumerator _setupLocations + void _setupAgentHomeLocations

Abbildung 6.10.: Methoden der Klasse LocationHandler

Nach der Initialisierung durch `init` wird zunächst allen Agenten in der Funktion `_setupAgentHomeLocations` ein Heim zugewiesen, in dem sie wohnen. In der Funktion `_setupLocations` werden weiterhin alle Gebäude abhängig von ihrem Typ in einem Assoziativen Array gespeichert. In C# wird dafür typischerweise ein `Dictionary` verwendet, daher ein Feld vom Typen `Dictionary<string, List<Building>>` genutzt. Die verschiedenen Gebäudetypen werden in der Klasse `LocationTypes` definiert, beispielsweise `FIELD_GRAIN`, `BAKERY` oder `SCHOOL`. Diese werden als Schlüssel verwendet. Die Werte des `Dictionary` enthalten Listen von Gebäuden (`List<Building>`). Wenn der `LocationHandler` initialisiert wird, werden alle Gebäude des Dorfes gesucht und in die jeweilige Kategorie sortiert. Diese Suche wird beschleunigt, indem alle Gebäude mit einem Tag versehen werden. Jedes Gebäude und jeder navigierbare Ort in der Stadt verfügt über die Komponente `Building`. In jedem Gebäude-Prefab werden sogenannte `BuildingNavPoints` definiert, welche die tatsächlichen Orte bzw. *Slots* darstellen, zu denen die Agenten navigieren können. Dafür werden Objekte erstellt, die eine Position innerhalb des Gebäudes zugewiesen bekommen. Diesen wird die Komponente `BuildingNavPoint` angefügt. Die Menge der `BuildingNavPoint` in einem Gebäude bestimmt die Kapazität des Gebäudes. `BuildingNavPoint` können von nur einem NPC zur Zeit durch den Aufruf der Funktion `occupy` besetzt werden und müssen, sobald der Agent den Ort verlässt, anschließend durch die Funktion `vacate` wieder geräumt werden. Zusätzlich wird ein `DistanceNavPoint` definiert. Dieser wird verwendet, um die Distanz des Agenten zu einem bestimmten Gebäude zu ermitteln und befindet sich daher im Eingang des Gebäudes. Falls es mehrere Eingänge gibt, wird die Mitte des Hauses gewählt. Die Abbildung 6.11 zeigt beispielhaft, wie `BuildingNavPoint` und der `BuildingDistanceNavPoint` in einem beispielhaften Gebäude definiert werden. Zur besseren Darstellung wurde jedem `BuildingNavPoint` ein Mesh in Form von einer Kugel zugewiesen, die normalerweise deaktiviert ist. Für die Platzierung der Objekte kann dieses Mesh aktiviert werden.

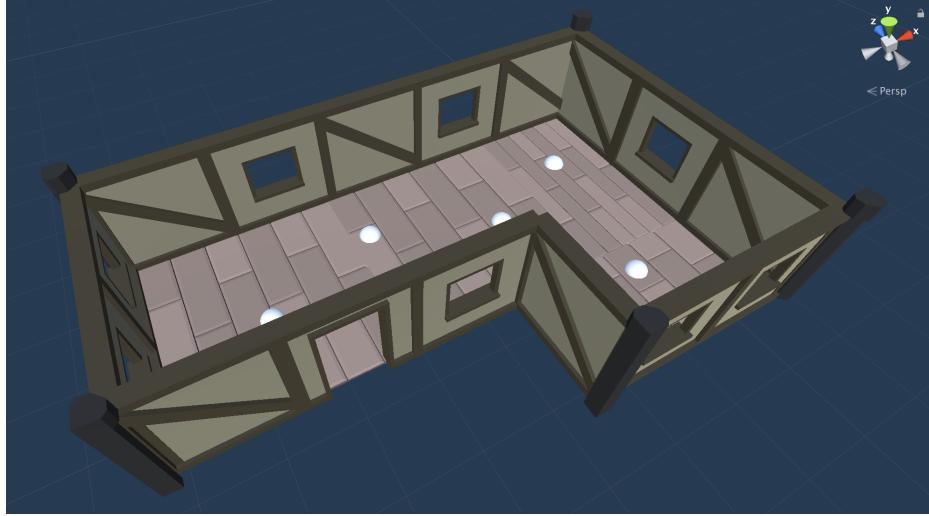


Abbildung 6.11.: Definition von `BuildingNavPoint` und `BuildingDistanceNavPoint`

`Building` enthält alle `BuildingNavPoint` des Gebäudes und verwaltet diese. Es werden Referenzen auf die bereits besetzten Slots in Gebäuden sowie den jeweiligen NPCs gehalten. Weiterhin ist für jeden Typ aus `LocationTypes` bekannt, wie viele offene Slots noch vorhanden sind. Der `LocationHandler` bietet nun eine Menge von Funktionen. Diese stellen allesamt NPCs Orte zur Verfügung, unterscheiden sich jedoch leicht in ihrer Semantik:

- `occupySlotInFoundBuilding` findet das erste Gebäude eines bestimmten Typs, das einen freien Slot hat und gibt die Position des ersten Slots in dem Gebäude zurück.
- `occupySlotInRandomBuilding` wird genutzt, um ein zufälliges Gebäude mit einem zufälligen Slot auszuwählen.
- `occupySlotInClosestBuilding` soll das nicht besetzte Gebäude finden, das sich von der Position des Agenten örtlich am wenigstens weit weg entfernt befindet.
- `occupySlotInNonEmptyBuilding` findet das erste Gebäude, das bereits mit mindestens einem Agenten besetzt ist. Falls kein Gebäude gefunden wird, wird eine zufällige Position zurückgegeben. Die Funktion wird unter anderem verwendet um zu simulieren, dass sich mehrere Agenten auf einem Marktplatz treffen, um miteinander zu kommunizieren. In diesem Fall wird stets ein Ort zurückgegeben, der bereits von einem NPC beansprucht wird, sodass diese sich „treffen“ können und Gruppen entstehen.

Die Funktion `occupyFreeSlotInRandomBuilding` wird beispielhaft in Listing 6.2 aufgeführt. Zunächst wird für alle Gebäude des Typs geprüft, ob sie über freie Plätze verfügen. Für diese Gebäude wird dessen Index in einer Liste gespeichert. Aus dieser Liste wird nun ein zufälliger Index `randomIndex` ausgewählt, der bestimmt, zu welchem Ort der Agent navigieren wird. In diesem Gebäude `chosenBuilding` wird ein Slot für den Agenten reserviert. Anschließend wird die Position des Slots, der einen `BuildingNavPoint` darstellt, anhand eines `Transform` in `_calculatedPositions` hinterlegt. Später wird der Agent dieses Feld nutzen, um die Position abzufragen. Das ist nötig, weil die Funktion durch eine asynchrone Coroutine gestartet wird und diese deshalb keinen anderen Typen als `IEnumerator` zurückgeben kann (siehe Kapitel 5). Coroutinen werden bei der Gebäudesuche verwendet, um durch Threading eine höhere Leistung zu erreichen.

```

1  public IEnumerator occupyFreeSlotInRandomBuilding(string locationType
2      , int agentId, bool forceRandomSlot = false, bool getPositionOnly
3      = false) {
4      _calculatedPositions[agentId] = null;
5      List<Building> locationTypeBuildings = _locations[locationType];
6      List<int> freeBuildingIndices = new List<int>();
7      int len = locationTypeBuildings.Count;
8      for (int i = 0; i < len; i++) {
9          if (!locationTypeBuildings[i].IsBuildingOccupied)
10             freeBuildingIndices.Add(i);
11     }
12     if (freeBuildingIndices.Count == 0) {
13         Debug.LogWarning("Attempted to occupy a free building for
14             locationtype " + locationType + ", but there is none.");
15         yield break;
16     }
17     int randomIndex = freeBuildingIndices[Mathf.RoundToInt(
18         UnityEngine.Random.value * (freeBuildingIndices.Count - 1))];
19     Building chosenBuilding = locationTypeBuildings[randomIndex];
20     _agentsInBuildings[agentId] = chosenBuilding;
21
22     Transform slotLocation = forceRandomSlot ? chosenBuilding.
23         occupyRandomSlot(agentId) : chosenBuilding.occupySlot(agentId)
24     ;
25     if (slotLocation == null) {
26         Debug.LogWarning("Failed to occupy free slot in random
27             building for type " + locationType + ".");
28         yield break;
29     }
30     _availableLocationTypeSlots[locationType]--;
31     _calculatedPositions[agentId] = slotLocation;
32     yield break;
33 }
```

Listing 6.2: Auswahl von zufälligen Gebäuden in der Klasse `LocationHandler`

Vor einer Zuweisung eines Agenten zu einem Slot kann die Funktion `hasFreeSpots` von außerhalb verwendet werden, um zu prüfen, ob Gebäude mit freien Plätzen existieren. Die Funktion prüft das Feld `private Dictionary<string, int> _availableLocationTypeSlots`. Initial beschreibt das Feld die Gesamtanzahl der Slots, die für einen Gebäudetypen verfügbar sind. Jedes Mal wenn ein Agent einem Slot zugewiesen oder von ihm getrennt wird, wird die hinterlegte Anzahl für den Gebäudetypen angepasst.

6.4.4 ResourceHandler

Es kommt ein Ressourcensystem zum Einsatz, bei dem die Klasse `ResourceHandler` als zentrale Verwaltungsstelle für alle Ressourcen eines Dorfes agiert. Zu keiner Zeit verfügt ein Agent allein über eine Ressource, da der `ResourceHandler` alle im Dorf vorhandenen Ressourcen kennt und verwaltet. Aktionen können Ressourcen verbrauchen und Ressourcen produzieren. Daher nutzen besonders die Aktionen, die mit Berufen verbunden sind, das Ressourcensystem. Es sollen Abhängigkeiten von den NPCs untereinander geschaffen werden, da diese auf Ressourcen angewiesen sind, die von anderen NPCs produziert werden. Den Aktionen wird damit ein Sinn verliehen, der über die Bedürfnisbefriedigung und den visuellen Aspekt hinausgeht. Die Agenten sollen lernen, mit Ressourcen umzugehen. Die Funktionen der Klasse sind in [6.12](#) zu erkennen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

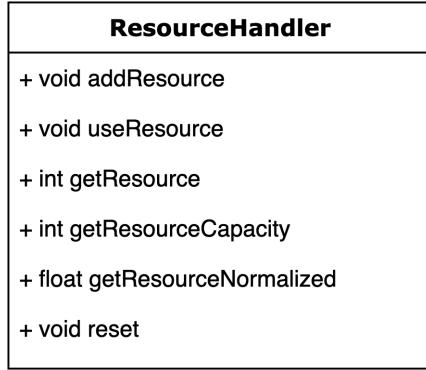


Abbildung 6.12.: Methoden der Klasse **ResourceHandler**

Eine Instanz des **ResourceHandlers** wird in der Klasse **Village** erzeugt und schließlich allen Agenten zugänglich gemacht. In der Funktion **reset** werden zunächst Obergrenzen für die Kapazitäten der einzelnen Ressourcen aufgrund der Anzahl der Agenten des Dorfes definiert. Je mehr Agenten vorhanden sind, desto höher ist die Kapazitätsgrenze, da sich die Menge der produzierten Ressourcen pro Aktion sowie die Menge der pro Agent verbrauchten Ressourcen nicht verändert. Weiterhin werden die Ressourcen zufällig initialisiert, werden aber durch Maximal- und Minimalwerte eingeschränkt. Die Funktion **reset** wird außerdem während des Trainings nach einer bestimmten Zeit aufgerufen, um die Ressourcen zurückzusetzen. Die verschiedenen Typen von Ressourcen werden ebenfalls im **ResourceHandler** definiert. Über die Funktionen **addResource** und **useResource** können Agenten mit Ressourcen interagieren. Weiterhin besteht die Möglichkeit, in Abhängigkeit von der Kapazität einer Ressource durch die Funktion **getResourceNormalized** eine normalisierte Anzahl für eine Ressource zu erhalten. Der Wertebereich dieser Anzahl wird durch den Intervall $[0, 1]$ definiert. Diese Normierung ist für die Observations wichtig, die idealerweise in dem genannten Intervall liegen sollten. Bei der Initialisierung der Simulation werden zufällige Werte für alle Ressourcen geliefert, die aber über einem Minimum und unter einem Maximum liegen.

6.4.4.1 TimeController

Der **TimeController** verwaltet das Voranschreiten der Zeit im Spiel. Er wird benötigt, um den Tagesablauf zu simulieren und die Bedürfnisse der Agenten aufgrund der vergangenen Zeit anzupassen. Abbildung 6.13 zeigt den Aufbau der Klasse.

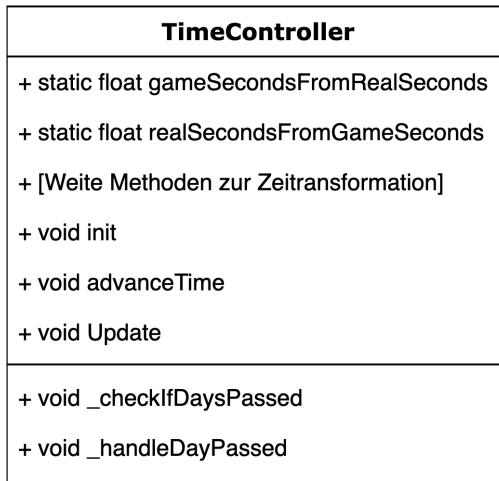


Abbildung 6.13.: Methoden der Klasse **TimeController**

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Die Unityfunktion `Update` wird in Intervallen aufgerufen und ist dafür zuständig, alle Komponenten, die von der Zeit abhängig sind, mit der aktuellen Zeit zu versorgen. Konkret müssen der `LightHandler` und der `NpcState` jedes Agenten für jeden neuen Frame aktualisiert werden. Wie weit die Zeit voranschreitet, ist von der vergangen Zeit seit dem letzten Aufruf abhängig. Es wird in Sekunden gerechnet, da `Time.deltaTime` in Sekunden angegeben wird. Die `Update`-Funktion ruft für jeden neuen Frame die Funktion `advanceTime` auf:

```

1  public void advanceTime(float deltaTime) {
2      _setCurrentTime(deltaTime);
3      _village.updateTime(deltaTime, _currentTime, _daysPassed);
4  }

```

Listing 6.3: Funktion `advanceTime` im `TimeController`

In der Funktion `_setCurrentTime` findet anschließend die eigentliche Aktualisierung der Zeit statt:

```

1  private void _setCurrentTime(float deltaTime) {
2      _currentTime += deltaTime * ONE_SECOND_NORMALIZED *
                     REAL_SECOND_INGAME_DURATION;
3      _checkIfDaysPassed();
4  }

```

Listing 6.4: Funktion `_setCurrentTime` im `TimeController`

Wichtig ist dabei die Zeile 3. Die Konstante `ONE_SECOND_NORMALIZED` ist der auf einen ganzen Tag normierte Anteil einer Sekunde an diesem Tag:

$$ONE_SECOND_NORMALIZED = \frac{1}{24 * 60 * 60} \approx 0.00012$$

Der Wert von `_currentTime` stellt die aktuelle Tageszeit dar, genormt auf einen Wert zwischen 0 und 1. Für jede im Spiel vergangene Sekunde wird `_currentTime` also mit dem Wert 0.00012 addiert. Im Spiel vergehen die Spiele jedoch schneller als in der Realität, weswegen `ONE_SECOND_NORMALIZED` mit `REAL_SECOND_INGAME_DURATION` multipliziert wird.

Die Funktion `_checkIfDaysPassed` prüft schließlich, ob ein Tag vergangen ist. Das ist der Fall, wenn `_currentTime > 1f`.

```

1  private void _checkIfDaysPassed() {
2      _daysPassed = 0;
3      while (_currentTime > 1f) {
4          _days++;
5          Monitor.Log("Days", _days.ToString());
6          _currentTime -= 1f;
7          _daysPassed += 1;
8          _handleDayPassed();
9      }
10 }

```

Listing 6.5: Funktion `_checkIfDaysPassed` im `TimeController`

Die Funktion `advanceTime` ruft für jedes Update die Funktion `Village.updateTime` auf. Darin wird die neue Zeit an alle weiteren Komponenten weitergeleitet, die diese benötigen. Konkret sind das alle mit dem Dorf verbundenen Agenten mit ihren Bedürfnissen sowie der `LightHandler`:

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

```
1 public void updateTime(float deltaTime, float currentTime, int
2     daysPassed) {
3     foreach (NpcAgent npc in _villageNpcs) {
4         npc.updateNpc(deltaTime, currentTime, daysPassed);
5     }
6     _lightHandler.updateLights(currentTime);
```

Listing 6.6: Funktion updateTime im TimeController

Weiterhin bietet der TimeController durch die Funktionen `gameSecondsFromRealSeconds` und `realSecondsFromGameSeconds` sowie weiteren Methoden eine Schnittstelle für die Umrechnung zwischen Echtzeit und Spielzeit. In einer Konstante wird dafür die Dauer einer reellen Sekunde für spätere Berechnungen gespeichert. Zusätzlich wird ein normalisierter Wert für den Anteil einer Sekunde an einem Tag festgelegt:

```
1 public const float REAL_SECOND_INGAME_DURATION = SECONDS_PER_DAY /
2     DAY_CYCLE_REAL_SECONDS;
2 public const float ONE_SECOND_NORMALIZED = 1f / SECONDS_PER_DAY;
```

Listing 6.7: Berechnung der Dauer einer Sekunde in Simulationszeit

`SECONDS_PER_DAY` beträgt logischerweise $60 * 60 * 24$ Sekunden. Mithilfe dieser Werte kann die Realzeit nun in Simulationszeit umgerechnet werden:

```
1 public static float realSecondsFromGameSeconds(float gameSeconds) {
2     return gameSeconds / REAL_SECOND_INGAME_DURATION;
3 }
4
5 public static float gameSecondsFromRealSeconds(float realSeconds) {
6     return ONE_SECOND_NORMALIZED * DAY_CYCLE_REAL_SECONDS *
7         realSeconds;
```

Listing 6.8: Umrechnung von Simulationszeit und Realzeit

6.4.5 LightHandler

Der LightHandler ist dafür zuständig, Tageszeiten zu simulieren, indem der Sonnenstand geregelt und Lichtquellen kontrolliert werden. In der Abbildung 6.14 wird der Aufbau der Klasse anhand ihrer Funktion verdeutlicht. Im LightHandler werden zusätzlich folgende Felder definiert:

- `List<Light> _agentLights`: Alle Lampen der Agenten. Jeder NPC verfügt über eine Lichtquelle, die sie an ihrem Körper tragen.
- `List<Light> _villageLamps`: Alle Laternen, die im und um das Dorf herum positioniert wurden.
- `List<Light> _bonfires`: Alle Lagerfeuer in der Simulation.
- `List<Transform> _bonfireParticleTransforms`: Die Partikeleffekte der Lagerfeuer.
- `Light _sun`: Eine Lichtquelle, die die Sonne simuliert.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

LightHandler
+ void init
+ void updateLights
+ void _initLights
+ void _handleSun
+ void _disableLights

Abbildung 6.14.: Methoden der Klasse LightHandler

Die Funktion `_initLights` sucht nach allen in der Szene enthaltenen Lichtquellen vom Typ `Light` und speichert diese. `Light` ist der Standardtyp für Lichtquellen in Unity. `Light` ist eine Komponente, die an ein `GameObject` angefügt wird. Innerhalb der Funktion `updateLights` werden bestimmte Parameter der Lichtquellen in Abhängigkeit von der aktuellen Zeit verändert. Die Lichtquellen der Agenten sowie die Laternen im Dorf werden, zu einer bestimmten Uhrzeit an- und zu einer bestimmten Uhrzeit wieder ausgeschaltet. Nachts leuchten diese Lichtquellen also durchgehend. Weiterhin gibt es in der Stadt eine Menge von Lagerfeuern. Tagsüber glühen diese leicht, mit zunehmender Dunkelheit erhöht sich ihre Intensität. Die Partikeleffekte werden dann zusätzlich leicht nach oben verschoben, sodass sie weiter aus dem Boden herausragen und es so wirkt, als würde das Feuer stärker brennen. Jeder einzelne Partikel des Lagerfeuers stellt eine kleine Lichtquelle dar, um einen realistischen Feuereffekt zu simulieren. Weiterhin wird in der Funktion `_handleSun` sowohl die Intensität als auch die Position und Ausrichtung der Sonne in Abhängigkeit von der Zeit angepasst, wie in 6.9 zu sehen.

```

1  private void _handleSun(float currentTime) {
2      _sun.transform.localRotation = Quaternion.Euler((currentTime *
3          360f) - 90f, 170f, 0f);
4
5      float multiplier = 1f;
6      if (currentTime <= 0.28f) {
7          multiplier = Mathf.Clamp01((currentTime - 0.25f) * (1f / 0.03
8              f));
9      }
10     else if (currentTime >= 0.75f) {
11         multiplier = Mathf.Clamp01(1f - ((currentTime - 0.78f) * (1f
12             / 0.03f)));
13     }
14     _sun.intensity = multiplier * 1.5f;
15 }
```

Listing 6.9: Anpassung der Sonne durch den LightHandler

Damit die Sonne sich wie eine Sonne verhält, muss die entsprechende Lichtquelle in den Lichteinstellungen der Unity-Szene als `Sun` markiert werden. Sie erhält dann automatisch das Aussehen einer Sonne und wird entsprechend positioniert. Unity übernimmt dann die eigentliche Beleuchtung und passt das `Light` so an, dass es eine überzeugende Sonne darstellt. Weiterhin verändert die Position der Sonne nun die Farbgebung der verwendeten Skybox - sobald die Sonne sich an den Rand des Horizonts bewegt, wird ein Sonnenuntergang bzw. Sonnenaufgang dargestellt. Wenn sie sich unter dem Horizont befindet, wird eine andere Textur für die Skybox verwendet, die in diesem Fall einen dunklen Sternenhimmel zeigt. Nun muss nur noch die Rotation der Sonne angepasst

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

werden. Dadurch beschreibt die Position der Sonne im Verlauf des Tages eine Kreisbahn um das Dorf herum. Weiterhin wird die Intensität angepasst, sodass die Lichtverhältnisse mit der Tageszeit übereinstimmen. In den Abbildungen 6.15 und 6.16 ist beispielhaft zu erkennen, wie die Stadt nachts beleuchtet wird. Zum Vergleich wurden hier die selben Motive und Kameraeinstellungen verwendet, wie bereits in den Grafiken 6.3 und 6.4.



Abbildung 6.15.: Modellierte Stadt, äußere Ansicht, nächtliche Beleuchtung



Abbildung 6.16.: Modellierte Stadt, innere Ansicht, nächtliche Beleuchtung

Für das Training des Agenten ist die Performance der Anwendung ausschlaggebend. Da die Entscheidungsfindung der Agenten in keiner Weise von der Beleuchtung der Szene abhängt, jedoch einen

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

großen Teil der aufgewendeten Leistung in einer Simulation beansprucht, werden alle Lichtquellen während des Trainings automatisch deaktiviert. Dafür wird `GameAcademy.IS_TRAINING` geprüft. In diesem Fall wird der `Lighthandler` nur initialisiert und dann deaktiviert, da keine Lichtquellen mehr zeitabhängig angepasst werden müssen.

6.4.6 Implementierung des Agenten und dessen Subkomponenten

Insgesamt wurden 16 Observations und 29 diskrete Aktionen verwendet, die dem Agenten zur Auswahl stehen. Es werden ausschließlich On Demand Decisions verwendet (siehe Kapitel 5). Der Agent wird während des Trainings nach einer festgelegten Anzahl von Entscheidungen zurückgesetzt. Das Training erfolgt mit dem PPO-Algorithmus (siehe Abschnitt 3.3.2.5) und verwendet ein neuronales Netz mit mehreren Schichten und mehreren hundert Neuronen. Es können beliebig viele Agenten verwendet werden. Zunächst werden die grundsätzlichen Funktionen der Klasse `NpcAgent` betrachtet, welche für die Realisierung der Agenten erstellt wurde. Die Funktionen der Klasse werden in der Abbildung 6.17 dargestellt.

NpcAgent	
+ void	<code>Awake</code>
+ void	<code>init</code>
+ void	<code>CollectObservations</code>
+ void	<code>AgentAction</code>
+ void	<code>AgentReset</code>
+ void	<code>resetMoveAndWait</code>
+ void	<code>requestModelOrManualDecision</code>
+ void	<code>setJob</code>
+ void	<code>_executeAction</code>
+ void	<code>_actionPlanFinishedCallback</code>
+ void	<code>_addStepPenalties</code>

Abbildung 6.17.: Methoden der Klasse `NpcAgent`

Der `NpcAgent` verwendet sogenannte *ActionPlans* für die sequentielle Ausführung von Aktionen. Diese werden im späteren Abschnitt 6.4.6.1 behandelt.

Die Funktionen `CollectObservations`, `AgentAction`, `AgentReset` und `Awake` sind bereits bekannt (siehe Kapitel 5). Die Funktionen `Awake` `setJob` und `init` werden benötigt, um den Agenten zu initialisieren. Weiterhin werden `resetMoveAndWait` und `AgentReset` verwendet, um den Agenten beispielsweise nach dem Abschließen einer Episode zurückzusetzen. In der Funktion `_actionPlan-FinishedCallback` wird nach dem Ausführen einer Aktion geprüft, ob eine Episode abgeschlossen ist oder eine neue Entscheidung mit `requestModelOrManualDecision` angefragt werden soll. In der Funktion `_addStepPenalties` werden darüber hinaus Bestrafungen ausgeschüttet. Konkret wird für jedes Bedürfnis des Agenten, das unter einer bestimmten Grenze liegt, eine Bestrafung zugewiesen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Der Beobachtungsraum des Agenten wird durch die in der Funktion `CollectObservations` getätigten Observations beschrieben, wie in Listing 6.10 zu sehen:

```

1  public override void CollectObservations() {
2      AddVectorObs(_npcState.NeedHunger);
3      AddVectorObs(_npcState.NeedThirst);
4      AddVectorObs(_npcState.NeedEducation);
5      AddVectorObs(_npcState.NeedCommunication);
6      AddVectorObs(_npcState.NeedFaith);
7      AddVectorObs(_npcState.Recreation);
8      AddVectorObs(_npcState.Exhaustion);
9      AddVectorObs(_npcState.AttFitness);
10     AddVectorObs(_npcState.AttIntelligence);
11     AddVectorObs(_npcState.AttCompanionship);
12     AddVectorObs(_npcState.AttCommodities);
13     AddVectorObs(_npcState.AttFerocity);
14     AddVectorObs(_npcState.Sleep);
15     AddVectorObs(_npcState.Work);
16     AddVectorObs(_npcState.getMoneyNormalized());
17     AddVectorObs(_currentTime);
18
19     SetActionMask(_actionMaskingHandler.getIntActionMask(this));
20 }
```

Listing 6.10: Beobachtungen des Agenten

Als Input für das neuronale Netz werden also sämtliche Bedürfnisse sowie die Attribute, die Menge des Geldes des Agenten und die aktuelle Zeit verwendet. Insgesamt kommen so 16 Observations zustande. Der Handlungsräum besteht aus 29 möglichen Aktionen für die Befriedigung von Bedürfnissen. Die Funktion `AgentAction` ist in Listing 6.11 zu erkennen:

```

1  public override void AgentAction(float[] vectorAction, string
        textAction) {
2      _executeAction(NpcActions.actionNameFromId(Mathf.RoundToInt(
            vectorAction[0])));
3  }
4
5  private void _executeAction(string actionId) {
6      [...] // Logging etc.
7      _action = actionId;
8      _academy.IncrementNumAgentActions();
9      _numActions++;
10     _actionHandler.runActionPlan(actionId);
11 }
```

Listing 6.11: Funktion `AgentAction` des Agenten

Zunächst wird aus dem Float-Wert, der von ML-Agents geliefert wird, ein Integer gemacht. Das ist sinnvoll, weil hier diskrete Entscheidungen verwendet werden, die durch einen Integer-Wert referenziert werden. In der Funktion `_executeAction` werden schließlich nach einigen Logging-Anweisungen Zähler inkrementiert und durch die Anweisung `_actionHandler.runActionPlan` der `ActionHandler` aufgerufen, der dem Identifier der Aktion einen ausführbaren ActionPlan zuweist. Im nächsten Abschnitt wird dieses Vorgehen erläutert. Es wird darauf eingegangen, wie die Aktionen des Agenten beschrieben und ausgeführt werden können.

6.4.6.1 ActionPlans

Für die Ausführung von Aktionen wurde eine eigene Kontroll- und Datenstruktur entwickelt, die eine Reihe von untergeordneten Aktionen enthält. Die Struktur ist allgemein für sequentielle Ausführungen von Aufgaben nutzbar, deren zeitliche Dauer zum Zeitpunkt der Ausführung unbekannt sein kann. In dieser Arbeit wurden diese *ActionPlans* genannten Konstrukte für die Ausführung der Aktionen des Agenten angefertigt. Im Folgenden werden sie näher erläutert. ActionPlans erben von der Basisklasse `BaseActionPlan`. Die Abbildung 6.18 zeigt die Funktionen der Klasse:

BaseActionPlan	
+ virtual bool	areRequirementsMet
+ virtual void	runActions
+ virtual void	_addPlanActions
+ virtual void	_initRun
+ virtual float	_getReward
+ virtual void	_additionalFinishAction
+ void	_executeNextAction
+ void	_onFinish
+ void	_actionPlanCancelledCallback
+ [Diverse Wrapper- und Hilfsmethoden]	

Abbildung 6.18.: Methoden der Klasse `BaseActionPlan`

In der Klasse `NpcActions` werden String-Identifier für alle Aktionen definiert, zum Beispiel `NpcActions.HUNGER_BREAD`. `NpcActions` liefert weiterhin ein Mapping dieser Identifier zu den numerischen Identifizieren von Aktionen, die ML-Agents beim Aufruf der Funktion `AgentAction` liefert. Anhand dieser Identifier wird jeder Aktion aus dem Aktionsraum des neuronale Netzes genau ein übergeordneter ActionPlan zugewiesen. Die Menge aller Sub-Aktionen eines ActionPlans beschreibt genau eine übergeordnete Aktion. Von der Basisklasse `BaseActionPlan` müssen bestimmte Funktionen überschrieben werden. Innerhalb der Funktion `_addPlanActions` werden eine Reihe von Aktionen definiert, die sequentiell ausgeführt werden, wie beispielsweise das Navigieren des Agenten zu einer Zielposition. Zusätzlich zur Ausführung von Aktionen werden hier Effekte und Voraussetzungen für Aktionen verwaltet. Die Ausführung der einzelnen Teilaktionen erfolgt in der Funktion `nextActionExecutedCallback`. Nach der Ausführung jeder Einzelaktion wird ein Callback aufgerufen, wie in Listing 6.12 zu erkennen. Es handelt sich um eine einfache Funktion ohne Parameter, die jeder Einzelaktion im Konstruktor mitgegeben wird.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

```

1  private void nextActionExecutedCallback()
2  {
3      _currentAction += 1;
4
5      if (_currentAction >= _actions.Count) {
6          _onFinish();
7          _allActionsExecutedCallback();
8      } else {
9          _executeNextAction();
10     }
11 }

```

Listing 6.12: ActionPlan-Callback

Wenn weitere Aktionen vorhanden sind, wird die nächste Aktion aus der Liste ausgeführt. Auf die Terminierung der letzten Aktion folgt der Aufruf eines weiteren, übergeordneten Callbacks `_allActionsExecutedCallback`. Auch hier handelt es sich um eine einfache parameterlose Funktion. Der Callback ruft im `NpcAgent` die Funktion `_actionPlanFinishedCallback` auf und führt zu einer Anfrage an das ML-Agents Toolkit für eine neue Entscheidung. Nach der Ausführung der Aktionen wird weiterhin eine Belohnung oder eine Bestrafung für den Agenten ausgeschüttet, indem `_onFinish` aufgerufen wird. Zusätzlich können abschließend beispielsweise Ressourcen hinzugefügt werden, die die Aktion produziert hat.

Zum Start des ActionPlans wird die in Listing 6.13 dargestellte Funktion `runActions` verwendet:

```

1  public void runActions() {
2      _npc.VillageInfo.changeAmountForAction(true, _actionType);
3      _currentAction = 0;
4      _initRun();
5
6      if (_actions == null || _actions.Count == 0) {
7          _onFinish();
8          _allActionsExecutedCallback();
9      } else {
10         _executeNextAction();
11     }
12 }

```

Listing 6.13: Starten eines ActionPlans

Im Konstruktor von `BaseActionPlan` wird die abstrakte Funktion `_addPlanActions` definiert, welche von Unterklassen überschrieben werden muss. Darin müssen die Aktionen des ActionPlans hinzugefügt werden, indem sie durch `_actions.Add(newAction)` in eine `List<BaseAction>` geschrieben werden. In `runActions` wird zunächst die erste dieser Aktionen mit dem Index `_currentAction = 0` ausgeführt. Nach jeder ausgeführten Aktion wird der Index erhöht und die nächste Funktion ausgeführt.

Ein Beispiel für eine Implementierung eines vollständigen ActionPlans ist in 6.14 zu finden. Der ActionPlan `ReligionChurchActionPlan` beschreibt den Vorgang eines Agenten, die Kirche zu besuchen, um sein Bedürfnis nach *Glauben* aufzufüllen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

```

1  public class ReligionChurchActionPlan : BaseActionPlan
2  {
3      public ReligionChurchActionPlan(Action allActionsExecutedCallback ,
4                                      NpcAgent npc)
5          : base(NpcActions.RELIGION_CHURCH, allActionsExecutedCallback, npc) { }
6
7      public override bool areRequirementsMet() {
8          if (!_hasFreeSpot(LocationTypes.CHURCH)) return false;
9          return true;
10     }
11
12     protected override float _getReward() {
13         return _getNeedReward(NpcState.NEED_FAITH, NEED_REWARD_MULTIPLIER);
14     }
15
16     protected override void _addPlanActions() {
17         _addMoveAction(LocationTypes.CHURCH);
18         _addWaitAction(DEFAULT_ACTION_SECONDS);
19         _addVacateAction(LocationTypes.CHURCH);
20     }
21
22     protected override void _additionalFinishAction() {
23         _npcState.satisfyNeedByType(NpcState.NEED_FAITH, DEFAULT_NEED_GAIN);
24     }
25 }
```

Listing 6.14: Beispielhafter ActionPlan `ReligionChurchActionPlan`

Bevor die Entscheidung getroffen wird, welcher ActionPlan ausgeführt werden soll, wird die Funktion `areRequirementsMet` jedes ActionPlans aufgerufen. Diese prüft einige Bedingungen, die erfüllt sein müssen, damit der ActionPlan ausgeführt werden darf. Die Voraussetzung für die Ausführung in dem obigen Beispiel stellt die Bedingung auf, dass mindestens ein freier Slot in einem Gebäude des Typs CHURCH vorhanden sein muss. Die einzelnen Teilaktionen des ActionPlans werden in `_addPlanActions` definiert. Zunächst wird eine `MoveAction` definiert, die den Agenten zur Kirche bewegt. Diese Aktion beinhaltet die Reservierung eines Slots für den Agenten. Weiterhin wird eine `WaitAction` definiert, welche typischerweise genutzt wird, um die Ausführung einer beliebigen Tätigkeit zu simulieren. Effektiv bleibt der Agent an einem Ort stehen und wartet, bis eine gewisse Zeit abgelaufen ist. Anschließend wird eine `VacateAction` ausgeführt, welche dafür sorgt, dass zum Ende des Ausführungsprozesses des ActionPlans der reservierte Slot für den Agenten wieder freigegeben wird. Nach der Ausführung der Aktionen wird schließlich durch die Funktion `_getReward` die Belohnung definiert. Diese hängt von dem initialen Bedürfnis nach Glauben ab, das zu Beginn der Ausführung des ActionPlans festgestellt wurde. Der Vorgang wird im `BaseActionPlan`, unter anderem in der Funktion `_initRun`, beschrieben. Als letztes wird das Bedürfnis des Agenten nach Glauben aufgefüllt.

Alle ActionPlans eines Agenten werden im `ActionHandler` erzeugt. Mithilfe des in `NpcActions` definierten Mappings werden die ActionPlans in einem `Dictionary<String, BaseActionPlan>` hinterlegt. Sobald eine neue Aktion des Agenten mit `AgentAction` gestartet wird, wählt der `ActionHandler` den zum Identifier der Aktion passenden ActionPlan aus.

In der Abbildung 6.19 ist der Aufbau der Klasse `ActionHandler` zu erkennen:

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

ActionHandler
+ void runActionPlan
+ void runInitialAction
+ bool areRequirementsMetForAction
+ void logReward
+ void _setupVars
+ void _setupActionPlans

Abbildung 6.19.: Methoden der Klasse ActionHandler

Die Funktion `runInitialAction` wird verwendet, um nach der Initialisierung des Dorfes für jeden Agenten eine `WaitAction` mit zufälliger Dauer durchzuführen, sodass nicht alle Agenten zur selben Zeit anfangen, Aktionen auszuführen. Damit wird vermieden, dass Engpässe bezüglich der Kapazität bestimmter Gebäude entstehen. Die Funktion `areRequirementsMetForAction` dient lediglich als Wrapper für die `areRequirementsMet`-Funktionen der einzelnen ActionPlans und wird vom `ActionMaskingHandler` aufgerufen.

Belohnungen Die Belohnungen der Aktionen des Agenten werden ebenfalls in den ActionPlans definiert. Dafür wurde eine Reihe von Hilfsfunktionen erstellt, zum Beispiel `_getNeedReward` oder `_getRecreationReward`. Die Zusammensetzung der Belohnungen kann in den jeweiligen ActionPlans im Programmcode betrachtet werden.

Teilaktionen des ActionPlans Zusätzlich zu den genannten Methoden werden diverse weitere Wrapper- und Hilfsmethoden implementiert, wie beispielsweise die Funktion `_moveAndWaitRealSeconds`, die automatisch die Teilaktionen `MoveAction`, `VacateAction` und `WaitAction` hinzufügt, um zu einem Ort zu navigieren und dort zu verweilen. Diese Teilaktionen des ActionPlans erben von der Klasse `BaseAction`. Die Implementierung dieser Basisklasse ist im folgenden Listing zu erkennen:

```

1  public class BaseAction {
2      protected string _actionType;
3      protected NpcAgent _npc;
4      protected TimeController _timeController;
5
6      public BaseAction(string actionType, NpcAgent npc) {
7          _actionType = actionType;
8          _npc = npc;
9          _timeController = GameObject.FindObjectOfType<TimeController>();
10     }
11
12     public virtual IEnumerator execute(Action callback,
13                                         Action cancelCallback) {
14         throw new System.NotImplementedException("Function execute not
15             implemented for " + this.GetType());
16     }
17 }
```

Listing 6.15: Implementierung von `BaseAction`

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Klassen, die von `BaseAction` erben, müssen nur die `execute`-Funktion überschreiben und eine Callbackfunktion aufrufen, sobald die Teilaktion abgeschlossen wurde.

Um den Agenten zu einem Zielort zu bewegen wird die `MoveAction` verwendet. Diese interagiert zunächst mit dem `LocationHandler`, um einen Zielort zu erlangen und schließlich mit dem `MoveHandler`, um den Agenten zum Ziel zu navigieren. In der Abbildung 6.20 wird der Aufbau der Klasse `MoveHandler` verdeutlicht:

MoveHandler	
+ void moveAgent	
+ void Update	
+ void reset	
+ int _getMoveStatus	
+ void _endMove	
- NavMeshAgent _navAgent	
- Action _moveCallback	
- string _moveAction	
- float _startTime	

Abbildung 6.20.: Methoden der Klasse `MoveHandler`

Die Funktion `moveAgent` startet die Navigation zum Ziel. Das Feld `_navAgent` referenziert den `NavMeshAgent`, der verwendet wird, um auf dem NavMesh zu navigieren (siehe Abschnitt 6.4.6.4). Durch die Anweisung `_navAgent.SetDestination(destination)` wird die Navigation des Agenten zu seinem Zielort gestartet. Zu Logging-Zwecken wird außerdem die Startzeit der Aktion im Feld `_startTime` gespeichert. Die `Update`-Funktion prüft anschließend mit jedem neuen Frame den Status des Agenten mithilfe der Funktion `_getMoveStatus`. Sobald diese Funktion den Status `MOVE_STATUS_STOPPED` zurückgibt, hat der Agent sein Ziel erreicht und ein Callback wird aufgerufen, um zurück zur `MoveAction` zu springen, welche wiederum einen weiteren Callback zum jeweiligen `ActionPlan` ausführt.

WaitHandler	
+ void wait	
+ void Update	
+ void reset	
- Action _waitFor	
- float _secondsToWait	
- float _secondsPassed	
- bool _isWaiting	

Abbildung 6.21.: Methoden der Klasse `WaitHandler`

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Eine weitere essentielle Teilaktion ist die `WaitAction`. Diese funktioniert grundsätzlich wie die `MoveAction`, gibt dem zugrunde liegenden `WaitHandler` jedoch kein örtliches Ziel vor, sondern teilt diesem eine Wartezeit mit. Der Aufbau der Klasse `WaitHandler` ist in Abbildung 6.21 zu erkennen.

Durch die Funktion `wait` wird der Wartevorgang eingeleitet. Der Parameter der Funktion bestimmt, wie lange gewartet werden muss. Dieser Wert wird im Feld `_secondsToWait` gespeichert. Die Variable `_secondsPassed` sagt aus, wie lange der Wartevorgang bereits andauert. In der Funktion `Update` wird nun `_secondsPassed` mit `Time.deltaTime` addiert. Sobald gilt, dass `_secondsPassed >= _secondsToWait`, wird der Callback `_waitCallback` aufgerufen und die Variablen der Instanz des `WaitHandler` durch die Funktion `reset` zurückgesetzt. Da `Time.deltaTime` automatisch skaliert wird, funktioniert das Vorgehen auch bei hohen Timescale-Faktoren.

ActionPlans für Berufe Die ActionPlans, die berufliche Tätigkeiten darstellen sollen, bauen indirekt aufeinander auf. Durch eine verkettete Beziehung der Ressourcen untereinander entstehen zwischen den einzelnen Ressourcen gewisse Abhängigkeiten. Insgesamt gibt es fünf Endressourcen, die produziert werden müssen: *Werkzeuge*, *Feuerholz*, *Bier*, *Fleisch* und *Brot*. Für die Herstellung der Werkzeuge muss zunächst ein Minenarbeiter Erz produzieren, welches dann vom Schmied in Werkzeuge verarbeitet werden kann. Für das Feuerholz muss zunächst ein Waldarbeiter Holz hacken, für das Bier muss zunächst Hopfen geerntet und anschließend zu Bier verarbeitet werden und für die Produktion von Fleisch ist zunächst das Jagen von Wild nötig (das nicht dargestellt wird), welches dann zu Fleisch weiterverarbeitet wird. Die einzige Produktionskette mit drei Schritten ist die Herstellung von Brot - erst muss dafür Weizen abgebaut werden, dann muss dieser in der Windmühle zu Mehl verarbeitet und erst dann kann daraus vom Bäcker Brot gebacken werden. Zwischen den ActionPlans besteht kein direkter Zusammenhang, doch falls eine Ressource nicht ausreichend vorhanden ist, kann die Kette unterbrochen werden, sodass ebenfalls weitere Ressourcen nicht hergestellt werden können. Alle Berufe verbrauchen zudem Werkzeuge. Das gilt nur nicht für Minenarbeiter und Schmiede, da diese die Werkzeuge herstellen.

Übersicht über ausführbare Aktionen Es wurden insgesamt 29 ausführbare ActionPlans erstellt, die die Aktionen des Agenten beschreiben. Diese werden durch den `ActionHandler` verwaltet und dort in einem Mapping den Identifizierten Aktionen zugewiesen.

