Independent Coursework 1

Hochschule für Technik und Wirtschaft Berlin

Lösen von 3D Labyrinthen mit Unitys ML Agents

Dokumentation

Faye Bullwinkel
im Studiengang Internationale Medieninformatik
Matrikelnummer: 581885

10. Oktober 2023

Bewertung: Prof. Dr. Tobias Lenz

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren hat das Gebiet des maschinellen Lernens enorme Fortschritte gemacht und sich zu einem der dynamischsten und vielversprechendsten Bereiche der Informatik entwickelt. Nicht zuletzt durch die Veröffentlichung von Chatbot-Modellen, wie ChatGPT, zeigte sich das bemerkenswerte Potenzial von maschinellem Lernen. Es ist anzunehmen, dass der Einsatz von Machine Learning wesentlich zur Veränderung zahlreicher Bereiche in verschiedenen Branchen beitragen wird. Dies eröffnet spannende Möglichkeiten und wirft gleichzeitig faszinierende Fragen auf, wie maschinelles Lernen funktioniert und wie es eingesetzt werden kann.

Das sogenannte *Independant Coursework* an der HTW Berlin dient dazu, den Studierenden gezielt Zeit zu geben, sich mit einem Thema oder einer Technologie zu beschäftigen, die sie interessiert. Im Rahmen dieser Arbeit wurde sich dafür entschieden, sich genauer mit dem Thema Machine Learning zu beschäftigen. Diese Entscheidung wurde von persönlichen Interessen und bisherigen Erfahrungen mit der Implementierung künstlicher Intelligenz beeinflusst.

Das Gesamtziel dieses Projekts ist es, eine Unity-Erfahrung zu schaffen, die zwei KI-Systeme umfasst. Ein System erzeugt dabei dreidimensionale Labyrinthe, während das zweite System diese Labyrinthe löst. Dabei steht die persönliche, praktische Auseinandersetzung mit dem Thema Machine Learning im Vordergrund. Diese Arbeit soll eine Grundlage schaffen, das Potenzial von maschinellem Lernen in einem konkreten Anwendungsfall zu erkunden.

Der Quellcode der folgenden Implementierung kann hier[1] eingesehen werden.

[1] https://github.com/fayebullwinkel/fifi-ml-agents

2 Zielsetzung

Das Ziel dieses Projekts ist die Entwicklung einer Unity-Umgebung, die die Interaktion zwischen einem generativen und einem lösenden KI-System ermöglicht. Die fertiggestellte Umgebung besteht aus zwei Szenen, wobei der *MazeGenerationAgent* in jeder Spielrunde ein neues Labyrinth mit variabler Komplexität und Schwierigkeit erzeugt und der *MazeSolvingAgent* dieses dann löst, indem er einen Weg vom Start- zum Zielpunkt findet.

Während des Trainingsprozesses werden beide KI-Systeme kontinuierlich lernen und sich verbessern. Der *MazeGenerationAgent* soll dabei seine Fähigkeiten verfeinern und lernen, immer komplexere und vielfältigere Labyrinthe zu generieren. Gleichzeitig soll der *MazeSolvingAgent* aus seinen eigenen Erfahrungen und Fehlern lernen, um effizienter durch immer anspruchsvollere Labyrinthe navigieren zu können.

Die Implementierung des Projekts erfolgt in der Unity Entwicklungsumgebung. Nach anfänglichen Recherchen wurde das interne Machine-Learning-Framework von Unity, ML Agents, für die Implementierung der Agenten ausgewählt. Diese Entscheidung basiert auf der Tatsache, dass ML Agents speziell für die Unity Engine entwickelt wurde und somit eine nahtlose Integration von Machine Learning Modellen in ein Unity Projekt ermöglicht. Das Open-Source-Toolkit von Unity Technologies bietet zudem die Möglichkeit, Agenten mit verschiedenen Techniken zu schulen, darunter die in diesem Projekt verwendete Proximal Policy Optimization (PPO).

Im weiteren Verlauf dieser Arbeit wird der Schwerpunkt hauptsächlich auf der Entwicklung und Optimierung des *MazeSolvingAgent* liegen. Das liegt daran, dass die Umsetzung des *MazeGenerationAgent* Hauptbestandteil eines anderen parallelen Projekts ist, jedoch wird auf die Integration und das Zusammenspiel der beiden Agenten im Detail eingegangen.

Um erfolgreich durch die zuvor vom *MazeGenerationAgent* erstellten Labyrinthe zu navigieren, wird der *MazeSolvingAgent* eine Vielzahl von Parametern und Funktionen benötigen, um seine Umgebung zu erfassen und auf sie zu reagieren. Dazu gehören unter anderem eine effiziente Datenstruktur zur Repräsentation des Labyrinths und ein Belohnungssystem, das dem Agenten hilft, sein Ziel zu erreichen. Die Implementierung dieser und weiterer Parameter wird nachfolgend beschrieben und im Verlauf der Arbeit getestet, um ihre Auswirkungen auf das Lernergebnis zu bewerten.

3 Methodik

Der methodische Teil der Arbeit umfasst mehrere Schritte. Zunächst wird ein Algorithmus zur Erstellung von dreidimensionalen Labyrinthen auf einem Würfel vorgestellt. Dann wird die Implementierung von Unitys ML Agents erläutert. Insbesondere wird dabei auf die Navigation des Agenten und das entwickelte Beobachtungs- und Belohnungssystem eingegangen, das dem Agenten hilft, Entscheidungen zu bewerten und daraus zu lernen. Im Anschluss wird der Trainingsprozess beschrieben und die gewonnenen Erkenntnisse diskutiert. Abschließend wird gezeigt, wie diese Studie mit einer anderen kombiniert werden kann, um nicht nur zufällig generierte, sondern auch von einem anderen Agenten erstellte dreidimensionale Labyrinthe zu lösen.

3.1 Labyrintherstellung

Bevor mit der Implementierung der Machine-Learning-Agenten begonnen werden konnte, musste eine geeignete Trainingsumgebung geschaffen werden. Im Laufe des Projekts sollen die Agenten ein Labyrinth erhalten, das von einem anderen Agenten erstellt wurde. Zunächst musste jedoch unabhängig davon ein Algorithmus implementiert werden, der dreidimensionale Labyrinthe erzeugt.

3.1.1 Prim Algorithmus

Mithilfe des Prim Algorithmus kann ein minimaler Spannbaum in einem zusammenhängenden, ungerichteten Graphen mit gewichteten Kanten erzeugt werden („Algorithmus von Prim“, 2022). Ein Spannbaum verbindet alle Knoten eines Graphen und wird als minimal bezeichnet, wenn seine Kanten das kleinstmögliche Gesamtgewicht haben („Prim Algorithmus - minimaler Spannbaum: Beispiel“, n. d.). Da die Zielsetzung dieses Projekts die Erstellung eines Labyrinths auf der Oberfläche eines Würfels beinhaltet, wurde darauf aufbauend der Prim Algorithmus benutzt, um ein dreidimensionales geschlossenes Labyrinth (ohne Inseln) zu generieren.

