IM Fakultät
Informatik und Medien

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Studiengang Informatik
der Fakultät Informatik und Medien
der Hochschule für Technik, Wirtschaft und Kultur Leipzig

Evaluierung von Reinforcement Learning-Algorhitmen anhand eines Würfelspiels in Unity

vorgelegt von: Tony Lenz
Geburtsort- und datum: Leipzig, den 07.01.1993
Abgabe: Leipzig, den 3. März 2024

Erstgutachter: Prof. Dr.-Ing. Müller ERSTGUTACHTER, HTWK Leipzig
Zweitgutachter: Prof. Dr.-Ing. Bleymehl ZWEITGUTACHTER, HTWK Leipzig

HTWK Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Inhaltsverzeichnis

1 Einleitung

In der heutigen Zeit stehen wir an der Schwelle einer digitalen Revolution, in der maschinelles Lernen und künstliche Intelligenz eine immer bedeutendere Rolle spielen. Insbesondere das Gebiet des Reinforcement Learning hat in den letzten Jahren enorme Fortschritte gemacht und findet Anwendung in einer Vielzahl von Bereichen, von der Robotik bis hin zu Finanzen. Diese Entwicklung bietet aufregende Möglichkeiten, komplexe Probleme zu lösen und intelligente Systeme zu entwickeln, die in der Lage sind, eigenständig zu lernen und Entscheidungen zu treffen.

Reinforcement Learning (RL) hat bereits beeindruckende Erfolge erzielt, indem es Algorithmen entwickelt hat, die komplexe Spiele wie Go auf einem kompetitiven Niveau spielen können und sogar über menschliche Fähigkeiten hinausgehen. Darüber hinaus wurde RL erfolgreich eingesetzt, um die Effizienz und Leistung von Serverfarmen bei Unternehmen wie Google zu optimieren. Diese Anwendungen verdeutlichen die Vielseitigkeit und Leistungsfähigkeit von Reinforcement Learning bei der Bewältigung verschiedenster Herausforderungen und unterstreichen seine Fähigkeit, komplexe Probleme zu lösen und neue Lösungswege zu finden.

In dieser Bachelorarbeit liegt der Fokus auf der Implementierung und dem Training eines Reinforcement Learning-Agenten für das Würfelspiel 'Noch mal'. 'Noch mal' ist ein Würfelspiel, das Strategie und Glück erfordert. Ziel dieser Arbeit ist es, einen Agenten zu entwickeln, der in der Lage ist, das Spiel zu erlernen und auf einem kompetitiven Niveau zu spielen.

Der Einsatz von Reinforcement Learning zur Bewältigung komplexer Spiele wie 'Noch mal' bietet eine interessante Herausforderung und die Möglichkeit, die Leistungsfähigkeit dieser Techniken zu demonstrieren. Durch die Implementierung eines RL-Agenten für dieses Spiel lässt sich untersuchen, wie gut maschinelle Lernmodelle in der Lage sind, komplexe Entscheidungsprobleme zu lösen und Strategien zu entwickeln, um ein definiertes Ziel zu erreichen.

Diese Bachelorarbeit zielt darauf ab, einen Beitrag zum Verständnis der Anwendung von Reinforcement Learning in der Spieleentwicklung zu leisten und Einblicke in die Leistungsfähigkeit dieser Techniken zu bieten. Durch die Implementierung eines RL-Agenten für 'Noch mal' sollen neue Erkenntnisse darüber gewonnen werden, wie maschinelles Lernen zur Entwicklung intelligenter Systeme in spielerischen Umgebungen eingesetzt werden kann.

2 Theoretische Grundlagen

2.1 Neuronale Netze

Neuronale Netze (=NN) sind ein Modell für künstliche Intelligenz nach dem Vorbild des Gehirns. Es besteht aus mehreren Schichten verknüpften Neuronen, welche numerische Informationen verarbeiten und in einen Output umwandeln. Die Ausgänge der vorderen Neuronen sind dabei immer mit den Eingängen der Neuronen der nächsten Schicht verknüpft. Jedes Neuron besitzt eine Aktivierungsfunktion, welche entscheidet ob es aktiv ist oder nicht. Neuronen geben Werte von 0 (passiv) bis 1 (aktiv) an dahinterliegende Neuronen. Richtig trainierte Neuronale Netze können gute Antworten für komplexe Problemstellungen geben, so liefern sie Beispielsweise in der Mustererkennung gute Ergebnisse.
Wird ein NN initialisiert werden Kantengewichte zwischen den Neuronen zufällig verteilt. Diese Kantengewichte sorgen dafür wie stark einzelne Neuronen in die nachfolgende Rechnung eingehen. Deshalb liefern untrainierte NN schlechte Ergebnisse und müssen trainiert werden, bevor sie Problemstellungen richtig lösen können.
Während des Trainings eines NN werden Kantengewichte angepasst um die Heuristik an ein optimales Ergebnis anzupassen.
Für das Training von NN wird viel Rechenleistung gebraucht, da der Prozess mit sehr vielen Rechenoperationen verbunden ist. [1]

Bild aufbau Neuronale Netze

2.2 Reinforcement Learning

Reinforcement Learning ist ein Bereich des maschinellen Lernens, bei welchem ein Agent durch Interaktion mit seiner Umgebung lernt, welche Aktionen in welchen Situationen am besten geeignet sind um ein bestimmtes Ziel zu erreichen. Da ein Agent keine konkrete Vorgehensweise besitzt, versucht er über Trial-and-Error herrauszufinden welche Aktionen zu Belohnungen führen.
Bestandteile des RL sind der Agent und die Umwelt. Der Agent bekommt die Informationen der Umwelt übergeben und entscheidet welche Handlungen daraus folgen sollen. Das Environment setzt die Aktionen des Agenten in Hanldungen um und bewertet diese mit numerischen Rewards, welche als Belohnung fungieren. Anhand der erhaltenen Rewards, versucht der Agent seine Aktionen anzupassen um die zukünftigen Belohnungen zu maximieren.
Ein RL-Agent nimmt seine Umgebung als eine Menge an bestimmten Zuständen war. Jeder Zustand enthält eine Vielzahl von Merkmalen und Eigenschaften welche für die Entscheidungsfindung relevant sein können. Als Zustandsraum wird die Menge aller möglichen Zustände bezeichnet.
Auf jedem dieser Zustände ist es möglich eine bestimmte Menge an Aktionen auszuführen, welche das Umfeld in einen anderen Zustand überführen. Die Menge aller möglichen Aktionen wird als Aktionsraum bezeichnet.
Wird eine Aktion auf einem bestimmten Zustand ausgeführt, so bezeichnet man die Zustandsübergänge bei Ausführung der Aktion als Zustandsübergangsfunktion.
Die Verhaltensstrategie πeines Agenten wird auch als Policy bezeichnet und liefert zu jedem Zustand eine Aktion. Die Policy wird im Verlauf des Trainings erlernt.
Das Erlernen einer Policy erfolgt durch Training des Neuronalen Netzes. Zu Beginn des Trainings werden Kantengewichte des zur trainierenden Neuronalen Netzes zufällig verteilt. Um zu überprüfen ob seine Aktionen gut oder schlecht sind, muss der Agent durch einen numerischen Rückgabewert die Information erhalten. Diese numerischen Rückgabewerte werden als Belohnungen oder auch Rewards bezeichnet. Belohnungen werden im Zustandsraum verteilt und können gutes Verhalten des Agenten durch positive Rewards und falsches Verhalten durch negative Rewards bestärken. Der Agent versucht den Wert der erhaltenen Belohnungen zu maximieren und erlernt auf diese Weise eine optimale Policy.
Probleme welche beim RL auftreten können sind unteranderem Überanpassung, Beloh-

nungsumgehung, Sparse Rewards. Wenn der Agent ein Problem gut in der Lernumgebung in welcher er trainiert wurde lösen kann allerdings nicht auf abweichenden Umgebungen spricht man von Überanpassung. Es tritt auf, wenn der Agent zu spezialisiert auf seine Trainignsumgebung ist und sich nicht an andere Situationen anpassen kann. Diese Problem lässt sich durch Generalisierung des Trainingsprozesses beheben. Ein Agent welcher auf vielen (zufällig) generierten Umgebungen trainiert, kann deutlich besser mit neuen Situationen umgehen, benötigt allerdigns auch mehr Zeit zum trainieren.
Belohnungsumgehung tritt auf, wenn der Agent lernt die Belohnungsfunktion zu manipulieren, um erhaltene Belohnungen zu maximieren, ohne das eigentliche Ziel der Aufgabe zu erreichen. Dies führt zu unerwünschtem Verhalten des Agenten. Verhindert werden kann dieses Problem, indem Belohnungen direkt mit dem Ziel verknüpft werden, also nur Belohnugnen ausgelöst werden, wenn tatsächlich positive Aktionen ausgeführt werden.
Von Sparse Rewards spricht man, wenn die Lernumgebung dem Agenten nur selten oder unregelmäßig Belohnungen vergibt. Dies führt dazu, dass der Agent Schwierigkeiten hat, die ihm gegebene Aufgabe richtig zu erlernen. Um dieses Problem zu umgehen, kann man zusätzliche künstliche Belohnungen einführen, welche dem Agenten auf den Weg zum richtigen Verhalten führen. Dies kann im Rückschluss jedoch wieder zur Belohnungsumgehung führen, wenn der Agent die Zwischenbelohnungen nutzt um maximale Belohnungen zu erhalten.
Auch ein zu großer Beobachtungsvektor kann zu Komplikationen führen. Wird dem Agenten eine Vielzahl an Observationen zugeführt, müssen erheblich mehr Parameter im Modell verarbeitet werden, was zu größerem Berechnungsaufwand führt. Dadurch wird auch die Lerngeschwindigkeit verlangsamt und der Bedarf an Speicherplatz für die Modelle steigt. So ist es möglich, dass einfach erscheinende Aufgaben mehrere Tage an Rechenzeit benötigen um ein gutes Modell zu generieren. [2]