Die ActionPlans werden in die Kategorien *Communication*, *Education*, *Exhaustion*, *Hunger*, *Recreation*, *Religion*, *Sleep*, *Thirst* und *Work* eingeteilt. Für jedes der Bedürfnisse der Agenten gibt es mindestens einen ActionPlan, der dieses Bedürfnis befriedigt.

Das folgende Listing enthält eine kurze Beschreibung der jeweiligen Aktionen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

ActionPlan	Beschreibung	Ort	Dauer
CommunicationChatActionPlan	Befriedigt Kommunikationsbedürfnis. Agenten treffen sich dafür in Gruppen und kommunizieren.	Marktplatz	Kurz
EducationCollegeActionPlan	Befriedigt Bildungsbedürfnis. Verfügbar für Agenten zwischen 19 und 35 Jahren.	College	Mittel
EducationLibraryActionPlan	Befriedigt Bildungsbedürfnis. Verfügbar für Agenten ab 36 Jahren.	Bibliothek	Mittel
EducationSchoolActionPlan	Befriedigt Bildungsbedürfnis. Verfügbar für Agenten zwischen 0 und 18 Jahren.	Bibliothek	Mittel
ExhaustionRestActionPlan	Verbessert Erschöpfungslevel des Agenten.	Zuhause	Mittel
HungerBreadActionPlan	Befriedigt Hungerbedürfnis und verbraucht Brot.	Markt	Kurz
HungerMeatActionPlan	Befriedigt Hungerbedürfnis und verbraucht Fleisch.	Markt	Kurz
RecreationBarbecueActionPlan	Freizeitaktion. Agenten treffen sich in Gruppen am Lagerfeuer.	Lagerfeuer	Mittel
RecreationCafeActionPlan	Freizeitaktion. Agent besucht Cafe.	Cafe	Mittel
RecreationShootingActionPlan	Freizeitaktion. Agent besucht Schießstand.	Schießstand	Mittel
RecreationShopActionPlan	Freizeitaktion. Agent besucht Laden.	Shop	Mittel
RecreationSoccerActionPlan	Freizeitaktion. Agent besucht Fußballplatz.	Fußballplatz	Mittel
RecreationTavernActionPlan	Freizeitaktion. Agent besucht Taverne.	Taverne	Mittel
RecreationTheaterActionPlan	Freizeitaktion. Agent besucht Theater.	Theater	Mittel
RecreationVisitActionPlan	Freizeitaktion. Agent besucht Haus eines anderen NPC (Auswahl erfolgt zufällig).	Fremdes Zuhause	Mittel
ReligionChurchActionPlan	Befriedigt Glaubensbedürfnis. Agent besucht Kirche.	Kirche	Kurz
SleepActionPlan	Befriedigt Schlafbedürfnis.	Zuhause	Lang
ThirstActionPlan	Befriedigt Durstbedürfnis. Der Agent trinkt vom Brunnen.	Brunnen	Sehr Kurz
WorkBeerBreweryActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Braumeister. Verbraucht Hopfen.	Brauerei	Lang
WorkBeerFarmHopsActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Hopfenbauer.	Hopfenfarm	Lang
WorkBreadBakeryActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Bäcker. Stellt Brot her. Verbraucht Mehl.	Bäckerei	Lang
WorkBreadFarmGrainActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Weizenbauer. Stellt Weizen her.	Weizenfarm	Lang
WorkBreadWindmillActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Müller. Stellt Mehl her. Verbraucht Weizen.	Windmühle	Lang
WorkMeatButcherActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Fleischer. Stellt verarbeitetes Fleisch her. Verbraucht rohes Fleisch.	Fleischerei	Lang
WorkMeatHuntingActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Jäger. Stellt rohes Fleisch her.	Wald	Lang
WorkToolsBlacksmithActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Schmied. Stellt Werkzeuge her. Verbraucht Eisen.	Schmied	Lang
WorkToolsMiningActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Minenarbeiter. Stellt Eisenerz her.	Mine	Lang
WorkWoodCarpenterActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Zimmermann. Stellt Holz her. Verbraucht Holzstämme.	Zimmermann	Lang
WorkWoodLoggingActionPlan	Befriedigt Arbeitsbedürfnis. Agent arbeitet als Holzfäller. Stellt Holzstämme her.	Wald	Lang

Tabelle 6.2.: Auflistung aller ActionPlans

6.4.6.2 NpcState

In der Klasse `NpcState` wird der Zustand des Agenten inklusive seiner Bedürfnisse und Attribute beschrieben. Die Abbildung 6.22 zeigt die Funktionen der Klasse:

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

NpcState
+ void satisfyNeedByType
+ void updateState
+ void satisfyNeedByType
+ void updateExhaustion
+ void resetState
+ void getNeedByType
+ void getAttributeByType
+ void payMoney
+ void getMoney
+ Dictionary<string, float> getNeedsClone

Abbildung 6.22.: Methoden der Klasse `WaitHandler`

Die Methoden `payMoney` und `gainMoney` werden genutzt, um die Geldmenge des Agenten zu verändern, beispielsweise wenn eine Aktion als Voraussetzung gewisse Kosten erfordert. Zusätzlich zu den im `ResourceHandler` definierten Ressourcen verfügt der Agent über eine spezielle Ressource `Money`, die im `NpcState` definiert wird. Im Gegensatz zu normalen Ressourcen verfügt jeder Agent über einen individuellen Wert für die Geldmenge, die er besitzt. Bestimmte Aktionen sind mit Kosten verbunden, die der Agent bezahlen muss. Geld kann nur durch Arbeits-Aktionen dazugewonnen werden. In einem `ActionPlan` kann durch die Veränderung der Variable `_money` ein Wert für die Kosten der Aktion gesetzt werden. Dadurch soll der Agent über den Reward hinaus dazu angeregt werden, zu arbeiten. Außerdem führt dies dazu, dass bestimmte teure Aktionen von bestimmten Bevölkerungsgruppen, die mehr Geld verdienen, bevorzugt werden. Ein Bäcker verdient beispielsweise mehr Geld als ein Bauer.

Es werden einige Funktionen definiert, um Bedürfnisse und Attribute zu aktualisieren und abzufragen. In der Funktion `resetState` findet weiterhin nach dem Abschließen einer Episode ein Zurücksetzen der Werte statt. Da für die verteilte Belohnung in `ActionPlans` die initialen Bedürfnisse bekannt sind wurde die Funktion `getNeedsClone` implementiert. Sie gibt eine Kopie der aktuellen Werte aller Bedürfnisse zurück, die zu Beginn der Ausführung eines `ActionPlans` bezogen werden und nach der Ausführung verwendet werden kann. Die Funktion `updateState` wird vom `TimeController` in regelmäßigen Abständen aufgerufen. Sie aktualisiert den Zustand des Agenten durch die Bereitstellung der neuen aktuellen Zeit und verändert damit seine Bedürfnisse. Die Funktion ist in Abbildung 6.16 zu sehen:

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

```

1  public void updateState( float currentTime , float deltaTime ) {
2      _exhaustion += 0.0002f * deltaTime ;
3
4      List<string> needsToUpdate = new List<string>();
5      foreach ( string needType in _needs . Keys ) needsToUpdate . Add(needType) ;
6      foreach ( string needType in needsToUpdate ) {
7          _secondsSinceSatisfaction [needType] += deltaTime ;
8          _needs [needType] = _needCycleHandler . getUpdatedNeedValue(needType ,
9                          _secondsSinceSatisfaction [needType]) ;
9          if ( _useGoap && _needs [needType] < 0.1f ) _goapEffectSetter .
10             setGoapEffect (needType , true) ;
10     }
11     _recreation = _needCycleHandler . calculateRecreationValue(_needs) ;
12     _sleep = _needCycleHandler . calculateSleepValue (currentTime) ;
13     _work = _needCycleHandler . calculateWorkValue (currentTime) ;
14 }
```

Listing 6.16: Funktion `updateState` in `NpcState`

Neben den Werten für Exhaustion und Recreation (siehe Kapitel 6.4.6.3) werden ebenfalls die Bedürfnisse des Agenten angepasst, indem zusätzliche Funktionen aufgerufen werden. Die Bedürfnisse Schlaf und Arbeit werden dabei gesondert behandelt. Die genaue Funktionsweise der Bedürfnisse wird im folgenden Abschnitt 6.4.6.3 behandelt.

6.4.6.3 Bedürfnisse

Für die Simulationen werden Utility-Based Agents verwendet (siehe Abschnitt 2.5). Konkret werden diese so implementiert, dass sie über verschiedene menschliche Bedürfnisse verfügen. Im `NeedCycleHandler` werden jeweils Funktionen $g_b(x)$ definiert, die diese Bedürfnisse in Abhängigkeit von der verstrichenen Zeit verändern.

Ein Bedürfniswert von $g_b(x) = 1$ bedeutet, dass der Agent hinsichtlich dieses spezifischen Begehrns vollkommen befriedigt ist. Ein Bedürfniswert von $g_b(x) = 0$ hingegen bedeutet, dass der Agent ein starkes Bedürfnis hat, das es zu befriedigen gilt. Als Grundbedürfnisse werden *Hunger*, *Durst*, *Bildung*, *Kommunikation*, und *Glauben* definiert. Weiterhin gibt es die Bedürfnisse *Schlaf* und *Arbeit*. Im Spiel werden diese als *NEED_HUNGER*, *NEED_THIRST*, *NEED_EDUCATION*, *NEED_COMMUNICATION*, *NEED_FAITH*, *SLEEP* und *WORK* implementiert.

Die Werte für die fünf Grundbedürfnisse hängen davon ab, wann das Bedürfnis zuletzt befriedigt wurde. Sie verwenden die folgende Grundfunktion $g_b(x)$:

```

1  private float _getFunctionValue( float cycleLengthSeconds , float
2      exponent , float xValue ) {
3      float baseVal = xValue / cycleLengthSeconds ;
4      float pow = Mathf . Pow (baseVal , exponent) ;
5      return Mathf . Clamp (1f - pow , 0f , 1f) ;
```

Listing 6.17: Grundfunktion für die Abnahme der Grundbedürfnisse

Für jede der Basisfunktionen $g_b(x)$ werden die in der Grundfunktion verwendeten Parameter wie `exponent`, `baseVal` und `cycleLengthSeconds` eigens definiert. Wenn die Bedürfnisse aktualisiert werden, wird für die Berechnung der einzelnen Werte ein *Switch-Case*-Konstrukt verwendet, wie in Listing 6.18 zu sehen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

```

1  public float getUpdatedNeedValue(string needType, float
2      timeSinceSatisfaction) {
3      switch(needType) {
4          case NpcState.NEED_HUNGER:
5              return _getFunctionValue(HUNGER_CYCLE_SECONDS,
6                  HUNGER_DROP_RATE, timeSinceSatisfaction);
7          case NpcState.NEED_THIRST:
8              return _getFunctionValue(THIRST_CYCLE_SECONDS,
9                  THIRST_DROP_RATE, timeSinceSatisfaction);
10         [...] // Cases with other needs.
11     }
12 }
```

Listing 6.18: Aufruf der Grundfunktion bei einer Anpassung der Bedürfnisse

Das folgende Beispiel betrachtet exemplarisch die Hungerfunktion $g_{\text{hunger}}(x)$ des Agenten. Wenn die verwendeten Konstanten eingesetzt werden, ergibt sich der folgende Aufruf:

```

1  _getFunctionValue(
2      cycleLengthSeconds = DAY_CYCLE_SECONDS * 0.5f,
3      exponent = 2.5f,
4      xValue = timeSinceSatisfaction
5 );
```

Listing 6.19: Aufruf der Grundfunktion bei einer Anpassung der Bedürfnisse

Durch Einsetzen der Werte verändert das die Grundfunktion wie folgt:

```

1  float baseVal = timeSinceSatisfaction / (DAY_CYCLE_SECONDS * 0.5f);
2  float pow = Mathf.Pow(baseVal, 2.5f);
3  return Mathf.Clamp(1f - pow, 0f, 1f);
```

Listing 6.20: Grundfunktion für Bedürfnisse (Beispiel Hungerbedürfnis)

Nun ist leicht zu sehen, dass sich folgende mathematische Funktion daraus ergibt:

$$g_{\text{hunger}}(x) = 1 - (x/500)^2.5$$

Die Funktionen für das Schlaf- und Arbeitsbedürfnis wurden anhand von Abbildung 6.23 visualisiert. Für die Variable `DAY_CYCLE_SECONDS`, die den Tageszyklus bestimmt, wird dabei eine Dauer von 1000 Sekunden genutzt.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

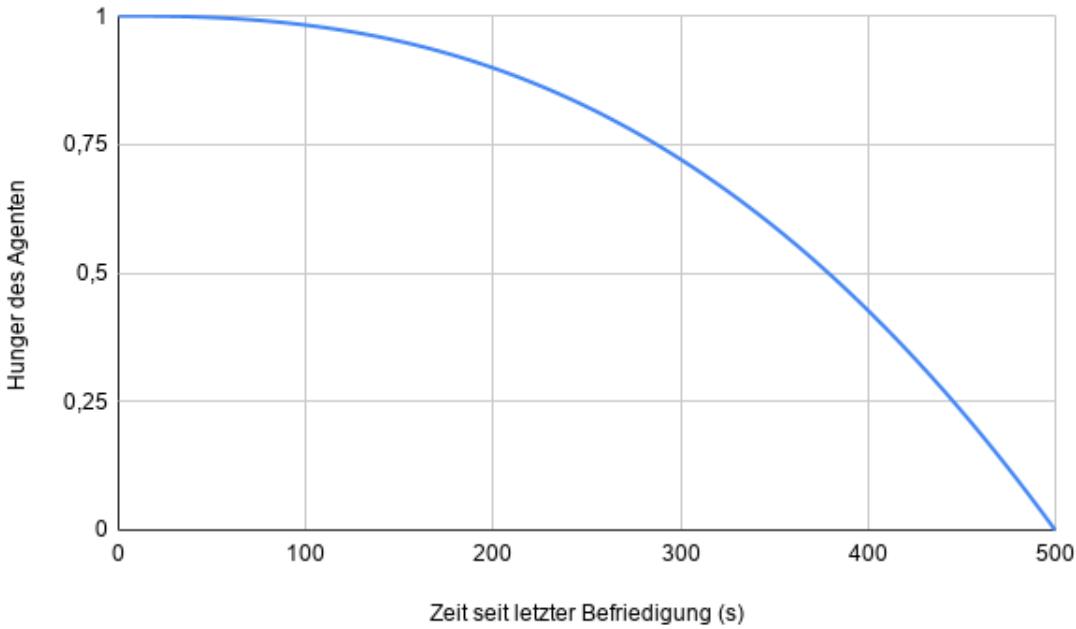


Abbildung 6.23.: Hungerfunktion des Agenten

Die Bedürfnisse *Schlaf* und *Arbeit* verhalten sich anders. Diese sind nicht davon abhängig, wann das Bedürfnis das letzte mal befriedigt wurde, sondern hängen von der Tageszeit ab. Das Listing 6.21 zeigt, wie das Schlafbedürfnis berechnet wird.

```

1  public float calculateSleepValue(float currentTime) {
2      float baseOffsetX = -0.125f;
3
4      bool exceedsBorders = _exceedsBorders(0.125f - baseOffsetX, 0.625
5          f - baseOffsetX, currentTime);
6      if (!exceedsBorders) return 1f;
7
8      float frequency = 4f * Mathf.PI;
9      float offset = currentTime + baseOffsetX;
10     float funcRes = Mathf.Sin(frequency * offset) * 0.5f + 0.5f;
11     return Mathf.Clamp(funcRes, 0f, 1f);
12 }
```

Listing 6.21: Funktion für das Schlafbedürfnis

Daraus ergibt sich folgende Funktion:

$$g_{sleep}(x) = \sin(4\pi * (x - 0.125)) * 0.5 + 0.5$$

Diese Funktionen werden als Basis für eine partielle Funktion verwendet. Die Funktion `_exceedsBorders` prüft, ob die Werte der X-Achse bestimmte Grenzen überschreiten. Falls dies der Fall ist, wird der Funktionswert nicht durch die oben definierten Funktionen berechnet, sondern entspricht automatisch dem Wert 1. Die Intervalle, die diese Grenzen für das Schlaf- und Arbeitsbedürfnis definieren, heißen I_{sleep} bzw. I_{work} .

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

$$g_{sleep}(x) = \begin{cases} x \in I_{sleep}, & \sin(4\pi * (x - 0.125)) * 0.5 + 0.5 \\ x \notin I_{sleep}, & 1 \end{cases}$$

Die Berechnung des Arbeitsbedürfnis verläuft auf die gleiche Art und Weise. Hier wird lediglich der Offset in der Funktion anders berechnet, wie in Listing 6.22 zu sehen. In das Offset wird die Schlafdauer des Agenten mit einbezogen und ein zusätzliches Offset einberechnet. Der Agent soll idealerweise morgens aufwachen und nach kurzer Zeit einen Zuwachs seines Arbeitsbedürfnisses verspüren.

```
1 float sleepDuration = SleepActionPlan.SLEEP_DURATION_HOURS / 24f;
2 float baseOffsetX = -0.125f - sleepDuration - WORK_SLEEP_OFFSET;
```

Listing 6.22: Berechnung des Offset für das Arbeitsbedürfnis

Für das Arbeitsbedürfnis ergibt sich folgende Funktion:

$$g_{work}(x) = \begin{cases} x \in I_{work}, & \sin(4\pi * (x - O_{sw})) * 0.5 + 0.5 \\ x \notin I_{work}, & 0 \end{cases}$$

wobei $O_{sw} = -0.125f - sleepDuration - WORK_SLEEP_OFFSET$

O_{sw} beschreibt den Offset zwischen Werten der Schlaf- und der Arbeitsfunktion. $g_{work}(x)$ und $g_{sleep}(x)$ können nun berechnet werden. Sie können in Abbildung 6.24 betrachtet werden.

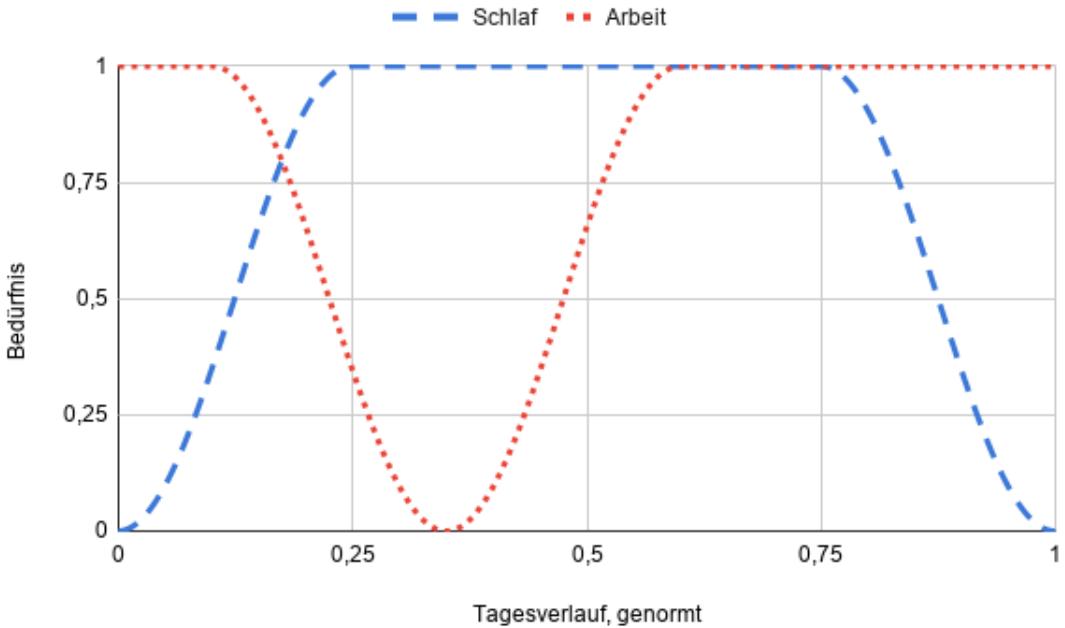


Abbildung 6.24.: Bedürfnisfunktionen des Agenten für Arbeit und Schlaf

Für die Berechnung der Schlaf- und Arbeitsfunktion wird weiterhin ein für jeden Agenten unterschiedlicher Wert verwendet. Bei der Initialisierung des Agenten wird dem Feld `_rythmVariation` ein zufälliger Wert zugewiesen, um jedem Agenten einen leicht versetzten Rhythmus zu verleihen. Die maximale Amplitude dieses Werts kann durch eine Anpassung der Variable `agentRythmVariation`

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

in der Klasse `GameAcademy` gesteuert werden. Ohne diesen Offset kann es dazu kommen, dass viele - oder alle - Agenten zur selben Zeit schlafen bzw. arbeiten gehen und sich identische Tagesabläufe abbilden.

Zusätzlich zu den genannten Bedürfnissen wurden ein *Recreation*-Wert und ein *Exhaustion* eingeführt. Der Recreation-Wert ist für das Auslösen von Freizeitaktivitäten verantwortlich. Er berechnet sich wie folgt:

```
1  public float calculateRecreationValue(Dictionary<string, float> needs
2      ) {
3      float recreationValue = 0f;
4      foreach (string needType in needs.Keys)
5      {
6          float need = needs[needType];
7          if (need < BaseActionPlan.NEED_PENALTY_THRESHOLD)
8              recreationValue += (1f - need) * 100;
9          else if (need < 0.3f) recreationValue += (1f - need) * 10f;
10         else if (need < 0.4f) recreationValue += (1f - need) * 3f;
11         else recreationValue += 1f - need;
12     }
13     recreationValue = recreationValue / needs.Count / 1.5f;
14     return Mathf.Clamp(recreationValue, 0f, 1f);
15 }
```

Listing 6.23: Berechnung des Recreationwerts

Recreation ist also abhängig von allen Grundbedürfnissen des Agenten. Je „glücklicher“ ein Agent ist, also je höher die Werte seiner Grundbedürfnisse sind, desto niedriger ist der Recreationwert. Sobald ein Agent hinsichtlich all seiner Bedürfnisse befriedigt ist, soll er durch einen dann niedrigen Recreationwert Freizeitaktivitäten nachgehen.

Exhaustion wird verwendet, um die Erschöpfung des Agenten zu simulieren. Der Exhaustionwert wird mit jeder *MoveAction* und durch das Arbeiten reduziert. Der Wert kann durch Ausrufen, Schlafen oder langes Warten wieder aufgefüllt werden.

6.4.6.4 Navigation

Während der `LocationHandler` für die Auswahl von Gebäuden zuständig ist und dahingehend Zielpositionen zur Verfügung stellt, ist der `MoveHandler` für die tatsächliche Navigation der Agenten verantwortlich. Dafür wird ein sogenannter *NavMeshAgent* verwendet [Tec17b]. Es handelt sich dabei um ein Feature, das lediglich in der kommerziellen Pro-Version von Unity nutzbar ist. Der *NavMeshAgent* nutzt ein *NavMesh*, das vorher generiert werden muss. Die Vorberechnung des NavMesh nennt sich *Baking*. Das NavMesh ist eine abstrakte Datenstruktur aus zweidimensionalen Polygonen, die für die Wegfindung des Agenten genutzt wird [Tec17a]. Es definiert Flächen, die von einem Agenten traversiert werden können. Um Wege zwischen den Polygonen des NavMesh zu finden, können klassische Pathfinding-Algorithmen wie A* verwendet werden. Durch die Verwendung performanter Algorithmen und der Vorberechnung des NavMesh verläuft das Pathfinding sehr effektiv. Im weiteren Verlauf wird angenommen, dass das gesamte NavMesh in der Simulation statisch ist. Kollisionen von Agenten mit der Umgebung kommen daher nicht vor. Weiterhin werden Kollisionen von Agenten untereinander ignoriert.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

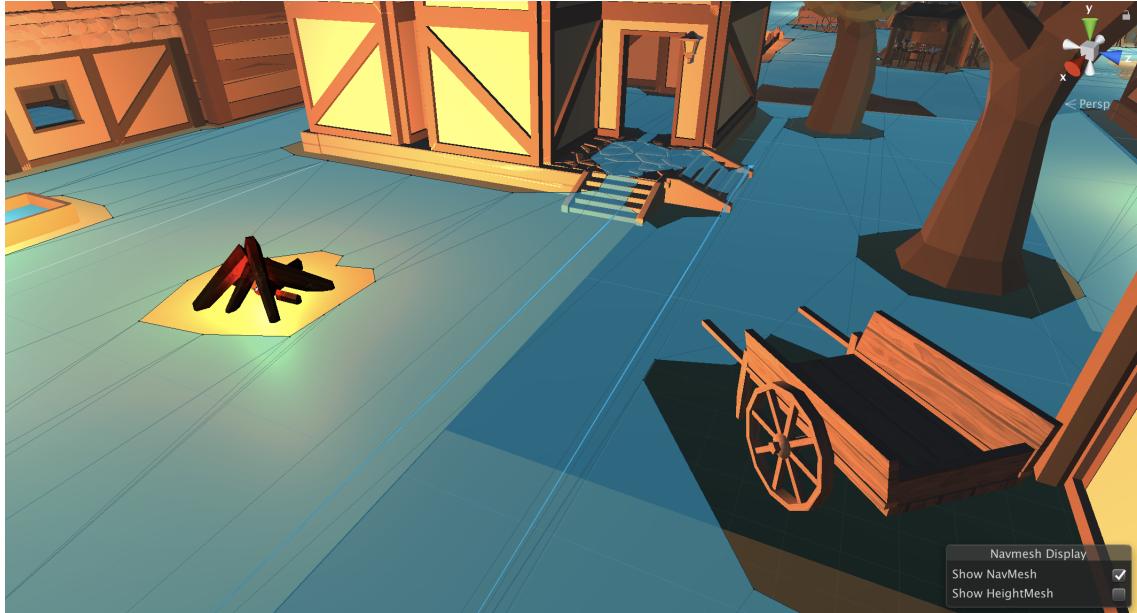


Abbildung 6.25.: Beispielhafter Ausschnitt des generierten NavMesh

Abbildung 6.25 zeigt einen beispielhaften Ausschnitt aus dem für die Dorfsimulation generierten NavMesh. Auf der Abbildung ist gut zu erkennen, dass nicht-begehbarer Objekte das NavMesh unterbrechen. Beispielsweise bilden der Karren im Vordergrund sowie die Bäume, die Häuser oder das Lagerfeuer Einschnitte in das NavMesh. Um das NavMesh zu erzeugen, muss für Objekte gekennzeichnet werden, ob sie in die Berechnung mit einbezogen werden oder nicht. Für die Generierung müssen weiterhin einige Parameter gesetzt werden, die die Maße des Agenten beschreiben, sodass Abstände korrekt berücksichtigt werden. Außerdem muss festgelegt werden, wie hoch der Agent springen und welche Steigungen er überwinden kann. Im hinteren Teil der Abbildung 6.25 ist dies anhand der Treppe gut zu erkennen. Das blaue NavMesh liegt ebenfalls über der Treppe, weil ihre Steigung vergleichsweise gering ist, sodass der Agent sie ohne weiteres erklimmen kann. Für die Berechnung des NavMesh wurde die Voxelgröße auf 0,125 festgelegt, was bei einem Agentenradius von 0,6 Einheiten 4,8 Voxeln pro Agentenradius entspricht. Dieser Parameter steuert effektiv die Auflösung des NavMesh. Bei der gewählten Größe benötigt die Berechnung, bzw. der *Baking*-Vorgang, für ein einzelnes Dorf etwa 30 Sekunden. Um das NavMesh zu nutzen, muss nach erfolgreicher Generierung des NavMesh die Komponente NavMeshAgent an den zu steuernden Agenten angefügt werden. Eine Menge von Einstellungen wie die Geschwindigkeit und Beschleunigung des Agenten können darin gewählt werden.

Der MoveHandler hat die Aufgabe, dem NavMeshAgent ein Ziel zu geben. Über die Funktion moveAgent, welche als Parameter unter anderem eine Zielposition erhält, wird ein Pfad zum ausgewählten Ziel berechnet und die Wegfindung des Agenten gestartet. Für jeden neuen Frame prüft anschließend die Update-Methode der Klasse, ob das Ziel des Agenten erreicht wurde. Sobald dies der Fall ist wird eine Callbackfunktion aufgerufen, welche die Move-Aktion des Agenten abschließt und die nächste auszuführende Aktion des entsprechenden ActionPlans startet. Die Abbildung 6.26 zeigt einige Agenten, die über das NavMesh navigieren.

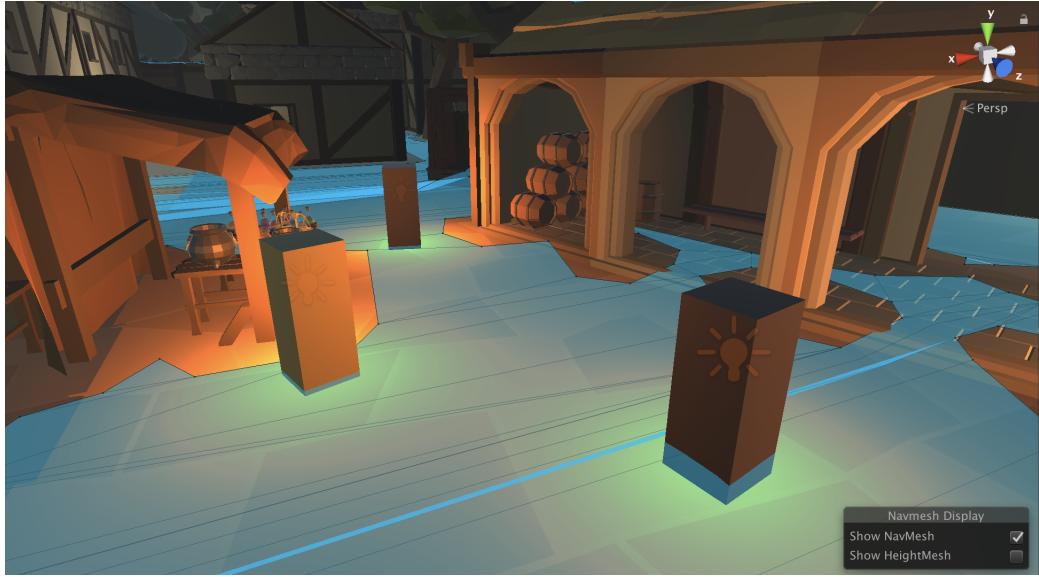


Abbildung 6.26.: Agenten navigieren über das NavMesh

6.4.6.5 Maskieren von Aktionen

Die Größe des Aktionsraum des Agenten ändert sich mit dem Simulationszustand und den Parametern des Agenten. Nicht zu jeder Zeit sind alle Aktionen ausführbar. Stattdessen ist jeder ActionPlan, den der Agent ausführen kann, an bestimmte Bedingungen geknüpft. ML-Agents bietet die Möglichkeit, Aktionen zu *maskieren*. Das bedeutet, dass diese Aktionen bei der Entscheidungsfindung nicht in Erwägung gezogen werden. Das neuronale Netz liefert also garantiert einen Output, bei dem die maskierten Aktionen nicht gewählt werden. Die Erzeugung der *ActionMask* ist im folgenden Listing 6.24 erkennbar.

```

1 _actionMask = new List<string>();
2
3 foreach (string actionId in NpcActions.ALL_ACTIONS) {
4     if (!_actionHandler.areRequirementsMet(actionId)) _actionMask.Add
5         (actionId);
6 }
```

Listing 6.24: Erzeugung der *ActionMask* für Aktionen

Im Abschnitt 6.4.6.1 wurde bereits beschrieben, wie Aktionen mit String-Identifiern verbunden werden. Die Maskierung wird im **ActionMaskingHandler** vorgenommen. Der Aufbau dieser Klasse ist in der folgenden Abbildung 6.27 zu erkennen:

ActionMaskingHandler
+ List<string> getStringActionMask
+ List<int> getIntActionMask
+ HashSet<string> _getMaskHashSet
+ List<int> _createIntMask
+ void _addJobMask

Abbildung 6.27.: Methoden der Klasse ActionMaskingHandler

Die eigentliche Erzeugung der ActionMask geschieht in der Funktion `_getMaskHashSet`. Der `ActionMaskingHandler` iteriert dort über die Identifier aller möglichen Aktionen. Über den Identifier der Aktion erhält dieser vom `ActionHandler` den jeweiligen ActionPlan und ruft dessen Funktion `areRequirementsMet` auf. Wenn eine oder mehrere Bedingungen nicht zutreffen, wird die Aktion maskiert, sodass sie nicht ausgeführt wird. Es wird ein `HashSet<string>` verwendet, welches wie ein Dictionary funktioniert, in dem jedem Schlüssel nur ein einzelner Wert zugewiesen werden kann. Maskierte Aktionen können daher nur einmal in der Kollektion vorkommen - dies wird von ML-Agents gefordert. Weiterhin wird für das Maskieren ein numerischer Identifier erwartet. Für spätere Verfahren werden hingegen Identifier in Form von `strings` erwartet. Daher wird ein umgekehrtes Mapping verwendet, das aus den String-Identifiern erneut numerische Identifier erzeugt. Weiterhin wird ein Wert vom Typ `List` statt `HashSet` erwartet. Für das Abfragen von integerbasierten oder stringbasierten ActionMasks gibt es daher die Funktionen `getStringActionMask` und `getIntActionMask`.

6.4.6.6 Ausführungsfluss des Agenten

Während der Durchführung von Aktionen interagiert der Agent mit einer Reihe verschiedener Komponenten. Die Architektur des Ausführungsflusses lässt sich anhand eines Komponentendiagramms beschreiben, wie in Abbildung 6.28 zu sehen.

Das Diagramm dient als ergänzende Beschreibung zusätzlich zur generellen Architekturübersicht in Abbildung 6.6 und zeigt daher Gemeinsamkeiten. Hier wird jedoch mehr ins Detail gegangen und es besteht ein näherer Bezug zum Agenten selbst.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

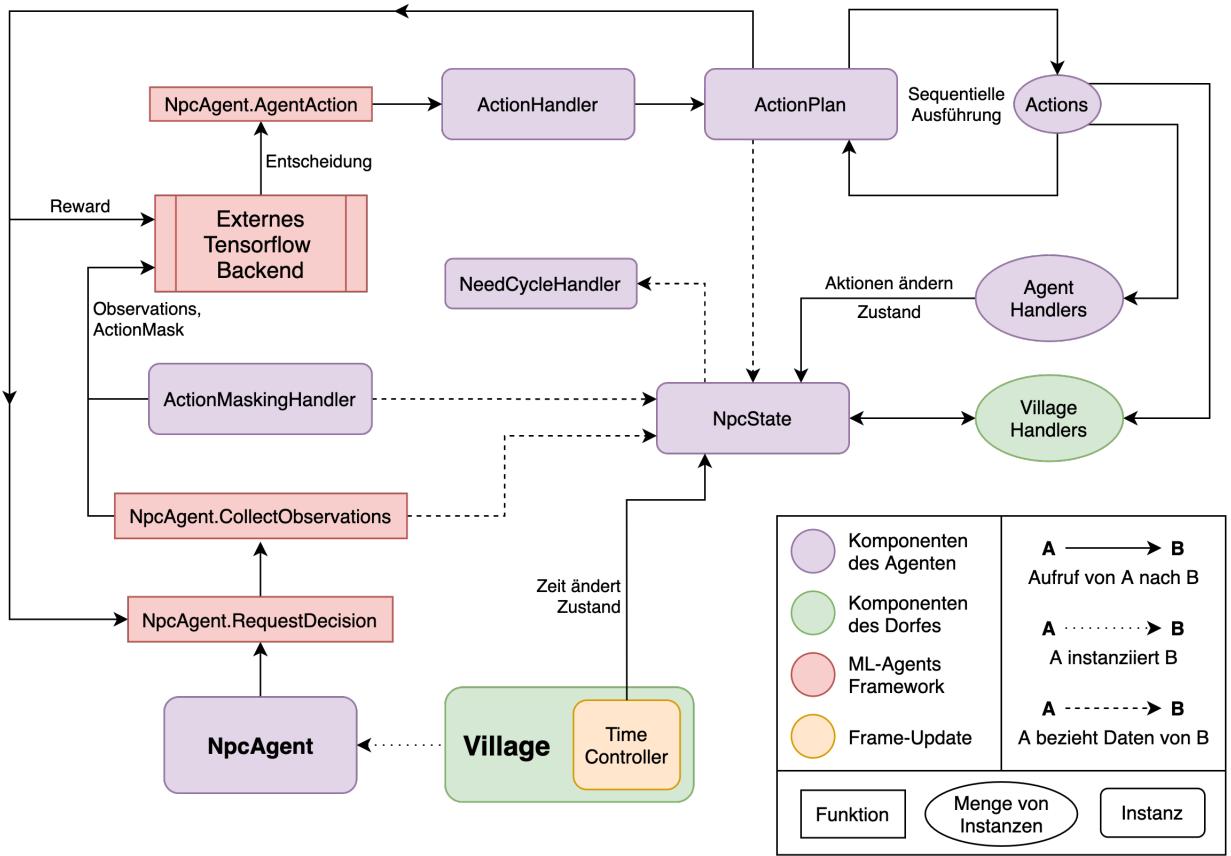


Abbildung 6.28.: Architektur der Agenten

Als erstes sammelt der NPC eine Menge von Observations. Für die Observations wird der **NpcState** konsultiert. Weiterhin wird der **ActionMaskingHandler** aufgerufen, um den Aktionsraum zu begrenzen. **ActionMask** und **Observations** werden dann über die Verbindung des ML-Agents-Frameworks zum externen Tensforflow-Backend geschickt. Als Antwort darauf erhält der Agent eine neue Aktion. Die Funktion **AgentAction** wird aufgerufen und stellt die neue Aktion bereit. Diese wird durch den **ActionHandler** behandelt, der den entsprechenden **ActionPlan** bereitstellt. Nun werden die einzelnen Sub-Aktionen des **ActionPlans** ausgeführt. In der Regel sind das *Move*- und *Wait*-Aktionen. Weitere Aktionen werden in diesem Diagramm vernachlässigt. Die *Move*-Aktionen werden durch den **MoveHandler** ausgeführt, der mit dem **LocationHandler** kommuniziert. Der **WaitHandler** führt die *Wait*-Aktionen aus. Wenn der Agent schließlich alle Sub-Aktionen ausgeführt hat, wird der **NpcState** aktualisiert. Aktionen haben Auswirkungen sowohl auf den Zustand des Agenten als auch unter Umständen auf den Zustand des Dorfes. Die neuen Bedürfnisse werden vom **NeedCycleHandler** berechnet. Ressourcen werden aktualisiert und Bedürfnisse angepasst oder zurückgesetzt. Außerdem wird der **MonitorHandler** mit neuen Werten aus dem **NpcState** aktualisiert, um im HUD der Simulation aktuelle Werte anzuzeigen. Außerdem wird eine Belohnung bzw. Bestrafung für die Aktion verteilt. Über eine Callbackfunktion im **ActionPlan** schließt sich der Ausführungskreislauf. Eine neue Aktion wird durch den Agenten gestartet, indem **RequestDecision** aufgerufen wird. Zusätzlich zu diesem Ausführungsablauf wird in regelmäßigen Abständen die Update-Funktion der Komponente **TimeController** des Moduls **Village** aufgerufen, welche die Zustände aller Agenten verändert, indem sich durch das Voranschreiten der Zeit deren Bedürfnisse verändern.

6.4.6.7 Belohnungen

Die Belohnungen des Agenten hängen grundsätzlich von seinen Bedürfnissen und dem Zweck der Aktion ab. Sie werden jeweils am Ende einer Aktionen verteilt und in den ActionPlans definiert. Jeder ActionPlan ist mit einem bestimmten Bedürfnis des Agenten verbunden, das durch ihn befriedigt wird. Die Berechnung der Belohnung berücksichtigt, wie ausgeprägt dieses bestimmte Bedürfnis zum Zeitpunkt der Ausführung war. Wenn der Agent beispielsweise sehr hungrig ist, also etwa einen Hungerwert $g_{\text{hunger}} = 0$ hat und eine Aktion *Essen* auswählt, welche das Hungergefühl stillt, so wird er eine hohe Belohnung bekommen. Wenn er bereits vollkommen gesättigt ist ($g_{\text{hunger}} = 1$) und die Aktion *Essen* auswählt, wird er eine Bestrafung erhalten. Der Agent soll dadurch lernen, Aktionen zu wählen, die seine Bedürfnisse sinnvoll befriedigen. Wenn alle Bedürfnisse befriedigt sind hat der Agent einen niedrigen Recreationwert, sodass er Freizeitaktivitäten nachgehen kann. Die Belohnungen funktionieren hier auf eine ähnliche Art und Weise. Dies ist ebenfalls beim Arbeits- und Schlafbedürfnis der Fall, obwohl diese Bedürfnisse nicht durch die Aktion aufgefüllt werden, sondern von der Tageszeit abhängen.

Die Belohnungen bestimmter Aktionen hängen außerdem von den Attributen der Agenten ab. Um den Agenten eine Persönlichkeit zu verleihen, werden Attribute wie *Intelligence*, *Companionship* oder *Fitness* im *NpcState* festgelegt. Dadurch sollen allgemeine Unterschiede zwischen den Agenten, aber auch Unterschiede zwischen verschiedenen Gruppen von Agenten entstehen. Diese Gruppen werden durch die Berufe der Agenten bestimmt. Grundsätzlich werden die Attribute zufällig verliehen, zusätzlich wird bestimmten Berufen die Tendenz gegeben, bei bestimmten Attributen durchschnittlich höhere Werte zu erzielen als andere Berufe. Die Verleihung von Attributen aufgrund des Berufs spiegelt nicht die realen Welt, sondern wurde willkürlich festgelegt. Die Festlegung dieser Attribute wird in der statischen Klasse *AttributeHandler* durchgeführt. Die statische Methode *getAttributesByJobId* erhält als Parameter einen Identifier für den Job des Agenten und gibt ein *Dictionary<string, float>* mit den jeweiligen Attributen zurück. In einer *switch-case*-Anweisung werden die Attribute aufgrund des Berufs vergeben. Die Funktion wird vom *NpcState* aufgerufen. Beispielsweise wird im ActionPlan *RecreationTavernActionPlan* eine Freizeitaktion definiert, die Agenten besuchen dabei die Taverne. Die Belohnung der Aktion skaliert invertiert mit dem Fitness- und Intelligenzlevel des NPC. Je niedriger die Fitness- und Intelligenzattribute des Agenten sind, desto höher fällt die Belohnung aus. So kommt es dazu, dass man mehr Minenarbeiter als Jäger in der Taverne antrifft, weil diese durchschnittlich über einen niedrigen Fitnesslevel sowie einen leicht niedrigeren Intelligenzlevel verfügen. Jäger werden mit einem überdurchschnittlich hohen Fitnesslevel häufiger auf dem Fußballplatz statt in der Taverne angetroffen. Beim Training werden diese Gegebenheiten durch die Veränderung des Rewards automatisch in den Lernvorgang mit einbezogen.

Für die Festlegung von Belohnung werden unter anderem die Hilfsfunktionen *_getNeedReward*, *_getRecreationReward*, *_getNeedReward* der Klasse *BaseActionPlan* und die Funktion *_getWorkReward* der Klasse *WorkActionPlan* verwendet. Mithilfe von Funktionen wie *_calculateAttributeMultReward* werden Multiplikatoren für die Attribute berechnet, um die Belohnungen negativ oder positiv zu beeinflussen. Weiterhin wird für sämtliche Belohnungen, die am Ende einer Aktion ausgeführt wurden, stets die Werte des Zustands des Agenten verwendet, die zum Anfang der Ausführung der Aktion vorhanden waren. Dadurch wird verhindert, dass der berechnete Reward direkt mit den vorher gesammelten Observations des Agenten übereinstimmen und eine Beziehung aufgebaut werden kann.