Als geeignete Datenstruktur für die Labyrinthdarstellung wurde ein dreidimensionales Array gewählt. Jedes Element des Arrays enthält eine Instanz der Klasse *Cube*. Ein Würfel besitzt ein bei der Initialisierung zufällig gesetztes Gewicht, seine Position im Array (x-, y- und z-Koordinaten) und

Informationen darüber, ob der Würfel eine Wand ist und ob es sich um den Start- oder Zielwürfel handelt.

Nach der Initialisierung wird ein zufälliges Element des Arrays als Startpunkt ausgewählt, wenn sich dieses auf der Oberfläche des Labyrinths befindet[2] und als Wand markiert ist. Der Startwürfel wird dann zu einer Liste von Pfadwürfeln hinzugefügt und die Wandmarkierung wird entfernt. Um zu bestimmen, welcher Würfel dem Pfad als nächstes hinzugefügt werden soll, werden die nicht-diagonalen Nachbarn der bereits hinzugefügten Würfel, die sich ebenfalls auf der Oberfläche des Labyrinths befinden, durchlaufen. Dabei wird der Nachbar mit dem geringsten Gewicht ausgewählt, sofern er noch als Wand markiert ist. Dieser Prozess wird so lange wiederholt, bis keine gültigen Würfel mehr übrig sind, die dem Labyrinth hinzugefügt werden können. Abschließend wird jeweils ein Würfel an einer zufällig gewählten Stelle entlang der begehbaren, nicht mehr als Wand markierten Felder als Start- und Zielwürfel markiert. Das Ergebnis ist ein zusammenhängendes und lösbares Labyrinth auf der Oberfläche eines Würfels.

3.1.2 Darstellung in Unity

Die unter 3.1.1 definierte Datenstruktur wurde anschließend mithilfe der Unity Game Engine visualisiert. Zunächst werden dafür Prefabs für den Agenten, den Zielwürfel und die Wände erstellt und in Unity instanziiert. Dabei zu beachten ist, dass ihre RigidBody-Komponenten auf *kinematisch* gesetzt werden, damit sie nicht von physikalischen Kräften beeinflusst werden und von dem Labyrinth fallen.

Anschließend wird ein leeres GameObject *mazeObj* erzeugt, das als Container für das Labyrinth dient und zur Laufzeit gefüllt wird. Dazu werden alle Elemente der erstellten Datenstruktur durchlaufen und Instanzen des Prefabs *Wall* erzeugt, wenn das entsprechende Element vom Prim-Algorithmus als Wand markiert zurückgegeben wird. Die relative Position eines Wand-Würfels in Bezug auf die gegebene lokale Skalierung des Labyrinth-Würfels wird berechnet, wie im Codebeispiel 1 gezeigt.

[2]Mindestens x, y, oder z ist 0 oder gleich der Labyrinth-Kantengröße - 1

```
public Vector3 GetRelativePosition(Vector3 localScale)
{
        return new Vector3(
            (_x + 0.5f - (localScale.x / 2)) / localScale.x,
            (_y + 0.5f - (localScale.y / 2)) / localScale.y,
            (_z + 0.5f - (localScale.z / 2)) / localScale.z
        );
}
```

Codebeispiel 1: Berrechnung der relativen Labyrinth-Position

Daraufhin werden der Agent und der Zielwürfel an ihre entsprechenden Positionen, wie während des Prim-Algorithmus bestimmt, als Kinder des *mazeObj* gesetzt und eine Würfelbasis für das Labyrinth, also der Boden, auf dem sich der Agent bewegen wird, hinzugefügt. Alle instanziierten Prefabs sind würfelförmig und ragen zur Hälfte aus der Grundfläche des Labyrinth-Würfels, sodass auf dieser ein Weg entsteht. Abbildung 1 zeigt das Ergebnis beispielhaft auf einem Würfel mit lokaler Skalierung 5x5x5.

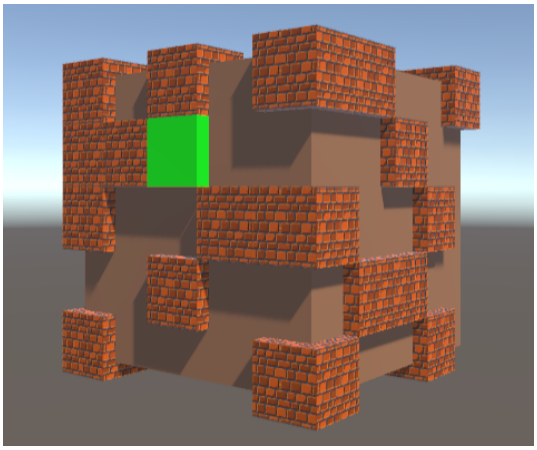

Abbildung 1: Mithilfe des Prim-Algorithmus generiertes Labyrinth.

Da sich im Laufe des Trainings das Labyrinth dynamisch neu generieren soll, wird vor der Erstellung eines neuen Labyrinths überprüft, ob bereits ein befülltes *mazeObj* in der Szene vorhanden ist. Dieses muss zu Beginn einer neuen Runde gelöscht werden. Befindet sich bereits ein aktiver Agent in der Szene, darf dieser nicht gelöscht werden, damit seine Trainingsdaten nicht verloren gehen. Der Agent und der Zielwürfel werden von ihrem Elternobjekt getrennt und das alte Labyrinth wird gelöscht. Anschließend wird ein neu-

es Labyrinth generiert und die Positionen des Agenten und des Zielwürfels werden als Kinder des Labyrinths neu gesetzt.

Für Test- und Debugging-Zwecke wurde zudem eine Funktion implementiert, die das Labyrinth so dreht, dass die Seite mit dem Startwürfel beim Start zur Kamera ausgerichtet ist.

Als optionale Erweiterung zur Beschleunigung des Trainings (siehe 3.3) wurde der Code so erweitert, dass eine variable Anzahl von Trainingsumgebungen erzeugt werden kann. Dabei besteht eine Trainingsumgebung aus je einem Labyrinth, einem Zielwürfel und einem Agenten, der das Labyrinth löst. Die Anzahl wird vor Spielbeginn manuell festgelegt, ebenso die Größe des zu erzeugenden Labyrinths. In Unity werden die einzelnen Trainingsumgebungen in einem Raster mit vordefiniertem Abstand angeordnet. Erst nach der Implementierung fiel auf, dass das gewünschte Verhalten auch mithilfe der von Unitys ML Agents bereitgestellten *TrainingAreaReplicator*-Klasse[3] erreicht werden kann. Da der Code allerdings ohnehin an die Verwendung mehrerer ML Agents in einer Szene angepasst werden musste, wurde beschlossen, die eigene Implementierung beizubehalten. Abbildung 2 zeigt das Ergebnis.

