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 Trainingsstrategien im Reinforcement Learning anhand eines Würfelspiels in Unity

— —

vorgelegt von: Tony Lenz
Geburtsort und -datum: Leipzig, den 07.01.1993
Abgabe: Leipzig, den 23. März 2024

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

HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die an der Hochschule für Technik, Wirtschaft und Kultur Leipzig, konkret an der Fakultät Informatik und Medien (FIM), eingereichte Arbeit zum Thema Evaluierung von Trainingsstrategien im Reinforcement Learning anhand eines Würfelspiels in Unity selbstständig, ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht. Die Abbildungen in dieser Arbeit wurden von mir selbst erstellt oder mit einem entsprechenden Hinweis auf die Quelle versehen.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

. . . . . .

Unterschrift Tony Lenz
Leipzig, den 23. März 2024

Kurzfassung

Die Kurzfassung - präziser: Abstrakt, sollte auf eine Seite beschränkt sein. Der Abstrakt sollte das Ziel der Arbeit explizit beschreiben, die angewandten *Methoden* aufführen, die wichtigsten *Ergebnisse* aufzählen und die *Hauptschlussfolgerungen* aufzeigen.

Einen Weg zu finden, hunderte von Seiten an Informationen in wenigen Sätzen zusammenzufassen ist eine Herausforderung. Jedes Wort muss sorgfältig abgewogen werden (Gibt es eine bessere, prägnantere Möglichkeit, die Hauptaussage auszudrücken?). Die einzelnen Sätze sollten mit größter sorgfalt kombiniert werden.

Das Verfassen des Abstraktes sollte bis zum Schluss aufgeschoben werden, aus dem selben Grund, warum der Titel der Arbeit erst dann endgültige Form erhalten sollte, wenn das entsprechende Arbeit ansonsten vollständig ist.

Die nachfolgenden Ausführungen sollen nochmal ins Gedächtnis rufen was die Anforderungen an einzelne Abschnitte einer Abschlussarbeit an der FIM der Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK) sind.

Die Kurzfassung schließt mit der Nennung von 5 bis 10 Schlagwörtern (Keywords) ab, welche der Verschlagwortung bspw. für die Recherche nutzbar gemacht werden. Genauer gesagt, sind das inhaltliche Schlüsselwörter Ihrer Arbeit.

Die Struktur dieser Arbeit gliedert sich wie folgt:

- **Theoretische Grundlagen:** In diesem Abschnitt werden die grundlegenden

Konzepte des Reinforcement Learning erläutert, darunter Neuronale Netze Markov-Entscheidungsprozesse und Probleme, welche bei dem RL auftreten können. Weiterhin wird eine detaillierte Beschreibung des Spiels 'Noch mal' gegeben, einschließlich der Regeln, Spielziele und möglicher Strategien.

- **Konzeption:** Dieser Abschnitt umfasst eine Anforderungsanalyse und beschreibt den Ansatz zur Implementierung des Reinforcement Learning-Agenten für 'Noch mal'.
- **Experimente und Ergebnisse:** Es werden die Ergebnisse der Experimente präsentiert, einschließlich der Leistung des RL-Agenten beim Spielen von 'Noch mal' und einer Analyse seiner Fähigkeiten und Schwächen.
- **Diskussion:** Eine Diskussion über die Ergebnisse, die Einschränkungen der Studie und mögliche Verbesserungen wird vorgenommen.
- **Fazit und Ausblick:** Abschließend werden die wichtigsten Erkenntnisse zusammengefasst und ein Ausblick auf mögliche zukünftige Forschungsrichtungen gegeben.

Keywords: IoT, SD-WAN, Machine Learning (ML), Fiber to the Home (FTTH)

Inhaltsverzeichnis

1 Einleitung

In der heutigen Zeit steht die Menschheit an der Schwelle einer digitalen Revolution, in der künstliche Intelligenz eine immer bedeutendere Rolle spielt. Insbesondere das Gebiet des Reinforcement Learnings hat in den letzten Jahren enorme Fortschritte gemacht und findet Anwendung in einer Vielzahl von Bereichen. Diese Entwicklung bietet vielfältige 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 beeindruckende Erfolge erzielt, indem es Algorithmen entwickelt hat, die komplexe Spiele wie 'Go' auf einem kompetitiven Niveau spielen können und sogar menschliche Fähigkeiten übersteigt. Diese Anwendung verdeutlicht die Vielseitigkeit und Leistungsfähigkeit von Reinforcement Learning bei der Bewältigung verschiedenster Herausforderungen und unterstreicht die Fähigkeiten von RL , komplexe Probleme zu lösen und neue Lösungswege zu finden.[1, 2]

1.1 Themenstellung

In dieser Arbeit liegt der Fokus auf der Implementierung und dem Training eines Reinforcement Learning-Agenten für ein, in Unity umgesetztes, Würfelspiel. 'Noch mal' ist ein Würfelspiel, das Strategie und Glück erfordert. 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 Arbeit zielt darauf ab, einen Beitrag zum Verständnis der Anwendung von Reinforcement Learning zu leisten.
Um den Trainingsprozess zu evaluieren, werden unterschiedliche Agenten in verschiedenen Szenarien trainiert. Dabei werden Probleme des RL aufgezeigt und untersucht,

welche Szenarien sich positiv auf das Training auswirken können. Durch diese systematische Untersuchung lassen sich Einblicke gewinnen, wie verschiedene Faktoren in der Trainingsumgebung den Lernfortschritt und die Leistung des RL-Agenten beeinflussen.

1.2 Werkzeuge

In dieser Arbeit werden verschiedene Werkzeuge verwendet, um dem Rahmen gerecht zu werden. **Unity** wird verwendet, um das Würfelspiel zu implementieren und dem Agenten eine Lernumgebung bereitzustellen. Mit **Pytorch** werden die RL-Modelle trainiert. Die Bibliothek **ML-Agents** stellt die Schnittstelle beider Technologien dar und ermöglicht das Trainieren von Agenten in Unity. Mittels **Tensorboard** werden aufgezeichnete Trainingsprozesse visualisiert, was eine Evaluierung ermöglicht. **Mathplotlib** wird verwendet, um Metriken zu visualisieren, welche nicht von Tensorboard unterstützt werden.

1.3 Gliederung

- **Theoretische Grundlagen:** Die theoretischen Grundlagen zum Reinforcement Learning und neuronalen Netzen werden hier dargestellt. Außerdem werden die Spielregeln des modellierten Spiels beschrieben.
- **Konzeption:** Dieses Kapitel beinhaltet eine Anforderungsanalyse. Auf Grundlage dieser wird die Auswahl der verwendeten Technologien begründet.
- **Implementierung:** In diesem Kapitel wird dargestellt, wie das Spiel und der Agent implementiert wurden. Darüber hinaus wird der Aufbau des Beobachtungsvektors und des Aktionsspeichers erläutert. Außerdem wird der Algorithmus zur Auswahl der Kästchen erläutert.
- **Präsentation der Ergebnisse:** Für diesen Abschnitt wurden Versuche mit unterschiedlichen Szenarien durchgeführt. Die Ergebnisse dieser werden in jenem Kapitel dargestellt.
- **Auswertung und Ausblick:** In diesem Kapitel werden die Ergebnisse bewertet und weitere Schritte zur Verbesserung des Agenten diskutiert.

2 Theoretische Grundlagen

2.1 Neuronale Netze

Neuronale Netze (=NN) sind ein Modell für künstliche Intelligenz nach dem Vorbild des Gehirns. Sie bestehen aus mehreren Schichten verknüpfter Neuronen. Diese verarbeiten numerische Informationen und wandeln diese in einen Output. In 2.1 ist der Aufbau eines künstlichen Neurons schematisch dargestellt. Jedes Neuron besitzt eine Aktivierungsfunktion, welche entscheidet, ob es aktiv ist oder nicht. Neuronen geben Werte von 0 (passiv) bis 1 (aktiv) als Output weiter.

Um ein NN zu erzeugen, wird eine Vielzahl der Neuronen miteinander verknüpft. Die Ausgänge der vorderen Neuronen sind dabei immer mit den Eingängen der Neuronen der nächsten Schicht verknüpft. 2.2 zeigt den Aufbau eines solchen Netzes.