2.2.1 Markov- Entscheidungsprozess

Jedes RL Problem lässt sich durch den Markov Entscheidungsprozess (= MDP) beschreiben. Der MDP ist ein mathematisches Modell für Entscheidungsprobleme, bei welchem der Agent abhängig von der Umwelt Entscheidungen trifft um ein bestimmtes Ziel zu erreichen. MDP setzen die Einhaltung der Markov-Eigenschaft vorraus. Diese ist erfüllt, wenn ein Zustandsübergang nur vom letzten Zustand und der letzten Aktion abhängig ist. Hauptkomponenten des MDP sind eine endliche Menge valider Zustände, eine endliche Menge valider Aktionen, Belohnungsfunktion und Verhaltensstrategie. Das Ziel eines MDP ist es eine optimale Policy, durch Maximierung der erhaltenen Belohnungen zu ermitteln.

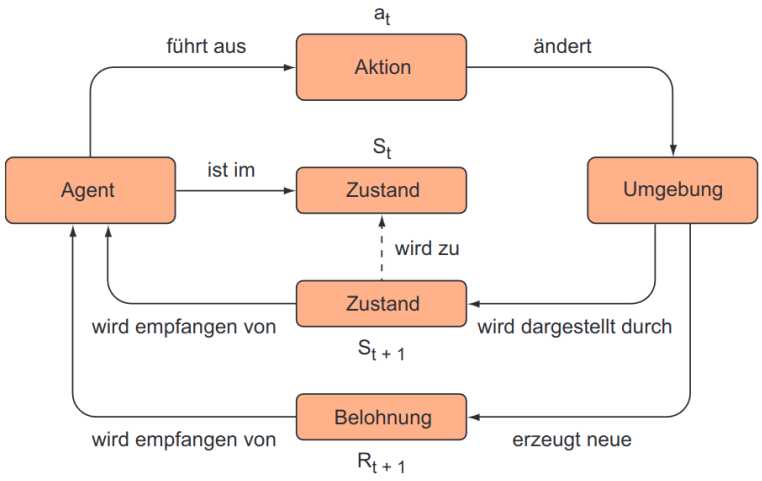

Abbildung 2.1: Ablauf eines MDP [2]

Abbildung2.1 stellt den Ablauf eines MDP dar. Zu Beginn einer Episode, befindet sich die Lernumgebung in einem gewissen Zustand. Dieser Zustand wird dem Agenten übergeben. Anhand der Beobachtungen führt der Agent eine Aktion aus welche die Umgebung verändert. Die Veränderung der Umgebung erzeugt einen neuen Zustand, welcher im nächsten Schritt wiederum dem Agenten übergeben wird. Durch auslösen von Aktionen in der Lernumgebung werden Belohnungen erzeugt welche dem Agenten einen numerischen Wert liefern, ob die Aktion gut oder schlecht war. Der Agent versucht die erhaltenen Belohnungen zu maximieren. Eine hohe Belohnung impliziert das richtige Verhalten zum Lösen des Problems welches der Agent bewähltigen muss. Dieser Prozess wiederholt sich bis das Problem gelöst wurde.

2.3 Das Würfelspiel: Noch Mal

Im modellierten Spiel 'Noch Mal!' geht es darum so viele Kästchen wie möglich anzukreuzen und damit viele Spalten und gleichfarbige Kästchen auszufüllen. Farb- und Zahlenwürfel müssen kombiniert werden um entsprechend zusammenhängende Felder der gewählten Farbe abzukreuzen. Im Spiel gibt es folgende Regeln:

1. Felder in der Spalte H sind von Beginn an verfügbar
2. Alle Kreuze müssen immer zusammenhängend in genau einem Farbblock der gewählten Farbe platziert werden
3. Kreuze müssen waagerecht oder senkrecht benachbart zu einem bereits abgekreuzten Feld oder Teil der Spalte H sein um verfügbar zu werden
4. Es müssen genau so viele Felder angekreuzt werden wie das Ergebnis des gewählten Zahlenwürfels
5. Es könnte nicht mehr als 5 Kästchen in einem Zug abgekreuzt werden
6. Wird ein Zahlenjoker gewählt, darf der Spieler eine Zahl von 1-5 bestimmen
7. Wird ein Farbjoker gewählt, darf der Spieler eine Farbe bestimmen

Um im Spiel 'Noch mal' eine möglichst hohe Anzahl an Punkten zu erhalten, ist es wichtig nach folgenden Strategien zu spielen:
Priorisierung äußerer Spalten: Äußere Spalten geben mehr Punkte, weshalb es wichtig ist diese komplett auszufüllen.
Beenden von Farben: Vollständig ausgefüllte Farben geben viele Extrapunkte. Im späten Spielverlauf kann es besser sein Farben komplett zu füllen anstatt Spalten zu werten.
Priorisieren von Sternfeldern: Jedes ausgefüllte Sternfeld gibt 2 Punkte, eine gute Spielweise ist es so viele Sternfelder wie möglich auszufüllen.
Strategische Nutzung von Jokern: Ungenutzte Joker geben zum Ende des Spiels Punkte. Es ist gut diese so wenig wie möglich zu nutzen um Extrapunkte zu bekommen. Jedoch können mit Hilfe von Jokern einfach bestimmte Felder gewählt werden, welche benötigt werden um eine Wertung zu erzielen.

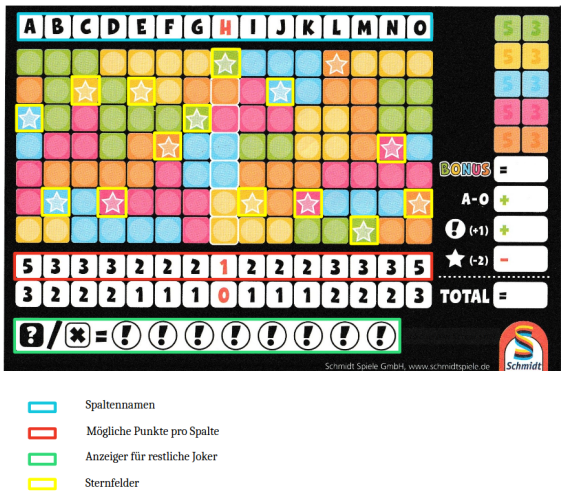

Abbildung 2.2: Vorlage des Spielfedes mit Indikation der Punktewertung [**Spielregeln**]

2.2 ist ein Spielfeld aus 'Noch mal!'. Die blau markierten Felder geben den Spaltennamen der Spalten an. Die rot markierten Felder zeigen an, wie viele Punkte beim ausfüllen der jeweiligen Spalte erzielt werden. Das grün markierte Feld zeigt die Anzahl der verbleibenden Joker an, wird ein Joker benutzt, muss eines der Felder abgestrichen werden. Zum Ende des Spiels erhält der Spieler Extrapunkte für die verbleibenden Joker. In jeder Spalte befindet sich ein geld markiertes Sternfeld. Diese geben zum Ende des Spiels Minuspunkte, weshalb es wichtig ist alle Sternfelder auszufüllen.