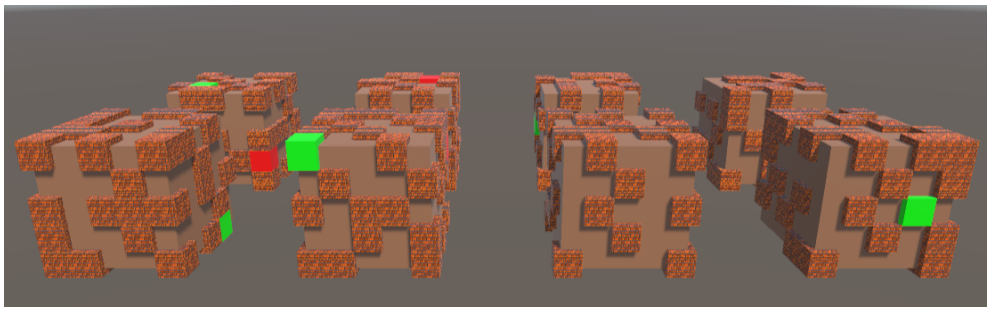

Abbildung 2: Raster mit erstellten Trainingsumgebungen für simultanes Training von acht Agenten.

3.2 Implementierung von Unitys ML Agents

Nach der Erstellung einer dreidimensionalen Testumgebung mithilfe des Prim-Algorithmus wird nachfolgend die Integration von ML Agents in das bestehende Projekt schrittweise beschrieben. Dabei werden die Installation und

[3] „Class TrainingAreaReplicator | ML Agents | 2.3.0-Exp.3“, n. d.

Konfiguration der Unity-Umgebung, die Navigation des Agenten, das Beobachtungs- und Belohnungssystem sowie der Trainingsprozess der Agenten detailliert beschrieben.

3.2.1 Erste Schritte

Nach umfangreichen Recherchen und diversen Versuchen wurde auf Basis eines Tutorials (Kelly, 2020) eine erste eigene Unity-Umgebung erstellt, die mithilfe von Reinforcement Learning einem Pinguin-Agenten beibringt, Fische zu fangen und sein Junges zu füttern (siehe Abbildung 3).

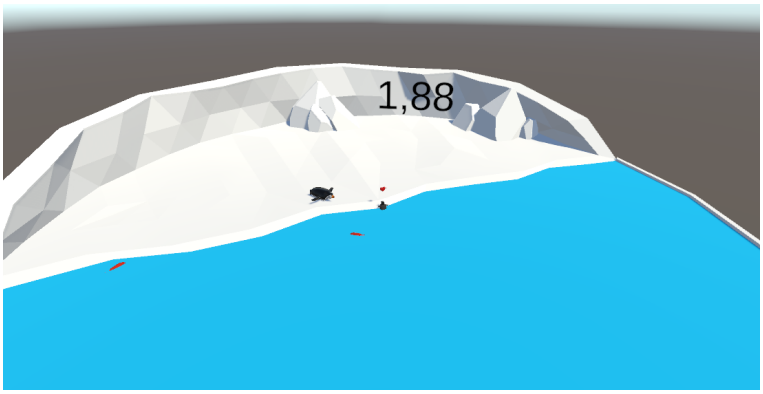

Abbildung 3: Ausschnitt des Testprojekts mit Unitys ML Agents.

Eine besondere Herausforderung war das Aufsetzen einer Python-Umgebung, um die Agenten erfolgreich zu trainieren. Es wurde Anaconda installiert, da es die Möglichkeit bietet, mehrere Python-Umgebungen zu verwalten, frei zugänglich und unabhängig vom Betriebssystem nutzbar ist. Nachdem die zur verwendeten Version von ML Agents (*v2.3.0-exp.3*) passende Python-Version (*v0.30.0*) ermittelt und installiert wurde, wurden *Torch*, *Protobuf* und *Onnx* installiert. Torch wird in einer Python-Umgebung benötigt, um maschinelles Lernen und neuronale Netze zu implementieren und zu trainieren („torch — PyTorch 2.1 documentation“, n. d.). Protobuf bietet in einer Python-Umgebung eine effiziente und plattformunabhängige Möglichkeit zur Serialisierung und Deserialisierung von Datenstrukturen für die Datenkommunikation zwischen verschiedenen Anwendungen oder Systemkomponenten („Protocol buffers“, n. d.). Mithilfe von Onnx werden Machine Learning Modelle dargestellt („ONNX | Home“, n. d.). Mit diesen Paketen wurde ein *Con-*

da environment erstellt, welches zum Training von Unitys ML Agents verwendet werden kann, indem dieses aktiviert und eine *academy* gestartet wird.

Die Academy greift auf eine zuvor erstellte Konfigurationsdatei zu. Eine der ersten Entscheidungen, die in Bezug auf die Trainingskonfiguration getroffen werden muss, ist, welche Art von Training verwendet werden soll: PPO (Proximity Policy Optimization) oder SAC (Soft Actor-Critic). Es wurde PPO gewählt, da dieser Lernalgorithmus intuitiv zu implementieren ist, in Unitys ML Agents als Standard verwendet wird und in vielen verschiedenen Umgebungen zuverlässige Ergebnisse liefert (Hanuelcp, 2020). Alle anderen Einstellungen wurden zunächst wie von Kelly, 2020 vorgeschlagen übernommen.

3.2.2 Navigation und Aktionen des Agenten

Die Navigation von Agenten in Unity erfolgt über sogenannte *actions*. Aktionen können entweder diskret oder kontinuierlich sein, wobei kontinuierliche Aktionen einen unendlichen Raum von möglichen Aktionen in Form von kontinuierlichen Werten zulassen. Diskrete Aktionen hingegen stellen eine begrenzte Menge klar definierter, separater Aktionen dar (Unity-Technologies, 2017-2020a).

Es wurde entschieden, dass sich der *MazeSolvingAgent* entlang der Datenstruktur bewegt und seine Bewegungen anschließend in Unity visualisiert werden. Dies beschränkt die möglichen Aktionen des Agenten auf sechs Schritte, je einen in positiver und negativer x-, y- und z-Richtung, sowie die Entscheidung, sich nicht zu bewegen. Entsprechend wurde ein diskreter Aktionsraum mit einer übergeordneten Aktion *Bewegen* und sieben Abstufungen gewählt. Gleichzeitig wurde die gegebene *Heuristic()*-Funktion überschrieben, die Aktionen auf Tastaturbefehle abbildet und somit das Testen nur mit menschlicher Eingabe ermöglicht.