6.4.7 VillageAgent

Den Agenten werden bei der Initialisierung Berufe zugewiesen. Für die Behandlung der Berufe der Agenten können mit der Anpassung des Parameters `ignoreJobMask` der Klasse `GameAcademy` zwei Modi gewählt werden. Der Parameter wurde eingeführt da sich beim Training herausgestellt hat, dass die Wahl der beruflichen Aktionen der Agenten eine Schwierigkeit darstellt.

Wenn der Parameter gesetzt ist wird der Beruf des Agenten ignoriert. Der Aktionsraum eines jeden Agenten verfügt dann über alle möglichen Berufsaktionen. Es wird also jeweils die berufliche Aktion gewählt, die für den Agenten voraussichtlich die höchste zukünftige Belohnung zuweist. Dadurch soll gewährleistet werden, dass das Ressourcensystem in einem stabilen Zustand gehalten wird, da das Ressourcenmanagement durch die Auswahl der beruflichen Aktivitäten der Agenten automatisch in einer Balance gehalten wird. Wenn von einer benötigte Ressource nur noch geringe Mengen vorhanden sind, sollten Agenten automatisch den Beruf ausführen, der die entsprechende Ressource herstellt. Die Agenten müssen in diesem Fall also Observations über die vorhandenen Ressourcen sammeln.

Wenn der Parameter `ignoreJobMask` nicht gesetzt wird werden hingegen alle Jobs, die nicht dem Job des Agenten entsprechen, permanent maskiert und deren Ausführung dadurch blockiert. Der Agent hat dann also einen festen Job und wird von allen weiteren Jobs ausgeschlossen. Vor dem Trainingsvorgang ist allerdings nicht bekannt, wie viele Agenten welchen Job ausführen müssen, um eine Balance im Ressourcensystem herzustellen, wie oben beschrieben. Die Agenten müssen in diesem Fall keine Observations über die vorhandenen Ressourcen sammeln, da ohnehin keine Auswahl des Jobs möglich ist.

Aus diesem Grund wird neben dem `NpcAgent` ein weiterer RiL-basierter Agent verwendet, der ML-Agents nutzt und sich darum kümmern soll, den Agenten in Abhängigkeit von den vorhandenen Ressourcen Jobs zuzuweisen. Die Implementierung ist in der Klasse `VillageAgent` zu finden. Die Aufgabe dieses Agenten ist es, den NPCs in Abhängigkeit von den vorhandenen Ressourcen in der Simulation Jobs zuzuweisen. Der Agent hat keinerlei visuelle Präsentation im Spiel, sondern nimmt eine Art Beobachterrolle ein. Nach einer gewissen Anzahl von Aktionen der Agenten wird durch die Academy automatisch eine neue Aktion des `VillageAgent` mittels `RequestDecision` gestartet. Der `VillageAgent` analysiert die Ressourcen und ermittelt, welche Ressource am wenigsten vorhanden ist. Weiterhin prüft er, welche Ressource am meisten vorhanden ist. Durch ein Mapping von den Ressourcen zu den dazugehörigen Jobs werden anschließend neue Jobs zugewiesen, wie in Listing 6.25 zu sehen. Der Agent verwendet anders als der `NpcAgent` insgesamt elf kontinuierliche Aktionen. Statt 24 diskreten, auswählbaren Aktionen wird hier für jede der elf Ressourcen immer ein kontinuierlicher Wert geliefert.

```

1  public override void AgentAction(float[] vectorAction, string
2      textAction) {
3      float smallestValue = 1;
4      int smallestIndex = -1;
5      float largestValue = -1;
6      int largestIndex = -1;
7      for (int i = 0; i < vectorAction.Length; i++) {
8          float value = vectorAction[i];
9          if (value < smallestValue) {
10              smallestValue = value;
11              smallestIndex = i;
12          } else if (value > largestValue) {
13              largestValue = value;
14              largestIndex = i;
15          }
16      }
17      // Don't do anything, no change needed.
18      if (largestValue < 0.3f && smallestValue > -0.3f) return;
19
20      string negJob = _indexToJob[smallestIndex];
21      string negResource = _jobToResource[negJob];
22      string posJob = _indexToJob[largestIndex];
23      string posResource = _jobToResource[posJob];
24
25      _assignNewWorkerJob(negJob, posJob);
26      _addRewardsNew(negJob, posJob);
27  }

```

Listing 6.25: Aktionen des `VillageAgent`

Wenn die Differenz groß genug ist, wird dem Beruf, der am meisten Ressourcen produziert hat, ein `NpcAgent` entzogen und dem Beruf, der am wenigsten produziert hat, hinzugefügt. So soll automatisch eine Balance gefunden werden, da die Berufe nicht alle gleich viele Ressourcen produzieren. Die kontinuierlichen Werte, die im Parameter `vectorAction` enthalten sind, beschreiben also, in welchen Berufen zu viele und in welchen Berufen zu wenige Agenten arbeiten. Für die Belohnung des Agenten werden die durchschnittlichen Abweichungen der jeweiligen Ressourcen für den hinzugefügten und entfernten Job verwendet. Wenn einem Agent ein Job zugewiesen wird, in dem aktuell nur wenige Ressourcen vorhanden sind, so resultiert dies also in einer hohen Belohnung. Eine Zuweisung in einen Beruf mit bereits hohen verbundenen Ressourcenwerten resultiert in einer Bestrafung. Der `VillageAgent` wird durch die Funktion `IncrementNumAgentActions` in der Klasse `GameAcademy` zurückgesetzt. Für jede Aktion eines Agenten wird diese Funktion aufgerufen und inkrementiert einen Zähler. Wenn dieser Zähler einen bestimmten Wert erreicht, der von den Parametern der `GameAcademy` abhängt, werden der `VillageAgent` und die Berufe aller Agenten zurückgesetzt.

6.4.8 UI

Um Informationen zur Simulation an einem zentralen, leicht zugänglichen Ort anzeigen zu lassen, wurden verschiedene UI-Elemente entwickelt. Im Folgenden werden diese erläutert.

Dorfstatistiken Für das UI wird die Klasse `Monitor` aus dem ML-Agents Framework verwendet. Auf dieser Grundlage wurde eine eigene Klasse `GameMonitor` erstellt. Sie bietet eine einfache Möglichkeit, Informationen als UI-Elemente anzeigen zu lassen und erweitert die Klasse `Monitor`

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

um einige anwendungsspezifische Funktionen. Bei den verwendeten UI-Elementen handelt es sich um Textelemente und Balken. Der visuelle Aspekt spielt hier keine wesentliche Rolle, sondern die Vermittlung von Informationen. Zusätzlich wurden in der Basisklasse `Monitor` einige Änderungen vorgenommen, um die Darstellung von Informationen anzupassen und für das zugrunde liegende Szenario zu verbessern. Die nötigen Informationen werden in der Klasse `VillageInfo` gesammelt. Zum Beispiel wird immer dann, wenn ein Agent eine Aktion ausführt, ein entsprechender Zähler in der Klasse verändert. Im UI wird für alle möglichen Aktionen ein Textfeld eingeblendet, das die jeweilige Anzahl der Agenten beschreibt, die in diesem Moment die Aktion ausführen. Zusätzlich werden Werte wie Ressourcen und die aktuelle Zeit dargestellt.

Agentenspezifische Informationen Während der Ausführung der Simulation kann auf Agenten geklickt werden, um im UI agentenspezifische Informationen anzeigen zu lassen. Dafür muss eine bestimmte Kamera ausgewählt werden, die über die Komponente `OrbitCameraBehaviour` verfügt. Diese verfügt über ein Script, das eine Steuerung der Position und Rotation der Kamera über Tastatureingaben ermöglicht [Unb10]. Das Script wurde so modifiziert, dass es nun möglich ist, innerhalb der Simulation NPCs anzuklicken. Dieses Verhalten wird durch die Klasse `InteractionHandler` geregelt. Die Kamera wird dann mit dem Transform des Agenten verbunden, sodass diese ihm folgt und man die virtuelle Welt aus den Augen des NPCs betrachtet. Weiterhin wird in der Klasse `NpcAgent` der Boolesche Wert `_shouldMonitor` gleich `true` gesetzt. Eine Instanz der Klasse `MonitorHandler` prüft in regelmäßigen Abständen den Zustand dieser Variable. Dadurch werden weitere UI-Elemente aktiviert, die für diesen spezifischen Agenten Informationen zu seinen Bedürfnissen, der aktuellen Aktion etc. anzeigt. Alternativ kann im Editor der Wert `_shouldMonitor` für beliebige Agenten direkt beeinflusst werden, um bestimmte Informationen anzuzeigen. Jeder Agent besitzt die beiden Komponenten `MonitorHandler` und `InteractionHandler`. In der Abbildung 6.29 ist ein Beispiel für das im Spiel angezeigte UI aus der Perspektive eines Agenten erkennbar.



Abbildung 6.29.: UI aus Sicht eines Agenten

Anpassung der Szene Weiterhin wurden UI-Elemente entwickelt, um Anpassungen in der Szene vorzunehmen. Sämtliche Dächer der Gebäude im Dorf können über einen Agenten deaktiviert und aktiviert werden. Dadurch kann besser gesehen werden, wo die Agenten sich aufhalten. Zusätzlich

können weitere Mesh-Elemente deaktiviert werden, um die Performance der Simulation zu erhöhen oder einen besseren Überblick zu bekommen. Konkret lässt sich das Mesh von Bäumen, Zäunen, Gebäuden und Felsen über einen Button deaktivieren. Zusätzlich können sämtliche Lichter in der Stadt deaktiviert werden. Dabei wird zwischen den Lichtquellen der Agenten und den Lichtquellen der Stadt unterschieden. Weiterhin besteht die Möglichkeit, über einen Dropdown-Menü die Wahl der aktiven Kamera anzupassen. Dadurch kann ebenfalls die Spielerkamera aktiviert werden, sodass der *Player* gesteuert werden kann (siehe Abschnitt 6.4.11). Diese Anpassungen werden in der Klasse `GameUI` vorgenommen. Die Klasse steuert die entsprechenden UI-Elemente bzw. nimmt Befehle von ihnen entgegen. Bei einer Verwendung mehrerer Dörfer wird nicht zwischen einzelnen Dörfern unterschieden. Die Änderungen treten stets für die gesamte Szene in Kraft. Die Komponente `GameUI` ist ein MonoBehavior und wird schon vor der Ausführung an ein GameObject angehängt. Innerhalb der Funktion `Start` werden Meshelemente und Lichtquellen innerhalb der Szene gesucht. Anschließend wird durch die Klasse `GameAcademy` die Funktion `initAgents` aufgerufen, nachdem die Initialisierung der Dörfer und Agenten abgeschlossen ist. Die Funktion `initAgents` bekommt eine Liste aller Agenten als Parameter und bezieht daraus deren Lichtquellen.

6.4.9 Logging

Für eine erleichterte Analyse der Simulation wird eine Menge von Daten mitgeschrieben. Informationen zu den Aktionen der Agenten werden gesammelt, die für die Auswertung der trainierten Policy relevant sind. Dabei werden keine Daten in Dateien geschrieben, sondern in Dictionaries gespeichert, die durch Debugging ausgelesen werden können. Neben den für die Darstellung im UI gesammelten Daten werden zusätzlich die Dauer von Aktionen sowie deren durchschnittliche Dauer mitgeschrieben (`VillageInfo.addActionDuration`). Außerdem werden die akkumulierten Belohnungen sowie die durchschnittlichen Belohnungen für getätigte Aktionen gespeichert (`ActionHandler.logReward`). Zusätzlich wird für alle Aktionen eine `Queue` gespeichert, sodass die letzten 100 konkreten Belohnungen abgefragt werden können. In der Klasse `VillageInfo` werden außerdem die durchschnittlichen Attribute der Agenten für die jeweiligen Aktionen gespeichert. Die `ActionMasks` können darüber hinaus nachvollzogen werden, indem sie im `ActionMaskingHandler` mitgeschrieben werden.

Um den Tagesablauf der Agenten anhand ihrer Bedürfnisse sichtbar zu machen, musste weiterhin ein Weg gefunden werden, die Bedürfnisse der Agenten zu bestimmten Zeitpunkten zu speichern. In der Klasse `Village` werden in der Funktion `updateTime` immer dann, wenn bestimmte zeitliche Thresholds überschritten wurden, die Bedürfnisse aller Agenten in einer verschachtelten Datenstruktur aus `List` und `Dictionary` hinterlegt. Durch die Nutzung eines Debuggers kann ein Schreibvorgang gestartet werden, der einen String aus der Datenstruktur erstellt und die enthaltenen Daten in eine CSV-Datei schreibt. Dieser Vorgang wurde verwendet, um die im nachfolgenden Kapitel erkennbaren Abbildungen 7.16 und 8.3 zu erstellen. Die Granularität der Datenpunkte kann im `TimeController` durch eine Anpassung der Konstante `NEED_LOG_TIME_INTERVAL` angepasst werden. Standardmäßig wurde der Wert so gewählt, dass nach jeweils zehn im Spiel vergangenen Minuten ein Logpunkt erstellt wird.

Das Logging kann in den Parametern der `GameAcademy` aktiviert werden (siehe Abschnitt 6.4.2).

6.4.10 Fake Inference

Mit der Erhöhung des TimeScale-Faktors werden alle physikalische Vorgänge, die in der Engine beschrieben werden, ungenauer, da für jedes Update relativ gesehen mehr Zeit vergeht. Simuliert man beispielsweise den Wurf eines Balls und verzehnfacht die TimeScale, so werden während dieses Wurfes bei gleichbleibender Framerate nur ein Zehntel so häufig die Update-Funktionen aufgerufen.

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

Für die Navigation des Agenten über das NavMesh stellt dies ein essentielles Problem dar. Besonders die Navigation in engen Bereichen wird dadurch erschwert. Das hat zur Folge, dass NavMeshAgents mit steigender Timescale Türen verfehlten. Man stelle sich vor, dass ein Agent links von einer Tür steht und hindurch gehen möchte. Dazu muss er ein Stück weit nach rechts gehen. Wenn die Timescale zu hoch ist, so hat der Agent sich unter Umständen bereits beim Aufruf der nächsten Update-Funktion so weit nach rechts bewegt, dass er nun rechts von der Tür steht. Der Agent fängt nun an, zu oszillieren und schafft es nicht, die Tür zu durchschreiten. Durch eine Anpassung von Geschwindigkeit, Beschleunigung und Drehgeschwindigkeit kann dieses Phänomen nur bis zu einem bestimmten Grade unterbunden werden. Weiterhin wirken die Bewegungen der Agenten dann möglicherweise bei einer normalen Timescale falsch.

Aus diesem Grund wurde der sogenannte *Fake Inference*-Modus entwickelt, in dem keinerlei Bewegung der Agenten durch das NavMesh vorkommt. Die **MoveAction** agiert dann auf andere Weise und ruft nicht mehr den **MoveHandler** für die Navigation auf. Stattdessen wird die Zeit berechnet, die ein Agent normalerweise bei seiner durchschnittlichen Geschwindigkeit benötigen würde, um den Weg zu einem Ziel zu beschreiten. Mithilfe des NavMesh findet im **LocationHandler** die Berechnung des Pfades statt, den der NavMeshAgent normalerweise wählen würde. Diese Berechnung wurde in der Funktion **calculatePathLength** implementiert, die in Listing 6.26 zu erkennen ist.

```

1  public IEnumerator calculatePathLength(Vector3 origin, Vector3
2      destination, NavMeshAgent navMeshAgent, int agentId) {
3      yield return Ninja.JumpToUnity;
4      NavMeshPath path = new NavMeshPath();
5      navMeshAgent.CalculatePath(destination, path);
6      yield return Ninja.JumpBack;
7
8      float pathLength = 0f;
9      Vector3[] wayPoints = new Vector3[path.corners.Length + 2];
10     wayPoints[0] = origin;
11     wayPoints[wayPoints.Length - 1] = destination;
12
13     for (int i = 0; i < path.corners.Length; i++) {
14         wayPoints[i + 1] = path.corners[i];
15     }
16     for (int i = 0; i < wayPoints.Length - 1; i++) {
17         pathLength += Vector3.Distance(wayPoints[i], wayPoints[i +
18             1]);
19     }
20     _calculatedPathLengths[agentId] = pathLength;
21     yield return null;
22 }
```

Listing 6.26: Berechnung der Pfadlänge durch **calculatePathLength**

Hierfür muss eine Coroutine verwendet werden, da die Berechnung eines Pfades über das NavMesh kein unmittelbares Ergebnis liefert. Die Funktionen **Ninja.JumpToUnity** und **Ninja.JumpBack** aus der *Thread Ninja*-Bibliothek [Spi14] müssen verwendet werden, da in C#-Coroutinen normalerweise kein Unity-Code verwendet werden können. Durch diese Anweisung findet ein Sprung aus dem Thread heraus in den Unity-Kontext und wieder zurück statt. **navMeshAgent.CalculatePath** berechnet dann den eigentlichen Pfad mit dem Typen **NavMeshPath**. Alle Wegpunkte des Pfades inklusive dem Start- und Zielort werden anschließend in ein Array geschrieben. Anschließend wird für jeden Wegpunkt die jeweilige Distanz zum nächsten Wegpunkt durch **Vector3.Distance** ermittelt und in einer Variablen aufaddiert. Da kein anderer Wert als **IEnumerator** zurückgegeben werden kann, wird

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

das Ergebnis in `_calculatedPathLengths` gespeichert und kann anschließend durch die Funktion `getCalculatedPathLength` abgerufen werden.

In der `GameAcademy` kann festgelegt werden, ob der normale Inference-Modus oder der Fake Inference-Modus verwendet werden soll. Entsprechend ändert sich der in der statischen Variable `GameAcademy.GAME_MODE` festgelegte Spielmodus. Dieser Wert wird in der Klasse `MoveAction` geprüft. Dafür werden die Methoden `_handleInference` und `_handleFakeInference` gebraucht. Diese wird im Listing 6.27 beschrieben.

```

1  private IEnumerator _handleFakeInference(Vector3 destination) {
2      yield return _moveHandler.StartCoroutine(_locationHandler.
3          calculatePathLength(_npc.transform.position, destination,
4              _moveHandler.NavAgent, _npc.AgentId));
5      float pathLength = _locationHandler.getCalculatedPathLength(_npc.
6          AgentId);
7      float speed = _moveHandler.NavAgent.speed;
8      if (pathLength < 1f) _duration = 1f;
9      else _duration = (pathLength / speed) *
10         PATH_LENGTH_DURATION_MULTIPLIER;
11
12     _npc.transform.position = destination;
13
14     WaitAction waitAction = new WaitAction(_actionType, _npc,
15         _duration);
16     _npc.StartCoroutine(waitAction.execute(_finishedMoving,
17         _cancelCallback));
18 }
```

Listing 6.27: Behandlung der Fake Inference in der Klasse `MoveAction`

Im Fake Inference-Modus wird zunächst die Länge des Pfades ermittelt, wie bereits oben beschrieben. Schließlich wird die Länge des Pfades durch die Maximalgeschwindigkeit des Agenten geteilt, um die ideal nötige Dauer zu erhalten. Diese wird mit einem Faktor multipliziert, der empirisch ermittelt wurde und mit ausreichender Genauigkeit ähnliche Ergebnisse liefert, wie durch die tatsächliche Navigation eines NavMeshAgent. Der Agent wird anschließend zum Zielort teleportiert. Die berechnete Dauer, die eine Navigation zum Zielort in Anspruch nehmen würde, wird nun genutzt, um stattdessen eine `WaitAction` zu erzeugen. Der Agent wartet also so lange, wie die Aktion normalerweise benötigen würde. Da die Navigation der Agenten zum Ziel lediglich aus visuellen Gründen nötig ist, stellt dieses Verfahren einen legitimen Ersatz dar und beeinflusst nicht die Entscheidungsfindung des Agenten.

Nutzung des Fake Inference-Verfahrens für den Trainingsfall Das gleiche Verfahren wird ebenfalls für das Training des Agenten angewendet, da hier mit sehr hohen TimeScale-Faktoren gearbeitet wird, um möglichst schnell Ergebnisse zu erhalten. Im Trainingsmodus wird dann ebenfalls die Funktion `_handleFakeInference` aufgerufen. Eine Ausnahme besteht jedoch, wenn mit lediglich einem einzelnen Agenten gearbeitet wird. Normalerweise regelt der `TimeController` das Voranschreiten der Zeit durch den Aufruf einer `Update`-Funktion. Wenn nur ein Agent verwendet wird, kann eine drastische Beschleunigung erzielt werden, indem der Agent selbst die Zeit voranschreiten lässt. Wie bei der Fake Inference wird jede `MoveAction` des Agenten in eine `WaitAction` umgewandelt, die auf der Länge des theoretischen Pfades basiert. Anstatt zu warten, werden alle `WaitActions` des Agenten allerdings direkt übersprungen. Der Zeit des `TimeControllers` schreitet äquivalent zur Dauer der `WaitAction` voran. Dadurch muss nicht auf die `WaitAction` gewartet werden und der `TimeController` berechnet trotzdem korrekte Werte für die Bedürfnisse des Agenten, die nun für

die Observations einer neuen Entscheidung genutzt werden können. Anstatt Aktionen auszuführen, werden diese übersprungen und die Zeit wird künstlich vorgespielt, um sofort eine weitere Aktion auszuführen. Dadurch wird der Intervall zwischen der Ausführung von Aktionen extrem verringert. Der große Nachteil dieser Methode ist jedoch, dass nur ein einzelner Agent verwendet werden kann, da mehrere Agenten schon nach der ersten Aktion nicht mehr synchron zueinander wären bzw. die Zeit zu weit voranschreiten lassen würden. Jeder Agent bräuchte dann seinen eigenen `TimeController`, was den Sinn einer Dorfgemeinschaft zunichte machen würde.

6.4.11 Player

Um die erreichte Immersion der Simulation zu beurteilen, wurde ein einfacher spielbarer Charakter implementiert. Für dessen Steuerung wird eine Instanz der bereits in Unity vorhandenen Klasse `CharacterController` genutzt, die die Bewegung eines Charakters erlaubt. Dafür werden Tastatureingaben genutzt, die dem `CharacterController` zur Verfügung gestellt werden müssen. Dafür wird die Klasse `PlayerController` verwendet. Die Implementierung der Klasse ist in Listing 6.28 zu sehen.

```

1  public class PlayerController : MonoBehaviour {
2      public CharacterController characterController;
3      private float _lastYPos;
4
5      private void Start() {
6          characterController.detectCollisions = true;
7          _lastYPos = transform.position.y;
8      }
9
10     private void Update() {
11         float moveInput = Input.GetAxis("Vertical");
12         float turnInput = Input.GetAxis("Horizontal");
13         float currentYPos = transform.position.y;
14         bool alreadyJumping = !Mathf.Approximately(_lastYPos,
15             currentYPos);
16
17         float moveSpeed = 0.25f;
18         if (Input.GetKey(KeyCode.LeftShift)) moveSpeed = 0.65f;
19         else if (Input.GetKey(KeyCode.LeftAlt)) moveSpeed = 0.1f;
20
21         Vector3 move = transform.TransformDirection(Vector3.forward *
22             moveInput * moveSpeed);
23         if (Input.GetKey(KeyCode.Space) && !alreadyJumping) move.y =
24             4f;
25         else move.y -= 9.81f * Time.deltaTime;
26
27         characterController.Move(move);
28         transform.Rotate(0, turnInput * 2.5f, 0);
29         _lastYPos = currentYPos;
30     }
31 }
```

Listing 6.28: Implementierung des `PlayerController`

Die Inputs werden kontinuierlich in der Funktion `Update` durch eine Abfrage von `Input.GetAxis("Vertical")` und `Input.GetAxis("Horizontal")` geprüft und dem `CharacterController` zur Verfügung gestellt, um den Spieler zu bewegen. Die verwendeten Achsen und die mit ihnen assoziierten Eingabe-

6. Entwicklung bedürfnisbasierter Agenten durch Reinforcement Learning

tasten können im *Input*-Fenster in den Projekteinstellungen des Unity-Editors verändert werden. Klassischerweise werden die Tasten *W*, *A*, *S* und *D* verwendet. Zusätzlich werden die Tasten *Alt*, *Shift* und *Space* genutzt, um den Spieler langsamer oder schneller laufen zu lassen bzw. einen Sprung auszuführen.

Aus den einzelnen Inputs wird ein Vektor konstruiert, der die Bewegung des Charakters durch `characterController.Move(move)` herbeiführt. Weiterhin wird der Spieler durch `transform.Rotate(0, turnInput * 2.5f, 0)` gedreht.

6.5 Start und Bedienung der Dorfsimulation

Für die Nutzung der Simulation muss Unity installiert werden [Tec20]. Eine der Versionen *2019.2.x* ist zwingend erforderlich. Idealerweise sollte die Version *2019.2.8f1* verwendet werden - für alle weiteren Versionen *2019.2.x* wird ansonsten eine automatische Migration des Projekts durch Unity durchgeführt. Im Rahmen dieser Thesis wurde eine Migration des Projektes nicht überprüft oder getestet, sodass Nebenerscheinungen auftreten können. Das Projekt, das die Simulation beinhaltet, ist auf dem mitgelieferten Speichermedium oder im entsprechenden GitHub-Repository zu finden [Jan20]. Für die Einstellung bestimmter Features wie dem NavMesh muss Unity Pro installiert sein. Diese Features können auch mit der normalen Version verwendet, aber nicht verändert werden. Die wichtigsten Parameter zur Einstellung der Simulation (siehe Abschnitt 6.1) können in der `GameAcademy` angepasst werden, die in einem GameObject mit dem Titel *Academy* als Komponente zu finden ist. Weiterhin können den Agenten und das Dorf betreffende Einstellungen in Prefabs mit den Namen *Agent* und *Village* angepasst werden. Dafür werden die Komponenten `NpcAgent` und `Village` verwendet.

7

Training und Ergebnisse

Im folgenden Kapitel werden die Trainingsvorgänge der Agenten beschrieben und die Ergebnisse vorgestellt und diskutiert.

7.1 Optimierung der Trainingsumgebung

Verschiedene Optimierungen wurden vorgenommen, um die Performance der Dorfsimulation zu erhöhen und sowohl den Trainingsvorgang als auch die Framerate im Inference-Modus zu verbessern.

Training Für das Training wurde ein Rechner mit einer Intel i5-8400 CPU mit sechs Kernen und 16 GB RAM verwendet. Für Testdurchläufe wurde das Training im Unity Editor durchgeführt, für längere Trainingsvorgänge wurden Unity-Builds des Projekts ausgespielt, um diese zu beschleunigen. Für das Training wurden gleichzeitig sechs Dörfer verwendet, in denen jeweils 50 Agenten erzeugt wurden. Dadurch sollten die sechs Kerne der CPU sinnvoll belastet werden. Empirisch wurde ermittelt, dass die Verwendung mehrerer Dörfer keine Beschleunigung des Vorgangs zur Folge hatte. Um das Training weiter zu beschleunigen, wird automatisch sämtliches irrelevantes Mesh der Szene deaktiviert, wie in Abschnitt 6.4.8 beschrieben. Das bringt besonders für das Training innerhalb des Unity-Editors Geschwindigkeitsvorteile. Für die Erzeugung der Unity-Builds wird ein sogenannter *Headless-Modus* oder *Server-Modus* verwendet. Damit wird eine ausführbare Version des Projekts erzeugt, die sich besonders für die netzwerkbasierte Anwendungen eignet, die auf einem Server laufen. In diesem Fall stellt die Anwendung lediglich die Ausgabe der Unity-Konsole dar. Die Darstellung der Szene fällt gänzlich weg. Es müssen keine Texturen geladen werden und es findet keinerlei Rendering statt. Dadurch ergibt sich ein Geschwindigkeitsvorteil durch eine geringere Belastung der Hardware. Weiterhin können etwaige Fehlermeldungen während des Trainings innerhalb der Konsole betrachtet werden, was mit einem auf herkömmliche Art und Weise gebauten Projekt nicht möglich wäre. Um das Projekt im Headless-Modus auszuspielen, muss lediglich eine entsprechende Flag in den Build-Einstellungen des Editors gesetzt werden. Die Verwendung eines exportierten Builds im Headless-Modus führt zu einer Beschleunigung des Trainingsvorgangs um ca. den Faktor 1,9. Wenn man die Anzahl der Dörfer von 1 auf 6 erhöht, wird eine zusätzliche Beschleunigung um den Faktor 1,5 erreicht. So konnte in einem Testtrainingslauf mit 100.000 ausgeführten Schritten und 50 aktiven Agenten pro Dorf die benötigte Zeit für den gesamten Trainingsvorgang von ca. 209 Minuten ohne jegliche Optimierungen auf ca. 134 Minuten durch die Verwendung eines Headless-Builds reduziert werden. Die zusätzliche Nutzung eines Multi-Village-Ansatzes führte zu einer Verringerung der benötigten Zeit auf ca. 71 Minuten. Insgesamt wurde das Verfahren so fast um den Faktor 3 beschleunigt.

Inference-Modus Auch für den Inference-Modus wurden Optimierungen vorgenommen, um ein flüssigeres Spielerlebnis zu gewährleisten. Ein Verfahren namens *Occlusion Culling* wird verwendet, sodass das Rendering der Simulation auf genau die Bereiche limitiert wird, die von der Kamera aus

sichtbar sind. Objekte können die Sicht der Kamera einschränken und dahinter liegende Objekte verdecken, sodass diese für den Rendervorgang ignoriert werden können. Effektiv muss die GPU dadurch weniger Berechnungen durchführen. Occlusion Culling wird von Unity über das Menü *Occlusion* angeboten. Dafür müssen alle gewünschten Objekte in diesem Menü für das Occlusion Culling gekennzeichnet werden. In der Dorfsimulation wurden dafür alle statischen Objekte mit einer Mesh-Komponente in der Szene genutzt.

7.2 Ablauf des Trainings

Im Folgenden wird der generelle Ablauf des Trainings erläutert. Zunächst wird dafür beschrieben, welche Änderungen sich für die Simulation im Trainingsmodus ergeben.

Das Training verändert die Ausführung einiger Vorgänge innerhalb der Simulation. In der **GameAcademy** wird die statische Variable **IS_TRAINING** verwendet. Alle Komponenten der Simulation können auf diese Variable zugreifen.

Wie bereits im Abschnitt [6.4.10](#) angedeutet, haben hohe TimeScale-Faktoren die Eigenschaft, alle physikbasierten Vorgänge innerhalb der Engine zum Oszillieren zu bringen. Da für das Training eine möglichst hohe TimeScale verwendet werden sollte, um die Dauer des Vorgangs maßgeblich zu beschleunigen, ist dieses Problem hier von hoher Relevanz. Der beschriebene *Fake Inference*-Modus wird daher auch während des Trainings für alle Agenten verwendet. Jede **MoveAction** der NPCs wird in eine **WaitAction** transformiert, die in einer Teleportation des Agenten zu seinem Zielort endet.

Die bereits im vorherigen Abschnitt diskutierten Optimierungen verändern ebenfalls den Ausführungsablauf. Der **LightHandler** wird deaktiviert, da er während des Trainings nicht benötigt wird. Weiterhin wird der **GameMonitor** deaktiviert. UI-Funktionen sind während des Trainings nicht nutzbar. Logischerweise werden außerdem Vorgänge innerhalb des ML-Agents Frameworks verändert. Die Entscheidungsfindung des Agenten beruht nicht mehr auf einem trainierten Modell, sondern wird von dem verwendeten Tensorflow-Backend gesteuert.

Die nötige Vorgehensweise für den Start des Trainings wurde bereits in Abschnitt [5](#) erläutert. Das Training läuft so lange, bis ein maximaler Wert für die von der **Academy** ausgeführten Zeitschritte erreicht wurde. Da ausschließlich On-Decmand-Decisions verwendet werden, resultiert nicht jeder Schritt der **GameAcademy** zwangsläufig in einer Aktion der Agenten. Es ist daher nicht bekannt, wie viele Aktionen tatsächlich ausgeführt werden. Die Anzahl der maximalen Zeitschritte wird über den Parameter **max_steps** in der verwendeten YAML-Konfigurationsdatei festgelegt. Typischerweise liegt dieser im Bereich von mehreren Zehntausend bis wenigen Millionen Schritten.

7. Training und Ergebnisse

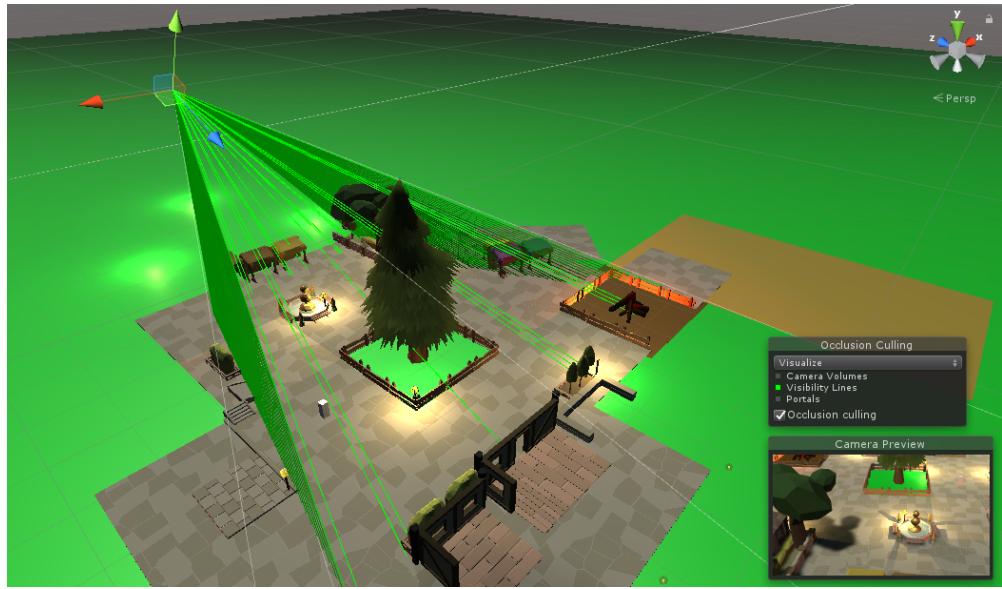


Abbildung 7.1.: Occlusion Culling, Ignorieren von Objekten außerhalb des Sichtfeldes

Unity stellt weiterhin eine Visualisierung für das Verfahren zur Verfügung. In der Abbildung 7.1 ist ein Beispiel für Occlusion Culling zu sehen. Das ausgewählte Objekt, das an den drei Achsen erkannt werden kann, ist die aktive Kamera der Szene. Ihre Position, Ausrichtung und die Breite ihres Sichtfelds bestimmen, welche Objekte berechnet werden müssen. In der rechten unteren Ecke der Abbildung ist der Ausschnitt der Szene zu erkennen, der von der Kamera wahrgenommen wird. In der Szene kann einfach erkannt werden, dass alle weiteren Objekte des Dorfes nicht angezeigt werden, weil sie vom Blickwinkel der Kamera nicht erfasst werden. Die folgende Grafik 7.2 zeigt weiterhin, wie Objekte durch andere Objekte verdeckt werden. Hinter dem großen Gebäude im Vordergrund werden keine Objekte berechnet, das sie durch das Haus verdeckt werden.



Abbildung 7.2.: Occlusion Culling, Ignorieren von Objekten durch Verdeckung

7.3 Verschiedene Ansätze für Ressourcensystem und VillageAgent

Wie bereits in den Abschnitten 6.4.7 und 6.4.2 beschrieben, sind durch die Parameter der GameAcademy verschiedene Ansätze für die Behandlung von Ressourcen und Berufen in der Simulation auswählbar. Insgesamt ergeben sich drei Möglichkeiten für die Verwaltung von Ressourcen:

- **Ansatz 1:** Keine Zuweisung von Berufen für Agenten. Es werden keine Berufsaktionen maskiert, stattdessen können Agenten die Aktionen frei wählen. Die Ressourcen werden durch die Auswahl der Aktionen der Agenten indirekt in der Balance gehalten.
- **Ansatz 2:** Nutzung fest zugewiesener Berufe für Agenten sowie Maskierung aller Aktionen, die nicht zum Beruf gehören. Kontinuierliche Anpassung der Berufe durch den VillageAgent zur indirekten Verwaltung der Ressourcen.
- **Ansatz 3:** Nutzung fest zugewiesener Berufe für Agenten ohne eine Anpassung durch den VillageAgent. Statt eine Lösung für das Ressourcenproblem zu finden, werden Ressourcen vollständig ignoriert und sind für die Entscheidungsfindung nicht mehr relevant.

Die Verwendung eines Ressourcensystems führt dazu, dass die Agenten lernen müssen, mit diesen Ressourcen umzugehen. Es besteht das Risiko, dass die Ressourcen des Dorfsystems ab einem gewissen Punkt nicht mehr in der Balance gehalten werden. Vor dem Training ist nicht absehbar, wie viele Ressourcen von den Agenten verbraucht werden, da nicht klar ist, welche Entscheidungen diese treffen. Es ist daher schwierig, für die Aktionen der Agenten sinnvolle Parameter bezüglich der Kosten und Erlöse von Ressourcen festzulegen. Daher werden Tests durchgeführt, die bestimmen sollen, mit welcher der drei oben genannten Verfahren das Problem am besten gelöst werden kann. Dazu gehört auch unter Umständen die Maßnahme, auf die Nutzung von Ressourcen gänzlich zu verzichten (Ansatz 3), falls kein zufriedenstellendes Ergebnis erzielt werden kann.

Bevor weitere Aspekte des Trainings betrachtet werden, wird zunächst ein Vergleich zwischen drei Testtrainingsvorgängen der drei vorgeschlagenen Ansätze aufgestellt. In der Abbildung 7.3 werden die Trainingsverläufe dargestellt.



Abbildung 7.3.: Trainingsverläufe für NpcAgents, verschiedene Ansätze für das Ressourcensystem

7. Training und Ergebnisse

Grundsätzlich haben sich für alle Ansätze ähnliche Verläufe ergeben. Anhand der eingezeichneten Trendlinien kann man erkennen, dass sich für den ersten Ansatz insgesamt die niedrigsten akkumulierten Belohnungen ergeben haben. Das kann dadurch erklärt werden, dass ein insgesamt größerer Aktionsraum verwendet wurde. Dadurch wird das Training erschwert, weil dieser größere Aktionsraum erforscht werden muss, um Assoziationen zwischen Aktionen und Beobachtungen herzustellen. Für den zweiten Ansatz ergibt sich eine nur geringfügig niedrigere Belohnung im Vergleich zum ersten Ansatz. Aktionen werden im zweiten Ansatz öfter maskiert, weil Ressourcen nicht zur Verfügung stehen. Das kann dazu führen, dass Agenten gewünschte Aktionen nicht ausführen können, weil zu wenig Ressourcen vorhanden sind. Da diese Limitierungen für den dritten Ansatz nicht gelten und der Aktionsraum kleiner ist, wurden dort die höchsten Belohnungen erreicht. Nur im zweiten Ansatz wurde der *VillageAgent* verwendet. Neben den *NpcAgents* mussten im entsprechenden Trainingsvorgang also auch *VillageAgents* trainiert werden. Insgesamt konvergieren alle drei Ansätze bereits nach etwa 400.000 Trainingsschritten. Die Graphen beschreiben anschließend einen nahezu flachen Verlauf. Ein signifikanter Lernvorgang ist nach dem Erreichen dieses Punktes nicht mehr zu erwarten.

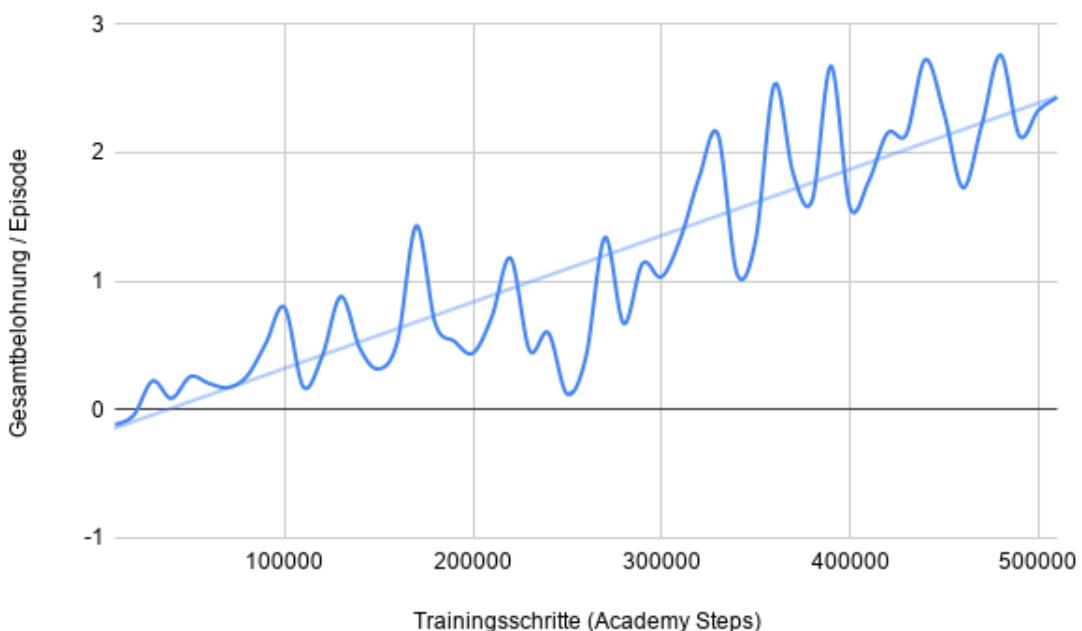


Abbildung 7.4.: Trainingsverlauf des *VillageAgent*

In Abbildung 7.4 ist der Trainingsverlauf des *VillageAgent* zu erkennen. Der Verlauf der akkumulierten Belohnung steigt nicht monoton an. Es bildet sich ein eher unstabiles Training mit vielen lokalen Maxima und Minima ab. An der Trendlinie ist jedoch klar erkennbar, dass insgesamt ein Lernvorgang durch ein Wachstum der Belohnungen stattfindet. Weiterhin wird durch den Graphen bis zum Ende des Trainingsvorgangs kein Plateau abgebildet. Die akkumulierten Belohnungen verändern sich kontinuierlich. Aus diesem Grund wird ein erweiterter Trainingsprozess für den *VillageAgent* mit einer höheren Anzahl von Trainingsschritten durchgeführt, wie in Abbildung 7.5 zu sehen.