Wird ein NN initialisiert, werden Kantengewichte zwischen den Neuronen zufällig verteilt. Diese Kantengewichte entscheiden, wie stark einzelne Neuronen in die nachfolgende Rechnung eingehen. Deshalb liefern untrainierte NN schlechte Ergebnisse und müssen lernen die Problemstellung richtig zu lösen. Dieser Prozess wird als Training bezeichnet. Während des Trainings eines NN werden Kantengewichte angepasst, um die Heuristik an ein optimales Ergebnis anzupassen. Richtig trainierte NN können gute Antworten für komplexe Problemstellungen geben, so liefern sie beispielsweise in der Mustererkennung gute Ergebnisse.[3] Für das Training von NN wird viel Rechenleistung gebraucht, da der Prozess mit vielen Rechenoperationen verbunden ist. [4]

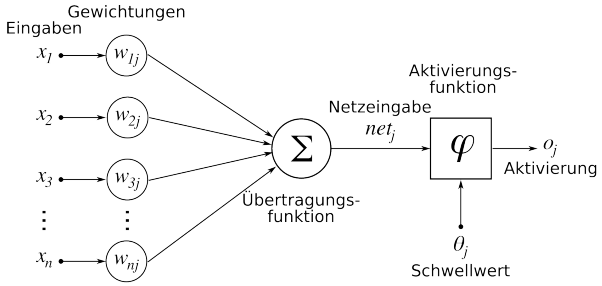

Abbildung 2.1: Aufbau eines künstlichen Neurons [5]

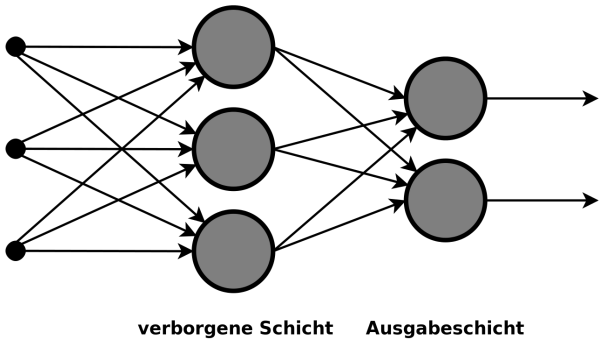

Abbildung 2.2: Schematische Darstellung eines neuronalen Netzes [5]

2.2 Reinforcement Learning

Reinforcement Learning ist ein Bereich des maschinellen Lernens, bei welchem ein Agent durch Interaktion mit seiner Umgebung (Environment) 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 Handlungen (Aktionen) um und bewertet diese mit numerischen Belohnungen (Rewards). 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 wahr. Jeder Zustand enthält eine Vielzahl von Merkmalen, 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 eines Agenten 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, erhält der Agent einen numerischen Rückgabewert, welcher jene Information beinhaltet. Diese numerischen Rückgabewerte werden als Belohnungen (Rewards) bezeichnet. Belohnungen werden vom 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 unter anderem:

- Überanpassung
- Belohnungsumgehung,
- Sparse Rewards
- große Beobachtungsvektoren

Von **Überanpassung** spricht man, wenn der Agent ein Problem gut in der Lernumgebung, in welcher er trainiert wurde, lösen kann, allerdings nicht auf abweichenden Umgebungen. Es tritt auf, wenn der Agent zu spezialisiert auf seine Trainingsumgebung ist und sich nicht an andere Situationen anpassen kann. Diese Problem lässt sich durch Generalisierung des Trainingsprozesses beheben. Ein Agent, welcher in vielen zufällig generierten Umgebungen trainiert, kann deutlich besser mit neuen Situationen umgehen, benötigt im Gegenzug dazu auch mehr Zeit zum Trainieren. [6]
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. [7]
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 den 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. [8]
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. [9]

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.
2.3 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 bewältigen muss. Dieser Prozess wiederholt sich, bis das Problem gelöst wurde. [9]

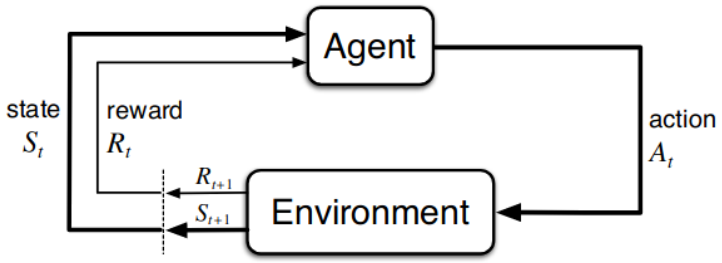

Abbildung 2.3: Ablauf eines MDP [10]

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. Jeweils ein Farb- und ein Zahlenwürfel müssen kombiniert werden, um entsprechend verfügbare und zusammenhängende Felder der gewählten Farbe abzukreuzen. Dabei kann der Zahlenwürfel die Werte von eins bis fünf oder einen Zahlenjoker ergeben. Der Farbwürfel besitzt die fünf Farben des Feldes und einen Farbjoker. Gespielt wird nach folgenden 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

Die Grafik 2.4 ist ein Spielfeld aus 'Noch mal!'. Nach Vorlage dieses Spielfelds wurde die Lernumgebung für den Agenten modelliert. 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 gelb markiertes Sternfeld. Diese geben zum Ende des Spiels Minuspunkte, weshalb es wichtig ist alle Sternfelder auszufüllen. Abbildung 2.5 zeigt Beispiele für korrektes und fehlerhaftes Ankreuzen.

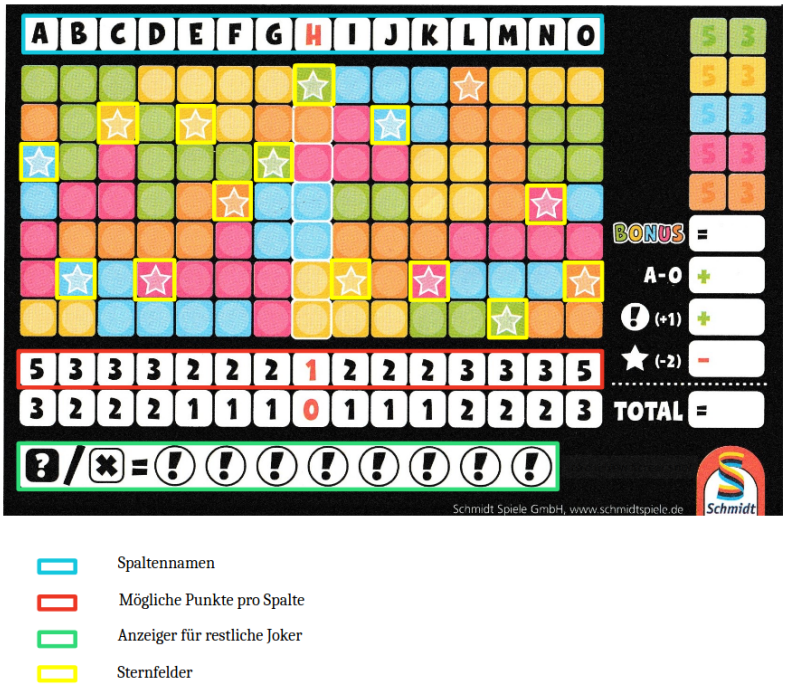

Abbildung 2.4: Vorlage des schwarzen Spielfeldes mit Indikation der Punktewertung [11]

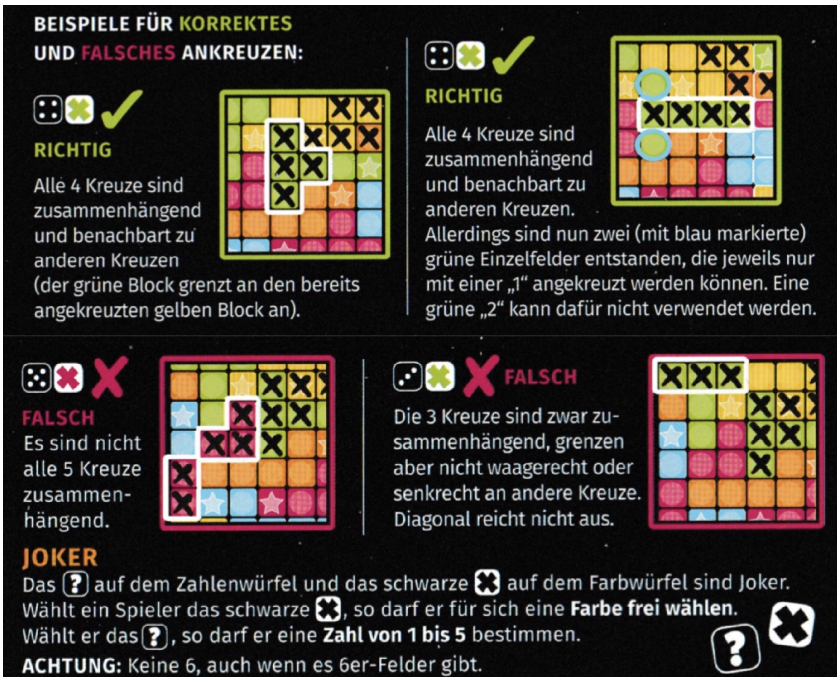

Abbildung 2.5: Beispiele für korrektes und falsches Ankreuzen aus der Spielanleitung [11]

2.3.1 Spielablauf Einspielervariante

Der Spieler hat 30 Züge Zeit maximale Punkte zu erreichen und würfelt jede Spielrunde alle 4 Würfel, bestehend aus 2 Farb- und 2 Zahlenwürfeln. 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- und 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 werden mehr Punkte vergeben als für die inneren Spalten. Für das komplette Ausfüllen einer Farbe erhält der Spieler fünf Punkte pro ausgefüllter Farbe. Eine Farbe gilt als komplett ausgefüllt, wenn jedes Feld dieser Farbe abgekreuzt wurde. Jedes nicht angekreuzte Sternfeld gibt zum Spielende zwei Minuspunkte. Für jeden übrig gebliebenen Joker erhält der Spieler zum Ende des Spiels einen Punkt.
2.6 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.