Bei jeder neuen Aktion, ob vom Agenten gesendet oder durch Eingabe ausgelöst, wird geprüft, ob die Bewegung in die gewählte Richtung erlaubt ist oder nicht. Eine Aktion ist zulässig, wenn die neue Position innerhalb der Grenzen des Labyrinths bleibt, auf der Oberfläche des Würfels liegt und sich an der entsprechenden Stelle im Labyrinth keine Wand befindet. Ist dies der Fall, wird die aktuelle Position mit der neuen Position im Array überschrieben, die Position des Agenten im Labyrinth wie im Codebeispiel 1 berechnet und anschließend in Unity gesetzt. Damit sich der Agent nur um eine Einheit in die gewählte Richtung bewegt, muss in der Komponente *Decision Re-*

quester des Agenten zusätzlich die *Decision Period* auf eins gesetzt werden. Wenn der Agent den Zielwürfel erreicht (der *Cube* an der aktuellen Position im Array entspricht dem Zielwürfel), wird die aktuelle Trainingsumgebung gelöscht, ein neues Labyrinth erzeugt und der Agent und der Zielwürfel auf die neuen Start- und Endpositionen gesetzt.

Tritt während der Generierung des Labyrinths mit dem Prim-Algorithmus oder während der Instanziierung der Objekte in Unity ein Fehler auf, wird dieser abgefangen und der Prozess beginnt von vorne. Dieser Fall tritt in der Regel nicht ein, ist aber als Fallback-Option in den Code integriert, da z.B. bei zwei unmittelbar hintereinander gestarteten Episoden das zufällige Landen des Agenten auf dem Zielwürfel zu einem Ladefehler des neuen Labyrinths führt. Dieser Fall kann in der finalen Umgebung nicht auftreten, da die vom *MazeGenerationAgent* gesetzten Start- und Endpositionen ohnehin unterschiedlich sind, musste aber während des Trainings berücksichtigt werden.

3.2.3 Beobachtungs- und Belohnungssystem

Damit der *MazeSolvingAgent* das unter 3.1 erzeugte Labyrinth lösen kann, muss er Informationen über seine Umgebung erhalten, die ihm helfen, die Welt selbständig zu erkunden und zu verstehen. Außerdem müssen seine Aktionen belohnt oder bestraft werden, damit er lernen und die Ziele der ihm zur Verfügung gestellten Umgebung verstehen kann. Im Folgenden werden hauptsächlich die anfänglich verwendeten Beobachtungen und Belohnungen beschrieben. Im Laufe der Versuche wurden verschiedene Kombinationen ausprobiert. Mehr dazu im Abschnitt 3.3.

Um dem Agenten Informationen über seine Umgebung zur Verfügung zu stellen, können dem ihm zur Verfügung stehenden *VectorSensor* sogenannte *Observations* hinzugefügt werden. Zu Beginn wurden dem Agenten seine aktuelle Position im Raum und die Position des Zielwürfels mitgeteilt. Im Laufe der Tests wurde jedoch entschieden, dem Agenten die entsprechenden Positionen innerhalb der Datenstruktur zu übergeben. Zusätzlich wird dem Agenten ein dreidimensionales Array von *booleans* zur Verfügung gestellt, wobei jedes Element Informationen darüber enthält, ob sich an der entsprechenden Stelle in der Labyrinth-Datenstruktur eine Wand befindet oder nicht. Elemente innerhalb des Würfels werden als Wände behandelt. Außerdem erhält der Agent Informationen darüber, wie viele Würfel er bereits entdeckt hat. Daraus ergeben sich für einen Würfel der lokalen Skalierung 7x7x7 ins-

gesamt 350 Beobachtungen, jeweils drei für die dreidimensionale Position des Agenten und des Zielwürfels, 343 für die Wandinformation und eine für die Anzahl der neu besuchten Würfel.

Es fällt auf, dass die Gesamtzahl der Beobachtungen stark von der Größe des Labyrinths abhängt und bei jeder Größenanpassung in Unity neu eingestellt werden muss. Als Alternative wurde erwogen, dem Agenten statt der gesamten Darstellung des Labyrinths nur die Seite des Würfels zu übergeben, auf der er sich gerade befindet. Da sich dadurch die Problematik des dynamischen *VectorSpace* nicht ändert und die Datenmenge bei Betrachtung des gesamten dreidimensionalen Feldes kein Problem darzustellen scheint, wurde diese Idee verworfen. Weitere Überlegungen waren, dem Agenten Informationen darüber zu geben, ob seine jeweiligen Nachbarn bereits besucht wurden oder nicht, und ob es ausreicht, ihm nur Informationen über Nachbarn in einer bestimmten Entfernung zu geben. Ein entsprechender Ansatz wurde verfolgt, wurde aber aus Zeit-/Kostengründen verworfen, da die Übergabe von Nachbarn an den Kanten nicht trivial ist.

Das ursprüngliche Belohnungssystem besteht aus fünf Komponenten. In diesem Fall werden auch Strafen als negative Belohnungen in das Belohnungssystem einbezogen und alle Belohnungswerte liegen zwischen -1 und 1. Zunächst wird dem Agenten pro Runde eine Mindeststrafe auferlegt, um ihn zu weiteren Entscheidungen zu motivieren. Erreicht der Agent das Ziel, wird ihm eine Belohnung von +1 zugeschrieben. Führt der Agent eine nicht erlaubte Aktion durch (wie unter 3.2.2 beschrieben), wird er mit einem Wert von -1 bestraft. Außerdem wird nach jeder erfolgreichen Bewegung überprüft, ob sich der Agent in einer Sackgasse befindet. Um dies zu überprüfen, werden von der neuen Position aus alle sechs möglichen Richtungen durchlaufen und gezählt, wie viele benachbarte Würfel sich noch innerhalb des Labyrinths befinden, aber nicht in dessen Mitte sind und keine Wand repräsentieren. Die Aktion *Nicht bewegen* wird dabei übersprungen. Die zurückgegebenen Würfel entsprechen jeweils einem möglichen Zug. Ist die Anzahl der möglichen Züge gleich eins, befindet sich der Agent in einer Sackgasse und erhält eine Strafe. Die fünfte und letzte Strafe erhält der Agent abhängig davon, wie oft er einen Würfel bereits besucht hat. Dabei wird nach jeder Bewegung ein Zähler erhöht, der angibt, wie oft ein Würfel bereits besucht wurde. Um den Strafwert zwischen -1 und 0 zu halten, wird anschließend 1 durch den Zähler dividiert, das Ergebnis von 1 subtrahiert (invertiert) und als negative Belohnung an den Agenten zurückgegeben.