7. Training und Ergebnisse

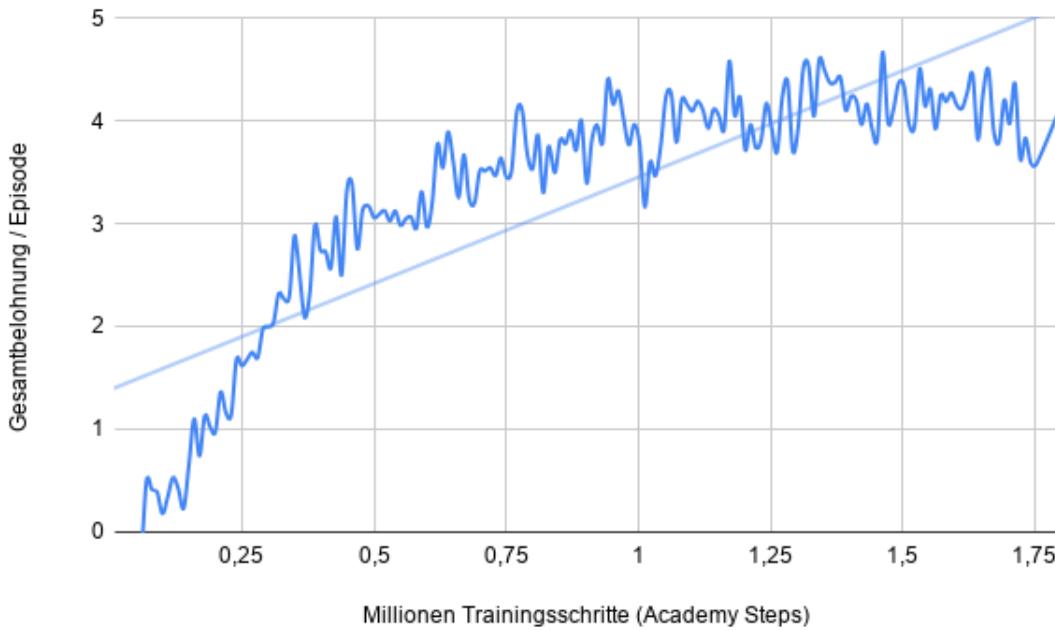


Abbildung 7.5.: Erweiterter Trainingsverlauf des VillageAgent

Auch im erweiterten Trainingsprozess ist zu erkennen, dass der entsprechende Graph nicht monoton steigt, sondern starke Schwankungen aufweist. Trotzdem ist wie bereits oben beschrieben ein konstantes Wachstum zu verzeichnen. Zum Ende des Verlaufs ist zu erkennen, dass der Graph einen flachen Verlauf beschreibt und das Wachstum der Belohnungen zum Stillstand kommt. Nach knapp mehr als einer Million Schritten wird annähernd ein Plateau erreicht.

Die Betrachtung der akkumulierten Belohnungen alleine reicht allerdings noch nicht aus, um das Verhalten der Agenten zu beurteilen. Die aus den Trainingsvorgängen entstandenen Modelle bzw. Policies wurden daher im Inference-Modus der Simulation verwendet, um das Verhalten der Agenten zu betrachten und dadurch die Auswahl der verwendeten Ansätze zu beeinflussen. Es wurden jeweils mehrere Spielzeittage betrachtet, um das resultierende Verhalten der Agenten zu analysieren.

Im ersten Ansatz war auffällig, dass häufig viele Agenten den selben Beruf gewählt haben. Zeitweise haben etwa die Hälfte der 50 pro Dorf verwendeten Agenten dieselbe berufliche Tätigkeit ausgeführt. Die Agenten haben also nicht gelernt, Berufsaktionen aufgrund der aktuell vorhandenen Ressourcen zu wählen. Das führte dazu, dass das Ressourcensystem nicht in der Balance gehalten wurde und Defizite für bestimmte Ressourcen entstanden. Aus diesem Grund konnten nach einiger Zeit bestimmte Aktionen nicht mehr ausgeführt werden, weil ihre ressourcenbasierten Bedingungen nicht erfüllt wurden. Die trotzdem hohe Belohnung der Agenten ist darauf zurückzuführen, dass für maskierte Aktionen aufgrund von fehlenden Ressourcen in der Regel Alternativen verfügbar sind. Wenn zum Beispiel kein Bier mehr vorhanden ist, kann zwar die Freizeitaktion *Taverne* nicht mehr ausgeführt werden, wohl aber die Freizeitaktion *Visit*. Die Vielfältigkeit der Aktionen der Agenten wird dadurch jedoch eingeschränkt. Weiterhin bringt etwa das Ausführen der Berufsaktion *Logging* bei einem gegebenen Arbeitsbedürfnis die gleiche Belohnung wie die Aktion *Hunting*, sodass kein direkter Anreiz in Form von Belohnungen geboten wird, um unterschiedliche Berufsaktionen auszuführen. Die Verwendung der Ressourcen als Observations reicht daher nicht aus, um die Wahl der Berufsaktionen maßgeblich zu beeinflussen. Verschiedene Ideen wurden verfolgt, um dieses Problem einzudämmen. Beispielsweise wurde für jede ausgeführte Aktion des Agenten eine zusätzliche Belohnung bzw. Bestrafung verteilt, die auf die Menge der Ressourcen zurückgeführt wurde -

7. Training und Ergebnisse

unabhängig davon, welche Aktion ausgeführt wurde. Außerdem wurde der Versuch unternommen, die Belohnungen beruflicher Aktionen an den produzierten Ressourcen im Vergleich zur aktuellen Menge der entsprechenden Ressource zu messen. Eine hohe Belohnung wurde dann erzielt, wenn ein Job ausgeführt wurde, dessen mit ihm verbundene Ressource nicht ausreichend vorhanden war. Diese unternommenen Versuche konnten das Problem jedoch nicht beheben bzw. verschlechterten dieses teilweise.

Daher wurde der im zweiten Ansatz verwendete *VillageAgent* entwickelt, der die Menge Ressourcen auf einem stabilen Level halten soll. Weiterhin wurden dafür die festen Berufe bzw. *Jobs* für Agenten eingeführt, die dazu führen, dass alle Aktionen eines Agenten, die nicht mit seinem Job assoziiert sind, maskiert werden. Für den Inference-Modus wurde der zweite, erweiterte Trainingsverlauf des *VillageAgent* verwendet (siehe Abbildung 7.5), um die bestmögliche Policy zu nutzen. Wie bereits in Abschnitt 6.4.7 angedeutet, fällen alle verwendeten Agenten vom Typ *VillageAgent* immer dann eine Entscheidung, wenn ein Zähler in der *GameAcademy* einen bestimmten Wert erreicht. Die laufende Episode wird abgebrochen, sobald eine bestimmte Anzahl von Entscheidungen getroffen wurde. Der Zähler wird jedes mal inkrementiert, wenn ein *NpcAgent* eine Aktion ausführt. Die entsprechenden Parameter in der *GameAcademy* wurden so gewählt, dass alle *VillageAgents* Entscheidungen ausführen, sobald jeder *NpcAgent* fünf Aktionen ausgeführt hat. Insgesamt sind zehn dieser Entscheidungen notwendig, um die Episode abzuschließen. Mit sechs verwendeten Dörfer und 50 pro Dorf verwendeten Agenten werden daher 1500 NPC-Aktionen gebraucht, um eine Entscheidung auszulösen und 15.000, um die Episode zu beenden. Der *VillageAgent* erhält Belohnungen, wenn richtige Entscheidungen getroffen werden und Bestrafungen, wenn kontraproduktive Entscheidungen getroffen werden. Trotz eines kontinuierlichen Lernvorgangs und einer deutlich positiven akkumulierten Gesamtbewertung kommt es nicht dazu, dass das Ressourcensystem durch eine Anpassung der Berufe in Waage gehalten wird. Das System oszilliert stark und lernt nicht ausreichend, Entscheidungen zu treffen, die auf der aktuellen Menge der Ressourcen beruhen. Ein grundlegendes Problem bei diesem Vorgang ist die Häufigkeit der Entscheidungen, die durch den Agenten gefällt werden - auf 1000 gesammelte Erfahrungen durch *NpcAgents* kommen lediglich ca. 4 Erfahrungen des *VillageAgents*. Der *VillageAgent* muss also aus einer Menge von Entscheidungen eine Policy erlernen, die nur einem Anteil von 0.4% der entsprechenden Menge des *NpcAgent* entspricht. Um den Zustandsraum des Agenten hinreichend zu erkunden, muss vermutlich eine deutlich höhere Anzahl von Erfahrungen gesammelt werden. Ein zusätzliches potentielles Problem besteht darin, dass die Auswirkungen der Entscheidungen auf die tatsächlichen Ressourcenmengen eine starke zeitliche Verzögerung haben, da Agenten nur etwa ein mal am Tag arbeiten. Das kann theoretisch dazu führen, dass die gleichen Aktionen mehrfach ausgeführt werden, weil die Menge der Ressourcen sich seit der letzten Entscheidung nicht signifikant verändert hat. Es wurden Versuche unternommen, diese Probleme zu lösen, indem die Belohnungen und die Intervalle der *VillageAgent*-Entscheidungen angepasst wurden. Diese Versuche führten ebenfalls dazu, dass den NPCs falsche Berufe zugewiesen wurden.

Aus diesem Grund wird der dritte Ansatz verwendet, in dem das Ressourcensystem ignoriert wird. Aktionen produzieren und verbrauchen dann keinerlei Ressourcen mehr. Weiterhin werden keine ressourcenbasierte Observations getätigt und keine Belohnungen an Ressourcen gebunden.

Dadurch wird der Simulation ein kleiner Teil ihrer Komplexität genommen. Die Glaubwürdigkeit der Entscheidungsfindung der NPCs wird dadurch allerdings nicht beeinflusst, da diese maßgeblich von den Bedürfnissen des Agenten abhängt. Die Nutzung von Ressourcen fungiert viel mehr als limitierender Faktor. Um die Agenten tatsächlich authentischer wirken zu lassen, müsste das Ressourcensystem um Faktoren wie Handel zwischen verschiedenen Dörfern erweitert werden. Da dies nicht gegeben ist, stellt das Ignorieren des Ressourcensystems keine Verschlechterung dar.

7.4 Wahl der initialen Hyperparameter

Die Tabelle 7.1 zeigt eine Übersicht über die wichtigsten Hyperparameter für das Training der Agenten. Diese werden in einer Konfigurationsdatei definiert, wie in Abschnitt 5 beschrieben. Für die Erstellung der Übersicht dient eine Tabelle aus der ML-Agents Dokumentation als Grundlage [Tec17e], die Empfehlungen für die Parameter ausspricht. Alle nicht PPO-basierten Parameter wurden aus der Tabelle entfernt, da sie für diese Arbeit nicht weiter relevant sind.

Setting	Description
batch_size	The number of experiences in each iteration of gradient descent.
beta	The strength of entropy regularization.
buffer_size	The number of experiences to collect before updating the policy model.
epsilon	Influences how rapidly the policy can evolve during training.
hidden_units	The number of units in the hidden layers of the neural network.
lambd	The regularization parameter.
learning_rate	The initial learning rate for gradient descent.
max_steps	The maximum number of simulation steps to run during a training session.
memory_size	The size of the memory an agent while training with a recurrent network.
normalize	Whether to automatically normalize observations.
num_epoch	The number of passes to make through the experience buffer when performing gradient descent optimization.
num_layers	The number of hidden layers in the neural network.
behavioral_cloning	Use demonstrations to bootstrap the policy neural network.
reward_signals	The reward signals used to train the policy. Curiosity and GAIL can be enabled here. Gamma can also be set here.
sequence_length	Defines how long the sequences of experiences must be while training. Only used for training with a recurrent neural network.
summary_freq	How often, in steps, to save training statistics. This determines the number of data points shown by TensorBoard.
time_horizon	The amount of experiences collected per-agent before adding it to the experience buffer.
use_recurrent	Train using a recurrent neural network.

Tabelle 7.1.: ML-Agents-Hyperparameter für das Training [Tec17d]

Ein Großteil dieser Parameter wurde bereits durch mathematische Grundlagen im Kapitel 3 definiert. Die Funktion weiterer Parameter geht entweder aus der Beschreibung innerhalb der Tabelle hervor oder wird in der folgenden Tabelle 7.4 erläutert. Einige der Parameter sind optional.

Für das Training der Agenten wurden mehrere Trainingsvorgänge mit unterschiedlichen Hyperparametern durchgeführt, die später miteinander verglichen werden. Als Grundlage für das Training wird der im vorherigen Abschnitt beschriebene Trainingsvorgang für den dritten Ansatz der Lösung für den Umgang mit Ressourcen verwendet. Die in diesem Vorgang verwendete Trainingskonfiguration, die im Folgenden vorgestellt wird, stellt eine Basiskonfiguration dar, die das Fundament für weitere Trainingsvorgänge bildet. In diesen Vorgängen werden jeweils einzelne Parameter aus der Konfiguration angepasst. Dadurch sollen möglichst ideale Hyperparameter identifiziert werden. Bereits die in der Basiskonfiguration festgelegten Parameter wurden empirisch ermittelt, indem während der Entwicklung des Dorfes zahlreiche Trainingsvorgänge durchgeführt wurden. Schließlich wird ein finales Training durchgeführt, das die besten verwendeten Hyperparameter nutzt und so ein optimales Ergebnis erzielen soll. Im Folgenden werden Hyperparameter für den zugrunde

7. Training und Ergebnisse

liegenden Trainingsvorgang beschrieben. Für die initiale Festlegung der Trainingsparameter wurden teilweise Empfehlungen aus der Dokumentation des ML-Agents Frameworks hinzugezogen [Tec17e]. Ein Großteil der Parameter wurde bereits im Kapitel 3 erläutert und ist auf die mathematischen Grundlagen des Verfahrens zurückzuführen. Folgende Parameter wurden für das Basistraining verwendet:

Parameter	Beschreibung
<i>trainer</i>	ppo
<i>epsilon</i>	0.2
<i>lambd</i>	0.925
<i>learning_rate_schedule</i>	linear
<i>learning_rate</i>	4e-4
<i>max_steps</i>	5e5
<i>normalize</i>	false
<i>batch_size</i>	32
<i>buffer_size</i>	8192
<i>num_epoch</i>	4
<i>time_horizon</i>	512
<i>beta</i>	1.2e-2
<i>summary_freq</i>	2000
<i>num_layers</i>	3
<i>hidden_units</i>	256
<i>use_recurrent</i>	false
<i>memory_size</i>	128
<i>sequence_length</i>	128
<i>reward_signals</i>	extrinsic
<i>reward_strength</i>	1.0
<i>gamma</i>	0.93

Tabelle 7.2.: Gewählte Hyperparameter für das Basistraining.

Als zugrunde liegender Algorithmus wird PPO verwendet. *epsilon* wird auf 0.2 festgelegt. Der Wert bestimmt, wie groß Änderungen der Policy minimal sein müssen. Über den Parameter *lambd* kann definiert werden, wie sehr sich der Algorithmus auf die Vorhersage (bzw. *Value Estimate*) der Belohnung im Gegensatz zur eigentlichen erfahrenen Belohnung einer Aktion verlässt. Es wird eine Lernrate mit dem Wert $4 * 10^{-4}$ verwendet, die sich im Verlauf des Trainingsvorgangs linear bis zum Wert 0 beim letzten Trainingsschritt verringert. Insgesamt werden dafür 500.000 Trainingsschritte verwendet. Eine Normalisierung der Observations findet nicht statt, da alle Beobachtungen manuell normalisiert werden. Weiterhin muss ein Wert für die *buffer_size* festgelegt werden. Dadurch wird beschrieben, wie viele Entscheidungen gesammelt werden sollen, bevor ein Update der Policy stattfindet. Dieser Buffer wird in Batches von der Größe *batch_size* aufgeteilt. Für jeden dieser Batches wird ein Gradientenverfahren durchgeführt, um das neue, angepasste Modell zu berechnen. Der Parameter *num_epoch* legt fest, wie oft dieser Prozess wiederholt werden soll. Der Parameter *time_horizon* legt fest, wie viele Entscheidungen auf einmal in den Buffer geschrieben werden sollen. Wenn die Kapazität des Buffers vor dem Beenden einer Episode erreicht wurde, wird ein *Value Estimate* gebildet, um die zukünftigen Belohnungen des Agenten der Episode zu schätzen. *buffer_size* und *time_horizon* müssen daher logischerweise ein Vielfaches der *batch_size* sein. Diese Parameter wurden durch Empfehlungen des Frameworks und empirische Beobachtungen ausgewählt. Konkret wurden noch während der Entwicklung der Dorfsimulation verschiedene Trainingsvorgänge ausgeführt, wobei sich bestimmte Parameter bewährt haben.

Über den Wert *beta* wird definiert, wie zufällig die Aktionen der Agenten ausgewählt werden, um zu gewährleisten, dass der Aktionsraum des Agenten sinnvoll erkundet wird. Damit wird also direkt die Entropie beeinflusst, die festlegt, wie zufällig die Entscheidungen eines Brains sind. Der Wert sollte so ausgewählt werden, dass die Entropie des Trainingsvorgangs stetig sinkt.

Über die *summary_freq* kann die Schrittgröße für die Ausgabe der durchschnittlichen akkumulierten Belohnung der Agenten in der Konsole bestimmt werden. Größere Werte führen zu einem zu hohen Intervall zwischen Statusberichten. Zu kleine Werte führen zu einer starken Varianz bezüglich der erfahrenen Belohnungswerte. In diesem Fall wird für jeweils 2.000 durchgeführte Trainingsschritte ein Statusbericht angegeben.

Die Parameter *num_layers* und *hidden_units* sind essentiell für den Aufbau des neuronalen Netzes. Insgesamt werden drei Schichten mit je 512 enthaltenen Neuronen verwendet. Es wird kein rekurrentes neuronales Netz verwendet, da der Parameter *use_recurrent* auf *false* gesetzt wird. Die *memory_size* bestimmt die Größe für einzelne „Erinnerungen“ für die Rückkopplungen in dem rekurrenten Netz. Über den Parameter *sequence_length* wird zudem festgelegt, wie viele dieser Erinnerungen verwendet werden sollen. Diese Parameter werden nur für rekurrente neuronale Netze verwendet und sind daher zunächst nicht relevant.

Der Wert des *reward_signals* bestimmt, welche Art von Belohnungen das Verfahren verwendet und wird daher auf *extrinsic* festgelegt. Damit sind die Belohnungen gemeint, die in der Simulation festgelegt wurden. Es können zusätzliche *reward_signals* festgelegt werden, um mehrere Belohnungsarten zu nutzen. Durch *curiosity*-Belohnungen erhält der Agent immer dann Belohnungen, wenn er von seiner Policy abweicht, um den Aktionsraum zu erkunden. Das ist dann sinnvoll, wenn extrinsische Belohnungen sehr selten sind und nur durch „Ausprobieren“ gefunden werden können. Da in diesem Fall jede Aktion eine Belohnung zur Folge hat, werden zunächst keine derartigen Belohnungen verwendet. Zusätzlich können *GAIL*-Belohnungen verwendet werden, die nicht relevant sind, da sie bei Imitation Learning zum Einsatz kommen. Konkret werden extrinsische Belohnungen mit einem Multiplikator *reward_strength* von 1.0 verwendet, da die in der Simulation definierten Belohnungen nicht verändert werden sollen. Weiterhin wird ein relativ niedriger Wert *gamma* festgelegt. Dieser Parameter beschreibt den in Kapitel 3 definierten Wert γ und bestimmt, wie sehr zukünftige Belohnungen Entscheidungen des Agenten beeinflussen. Für γ wird ein niedriger Wert gewählt, sodass zukünftige Zustände des Agenten weniger relevant sind. Es wird davon ausgegangen, dass für die Befriedigung von menschlichen Bedürfnissen hauptsächlich der aktuelle Zustand relevant ist.

7.5 Güteüberprüfung

Beim überwachten Lernen werden häufig direkte Bewertungskriterien genutzt, indem Werte für *Precision*, *Recall* oder *F1* gebildet werden. Da beim RiL keine Klassifizierung stattfindet und keine gelabelten Daten vorhanden sind, ist ein derartiges Vorgehen hier nicht möglich. Ein exakter Wert für die Genauigkeit der Ergebnisse kann daher nicht gebildet werden. Um die Güte der Trainingsergebnisse zu überprüfen, können alternative Metriken verwendet werden. Neben der Höhe der Belohnungen wird die *Value Loss*-Funktion verwendet. Sie beschreibt die Verlustfunktion (siehe Abschnitt 3.3.2.3) bezüglich der Value Function des Agenten und gibt damit an, wie gut das Modell Belohnungen zukünftiger Zustände vorhersagen kann. Weiterhin wird der Entropieverlauf genutzt, der die Zufälligkeit der gewählten Aktionen bestimmt. Die Daten können im Tensorboard betrachtet und von dort exportiert werden. Im Normalfall kann man auch an der Länge der Episoden den Lernvorgang bewerten. Die Länge der Episoden der Agenten ist hier aber nicht relevant, da die Episoden stets nach einer festgelegten Anzahl von Aktionen enden.

Mit den genannten Metriken kann zwar der Erfolg des Trainingsvorgang aus einer mathematischen Sicht evaluiert werden, sie bieten jedoch nur Hinweise darüber, wie authentisch das resultierende Verhalten der Agenten ist. Um das tatsächliche Verhalten zu bewerten, reichen diese Metriken also nicht aus. Hohe Belohnungen könnten zum Beispiel auch entstehen, wenn ein Agent lernt, einen Fehler in der Simulation auszunutzen. Daher müssen die Agenten konkret innerhalb der Simulation im Inference-Modus beobachtet werden. Dafür können die entwickelten UI-Elemente, die Auskunft

7. Training und Ergebnisse

über verschiedene Informationen des Dorfes wie der Anzahl der aktuell ausgeführten Aktionen bieten, genutzt werden. Diese Informationen bieten einen schnellen Überblick über das erlernte Verhalten der Agenten. Im Abschnitt 7.6.4 wird neben Metriken wie der Entropie oder den Belohnungen der Agenten daher zusätzlich das Verhalten der Agenten betrachtet.

7.6 Trainingsergebnisse

Wie im vorherigen Abschnitt erwähnt, wurde eine Basiskonfiguration für Trainingsvorgänge erstellt, auf deren Grundlage Trainingsvorgänge durchgeführt werden. Einzelne Parameter werden für diese Vorgänge angepasst, anschließend wird ein Vergleich der Ergebnisse durchgeführt.

Zunächst wird der Basistrainingsvorgang näher betrachtet.

7.6.1 Basistraining

In den Abbildungen 7.8, 7.7 und 7.6 sind Informationen zum Basistrainingsverlauf zu erkennen. Auf den horizontalen Achsen der Diagramme ist jeweils die Anzahl der Trainingsschritte eingezeichnet.

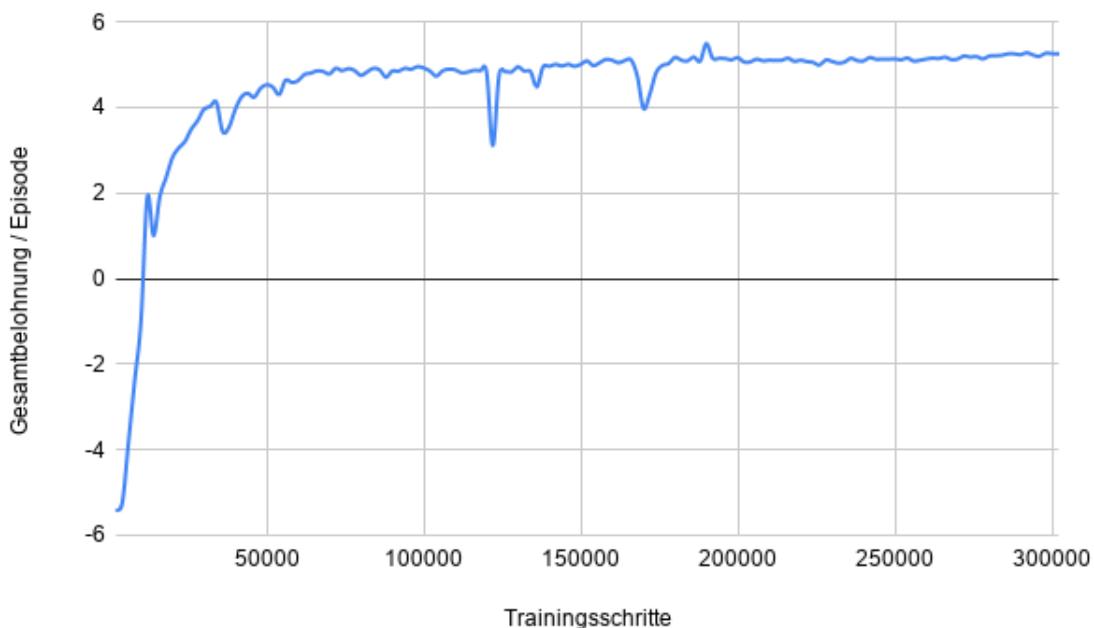


Abbildung 7.6.: Akkumulierte Belohnungen im Basistraining des NpcAgent

7. Training und Ergebnisse



Abbildung 7.7.: Value Loss im Basistraining des NpcAgent

Die Abbildung 7.7 zeigt den Verlauf des *Value Loss*, also die Verlustfunktion bezüglich der Value Function. Solange der Graph einen absteigenden Verlauf beschreibt, lernt der Agent, die Value Function zu optimieren.

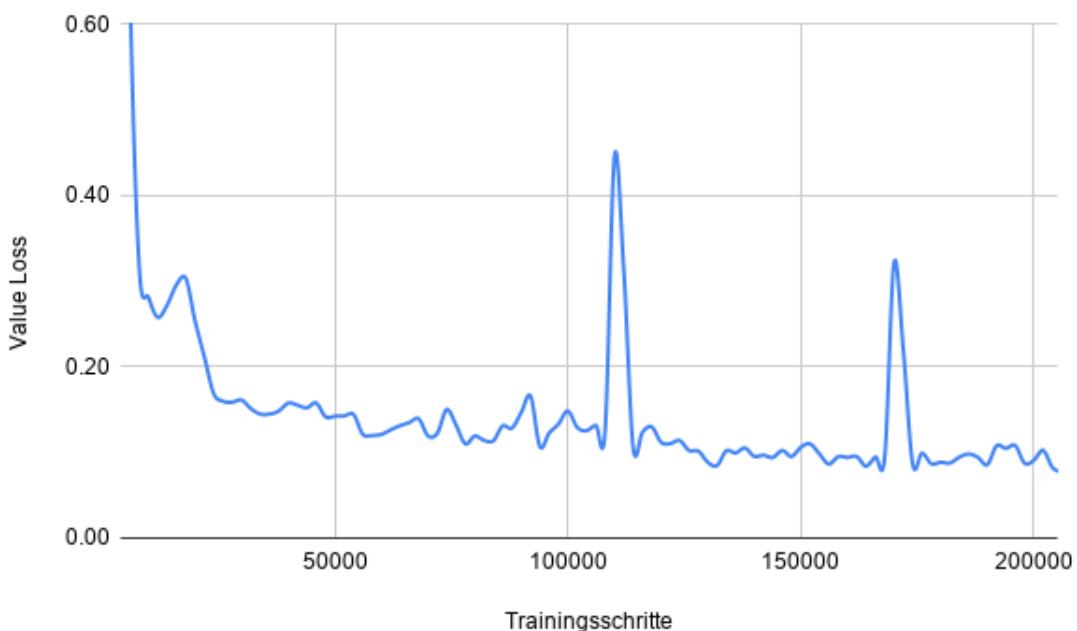


Abbildung 7.8.: Entropieverlauf im Basistraining des NpcAgent

7. Training und Ergebnisse

Die Entropie wurde auf den Wert $1.2 * 10^{-2}$ festgelegt. Daraus ergibt sich der in Abbildung 7.8 sichtbare Entropieverlauf. Die stetige Verringerung der Entropie mit einer steigenden Anzahl der Trainingsschritte belegt, dass ein guter Wert gewählt wurde.

Weiterhin beschreibt die Abbildung 7.6 die akkumulierten Belohnungen für das Basistraining, wie bereits in Teilen in Abbildung 7.3 zu sehen. Man kann erkennen, dass sich die Spitzen der Belohnungsfunktion bei etwa 125.000 und 170.000 Zeitschritten im Value Loss-Diagramm widerspiegeln. Typischerweise sinkt die Verlustfunktion so lange, wie die Belohnung des Agenten zunimmt. Weiterhin sind keine Auffälligkeiten zu erkennen.

7.6.2 Optimierung der Ergebnisse durch Anpassung der Hyperparameter

Um optimale Parameter zu definieren, wurde eine Reihe von Trainingsvorgängen durchgeführt. In jedem dieser Durchläufe wurde ein einzelner Parameter aus der Basiskonfiguration angepasst, um die Auswirkungen der Änderung isoliert zu betrachten. Das Ergebnis wird mit anderen Trainingsvorgängen verglichen, in dem der selbe Parameter verändert wurde. Es wird geprüft, ob die Veränderung eine Verbesserung erzielt. Der Parameter mit dem besten Trainingsresultat wird anschließend für ein finales Training gewählt und in die finale Konfigurationsdatei eingetragen. Über mehrere Tage wurden dafür Trainingsdurchläufe durchgeführt. Die jeweilige Dauer der durchgeföhrten Trainingsvorgänge kann in der Abbildung 7.9 betrachtet werden. Auffällig ist, dass das Training mit einem rekurrenten Netzwerk deutlich mehr Zeit beansprucht, da durch die zusätzlichen Verbindungen der Neuronen im Netzwerk ein höherer Rechenaufwand entsteht. Darüber hinaus sind keine Besonderheiten zu verzeichnen. Alle Trainingsvorgänge haben in etwa die selbe Zeit in Anspruch genommen. Für weitere leichte Schwankungen wird eine variable Auslastung der CPU verantwortlich gemacht, da die verwendete Hardware zur Trainingszeit teilweise mit weiteren Aufgaben gefordert wurde.

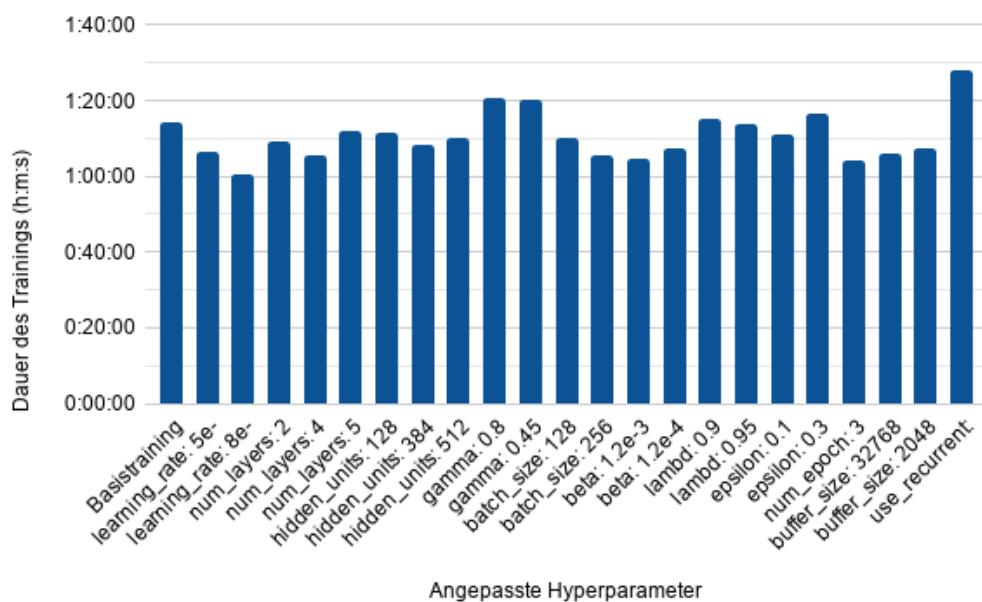


Abbildung 7.9.: Dauern der unterschiedlichen Trainingsvorgänge

Nach dem Vergleich der Ergebnisse sollte ein verlängertes, finales Training durchgeführt werden. Es wurde davon ausgegangen, dass die Parameter unabhängig voneinander verändert und schließlich kombiniert werden können, um ein optimales Ergebnis zu erzielen. Es hat sich allerdings gezeigt,

7. Training und Ergebnisse

dass die Nutzung mehrerer Parameteränderungen, die jeweils ein besseres Ergebnis im Vergleich zum Basistraining erzielt haben, in Kombination nicht zwangsläufig ein besseres Ergebnis zur Folge haben. Damit wurde gezeigt, dass es starke Abhängigkeiten zwischen den einzelnen Parametern gibt. In der Abbildung 7.10 ist die entstandene Belohnungsfunktion aus diesem Training zu sehen. Es ist leicht zu erkennen, dass die Gesamtbelohnungen mit einem Maximalwert von ca. 2.6 deutlich unter dem Niveau des Basistrainings liegen.

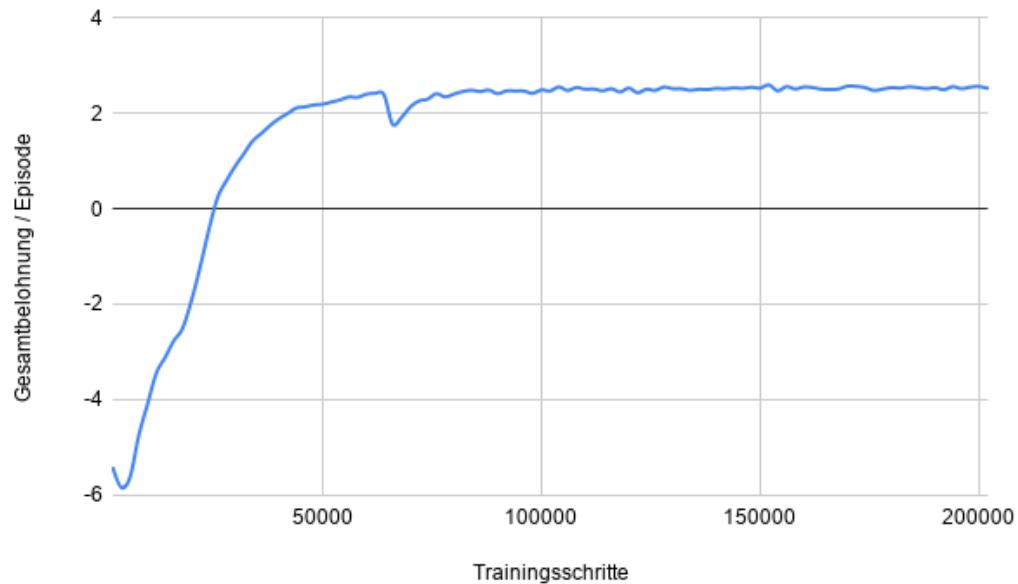


Abbildung 7.10.: Belohnungsfunktion durch Kombination der optimalen Einzelparameter

Die Ergebnisse der ausgeführten Einzeltrainings geben trotzdem eine Auskunft über die Anpassung der Parameter im Einzelnen. Sie sind im Anhang A zu finden und werden durch Diagramme dokumentiert.

Um bei vorhandenen Abhängigkeiten zwischen den Parametern die optimalen Werte zu finden, hätte nach der Ermittlung eines neuen, besseren Parameters bereits die Basiskonfiguration mit diesem Parameter angepasst werden müssen, sodass die Anpassung bei weiteren Trainingsvorgängen bereits miteinbezogen wurde.

Insgesamt stellt die Wahl der Hyperparameter ein schwieriges Problem dar, da extrem viele Kombinationen verschiedener Werte für Hyperparameter durchgeführt werden können. Einige Werkzeuge und Frameworks für Machine Learning bieten eine automatisierte Auswahl für Hyperparameter an, indem eine Menge von Kombinationen verschiedener Kandidaten für Parameter geprüft werden, um das beste Ergebnis zu erlangen. Ein Beispiel für dieses Vorgehen ist die sogenannte *Exhaustive Grid Search* [PVG+19] des Frameworks *scikit-learn* [PVG+11]. Im Falle von ML-Agents gibt es so eine automatisierte Suche der Parameter nicht. Die Dauer der nötigen Trainingsvorgänge würde weiterhin deutlich zu hohe Maßstäbe annehmen.

7.6.3 Finaler Trainingsvorgang

Im vorherigen Abschnitt wurde der Versuch unternommen, optimale Trainingsparameter zu ermittelt, in dem eine Reihe von Testtrainingsdurchläufen ausgeführt wurde. Anschließend wurde ein verlängertes Trainingsvorgang mit diesen Parametern durchgeführt, wodurch schlechte Werte für die Belohnungen der Agenten erreicht wurden.

Da nicht das gewünschte Ergebnis erzielt wurde, wurden weitere Testtrainingsvorgänge durchgeführt. Es wurde festgestellt, dass bei einer Verringerung der Lernrate auf $1e - 4$ und einer Erhöhung der Batchgröße auf 64 bessere Ergebnisse erzielt werden können. Zusätzlich wird die Anzahl der Trainingsschritte auf $1e6$ erhöht. Mit diesen Parametern wird nun ein finales, verlängertes Training durchgeführt, das zu einem verbesserten Lernvorgang führen soll. In den folgenden Diagrammen sind die Resultate dieses Trainingsvorgangs abgebildet.

Im Vergleich zum Basistraining wurden unmittelbar leicht verbesserte Ergebnisse für alle drei verwendeten Metriken erzeugt. Durch die Verlängerung des Trainings konnte weiterhin ein leichter, stetiger Anstieg bei der Belohnungsfunktion und ein leichter, stetiger Abstieg für die Entropie und den Value Loss ausgenutzt werden, um insgesamt bessere Ergebnisse zu erlangen. Die maximale Belohnung beträgt im Basistraining 5,241 und im finalen Training 5,429. Die Gesamtbelohnung pro Episode wurde damit um ca. 3,6% gesteigert. Der minimale Value Loss wurde von 0,073 auf 0,037 verbessert. Weiterhin konnte die Entropie von 1,127 auf 0,642 verringert werden.

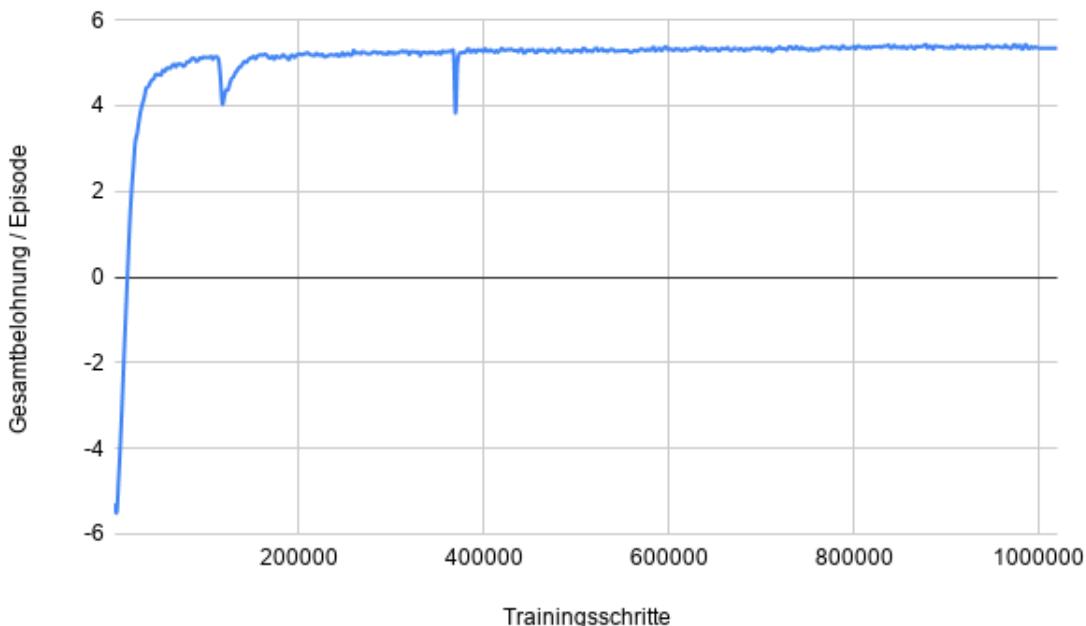


Abbildung 7.11.: Akkumulierte Belohnungen im finalen Training des NpcAgent

7. Training und Ergebnisse

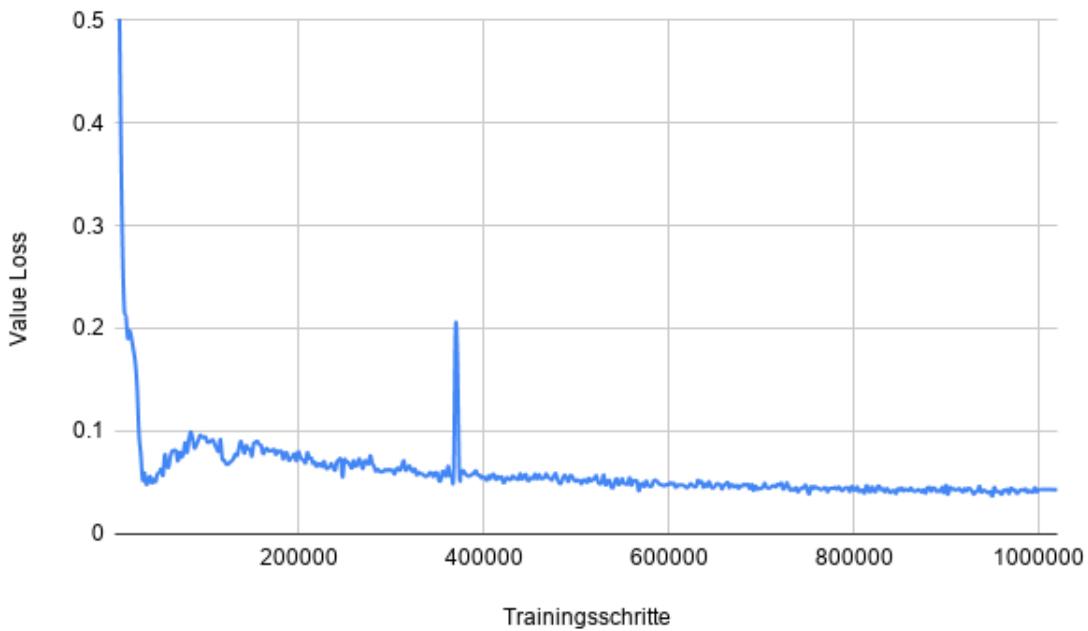


Abbildung 7.12.: Value Loss im finalen Training des NpcAgent

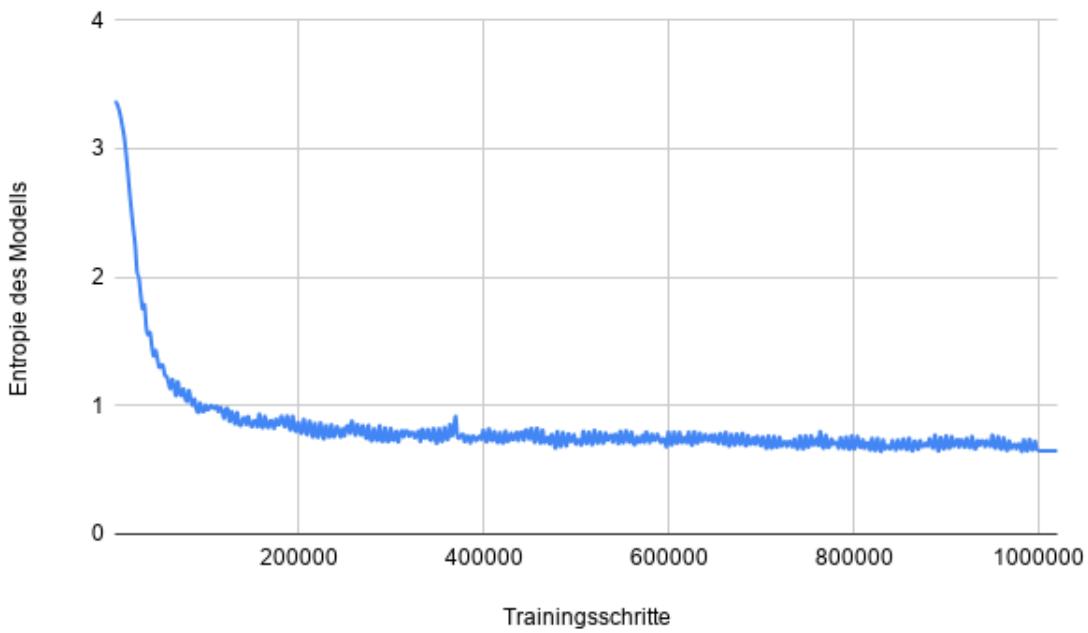


Abbildung 7.13.: Entropieverlauf im finalen Training des NpcAgent

7.6.4 Verhalten der Agenten

Durch den finalen Trainingsvorgang wurde ein Modell für die Entscheidungsfindung der NPCs erzeugt. Das Modell bestimmt das Verhalten der Agenten, das im Folgenden untersucht wird.