Abbildung 2.6: Grafik zur Bewertung der gesammelten Punkte der Einspieler Variante [11]

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. Eine Farbe gilt als vollständig ausgefüllt, wenn kein Feld dieser Farbe mehr vorhanden ist. 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.

3 Konzeption

3.1 Anforderungsanalyse

In dieser Arbeit soll das Spiel 'Noch mal!' implementiert 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. Spielparameter müssen konfigurierbar sein, um die Flexibilität des Trainings zu erhöhen.
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, welches 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. In 3.1 werden die Anforderungen zusammengefasst und in funktionale und nicht funktionale Anforderungen kategorisiert.

Funktional	**Nicht-funktional**
Implementierung des Spiels	Überwachen des Trainings
Bereitstellen der Lernumgebung	Parallelisierung des Trainingprozesses
Abdecken der Spielfunktionalitäten	Konfigurierbare und speicherbare Lernumgebungen
Unterbinden von illegalen Spielzügen	Visualisierung des Spiels
Schnittstelle für RL-Framework	Verwendung einer leistungsfähigen Gameengine
RL-FrameWork	Benutzerfreundliche Entwicklungsumgebung
	Konfigurierbare Spielparameter

Tabelle 3.1: Anforderungen an die verwendeten Technologien für die Nutzung von RL-Agenten

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. Mit der Auswahl von Unity als Gameengine ist bereits die Mehrheit der funktionalen Anforderungen aus 3.1 teilweise erfüllt, müssen jedoch programmiertechnisch umgesetzt werden.
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. Weiterhin 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. Durch die Nutzung des Frameworks, ist die Schnittstelle für die Nutzung von **Pytorch** geschaffen. Pytorch wird als RL-Framework eingesetzt und passt während des Trainings die Kantengewichte der NN an.
Zur 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, beziehungsweise deren Trainingsfortschritte zu bewerten.
Mit der Verwendung jener genannten Technologien und Frameworks ist es möglich den Rahmen dieser Arbeit zu bearbeiten und zu bewerten. In 3.2 wird ersichtlich, welche Anforderung über welche Technologie erfüllt wird.

Anforderung	**Technologie**
Implementierung des Spiels	Unity
Bereitstellen der Lernumgebung	Unity
Abdecken der Spielfunktionalitäten	Unity
Unterbinden von illegalen Spielzügen	Unity
Schnittstelle für RL-Framework	ML-Agents
Überwachen des Trainings	TensorBoard, Matplotlib
Parallelisierung des Trainingsprozesses	ML-Agents
Konfigurierbare Lernumgebungen	ML-Agents
Visualisierung des Spiels	Unity, Matplotlib
Benutzerfreundliche Entwicklungsumgebung	Unity
Konfigurierbare Spielparameter	Unity
RL-Framework	Pytorch

Tabelle 3.2: Anforderungen an die Implementierung des Spiels 'Noch mal!' für einen RL-Agenten und die entsprechenden Technologien

4 Implementierung

4.1 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 hilfreich beim Erstellen paralleler Trainings, da dieser mehrfach in die Szene aufgenommen werden musste, um mehrere Spielfelder, welche gleichzeitig bespielt werden, zu initialisieren.

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 aktuelle Sprite zerstört und ein neues initialisiert. 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. A.2 zeigt die verwendeten Sprites für den Zahlenwürfel.

Wie der Zahlenwürfel, implementiert der **ColorDice** die Funktionalität des Würfelns der Farben. Diese werden als String dargestellt und können folgende Werte annehmen: {'blue', 'green','red', 'yellow', 'orange', 'joker'}.
Zur Visualisierung wird ein Sprite erstellt, welches in der gewürfelten Farbe eingefärbt wird. Ein schwarzes Feld entspricht dem gewürfelten Farbjoker. Das Einfärben der Sprites wird mittels des Switch-Case Statements in C.1 ermöglich.

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. A.1 zeigt neun dieser Spielfelder, welche nach Vorlage von 2.4 modelliert wurden.

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 neues Spiel zu starten

Die **FieldSquares** stellen die einzelnen Teilfelder des Spielfeldes dar. In Tabelle 4.1 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 4.1: Übersicht Informationen der Fieldsquares

4.2 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 und in die Farben der hinterlegten Information eingefärbt. Ausgefüllte Kästchen werden grau eingefärbt, diese Funktionalität wird im Fieldsquare Prefab ausgeführt. A.1 zeigt den Aufbau der Trainingsumgebung. Jedes der Spielfelder besitzt einen eigenen Agenten und ist somit die Lernumgebung für genau einen Agenten. Alle Agenten trainieren gemeinsam ein Modell. Es können beliebig viele weitere Lernumgebungen hinzugefügt werden. Mehrere gleichzeitige Traingsvorgänge bedeuten jedoch wiederum einen höheren Rechenaufwand.

4.3 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 Agenten 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 4.2, zu welchem die Informationen der aktuellen Zustands hinzugefügt werden. Während des Trainings eines neuronalen Netzes, muss die Größe des Vektors und die Art der Werte gleich bleiben. Deswegen ist es nicht möglich ein Modell auf Spielfeldern mit unterschiedlichen Größen zu trainieren, da sich so die Anzahl der Beobachtungen unterscheiden würden. C.4 erstellt anhand der Daten den Vektor aus Beobachtungen und übergibt ihn an den Agenten.

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 4.2: 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 4.3: Aufbau FeldVektor

Anhand der Observations berechnet das neuronale Netz einen Ausgabevektor. Tabelle 4.4 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 werden die Kantengewichte des Netzes angepasst, um das bestmögliche Ergebnis zu erreichen.

Das Vergeben von Rewards lehnt sich an die Punktevergabe im Spiel, wie nach Vorlage 2.4, an. Der Agent erhält Belohnungen, wenn er auch Punkte im gespielten Spiel erlangen würde. In 4.5 ist eine Übersicht der Belohnungen. Aus dieser lässt sich ablesen, welche Aktion welchen Reward auslöst.

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	-

Tabelle 4.4: Index und Beschreibung der Variablen

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 4.5: Belohnungen für bestimmte Aktionen

4.4 Spezifikation 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 Spielfelds 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. C.7 Valide Felder werden in eine Liste (availableFields) aus Verfügbaren Feldern geschrieben. Abb. 4.1 zeigt den beschriebenen Zustand des Feldes.

Abb. 4.2 bildet den nächsten Schritt im Algorhitmus ab. Es wird geprüft, ob verfügbare Felder 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. C.8

Im nächsten Schritt wird jedem der verfügbaren Nachbarn, abhängig der Gesamtanzahl, ein Wertebereich zwischen 0 und 1 zugewiesen, dies wird in 4.3 und 4.6 verdeutlicht. Anhand des diskreten Wertes des Ausgabevektors, wird das zugehörige Feld in eine Liste aus den gewählten Feldern geschrieben. C.9

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

Tabelle 4.6: Bereiche für bestimmte Felder

Für alle gewählten Felder werden die benachbarten Felder zurückgegeben und der vorherige Schritt wiederholt. Wenn so viele Felder gewählt, wie erwürfelt wurden, werden die Felder anschließend ausgefüllt und auf Rewards überprüft. Dies wird in Abbildung 4.4 dargestellt.

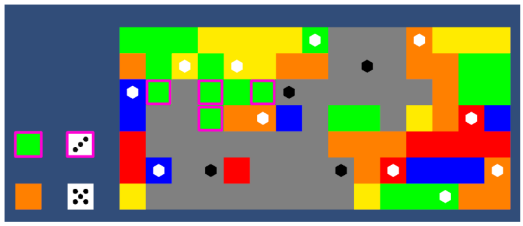

Abbildung 4.1: Valide Felder für das gewählte Würfelergebnis wurden markiert

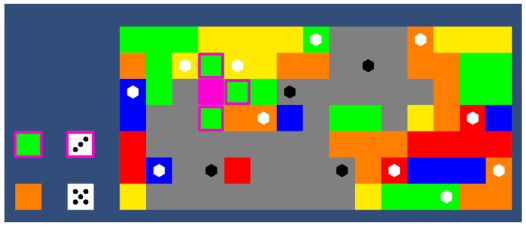

Abbildung 4.2: Feld(3,4) wird in die pickedField Liste aufgenommen und benachbarte Felder werden zurückgegeben.