Anfangs wurde auch die euklidische Distanz nach einem ähnlichen Ver-

fahren bewertet, wobei eine kürzere Distanz einen größeren Vorteil brachte. Es stellte sich jedoch schnell heraus, dass diese Belohnung einen stärkeren Effekt hatte als die Bestrafung und dass sich die Agenten während des Trainings entlang der Wände bewegten, wenn das Ziel auf der anderen Seite lag. Um das Ziel zu erreichen, hätten die Agenten andere Seiten des Würfels überqueren müssen, wodurch sich die Distanz deutlich vergrößert hätte. Dieses Verhalten wurde durch die verhängten Strafen nicht gefördert und der Ansatz wurde verworfen.

3.3 Training

Nach dem Aufbau der Unity-Umgebung und der Implementierung eines ML Agent, dessen Heuristik es ermöglicht, ein dreidimensionales Labyrinth mithilfe von Benutzereingaben zu lösen, war es an der Zeit, Unitys ML Agents für diese Aufgabe zu trainieren. Dazu wurde zunächst die unter 3.2.1 erstellte Python-Umgebung aktiviert und anschließend ein Trainingslauf auf Basis der Konfigurationsdatei gestartet. Jeder Trainingslauf erzeugt ein eigenes trainiertes neuronales Netz, das anschließend an den *MazeSolvingAgent* in Unity übergeben werden kann. Der Agent verwendet dieses dann, um weitere Labyrinthe zu lösen, ohne dass er selbst weiter lernt. Es ist zu beachten, dass ein neuronales Netz nur für die Trainingsumgebung funktioniert, in der es trainiert wurde. Verändert man die Beobachtungen, muss auch ein neues Netz trainiert werden (Kelly, 2020).

3.3.1 Testläufe mit verschiedenen Konfigurationen

In den ersten Trainingsläufen wurde jeweils nur eine Umgebung generiert und ein Agent getestet. Es fiel auf, dass der Agent nur einen kleinen Teil des Labyrinths erkundete und sich nach einigen Runden deutlich weniger zu anderen Feldern bewegte, sich also häufig entschied, keine Bewegung durchzuführen. Die Aktion *Nicht bewegen* wurde daraufhin während des Trainings aus dem diskreten Handlungsraum ausgeschlossen, sodass sich der Agent in den folgenden Durchläufen immer für eine Bewegung entscheiden muss. Außerdem wurden die Belohnungen nach Recherchen auf die unter 3.2.3 beschriebenen Werte zwischen -1 und 1 angepasst, um ein einheitliches Verhältnis zu haben (Unity-Technologies, 2017-2020b). Für eine detailliertere Untersuchung der Trainingsergebnisse wurde *TensorFlow* verwendet (Unity-Technologies, 2017-2022). Zwischenzeitlich wurde zudem das von Unitys ML Agents gege-

bene *Curiosity Reward Signal* getestet, die Ergebnisse wurden davon aber nicht wesentlich beeinflusst.

In den folgenden Testläufen wurden mehrere Agenten gleichzeitig und mit unterschiedlicher Länge getestet, ohne dass das Training signifikante Fortschritte zeigte. Daher wurde beschlossen, die Konfiguration der ML Agents so weit wie möglich zu vereinfachen. Die Beobachtungen wurden auf die Start- und Zielposition und das *boolean*-Array reduziert, und dem Agenten wurde nur eine kleine Strafe pro Zug und eine Belohnung bei Erreichen des Ziels zugewiesen. Zusätzlich wurde dem Agenten ein statisches Labyrinth (mit einer lokalen Skalierung von 7x7x7) für das Training zur Verfügung gestellt. Es stellte sich schnell heraus, dass diese Wahl den signifikantesten Unterschied in den Ergebnissen lieferte. Die Abbildungen 4 und 5 zeigen einen direkten Vergleich zwischen dem Training mit einem statischen und einem dynamischen Labyrinth. Es ist zu beachten, dass ein Agent in einem dynamischen Labyrinth nach einigen Durchgängen die maximale Strafe von -1 pro Durchgang erhält und sein Ziel nicht vor Ablauf der vorgegebenen maximalen Schrittzahl erreicht. In einem dynamischen Labyrinth fand der Agent das Ziel in 3/32 Durchgängen. In einem statischen Labyrinth dagegen in 165/175 Durchgängen bei einer konstanten maximalen Anzahl von 150000 Entscheidungen.

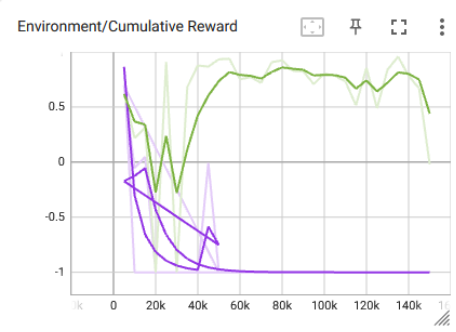

Abbildung 4: Vergleich der **Gesamtbelohnung** beim Training mit statischem (grün) und dynamischem (violett) Labyrinth.

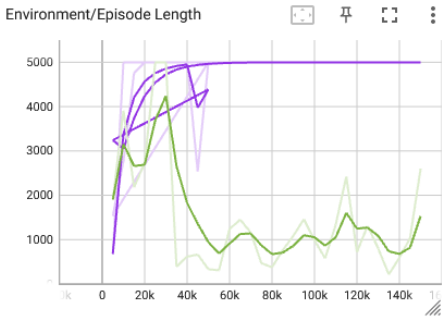

Abbildung 5: Vergleich der **Zeit pro Episode** beim Training mit statischem (grün) und dynamischem (violett) Labyrinth.

Nachdem bestätigt werden konnte, dass die ML Agents ein statisches Labyrinth mit den gegebenen Beobachtungen und Belohnungen lösen kön-

nen, wurden anschließend weitere Konfigurationen getestet, die es den ML Agents ermöglichen sollten, Labyrinthe zu lösen, die in jeder Runde neu erzeugt wurden. Dazu wurde zunächst das *Curiosity Reward Signal* wieder in die Konfigurationsdatei aufgenommen. Dabei wurde sein Einfluss auf das Gesamtergebnis erhöht (von *strength* 0,02 auf den Maximalwert 0,1. Im Verhältnis dazu steht das extrinsische Belohnungssystem, mit einer Stärke von 1,0). Dies scheint sich positiv auf das Lernergebnis auszuwirken, da der Agent relativ zuverlässig das Labyrinth erkundet und das Ziel findet. Allerdings benötigt er dafür eine erhebliche Anzahl von Schritten. Um die Erkundung seiner Umgebung weiter zu fördern, wurde die Entdeckung neuer Felder zusätzlich belohnt. Zunächst wurde in mehreren Durchläufen eine scheinbar gute Gewichtung von 0,05f pro neu entdecktem *Cube* ermittelt. Später stellte sich jedoch heraus, dass das *Curiosity Reward Signal* allein zu vergleichbaren Ergebnissen führte, und der Ansatz wurde verworfen.