7.6.4.1 Eindruck

Der erste Eindruck erweckt den Anschein einer belebten Stadt, in der viele NPCs unterschiedliche Tätigkeiten ausführen. Es wirkt lebendig, da in allen Bereichen des Dorfes NPCs umherlaufen. Damit wird zunächst genau der Eindruck erzielt, der vermittelt werden soll. Dem Anschein nach befindet sich der Spieler in einer belebten Welt, in der eine Vielzahl von Agenten ihrem persönlichen Leben nachgehen. Durch die Betrachtung der Informationen im UI ist weiterhin auffällig, dass unterschiedliche Aktionen ausgeführt werden.

7.6.4.2 Auswahl von Aktionen aufgrund von Bedürfnissen

Über einen längeren Zeitraum wurden mehrere Agenten näher betrachtet. Dabei wurde für jede Aktion, die sie ausgeführt haben, auf ihre Bedürfnisse geachtet, um zu bewerten, ob die Entscheidungen der Agenten sinnvoll sind. Dabei wurde festgestellt, dass die Agenten in fast allen Fällen eine Entscheidung treffen, die für einen Beobachter unter Berücksichtigung der Bedürfnisse des Agenten sinnvoll erscheinen. Aus diesen Beobachtungen kann gefolgert werden, dass die Agenten erfolgreich gelernt, haben, aufgrund ihrer Bedürfnisse Entscheidungen zu treffen, die diese befriedigen. In dem überwiegenden Großteil der Beobachtungen wählten die Agenten Aktionen für das Bedürfnis, das den niedrigsten Bedürfniswert hatte. Im `NeedCycleHandler` wurden für die Bedürfnisse *Hunger*, *Durst*, *Bildung*, *Kommunikation* und *Glauben* Konstanten festgelegt. Diese bestimmen den Zyklus der Bedürfnisse und damit die Steigung der Bedürfnisfunktion und die Dauer bis zum Erreichen eines kritischen Bedürfnislevels. In den Beobachtungen aus der Dorfsimulation spiegelt sich die unterschiedliche Dauer der Zyklen wider. Die verschiedenen Bedürfnisse werden in unterschiedlichen zeitlichen Abständen befriedigt. Die definierten Konstanten decken sich dabei mit dem Ausführungsrythmus der jeweiligen Aktionen. Das Glaubensbedürfnis erreicht zum Beispiel nach drei Tagen den Wert 0. Etwa alle drei Tage wird das Bedürfnis gestillt.

Im Gegensatz zu den genannten Basisbedürfnissen kann weiterhin festgestellt werden, dass die NPCs in regelmäßigen Abständen arbeiten und schlafen. Das ist sinnvoll, weil das Schlaf- und das Arbeitsbedürfnis direkt von der simulierten Zeit des Dorfes abhängt. Dadurch entsteht für die Agenten ein grober Tagesablauf. Zusätzlich wurde die Beobachtung aufgestellt, dass die Agenten relativ genau ein mal pro Tag schlafen und arbeiten.

7.6.4.3 Beitrag der Attribute zur Entscheidungsfindung

Während der Entwicklung der Simulation kam es häufig dazu, dass Agenten bestimmte Aktionen bevorzugt haben, da für diese die höchste Belohnung erwartet wurde. Das ist besonders für *Recreation*-Aktionen relevant, da diese Aktionen alle dasselbe Bedürfnis befriedigen. Um das Problem zu umgehen, wurden die Belohnungen der Aktionen von den Attributen der Agenten abhängig gemacht. Weiterhin ist der Beruf des Agenten ausschlaggebend für die Attribute der Agenten. In der folgenden Abbildung ist die resultierende Verteilung der Attribute in Abhängigkeit von den Berufen zu sehen:

7. Training und Ergebnisse

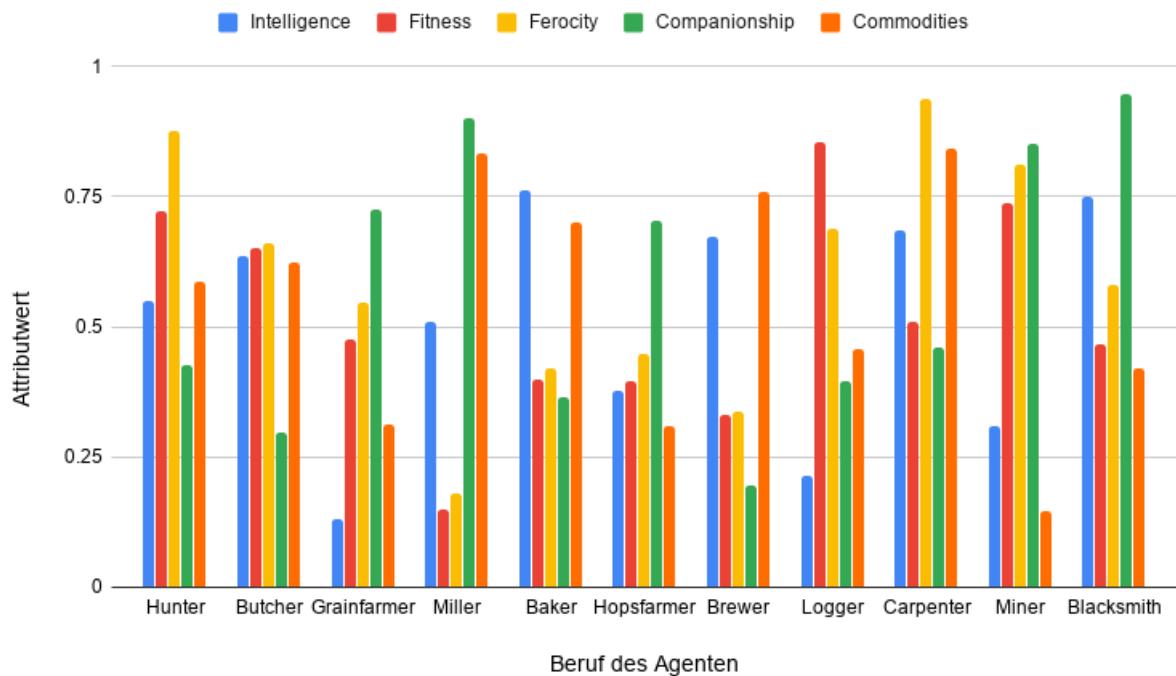


Abbildung 7.14.: Verteilung der Attribute und Berufe der Agenten

Weiterhin wird in der folgenden Grafik 7.15 die durchschnittlichen Höhe der Attribute von Agenten für Recreation-Aktionen dargestellt. In einer Tabelle werden zusätzlich die jeweiligen Attribute festgehalten, die für die Belohnungen der Recreation-Aktionen genutzt werden. Dafür werden die Identifier für die Aktionen und Attribute genutzt. Die Belohnungen für Attribute können invertiert werden, sodass ein hohes Attribut für die Aktion zu einer Strafe führt.

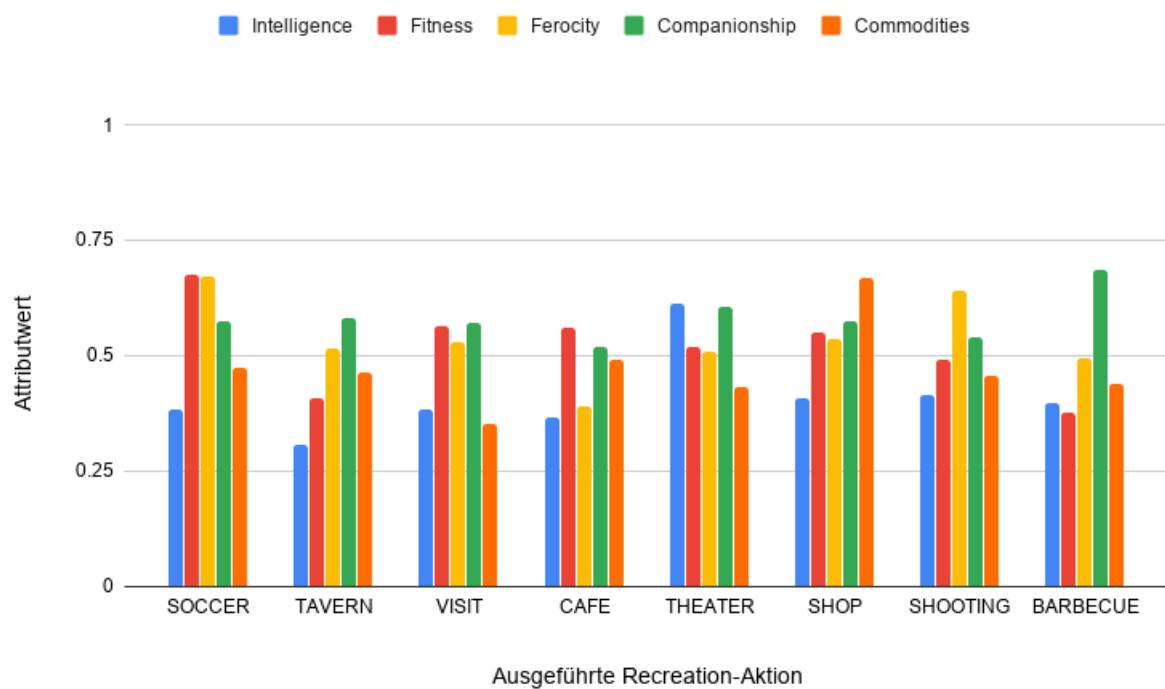


Abbildung 7.15.: Verteilung der Attribute und Berufe der Agenten

7. Training und Ergebnisse

Aktion	Attribute
RECREATION_SOCCER	ATT_FITNESS, ATT_FEROCITY.
RECREATION_TAVERN	ATT_INTELLIGENCE (invertiert), ATT_FITNESS (invertiert).
RECREATION_VISIT	ATT_COMPANIONSHIP, ATT_COMMODITIES (invertiert).
RECREATION_THEATER	ATT_INTELLIGENCE.
RECREATION_CAFE	ATT_COMPANIONSHIP, ATT_FEROCITY (invertiert).
RECREATION_SHOP	ATT_COMMODITIES, ATT_FEROCITY (invertiert).
RECREATION_SHOOTING	ATT_FEROCITY, ATT_COMPANIONSHIP (invertiert).
RECREATION_BARBECUE	ATT_COMPANIONSHIP, ATT_FITNESS (invertiert).

Tabelle 7.3.: Beeinflussung der Belohnungen von Recreation-Aktionen durch Attribute

Die Verteilung der Attribute deckt sich ungefähr mit den Attributboni, die in der obigen Tabelle dargestellt werden. Dadurch wird gezeigt, dass die Attribute der Agenten bei der Wahl der Recreation-Aktion eine Rolle spielen. Agenten mit Attributen, die einen Bonus für die Belohnung einer bestimmten Aktion zur Folge haben, führen diese Aktion also statistisch gesehen häufiger aus als andere Aktionen. Die im Abschnitt 6.4.6.7 angesprochene Einteilung der Agenten in Gruppen wurde daher erreicht. Auch durch simple Beobachtungen lässt sich diese Eigenschaft überprüfen; beispielsweise trifft man in der Taverne tatsächlich häufiger Agenten mit einem niedrigen Intelligenzlevel an, wie etwa Bauern oder Holzarbeiter.

7.6.4.4 Tagesablauf und Bedürfnisverlauf

Die durchschnittlichen Bedürfnisse der Agenten wurden über einen Zeitraum von 100 Simulationstagen beobachtet und zu bestimmten Tageszeiten festgehalten. Schließlich wurden von allen Agenten die mittleren Bedürfniswerte für bestimmte Tageszeiten berechnet, wie in Abschnitt 6.4.9 beschrieben. Daraus hat sich der folgende Bedürfnisverlauf ergeben:

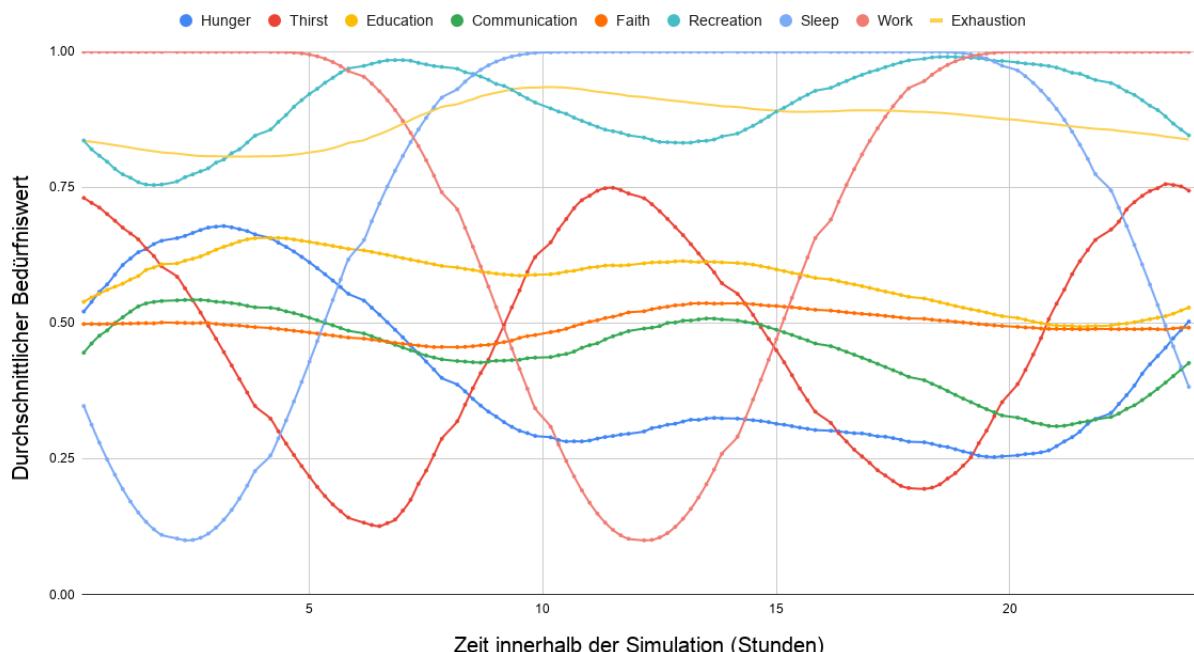


Abbildung 7.16.: RiL-Ansatz, Verlauf der durchschnittlichen Bedürfniswerte

7. Training und Ergebnisse

Zum Verlauf der Bedürfnisse wird angemerkt, dass die Variable `_rythmVariation` zur Erzeugung eines Offsetes zwischen Agenten im `NeedCycleHandler` (siehe Abschnitt 6.4.6.3) für die Aufzeichnung der Daten deaktiviert wurde. Dadurch sollen Unterschiede zwischen den Agenten eliminiert werden. Der gezeigte Verlauf ist zwar repräsentativ für alle Agenten, normalerweise würde für jeden einzelnen Agenten jedoch eine horizontale Verschiebung der Bedürfnisse durch die Verwendung des Offsets entstehen. Weiterhin entsprechen die gewählten Schlaf- und Arbeitszeiten nicht unbedingt dem durchschnittlichen Menschen unserer heutigen Gesellschaft, da bei der Entwicklung der Zyklen mit normalisierten Float-Werten zwischen 0 und 1 gearbeitet wurde. Die Schlaffunktion erreicht ihr Minimum erst um etwa 2:30 Uhr nachts. Für das Verhalten des Agenten spielt dies jedoch keine Rolle. Durch eine simple Transformation könnten die Zyklen verschoben werden, in dieser Analyse werden sie jedoch in der vorliegenden Form akzeptiert.

Weiterhin kann man feststellen, dass sich die Bedürfnisse nicht zufällig entwickeln, sondern von der jeweiligen Tageszeit abhängen. Es sind Kurven entstanden, die Hinweise darauf geben, wann Bedürfnisse befriedigt werden.

Die Hungerfunktion wurde so gewählt, dass sie nach einem halben Tag den Wert 0 erreicht. Am Verlauf des Hungers ist zu erkennen, dass das Bedürfnis in der Nacht zunächst zunimmt und dann sinkt, sobald sich die Agenten schlafen legen. Das Hungergefühl wächst dann automatisch an, weil der Agent mit der Aktion *Schlafen* beschäftigt ist. Über den Tag verteilt bleibt das Hungergefühl auf einem relativ gleichbleibenden aber akzeptablen Level. Das spricht dafür, dass die Agenten zu unterschiedlichen Zeiten essen. Ob der Agent direkt nach dem Aufwachen essen muss, hängt von den zuvor getätigten Aktionen und somit den aktuellen Bedürfnissen des Agenten ab. Das Durstgefühl des Agenten wird hingegen von einer deutlich stärker schwankenden Funktion beschrieben, die einer Sinuskurve ähnelt. Die Durstfunktion wurde so definiert, dass sie etwa drei mal am Tag den Wert 0 erreicht, wenn das Durstgefühl nicht gestillt wird. Tatsächlich ist an dem Graphen zu erkennen, dass die Agenten zwei mal am Tag ihr Durstgefühl stillen - nach dem Aufwachen und nach der Arbeit. Schlussfolgernd muss sich das Durstgefühl des Agenten für eine gewisse Zeit auf dem Wert 0 befinden. Das bedeutet nicht, dass der Agent die falschen Aktionen getroffen hat - zur gleichen Zeit können mehrere Bedürfnisse den Wert 0 annehmen. Die Belohnungen für die Bedürfnisse *Arbeit* und *Schlaf* sind darüber hinaus deutlich höher als die Belohnungen für weitere Bedürfnisse. Das bedeutet, dass ein Agent, der hungrig und durstig ist, sich trotzdem für die Aktion *Arbeit* entscheiden würde, wenn das entsprechende Bedürfnis klein genug ist. Die Belohnungen wurden so gewählt, um einen gleichbleibenden Tagesablauf des Agenten zu gewährleisten. Weiterhin haben alle Aktionen eine gewisse Dauer. Erst nachdem die Dauer der Aktion vorüber ist, kann eine weitere Aktion ausgeführt werden, die ein zwischenzeitlich auf den Wert 0 gesunkenes Bedürfnis befriedigen soll. Solange die erste Aktion und selbst die zweite Aktion noch ausgeführt wird, kann dieses Bedürfnis auf dem Wert 0 stagnieren. Erst nachdem die zweite Aktion beendet wurde, wird das Bedürfnis aufgefüllt. Es ist daher logisch, dass der Agent nur zwei mal am Tag seinen Durst stillt, obwohl das Bedürfnis theoretisch drei mal am Tag auf den Wert 0 sinkt, denn die Ausführung der *richtigen* Aktion ist nicht immer möglich, etwa, weil der Agent mit anderen Dingen beschäftigt ist.

Die Bedürfnisse nach Bildung und Glauben bleiben über den gesamten Tag auf einem relativen ähnlichen Level. Das liegt daran, dass erst nach 1,9 Tagen das Bildungs- und erst nach drei Tagen das Glaubensbedürfnis auf den Wert 0 abgefallen ist. Da Exponentialfunktionen für die Berechnung der Bedürfnisse verwendet wurden, sind die beiden Bedürfnisse für den Großteil dieser Zeit auf einem hohen Level in der Nähe des Wertes 1. Beide Bedürfnisse verzeichnen einen leichten Anstieg um etwa 10:00 Uhr, was darauf hindeutet, dass Agenten um diese Zeit häufig ihr Glaubens- und Bildungsbedürfnis befriedigen. Weiterhin hat das Bildungsbedürfnis einen leichten Anstieg um etwa 22:00 Uhr.

Das Bedürfnis nach Kommunikation verhält sich auf ähnlich wie das Bedürfnis nach Bildung. Dieses Bedürfnis sinkt nach 0,95 Tagen auf den Wert 0 ab. Besonders häufig wird es um etwa 20:30

7. Training und Ergebnisse

befriedigt, nachdem der Agent von der Arbeit gekommen ist und etwas getrunken hat. Analog zu diesem Verlauf sind etwa um diese Zeit in der Simulation viele Dorfbewohner auf dem Marktplatz zu sehen, die sich treffen, um zu kommunizieren. Die *Exhaustion* sinkt über den Tag langsam ab. Sobald die Agenten schlafen, erhöht sich das Bedürfnis automatisch wieder, da die Agenten sich erholen. Es sind keine starken Steigungen zu beobachten, die darauf hinweisen, dass sich Agenten zu bestimmten Zeiten ausruhen.

Sowohl das Schlaf- als auch das Arbeitsbedürfnis werden nicht durch die Aktionen der Agenten beeinflusst. Stattdessen hängen sie direkt von der Zeit in der Simulation ab. Sie erfüllen lediglich den Zweck, die Belohnungen der Agenten von einer Bedürfnisfunktion abhängig zu machen, die Bedürfnisse lassen sich aber nicht auffüllen. Sobald ein Agent schläft oder arbeitet, dauern die entsprechenden Aktionen so lange, dass bei der Beendigung wieder der Wert 1 für das Bedürfnis erreicht wird. Die in der Abbildung 6.24 dargestellten Funktionen sind in der oben zu sehenden Grafik gut wiederzuerkennen. Die Zeitpunkte der Befriedigung der weiteren Bedürfnisse hängt vom Schlaf- und Arbeitsbedürfnis indirekt ab. Durch die Etablierung dieser beiden zeitabhängigen Bedürfnisse wurde das Fundament für den weiteren Tagesablauf der Agenten gelegt. Sie bestimmen, wann der Agent Zeit für weitere Aktionen hat. Durch die hohe Dauer der Aktionen verringern sich weiterhin automatisch stark die Bedürfnisse des Agenten, sobald er arbeitet oder schläft. Es ist daher logisch und notwendig, dass ein NPC direkt nach dem Arbeiten oder Schlafen etwas trinken muss.

7.6.4.5 Dauer der Aktionen

Über einen Zeitraum von 100 Simulationstagen wurde die Dauer aller Aktionen gespeichert. Schließlich wurde aus der Menge der Dauer für eine Aktion der jeweilige Durchschnitt berechnet, wie in Abbildung 7.17 zu sehen.

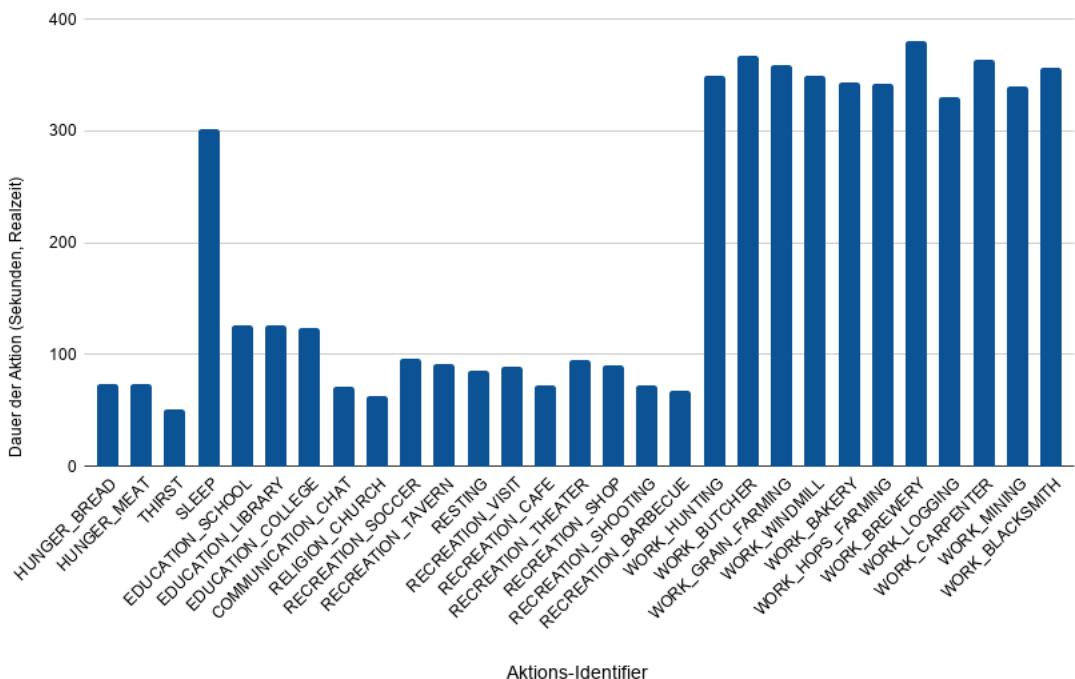


Abbildung 7.17.: Durchschnittliche Dauer der Aktionen

7. Training und Ergebnisse

Die Teilaktionen der ActionPlans wurden so festgelegt, dass Aktionen, die dasselbe Bedürfnis befriedigen, möglichst gleich viel Zeit benötigen. Die Abbildung zeigt, dass die Ausführung der Recreation-Aktionen ähnlich viel Zeit braucht. Weiterhin stimmen die Dauern der beruflichen Aktionen ungefähr überein. Die in den Teilaktionen festgelegten Dauern wurden daher korrekt gewählt.

Die Festlegung der Dauern ist besonders schwierig, weil sich der Startort eines Agenten bei der Ausführung einer Aktion stets ändert. Daher variiert die benötigte Dauer, um zum Zielort zu gelangen. Die Abbildung zeigt, dass die Teilaktionen der ActionPlans gut definiert wurden, da insgesamt die gewünschten Dauern erreicht wurden.

7.6.4.6 Anzahl der Aktionen im Verhältnis

Um einen Eindruck über die relativen Häufigkeiten der Aktionen zu erlangen, wurden über einen Zeitraum von 100 Simulationstagen die Mengen der jeweils ausgeführten Aktionen gespeichert. Dieses Verhältnis wird in der folgenden Abbildung dargestellt:

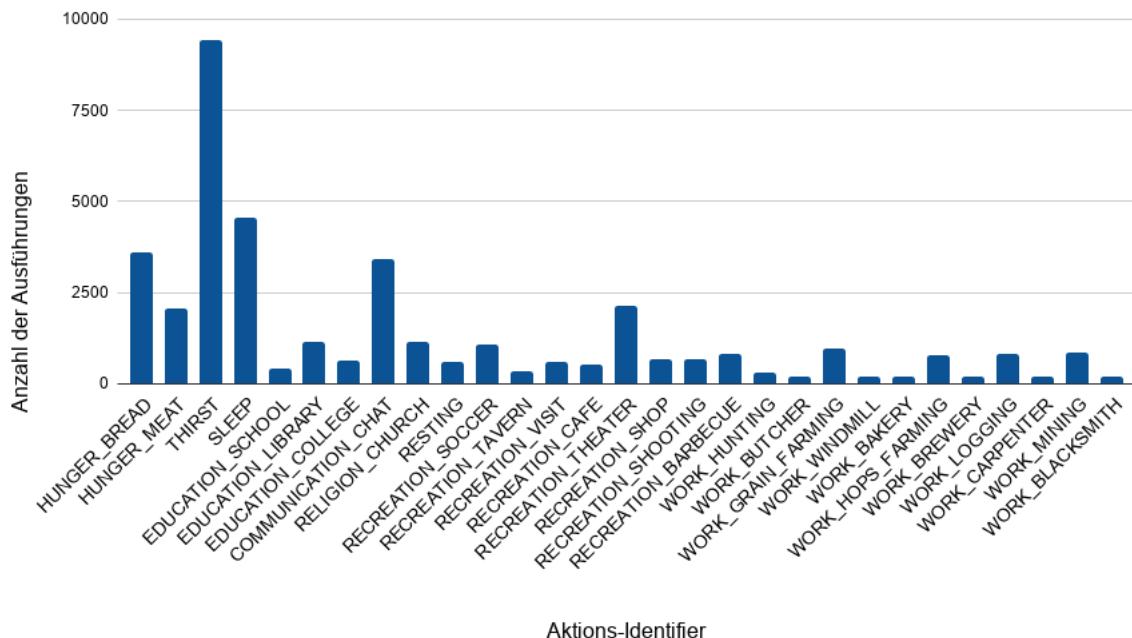


Abbildung 7.18.: Verhältnis der Menge der ausgeführten Aktionen

Insgesamt wurden 38.799 Aktionen von 50 Agenten ausgeführt. Im Durchschnitt hat jeder Agent also ca. 804 Aktionen insgesamt bzw. ca. 7,76 Aktionen pro Tag ausgeführt. Die jeweilige Anzahl der entsprechenden Aktionen deckt sich mit den im `NeedCycleHandler` festgelegten Zyklusdauern sowie den in der Abbildung 7.16 beobachteten Verläufen. Das Bedürfnis *THIRST* wird zum Beispiel deutlich häufiger befriedigt als das Bedürfnis *FAITH*, da unterschiedliche Zyklusdauern festgelegt wurden.

Die Aktion `EDUCATION_LIBRARY` wurde deutlich häufiger als andere Bildungs-Aktionen ausgeführt. Das liegt daran, dass die Altersgruppe für die Bibliothek deutlich mehr Agenten beinhaltet als die des College oder der Schule.

7. Training und Ergebnisse

Auffällig ist lediglich, dass unproportional viele Agenten die Aktion RECREATION_THEATER ausgeführt haben. Diese Anomalie könnte zum einen damit erklärt werden, dass ein Fehler im Programmcode vorliegt, oder zum anderen, dass der Trainingsvorgang nicht ideal verlaufen ist.

7.6.4.7 Maskierte Aktionen

Weiterhin wurde die Anzahl der maskierten Aktionen untersucht. Für die Entscheidungen aller Agenten wurde dafür die Menge der maskierten Aktionen gespeichert. Da NPCs feste Berufe haben und deswegen nur eine berufliche Aktion ausführen können, beträgt die Menge der maskierten Berufsaktionen - mit Ausnahme der Aktion, die zum Job des Agenten gehört - genau der Menge der insgesamt ausgeführten Aktionen. Ähnlich verhält es sich mit den Bildungs-Aktionen, da diese vom Alter des Agenten abhängen. Alle anderen Aktionen wurden im Durchschnitt lediglich zwei Mal maskiert, was bei über 800 ausgeführten Aktionen für das Training nicht nennenswert ins Gewicht fallen sollte.

8

Weitere entwickelte Ansätze

Um einen Vergleich des entwickelten Ansatzes zu weiteren Technologien aufstellen zu können, wurden zwei weitere Verfahren für die Implementierung der Entscheidungsfindung intelligenter Agenten verwendet. Die grundsätzliche Struktur der Dorfsimulation bleibt dabei erhalten. Die Entscheidungsfindung der Agenten dagegen ändert sich maßgeblich. Es wurde ein regelbasierter sowie ein GOAP-basierter Ansatz entwickelt. Im Folgenden werden diese vorgestellt.

8.1 Regelbasierter Ansatz

Für den regelbasierten Ansatz wurde die Klasse `ManualDecisionMaker` erstellt. Der Aufbau der Klasse ist in Abbildung 8.1 zu erkennen:

ManualDecisionMaker
+ string makeDecision
+ bool _isMasked
+ bool _shouldSleep
+ bool _shouldWork
+ bool _shouldRest
+ string _makeNeedDecision
+ string _makeHungerDecision
+ string _makeEducationDecision
+ string _makeThirstDecision
+ string _makeFaithDecision
+ string _makeCommunicationDecision
+ string _makeRecreationDecision
+ string _getPreferredRecreationAction

Abbildung 8.1.: Methoden der Klasse `Manual Decision Maker`

Der `ManualDecisionMaker` übernimmt die Entscheidungsfindung, indem er den Zustand des Agenten betrachtet und auf der Grundlage von bestimmten Regeln die beste Aktion auswählt. Die Klasse

8. Weitere entwickelte Ansätze

ersetzt im Architekturdiagramm (siehe Abbildung 6.6) die Rolle des externen Tensorflow-Backends welches für die Entscheidungsfindung zuständig ist. Die Klasse `NpcAgent` wird also weiterhin verwendet, alle Funktionen des ML-Agents-Frameworks werden aber deaktiviert. Da der `NpcAgent` bereits über alle nötigen Abhängigkeiten, Handler und Komponenten verfügt, wird der generelle Aufbau der Architektur nicht weiter verändert. Die Methode `requestModelOrManualDecision` ruft dann statt der ML-Agents-Funktion `RequestDecision` die Funktion `makeDecision` in der Klasse `ManualDecisionMaker` auf, um abhängig vom Zustand des Agenten Entscheidungen zu treffen. Die Funktion `makeDecision` ist im folgenden Listing 8.1 zu sehen:

```

1  public string makeDecision() {
2      _mask = _npc.ActionMaskingHandler.getStringActionMask(_npc);
3
4      if (_shouldSleep()) return NpcActions.SLEEP;
5      if (_shouldWork()) return _npc.NpcState.AgentInfo.JobId;
6      if (_shouldRest()) return NpcActions.RESTING;
7
8      string action = _makeRecreationDecision();
9      if (action != null) return action;
10
11     action = _makeNeedDecision();
12     if (action != null) return action;
13
14     Debug.LogWarning("Could not find a good action for the agent. Choosing a
15         random action that is not masked instead.");
16     List<string> unmaskedActions = new List<string>();
17     foreach (string actionId in NpcActions.ALL_ACTIONS) {
18         if (!_mask.Contains(actionId)) unmaskedActions.Add(actionId);
19     }
20     if (unmaskedActions.Count != 0) {
21         return unmaskedActions[Mathf.RoundToInt(Random.value * (unmaskedActions
22             .Count - 1))];
23     }
24     Debug.LogWarning("Could not manage to find ANY actions for the agent!
25         Choosing an action that IS MASKED instead.");
26     return NpcActions.ALL_ACTIONS[Mathf.RoundToInt(Random.value * (NpcActions.
27             ALL_ACTIONS.Length - 1))];
28 }
```

Listing 8.1: Funktion `makeDecision` im `ManualDecisionMaker`

Eine Reihe von Bedingungen werden der Reihe nach mit absteigender Priorität abgearbeitet. Unter anderem wird dabei die Ausführbarkeit der Aktion geprüft, indem ermittelt wird, ob diese durch `_mask` maskiert ist. Sobald eine Bedingung zutrifft, gibt die jeweils aufgerufene Funktion den Identifier einer Aktion in Form eines Strings zurück. Wenn der Rückgabewert `null` ist, wird die nächste Bedingung bearbeitet. Zunächst wird geprüft, ob der Agent schlafen, arbeiten oder ruhen muss. Wenn keine dieser Bedingungen zutrifft, wird mittels `_makeRecreationDecision` ermittelt, ob ein niedriger Wert für die Recreation des Agenten vorliegt, sodass dieser eine Freizeitaktivität ausführen sollte. Es werden ebenfalls die Attribute des Agenten mit einbezogen. Eine Liste aus allen Attributen des Agenten wird absteigend sortiert. Mit der Funktion `getPreferredAttributeAction` aus der Klasse `AttributeHandler` wird nun für jedes Attribut in der Liste nacheinander geprüft, ob die durch die Attribute des Agenten präferierte Aktion ausführbar ist.

Falls keine Recreation-Aktion gefunden wurde, werden schließlich die Bedürfnisse des Agenten betrachtet. Die entsprechende Funktion ist in Listing 8.2 zu finden. Ähnlich wie bei den Attributen wird eine Liste von allen Bedürfnissen sortiert, hier jedoch aufsteigend. Der Reihe nach wird dann in einem *switch-case* geprüft, ob die mit dem Bedürfniss verbundenen Aktionen ausführbar sind.

8. Weitere entwickelte Ansätze

```
1  private string _makeNeedDecision() {
2      List<KeyValuePair<string , float>> needs = new List<KeyValuePair<string , float>>();
3      foreach ( string needType in NpcState.ALL_NEEDS) {
4          needs.Add(new KeyValuePair<string , float>(needType , _npcState .
5              getNeedByType(needType)));
6      }
7      needs .Sort((x , y) => x .Value .CompareTo(y .Value));
8
9      foreach (KeyValuePair<string , float> need in needs) {
10         string decision = null;
11         switch (need.Key) {
12             case NpcState.NEED_HUNGER:
13                 decision = _makeHungerDecision();
14                 break;
15             case NpcState.NEED_THIRST:
16                 decision = _makeThirstDecision();
17                 break;
18             case NpcState.NEED_EDUCATION:
19                 decision = _makeEducationDecision();
20                 break;
21             case NpcState.NEED_COMMUNICATION:
22                 decision = _makeCommunicationDecision();
23                 break;
24             case NpcState.NEED_FAITH:
25                 decision = _makeFaithDecision();
26                 break;
27             }
28             if (decision == null) continue;
29         else return decision;
30     }
31 }
```

Listing 8.2: Funktion `makeNeedDecision` im `ManualDecisionMaker`

Wenn auch bei diesem Vorgehen keine gültige Aktion gefunden wird, wird eine zufällige Aktion gewählt, die nicht maskiert ist. Wenn auch das scheitert, wird eine zufällige maskierte Aktion ausgewählt. Alternativ könnte der Agent ebenfalls so lange keine Aktion ausführen, bis eine Aktion verfügbar ist.

8.2 Goal Oriented Action Planning

Weiterhin wurde ein Ansatz entwickelt, der auf GOAP basiert (siehe Kapitel 2.4). Nicht alle Features des Dorfes, die im Kapitel 6 vorgestellt wurden, sind in diesem Verfahren funktionsfähig. Es handelt sich eher um ein Proof-Of-Concept, das das generelle Vorgehen der Implementierung von GOAP dokumentieren soll.

Im Folgenden wird die Architektur des Ansatzes beschrieben. Zunächst wird auf die wichtigsten Komponenten der verwendeten Bibliothek eingegangen.

8.2.1 Verwendete GOAP-Bibliothek

Für die Implementierung wurde die Open Source-Bibliothek *GOAP* genutzt [Klo17]. Zunächst werden die wichtigsten Klassen der Bibliothek beschrieben.

8.2.1.1 GoapAction

Die Klasse `GoapAction` beschreibt eine ausführbare GOAP-Aktion, die bestimmte Bedingungen erfordert und Effekte verursacht. Sie dient als Grundlage für die Erzeugung des Aktionsbaumes, der den Weg von einem Startpunkt über Aktionen zu einem Ziel bzw. Goal beschreibt.

Im Konstruktor der Klasse werden Effekte hinzugefügt. Diese werden in der Klasse `Effects` definiert und im Feld `List<string> effects` gespeichert. Äquivalent dazu werden in der Variablen `preconditions` Bedingungen festgehalten. Weiterhin muss eine Reichweite `requiredRange` festgelegt werden. Später wird dieser Wert für die Navigation des Agenten zum Zielort benutzt. In dem Feld `cost` werden darüber hinaus die Kosten für die Aktion festgelegt. Schließlich muss die Funktion `Perform` überschrieben werden, um die eigentlich Aktion auszuführen - beispielsweise das Warten eines Agenten für einen bestimmten Zeitraum an einem bestimmten Ort.

8.2.1.2 GoapGoal

Um das Ziel eines Agenten zu beschreiben, wird in der Klasse `Goal` ein Identifier für ein neues Goal anhand einer String-Konstante erzeugt. Dieses Ziel wird einer bestimmten Aktion zugewiesen, deren Ausführung das Erreichen des Ziels beschreiben soll. Die Zuweisung erfolgt in dem Feld `goal` der `GoapAction`. Weiterhin muss eine neue Klasse erstellt werden, die von der Klasse `GoapGoal` erbt und direkt mit dem definierten Ziel verbunden ist. Die entsprechende String-Konstante, die das Ziel beschreibt, ist ein Parameter für den Konstruktor des `GoapGoal`. Nun wird eine Liste aller Aktionen erstellt, die das entsprechende Ziel haben.

8.2.1.3 GoapAgent

Die Klasse `GoapAgent` verwaltet alle Aktionen und Goals des Agenten. Der Aktionsbaum für die Entscheidungsfindung des Agenten wird hier aufgebaut. Zunächst werden alle verfügbaren Aktionen mithilfe der Funktion `AddAction` hinzugefügt. Ebenfalls werden alle erzeugten Goals in das Feld `Dictionary<string, GoapGoal>` mit aufgenommen. Für jedes der Goals wird nun anhand der Effekte und Bedingungen der Aktionen ein Aktionsbaum erzeugt. Mithilfe der jeweils definierten Kosten kann dann der kürzeste Weg zum Ziel berechnet werden, der mit den niedrigsten Kosten verbunden ist. Zusätzlich muss die Funktion `Move` überschrieben werden, um eine Navigation des Agenten zum Zielort der Aktionen zu ermöglichen.

8.2.2 Architektur

In der `GameAcademy` kann durch den Parameter `useGoap` bestimmt werden, ob GOAP-basierte Agenten anstelle eines trainierten Modells für RIL-basierte Agenten verwendet werden sollen. Die GOAP-Implementierung nimmt hier, ähnlich wie bereits der `ManualDecisionMaker` im vorherigen Abschnitt, die Rolle des externen Tensorflow-Backends ein, um Entscheidungen für den Agenten in Abhängigkeit von seinem Zustand zu treffen. Obwohl GOAP-basierte Agenten verwendet werden, müssen Agenten vom Typ `NpcAgent` erzeugt werden, da die Komponenten dieser Klasse für die GOAP-Agenten verwendet werden. Lediglich die eigentliche Entscheidungsfindung der `NpcAgents` wird deaktiviert. Dafür wird in der Funktion `requestModelOrManualDecision` der Klasse `NpcAgent` der entsprechende Parameter geprüft. In der Funktion `_initAgent` in der Klasse `Village` wird ein einzelner `NpcAgent` initialisiert. Zusätzlich werden dort GOAP-Agenten vom Typ `NpcGoapAgent` erzeugt, wie im Listing 8.3 zu sehen:

8. Weitere entwickelte Ansätze

```
1 if (GameAcademy.USE_GOAP) {
2     yield return Ninja.JumpToUnity;
3     goapAgent = npcAgent.gameObject.AddComponent<NpcGoapAgent>();
4     VillageGoapActionAdder goapActionAdder = new
5         VillageGoapActionAdder();
6     yield return goapActionAdder.createGoapActions(npcAgent,
7         goapAgent, _villageInfo);
8     goapAgent.Init(npcAgent, goapActionAdder.GoapActions);
9 }
```

Listing 8.3: Erzeugung GOAP-basierter Agenten NpcGoapAgent

Die Funktion `Init` der Klasse `NpcGoapAgent` initialisiert den Agenten mit zwei benötigten Parametern, dem verbundenen `NpcAgent` und einer Liste der möglichen Aktionen `List<GoapAction>`. Weiterhin werden dort alle Goals des Agenten hinzugefügt. Die Methode `Move` wird wie folgt überschrieben und zusätzlich eine Funktion `_moveFinished` hinzugefügt:

```
1 protected override void Move(GoapAction nextAction) {
2     Vector3 moveAgentPosition = _npcAgent.MoveHandler.NavAgent.
3         transform.position;
4     Vector3 moveAgentDestination = _npcAgent.MoveHandler.NavAgent.
5         destination;
6     Vector3 nextActionDestination = nextAction.target.transform.
7         position;
8     if (Equals(moveAgentDestination, nextActionDestination) ||
9         Equals(moveAgentPosition, nextActionDestination)) return;
10    _npcAgent.VillageInfo.changeAmountMoving(true);
11    _npcAgent.MoveHandler.moveAgent(nextAction.target.transform.
12        position, nextAction.GetType().ToString(), null);
13 }
14
15 private void _moveFinished() {
16     _npcAgent.VillageInfo.changeAmountMoving(false);
17 }
```

Listing 8.4: Implementierung der Funktion `Move` in der Klasse `NpcGoapAgent`

Für die Bewegung des Agenten wird also der `MoveHandler` des `NpcAgent` verwendet. Falls das Ziel des Agenten bereits mit dem Ziel der nächsten Aktion übereinstimmt, befindet sich der Agent bereits auf dem Weg zum Ziel, sodass ein erneuter Aufruf von `moveHandler.move` nicht nötig ist. Das gilt auch für den Fall, dass der Agent sich bereits an der Zielposition befindet. Sobald der Bewegungsvorgang abgeschlossen wurde, wird die Funktion `_moveFinished` aufgerufen, um im UI die korrekte Anzahl bewegender Agenten zu zeigen. Zusätzlich wird die Getter-Funktion `ActiveActionInRange` überschrieben:

8. Weitere entwickelte Ansätze

```
1 protected override bool ActiveActionInRange {
2     get {
3         if (ActiveAction == null) return true;
4         float distance = Vector3.Distance(transform.position,
5             ActiveAction.target.transform.position);
6         return distance <= GameAcademy.AGENT_STOPPING_DISTANCE;
7     }
}
```

Listing 8.5: Implementierung der Getter-Funktion `ActiveActionInRange` in der Klasse `NpcGoapAgent`

`ActiveActionInRange` wird regelmäßig aufgerufen um zu prüfen, ob ein Agent an der Zielposition der aktuellen Aktion angekommen ist. Wenn der Abstand des Ziels zum Agenten kleiner ist als die *Stopping Distance* des Agenten `GameAcademy.AGENT_STOPPING_DISTANCE`, so ist die Bewegung abgeschlossen und die Aktion kann ausgeführt werden.