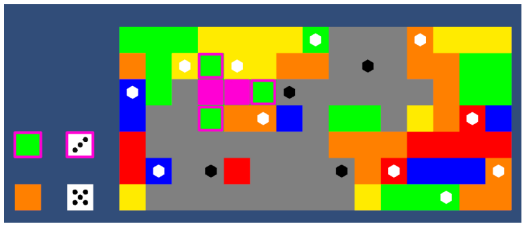

Abbildung 4.3: Feld(4,4) ist das nächste gewählte Feld und wird in pickedFields aufgenommen

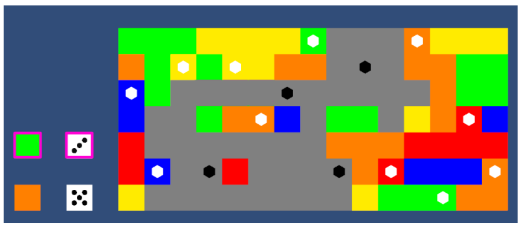

Abbildung 4.4: Felder wurden gewählt und ausgefüllt

4.5 Auswertung der erreichten Punkte

Um die verschiedenen Modelle bewerten zu können, wurden die erreichten Punkte aufgezeichnet und anschließend ausgewertet. Im Verlauf eines Spiels werden alle erreichten Punkte im Spielfeld gespeichert. Nach 30 gespielten Zügen wird die erreichte Punktzahl in eine Logdatei geschrieben. In 4.5 lässt sich ableiten, dass Belohnungen nur dann verteilt werden, wenn es auch zu einer Punktewertung kommt. Die beiden Werte sind lediglich um 30 Punkte verschoben, da der Spieler das Spiel mit -30 Punkten beginnt.

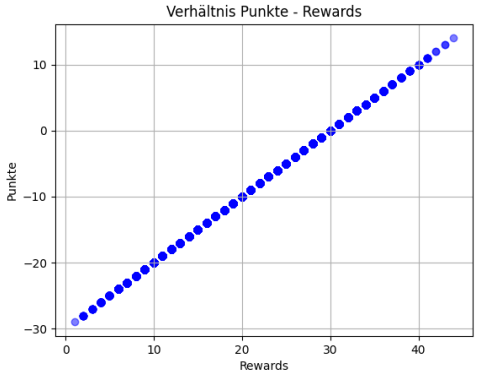

Abbildung 4.5: Verhältnis Rewards zu Punkten

Die Logdateien wurden anschließend mit Python ausgewertet. In C.5 werden die angegebenen Punkteaufzeichnugnen importiert und jedem Datensatz ein Titel zur Beschriftung der Graphen hinzugefügt. Im Anschluss wird der gleitende Durchschnitt,abhängig der gewählten Anzahl an Spielen berechnet.

Dabei werden erst alle Werte addiert und der Mittelwert gebildet. Danach wird der herrausrotierende Wert abgezogen und um den Neuen ergänzt. Auf diese Weise muss jeder Wert lediglich 2 mal gelesen werden.
In der Methode C.6 können nun variabel Graphen gewählt und anschließend gerendert werden. Vor Methodenaufruf, muss die Liste an anzuzeigenden Graphen bearbeitet werden und in den Methodenaufruf übergeben werden.

5 Präsentation der Ergebnisse

Um den Lernprozess evaluieren zu können, wurden verschiedene Experimente durchgeführt. Diese Experimente werden in diesem Kapitel vorgestellt und ausgewertet. Im folgenden Kapitel wird oft von dem trainierten Agenten gesprochen. Damit ist ein Modell gemeint, welches 25 Millionen Trainingsschritte auf dem schwarzen Spielfeld durchgeführt hat. Dies wird in 2.4 dargestellt. Dies entspricht etwa 830k gespielten Spielen.

5.1 Agent mit zusätzlichen Belohnungen

In diesem Versuch bekam der Agent, zusätzlich zu den ursprünglichen Rewards, weitere Belohnungen für bestimmte Aktionen. Diese Belohnungen sollten den Agenten zu komplexeren Spielzielen leiten, um so schneller zu einer optimalen Policy zu gelangen. In 5.1 sind alle zusätzlichen Rewards ersichtlich. In diesem Versuch wurde 2.4Mio Spielzüge ausgeführt. Es stellte sich, trotz geringer Anzahl an Lernschritten, ein negatives Ergebnis herraus. Deshalb wurde auf ein 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 5.1: Zusatzbelohnungen für verschiedene Aktionen

Die Grafiken 5.1 und 5.2 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. Dieser hielt das Abkreuzen von Clustern für deutlich effizienter, um seine Belohnungen zu maximieren, obwohl die Belohnungen für erreichte Spalten oder komplett ausgefüllte Farben mehr Punkte ergaben. Dies spricht dafür, dass in diesem Versuch Belohnungsumgehung aufgetreten ist. Verhindert werden könnte dies, indem dem Agenten die Exploration erleichtert wird. Beispielhaft kann die Anzahl der Spielzüge vergrößert werden. Eine verlängerte Spieldauer hätte zur Folge, dass größere Teile des Spielfeldes ausgefüllt werden, wodurch der Agent komplexere Belohnungen erreichen und erlernen kann.

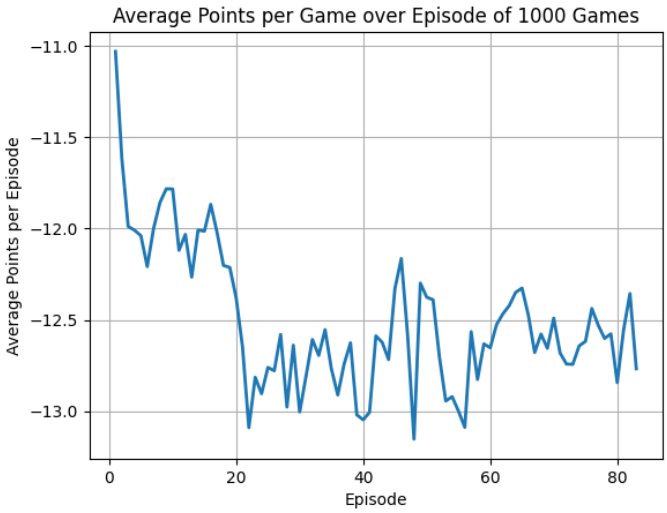

Abbildung 5.1: Durchschnittliche Punkte des Agenten mit Zusatzbelohnungen

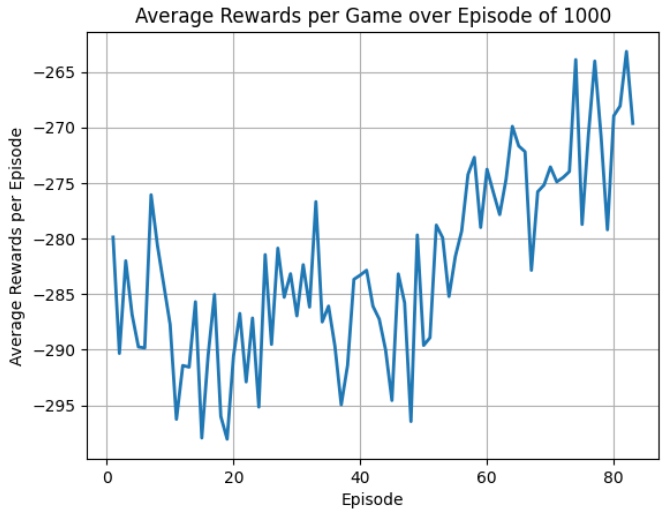

Abbildung 5.2: Durchschnittliche Belohnungen des Agenten mit Zusatzbelohnungen

5.2 Vergleich trainiert und untrainiert

In diesem Experiment werden die erreichten Punkte und Rewards eines untrainierten Agenten gegenüber den erzielten Ergebnissen eines trainierten Agenten gestellt. Der untrainierte Agent trainiert ein neu initialisiertes NN, welches zufällig gewählte Kantengewichte zwischen den Neuronen erhält. Um den Lernfortschritt zu bewerten, wurden eine Million Spielzüge von beiden Agenten ausgewertet. Beide nutzten das schwarze Spielfeld, welches in 2.4 dargestellt wurde, als Lernumgebung. Wie an den Grafiken 5.4 und 5.3 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.