In weiteren Iterationen wurde die Betrachtung der gesamten Datenstruktur auf eine festgelegte Anzahl von Schritten in alle Richtungen reduziert. Die Tests ergaben jedoch deutlich schlechtere Ergebnisse als bei Verwendung der gesamten Datenstruktur, weshalb dieser Ansatz verworfen wurde. Auch die aktuelle Entfernung des Agenten zum Ziel als weitere Beobachtung zu übergeben, verschlechterte das Ergebnis. Die Entscheidung, das *boolean*-Array, das die Elemente der Datenstruktur enthält, die eine Wand darstellen, zu erweitern und zusätzlich die aktuelle Position des Agenten und die des Zielwürfels zu übergeben, wirkte sich jedoch positiv auf die Ergebnisse aus.

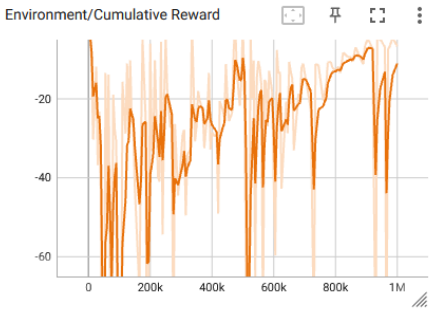

Abbildung 6: Lernkurve eines Trainings mit acht Agenten und eine Million getroffener Entscheidungen.

Der Ansatz, einen Agenten nicht nur zu bestrafen, wenn er eine unzulässige Aktion ausführt, sondern ihn auch wieder an den Ausgangspunkt zurück-

zubringen, zeigte eine Lernkurve bei der Lösung verschiedener Labyrinthe. Wie Abbildung 6 zeigt, steigt der erhaltene Belohnungswert, auch über mehrere unterschiedliche Labyrinthe hinweg an. Das Training wurde mit acht Agenten, jeweils auf einem 3x3x3 Labyrinth durchgeführt. Die Anpassung an ein Labyrinth geeigneter Größe bedeutet im Rahmen der für dieses Projekt zur Verfügung stehenden Ressourcen eine zu drastische Verlängerung der Trainingszeit und wurde daher nicht weiter verfolgt.

In den folgenden Testläufen wurden verschiedene Belohnungs- und Bestrafungskonfigurationen und deren Einfluss auf die Leistung des Agenten untersucht. Es zeigte sich, dass die Strafen zu hoch angesetzt waren und die Belohnungen überlagerten. Generell konnte festgestellt werden, dass die Verwendung mehrerer unterschiedlicher Belohnungen keinen signifikanten Einfluss auf den Erfolg des Agenten hatte. Es wurde daher beschlossen, eine möglichst einfache Konfiguration beizubehalten, um den Agenten für das Training mit den Daten des parallel entwickelten Agenten für die Labyrinth-Generierung vorzubereiten.

3.3.2 Erstellung einer erfolgversprechenden Konfiguration

Nachdem über mehrere Tage verschiedene Konfigurationen getestet und miteinander verglichen wurden, wurde auf Basis der gesammelten Ergebnisse die erfolgversprechendste Konfiguration festgelegt. Diese wurde anschließend verwendet, um nicht nur zufällig erzeugte Labyrinthe zu lösen, sondern auch Labyrinthe, die von einem anderen Agenten generiert werden.

Das Erreichen des Zielwürfels wurde als erstes und wichtigstes Ziel definiert. Dementsprechend erhält der Agent eine maximale Belohnung von $+1$, wenn er das Ziel des Labyrinths erreicht. Dies soll das gewünschte Verhalten, die Umgebung nach dem Zielwürfel abzusuchen, fördern. Als nächstes sollte der Fortschritt des Agenten belohnt werden. Die euklidische Distanz wurde bereits unter 3.2.3 ausgeschlossen, jedoch ergab sich während des Trainings ein weiterer vielversprechender Ansatz. Laut diesem erhält der Agent eine Belohnung, die von seinem Fortschritt im Labyrinth abhängt. Dabei erhält er immer dann eine konstante Belohnung von $+0.01$, wenn sein aktueller Abstand zum Ziel kleiner ist als der vorherige kleinste Abstand. Eine Belohnung wird zudem nur einmal pro besuchtem Feld vergeben. Außerdem wurde die Mindeststrafe pro Aktion beibehalten, um den Agenten zu motivieren, das Labyrinth so schnell wie möglich zu durchqueren.

3.4 Projektzusammenführung

Ziel des Projekts ist neben der Implementierung eines Agenten, der in der Lage ist, ein zufällig generiertes dreidimensionales Labyrinth auf der Oberfläche eines Würfels zu lösen, auch die Zusammenführung mit einem anderen Projekt. Das ersetzt den im Rahmen dieser Arbeit vorgestellten Teil des Prim-Algorithmus und liefert dem *MazeSolvingAgent* pro Spieldurchlauf ein generiertes Labyrinth, das dieser dann löst.

Um die beiden Projekte zusammenführen zu können, musste die im Projekt *MazeGeneration* verwendete Datenstruktur in die in dieser Arbeit vorgestellte Datenstruktur umgewandelt werden. Die *MazeGeneration*-Datenstruktur besteht aus sechs Rastern, wobei jedes Raster aus (in diesem Fall) neun Zellen besteht und jede Zelle vier Ecken und vier Wände hat, die der Agent entfernen kann. Überträgt man diese Datenstruktur in ein dreidimensionales Array, so entsprechen Zellen, Ecken und Wände jeweils einem *Cube*, wobei eine Zelle ihre Ecken und Wände mit anderen Zellen teilt. Für die Position in der neuen Datenstruktur ist es wichtig zu wissen, auf welcher Seite sich die aktuelle Zelle befindet, da abhängig davon zwei der drei Achsen durch die Koordinaten der Zelle bestimmt werden. Die dritte Koordinate ist entweder das Minimum (*0*) oder das Maximum (*Rastergröße * 2 + 1*) - 1. Für die hintere, linke und untere Seite muss zudem jeweils eine Achse gespiegelt[4] werden. Abbildung 7 zeigt die Umwandlung eines 3x3 Rasters mit einem Beispielpfad.

Da sich durch diese Transformation für die hier verwendete Datenstruktur eine neue Kantengröße durch *Rastergröße * 2 + 1* ergibt, musste eine Rastergröße vereinbart werden, die einerseits für den *MazeGenerationAgent* groß genug ist, um damit ein Labyrinth zu erzeugen, andererseits den Wertebereich der vom *MazeSolvingAgent* verwendeten Datenstruktur nicht zu sehr aufbläht. Es wurde eine Konvertierung von einem 3x3x3 Würfel in einen 7x7x7 Würfel gewählt.