Nun muss eine Reihe von Goals und Aktionen definiert werden, um für die jeweiligen Goals Aktionsbäume aufzubauen. Zunächst wurde für jedes Bedürfnis des Agenten ein Goal definiert. Dafür wurden die Klassen `CommunicationGoapGoal`, `EducationGoapGoal`, `ExhaustionGoapGoal`, `FaithGoapGoal`, `HungerGoapGoal`, `RecreationGoapGoal`, `SleepGoapGoal`, `ThirstGoapGoal` und `WorkGoapGoal` erzeugt, sowie entsprechende Konstanten für die jeweiligen Identifier der Goals. Anschließend wurde für die grundsätzlichen Aktionen des Agenten, die zuvor durch ActionPlans beschrieben wurden, eigene `GoapActions` entwickelt. Da in ActionPlans mehrere Subaktionen genutzt werden können, was in dem Fall von `GoapActions` nicht möglich ist, wurden zusätzlich weitere Aktionen erstellt, um die einzelnen in den ActionPlans definierten Teilaktionen zu modellieren. Die definierten Aktionen können im Unterordner `GoapActions` betrachtet werden.

8.3 Ähnliche Ansätze

Bedürfnisbasierte Agenten kommen in verschiedenen Spielen zum Einsatz. Recherchen legen allerdings die Vermutung nahe, dass es bisher keine Beispiele für die Nutzung von Reinforcement Learning in Kombination mit bedürfnisbasierten Agenten gibt. Es konnte allerdings eine Bachelorarbeit mit dem Titel *Verhaltenssteuerung bedürfnisorientierter NPCs in Rollenspielen mithilfe Neuronaler Netze* gefunden werden, in der überwachtes Lernen genutzt wurde [Sch18]. Der Lernvorgang wurde ebenfalls genutzt, um den Agenten beizubringen, Entscheidungen aufgrund ihrer Bedürfnisse zu treffen. In der Arbeit wird die Bedürfnistheorie nach Maslow berücksichtigt, um den Agenten ein möglichst menschliches Verhalten zu verleihen [Mas81]. Für den Lernprozess wurden auf der Grundlage von Formeln, die von Maslow aufgestellt wurden, Trainingsdaten generiert.

8.4 Vergleich der Ansätze

Im Folgenden werden die drei entwickelten Ansätze im Bezug auf den Entwicklungsaufwand und das resultierende Verhalten der Agenten untersucht und verglichen. Alle Verfahren beruhen auf derselben zugrunde liegenden Architektur der Dorfsimulation. Sie unterscheiden sich lediglich in der Art und Weise, wie die Agenten Entscheidungen fällen, um Aktionen auszuführen. Die Analyse bezieht sich nicht auf die Architektur der gesamten Dorfsimulation, sondern nur auf den jeweiligen Entscheidungsprozess und der jeweils damit verbundenen Implementierung.

8.4.1 Benötigter Entwicklungsaufwand

Im Folgenden wird der nötige Entwicklungsaufwand für die Implementierungen der einzelnen Ansätze betrachtet.

8.4.1.1 GOAP

Die GOAP-Implementierung benötigte einen höheren Entwicklungsaufwand, da bestehende ActionPlans nicht genutzt werden konnten. Diese Änderungen waren notwendig, weil GOAP und die verwendete Bibliothek für die Ausführung von Aktionen eine andere Architektur erwarten.

Ein großes Problem war dabei die Definition von Effekten und Bedingungen für GoapActions. Für das Erreichen eines Ziels werden bei GOAP einzelne Aktionen ausgeführt, die essentiell sind - für einen Bäcker beispielsweise *Hole Mehl* oder *Backe Brot*. Die für die anderen Ansätze verwendeten ActionPlans definieren ganze Sequenzen von Aktionen, die zusammen eine größere Aktion beschreiben. Solche Sequenzen können durch GOAP nicht beschrieben werden, da einzelne Aktionen dort jeweils genau einen Zielort haben und die Auswahl der elementaren Aktionen vollständig durch die Bibliothek übernommen wird. Zum Beispiel werden im ActionPlan `WorkBakeryActionPlan` sequentielle Teilaktionen definiert, sodass der Agent erst zur Bäckerei, dann zum Hauptlager, dann wieder zur Bäckerei und schließlich wieder zurück zum Hauptlager geht. Dadurch soll auf sehr einfache Weise simuliert werden, dass der Bäcker Mehl aus dem Lager holt (vorhandenes Mehl ist eine Voraussetzung für die Ausführung der Aktion), dann Brot backt und schließlich das fertige Brot in das Lager bringt. Wenn man diesen Vorgang in GOAP modellieren möchte, muss eine große Menge von Effekten und Bedingungen eingeführt werden. Zunächst braucht der Bäcker entweder ein eigenes Inventar und für jede Ressource einen Effekt *HAS_RESOURCE* mit der jeweiligen Aktion *FETCH_RESOURCE_FROM_STORAGE*. Für jede Ressource muss geprüft werden können, ob der Bäcker diese bereits in seinem Inventar besitzt. Falls dies nicht der Fall ist, muss die Ressource geholt

8. Weitere entwickelte Ansätze

werden. Anschließend benötigt man die Aktion *BAKE_BREAD* mit einem Effekt *HAS_BREAD* und einer weiteren Aktion *MOVE_RESOURCE_TO_STORAGE*, um das Brot zum Lager zu bringen. In welcher Reihenfolge diese Aktionen ausgeführt werden, ist dabei nicht kontrollierbar. Insgesamt erreicht man schnell eine sehr hohe Anzahl von Effekten, Bedingungen und Aktionen. Diese hohe Anzahl ist dadurch bedingt, dass die Dorfsimulation nicht mit der Idee entwickelt wurde, GOAP zu verwenden. Ein weiteres Problem besteht darin, dass in der verwendeten Bibliothek für jede GoapAction stets nur ein Zielort definiert werden kann. Die Dorfsimulation baut aber darauf auf, dass Gebäude über bestimmte Kapazitäten verfügen und die Zielorte von Aktionen daher dynamisch durch den **LocationHandler** definiert werden. GoapActions erwarten aber bereits bei der Initialisierung einen Zielort. Die entstandene Architektur eignet sich daher insgesamt wenig für die Anwendung von GOAP. Die GOAP-Bibliothek verwendet darüber hinaus die Bibliothek [Spi14] und beruht auf der asynchronen Ausführung von Coroutines. Wegen der Eigenheiten von GOAP und des verwendeten Multithreadings mussten zahlreiche Änderungen vorgenommen werden, um GOAP zu nutzen - beispielsweise im **LocationHandler**.

Konkret traten in der Bibliothek Fehler bei der Erzeugung der Aktionsbäume zum Erreichen von Zielen auf. Es wurden Änderungen im Programmcode der Bibliothek vorgenommen, sodass beispielsweise die Größe der entstandenen Bäume auf einen Maximalwert beschränkt wurden, was die Probleme nur teilweise lösen konnten. Tatsächlich entstanden dadurch im Gegenteil zusätzliche Probleme, da nicht alle eigentlich notwendigen Aktionen für das Erreichen eines Ziels in dem erzeugten Baum vorhanden waren. Es besteht die Vermutung, dass insgesamt zu viele Effekte, Bedingungen und Aktionen verwendet wurden, um effizient Bäume zu erzeugen und zu traversieren. Insgesamt spricht das nicht für die Nutzung von GOAP in größeren Anwendungen wie etwa komplexen Spielen. Zusätzlich ist die Möglichkeit nicht auszuschließen, dass bei der Definition von Aktionen und deren Effekten und Bedingungen Fehler gemacht wurden, die in sich Widersprüche bilden. Potentiell können solche Probleme dazu führen, dass ein Ziel niemals erreicht werden kann, weil eine Bedingung niemals wahr sein kann. Mit einer zunehmenden Anzahl von Effekten etc. entstehen solche Fehler leicht, was ein Risiko bei der Nutzung von GOAP darstellt.

Es hat sich herausgestellt, dass die Nutzung GOAP und speziell der Bibliothek nicht in jedem Anwendungsfall sinnvoll ist. GOAP sollte eher verwendet werden, wenn eine Umgebung mit elementaren Aktionen verwendet wird, deren Ausführungsreihenfolge nicht relevant ist. Erhebliche Probleme bei der Verwendung der Bibliothek haben zusätzlich dazu geführt, dass lediglich ein Prototyp entstanden ist, der nicht vollständig funktionsfähig ist. Stattdessen kommt es zu Performanceproblemen und Fehlersituationen, in denen Agenten aufhören, Aktionen auszuführen. Ab einem gewissen Zeitpunkt kommt die Simulation daher zu einem Stillstand.

Die Theorie von GOAP ist mit der Definition von Goals, Actions, Effekten und Bedingungen simpel. Mit steigender Anzahl nötiger Aktionen, Effekten etc. entsteht jedoch ein komplexes, unüberschaubares Konstrukt, in dem man leicht den Überblick verlieren kann.

8.4.1.2 Regelbasierter Ansatz

Für die Implementierung des regelbasierten Ansatzes war insgesamt wenig Zeit erforderlich, da kein Framework verwendet werden und nur eine Menge von Regeln aufgestellt werden musste. In Abhängigkeit vom Zustand des Agenten werden mit absteigender Priorität die Bedürfnisse des Agenten geprüft, um eine passende Aktion zu finden. Die Regeln werden durch einfache *if-else*-Konstrukte abgefragt. Sobald eine Regel zutrifft, ist klar, welche Aktion ausgeführt werden soll. Bestimmte Komponenten des **NpcAgent**, die für ML-Agents-Entscheidungen benötigt wurden, sind in diesem Fall obsolet. Der **ActionHandler**, der über ein Mapping von Identifizieren für Aktionen zu ActionPlans verfügt, könnte in einem rein regelbasierten Ansatz weggelassen werden, um stattdessen direkt ActionPlans auszuführen.

8. Weitere entwickelte Ansätze

8.4.1.3 Reinforcement Learning-basierter Ansatz

Für die Nutzung des ML-Agents-Frameworks müssen bestimmte Klassen und Funktionen implementiert werden, sodass der Entscheidungsprozess grundsätzlich an eine bestimmte Architektur gekoppelt ist. Diese Architektur ist intuitiv verständlich und einfach zu implementieren. Ein hoher Aufwand ist für die Erstellung der ActionPlans nötig, die hier aber nicht diskutiert werden, da sie nicht Teil des Entscheidungsprozesses sind und zum Beispiel auch vom regelbasierten Ansatz genutzt werden. Insgesamt benötigt die Implementierung des eigentlichen Entscheidungsprozesses nur einen geringen Aufwand. Dafür muss lediglich eine Klasse erzeugt werden, die von `Agent` erbt und wenige Funktionen überschreibt. Ein extrem hoher Aufwand ist jedoch mit dem Training der Agenten und der Wahl der Hyperparameter und Belohnungen verbunden, der durch die Verwendung von Ansätzen, die nicht auf Machine Learning bzw. Reinforcement Learning basieren, wegfällt.

8.4.1.4 Ansatz mit überwachtem Lernen

In dem oben genannten Ansatz (siehe Abschnitt 8.3) wurde ebenfalls eine Dorfsimulation entwickelt, in der intelligente Agenten leben. Im Gegensatz zu dieser Arbeit wurde statt Reinforcement Learning überwachtes Lernen verwendet. Dabei entsteht ein zusätzlicher Aufwand für die Generierung von Trainingsdaten. Diese Gegebenheit stellt ein essentielles Problem bei der Nutzung von überwachtem Lernen dar.

8.4.2 Verhalten der Agenten

Im Folgenden wird das Verhalten der Agenten, das aus den entwickelten Ansätzen resultiert, betrachtet. Der GOAP-Ansatz wird dabei ignoriert, da kein einwandfrei funktionierendes Ergebnis erzeugt werden konnte.

In der oben genannten Bachelorarbeit (siehe Abschnitt 8.3) ist insgesamt ein Verhalten entstanden, das dem Verhalten der auf Reinforcement Learning basierten Agenten in dieser Thesis scheinbar stark ähnelt. Es ist ebenfalls ein Tagesablauf entstanden, der in der Abbildung 8.2 zu sehen ist. Daraus kann abgeleitet werden, dass unterschiedliche Methoden aus Machine Learnings-Bereich für die Steuerung bedürfnisorientierter Entscheidungen gewählt werden können.

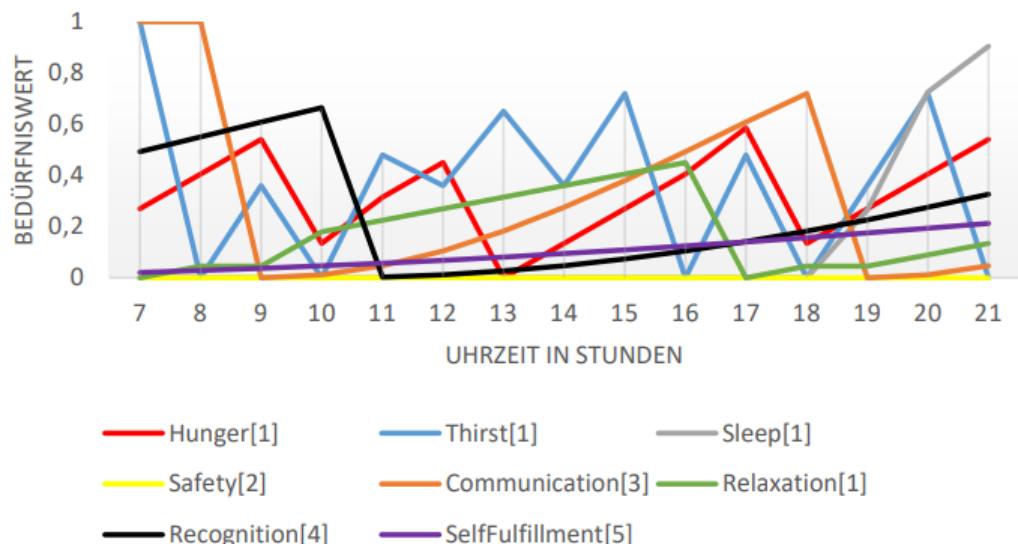


Abbildung 8.2.: Regelbasiertes Verhalten, Verlauf der durchschnittlichen Bedürfniswerte

8. Weitere entwickelte Ansätze

Aus der Verwendung von Regeln ergibt sich theoretisch ein statisches Verhalten, das unter Umständen vorhergesagt werden kann. Da es sich in der Simulation aber um Hintergrundcharaktere mit definierten Aktionen handelt, fallen diese statischen Eigenschaften nicht auf. Da die Bedürfnisse des Agenten nach außen hin nicht sichtbar sind, verhalten sich die Agenten zunächst ähnlich wie die trainierten NPCs. In der Abbildung 8.3 werden die Verläufe der Bedürfnisse gezeigt, die sich beim regelbasierten Ansatz ergeben. Weiterhin zeigt die Abbildung 8.4 einen Vergleich der Häufigkeiten der gewählten Aktionen zum RiL-Ansatz.

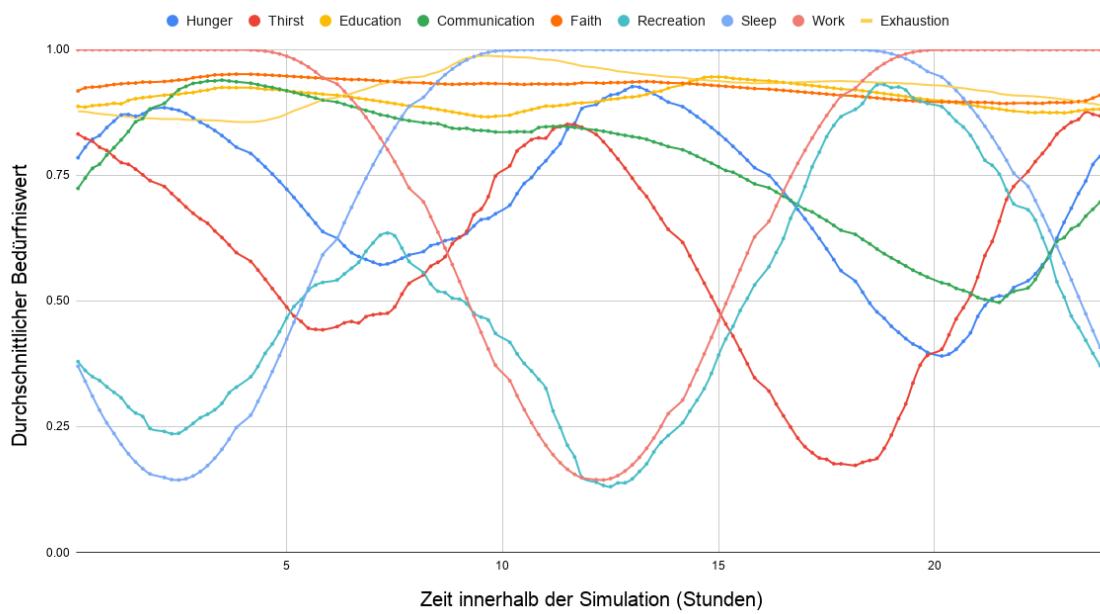


Abbildung 8.3.: Regelbasiertes Modell, Verlauf der durchschnittlichen Bedürfniswerte

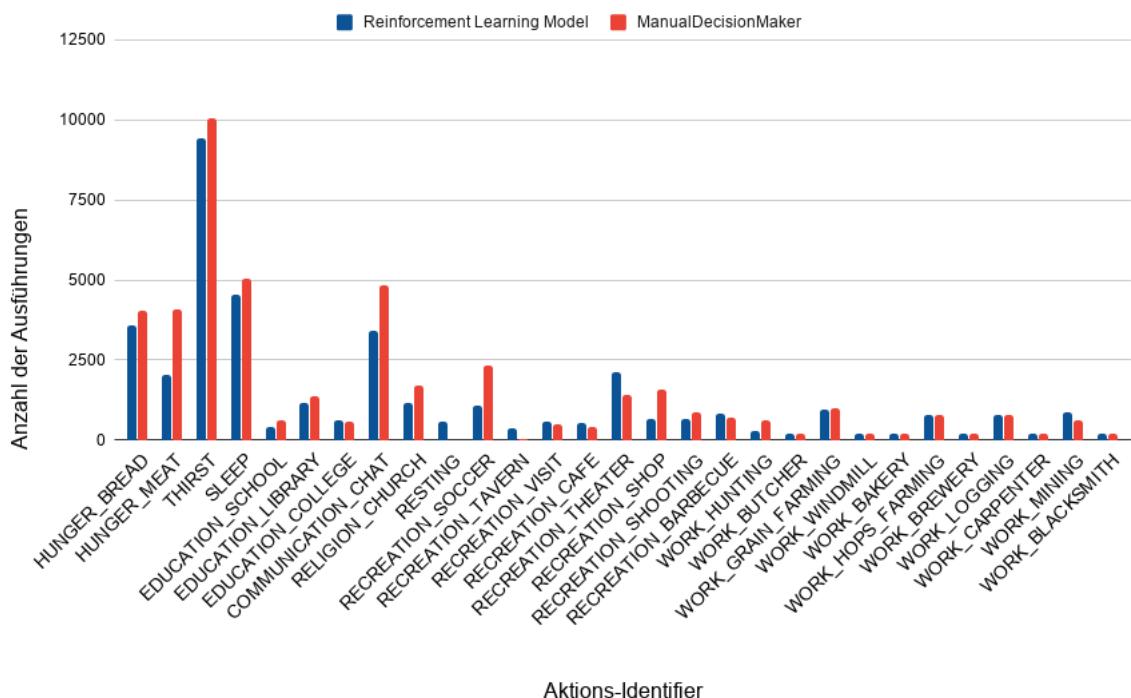


Abbildung 8.4.: Häufigkeiten der Aktionen, regelbasiertes Modell und ML-Agents-Ansatz

8. Weitere entwickelte Ansätze

Die beiden Abbildungen zeigen, dass sich insgesamt ein ähnliches Verhalten entwickelt hat. Ein ähnlicher Tagesablauf ist entstanden und die Häufigkeiten der gewählten Aktionen für beide Ansätze weisen hohe Gemeinsamkeiten auf. Erst sobald man einen regelbasierten NPC für einen längeren Zeitraum näher betrachtet, wird klar, dass sein Verhalten durch statische Regeln beeinflusst wird, da Aktionen sofort ausgelöst werden, sobald bestimmte Bedürfniswerte erreicht wurden. In einem realen Spiel kennt der Spieler die Bedürfnisse der Agenten jedoch nicht. Weiterhin handelt es sich hier um Hintergrundcharaktere. In der Regel würde ein Spieler keine Hintergrundcharaktere für eine so lange Zeit beobachten, dass ihm dieses statische Verhalten aufhalten könnte. Der Vorteil aus der Nutzung von Reinforcement Learning ist in diesem Fall also limitiert, da auch ein regelbasiertes Ansatz zu einem akzeptablen Ergebnis führt. Der größte Vorteil ergibt sich vielmehr aus der Tatsache, dass die Agenten Entscheidungen aufgrund von sich verändernden Bedürfnissen treffen, die wiederum eine Auswirkung auf die Bedürfnisse des Agenten haben. Dadurch wird glaubwürdig das menschliche Verhalten imitiert. Die Nutzung von Reinforcement Learning bietet jedoch Vorteile über das eigentliche Verhalten der Agenten hinaus an. Zwar ist der Ansatz mit einem höheren initialen Aufwand verbunden, doch resultiert daraus ein Verfahren, das sehr gut erweiterbar ist. Da der Trainingsvorgang die Auswahl von Aktionen übernimmt, können nun leicht neue Aktionen und Bedürfnisse hinzugefügt werden. Der initiale Aufwand für die Erzeugung eines regelbasierten Ansatzes ist geringer, allerdings ist dabei eine geringere Erweiterbarkeit gegeben. Je mehr Regeln definiert werden müssen, desto höher ist zudem die Wahrscheinlichkeit, dass Regeln sich widersprechen oder durch die hohe Anzahl Fehler im Programmcode eingebaut werden. Weiterhin muss stets eine Priorisierung von Bedürfnissen vorgenommen werden, was im RiL-Prozess automatisiert und intrinsisch passiert. Der Lernvorgang übernimmt in gewisser Weise die beim RiL-Ansatz die Definition der Regeln.

Für beide verwendete Ansätze ergibt sich eine hohe Skalierbarkeit, da Entscheidungen pro Agent in einem Abstand von mehreren Sekunden getroffen werden müssen. Die zusätzlich erforderliche Leistung der Nutzung einer höheren Anzahl von Agenten ist daher überschaubar. In einem Test wurden bis zu 1.000 Agenten innerhalb des Dorfes verwendet, die zwar die Kapazitätsgrenzen der Gebäude in der Stadt deutlich überschreiten, aber ohne drastische Performanceverluste dargestellt werden können.

9

Fazit

In diesem abschließenden Kapitel wird zunächst eine Zusammenfassung der weiteren Kapitel und der Ergebnisse der Arbeit vorgenommen. Darauf folgen eine Bewertung und eine Analyse von Problemen mit Machine Learning in Spielen sowie spezifischen Problemen des entwickelten Verfahrens. Ein Ausblick bildet anschließend den Abschluss der Arbeit.

9.1 Zusammenfassung

Im ersten Kapitel wurde nach einer Einleitung in das Thema mit einer Vorstellung der Beweggründe für die Arbeit die allgemeine Idee der Thesis vorgestellt, die die Entwicklung einer Dorfsimulation beinhaltet, die auf Reinforcement Learning basiert.

Mehrere Grundlagenkapiteln vermittelten anschließend nötiges Basiswissen. Zunächst wurde auf die Entwicklung intelligenter Agenten in Videospielen eingegangen. Es folgte ein Überblick über mögliche Technologien zur Implementierung intelligenter Agenten. Die vorgestellten Technologien wurden im Einzelnen erläutert.

Anschließend folgte eine Einführung in das Reinforcement Learning mit einer umfassenden Vorstellung des Verfahrens und einer Rekapitulation der grundlegenden Vorgehensweise von Machine Learning. Auch mathematische Aspekte wurden beispielsweise durch die Erläuterung von Markov-Entscheidungsproblemen berücksichtigt. Verschiedene modellfreie und modellbasierte Algorithmen für die Implementierung von Reinforcement Learning wurden anschließend beschrieben.

Im nächsten Kapitel wurden einige Beispiele für die Nutzung von Machine Learning in Spielen vorgestellt. Diese Beispiele verdeutlichten das Potential von Machine Learning im Bereich der Videospiele. Es wurde gezeigt, dass in diesem Bereich bereits erfolgreiche Projekte existieren, die Machine Learning nutzen, und dass dadurch das Spielerlebnis verbessert werden kann.

Im anschließenden Kapitel wurden durch die Vorstellung des verwendeten Frameworks Grundlagen für die Implementierung der Dorfsimulation geschaffen. Ein Beispielprojekt wurde implementiert, das die grundlegende Struktur und den Umgang mit dem Framework verdeutlicht.

Anschließend wurde die Implementierung der Dorfsimulation vorgestellt. Zunächst wurden dafür Anforderungen aufgestellt. Danach folgte eine Beschreibung der nötigen Modellierung des Dorfes. Die grundlegende Architektur und das Software-Design der Anwendung wurde beschrieben. Schließlich wurde die Implementierung anhand der einzelnen Klassen und Komponenten der Simulation umfassend erläutert.

Im nächsten Kapitel folgte die Beschreibung des Trainingsvorgangs für das in der Simulation genutzte Reinforcement Learning-Verfahren. Dabei wurde der generelle Ablauf des Trainings und die Auswahl der optimalen Hyperparameter ausführlich beschrieben. Verschiedene Metriken dokumentierten dabei das entstandene Verhalten der Agenten. Es wurde gezeigt, dass die Agenten sich sinnvoll verhalten.

9. Fazit

Schließlich wurden zwei weitere entwickelte Ansätze für die Nutzung intelligenter Agenten in der Dorfsimulation, die nicht auf Machine Learning basieren, vorgestellt. Ein Vergleich der drei Ansätze wurde angefertigt.

9.2 Bewertung

In der vorliegenden Arbeit sollte ein neuer Ansatz für die Erzeugung glaubwürdiger NPCs in Videospielen vorgestellt werden, die über menschliche Bedürfnisse verfügen und diese als Grundlage für einen intelligenten Entscheidungsprozess nutzen. Mithilfe von Reinforcement Learning sollten die Agenten lernen, ihre Bedürfnisse zu maximieren. Dadurch sollte ein authentisches Verhalten erreicht werden, dass die NPCs menschlicher wirken lässt. Dieser Ansatz wurde anhand einer Dorfsimulation umgesetzt, in der NPCs leben. Diese NPCs verfügen über Bedürfnisse und nutzen diese als Grundlage für ihre Entscheidungsfindung, die mit Reinforcement Learning implementiert wurde. Insgesamt ergibt sich dadurch ein lebendiges Dorf, das mit überzeugenden Agenten gefüllt ist. Die grundsätzlichen Ziele der Arbeit wurden damit erfüllt.

In der Einleitung der Thesis wurde von statischen NPCs gesprochen, die in stets gleiche Verhaltensmuster verfallen, wie beispielsweise in *The Witcher 3* oder *The Elder Scrolls V: Skyrim*. In *The Witcher 3* werden für die Befüllung der Welt Hintergrundcharaktere verwendet, die jedoch über keine Persönlichkeit verfügen, sondern je nach Bedarf erzeugt werden, um eine bestimmte Aktion wieder und wieder auszuführen. In *The Elder Scrolls V: Skyrim* werden keine namenlosen Hintergrundcharaktere verwendet, doch auch hier demonstrieren die Agenten häufig kein intelligentes Verhalten. Mit dem entwickelten, bedürfnisbasierten Ansatz ist eine Anwendung entstanden, die eine Mischung aus den beiden genannten Ansätzen für NPCs darstellt. Effektiv handelt es sich um Hintergrundcharaktere, die eine Umgebung mit Leben befüllen sollen. Ab dem Startzeitpunkt der Anwendung wird jedoch eine feste Anzahl von Agenten verwendet, die bis zum Beenden der Anwendung bestehen bleibt, sodass diese nicht nach Bedarf hinzugefügt und entfernt werden können. Alle NPCs verfügen über eine begrenzte eigene Persönlichkeiten, die durch Attribute beschrieben werden, und einen persistenten Wohnort.

Im Unterschied zu den genannten Beispielen ergibt sich für die entwickelten Agenten jedoch ein weniger statisches Verhalten. Stattdessen treffen die Agenten kontinuierlich Entscheidungen, die nicht auf Scripting, sondern auf ihren aktuellen Bedürfnissen beruhen. Dadurch, dass NPCs unterschiedlichen Berufen nachgehen und ihre Freizeitaktivitäten auf Basis ihrer persönlichen Präferenzen wählen, wird ein dynamisches Verhalten erreicht. Da ihre Aktionen auf menschlichen Eigenschaften basieren und die Agenten eigenständig einen Tagesablauf entwickelt haben, wirken ihre gewählten Entscheidungen glaubwürdig. Insgesamt erweckt das Dorf einen belebten Eindruck und die Agenten wirken authentisch. Das dynamischere Verhalten der Agenten ist nicht nur auf Reinforcement Learning, sondern auch zu großen Teilen auf die Bedürfnisorientierung der Agenten zurückzuführen. Im Abschnitt 8.4.2 wurde gezeigt, dass auch durch einen regelbasierten Ansatz ein Verhalten entstehen kann, das bis zu einem gewissen Grade ausreichend dynamisch ist, wenn bedürfnisorientierte Agenten verwendet werden. Aufgrund des initial niedrigeren Aufwands eines solches Verfahrens spricht diese Tatsache nicht für die Verwendung von Reinforcement Learning. Allerdings ergibt sich dadurch auch eine hohe Erweiterbarkeit der Anwendung, da der Entscheidungsprozess der Agenten durch das Modell übernommen wird und nicht explizit definiert werden muss, wodurch mehr Aufwand und Fehler entstehen können. Die hohe Erweiterbarkeit bietet nicht nur einen Vorteil gegenüber regelbasierten Ansätzen, sondern auch für weitere klassische Verfahren für die Entwicklung intelligenter Agenten, wie zum Beispiel Finite State Machines. Auch hier führt das Einführen neuer Aktionen und Bedürfnisse oder die Veränderung von bestehenden Komponenten zu einem höheren Entwicklungsaufwand und einer höheren Fehleranfälligkeit, da der Entwickler die eigentliche Entscheidungsfindung im Gegensatz

9. Fazit

zum Reinforcement Learning selbst implementieren muss. Insgesamt wurde außerdem gezeigt, dass Reinforcement Learning eine Alternative für die Entscheidungsfindung von NPCs darstellt. Durch eine hohe Erweiterbarkeit bieten sich Vorteile gegenüber den in Kapitel 2 genannten klassischen Verfahren. Insgesamt kann außerdem ein dynamischeres, authentischeres Verhalten erreicht werden, das nicht auf fest definierten Regeln beruht. Dafür muss ein häufig zeitintensiver Trainingsvorgang in Kauf genommen werden, dessen Ergebnis vorher nicht bekannt ist und der für jede Änderung des Programmcodes erneut ausgeführt werden muss.

9.2.1 Probleme mit Machine Learning und Reinforcement Learning

Im folgenden Abschnitt wird auf generelle Probleme aufmerksam gemacht, die sich aus der Verwendung von Machine Learning für intelligente Agenten ergeben. Weiterhin werden Probleme beschrieben, die während der Arbeit angetroffen wurden.

9.2.2 Beschaffenheit des Aktionsraumes

Die Art und Weise, in der Reinforcement Learning im Kontext dieser Thesis verwendet wurde, unterscheidet sich teilweise von den klassischen Anwendungsformen der Technologie. In typischen Anwendungsszenarien gibt es einen Agenten in einer Umgebung, in der die Kombination von beliebigen, unterschiedlichen Aktionen zu einem Ziel führt, das der Agent erreichen muss. Es ist nicht garantiert, dass jede Aktion des Agenten eine Belohnung oder eine Bestrafung zur Folge hat. Der Agent lernt, in Abhängigkeit seiner Zustandsvariablen eine Strategie zu entwickeln, die zu einem ganz bestimmten Ziel führt. In der Dorfsimulation hingegen gibt es solche Ziele nicht. Während des Trainingsvorgangs wird dort die laufende Episode nach einer beliebigen Anzahl ausgeführter Aktionen unterbrochen, anstatt nach dem erfolgreichen Erreichen eines Ziels. Statt durch das Erreichen eines Ziels führt hier jede einzelne Aktion zu einer Belohnung oder einer Bestrafung aufgrund der Bedürfnisse des Agenten. Die Aktionen, die dem Agenten zur Verfügung gestellt werden, sind dabei durch ActionPlans genau vorgegeben und beinhalten bereits exakt beschriebene Abläufe. In der Regel werden jedoch viel allgemeinere Aktionen vorgegeben, wie beispielsweise in der in Kapitel 5 vorgestellten Beispieldurchsetzung für die Lösung eines Problems durch das ML-Agents-Framework. Dort wurden lediglich die Aktionen *rechts/links bewegen* und *vorwärts/rückwärts bewegen* definiert, bis der Agent schlussendlich sein Ziel erreicht. Durch die Definition von allgemein gehaltenen Aktionen ermöglicht man es dem Agenten, eine eigene Strategie zu entwickeln. So entsteht die theoretische Möglichkeit, komplexe Muster, die vor dem Trainingsvorgang nicht bekannt waren, erlernen zu können. In einem Projekt der Firma OpenAI, in dem Agenten sich vor anderen Agenten verstecken müssen, wurde zum Beispiel gezeigt, dass mithilfe von simplen Aktionen wie *Schieben* und *Bewegen* äußerst komplexes Verhalten erreicht werden kann [BB19b] [BB19a]. Ohne jegliche Instruktionen haben die sich versteckenden Agenten sogar gelernt, ihre Umgebung so zu manipulieren, dass sie sich selbst in einem Käfig von Objekten einsperren, damit sie von den suchenden Agenten nicht gefunden werden können. Auch in der Dorfsimulation lernen die Agenten erfolgreich, eine Belohnungsfunktion zu maximieren. Durch die exakte Vorgabe und Beschreibung der möglichen Aktionen durch ActionPlans wird ihnen jedoch die Möglichkeit genommen, komplexere Verhaltensmuster zu entwickeln. Man entzieht den Agenten sozusagen ihre *Kreativität*, indem genaue Vorgaben gemacht werden. Ein wirklicher Lernvorgang im Bezug auf das intelligente Erlernen von Verhaltensmustern ist hier weniger gegeben. Stattdessen findet eher ein rein mathematischer Vorgang statt, indem Beziehungen zwischen Bedürfnissen und gesammelten Belohnungen gefunden werden.

Für die Entwicklung der Entscheidungsfindung von Hintergrundcharakteren ist das gewählte Verfahren trotzdem angemessen. Wenn alle Teilaktionen der ActionPlans in einzelne Aktionen konvertiert oder generell ein Aktionsraum mit einem deutlich höheren Level von Abstraktion gewählt worden

9. Fazit

wäre, hätte das die Komplexität des Problems deutlich erhöht und damit das Training erschwert. Beispielsweise war für das Training des genannten Versteckspiels der Firma OpenAI ein 98,8 Stunden langer Trainingsvorgang mit 167,4 Millionen abgeschlossenen Episoden nötig, bis eine Konvergenz des Modells erreicht wurde [BB19a]. OpenAI Five wurde sogar über mehrere Monate hinweg unter großem Hardwareaufwand trainiert, bis übermenschliches Verhalten erreicht werden konnte [CB19]. Es muss daher ein Kompromiss aus der *Intelligenz* der Agenten, den Möglichkeiten des Entwicklers und den Anforderungen gefunden werden.

9.2.3 Dauer der Trainingsvorgänge

Wie im vorherigen Abschnitt erwähnt, ist ein großes Problem bei der Entwicklung von Agenten mittels Machine Learning die notwendige Zeit, die für das Training und die Feinabstimmung von Belohnungen und Hyperparametern aufgewendet werden muss. Entscheidungsbäume, State Machines oder weitere Verfahren (siehe Kapitel 2) verlangen zwar unter Umständen mehr Zeit für die eigentliche Implementierung des Programmcodes, danach ist die Entwicklung der Agenten jedoch größtenteils abgeschlossen. Wenn Machine Learning-Verfahren wie Reinforcement Learning verwendet werden, muss auch nach der Anfertigung des Programmcodes noch Zeit und Rechenleistung investiert werden, um die Agenten effektiv zu trainieren. Hier muss also klar erörtert werden, was die eigentlichen Anforderungen an die Anwendung sind. Wenn keine hohen Voraussetzungen an die Intelligenz der Agenten gestellt werden, ist die Wahl einer State Machine mit hoher Wahrscheinlichkeit sinnvoller als die Nutzung von Reinforcement Learning. Wenn die Agenten jedoch den Eindruck erwecken sollen, sich an bestimmte Situationen anzupassen zu können und generell kluge Entscheidungen treffen sollen, kann Machine Learning einen großen Vorteil darstellen. OpenAI hat mit der Entwicklung von *OpenAI Five* gezeigt, dass durch eine durchdachte Anwendung von Reinforcement Learning übermenschliche Leistungen erbracht und Agenten mit extrem intelligentem Verhalten entwickelt werden können [Ope18a] (siehe Abschnitt 4.3). Um diese Erfolge möglich zu machen, waren jedoch ebenfalls eine große Menge von Hardware und Zeit erforderlich. Die Dauer des Trainings stellt also ein grundlegendes Problem dar. In der vorliegenden Arbeit bestand ebenfalls die Schwierigkeit, dass viele Trainingsvorgänge durchgeführt werden mussten. Dieses Verfahren beschreibt keine Notwendigkeit, sondern dient vielmehr zur Auffindung des optimalen Ergebnisses durch die Erzeugung einer Policy mit der höchsten Gesamtbelohnung und dem besten resultierenden Agentenverhalten. Ein weiterer legitimer Ansatz wäre es gewesen, das erste *brauchbare* Modell zu akzeptieren, das annehmbare Ergebnisse für das Verhalten der Agenten hervorbringt.

Weiterhin bringt die Verwendung der GPU für Berechnungen im neuronalen Netz keine Vorteile. Der limitierende Faktor ist nicht das Netz selbst, sondern die in Unity laufende Simulation, die einen großen Overhead erzeugt. Durch die parallele Verwendung mehrerer Dörfer kann dieses Problem nur bis zu einem gewissen Grad reduziert werden.

9.2.4 Nichtdeterminismus als Hindernis

Aus der Nutzung von Machine Learning resultiert unweigerlich ein nichtdeterministisches Verhalten. Für die Entwicklung von NPCs stellt das ein Problem dar. Wenn für ein Spiel Agenten entwickelt werden, mit denen der Spieler interagieren kann, so agieren diese typischerweise deterministisch. In den meisten Spielen müssen bestimmten Regeln folgt werden, damit Game Designer und Entwickler das Verhalten der Agenten vorhersagen und auf bestimmte Situationen anpassen können. Die Agenten müssen leicht zu kontrollieren sein und über ein verlässliches und vorhersehbares Verhalten verfügen. Game Designer haben klare Vorstellungen für das Verhalten eines NPC. Wenn dieser nicht über genau das modellierte Verhalten verfügt, behindert das die Entwicklung des Spiels. Daher erscheint es nicht möglich, das gesamte Verhalten eines komplexen NPCs mit Verfahren wie

9. Fazit

Reinforcement Learning zu modellieren - es sei denn es handelt sich um Hintergrundcharaktere, wie in dieser Thesis. Diese dienen hauptsächlich zum Befüllen der Welt, sodass sie kein konkretes Verhalten an den Tag legen müssen.

Machine Learning-Verfahren stellen häufig eine „Black Box“ dar. Der genaue Entscheidungsprozess des Verfahrens ist für den Entwickler daher nicht immer nachvollziehbar, sodass das resultierende Verhalten nicht genau vorhergesagt werden kann. Stattdessen müssen Nicht-Hintergrund-NPCs jedoch kontrollierbar und berechenbar sein. Wenn ein NPC zum Beispiel für das Storytelling eines Spiels relevant ist, dann muss er sich auch immer so verhalten, wie der Game Designer, der die Geschichte entwirft, es vorsieht. Statt des gesamten Verhaltens des NPCs könnte nur ein Teil davon durch Machine Learning geregelt werden. So könnte er etwa Entscheidungen aufgrund von Bedürfnissen treffen, sobald er sich in einem Idle-Modus befindet und aktuell nicht für andere Komponenten des Spiels benötigt wird.

9.2.5 Probleme mit ML-Agents

Während der Implementierung der Dorfsimulation kam es zu Problemen mit dem ML-Agents Framework, besonders während der Ausführung der Trainingsvorgänge. Ein bekanntes Problem bei der Nutzung des Frameworks unter Windows ist, dass das Training aus unbekannten Gründen anhält. Dieser Fehler kommt häufig vor, wenn das Training im Hintergrund ausgeführt wird, also wenn das entsprechende Fenster minimiert ist. Der Vorgang kann dann wieder gestartet werden, indem das Fenster geöffnet und eine beliebige Taste gedrückt wird. Da für das Training ohnehin viel Zeit investiert werden muss, stellt dieses Problem einen stark störenden Faktor dar.

Ähnliche Probleme entstehen, wenn ein Trainingsvorgang mit einer exportierten Unity-Anwendung läuft und zeitgleich im Unity-Editor gearbeitet wird. Das Training stürzt dann manchmal vollständig ab, auch im Editor kommt es zu Fehlermeldungen. Da es sich bei dem Framework noch um eine Beta-Version handelt, sind Fehler allerdings zu erwarten.

9.3 Ausblick

In diesem Abschnitt werden weitere, zukünftige Möglichkeiten zur Verbesserung der entwickelten Agenten und zur Anwendung von Machine Learning in Spielen vorgeschlagen.

Die entwickelte Dorfsimulation beinhaltet Agenten, die individuell handeln. Sie haben kein Gemeinschaftsgefühl, sondern kennen lediglich ihre eigenen Bedürfnisse. Interessant wäre die Entwicklung eines Gemeinschaftsgefühls, sodass die Agenten auf gemeinsame Ziele hinarbeiten. Für das Projekt OpenAI Five (siehe Abschnitt 4.3) wurde ein Hyperparameter *team spirit* eingeführt, sodass Agenten von den Belohnungsfunktionen anderer Agenten beeinflusst werden [Ope18a]. Die Erfolgsquote der Agenten konnte damit stark verbessert werden. Die Nutzung eines solchen Konzepts in einem Dorfsystem könnte dazu führen, dass Agenten für ein übergeordnetes Ziel zusammenarbeiten oder sich gegenseitig helfen.