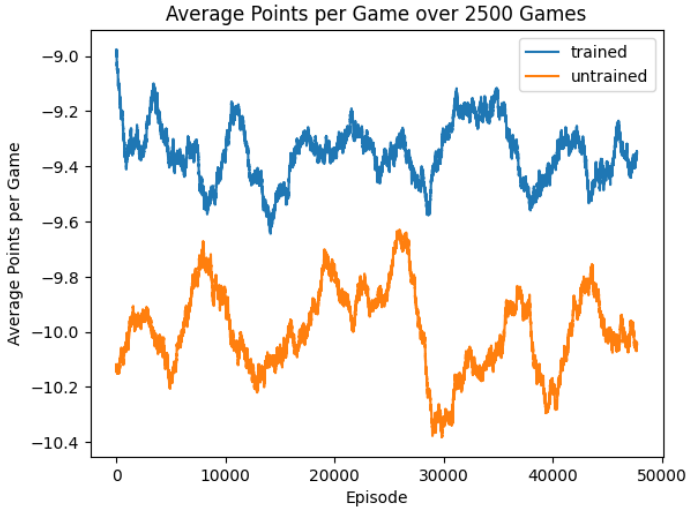

Abbildung 5.3: Durchschnitt der erreichten Punkte pro Spiel

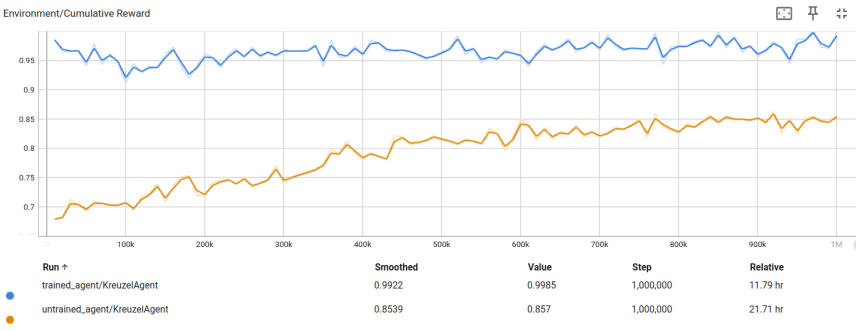

Abbildung 5.4: Übersicht der gesammelten Belohnungen

5.3 Training auf Sonderfeldern

In diesem Experiment wurden die erreichten Puntke und Rewards des trainierten Agenten gegen einen Agenten, welcher mit Sonderfeldern trainiert wurde verglichen. Beide Agenten wurden mit dem selben vortrainierten Modell instanziiert. Dieser Versuch sollte überprüfen, ob ein Agent im Verlauf besser erlernen kann, welche Stellen im Beobachtungsvektorvektor für welche Information zuständig sind.

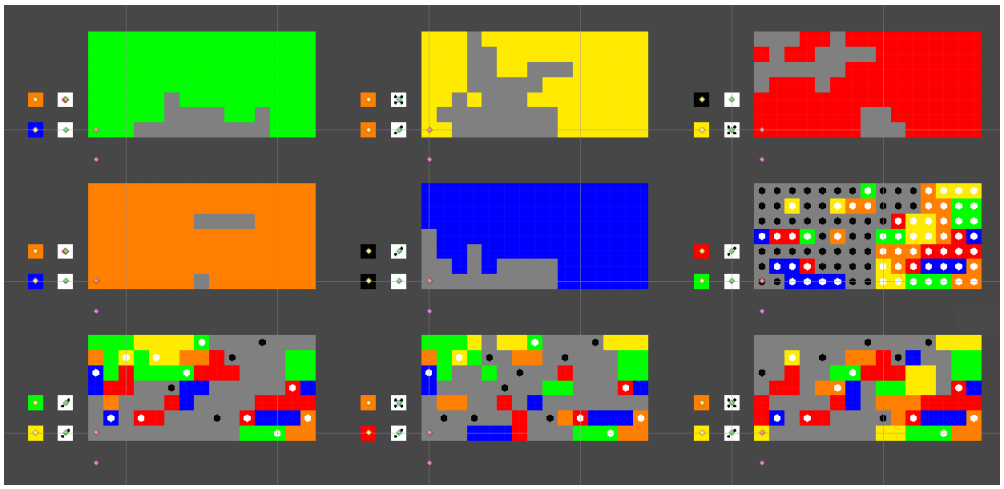

Abbildung 5.5: Übersicht der speziellen Felder

In der Grafik 5.5 ist das Trainingumfeld mit den speziellen Feldern dargestellt. Jedes der Felder hat gewisse Besonderheiten, welche sich von den normalen Spielfeldern abgrenzen. Fünf Felder sind in einer kompletten Farbidentität eingefärbt. In C.3 wurde der Zahlenwürfel manipuliert, 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 vornherein 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.
Aus der Grafik 5.6 lässt sich ableiten, dass das Modell, welches mit speziellen Feldern trainiert wurde, im Durchschnitt weniger Punkte erreichte, als der normal trainierte Agent. Dieses Training führte nicht zu einer Verbesserung des Modells. Ursache hierfür liegt sicher im Spielfeld, 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.

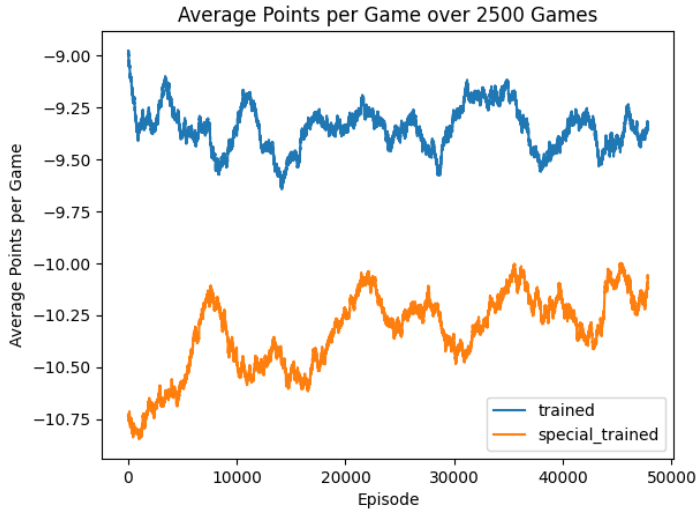

Abbildung 5.6: Durchschnitt der erreichten Punkte beider Agenten

5.4 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 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 5.7 zeigt, dass das Experiment eine Verbesserung des Models zur Folge hatte, da im Durchschnitt etwas mehr Punkte erreicht wurden.

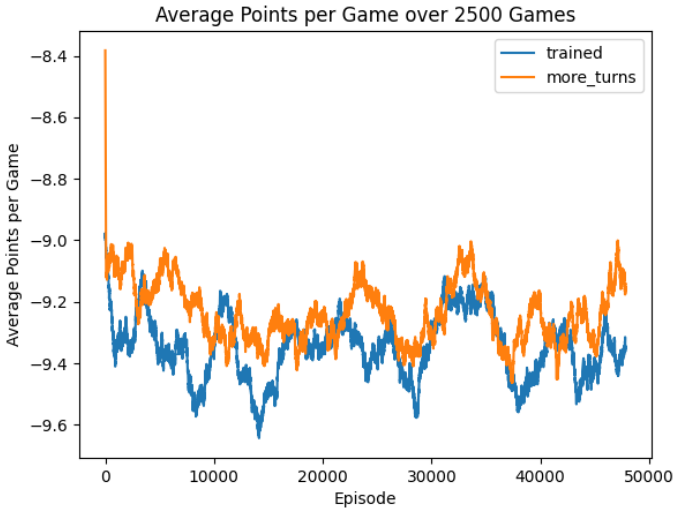

Abbildung 5.7: Vergleich 'Mehr Züge' und 'trainierter Agent'

5.5 Überprüfung auf Überanpassung

In diesem Experiment sollte der Agent auf Überanpassung überprüft werden. Trainierter und untrainierter Agent spielten das Spiel nach normalen Spielregeln auf einem anderen Spielfeld. 5.8 zeigt die angepasste Trainingsumgebung. Das Spielfeld wurde dem orginalen Spiel nachempfunden. Beide Agenten trainierten 1 Mio. Lernschritte, wobei erreichte Punkte und Belohnungen aufgezeichnet wurden. In den Grafiken 5.9 und 5.10 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.