Anschließend wurde die Unity-Umgebung an die Änderungen angepasst, indem die während der Entwicklung unabhängigen Szenen miteinander verbunden wurden. Wird das Programm gestartet, beginnt der *MazeGenerationAgent* ein Labyrinth in seiner Szene zu generieren, indem er die ihm zur Verfügung stehenden Informationen und Anforderungen auswertet. Die Startposition des Agenten wird zu Beginn zufällig gesetzt, die Zielposition

[4]Beispielhaft für die hintere Seite: *position.x = (Size - 1) - (cell.X * 2 + 1)*, mit *Size* gleich einer Kantenlänge im dreidimensionalen Array.

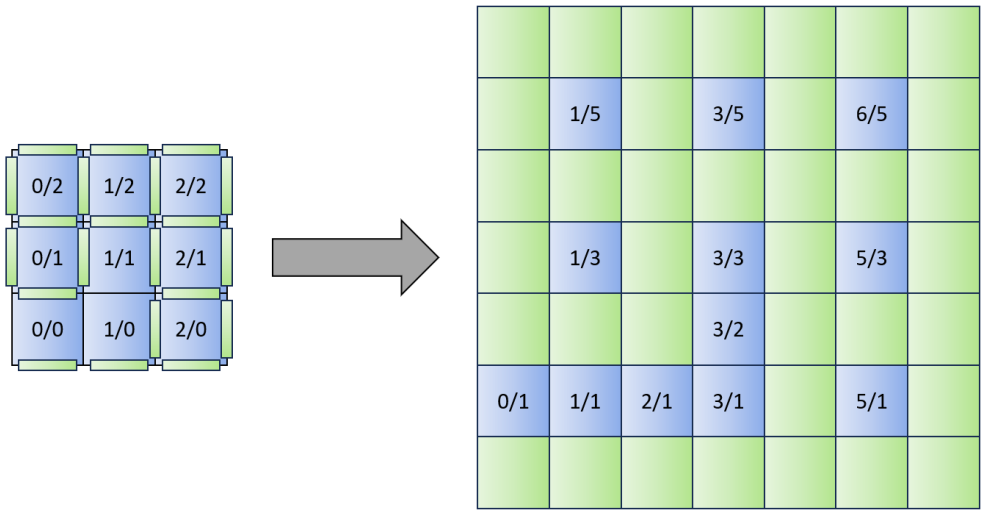

Abbildung 7: Umwandlung einer Seite der rasterbasierten Datenstruktur in eine Schicht des dreidimensionalen Arrays.

wird vom Agenten zur Laufzeit bestimmt. Wenn der Agent sich entschieden hat, ein Zielfeld zu platzieren, endet seine aktuelle *Episode*, die erzeugte Datenstruktur wird wie oben beschrieben konvertiert und das neu erzeugte *SharedMaze* wird an die Szene des *MazeSolvingAgent* übergeben. Der Agent wird an die Startposition bewegt und beginnt selbständig das für ihn generierte Labyrinth zu durchlaufen, bis er das Ziel gefunden hat.

Nach der erfolgreichen Erstellung einer Unity-Umgebung, die die Kernelemente beider Arbeiten vereint, mussten noch Änderungen am bestehenden Code vorgenommen werden. So werden zum Beispiel die Beleuchtungseinstellungen nicht automatisch übernommen, wenn eine Szene erst durch eine andere aufgerufen wird, sondern müssen separat erstellt und gespeichert werden. Außerdem wurde der Code so erweitert, dass standardmäßig ein von einem anderen Agenten generiertes Labyrinth verwendet wird und nur dann auf die Generierung durch den Prim-Algorithmus zurückgegriffen wird, wenn ein solches Labyrinth nicht vorhanden ist. Wichtig zu beachten ist dabei die Reihenfolge, in der die verschiedenen Funktionen zu Beginn des Spiels oder am Ende einer Episode aufgerufen werden. So wird beispielsweise zu Beginn des Spiels zunächst die Funktion *Start()* aufgerufen, die es ermöglicht, einen Agenten zu initialisieren und anschließend seine Eigenschaften festzulegen (das Labyrinth, zu dem er gehört, die Start- und Zielwürfel und

die relative Position im Raum, die sich jeweils dynamisch an die Anzahl der Trainingsumgebungen anpasst), bevor der Agent auf diese zugreift. Wenn der *MazeGenerationAgent* seine Episode beendet und das erzeugte Labyrinth an die neue Szene übergibt, wird zuerst die *OnEpisodeBegin()*-Funktion des *MazeSolvingAgent* aufgerufen. Diese greift auf die in der *Start()*-Funktion erzeugten Werte zu, die zu diesem Zeitpunkt noch nicht existieren. Der Agent führt also Aktionen aus, ohne dass zuvor ein Labyrinth erstellt wurde. Um dem entgegenzuwirken, wird vor der Ausführung einer neuen Aktion geprüft, ob alle erforderlichen Eigenschaften des Agenten bereits gesetzt wurden.

Die *MazeGeneration*-Szene enthält ein weitaus detaillierteres Kamerasystem zur Darstellung des Labyrinth-Würfels. Das System besteht aus einer Hauptkamera, die sich langsam um das in Unity erstellte Labyrinth dreht, und sechs zusätzlichen Kameras, die die sechs Seiten des Würfels aus der Vogelperspektive aufnehmen und auf einem Canvas neben dem sich drehenden Labyrinth abbilden. Bisher wurde in diesem Projekt zu Beginn des Spiels die Seite, auf der sich der Startwürfel befindet, zur Kamera gedreht und die Bewegungen im internen *Sceneview* von Unity beobachtet. Für die Präsentation der Ergebnisse bietet das von Vivienne Drongowski[5] implementierte Kamerasystem einen viel anschaulicheren und umfassenderen Einblick. Nach Absprache konnte das System wiederverwendet und nach kleinen Anpassungen in die *MazeSolving*-Szene eingebunden werden. Abbildung 8 zeigt das Ergebnis.

[5]Drongowski, V. (2023). *Generieren von 3D Labyrinthen mit Unitys ML Agents* [Independent Coursework Projektbericht, Hochschule für Technik und Wirtschaft Berlin]

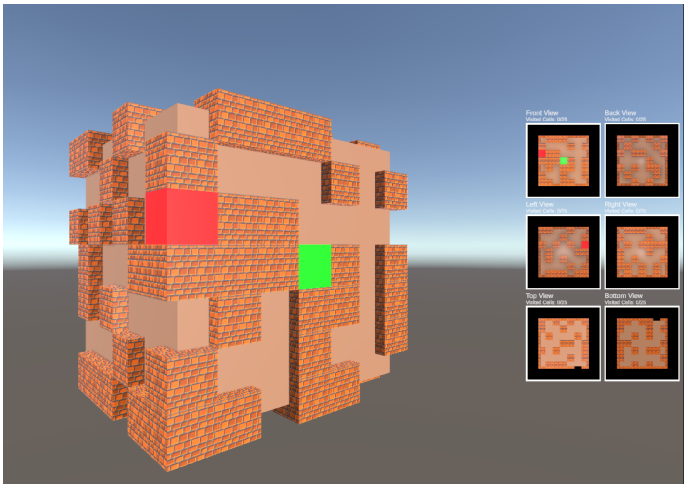

Abbildung 8: Darstellung der *MazeSolving*-Szene mit einem Kamerasystem, das aus dem Parallelprojekt übernommen werden konnte.

4 Fazit

Im Rahmen dieses Projekts wurde eine Unity-Umgebung geschaffen, in der dreidimensionale Labyrinthe in Würfelform zufällig generiert und anschließend von einem ML Agent gelöst werden. Darüber hinaus wurde das Projekt mit einem Agenten verknüpft, der in der Lage ist, selbständig ein Labyrinth auf einem Würfel zu erzeugen. Nach erfolgreicher Transformation einer seitenbasierten Datenstruktur in ein dreidimensionales Array, welches das Labyrinth repräsentiert, löst eine Instanz der im Rahmen dieser Arbeit erstellten ML Agents das Labyrinth selbständig. Ziel ist es dabei, den Zielwürfel mit möglichst wenigen Schritten zu erreichen.

4.1 Erkenntnisse

Die Kombination von Reinforcement Learning und Unitys ML Agents wurde als vielversprechende Methode bewertet, um autonome Agenten in 3D-Umgebungen zu trainieren. Gerade in statischen oder weniger umfangreichen Umgebungen lernen die ML Agents schnell, das Ziel mit hoher Genauigkeit zu finden. Gleichzeitig zeigten die Trainingsdurchläufe, dass das Thema sehr komplex ist und auf vielfältige Art und Weise erweitert und angewendet werden kann.

Im Rahmen dieser Arbeit wurden zudem wichtige Erkenntnisse darüber gewonnen, welchen Einfluss die ML-Agents-Konfigurationsdatei auf das maschinelle Lernen mit PPO hat und welche Werte geeignet sind, um einem Agenten das selbständige Lösen eines dreidimensionalen Labyrinths beizubringen. Es war hilfreich, zunächst eine möglichst einfache Testumgebung zu erstellen und diese dann schrittweise um weitere Anforderungen zu erweitern. Es ist anzumerken, dass die Integration von Visualisierungstools in die Unity-Umgebung eine nützliche Ergänzung war, um die Funktionsweise der Agenten besser zu verstehen.

Die Zusammenführung mit einem anderen Projekt, das dreidimensionale Labyrinthe erstellt, die dann mit dem in dieser Arbeit vorgestellten neuronalen Netz gelöst werden, hat unterschiedliche Zugänge zur Datenstruktur, die das Labyrinth darstellt, verdeutlicht. Die Kombination der beiden Projekte leistete einen wichtigen Beitrag zur Übertragbarkeit der gewonnenen Erkenntnisse auf andere Problemstellungen. Gleichzeitig förderte die Zusammenarbeit den Austausch über die Herausforderungen bei der Entwicklung von Unitys ML Agents und ermöglichte kollaborative Lösungsansätze.

Zusammenfassend wurde das Thema maschinelles Lernen ausführlich behandelt. Verschiedene Ansätze des Reinforcement Learning wurden näher beleuchtet und wertvolle Erfahrungen mit Unitys ML Agents gesammelt.

4.2 Ausblick

Neben der weiteren Optimierung der Agenten, um ihre Leistungsfähigkeit zu verbessern und sicherzustellen, dass sie ein breites Spektrum von Labyrinthen zuverlässig unterschiedlicher Größe bewältigen können, bleibt der Vergleich mit anderen Ansätzen des maschinellen Lernens, deren Details im Rahmen dieser Arbeit nur angerissen wurden.

Auch die Interaktion der beiden Agenten könnte weiter ausgebaut werden. So könnten sie sich beispielsweise während des Trainings jeweils gegenseitig beobachten. Der *MazeGenerationAgent* könnte vom *MazeSolvingAgent* lernen, bei welchen Labyrinthen dieser besonders lange braucht, um das Ziel zu finden. Umgekehrt könnte der *MazeSolvingAgent* durch Beobachtung der Bewegungen und Aktionen des anderen Agenten lernen, wie dieser Labyrinthe baut und welchen Weg er am besten wählen sollte.

Insgesamt gibt diese Arbeit einen Einblick in die Arbeit mit Unitys ML Agents, zeigt die Vielseitigkeit und Komplexität von Machine Learning und regt dazu an, sich weiter mit dem Thema zu beschäftigen.

Literatur

Algorithmus von Prim. (2022). *de.wikipedia.org*. https://de.wikipedia.org/wiki/Algorithmus_von_Prim

Class TrainingAreaReplicator | ML Agents | 2.3.0-Exp.3. (n. d.). Verfügbar 1. Oktober 2023 unter https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/Unity.MLAgents.Areas.TrainingAreaReplicator.html

Hanuelcp. (2020). Reddit - Dive into anything. https://www.reddit.com/r/reinforcementlearning/comments/gl1ov2/why_would_anyone_use_ppo_over_sac_td3_ddpg_and/

Kelly, A. (2020). Reinforcement Learning Penguins (Part 1/4) | Unity ML-Agents — Immersive Limit. *Immersive Limit*. https://www.immersivelimit.com/tutorials/reinforcement-learning-penguins-part-1-unity-ml-agents

ONNX | Home. (n. d.). Verfügbar 23. August 2023 unter https://onnx.ai/

Prim Algorithmus - minimaler Spannbaum: Beispiel. (n. d.). Verfügbar 12. September 2023 unter https://studyflix.de/informatik/prim-algorithmus-1293

Protocol buffers. (n. d.). Verfügbar 23. August 2023 unter https://protobuf.dev/

torch — PyTorch 2.1 documentation. (n. d.). Verfügbar 23. August 2023 unter https://pytorch.org/docs/stable/torch.html

Unity-Technologies. (2017-2020a). ml-agents/docs/Learning-Environment-Design-Agents.md at develop · Unity-Technologies/ml-agents. https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Design-Agents.md#actions-and-actuators

Unity-Technologies. (2017-2020b). ml-agents/docs/Learning-Environment-Design-Agents.md at develop · Unity-Technologies/ml-agents. https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Design-Agents.md#vector-observation-summary--best-practices

Unity-Technologies. (2017-2022). Using TensorBoard to observe Training - Unity ML-Agents Toolkit. https://unity-technologies.github.io/ml-agents/Using-Tensorboard/#exporting-data-from-tensorboard