2.3.1 Spielablauf Einspieler Variante

Der Spieler Würfelt alle 4 Würfel bestehend aus 2 Farb- und 2 Zahlenwürfeln. Der Spieler hat 30 Züge Zeit maximale Punkte zu erreichen. Anschließend wählt er ein paar aus Farb- und Zahlenwürfel aus und kreuzt entsprechend des gewürfelten Paares verfügbare Felder auf dem Spielfeld ab. Ein Spieler darf immer entscheiden ob er Würfelwürfe zum ankreuzen verwenden möchte oder nicht. Um Kästchen anzukreuzen, wählt der Spieler eine Kombination aus Zahlen bzw Farbwürfel aus. Wählt er Beispielsweise 'Grün' und '2' so müssen 2 zusammenhängende Grüne Felder angekreuzt werden.
Gelingt es dem Spieler eine Spalte komplett auszufüllen, erhält er je nach Spalte Punkte dafür. Für äußere Spalten mehr werden mehr Punkte vergeben als für die inneren Spalten. Für das komplette Ausfüllen einer Farbe erhält der Spieler fünf Punkte pro ausgefüllter

Farbe. Jedes nicht angekreuzte Sternfeld gibt zum Spielende zwei Minuspunkte. Für jeden übrig gebliebenen Joker erhält der Spieler zum Ende des Spiels einen Punkt.

Abbildung 2.3: Grafik zur Bewertung der gesammelten Punkte der Einspieler Variante [**Spielregeln**]

2.3 ist aus der Spielanleitung des Spiels. Sie zeigt an, wie viele Punkte erreichbar sind und bewertet das Ergebnis. Anhand dieser erreichten Punkte wird die Güte des Trainingsfortschritts bewertet.

3 Konzeption

3.1 Anforderungsanalyse

In dieser Arbeit soll das Spiel 'Noch mal!' implementiert werden und von einem RL Agenten gespielt werden. Das Spiel muss nicht vom Nutzer selbst spielbar sein, sondern dem Agenten lediglich eine Lernumgebung bereitstellen in welcher er trainieren kann. Es müssen alle Funktionalitäten des Spiels abgedeckt werden. Weiterhin muss die Lernumgebung das Durchführen von illegalen Zügen unterbinden. Die Visualisierung des Spiels steht nicht im Vordergrund. Trotzdessen soll sie vorhanden sein, da sie ermöglicht Verhaltensweisen des Agenten besser Überwachen zu können. Um das Spiel zu programmieren, wird eine leistungsfähige Gameengine vorrausgesetzt, welche die Umsetzung vereinfacht. Da der Lernprozess des Agenten im Vordergrund steht, wird eine benutzerfreundliche Schnittstelle vorrausgesetzt, welche ein RL Framework bereit stellt und die Verwaltung von Modellen vereinfacht. Um das Training zu Überwachen und verschiedene Modelle miteinander zu vergleichen, wird ein Framework benötigt, welche den Trainingsprozess grafisch darstellt. Dieses soll ohne großen Mehraufwand nutzbar sein. Das Training von RL Modellen, benötigt viel Rechenzeit. Deshalb ist es nötig Traningsprozesse zu parallelisieren um die Dauer des Trainings zu minimieren. Die Parallelisierung sollte von der Entwicklungsumgebung bereitgestellt werden. Um verschiedene Trainingsszenarien zu erstellen und situativ einsetzen zu können, ist es notwendig mehrere Lernumgebungen konfigurieren und speichern zu können.

3.2 Auswahl der Verwendeten Technologien

Basierend auf der Anforderungsanalyse ergeben sich Anforderungen an die Technologien welche verwendet werden. Um das Spiel 'Noch mal!' zu programmieren wurde Unity als bevorzugte Gameengine gewählt. Unity stellt eine Vielzahl von Bibliotheken zur Verfügung und ermöglicht die unkomplizierte Umsetzung der Visualisierung des Spiels. C# ist die gängige Programmiersprache in Unity, deshalb wird das Projekt in C# umgesetzt.
Für die Erstellung des RL-Agenten wurde das ML-Agents Framework verwendet. Es bietet alle Funktionalitäten zum übergeben von Beobachtungen an den Agenten und Schnittstellen zum Ausführen von Aktionen im Lernumfeld. Weiterhn ist es möglich Aktionen mit Belohnungen zu Bewerten. Durch die Integration von ML Agents wird sichergestellt, dass der Agent den aktuellen Zustand des Spiels richtig übergeben bekommt und darauf reagieren kann.
Zu grafischen Darstellung des Trainingsprozesses wurde Tensorboard genutzt. Die Integration erfolgt ohne großen Mehraufwand und bietet die automatisierte Erstellung von Grafiken des Trainingsprozesses. Dies erleichtert die Evaluierung von verschiedenen Modellen.
Zur Visualisierung der erreichten Punkte der Agenten, wurde Python mit Matplotlib verwendet. Mit Matplotlib ist es möglich grafische Darstellungen von Daten zu erstellen. Mithilfe dieser Grafiken ist es möglich Rückschlüsse auf das Verhalten der Agenten zu führen beziegungsweise deren Trainingsfortschritte zu bewerten.
Mit der Verwendung jener genannten Technologien und Frameworks ist es möglich den Rahmen dieser Arbeit zu bearbeiten und zu bewerten.

3.3 Umsetzung in Unity

Der **Controller** stellt alle Funktionalitäten bereit, welche gebraucht werden um die Lernumgebung zu initialisieren. Er besitzt Prefabs des Agents, des GameFields und der Würfel und initialisiert diese zum Start. Weiterhin implementiert der Controller die Funktionalität der Punktevergabe, welche für eine Mehrspielervariante genutzt werden kann. Der Controller ist das Parent aller anderen Elemente und so ist er das zentrale Element der Steuerung. Auch das wiederholte Rollen der Würfel wird im Controller

angestoßen. Der Controller war sehr beim Erstellen paralleler Trainings, da dieser einfach mehrfach in die Szene aufgenommen werden musste um mehrere Spielfelder, welche gleichzeitig bespielt werden zu initialisieren.

<Code ausschnitt?>

Der **NumberDice** implementiert die Logik, welche für das Würfeln und Visualisieren der Zahlenwürfel benötigt wird. Die Visualisierung funktioniert mit selbst angefertigten Sprites welche in einem sortierten Array liegen und je nach gewürfelter Zahl initialisiert und gerendert werden. Beim wiederholten Würfeln, wird das initialisierte Sprite destroyed und ein neues erzeugt. Damit ist gewährleistet, dass immer das aktuelle Würfelergebnis angezeigt wird. Die Zahl des Würfels wird als Integer gespeichert, wobei er die Zahlen 1-6 annehmen kann. Die Zahl '6' entspricht dem Zahlenjoker.

<Code>

Wie der Zahlenwürfel implementiert der **ColorDice** die Funktionalität des Würfelns der Farben. Diese werden als String dargestellt und kann folgende Werte annehmen: {'blue', 'green','red', 'yellow', 'orange', 'joker'} Zur Visualisierung wird ein Sprite erstellt, was in der gewürfelten Farbe eingefärbt wird. Ein schwarzes Feld entspricht dem gewürfelten Farbjoker.

<Code>

Das **GameField** stellt das tatsächliche Spielfeld dar. Es implementiert die benötigten Methoden um die SquareFields zu verwalten und rückzusetzen. Außerdem wird die Anzahl der Joker in ihm gehalten.

Funktionalitäten:

- Visualisierung des Spielfeldes
- Aktualisieren der Gruppen aller Felder
- Abkreuzen der Felder
- Berechnen der validen Nachbarn der Felder
- Berechnen der verbleibenden Felder einer bestimmten Farbe

- Rückgabe der validen Felder für die aktuell gewählten Würfel.
- Reduzieren der verbleibenden Joker
- Rücksetzen der Felder um ein neuest Spiel zu Starten

Die **FieldSquares** stellen die einzelnen Teilfelder des Spielfeldes dar. In Tabelle 3.1 wird dargestellt welche Informationen gehalten werden.

Beschreibung	Typ	Wertebereich
Feld ist ein Sternfeld	Boolean	True / False
Farbe des Feldes	String	-
Feld ist ausgefüllt	Boolean	True / False
Feld ist verfügbar	Boolean	True / False
Clustergröße	Integer	1-6
X-Koordinate des Feldes	Integer	0-14
Y-Koordinate des Feldes	Integer	0-6

Tabelle 3.1: Übersicht Informationen der Fieldsquares

3.3.1 Visualisierung

Die Visualisierung des Spielfeldes erfolgt über ein angefertigtes Prefab. In diesem wurden die 105 Kästchen in einem Raster von 15x7 instanziiert und manuell mit den Informationen versehen. Dieses manuell angefertigte Spielfeld wurde als Prefab gespeichert und dient als Umgebung für den Agenten. Zu Beginn des Spiels, werden die Felder instanziiert **Verweis auf Code** in die Farben der hinterlegten Information in den richtigen Farben eingefärbt**Verweis auf Code**. Ausgefüllte Kästchen werden grau eingefärbt, diese Funktionalität wird im Fieldsquare Prefab ausgeführt.