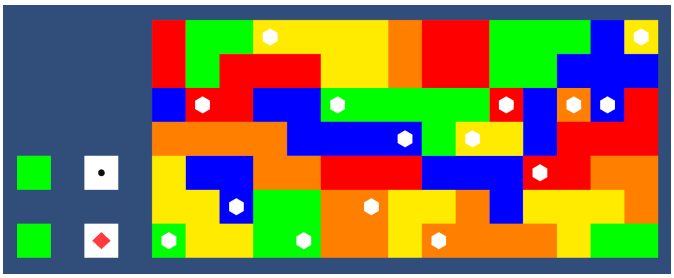

Abbildung 5.8: Trainingsumgebung für das Training auf dem orangenem Feld

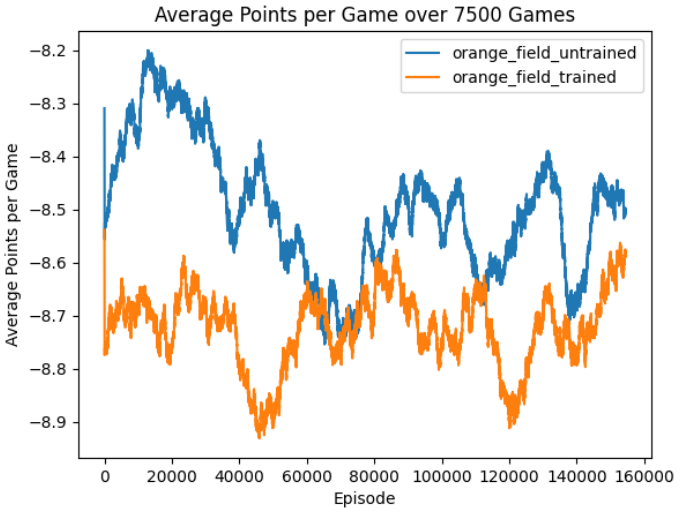

Abbildung 5.9: Vergleich Punkte ’trainiert’ und ’untrainiert’ auf orangen Spielfeld

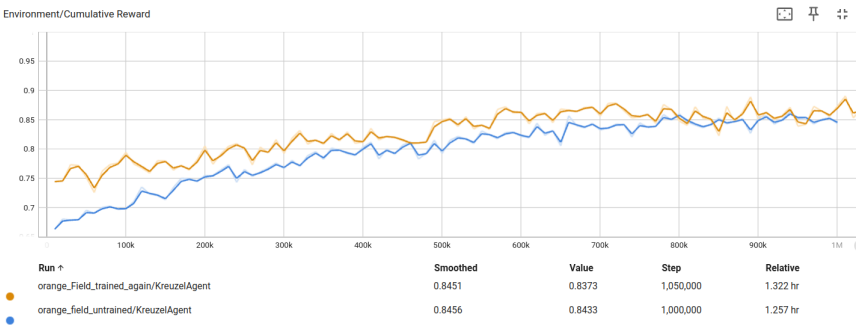

Abbildung 5.10: Übersicht gesammelte Rewards auf orangen Spielfeld

5.6 Training auf Minifeld

Um zu überprüfen, ob ein kleineres Feld einen positiven Effekt auf das Training hat, wurde ein Spielfeld um 3 Zeilen gekürzt und fungierte als Trainingsumgebung. 5.11 zeigt das angepasste Spielfeld. Im Anschluss wurde die Leistung des Agenten auf dem normalen Spielfeld gegenüber dem Ergebnis eines untrainierten Agenten gestellt. Beide Agenten wurde mit einem neuen NN initialisiert. Da die Beobachtungen von Modellen gleich bleiben müssen, wurden nicht mehr vorhandene Kästchen mit Nullen im Vektor präsentiert. 5.1 zeigt den Quellcode zum Übergeben eines leeren Feldes.

Quellcode 5.1: Befüllen des Beobachtungsvektor mit leerem Feld

```
1  private void PushEmptyField(VectorSensor sensor){
2      sensor.AddObservation(GetColorIndexOneHotFromColor(""));
3      sensor.AddObservation(0);
4      sensor.AddObservation(0);
5      sensor.AddObservation(0);
6  }
```

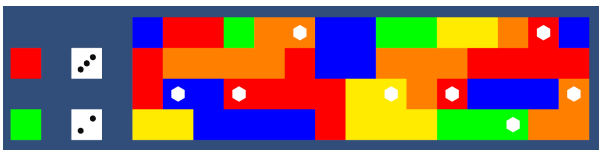

Abbildung 5.11: Lernumgebung für einen Agenten auf dem 'MiniFeld'

Die Grafiken 5.12 und 5.13 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. Belohnungen wurden auch hier nur verteilt, wenn es zur Punktewertung kommt. Deshalb ist es interessant, dass der untrainierte Agent mehr Punkte erreicht, als der auf dem kleinen Feld vortrainierte Agent, obwohl dieser wiederum einen höheren Durchschnitt an Belohnungen erhält.

Ursache für das schlechtere Ergebnis ist auf die Überanpassung zurückzuführen. So wurde beim Training auf dem kleinen Feld ausschließlich dieses angepasste Feld erlernt und nicht eine gute Spielweise. Weiterhin konnte der auf dem kleineren Feld trainierte Agent nicht den kompletten Beobachtungsvektor erlernen, da ein großer Teil leer übergeben wurde.

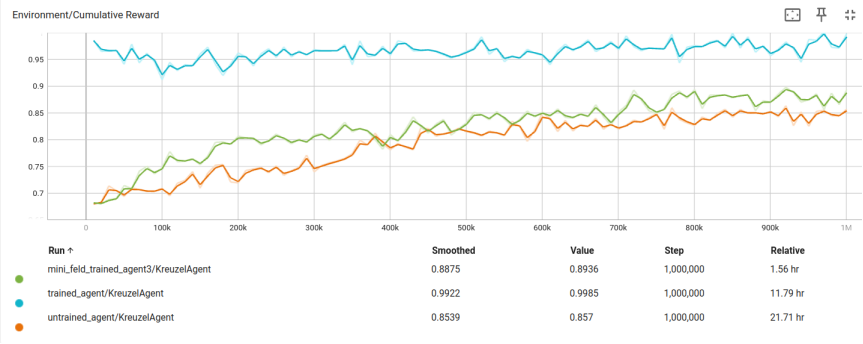

Abbildung 5.12: Gesammelte Belohnungen mit Minifeld

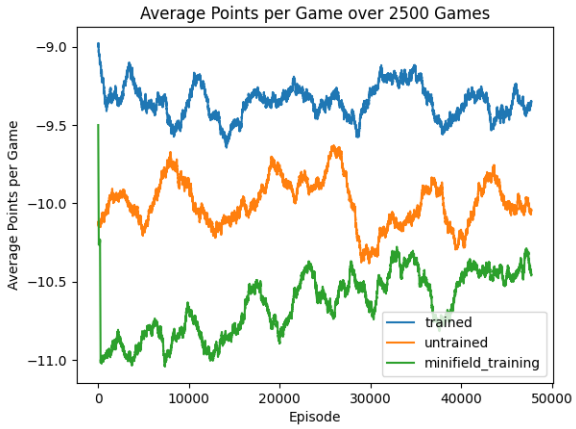

Abbildung 5.13: Vergleich erreichte Punkte

5.7 Training blinder Agent

In diesem Experiment wurden dem blinden Agenten lediglich die Würfel, die Anzahl der verbleibenden Joker und die aktuelle Runde des Spiels übergeben. Dieser Versuch sollte zeigen, wie gut ein Agent, der die aktuelle Informationen vom Spielfeld nicht erhält, perfomt. Dies sollte überprüfen, ob das trainierte Modell tatsächlich besser ist, als eine rein zufällige Spielweise. Da die Auswahl der Felder durch zufällige Interpolation aller möglichen Felder abläuft, kann der blinde Agent normal spielen. Durch den Versuchsaufbau verringert 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 übergeben bekommt, kann er dieses implizit erlernen. Dieser Prozess wäre allerdings nicht sonderlich robust, 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 5.14 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 und erreichte auch nach 25Mio Lernschritten noch keinen Maximalwert. Abbildung 5.15 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.

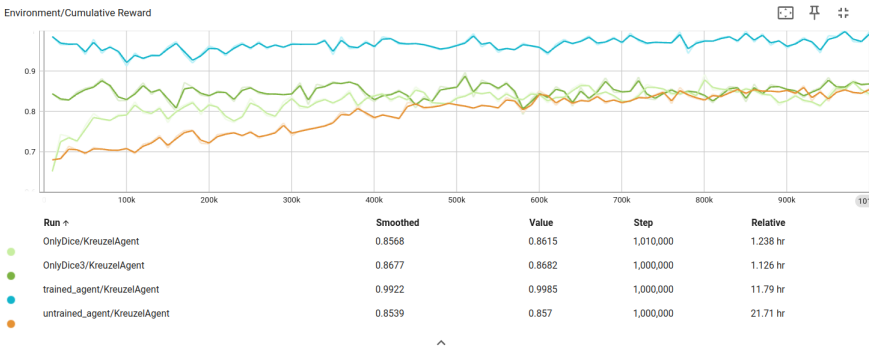

Abbildung 5.14: Gesammelte Belohnungen blinder Agent

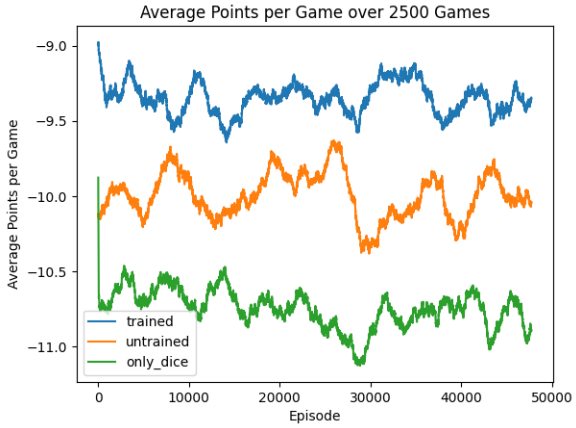

Abbildung 5.15: Vergleich erreichte Punkte blinder Agent

6 Auswertung und Ausblick

6.1 Bewertung der Ergebnisse

Die Experimente aus dem vorherigen Kapiteln zeigen deutlich, dass der Trainingsfortschritt des Agenten vorhanden war. Das trainierte Modell erhielt deutlich mehr Belohnungen als die untrainierten oder mit speziellen Lernumgebungen trainierten Agenten. Leider lässt sich dieser Fortschritt nur bedingt auf die Metrik der durchschnittlich erreichten Punkte anwenden. Wünschenswert wäre ein Modell gewesen, welches im Durchschnitt positive Spielergebnisse sammelt. Dem Agenten gelang es nicht kontinuierlich nach einer Strategie zu spielen, welche viele Punkte nach sich zieht. So konnte er fast nie eine Farbe komplett ausfüllen, was viele Punkte bringen würde. Dies liegt vorallem daran, dass er dieses Ziel von dem Agenten nur sehr selten erreicht wurde und somit auch nicht erlernt werden konnte.
Das Ergebnis auf einem anderen Spielfeld war für den trainierten Agenten überraschend schlecht. Dies spricht dafür, dass das Training auf einem einzelnen Spielfeld nicht förderlich ist für ein Modell, das auf verschiedene Situationen reagieren soll.
Während des Verlaufs dieser Arbeit, musste ich feststellen, dass das Problem, ein wenig kompliziertes Spiel wie 'Noch mal!' zu erlernen, für einen RL Agenten deutlich komplexer ist, als erwartet. Der Agent musste eine Vielzahl von Daten verarbeiten und diese semantisch zuordnen. Dies führte zu einer 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 aufweisen. Weiterhin kommt hinzu, dass das modelierte Spiel ein Glücksspiel ist. Es kann vorkommen, dass der Spieler Pech im Würfeln hat und so trotz perfekter Spielweise ein schlechtes Spielergebnis erziehlt. Deshalb ist es auch nicht möglich ein Modell zu erzeugen, was immer ein gutes Ergebnis liefert. Allerdings wäre mit weiterer Rechenzeit

und vermehrter Nutzung von verschieden Trainingsarten noch eine weitere Verbesserung des Modells möglich.

6.2 Schritte zur Verbesserung des Agenten

Mit mehr Rechenleistung und Trainingszeit könnte der Agent durchaus noch zu besseren Ergebnissen gelangen. Ein Flaschenhals war unter anderem, dass jede Trainingsiteration von 1 Mio. Schritten ungefähr 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 auch nur langsam, stetig verbessern. Mit einer deutlich erhöhten Trainingsdauer, kann das Modell deutlich verbessert werden.
Ein kompletter Verzicht der Visualisierung während des Trainings kann zu einer Verbesserung der Trainingsdauer führen. Es würden mehr Ressourcen zur Berechnung des neuronalen Netzes zur Verfügung stehen und damit den Prozess 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.
Ein weiteres Problem des trainierten Modells war die Überanpassung. Um dies zu umgehen und Modelle zu trainieren, welche auf verschiedene Zustände des Spielfeldes optimal reagieren können, wäre es hilfreich das Spielfeld für jede Spielrunde zufällig generisch zu erzeugen. Dies würde dazu führen, dass der Agent nicht ein einzelnes Spielfeld , sondern das Spielen des Spiels erlernt. Generisch erzeugte Trainingsumgebungen würden die Trainingsdauer erhöhen, dafür aber in ein robusteres Modell resultieren.
Anpassen der Hyperparameter des Agenten können zu einer weiteren Optimierung des Trainingsprozesses führen. Leider ist das Anpassen der Hyperparameter nicht intuitiv nachvollziehbar, weshalb Erfahrung im Umgang mit RL erforderlich ist. Durch geschicktes Verändern dieser Parameter lassen sich lokale Maxima überspringen oder die Exploration des Agenten beschleunigen.
Eine weitere Verbesserung des Trainingsprozessen könnte auch eine ausgedehnte Selektion nach Vorbild der Evolutionären Algorithmen liefern. Dies hätte einen höheren Aufwand der Trainings zur Folge, könnte aber dazu führen, dass sich gute Modelle durchsetzen die schneller eine optimale Spielstrategie etablieren.

Literaturverzeichnis

[1] 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).

[2] *AlphaGo*. en. Dez. 2020. URL: https://deepmind.google/technologies/alphago/ (besucht am 04. 03. 2024).

[3] Uwe Lorenz. *Reinforcement Learning: Aktuelle Ansätze verstehen - mit Beispielen in Java und Greenfoot*. de. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020. ISBN: 978-3-662-61650-5 978-3-662-61651-2. DOI: 10.1007/978-3-662-61651-2. URL: http://link.springer.com/10.1007/978-3-662-61651-2 (besucht am 08. 03. 2024).

[4] 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).

[5] *Künstliche Neuronale Netze | EF Informatik 2023*. de-CH. URL: https://informatik.mygymer.ch/ef2023/07-ki/08-knn.html#kunstliches-neuron (besucht am 06. 03. 2024).

[6] Chiyuan Zhang u. a. *A Study on Overfitting in Deep Reinforcement Learning*. en. arXiv:1804.06893 [cs, stat]. Apr. 2018. URL: http://arxiv.org/abs/1804.06893 (besucht am 18. 03. 2024).

[7] Alexander Pan, Kush Bhatia und Jacob Steinhardt. *The Effects of Reward Misspecification: Mapping and Mitigating Misaligned Models*. en. arXiv:2201.03544 [cs, stat]. Feb. 2022. URL: http://arxiv.org/abs/2201.03544 (besucht am 18. 03. 2024).

[8] Joshua Hare. *Dealing with Sparse Rewards in Reinforcement Learning*. en. arXiv:1910.09281 [cs, stat]. Nov. 2019. URL: http://arxiv.org/abs/1910.09281 (besucht am 18. 03. 2024).

[9] 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.

[10] 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.

[11] 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

FIM Fakultät Informatik und Medien
FTTH Fiber to the Home
HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig
ML Machine Learning