Wenn eine sinnvoller Ansatz für die Implementierung eines Ressourcensystems gefunden wird, könnten außerdem mehrere Dörfer parallel verwendet werden, die miteinander interagieren können. Interessant wären nun die möglichen Auswirkungen eines Gemeinschaftsgefühls, wie oben beschrieben. Dadurch könnte der Versuch der Agenten entstehen, den Erfolg des eigenen Dorfes zu steigern. Durch Interaktionen zwischen den Dörfern wie Handel oder Krieg könnten dann interessante Verhaltensweisen entstehen.

9. Fazit

Um die Entwicklung von komplexeren Verhaltensmustern anzuregen, könnten allgemeinere Aktionen für die Agenten definiert werden, wie in Abschnitt 9.2.2 vorgeschlagen. Das würde zwar die Komplexität des zugrunde liegenden Optimierungsproblems erhöhen, aber gleichzeitig die Möglichkeit bieten, ein intelligenteres Verhalten entstehen zu lassen. Interessant wäre es, dem Agenten auch die Navigation innerhalb des Dorfes zu überlassen, ähnlich wie im vorgestellten Beispiel in Kapitel 5. Bestimmte Aktionen würden dann erst verfügbar sein, wenn der Agent sich in der Nähe eines Gebäudes befindet. In diesem Fall müssten bestimmte Variablen in den Vektor von Observations mit aufgenommen werden, wie die Orte im Umkreis des Agenten und deren Positionen sowie der Position des Agenten.

Vorstellbar wäre auch die Entwicklung eines Ökosystems, in dem sich neben Dorfbewohnern auch Tiere aufhalten, die primitive Bedürfnisse erfüllen müssen. Beispielsweise könnten Jäger aus dem Dorf diese Tiere jagen, wobei die Tiere versuchen zu überleben.

Insgesamt könnte man äußere Einflüsse in Simulationen integrieren. Um ein absolut glaubhaftes Verhalten der Dorfbewohner zu erzeugen, müssen diese auf Aktionen des Spielers und auf Umstände aus der Umgebung reagieren. Diese sind nicht Teil der Arbeit, da es sich nicht um ein Spiel handelt, sondern um eine Simulation, in der die Bedürfnisse der Agenten im Vordergrund stehen. Zum Beispiel könnte es die Möglichkeit geben, dass sich Wetterbedingungen und Jahreszeiten verändern. Dies hätte etwa Auswirkungen auf die Ernte, sodass die NPCs lernen müssen, andere Nahrungsquellen zu finden oder im Sommer Vorräte anzuhäufen. Außerdem könnten die Agenten lernen, auf die Aktionen eines Spielers zu reagieren - zum Beispiel könnte dieser einen Diebstahl begehen oder einen Dorfbewohner angreifen. Hier könnte wieder das vorgeschlagene Gemeinschaftsgefühl genutzt werden. Eine weitere Möglichkeit besteht in der Implementierung von Moralvorstellungen, die abhängig von der Persönlichkeit eines Agenten sind.

Um das Verhalten der Agenten besser beurteilen zu können, wäre die parallele Ausführung von Agenten mit unterschiedlichen Verfahren zur Entscheidungsfindung denkbar. In diesem Fall würden zum Beispiel GOAP-Agenten, RiL-Agenten und regelbasierte Agenten in demselben Dorf leben, um einen einfachen Vergleich erstellen zu können.

Die meisten vorgeschlagenen Ideen gehen deutlich über die in dieser Arbeit entwickelten Möglichkeiten hinaus, bieten aber interessante Ansatzpunkte für weitere Forschungsarbeit. Dazu wäre voraussichtlich deutlich leistungsstärkere Hardware für den Trainingsvorgang nötig.

In den vorherigen Abschnitten wurde bereits gezeigt, dass Machine Learning sich nicht für die vollständige Modellierung des Verhaltens bestimmter NPCs eignet. Dazu gehören alle NPCs, die für den Spielfortschritt relevant sind und sich deterministisch verhalten müssen. Für diese Agenten könnten höchstens Teile ihrer Entscheidungsfindung dadurch implementiert werden. Stattdessen könnten zukünftig Verfahren wie Reinforcement Learning für die Entwicklung von Hintergrundcharakteren entwickelt werden. Besonders in Verbindung mit der Modellierung von menschlichen Eigenschaften wie Bedürfnissen ist die Kombination der Verfahren in diesem Anwendungsbereich vielversprechend. Auch in kompetitiven Online-Spielen zeigt Reinforcement Learning für die Zukunft ein hohes Potential.

A

Anhang A - Wahl der optimalen Hyperparameter

Dieser Anhang beinhaltet die Dokumentation der Auswahl der optimalen Hyperparameter, die nicht zum gewünschten Erfolg geführt hat.

Lernrate

Zunächst wurde die Lernrate angepasst. Neben der bereits definierten Lernrate von $4 * 10^{-4}$, beziehungsweise $4e - 4$ in der Terminologie der ML-Agents-Konfigurationsdatei, wurden eine geringere Lernrate von $5e - 5$ und eine höhere Lernrate von $8e - 3$ verwendet. Die Ergebnisse sind in den folgenden Abbildungen zu erkennen.

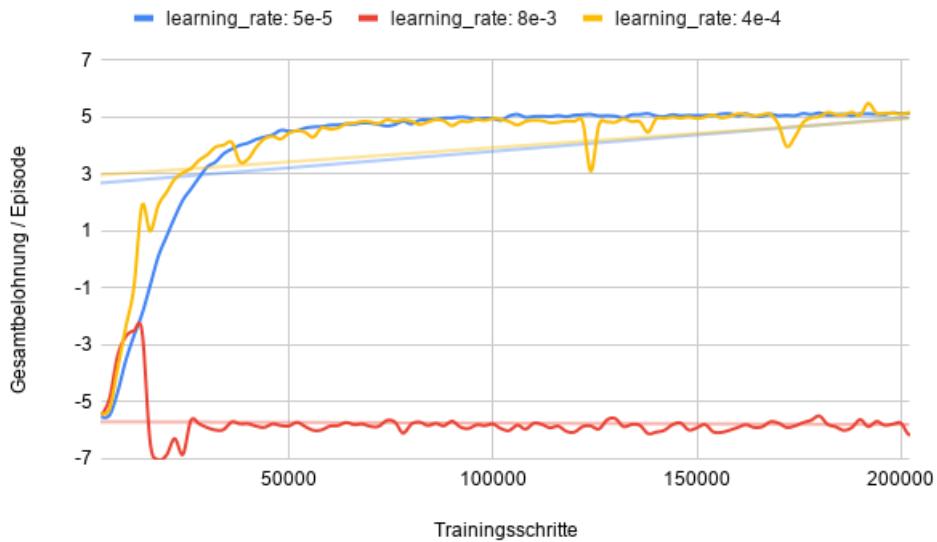


Abbildung A.1.: Belohnungsverlauf, Anpassung von *learning_rate*

A. Anhang A - Wahl der optimalen Hyperparameter



Abbildung A.2.: Verlauf des Value Loss, Anpassung von *learning_rate*

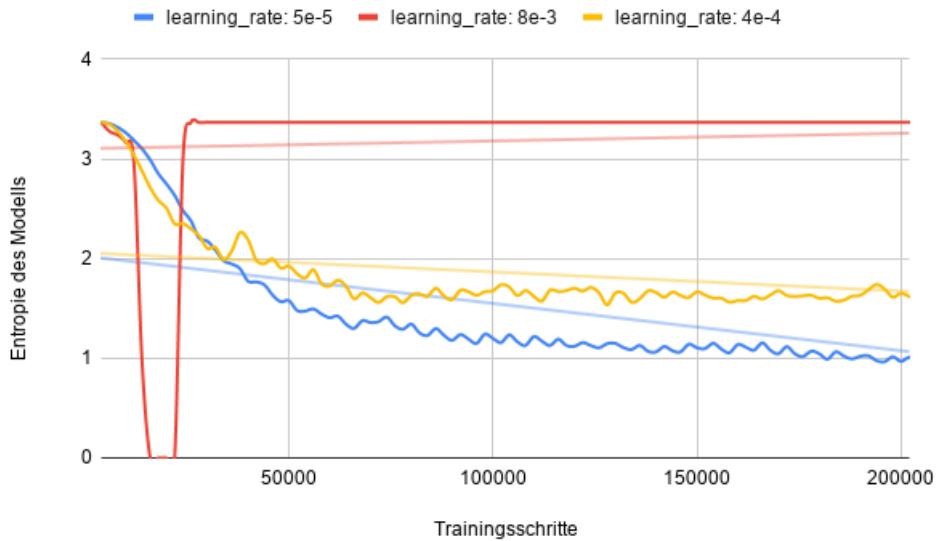


Abbildung A.3.: Entropieverlauf, Anpassung von *learning_rate*

Man erkennt deutlich, dass mit $8e - 3$ eine deutlich zu hohe Lernrate gewählt wurde. Zwar steigt die Belohnungsfunktion anfangs im Vergleich leicht schneller, doch schließlich fällt die Kurve rapide ab und bleibt anschließend auf einem sehr schlechten Niveau. Es ist zu vermuten, dass durch den hohen Wert beim Gradientenaufstieg Maxima übersprungen werden. Die Policy wird also jeweils zu stark geändert. Das führt zu einem völlig unbrauchbaren Ergebnis. Auch anhand der Entropie und dem Value Loss wird dies deutlich. Die Lernrate des Basistrainings mit dem Wert $4e - 4$ führt zu einem deutlich besseren Ergebnis. Im Vergleich zu der Kurve mit einer Lernrate von $5e - 5$ steigt sie schneller, weist jedoch zwischenzeitlich deutliche Einbrüche auf. Insgesamt wurde jedoch ein ähnlich gutes Ergebnis erzielt, da die Belohnungen nach etwa 100.000 Schritten fast identische Werte annehmen. Der Value Loss erreicht für die niedrigere Lernrate leicht schlechtere Ergebnisse, jedoch ist für die niedrige Lernrate auch zum Ende des Graphen hin eine Abnahme zu erkennen, während

A. Anhang A - Wahl der optimalen Hyperparameter

der Graph der mittelhohen Lernrate ein Plateau erreicht. Daher wird davon ausgegangen, dass sich der Value Loss mit einer höheren Anzahl von Trainingsschritten weiter verbessert. Die Entropie zeigt deutlich, dass mit der geringeren Lernrate weniger zufällige Entscheidungen getroffen werden. Aufgrund dieser Tatsache und der deutlich höheren Stabilität der Belohnungsfunktion wird die Lernrate für das finale Training auf den Wert $5e - 5$ festgelegt. Generell ist anzunehmen, dass mit einer geringeren Lernrate stets mindestens äquivalente Ergebnisse erzeugt werden können, sich aber insgesamt ein stabileres Training ergibt. Die Anzahl der Trainingsschritte kann dabei problemlos erhöht werden. Die höhere Steigung der Belohnungsfunktion für die mittelhohe Lernrate ist nicht relevant - das Endergebnis und die Stabilität des Trainings spielen eine höhere Rolle.

Anzahl der Schichten

Weiterhin wurde die Anzahl der Schichten im neuronalen Netz angepasst. Das Basistraining beinhaltet ein Netz mit drei Schichten. Zusätzlich wurden zwei, vier und fünf Schichten getestet.

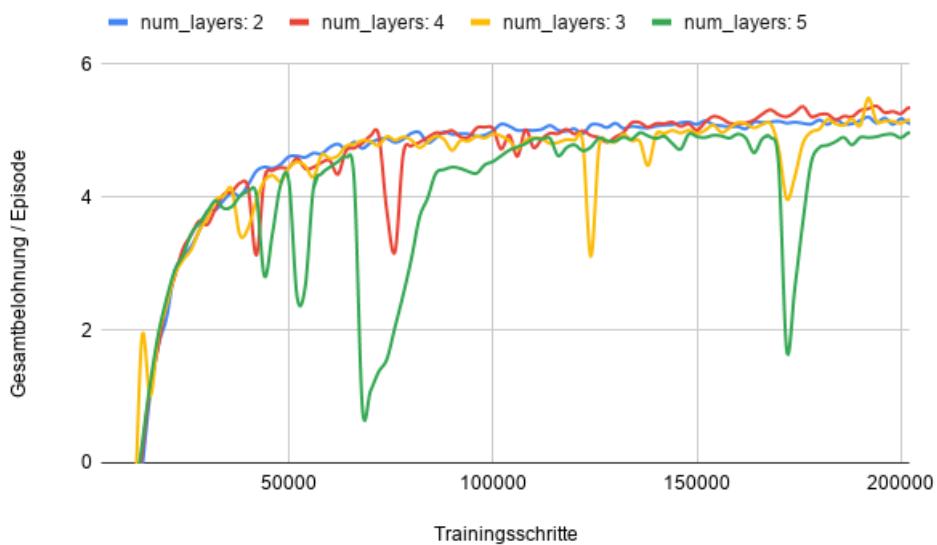


Abbildung A.4.: Belohnungsverlauf, Anpassung von *num_layers*

A. Anhang A - Wahl der optimalen Hyperparameter

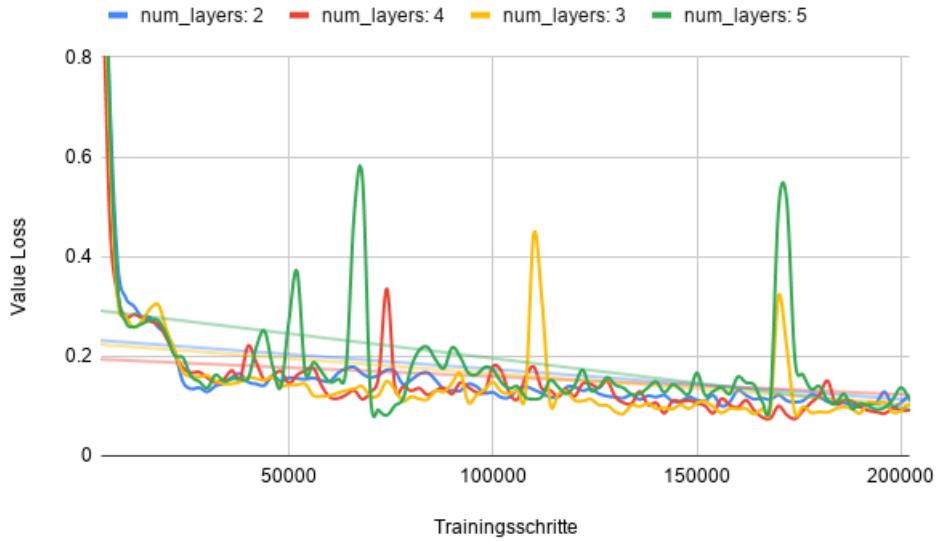


Abbildung A.5.: Verlauf des Value Loss, Anpassung von *num_layers*

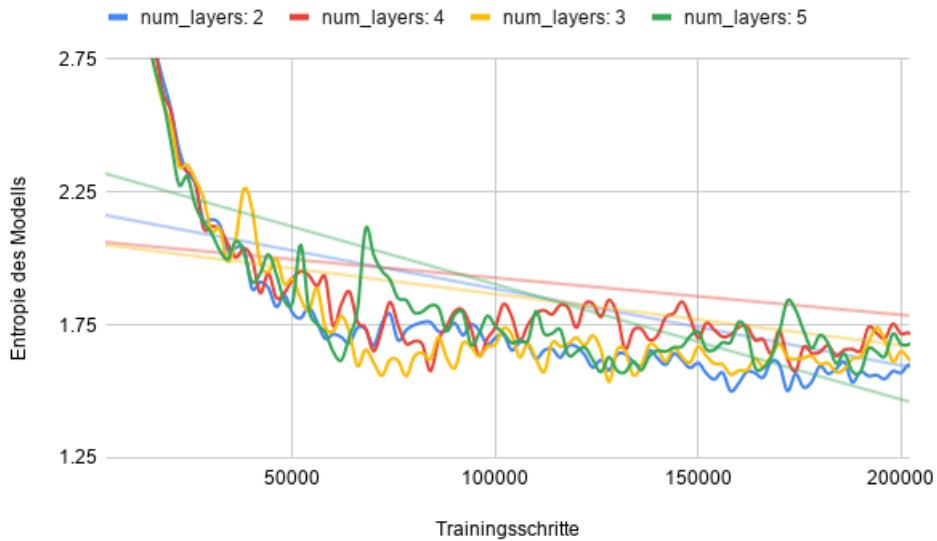


Abbildung A.6.: Entropieverlauf, Anpassung von *num_layers*

Anhand der Belohnungsfunktion kann man erkennen, dass die Verwendung von vier Schichten zu dem höchsten Reward führt. Insgesamt ist der allgemeine Verlauf der Funktionen ähnlich. Wenn allerdings fünf Schichten verwendet werden, entstehen starke Schwankungen und Einbrüche sowie eine insgesamt niedrigere Belohnung. Das lässt sich auch anhand des Entropieverlaufs und dem Value Loss erkennen. Die Nutzung von fünf Schichten ist daher nicht akzeptabel. Vier verwendete Schichten führen zu einem leicht erhöhten Entropieverlauf und einer Value Loss-Funktion, die sich mit den anderen Verläufen deckt. Da insgesamt aber eine höhere Belohnung erzielt wird, werden für das finale Training vier Schichten verwendet.

A. Anhang A - Wahl der optimalen Hyperparameter

Anzahl der Neuronen

Die Anzahl der Neuronen wurde auf eine Anzahl von 128, 256, 384 und 512 festgelegt. Aufgrund der Erkenntnis des letzten Abschnitts wurden hierfür vier Schichten genutzt.

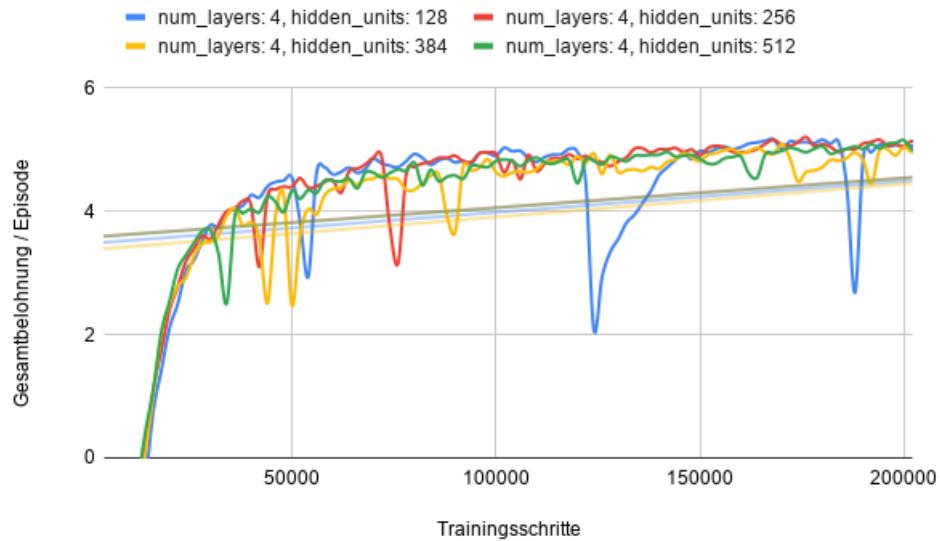


Abbildung A.7.: Belohnungsverlauf, Anpassung von *hidden_units*

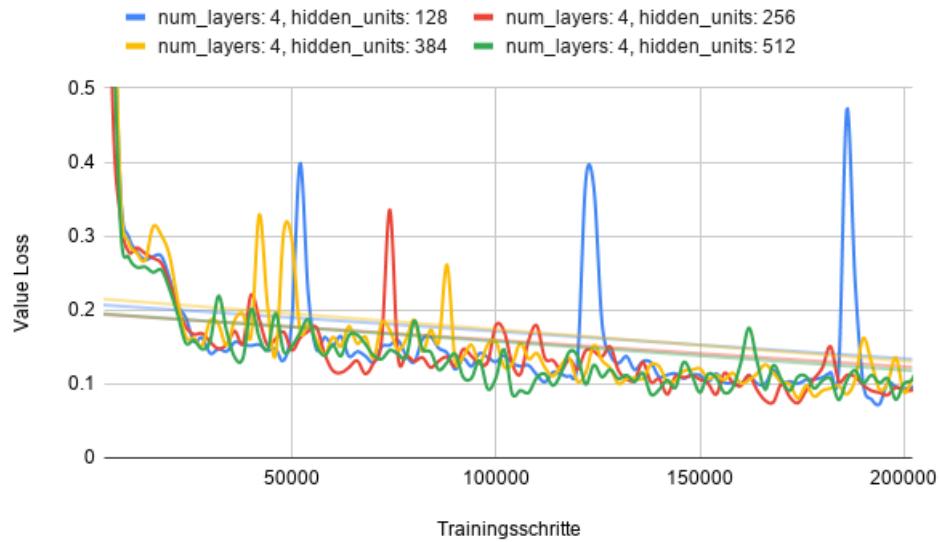


Abbildung A.8.: Verlauf des Value Loss, Anpassung von *hidden_units*

A. Anhang A - Wahl der optimalen Hyperparameter

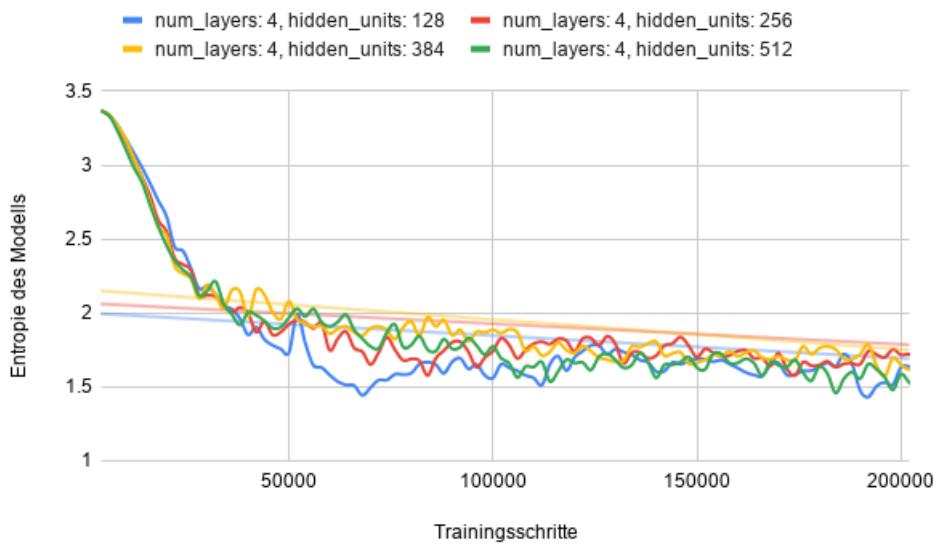


Abbildung A.9.: Entropieverlauf, Anpassung von *hidden_units*

An dem Verlauf der akkumulierten Belohnungen kann man erkennen, dass insgesamt sehr ähnliche Ergebnisse für die jeweils höchste Belohnung erzielt werden. Die Nutzung von 128 Neuronen pro Schicht führt allerdings zu häufigeren Einbrüchen, da durch die geringere Anzahl der Neuronen weniger kleinschrittige Änderungen des Modells möglich sind. 128 Neuronen sind daher nicht ausreichend. Für 384 und 512 Neuronen wird ein insgesamt leicht schlechteres und instabileres Ergebnis erreicht, als bei 256. Mit 384 Neuronen wurde also eine zu hohe Anzahl gewählt.

Die genannten Einbrüche mit 128 Neuronen sind ebenfalls in der Value Loss-Funktion zu betrachten. Die beiden Kurve für 256 und 384 Neuronen erreichen hingegen sehr ähnliche Ergebnisse. Ebenso ähneln sich deren Funktionen für die Entropie des Modells. Hier ist weiterhin zu erkennen, dass durch die Nutzung von weniger Neuronen eine schnellere Konvergenz erreicht wird, nach etwa 150.000 Trainingsschritten erreichen die Kurven jedoch ein ähnliches Level. Es besteht zusätzlich die Vermutung, dass bei einer Erhöhung der Anzahl der Neuronen mehr Trainingsschritte benötigt werden, um das Modell hinreichend anzupassen. Daher wird ein erweiterter Trainingslauf in der folgenden Abbildung A.10 betrachtet:

A. Anhang A - Wahl der optimalen Hyperparameter

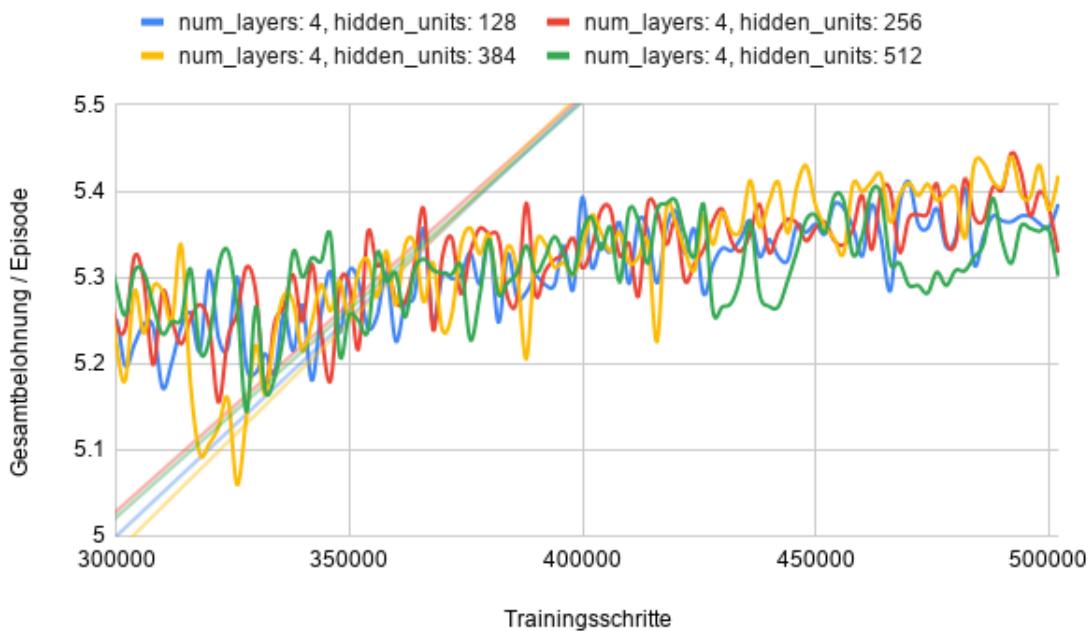


Abbildung A.10.: Erweiterter Belohnungsverlauf, Anpassung von *hidden_units*

In der Grafik ist zu erkennen, dass sich das Ergebnis für höhere Mengen von Neuronen mit steigender Anzahl von Trainingsschritten leicht verbessert. Aufgrund dieses Verlaufs und dem schlechteren Verlauf des Trainings mit 384 Neuronen werden insgesamt 256 Neuronen pro Schicht für das finale Training genutzt.

Rekurrentes Netz

Der Parameter *use_recurrent* wird auf *true* gesetzt, um ein rekurrentes neuronales Netz zu nutzen. Dabei wird eine *memory_size* von der Größe 128 genutzt. Die Ergebnisse sind in den folgenden Abbildungen zu sehen.

A. Anhang A - Wahl der optimalen Hyperparameter

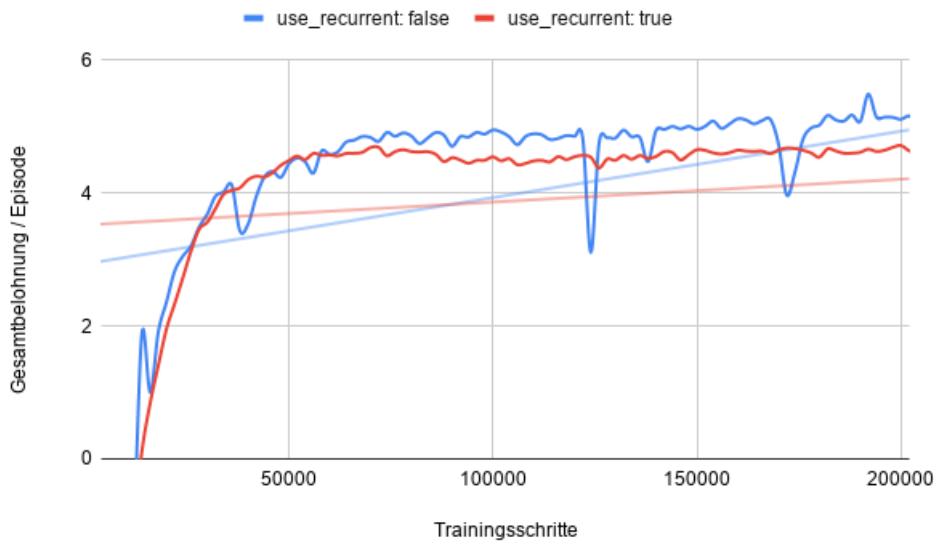


Abbildung A.11.: Belohnungsverlauf, Anpassung von *use_recurrent*

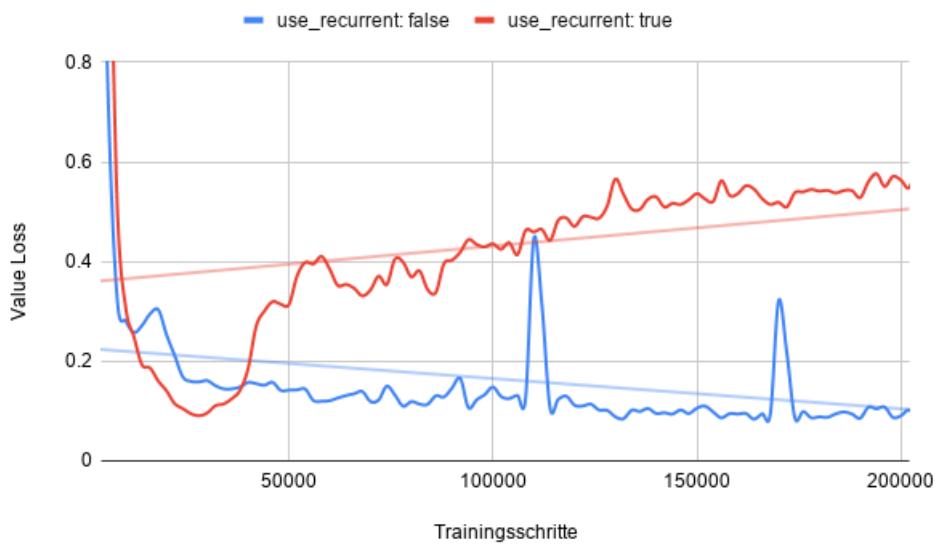


Abbildung A.12.: Verlauf des Value Loss, Anpassung von *use_recurrent*

A. Anhang A - Wahl der optimalen Hyperparameter

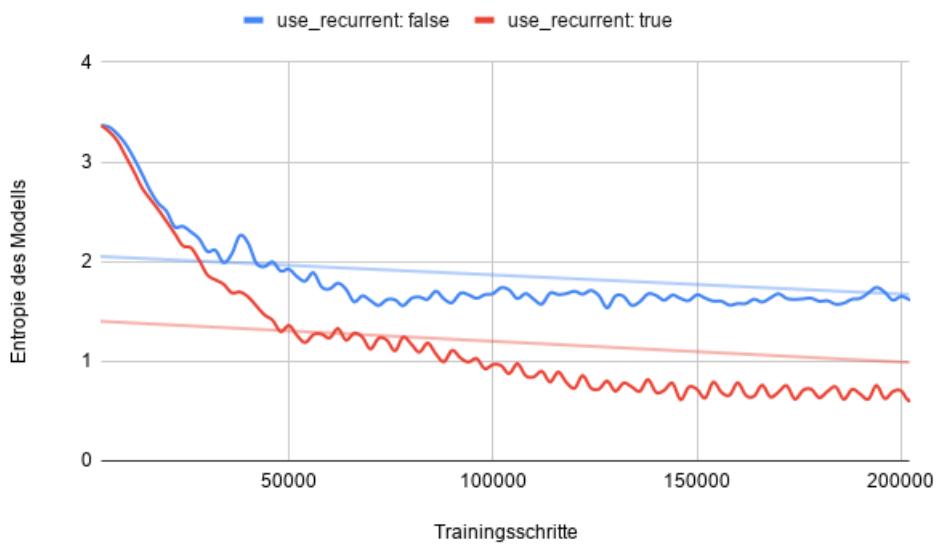


Abbildung A.13.: Entropieverlauf, Anpassung von *use_recurrent*

Insgesamt hat sich der Verlauf der Belohnungsfunktion stark verschlechtert. Die Kurve des Value Loss steigt nach einem initialen Abstieg stark an und erzielt schlussendlich ein extrem schlechtes Ergebnis. Zwar wird eine niedrigerer Wert für die Entropie erreicht, doch insgesamt ist das Ergebnis nicht akzeptabel. Daher wird von der Nutzung von rekurrenten Netzen abgesehen. Tatsächlich wurde durch das Training ein so viel schlechteres Ergebnis erzielt, dass von einer weiteren Anpassung der Parameter bezüglich rekurrenter Netze abgesehen wird.

Gamma

Um den Einfluss zukünftiger Belohnungen auf die Entscheidungen des Agenten zu beeinflussen, wird der Parameter Gamma (γ) angepasst (siehe Abschnitt 3.2).

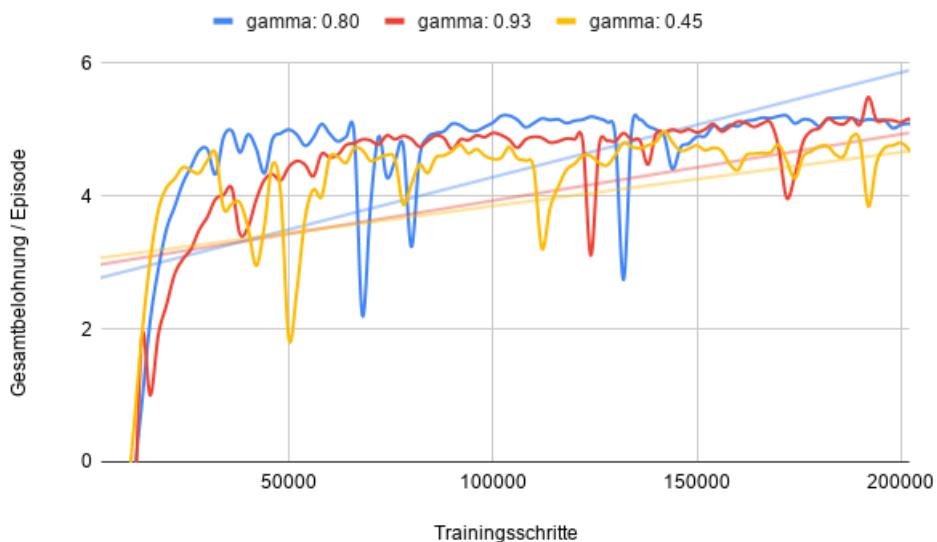


Abbildung A.14.: Belohnungsverlauf, Anpassung von *gamma*

A. Anhang A - Wahl der optimalen Hyperparameter

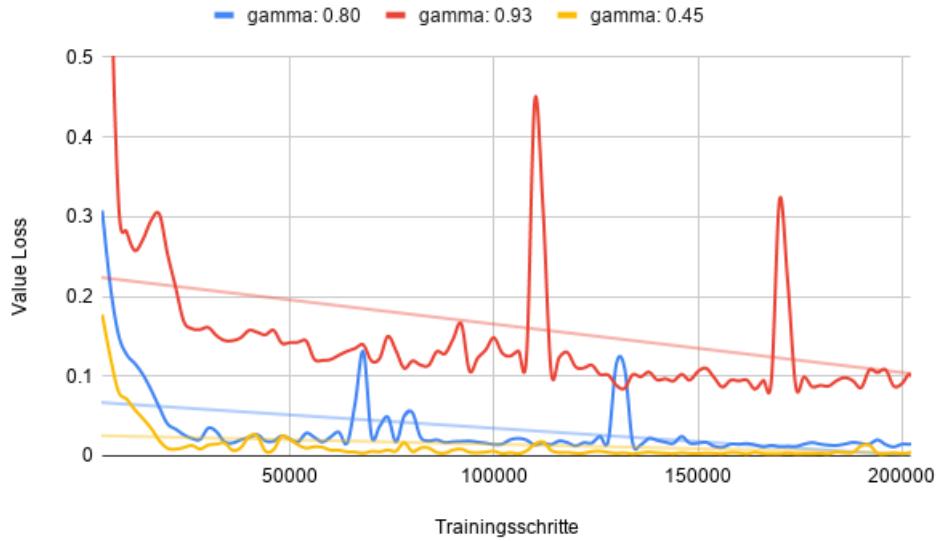


Abbildung A.15.: Entropieverlauf, Anpassung von γ

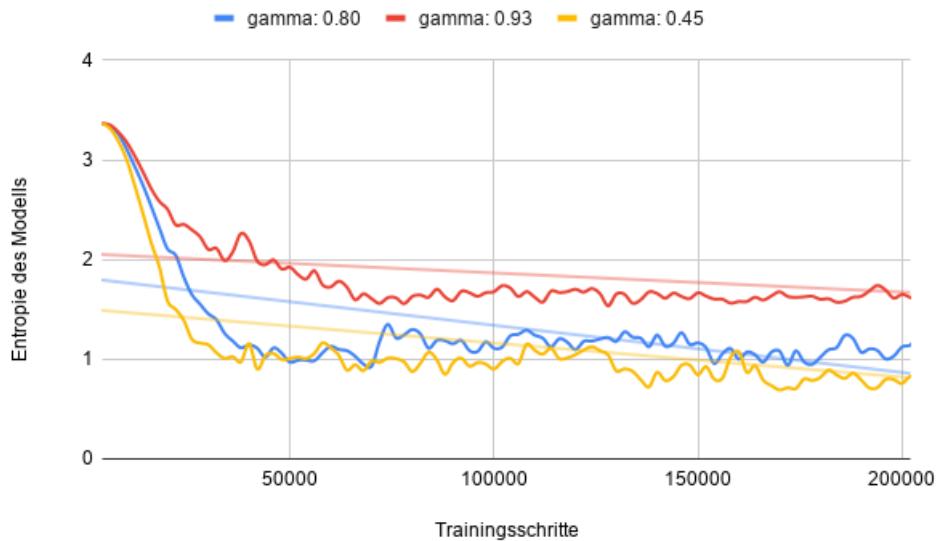


Abbildung A.16.: Verlauf des Value Loss, Anpassung von γ

An dem Entropieverlauf ist klar zu erkennen, dass die Wahl eines niedrigen Werts für γ zu einem besseren Ergebnis führt. Ebenso verhält es sich mit der Entropie. Der Graph, der den niedrigsten Wert mit $\gamma = 0.45$ repräsentiert, zeigt das beste Ergebnis. Gleichzeitig erzielt der Trainingsvorgang mit diesem Wert deutlich schlechtere Werte für den Belohnungsverlauf. Das beste Ergebnis wird hier von dem Graphen erreicht, der durch den Wert $\gamma = 0.80$ entstanden ist. γ wird daher auf diesen Wert festgelegt.

Größe der Batches

Verschiedene Parameter innerhalb des empfohlenen Intervalls für den Parameter `batch_size` wurden getestet. Die Ergebnisse sind im Folgenden zu erkennen:

A. Anhang A - Wahl der optimalen Hyperparameter

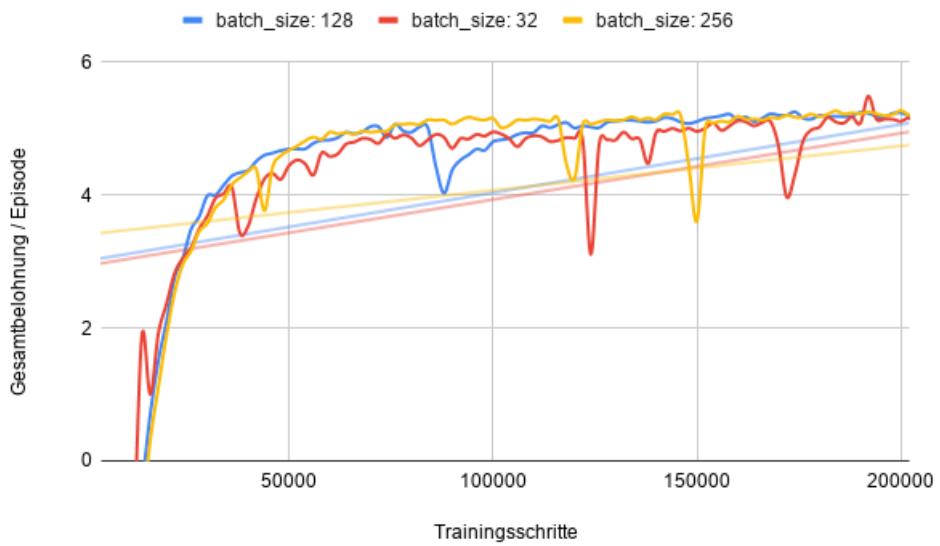


Abbildung A.17.: Belohnungsverlauf, Anpassung von *batch_size*

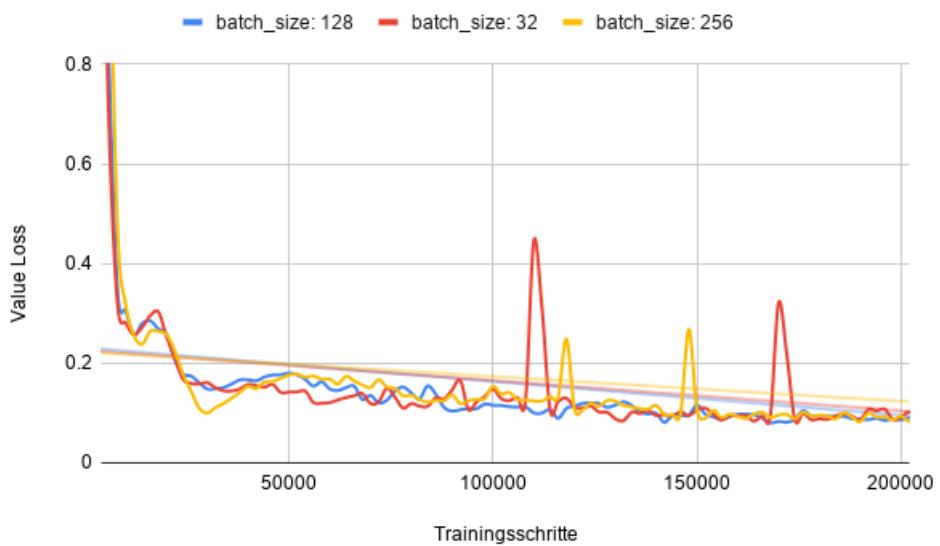


Abbildung A.18.: Verlauf des Value Loss, Anpassung von *batch_size*

A. Anhang A - Wahl der optimalen Hyperparameter

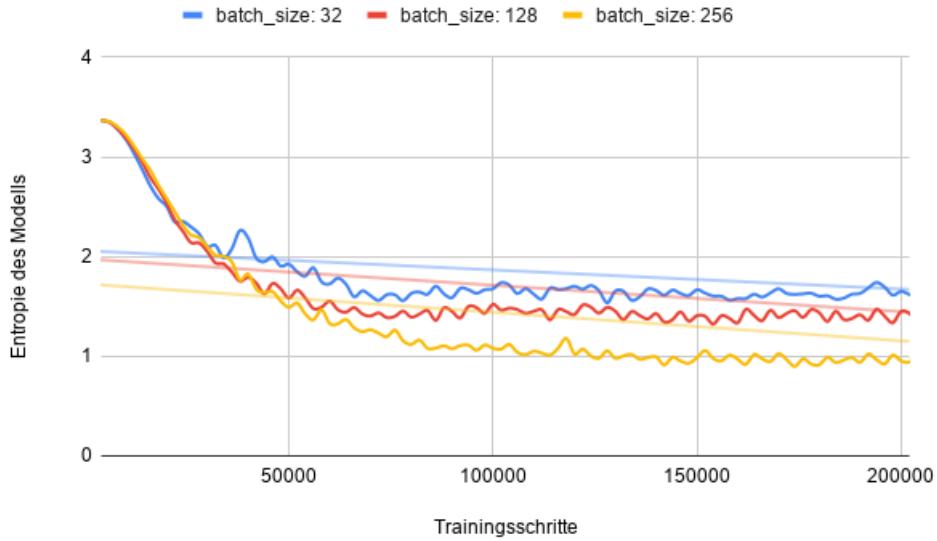


Abbildung A.19.: Entropieverlauf, Anpassung von *batch_size*

Es ist deutlich zu erkennen, dass sich für die höchste *batch_size* die am wenigsten zufälligen Entscheidungen ergeben. Die Verläufe der Value Loss- und Belohnungsfunktionen sind weitestgehend identisch. Daher wird eine Batchgröße von 256 gewählt.