3.4 Implementierung des Agenten

Der Agent ist die Schnittstelle zwischen dem Environment und dem RL. Dem Agent werden alle nötigen Informationen des Spielfeldes übergeben. Diese werden in ein Neuronales Netz übertragen, welches wiederum die Ausgabewerte in einem Vektor zurück an

den Agent leitet. Anschließend wird der Vektor verarbeitet und die gewählten Aktionen werden ausgeführt. Für gute Aktionen erhält der Agent positive Rewards, bei schlechten Aktionen wird der Zug übersprungen.
Zu Beginn jeder Episode, welche einem Spielzug entspricht, muss dem Agenten der aktuelle Zustand des Feldes übermittelt werden, aus welchem er die bestmögliche Option für einen Zug berechnet. In der ML Agents Bibliothek gibt es hierfür eine vorgefertigte Methode mit dem Namen CollectObservations. Diese erzeugt einen Observationsvektor 3.2 zu welchem die Informationen der aktuellen Zustands hinzugefügt werden. Wärend des Trainings eines Neuronalen Netzes, muss die größe des Vektors gleich bleiben. Das bedeutet es ist nicht ohne weiteres möglich ein Model auf unterschiedlichen Spielfeldern zu trainieren, da sich so die Anzahl der Beobachtungen unterscheiden würden. **verweis collect observations**

Aufbau der Beobachtungen:

Index	**Beschreibung**	**Type**	**Wertebereich**
0	Anzahl der verbleibenden Joker	Float	$[0-1]$
1	Anzahl der gespielten Runden	Float	$[0-1]$
2	Ergebnis des ersten Zahlenwürfels	Float	$[0-1]$
3	Ergebnis des zweiten Zahlenwürfels	Float	$[0-1]$
4-9	Ergebnis des ersten Farbwürfels	Vector6 (Binary)	$(0,1)^6$
10-15	Ergebnis des zweiten Farbwürfels	Vector6 (Binary)	$(0,1)^6$
16-24	Informationen für Feld 1	FeldVektor	-
25-33	Informationen für Feld 2	FeldVektor	-
...	...	...	...
953-961	Informationen für Feld 105	FeldVektor	-

Tabelle 3.2: Zusammenfassung der Observations und Feldinformationen

Stelle im Vektor	**Beschreibung**	**Type**	**Wertebereich**
$k*9+16$ - $k+21$	Farbe des Feldes k	Vector6 (Binary)	$(0,1)^6$
$k*9+22$	Ist Feld k verfügbar	Boolean	True / False
$k*9+23$	Ist Feld k abgestrichen	Boolean	True / False
$k*9+24$	Ist Feld k ein Sternfeld	Boolean	True / False

Tabelle 3.3: Observation jedes einzelnen Feldes

Anhand der Observations berechnet das Neuronale Netz einen Ausgabevektor. Tabelle 3.4 zeigt den Aufbau der hier Verwendeten Beobachtungen. Mit diesem führt der Agent nun

bestimmte Aktionen aus und versucht sein Ergebnis (Rewards) zu maximieren. Anhand der gesammelten Rewards wird das Neuronale Netz nun angepasst um das bestmögliche Ergebnis zu erreichen.

Tabelle 3.4: Index und Beschreibung der Variablen

Index	**Beschreibung**	**Typ**	**Wertebereich**
1	Index des gewählten ZahlenWürfels	Integer	0-1
2	Index des gewählten Farbwürfels	Integer	0-1
3	Jokerzahl	Integer	0-4
4	X-Koordinate des gewählten Feldes	Integer	0-14
5	Y-Koordinate des gewählten Feldes	Integer	0-7
6	Action 1 für die Auswahl der Nachbarn	Continuous	-
7	Action 2 für die Auswahl der Nachbarn	Continuous	-
8	Action 3 für die Auswahl der Nachbarn	Continuous	-
9	Action 4 für die Auswahl der Nachbarn	Continuous	-

Das vergeben von Rewards lehnt sich an die Punktevergabe im Spiel an. Der Agent erhält Belohnungen wenn er auch Punkte im gespielten Spiel erlangen würde. In reftab:rewards ist eine Übersicht der Belohnungen. Daraus lässt sich ablesen welche Aktion welchen Reward auslöst.

Aktion	**Erhaltene Belohnung**
Abkreuzen eines Sternfeldes	2
Ausfüllen einer kompletten Spalte	1-5 abhängig der Spalte
Ausfüllen einer kompletten Farbe	5
Verbleibende Joker zum Ende des Spiels	1 / verbleibendem Joker

Tabelle 3.5: Belohnungen für bestimmte Aktionen

3.4.1 Erklärung des Alghorithmus

Im folgenden wird der Ablauf zum Wählen der Felder erläutert. Im Beispiel wird der Ausgabevektor **(1 , 1 , 0 , 3 , 4 , 0.6 , 0.5 , 0.4 , 0.8)** verwendet.
Die ersten beiden Stellen des Ausgabevektors entsprechen den gewählten Würfeln. Im ersten Schritt werden alle Felder des Spielfedes untersucht, ob sie ein valides Ziel für das gewürfelte Ergebnis bilden. Dies ergibt sich aus der Gruppe der Spielfelder, der Farbe und ob das Kästchen verfügbar ist. Valide Felder werden in eine Liste (availableFields) aus Verfügbaren Feldern geschrieben. Abbildung 3.1 zeigt den beschriebenen Zustand des Feldes.

Abbildung 3.2 bildet den nächsten Schritt im Algorhitmus ab. Es wird geprüft, ob die gewählten Koordinaten in availableFields vorhanden sind. Wenn kein Feld verfügbar ist, wird die Episode abgebrochen und der Agent überspringt seinen Zug. Sofern der Agent ein valides Feld gewählt hat, wird dieses in eine weitere Liste (pickedFields) geschrieben und benachbarte Felder der selben Gruppe werden zurückgegeben.

Im nächsten Schritt wird jedem der verfügbaren Nachbarn abhängig der Gesamtanzahl ein Wertebereich zwischen 0 und 1 zugewiesen, dies wird in 3.3 und 3.6 verdeutlicht.. Anhand des discreten Wertes des Ausgabevektors wird das Zugehörige Feld in pickedFields geschrieben.

FeldKoordinaten	**von**	**bis**
(3,5)	0	0.33
(4,4)	0.33	0.66
(3,3)	0.66	0.99

Tabelle 3.6: Bereiche für bestimmte Felder

Für alle Felder in PickedFields werden die benachbarten Felder zurückgegeben und der vorherige Schritt wiederholt. Wenn so viele Felder gewählt wurden, wie erwürfelt wurden, werden die Felder anschließend ausgefüllt und auf Rewards überprüft. Wie Abbildung 3.4 verdeutlicht.

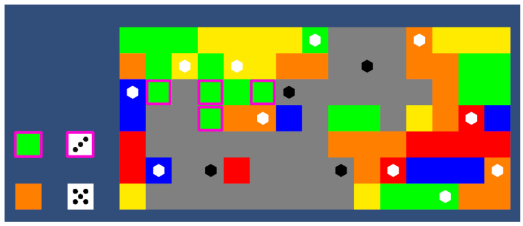

Abbildung 3.1: Valide Felder für das gewählte Würfelergebnis wurden markiert

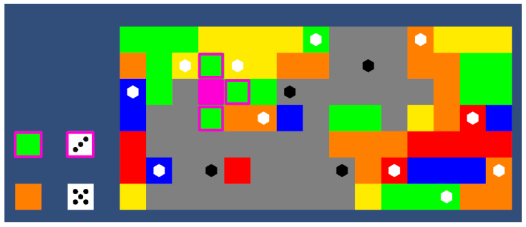

Abbildung 3.2: Feld(3,4) wird in die pickedField Liste aufgenommen und benachbarte Felder werden zurückgegeben.