A Anhang - Abbildungen

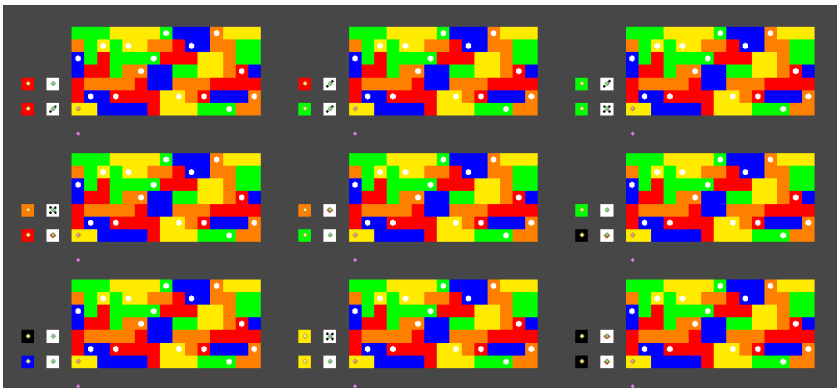

Abbildung A.1: Aufbau der Trainingsumgebung

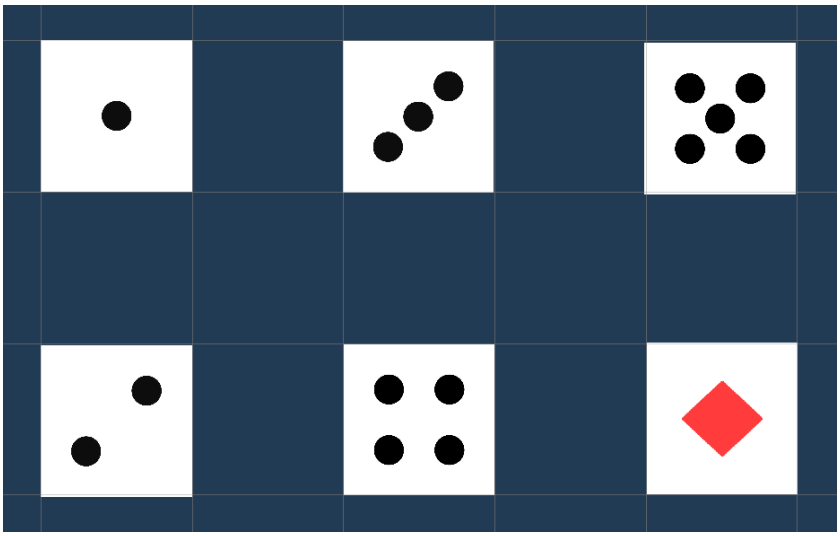

Abbildung A.2: Verwendete Sprites für den Zahlenwürfel

B Anhang - Tabellen

C Anhang - Quelltexte

Quellcode C.1: Switch Case Statement zum Anpassen der Farben des Farbwürfels

```
1  void SetColor(string color)  {
2      GameObject square = Instantiate(squarePrefab, transform.position, Quaternion.identity);
3      square.transform.parent = this.transform;
4      SpriteRenderer squareRenderer = square.GetComponent<SpriteRenderer>();
5
6      switch (color) {
7          case "red":     squareRenderer.color = Color.red;                    break;
8          case "blue":    squareRenderer.color = Color.blue;                   break;
9          case "green":   squareRenderer.color = Color.green;                  break;
10         case "yellow":  squareRenderer.color = Color.yellow;                 break;
11         case "orange":  squareRenderer.color = new Color(1.0f, 0.5f, 0.0f);    break;
12         case "joker":   squareRenderer.color = Color.black;                  break;
13     }
14 }
```

Quellcode C.2: Befüllen des Beobachtungsvektor mit validem Feld

```
1  private void PushValidField(Transform squareField, VectorSensor sensor) {
2      sensor.AddObservation(GetColorIndexOneHotFromColor(squareField.GetComponent<FieldSquare>().color));
3      sensor.AddObservation(squareField.GetComponent<FieldSquare>().available);
4      sensor.AddObservation(squareField.GetComponent<FieldSquare>().starField);
5      sensor.AddObservation(squareField.GetComponent<FieldSquare>().crossed);
6  }
```

Quellcode C.3: Erhöhen der Warscheinlichkeit eine bestimmte Farbe zu würfeln

```
1  public void Roll() {
2      List<string> colors = new List<string> {"red", "blue", "green", "yellow", "orange", "joker"};
3      if(colorShift != ""){
4          for (int i=0;i<10;i++){
5              colors.Add(colorShift);
6          }
7      }
8      int randomIndex = Random.Range(0, colors.Count);
9      string colorResult = colors[randomIndex];
10     this.color = colorResult;
11     SetColor(color);  // show dice
12 }
```

Quellcode C.4: Erstellen des Beobachtungsvektors

```
1
2   public override void CollectObservations(VectorSensor sensor)    {
3       sensor.AddObservation(GetJokersObservation());
4       sensor.AddObservation(GetRoundCountObservation());
5
6       foreach (Transform childTransform in controller.transform) {
7           if (childTransform.CompareTag("NumberDice")) {
8               GameObject child = childTransform.gameObject;
9               sensor.AddObservation(GetNumberDiceObservation(child));
10          }
11      }
12
13      foreach (Transform child in controller.transform) {
14          if (child.CompareTag("ColorDice")) {
15              sensor.AddObservation(GetColorIndexOneHotFromColor(child.GetComponent<ColorDice>().color));
16          }
17      }
18
19      int counter = 0;
20      foreach (Transform squareField in GameField.transform) {
21          if (squareField.CompareTag("Square")) {
22              PushValidField(squareField, sensor);
23              counter += 1;
24          }
25      }
26      do {
27          if (counter < 105){
28          PushEmptyField(sensor);
29          counter += 1;
30          }
31      }
32          while (counter < 105);
33  }
```

Quellcode C.5: Einlesen aller angegebenen Logdateien

```
import matplotlib.pyplot as plt
import numpy as np

def format_float(value):
    return "{:.3f}".format(value)

file_paths = [
    'LogPoints_trained.txt',
    'LogPoints_trained_with_more_turns_2.txt',
    'LogPoints_orange_field_untrained.txt',
    'LogPoints_orange_field_trained2.txt',
    'LogPoints_special_trained.txt',
    'LogPoints_untrained.txt',
    'miniField_trained.txt',
    'OnlyDice.txt'
    ]

filenames = [
    'trained',
    'more_turns',
    'orange_field_untrained',
    'orange_field_trained',
    'special_trained',
    'untrained',
    'minifield_training',
    'only_dice'
    ]


pointLists = []
for file_path in file_paths:
    points_list = []
    with open(file_path, 'r') as file:
        points = []
        amount_of_games = 2500
        for line in file:
            if line.strip():
                points.append(float(line.strip()))
                if len(points) > amount_of_games:
                    moving_average = sum(points[-amount_of_games:]) / amount_of_games
                    points_list.append(moving_average)
    pointLists.append(points_list)

min_length = min(len(points_list) for points_list in pointLists)
pointLists_clipped = [points_list[:min_length] for points_list in pointLists]
```

Quellcode C.6: Methode zum Anzeigen des Graphen

```
1
2  def display_selected_graphs(pointLists, filenames, selected_indices):
3      selected_pointLists = []
4      for index in selected_indices:
5          selected_pointLists.append(pointLists[index])
6
7
8      min_length = min(len(points_list) for points_list in selected_pointLists)
9      selected_pointLists_clipped = [points_list[:min_length] for points_list in selected_pointLists]
10
11     fig, ax = plt.subplots()
12
13     for i, points_list in zip(selected_indices, selected_pointLists_clipped):
14         x = np.arange(1, len(points_list) + 1)
15         ax.plot(x, points_list, label=f'{filenames[i]}')
16
17     ax.set_xlabel('Episode')
18     ax.set_ylabel('Average Points per Game')
19     ax.set_title(f'Average Points per Game over {amount_of_games} Games')
20     ax.legend()
21
22     plt.show()
23
24
25 selected_indices = [0,5] # Beispiel Trained vs Untrained
26 display_selected_graphs(pointLists, filenames, selected_indices)
```

Quellcode C.7: Gibt alle validen Felder für das gewählte Würfelpaar zurück

```
1      public List<GameObject> GetAvailableFieldsForGroupAndColor(string color, int number){
2          List<GameObject> availableFields = new List<GameObject>();
3          int counter = 0;
4          foreach(GameObject square in squares){
5              counter ++;
6              string squareColor = square.GetComponent<FieldSquare>().color;
7              bool available = square.GetComponent<FieldSquare>().available;
8              bool crossed = square.GetComponent<FieldSquare>().crossed;
9              int group = square.GetComponent<FieldSquare>().group;
10             if (color == "joker"){
11                 if (available && !crossed && group >= number){
12                     availableFields.Add(square);
13                 }
14             } else {
15                 if (available && !crossed && group >= number && squareColor == color){
16                     availableFields.Add(square);
17                 }
18             }
19         }
20         return availableFields;
21     }
```

Quellcode C.8: Gibt alle benachbarten Felder der selben Farbe zurück

```
1      public List<GameObject> CalculateNeighbours(List<GameObject> pickedFields){
2          List<GameObject> validNeighbors = new List<GameObject>();
3          foreach (GameObject field in pickedFields) {
4              List<GameObject> possibleNeighbors= GetNeighborsOfTheSameColor(field);
5              if (possibleNeighbors.Count != 0){
6                  foreach(GameObject  neighbor in possibleNeighbors) {
7                      if (!pickedFields.Contains(neighbor)){
8                          validNeighbors.Add(neighbor);
9                          }
10                     }
11                 }
12             }
13         return validNeighbors;
14     }
```

Quellcode C.9: Interpolation über alle möglichen Felder

```
1  private GameObject PickField(float action, List<GameObject> AvailableFields) {
2      action = Mathf.Clamp01(Mathf.Abs(action)*0.9999f);
3
4      int index = Mathf.FloorToInt(action * AvailableFields.Count);
5
6      index = Mathf.Clamp(index, 0, AvailableFields.Count-1);
7      return AvailableFields[index];
8  }
```