Beta

Weiterhin wurden verschiedene Werte für den Parameter Beta (β) ausprobiert. Das Resultat ist in den folgenden Abbildungen erkennbar:

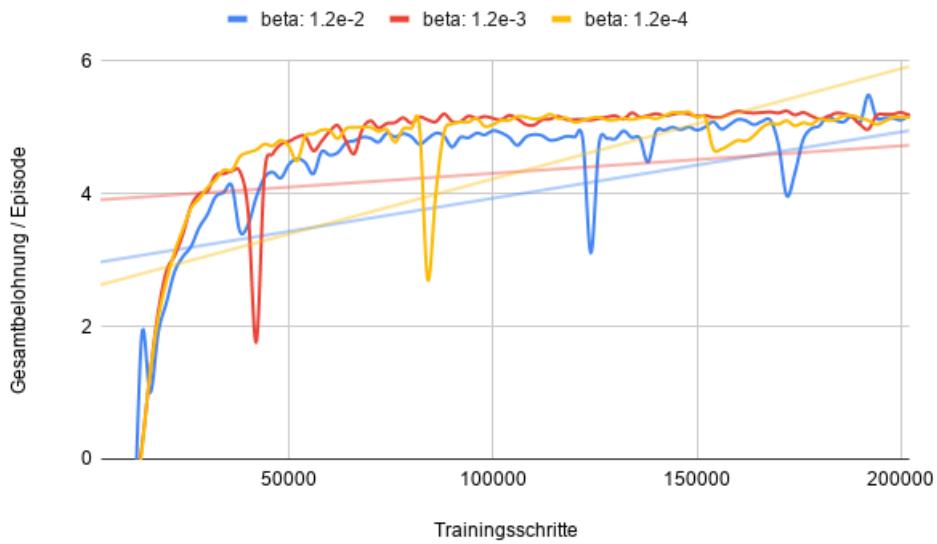


Abbildung A.20.: Belohnungsverlauf, Anpassung von *beta*

A. Anhang A - Wahl der optimalen Hyperparameter

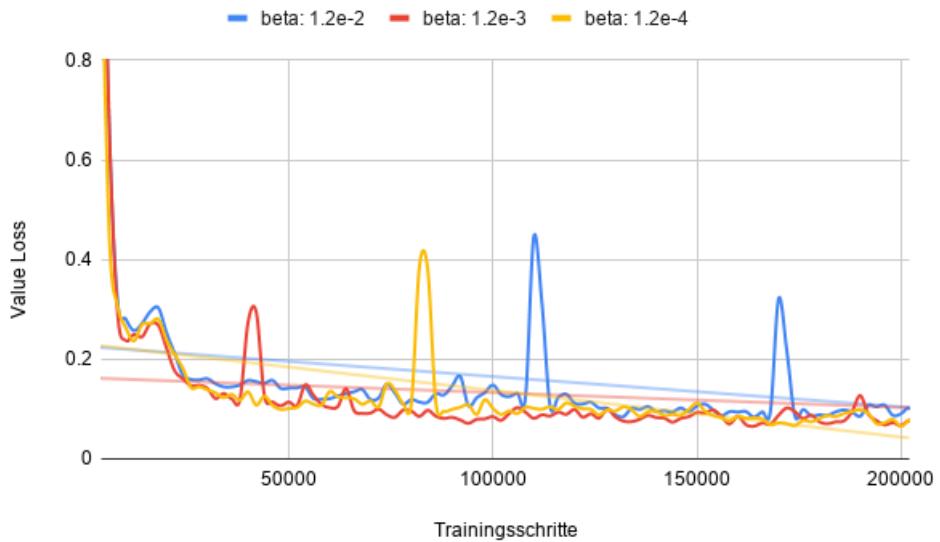


Abbildung A.21.: Verlauf des Value Loss, Anpassung von β

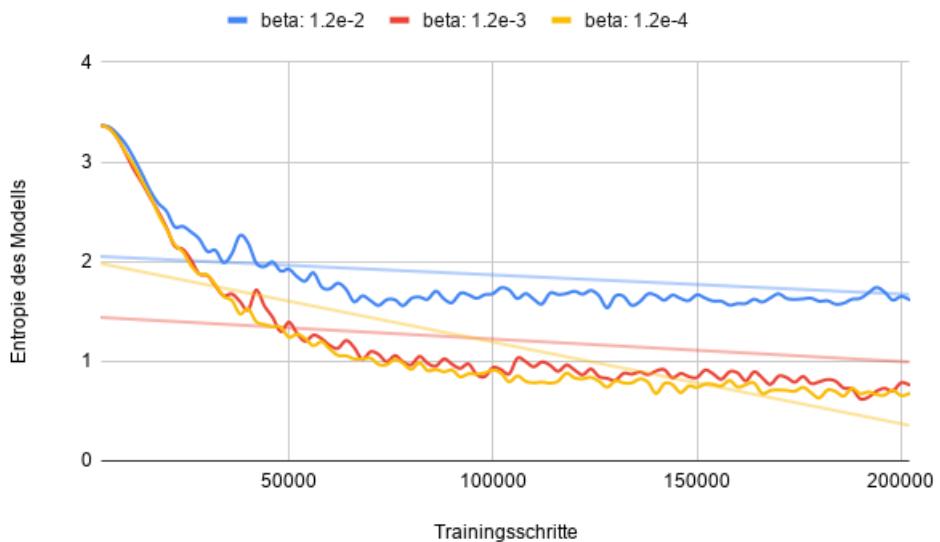


Abbildung A.22.: Entropieverlauf, Anpassung von β

Insgesamt konnten mit niedrigeren Werten für β schneller ein besseres Ergebnis erzielt werden. Das gilt für die Belohnungsfunktion, den Value Loss und die Entropie des Modells. Da für den Wert $\beta = 1.2e - 3$ jedoch höhere Belohnungen als für den Wert $\beta = 1.2e - 4$ erreicht wurden, wird Beta auf den Wert $\beta = 1.2e - 3$ festgelegt.

Lambda

Der Parameter Lambda (λ) wurde ebenfalls angepasst, wie in den folgenden Grafiken zu erkennen:

A. Anhang A - Wahl der optimalen Hyperparameter

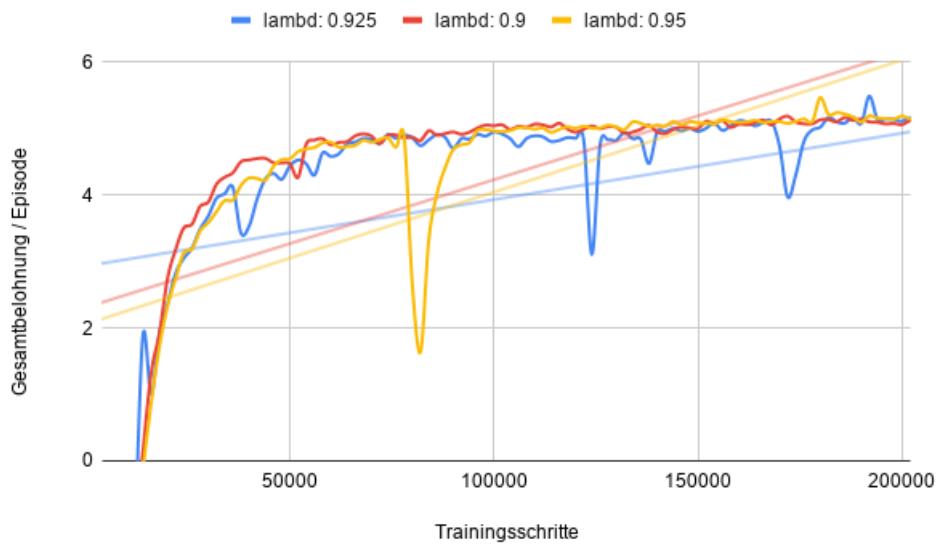


Abbildung A.23.: Belohnungsverlauf, Anpassung von λ

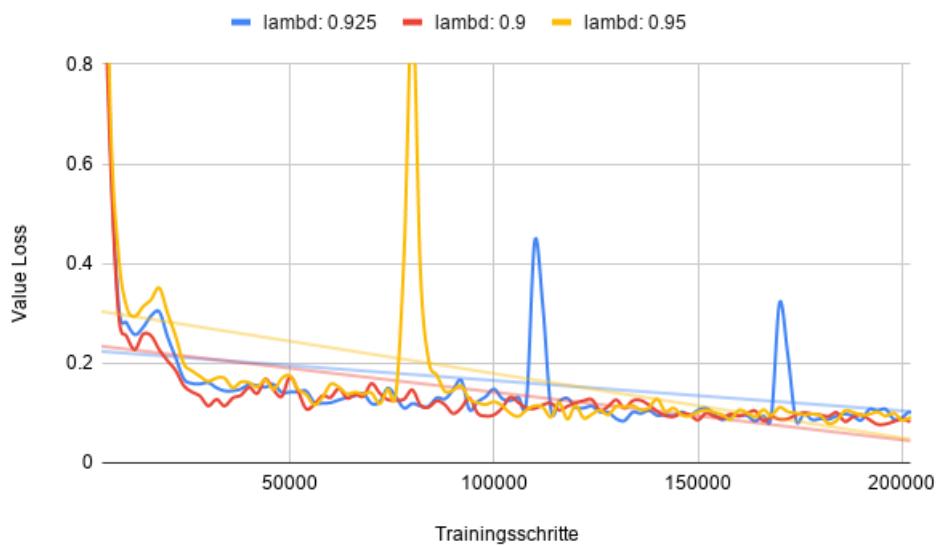


Abbildung A.24.: Verlauf des Value Loss, Anpassung von λ

A. Anhang A - Wahl der optimalen Hyperparameter

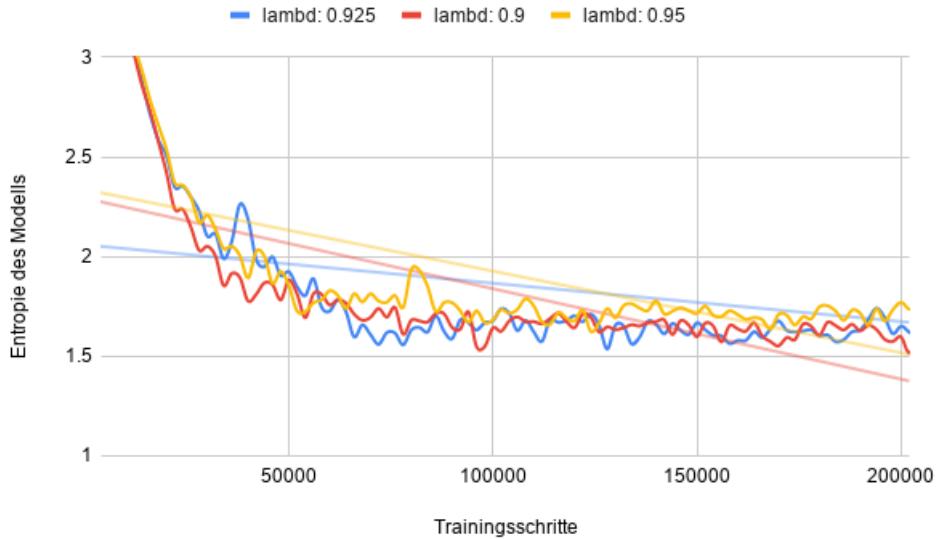


Abbildung A.25.: Entropieverlauf, Anpassung von λ

Die Belohnungsfunktion stabilisiert sich leicht durch die Verwendung einer niedrigeren Wertes für λ . Auch für die Value Loss- und Entropiefunktion ergibt sich ein leicht verbessertes Ergebnis. Der niedrigste Wert $\lambda = 0.9$ führt also insgesamt zum besten Ergebnis.

Anzahl der Epochen

Ein weiterer Trainingsdurchlauf wird mit einer reduzierten Anzahl num_epoch von vier auf drei Epochen durchgeführt. Dadurch wird derselbe Buffer weniger oft für den Gradientenabstieg verwendet. Geringere Werte sollten in einem stabileren, aber langsameren Training resultieren. ML-Agents empfiehlt einen Minimalwert von $num_epoch = 3$.

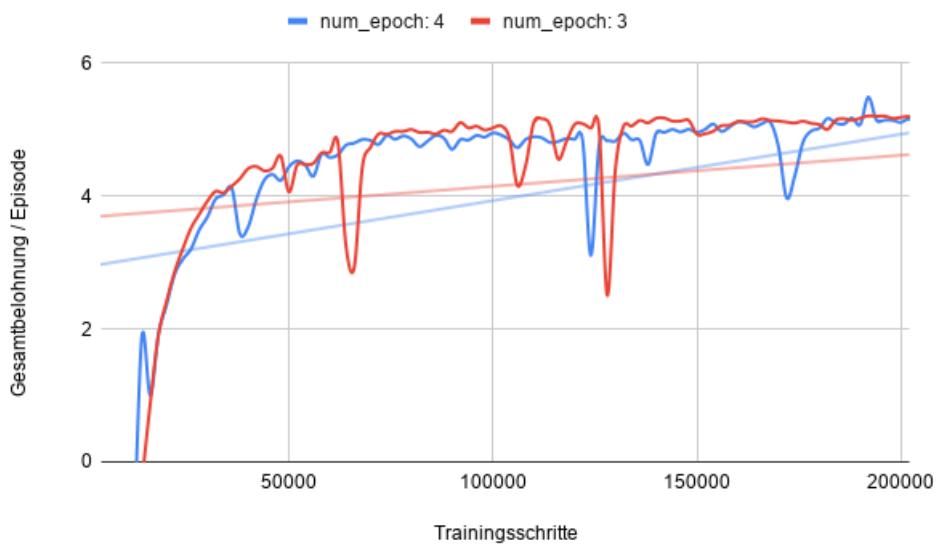


Abbildung A.26.: Belohnungsverlauf, Anpassung von num_epochs

A. Anhang A - Wahl der optimalen Hyperparameter

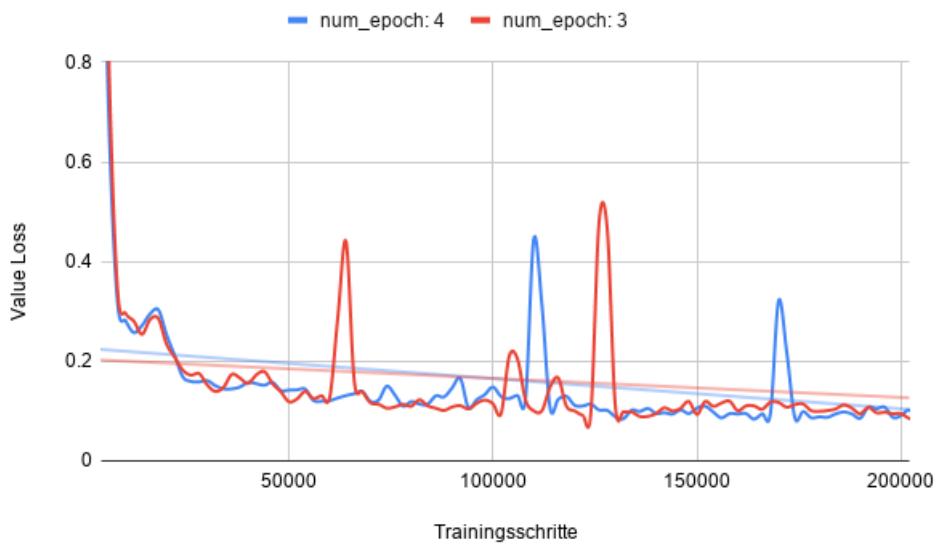


Abbildung A.27.: Verlauf des Value Loss, Anpassung von *num_epochs*

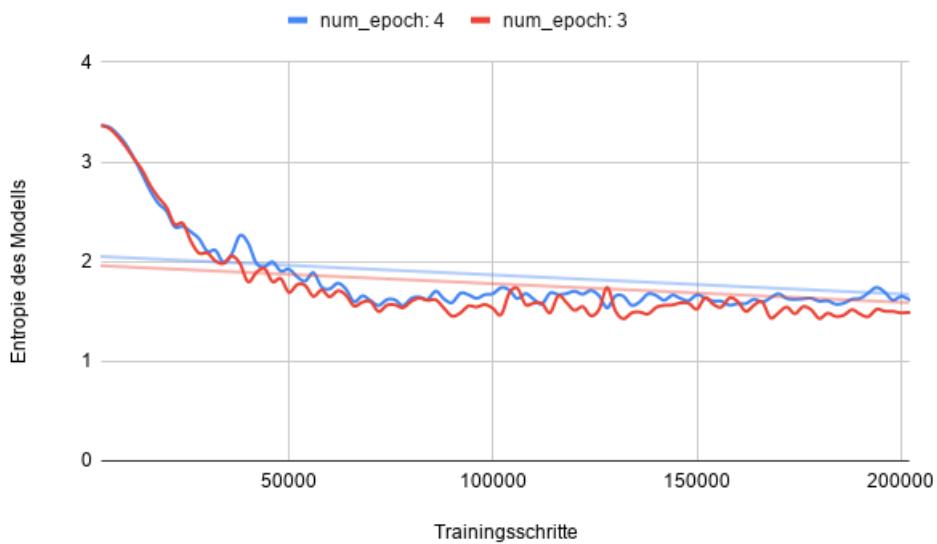


Abbildung A.28.: Entropieverlauf, Anpassung von *num_epochs*

Da die Belohnungsfunktion schneller ansteigt und insgesamt einen hohen Level erreicht und wegen der leicht verbesserten Entropie, wird der Parameter *num_epoch* auf 3 festgelegt.

Epsilon

Der Hyperparameter *epsilon* wird angepasst, um zu bestimmen, wie schnell sich eine Policy im Verlauf des Trainings verändern darf. Die Ergebnisse sind in den folgenden Abbildungen zu erkennen:

A. Anhang A - Wahl der optimalen Hyperparameter

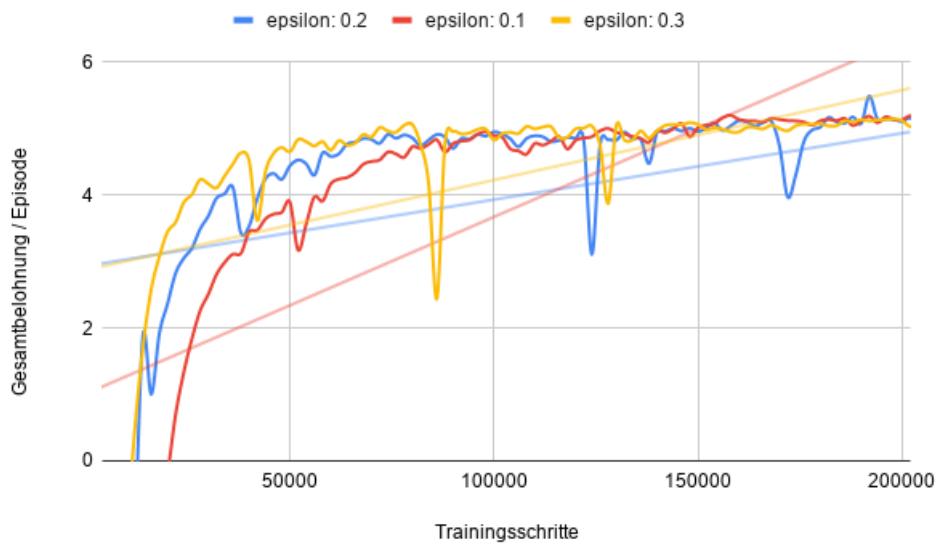


Abbildung A.29.: Belohnungsverlauf, Anpassung von *epsilon*

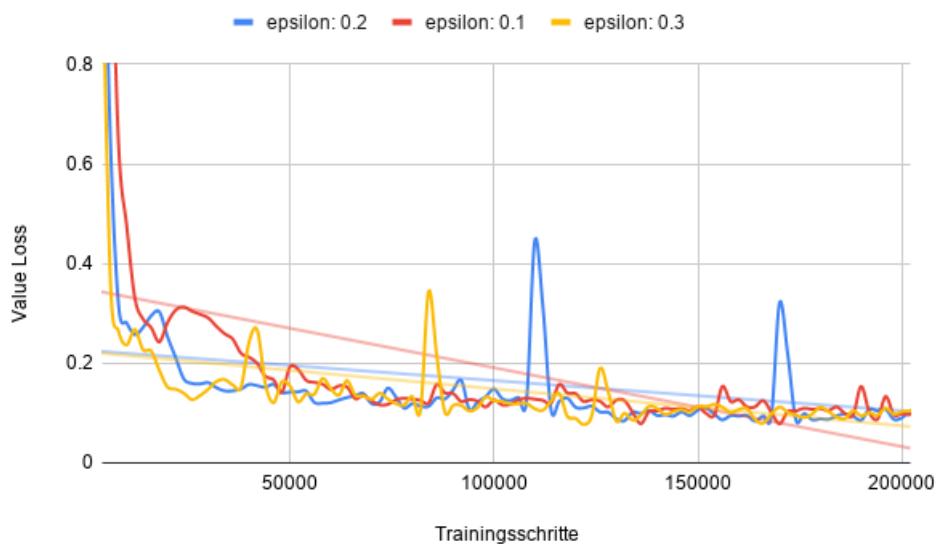


Abbildung A.30.: Verlauf des Value Loss, Anpassung von *epsilon*

A. Anhang A - Wahl der optimalen Hyperparameter

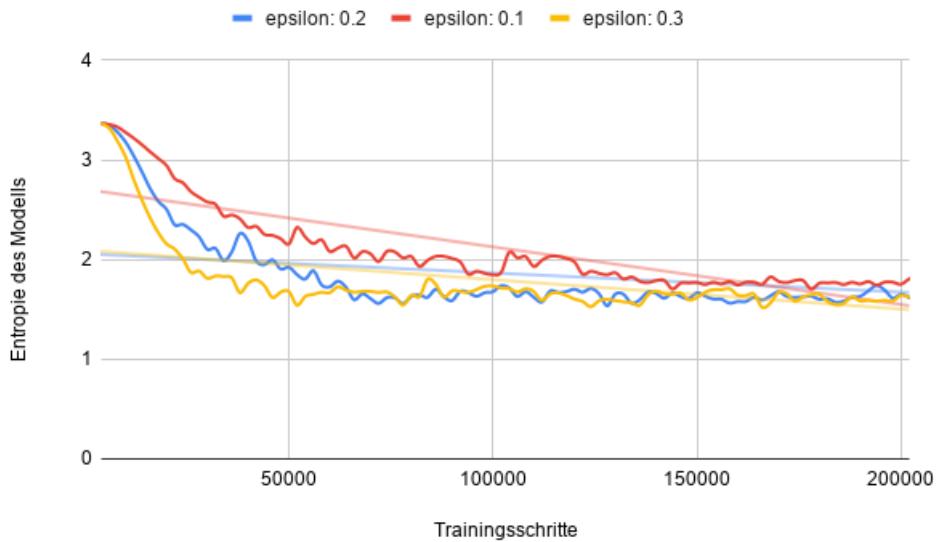


Abbildung A.31.: Entropieverlauf, Anpassung von *epsilon*

Wenn der Wert 0.3 für den Parameter *epsilon* gewählt wird, erfolgen daraus insgesamt leicht verbesserte Ergebnisse für alle drei Metriken. Weiterhin wird die Geschwindigkeit des Lernvorgangs erhöht. Dieser Wert wird daher für das finale Training verwendet.

Größe des Buffers

Weiterhin wird die Größe des Buffers angepasst, um die Auswirkungen dieser Änderungen im Training zu betrachten.

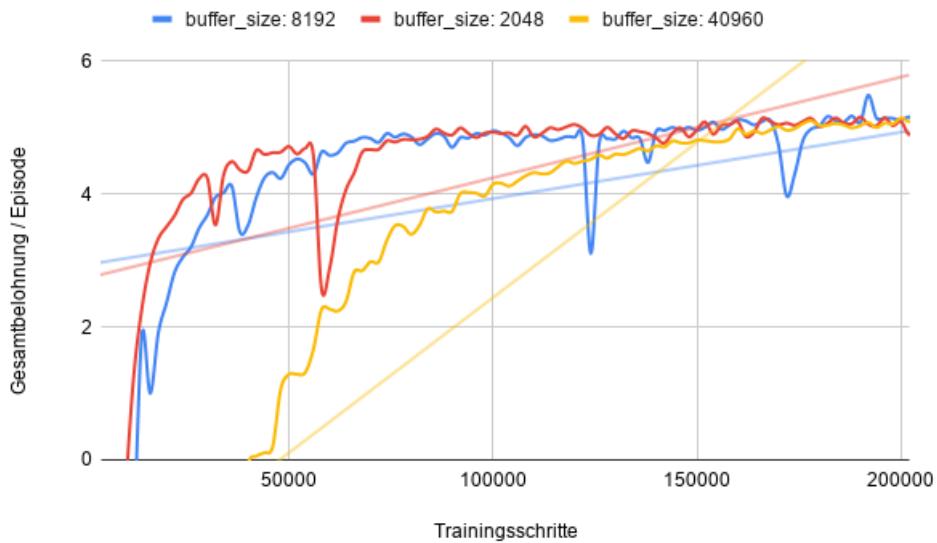


Abbildung A.32.: Belohnungsverlauf, Anpassung von *buffer_size*

A. Anhang A - Wahl der optimalen Hyperparameter

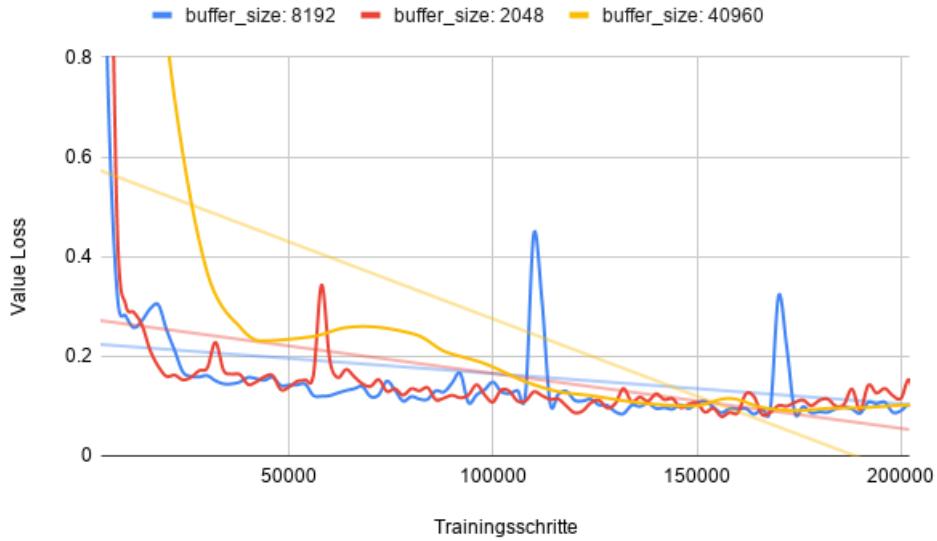


Abbildung A.33.: Verlauf des Value Loss, Anpassung von *buffer_size*

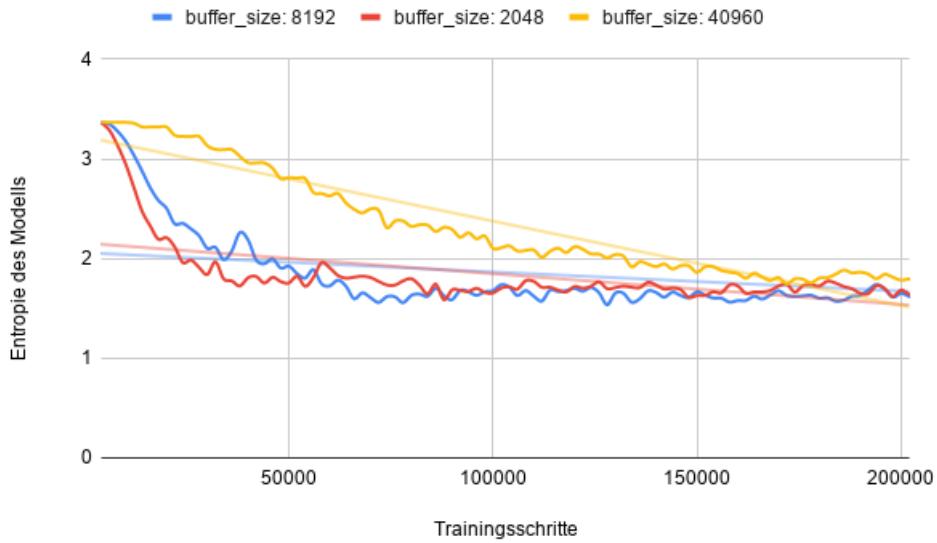


Abbildung A.34.: Entropieverlauf, Anpassung von *buffer_size*

Die höchste Buffergröße führt zu einem deutlich verlangsamten Trainingsvorgang mit insgesamt schlechteren Ergebnissen für Belohnungen, Value Loss und die Entropie. Für niedrige Werte erfolgt ein leicht schnellerer Lernvorgang mit ähnlichen Endergebnissen für die Belohnungswerte. Da sich die Entropie und der Value Loss jedoch nicht verändern, wird weiterhin der Wert 8192 für die Buffergröße gewählt.

Literaturverzeichnis

- [AB09] Mojang AB. Minecraft, 2009.
- [AFP19] Lilyana R. Andrea E. Yulsilviana S. Mallala A. F. Pukeng, R. R. Fauzi. An intelligent agent of finite state machine in educational game flora the explorer, 2019.
- [Ait13] Alastair Aitchison. At-a-glance functions for modelling utility-based game ai, 2013.
- [AJ18] Esh Vckay Yuan Gao Hunter Henry Marwan Mattar Danny Lange Arthur Juliani, Vincent-Pierre Berges. Unity: A general platform for intelligent agents, 2018.
- [Alp10] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, Massachusetts Institute of Technology, 2010.
- [AN19] Esteban Clua Ashey Noblega, Aline Paes. Towards adaptive deep reinforcement game balancing, 2019.
- [Ash18] Mohammad Ashraf. Reinforcement learning demystified: Markov decision processes, 2018.
- [BB19a] Todor Markov Yi Wu Glenn Powell Bob McGrew Igor Mordatch Bowen Baker, Ingmar Kanitscheider. Hide and seek - emergent tool use from multi-agent interaction, 2019.
- [BB19b] Todor Markov Yi Wu Glenn Powell Bob McGrew Igor Mordatch Bowen Baker, Ingmar Kanitscheider. Hide and seek - emergent tool use from multi-agent interaction, openai blog, 2019.
- [Bel57] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [Box20] BoxOfficeMojo.com. Avengers: Endgame box office, 2020.
- [Buc04] Mat Buckland. Programming game al by example, 2004.
- [CB19] Brooke Chan Vicki Cheung Przemyslaw Dębiak Christy Dennison David Farhi Quirin Fischer Shariq Hashme Chris Hesse Rafal Józefowicz Scott Gray Catherine Olsson Jakub Pachocki Michael Petrov Henrique Pondé de Oliveira Pinto Jonathan Raiman Tim Salimans Jeremy Schlatter Jonas Schneider Szymon Sidor Ilya Sutskever Jie Tang Filip Wolski Susan Zhang Christopher Berner, Greg Brockman. Dota 2 with large scale deep reinforcement learning, 2019.
- [CJ12] Ben Sunshine-Hill Chris Jurney, Michael Robbins. Off the beaten path: Non-traditional uses of ai, 2012.
- [Cor13] Valve Corporation. Dota 2, 2013.
- [Cor17] NVidia Corporation. Generating expressive 3d facial animations from audio, 2017.
- [CR16] John Co-Reyes. Deep reinforcement learning with openai gym, 2016.
- [CS03] Wolfram Burgard Cyrill Stachniss. Autonomous mobile systems: The markov decision problem - value iteration and policy iteration, 2003.
- [DA18] Danny Hernandez Dario Amodei. Ai and compute, 2018.
- [Dam] Patrick Dammann. Einführung in das reinforcement learning.
- [Dar17] Unity-Nutzer Darth_Artisan. Free trees (unity asset pack), 2017.

- [Dee16] DeepMind Technologies, Alphabet Inc. *Continuous Control With Deep Reinforcement Learning*. International Conference on Learning Representations, 2016.
- [DGCFM15] Mobilegeeks.de Dan Greenwald (Creative Director Forza Motorsport). Forza motorsport 6 & the evolution of machine learning in drivatars, 2015.
- [DH17a] Jun Saito Daniel Holden, Taku Komura. Phase-functioned neural networks for character control, github repository, 2017.
- [DH17b] Jun Saito Daniel Holden, Taku Komura. Phase-functioned neural networks for character control, paper, 2017.
- [DM18] Roger Eastman Dave Mount. Artificial intelligence for games: Decision making, cmsc 425: Lecture 21, 2018.
- [DMB04] Glenn Seeman David M. Bourg. *AI for Game Developers*. O'Reilly, 2004.
- [Don18] Joe Donnelly. Gta 5 estimated to be the most profitable entertainment product of all time, 2018.
- [DS17] Julian Schrittwieser Ioannis Antonoglou Matthew Lai Arthur Guez Marc Lanctot Laurent Sifre Dharshan Kumaran Thore Graepel Timothy Lillicrap Karen Simonyan Demis Hassabis David Silver, Thomas Hubert. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.
- [DT10] Alphabet Inc DeepMind Technologies. Deepmind, 2010.
- [Eni13] Square Enix. Reinforcement learning based character locomotion in hitman: Absolution, 2013.
- [Ent13] Uber Entertainment. Planetary annihilation, 2013.
- [Fal13] John Fallon. Believable behaviour of background characters in open world games. Master's thesis, University of Dublin, Trinity College, 8 2013.
- [FGG18] Michael G. Madden Frank G. Glavin. Skilled experience catalogue: A skill-balancing mechanism for non-player characters using reinforcement learning, 2018.
- [GA06] Alex Sandro Gomes Vincent Corruble Gustavo Andrade, Geber Ramalho1. Dynamic game balancing: an evaluation of user satisfaction, 2006.
- [Gam10] Gas Powered Games. Supreme commander 2, 2010.
- [HMG07] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueskillTM: A bayesian skill rating system. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 569–576. MIT Press, 2007.
- [HN89] Robert Hecht-Nielsen. Theory of the backpropagation neural network, 1989.
- [Hui18] Jonathan Hui. Rl - proximal policy optimization (ppo) explained, 2018.
- [Int14] IO Interactive. Hitman: Absolution, 2014.
- [iS92] id Software. Wolfenstein 3d, 1992.
- [Isl05] Damian Isla. Gdc 2005 proceeding: Handling complexity in the halo 2 ai, 2005.
- [Jan20] Tobias Jansing. Anwendung von reinforcement learning zurentwicklung authentischer spielwelten mithilfe bedürfnisbasierter agenten, github repository der thesis, 2020.
- [Joy11] James M. Joyce. *Kullback-Leibler Divergence*, pages 720–722. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

Literaturverzeichnis

- [JR17] Martin Emborg John Robertson. Echo interview q&a with ultra ultra's martin emborg, 2017.
- [JS15] Philipp Moritz Michael I. Jordan Pieter Abbeel John Schulman, Sergey Levine. Trust region policy optimization, 2015.
- [JS17] Filip Wolski Prafulla Dhariwal Alec Radford OpenAI John Schulman, Oleg Klimov. Proximal policy optimization, 2017.
- [Jul17] Arthur Juliani. Unity ai – reinforcement learning with q-learning, 2017.
- [KD09] Dave Mark Kevin Dill. Improving ai decision modeling through utility theory, 2009.
- [Klo17] Peter Klooster. Goap - a multi-threaded goap (goal oriented action planning) system for unity3d, github repository, 2017.
- [KS51] Leibler RA Kullback S. *On information and sufficiency*. JSTOR, 1951.
- [Lea17] Jonathan Leack. World of warcraft leads industry with nearly \$10 billion in revenue, 2017.
- [Lem00] Joerg Lemm. Markov-eigenschaft, 2000.
- [LESM19] Andy Luc Laura E Shummon Maass. Artificial intelligence in video games - an overview of how video game a.i. has developed over time and current uses in games today, 2019.
- [MA17a] Google LLC Moustafa Alzantot. Deep reinforcement learning demystified (episode 0), 2017.
- [MA17b] Google LLC Moustafa Alzantot. Deep reinforcement learning demystified (episode 2), 2017.
- [MAP11] Lead Developer *Minecraft* Markus Alexej Persson. The world of notch: Terrain generation, part 1, 2011.
- [Mas81] A. H Maslow. Motivation und persönlichkeit. 14. auflage, 1981.
- [Mat15] Tambet Matiisen. Demystifying deep reinforcement learning, 2015.
- [MC19] Colin McGourty Magnus Carlsen. Carlsen: My opponent is an idiot till proven otherwise!, 2019.
- [MG17] Arnav Jhala Manuel Guimaraes, Pedro Santos. Cif-ck: An architecture for social npcs in commercial games, 2017.
- [MR14] Forumname Sorian Mike Robbins. Upcoming ai neural net change, 2014.
- [Nie14] Daniela Niemeyer. Goal-oriented action planning für die simulationsplattform mars, 2014.
- [NK98] Robert Kozma Nikola Kasabov. Introduction: Hybrid intelligent adaptive systems, 1998.
- [Ope16] OpenAI. Openai gym, 2016.
- [Ope18a] OpenAI. Openai five, 2018.
- [Ope18b] OpenAI. Proximal policy optimization, 2018.
- [Orf14] Darren Orf. The ai that powers the sims 4 is almost too smart, 2014.
- [Ork] Jeff Orkin. Goal-oriented action planning (goap).

- [Owe14] Brent Owens. Goal oriented action planning for a smarter ai, 2014.
- [PLC14] Informa PLC. Game developers conference, 2014.
- [Pog19] Marin Vlastelica Pogancic. The almighty policy gradient in reinforcement learning, 2019.
- [Pol17] Unity-Nutzer Polygrade. Medieval windmill (unity asset pack), 2017.
- [Pon18] Vitchyr Pong. Tdm: From model-free to model-based deep reinforcement learning, 2018.
- [PVG⁺11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [PVG⁺19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Tuning the hyper-parameters of an estimator, api dokumentation, 2019.
- [QZ18] Quanjun Yin Yabing Zha Qi Zhang, Jian Yao. Learning behavior trees for autonomous agents with hybrid constraints evolution, 2018.
- [RED15] CD Projekt RED. The witcher 3: Wild hunt, 2015.
- [Ric] Charles Rich. Basic game ai - technical game development ii.
- [RSS18] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning - An Introduction, Second Edition*. The MIT Press, 2018.
- [Sch18] Kevin Schliecker. Verhaltenssteuerung bedürfnisorientierter npcs in rollenspielen mithilfe neuronaler netze, 2018.
- [SH17] Sam Snider-Held. Neural networks and the future of 3d procedural content generation, 2017.
- [Shi17] Matt Shipman. New tool increases adaptability, autonomy of ‘skyrim’ nonplayer characters, 2017.
- [Sim14] Chris Simpson. Behavior trees for ai: How they work, 2014.
- [Sim18] Thomas Simonini. An intro to advantage actor critic methods: let’s play sonic the hedgehog!, 2018.
- [Sof11] Bethesda Softworks. The elder scrolls v: Skyrim, 2011.
- [Spi14] Ciela Spike. Thread ninja - multithread coroutine, 2014.
- [SR03] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach, Second Edition*. Addison Wesley, Prentice Hall, 2003.
- [Sta14] Xbox Wire Staff. Forza horizon 2: What’s a drivatar, and why should i care?, 2014.
- [Sta19] Nick Statt. Openai’s dota 2 ai steamrolls world champion e-sports team with back-to-back victories, 2019.
- [Stu01] Lionhead Studios. Black & white, 2001.
- [Stu04] Bungie Studios. Halo 2, 2004.

Literaturverzeichnis

- [Stu05] Microsoft Studios. Forza motorsport, 2005.
- [Stu13] Unity-Nutzer Dreamdev Studios. Campfire pack (unity asset pack), 2013.
- [Stu18] Rockstar Studios. Red dead redemption 2, 2018.
- [SWD⁺17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
- [Tec05] Unity Technologies. Unity, 2005.
- [Tec17a] Unity Technologies. Navmesh, unity api documentation, 2017.
- [Tec17b] Unity Technologies. Navmeshagent, unity api documentation, 2017.
- [Tec17c] Unity Technologies. Unity ml-agents, github repository, 2017.
- [Tec17d] Unity Technologies. Unity ml-agents, github repository, dokumentation, 2017.
- [Tec17e] Unity Technologies. Unity ml-agents, github repository, dokumentation, training with proximal policy optimization, 2017.
- [Tec19a] Unity Technologies. The heretic, 2019.
- [Tec19b] Unity Technologies. Unity - manual: Coroutines, 2019.
- [Tec19c] Unity Technologies. Unity - manual: Occlusion culling, 2019.
- [Tec20] Unity Technologies. Unity herunterladen, 2020.
- [Thu18] Rob Thubron. Red dead redemption 2 details revealed, 2018.
- [TK17] Samuli Laine Antti Herva Jaako Lehtinen Tero Karras, Timo Aila. Audio-driven facial animation by joint end-to-end learning of pose and emotion, 2017.
- [TSS14] Maxis The Sims Studio. Die sims 4, 2014.
- [TU 18] TU Dortmund University, Southwestern University, Queen Mary University of London, University of California, IT University of Copenhagen. *Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network*. Genetic and Evolutionary Computation Conference, 2018.
- [Ult17] Ultra Ultra. Echo, 2017.
- [Unb10] Unbekannt. Orbitcamerabehavior, 2010.
- [Uni01] Carnegie Mellon University. Mocap, cmu graphics lab motion capture database, 2001.
- [UNTNC19] Unity Technologies Unity-Nutzer Terra Nova Creations. Fantasy town low poly pack ((unity asset pack)), 2019.
- [VG20] viscircle.com VisCircle GmbH. Unreal engine vs. unity: welche software sollten gamedeveloper wählen?, 2020.
- [VM13] David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller Volodymyr Mnih, Koray Kavukcuoglu. Playing atari with deep reinforcement learning, 2013.
- [Wal15] Kimberley Wallace. Cd projekt explains the witcher 3's ever-changing weather with exclusive screenshots, 2015.

Literaturverzeichnis

- [WC18] Unity-Nutzer Wand and Circles. Polygon city pack - environment and interior (unity asset pack), 2018.
- [Wex01] James Wexler. Artificial intelligence in games: A look at the smarts behind lionhead studio's black & white and where it can and will go in the future, 2001.
- [Wik19a] Inc. Wikipedia, Wikimedia Foundation. Behavior tree (artificial intelligence, robotics and control), 2019.
- [Wik19b] Inc. Wikipedia, Wikimedia Foundation. Long short-term memory, 2019.
- [YY18] Hoang M. Le Yisong Yue. Imitation learning tutorial, presentation, 2018.
- [Zub] Robert Zubek. Needs-based ai.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum

Tobias Jansing