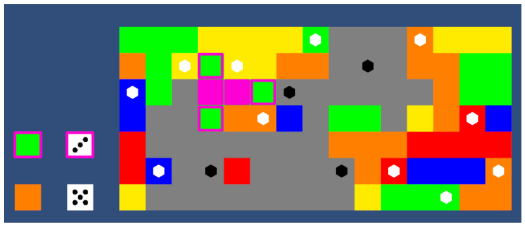

Abbildung 3.3: Feld(4,4) ist das nächste gewählte Feld und wird in pickedFields aufgenommen

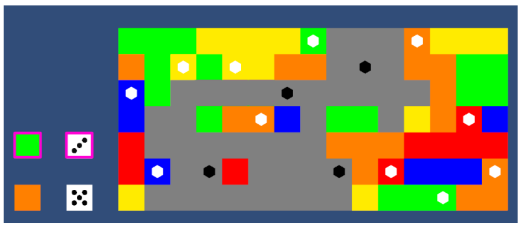

Abbildung 3.4: Felder wurden gewählt und ausgefüllt

4 Präsentation der Ergebnisse

4.1 Trainingsversuche

4.1.1 Agent ohne Spielreglementierung

Im ersten Schritt überprüfte ich den Agenten, ob er die Spielregeln selbstständig über Belohnungen erlernen kann. In diesem Versuch sollte der Agent Felder nach Index auswählen. Jedes Kästchen im Spielfeld besaß einen Index zwischen 0-104. Je nach Höhe der gewürfelten Zahl, wurden ebenso viele Feldindizes überprüft. In 4.1 ist der Aufbau des Actionbuffers dargestellt. Nach der Auswahl aller potenziell abzukreuzenden Feldern wurde überprüft ob der Zug legal ist. Mindestens ein Feld musste verfügbar sein, alle Felder mussten benachbart und der selben Farbe sein. Illegale Züge zogen negative Rewards mit sich, legal ausgeführte Züge dagegen positive.
Diese herangehensweise führte nicht zum gewünschten Ergebnis. Auch nach einigen Stunden des Trainings, konnte der Agent nur sehr selten legale Züge durchführen. Dem Agent war es nicht möglich in der begrenzten Trainignszeit den Zusammenhang der Observations zu den gegebenen Rewards festzustellen. Dennoch ist der Ansatz nicht gänzlich Falsch. Mit einem hohen Rechenaufwand, könnte der Agent auch die Spielregeln erlernen, es würde nur sehr viel Zeit kosten.

Index	Bezeichnung	Datentyp	Wertebereich
1	Index des zu wählenden Farbwürfels	int	0-1
2	Index des zu wählenden Zahlenwürfels	int	0-1
3	Feldindex	int	0-104
4	Feldindex	int	0-104
5	Feldindex	int	0-104
6	Feldindex	int	0-104
7	Feldindex	int	0-104

Tabelle 4.1: Aufbau Actionbuffer

4.1.2 Agent mit zusätzlichen Belohnungen

In diesem Versuch bekam der Agent zusätzlich zu den Belohnungen welche den Punkten entsprechen Punkte für Aktionen. Diese Belohnungen sollten dazu dienen schneller zu einer optimalen Policy zu gelangen. In reftab:rewards2 sind alle zusätzlichen Rewards ersichtlich. Der Versuch wurde über 2.4Mio Spielzüge ausgeführt. Dieser Zeitraum ist wie sich im Verlauf der folgenden Experimente herrausstellt relativ kurz. Da sich jedoch ein negatives Ergebnis abzeichnete wurde auf längeres Training verzichtet.

Aktion	Erhaltene Belohnung
Abkreuzen von Feldern	0.02f pro Feld
Abkreuzen eines gesamten Clusters	0.04f pro Feld
Wahl eines Würfelpaars ohne legale Züge	-50.0f
Wahl eines Jokers ohne verfügbare Joker	-50.0f

Tabelle 4.2: Zusatzbelohnungen für verschiedene Aktionen

Die Grafiken 4.1 und 4.2 zeigen deutlich, dass je länger das Training voranschritt, die durchschnittlich erreichten Punkte pro Spiel sanken. Die zusätzlichen Belohnungen führten dazu, dass der Agent nicht versuchte Spalten abzukreuzen oder Farben komplett auszufüllen, da er das Abkreuzen von Clustern für deutlich effizienter hielt um seine Belohnungen zu maximieren. Obwohl die Belohnungen für Erreichte Spalten oder komplett ausgefüllte Farben mehr Punkte ergaben.

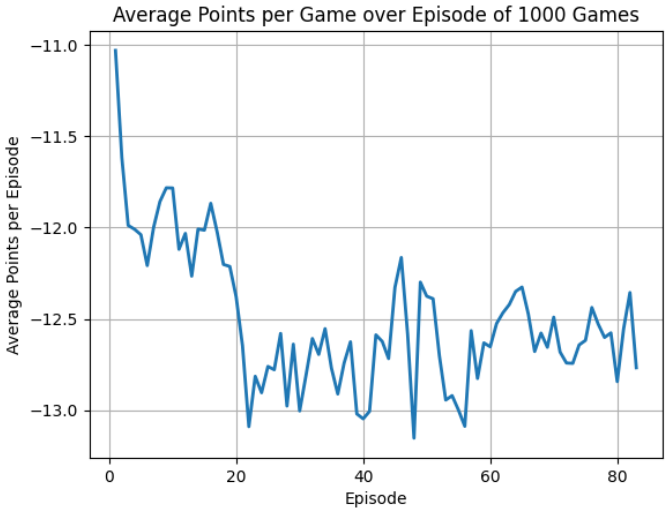

Abbildung 4.1: Durchschnittliche Punkte des Agenten mit Zusatzbelohnungen

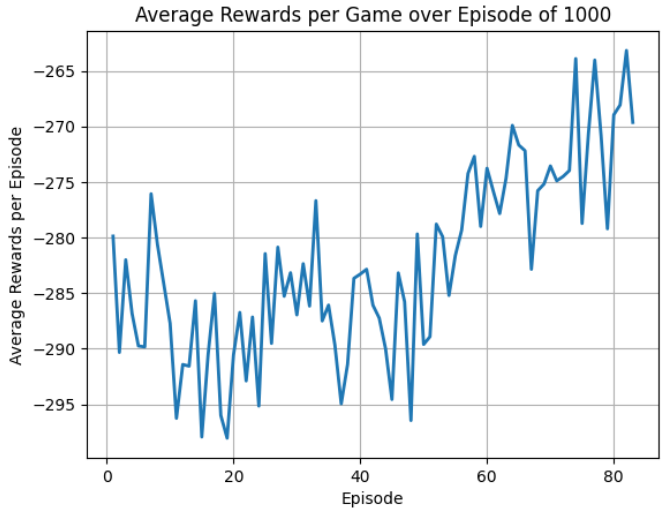

Abbildung 4.2: Durchschnittliche Belohnungen des Agenten mit Zusatzbelohnungen

4.1.3 Trainiert vs Untrainiert

In diesem Experiment, werden die erreichten Punkte und Rewards eines untrainierten Agenten gegenüber den erzielten Ergebnissen eines trainierten Agenten gegenübergestellt. Der trainierte Agent hat bereichts 25 Mio. Spielzüge absolviert, was ungefähr 830k gespielten Spielen entspricht. Der untrainierte Agent bekommt ein neu initialisiertes NN, welches zufällig gewählte Kantengewichte zwischen den Neuronen erhält. Wie an den Grafiken 4.4 und 4.3 zu erkennen ist, hat der trainierte Agent tatsächlich einen höheren

Durchschnitt an erzielten Punkten pro Spiel. Auch die gesammlten Rewards sind bei dem trainierten Agenten höher. Dies liegt daran, dass die Rewards so festgelegt sind, dass der Agent sie nur erhält, wenn er auch im Spiel punktet. Schon während des Trainings war ein merklicher Unterschied festzustellen, deshalb war das Ergebnis dieses Experiments zu erwarten.

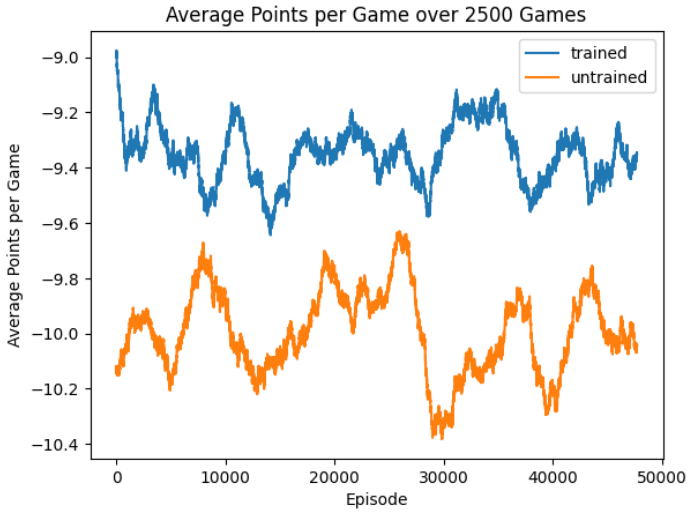

Abbildung 4.3: Durchschnitt der erreichten Punkte pro Spiel

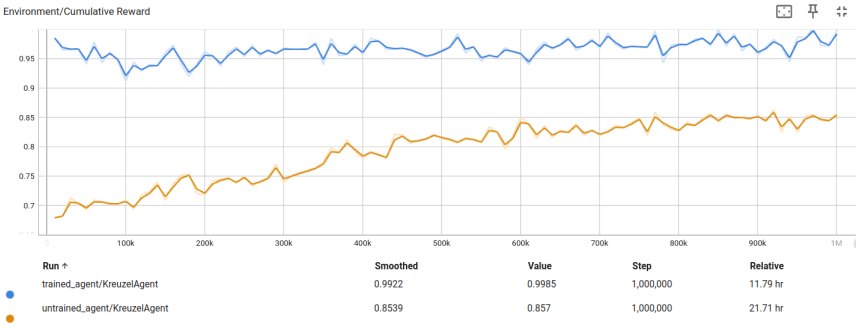

Abbildung 4.4: Übersicht der gesammelten Belohnungen

4.1.4 Trainiert vs Training mit Sonderfeldern

In diesem Experiment wurden die erreichten Puntke und Rewards des trainierten Agenten gegenüber einem Agenten, welcher mit Sonderfeldern trainiert wurde gegenüber gestellt. 4.5 stellt den Aufbau der Lernumgebung für das Training dar. Diese speziellen Felder waren einheitlich in die verschiedenen Farben eingefärbt bzw jedes Feld wurde mit Sternfeldern versehen. Dies sollte dazu führen, dass der Agent besser zuweisen kann welche Stellen im Observationsvektor für welche Information zuständig sind.

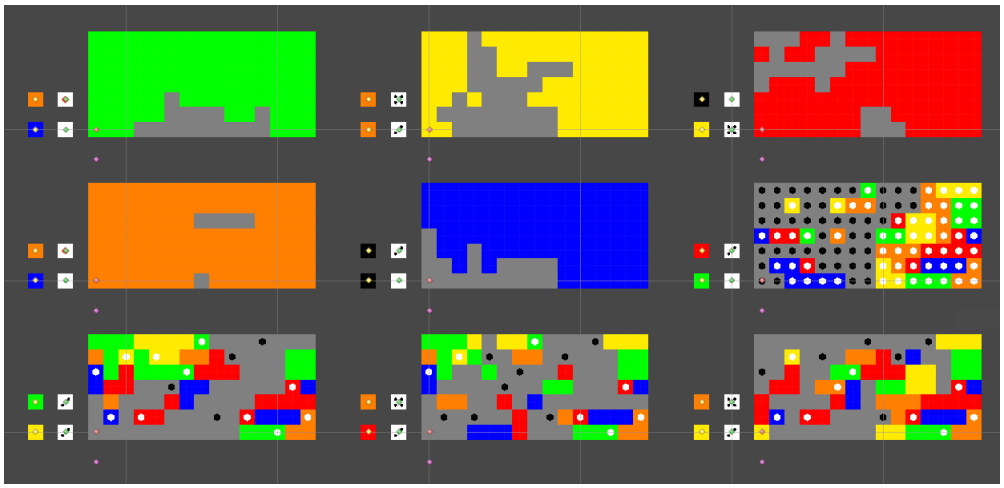

Abbildung 4.5: Übersicht der speziellen Felder

In der Grafik 4.5 ist das Training mit den speziellen Feldern dargestellt. Jedes der Feld hat gewisse Besonderheiten, welche sich zu den normalen Spielfeldern abgrenzen. Fünf Felder sind in einer kompletten Farbidentität eingefärbt. Hierbei wurde der Zahlenwürfel manipuliert **<verweis auf Würfelshift>** um häufiger die entsprechende Farbe zu werfen. Diese Felder sollten dem Agenten besser den Zusammenhang des Farbwürfels und des gewählten Farbidentität der Kästchen näherbringen. In dem anderen Spielfeld ist jedes Feld als Sternfeld markiert. Dies sollte dem Agenten zeigen, dass jedes Feld mit markierten Sternen mehr Punkte bringt. Bei den anderen drei Feldern ist jedes Feld von vorn herein als verfügbar markiert. Dies sollte zum einen das Konzept des verfügbaren Feldes vermitteln zum anderen dem Agenten ermöglichen das Feld weiter als normal zu explorieren, um die komplexen Ziele des Spiels leichter zu erreichen. Das Training mit speziellen Feldern führte zu einer Verschlechterung des Ergebnisses wie die nachfolgenden Grafiken zeigen.

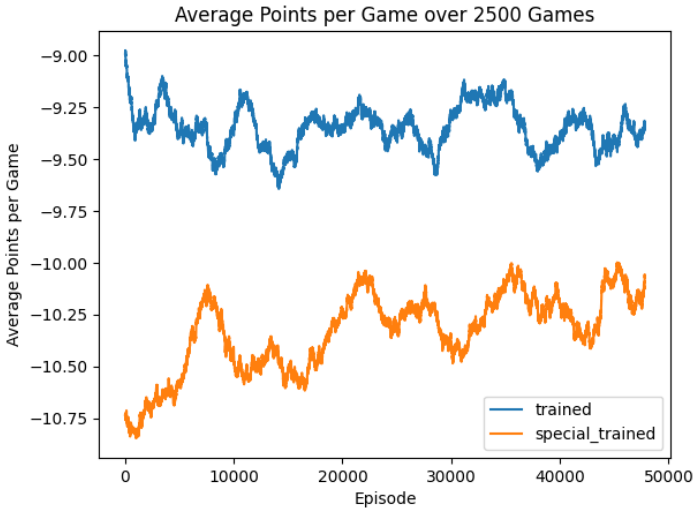

Abbildung 4.6: Durchschnitt der erreichten Punkte beider Agenten

Aus der Grafik 4.6 lässt sich ableiten, dass das Modell welches mit speziellen Feldern trainiert wurde im Durchschnitt weniger Punkte erhalten hat als der normal trainierte Agent. Dieses Training führte nicht zu einer Verbesserung des Modells. Ursache hierfür liegt sicher im Spielffeld in welchem alle Felder als Sternfelder markiert wurden. Hier konnte der Agent willkürlich Züge ausführen und bekam überdurchschnittlich viele Punkte. Deshalb priorisierte der Agent nicht mehr die eigentlichen Ziele, was wiederum zur Folge hatte, dass die Leistung des Agenten auf dem eigentlichen Feld schlechter wurde.

4.1.5 Trainiert vs Training mit mehr Spielzügen

In diesem Experiment trainierte der Agent mit mehr zur Verfügung stehenden Spielzügen. Dadurch konnte der Agent das Feld besser explorieren und insgesamt mehr Aktionen auslösen welche zu Belohnungen führten. Dies hat zur Folge, dass auch schwierig erreichbare Rewards ausgelöst wurden, welche somit vom Agent erlernt werden konnten. Die Grafik 4.7 zeigt, dass das Experiment eine Verbesserung des Models zur Folge hatte. Da im Durchschnitt mehr Punkte erreicht wurden.

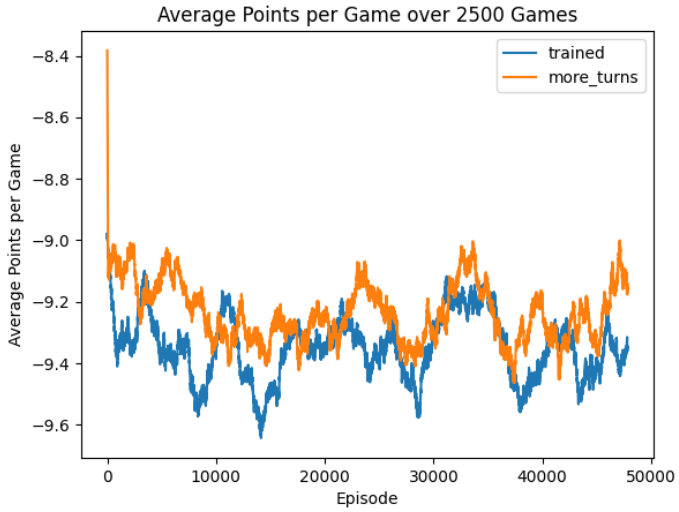

Abbildung 4.7: Vergleich 'Mehr Züge' und 'trainierter Agent'

4.1.6 Überprüfung auf Overfitting

In diesem Experiment, sollte der Agent auf Overfitting überprüft werden. trainierte und untrainierter Agent spielten das Spiel nach normalen Spielregeln auf einem anderen Spielfeld. In den Grafiken 4.8 und 4.9 ist erkennbar, dass beide Agenten ungefähr die selben Rewards gesammelten haben. Der untrainierte Agent konnte im Durchschnitt jedoch etwas mehr Punkte sammeln. Dies schließt darauf, dass der Agent tatsächlich nur auf dem im Training verwendeten Spielfeld gut performen kann und neue Spielfelder erst erlernen muss. Interessant ist weiterhin, dass der Durchschnitt aller Punkte etwa 2 Punkte über dem des anderen Spielfeldes liegt, was auf eine höhere Schwierigkeit des schwarzen Spielfeldes hinweist.

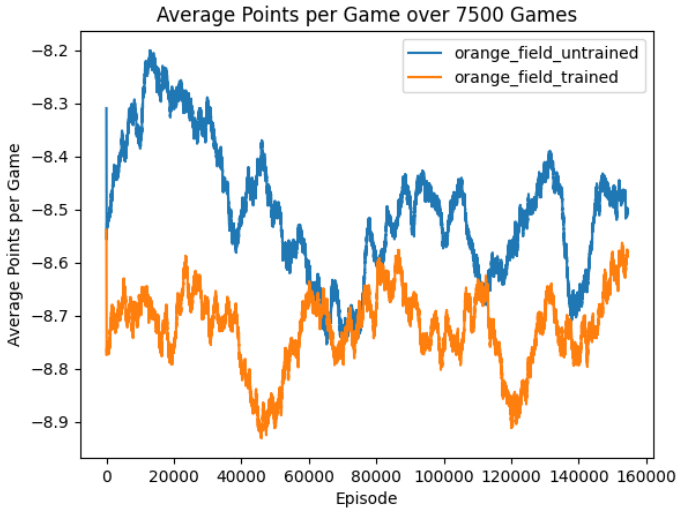

Abbildung 4.8: Vergleich Punkte 'trainiert' und 'untrainiert' auf orangen Spielfeld

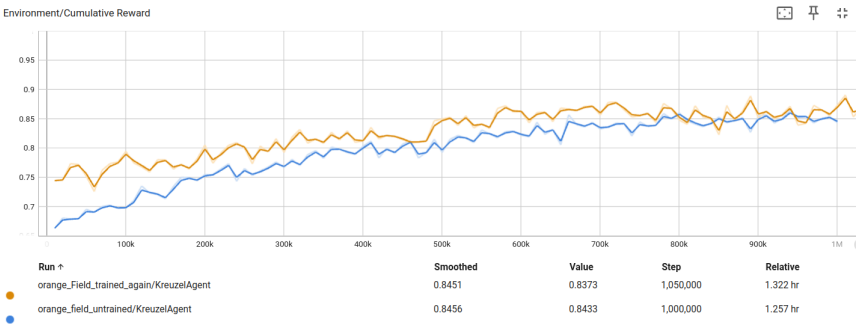

Abbildung 4.9: Übersicht gesammelte Rewards auf orangen Spielfeld

4.1.7 Training auf Minifeld

Um zu Überprüfen, ob ein kleineres Feld **Verweis auf Minifeld.png** einen positiven Effekt auf das Training hat, habe ich bei einem Spielfeld 3 Zeilen abgeschnitten und ließ einen Agenten darauf trainieren. Im Anschluss überprüfte ich die Leistung des Agenten auf dem normalen Spielfeld gegenüber einem untrainierten Agenten. Da die Observations von Modellen gleich bleiben müssen, entscheid ich mich nicht mehr vorhandene Kästchen mit Nullen im Vektor zu präsentieren. **Verweis auf auffüllen von leeren feldern**

Die Grafiken 4.10 und 4.11 zeigen, den Durchschnitt der erreichten Punkte pro Spiel und die gesammelten Blohnungen während des Trainings. Es wird ersichtlich, dass der mit einem kleinen Feld vortrainierte Agent schlechter performt, als die beiden anderen.

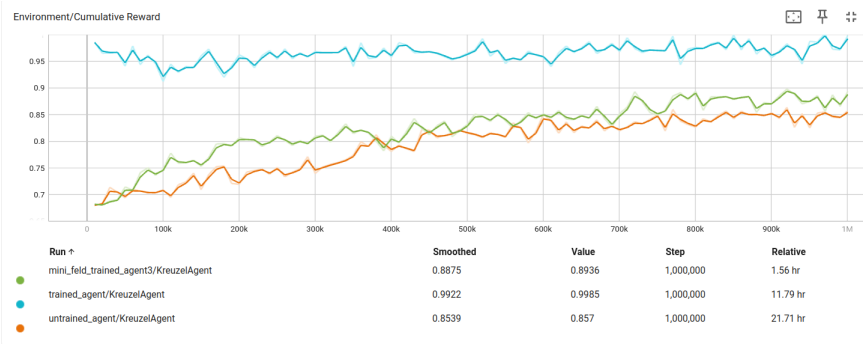

Abbildung 4.10: Gesammelte Belohnungen mit Minifeld

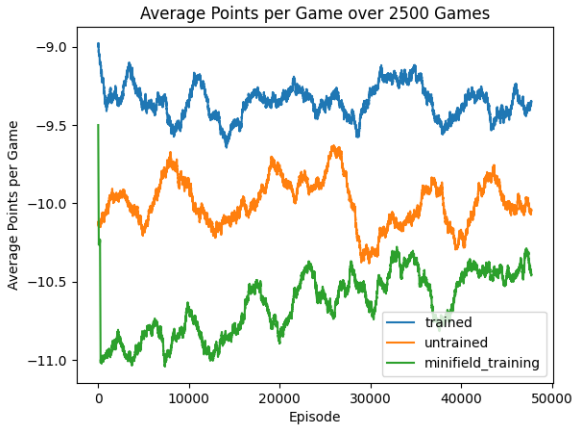

Abbildung 4.11: Vergleich erreichte Punkte

Belohnungen wurden auch hier nur verteilt, wenn es zur Punktewertung kommt. Deshalb ist es interessant, dass der untrainierte Agent mehr im Durchschnitt Punkte bekommt, als der auf dem kleinen Feld vortrainierte Agent, obwohl dieser wiederum einen höheren Durchschnitt an Belohnungen erhält.

4.1.8 Training Auswahl KoordinatenPicker

Da der Agent keine großen Fortschritte erzielen konnte, entschied ich den Agenten das erste Feld durch Koordinaten zu wählen. Dies setzte Vorraus, dass die Koordinaten der einzelnen Felder in die Observations mit aufgenommen werdne musste und die Observations noch größer wurden. Damit der Agent lernen kann, welche Koordinaten zu welchen Feldern gehören, entschied ich mich dazu ihn auf einem Spielfeld trainieren zu lassen, wo alle

Teilfelder verfügbar sind. Rewards wurden vergeben für Valide ausgewählte Felder, in Abhängigkeit der gewürfelten Zahlen. Im nächsten Schritt wird dieses vortrainierte NN genutzt um das Spiel mit richtigen Regeln zu spielen.

4.1.9 Training blinder Agent

In diesem Experiment bekam der Agent lediglich die Würfel, die Anzahl der verbleibenden Joker und die aktuelle Runde des Spiels übergeben. Dieser Versuch sollte überprüfen, wie gut ein Agent der das aktuelle Spielfeld nicht sieht perfomt. Da die Auswahl der Felder durch mehr oder weniger zufällige Interpolation aller möglichen Felder abläuft, kann der Agent dennoch normal spielen. Durch den Versuchaufbau veringert sich der Beobachtungsvektor von 916 auf eine Größe von 15 Informationen. Dies führte dazu, dass der Agent schnell zu seiner optimalen Policy gelangen konnte. Auch wenn der Agent das Spielfeld nicht sieht, kann er dieses implizit erlernen. Dieser Prozess wäre allerdings nicht sonderlich robust, wäre sehr Lernintensiv und würde nicht auf anderen Feldern funktionieren. Der Agent konnte bereits nach sehr kurzer Zeit von etwa 400k Lernschritten gegen sein Maximum konvergieren, wie in 4.12 ersichtlich ist. Dort schneiden sich die beiden grünen Graphen und verbleiben auf ungefähr dem selben Niveau. Dieses Modell wurde insgesamt 5 Mio. Episoden trainiert. Der trainierte Agent konnte sich dagegen kontinuierlich minimal Verbessern.
Abbildung 4.13 zeigt, dass auch der blinde Agent, im Durchschnitt weniger Punkte erreichen konnte. Dies beweist, dass das Training trotz der geringen erreichten Punkte positiv verlaufen ist.

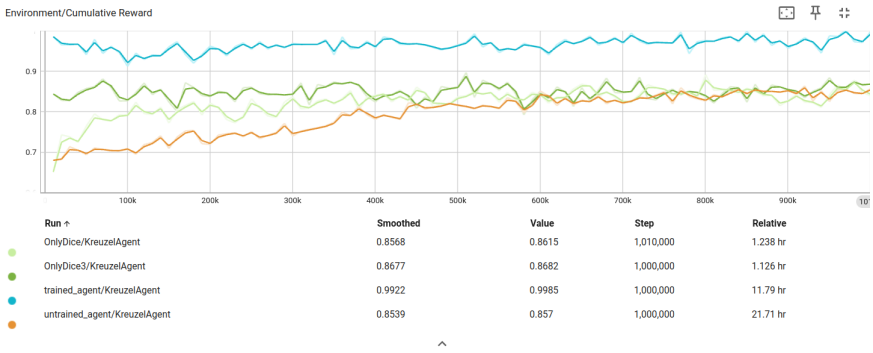

Abbildung 4.12: Gesammelte Belohnungen blinder Agent

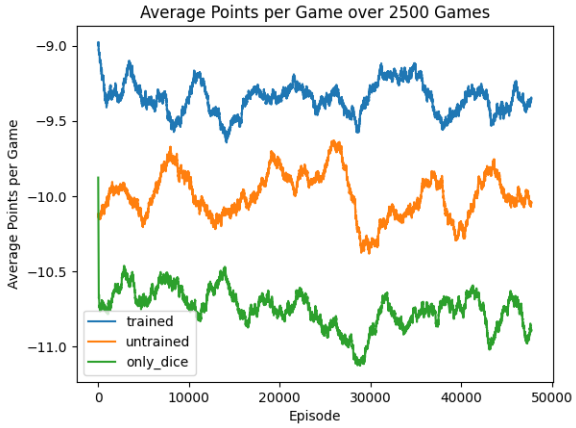

Abbildung 4.13: Vergleich erreichte Punkte blinder Agent

5 Auswertung und Ausblick

5.1 Bewertung der Ergebnisse

Die Experimente aus dem vorigen Kapiten zeigen deutlich, dass der Trainignsfortschritt des Agenten vorhanden war. Der trainierte Agent erhielt deutlich mehr Belohnungen als die untrainierten oder mit speziellen Lernumgebungen trainierten Agenten. Leider lässt sic hdieser Fortschritt auf die Metrik der durchschnittlich erreichten Punkte anwenden. Wünschenswert wäre ein Modell gewesen, welches im Durchschnitt positive Spielergebnisse sammelt. De mAgenten gelang es nicht kontinuierlich nac heiner Strategie zu spielen welche viele Punkte nach sic hzieht. So konnte er fast niemals eine Farbe komplett ausfüllen, was viele Punkte bringen würde. Die liegt vorallem daran, dass er dieses für RL Agenten schwere Ziel nur sehr selten erreichte und somit auch nicht erlernen konnte. Das Ergebnis auf einem anderen Spielfeld war für den Tranierten Agenten überraschend schlecht. Dies spricht dafür, dass das Training auf einem einzelnen Spielfeld nicht förderlich ist für einen Agenten der auf verscheidene Situationen gut agieren soll.
Während des Verlaufs dieses Bachelorarbeit musste ich feststellen, dass das Problem ein wenig kompliziertes Spiel wie 'Noch mal!' für einen RL Agenten deutlich komplexer ist als erwartet. Mein Agent musste eine Vielzahl von von Daten verarbeiten und diese semantisch Zuordnen. Dies führte zu eines immensen Dauer des Trainingsprozesses. Der Agent welcher 25 Mio Trainingsschritte durchlaufen hatte, brauchte für diesen Prozess etwa 30 Stunden. Während dieser Zeit konnte das Modell nur minimale Verbesserungen auf zeigen. Weiterhin kommt hinzu, dass das modelierte Spiel zum großen Teil ein Glückspiel ist. Es kann vorkommen, dass der Spieler einfach Pech im Würfeln hat und so trotz perfekter Spielweise ein schlechtes Spielergebnis erreicht. Deshalb ist es auc hnicht möglich ein Modell zu erzeugen, was immer ein Perfektes Ergebnis liefert. Allerdings wäre mit weiterer Rechenzeit und vermehrter Nutzung von verschieden Trainingsarten

noch eine weitere Verbesserung des Modells möglich.

5.2 Schritte zur Verbesserung des Agenten

Mit mehr Rechenleistung und Trainignszeit könnte der Agent durchaus noch zu besseren Ergebnissen kommen. Ein Flaschenhals war unter anderem, dass jede Trainingsiteration von 1 Mio. Schritte ca. 1,5 Stunden dauerte. Falls es zu Fehlern während des Trainings kam, musste die aufgewendete Zeit erneut investiert werden. Weiterhin konnte sich der Agent, wenn auc hnur sehr lkangsam, stetig Verbessern. Mit einer deutlich längeren Trainignsdauer und mehr Rechenleistung kann das Modell deutlich Verbessert werden.
Ein kompletter Verzicht der Visualisierung während des Trainings kann zu einer Verbesserung der Trainignsdauer führen. Es würden mehr Ressourcen zur Berechnung des Neuronalen Netzes zur Verfügung stehen und damit die Dauer verkürzen. Weiterhin kam es insbesondere während langen Trainingsepisoden zu Crashes, welche durch Unity hervorgerufen wurden. Dies könnte durch eine Performantere Lernumgebung substituiert werden um einen robusteren Ablauf zu erschaffen.
Umsetzen in C direktes übergeben an pytorch? weniger probleme more power?
Ein weiteres Problem des trainierten Modells war die Überanpassung. Um dies zu umgehen und einen Modell zu trainieren welches auf verschiedene Zustände des Spielfeldes optimal reagieren zu können, wäre es hilfreich, das Spielfeld für jede Spielrunde zufällig generisch zu erzeugen. Dies würde dazu führen, dass der Agent nicht ein einzelnes Spielfeld erlernt, sondern tatsächlich erlernt das Spiel zu spielen. Generisch erzeugte Trainingsumgebungen würden die Trainingsdauer erhöhen dafür aber in ein robusteres Modell resultieren.
Anpassen der Hyperparameters des Agenten können zu einer optimierung des Trainings führen. Leider ist das Anpassen der Hyperparameter nicht sonderlich intuitiv und nachvollziehbar, weshalb etwas Erfahrung im Umgang mit RL erforderlich ist. Durch geschicktes Verändern dieser Parameter lassen sich lokale axima überspringen oder die Exploration des Agenten beschleunigen.
Eine weitere Verbesserung des Trainignsprozessen könnte auch eine ausgedehnte Selection nach Vorbild der Evolutionären Allgorhitmen liefern. Die hätte einen höheren Aufwand der Trainings zur Folge, könnte aber dazu führen, dass sich gute Modelle durchsetzen und so eine optimale Spielstrategie etablieren.

Literaturverzeichnis

[1] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. de. Computational Intelligence. Wiesbaden: Springer Fachmedien Wiesbaden, 2021. ISBN: 978-3-658-32074-4 978-3-658-32075-1. DOI: 10.1007/978-3-658-32075-1. URL: https://link.springer.com/10.1007/978-3-658-32075-1 (besucht am 09. 02. 2024).

[2] Alex Zai und Brandon Brown. *Einstieg in Deep Reinforcement Learning: KI-Agenten mit Python und PyTorch programmieren*. en. München: Hanser, 2020. ISBN: 978-3-446-45900-7.

[3] Richard S. Sutton und Andrew Barto. *Reinforcement learning: an introduction*. en. Nachdruck. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2014. ISBN: 978-0-262-19398-6.

[4] John Schulman u. a. *Proximal Policy Optimization Algorithms*. en. arXiv:1707.06347 [cs]. Aug. 2017. URL: http://arxiv.org/abs/1707.06347 (besucht am 26. 01. 2024).

[5] Jonathan Hui. *AlphaGo: How it works technically?* en. Mai 2018. URL: https://jonathan-hui.medium.com/alphago-how-it-works-technically-26ddcc085319 (besucht am 07. 02. 2024).

[6] Manfred Broy. *Logische und Methodische Grundlagen der Programm- und Systementwicklung: Datenstrukturen, funktionale, sequenzielle und objektorientierte Programmierung - Unter Mitarbeit von Alexander Malkis*. de. Wiesbaden: Springer Fachmedien Wiesbaden, 2019. ISBN: 978-3-658-26301-0 978-3-658-26302-7. DOI: 10.1007/978-3-658-26302-7. URL: http://link.springer.com/10.1007/978-3-658-26302-7 (besucht am 08. 02. 2024).

[7] Oliver Kramer. *Computational Intelligence: Eine Einführung*. de. Informatik im Fokus. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-540-79738-8 978-3-540-79739-5. DOI: 10.1007/978-3-540-79739-5. URL: https://link.springer.com/10.1007/978-3-540-79739-5 (besucht am 09. 02. 2024).

[8] *Using TensorBoard to Observe Training - Unity ML-Agents Toolkit*. URL: https://unity-technologies.github.io/ml-agents/Using-Tensorboard/ (besucht am 09. 02. 2024).

[9] Yinlong Yuan u. a. „A novel multi-step reinforcement learning method for solving reward hacking“. en. In: *Applied Intelligence* 49.8 (Aug. 2019), S. 2874–2888. ISSN: 0924-669X, 1573-7497. DOI: 10.1007/s10489-019-01417-4. URL: http://link.springer.com/10.1007/s10489-019-01417-4 (besucht am 11. 02. 2024).

[10] Schmidt Spiele GmbH. *Spielregeln 'Noch mal!'* URL: https://www.schmidtspiele.de/files/Produkte/4/49327%20-%20Noch%20mal!/49327_Noch_Mal_DE.pdf.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis