HAW HAMBURG

BACHELOR THESIS
Jonathan Siems

Deep Reinforcement Learning zum autonomen Erlernen eines nicht-deterministischen Spiels

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jonathan Siems

Deep Reinforcement Learning zum autonomen Erlernen eines nicht-deterministischen Spiels

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Marc Hensel
Zweitgutachterin: Prof. Dr. Ulrike Herster

Eingereicht am: 22. August 2025

Jonathan Siems

Thema der Arbeit

Deep Reinforcement Learning zum autonomen Erlernen eines nicht-deterministischen Spiels

Stichworte

Reinforcement Learning, Deep Learning, Q-Learning, Künstliche Intelligenz

Kurzzusammenfassung

Diese Bachelorthesis behandelt die Entwicklung und Untersuchung eines selbstlernenden Systems mit Fokus auf Lernmethoden des Reinforcement Learnings und dem Einsatz tiefer neuronaler Netzwerke.

Jonathan Siems

Title of Thesis

Deep Reinforcement Learning for autonomous learning of a non-deterministic game

Keywords

Reinforcement Learning, Deep Learning, Q-Learning, Artificial Intelligence

Abstract

This Bachelor Thesis discusses the development and study of an autonomous learning system with a focus on methods from reinforcement learning and the use of deep neural networks.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **ix**

1 Einleitung **1**

1.1 Motivation . . . 1
1.2 Zielsetzung . . . 2
1.3 Struktur der Arbeit . . . 2

2 Grundlagen und aktueller Stand der Technik **4**

2.1 Yahtzee . . . 4
2.1.1 Spielregeln . . . 4
2.2 Maschinelles Lernen . . . 7
2.2.1 Begriffseinordnung . . . 7
2.2.2 Grundlagen des maschinellen Lernens . . . 8
2.2.3 Arten des maschinellen Lernens . . . 12
2.3 Künstliche neuronale Netzwerke . . . 13
2.3.1 Geschichte . . . 14
2.3.2 Funktionsweise . . . 15
2.3.3 Arten künstlich neuronaler Netzwerke . . . 16
2.3.4 Begriffsdefinitionen . . . 17
2.4 Reinforcement Learning . . . 23
2.4.1 Geschichte . . . 23
2.4.2 Funktionsweise . . . 24
2.5 Stand der Technik . . . 27

3 Anforderungsanalyse **28**

3.1 Systembeschreibung . . . 28
3.2 Stakeholder . . . 29

3.3 Virtuelle Umgebung 30

4 Konzept **35**
4.1 Benötigte Software 35
4.1.1 Python 35
4.1.2 OpenAI Gymnasium 36
4.1.3 TensorFlow & Keras 36
4.1.4 Software-Übersicht 37
4.2 Virtuelle Umgebung 38
4.3 Konzept der Spiellogik und Environment 38
4.3.1 Spielablauf 38
4.3.2 Klassen der Spiellogik 39
4.3.3 Gymnasium Environment 40
4.4 Konzept des Agenten 44
4.4.1 Wahl des Vorgehens 45
4.4.2 Entwurf der Trainingsschleife 47
4.4.3 Agent- und DQN-Klasse 48
4.4.4 Aufbau des Trainings 49
4.4.5 Zusätzliche Überlegungen 52

5 Entwicklung **53**
5.1 Entwicklung der virtuellen Umgebung 53
5.1.1 Entwicklung der Spiellogik 53
5.1.2 Gymnasium Environment 54
5.2 Entwicklung des Agenten 56
5.3 Trainingsdurchführung 57
5.3.1 Vergleichswerte 57
5.3.2 Spielmodus 0 57
5.3.3 Spielmodus 1 61
5.3.4 Verbesserung des Belohnungssystems 62
5.3.5 Testen der Belohnungssysteme 67
5.3.6 Spielmodus 3 68

6 Auswertung der Ergebnisse **70**

7 Fazit und Ausblick **72**

Literaturverzeichnis **73**

Selbstständigkeitserklärung **76**

Abbildungsverzeichnis

2.1 Yahtzee: Ein Würfelbecher, eine Punktekarte und fünf Würfel 5
2.2 Bereiche der künstlichen Intelligenz in der Übersicht 8
2.3 Darstellung der threshold logic unit (TLU) 15
2.4 Schema eines neuronalen Netzes mit zwei verborgenen Schichten 16
2.5 Die Schwellenwertfunktion 20
2.6 Die Aktivierungsfunktionen Sigmoid, ReLu, Leaky-ReLu und ELU 21
2.7 Reinforcement Learning in einer schematische Darstellung 24

3.1 Die Systemumgebung 29
3.2 Anwendungsfalldiagramm 31

4.1 Flussdiagramm eines Yahtzee-Spiels (WW = Wiederholungswürfe) 39
4.2 Prototypen einer Würfel-, Punktekarte- und Spieler-Klasse 41
4.3 Prototypischer Aufbau der Environment in Anbindung an die Spielklassen 45
4.4 Flussdiagramm der Trainingsschleife 47
4.5 Klassendiagramm der Klassen DQAGENT und DQNET 49

5.1 Der vereinte Aktionsraum der vier Spielmodi (v.o) 54
5.2 Der Observationsraum bestehend aus mehreren Sub-Räumen 55
5.3 Klassendiagramm der implementieren Klassen 55
5.4 Übersicht über die Module des Programms 56
5.5 Klassendiagramm der Agent- und DQN-Klasse 56
5.6 Trainingsdurchlauf mit initialen Parametern – (Modus 0) 58
5.7 Angepasste Lernraten $\mathrm{lr} = 0,01$ (l.) und $\mathrm{lr} = 0,1$ (r.) – (Modus 0) 59
5.8 Training mit Batch-Größen 128 und 256 (o.v.l) sowie 512 und 1024 (u.v.l.) – (Modus 0) 60
5.9 Training mit angepassten Hyperparametern – (Modus 1) 62
5.10 Rewards bei angepasstem Diskontierungsfaktor $\gamma = 0,5$ und $\gamma = 0,95$ (v.l.) – (Modus 1) 63

5.11 Verluste bei angepasstem Diskontierungsfaktor $\gamma = 0,5$ und $\gamma = 0,95$ (v.l.) – (Modus 1) 63
5.12 Belohnungssysteme: Cat-Multi und Reroll-Inc (o.v.l), Adaptive-Target und Episode-Based-Average (u.v.l.) – (Modus 1) 67
5.13 Turn-Based-Shared Belohnungssystem: Rewards und Wiederholungswürfe je Episode – (Modus 1) 68
5.14 Durchschnittliche Rewards über 3.500 Episoden – (Modus 3) 69
5.15 Durchschnittliche Rewards über 10.000 Episoden – (Modus 3) 69

6.1 Unfertige CLI Anwendung 71

Tabellenverzeichnis

2.1 Exemplarische Würfelkombinationen und Punkte beim Yahtzee-Spiel . . . 6

4.1 Eingesetzte Software mit Versionsnummern 37

5.1 Durchschnittlich erreichte Punkte eines naiven Agenten 57
5.2 Initialer Satz an Trainings- und Hyperparametern 58
5.3 Batch-Größen und durchschnittlicher Reward 61
5.4 Angepasste Trainings- und Hyperparameter 61

1 Einleitung

In den vergangenen Jahren haben der Einsatz und die Forschung im Bereich des maschinellen Lernens rasant zugenommen. Gerade im Bereich der sogenannten künstliche Intelligenzen (KI) lassen sich regelmäßig neue Meilensteine verzeichnen. Die großen Sprachmodelle (engl. *large language models*, LLM), z. B. OpenAIs *GPT*, die aus sehr großen Datenmengen trainiert werden und augenscheinlich eine Antwort auf jede erdenkliche Frage bieten, haben inzwischen Einzug in alle Bereiche des Lebens gehalten. Möglich wurde das durch jahrzehntelange Forschungsarbeiten, die durch die heute verfügbare Rechenleistung sehr performante Lernalgorithmen zu Tage gefördert haben. Dabei bestehen diese Systeme häufig aus einer Reihe von Einzelsystemen, die auf Basis verschiedener Lernparadigmen trainiert werden. Sie sind auf bestimmte Aufgaben spezialisiert, so wie das Kategorisieren von Bildern, die Erkennung gesprochener Sprache oder die Verarbeitung von Text. Ein Bereich, der in den letzten Jahren einen starken Aufschwung erlebt hat, ist der des Reinforcement Learnings (RL), im deutschen auch *bestärkendes Lernen*. Dieses stellt den zentralen Punkt dieser Arbeit dar, in der ein RL-Algorithmus entwickelt werden soll, der eigenständig lernt das Würfelspiel Yahtzee zu spielen.

1.1 Motivation

Im Vergleich zu anderen Lernparadigmen lassen sich mit Reinforcement Learning Algorithmen entwickeln, die durch selbst gemachte Erfahrungen Strategien zur Lösung eines Problems finden. Voraussetzung dafür ist eine geeignete Umgebung, in der ein Agent die Möglichkeit hat durch Ausprobieren, Beobachtungen und durch Belohnungen auf eine Art zu lernen, die analog zum Lernverhalten intelligenter Tiere scheint. Beliebt ist die Erprobung von RL-Algorithmen insbesondere anhand von (Video-)Spielen, die in der Regel eine endliche Anzahl Aktionen (Eingangssignale), beobachtbare Zustände (Ausgangssignale) und mindestens ein Ziel (Belohnung) bieten. Insbesondere schwierig gestaltet sich dabei das Erlernen von nicht-deterministischen Systemen, z. B. Spielen deren Verlauf zu

Teilen zufallsabhängig ist, da mit der Menge an möglichen Zuständen die Komplexität und der damit verbundene Rechenaufwand exponentiell ansteigt. Das Spiel Yahtzee beinhaltet eine große Zufallskomponente aufgrund des Werfens mehrerer Würfel und stellt damit ein solches nicht-deterministisches System dar. Trotzdem ist Yahtzee kein reines Glücksspiel, da ein strategisch kluges Vorgehen im beeinflussbaren Teil des Spiels die Chancen auf einen Sieg erhöhen kann. Yahtzee bietet dadurch eine passende Grundlage, auf der sich untersuchen lässt, welche Lernalgorithmen geeignet sind und welche Schritte gegebenenfalls notwendig werden, damit trotz Zufallsfaktor ein stabiles Lernen möglich wird. Das Regelwerk von Yahtzee ist zudem genau definiert und überschaubar, sodass sich eine Softwareumsetzung als Testumgebung für verschiedene Algorithmen problemlos realisieren lässt.

1.2 Zielsetzung

Das Ziel dieser Arbeit soll die Entwicklung eines Systems sein, welches basierend auf Methoden des Reinforcement Learnings trainiert wird und eigenständig lernt Yahtzee zu spielen sowie eine möglichst hohe Punktzahl dabei erzielt. Bei optimaler Durchführung sollte der entstehende Algorithmus im Schnitt eine höhere Punktzahl erreichen können, als ein durchschnittlicher menschlicher Spieler. Es soll ermittelt werden, welche bestehenden Algorithmen für dieses System in Frage kommen und inwieweit der Einsatz von Deep Reinforcement Learning erforderlich ist, um ein lernfähiges System zu entwickeln. Zudem soll ein Verständnis für die Umsetzung im Hinblick auf die in dieser Arbeit behandelten Grundlagen geschaffen sowie dargelegt werden, welche Faktoren den Lernerfolg vorantreiben oder behindern können. Bereits bestehendes Wissen durch bisherige Berührungspunkte mit künstlichen neuronalen Netzen soll zusätzlich erweitert werden. Da Reinforcement Learning auf den ersten Blick wie ein universell einsetzbares Lernwerkzeug wirken kann, soll ebenfalls aufgezeigt werden, ob sich dieser Eindruck bewahrheitet, auf welche Kompromisse eingegangen werden muss und wo es gegebenenfalls unüberwindbare Grenzen gibt.

1.3 Struktur der Arbeit

In dieser Arbeit werden in Kapitel 2 zunächst die Regeln des Spiels Yahtzee sowie die Grundlagen des maschinellen Lernens, neuronaler Netzwerke und insbesondere des Rein-

forcement Learnings behandelt. Zudem werden bisherigen Studien und der aktuelle Stand der Technik aufgezeigt. In Kapitel 3 werden die Anforderungen an das zu entwickelnde System formuliert und die Stakeholder aufgelistet. Das Kapitel 4 beschäftigt sich mit den Schritten, die bei der Entwicklung unternommen werden müssen. Anschließend wird in Kapitel 5 die Entwicklung der virtuellen Umgebung und eines Agenten, der mithilfe eines künstlichen neuronalen Netzes in dieser Umgebung agiert, behandelt. Die Auswertung in Kapitel 6 nimmt Bezug auf die in Kapitel 3 formulierten Anforderungen. Es wird verglichen, welche Punkte erfüllt, respektive nicht erfüllt wurden und auf die jeweiligen Gründe dafür eingegangen. Den Abschluss bildet Kapitel 7 in dem die im Rahmen dieser Arbeit gewonnenen Erkenntnisse zusammengefasst werden und ein Ausblick auf weitere mögliche Vorgehen geboten wird.

2 Grundlagen und aktueller Stand der Technik

2.1 Yahtzee

Yahtzee ist ein Würfelspiel mit fünf sechsseitigen Würfeln und einer Punktekarte. Ziel des Spiels ist es, durch das Erreichen vorgegebener Würfelkombinationen einen höheren Punktestand zu erzielen als die Mitspielenden. Das Spiel wurde in den Vereinigten Staaten ab 1956 von E.S. Lowe vermarktet, welche später durch Milton Bradley (MB) aufgekauft wurde und 1984 wiederum von Hasbro Inc. akquisiert wurde. Grundlegende Teile des Spielkonzepts von Yahtzee existierten schon weitaus früher, eine Ähnlichkeit zu diversen Poker-Kartenspielen, insbesondere zu Würfelpoker, für das es schon Ende des 19. Jahrhunderts eigens angefertigte Würfel gab, ist nicht von der Hand zu weisen. Yahtzee wird, wie Poker oder vergleichbare Kartenspiele, häufig auf den ersten Blick als reines Glücksspiel wahrgenommen, lässt sich aber ob der Möglichkeit strategisch vorzugehen (nämlich durch die Reihenfolge gewählter Punktekategorien, oder durch selektives Neuwürfeln) zwischen Glücks- und Geschicklichkeitsspiel verorten. Die Spielregeln von Yahtzee wurden seit der Erstveröffentlichung nur geringfügig verändert, die im Folgenden behandelte Variante ist die heutzutage am häufigsten anzutreffende.

2.1.1 Spielregeln

Yahtzee wird in der Regel mit zwei bis zehn Personen gespielt, wobei es faktisch keine Begrenzung bei der Spielerzahl gibt. Jede Person erhält zu Anfang des Spiels eine vorgefertigte Punktekarte mit jeweils dreizehn Kategorien und es wird reihum gewürfelt.

Das folgende Schema gibt einen Überblick über den Verlauf jedes Spielzuges:

1. Initialer Wurf mit fünf Würfeln.

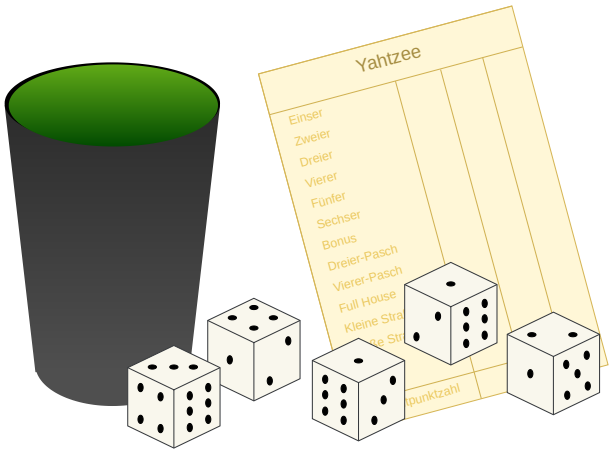


Abbildung 2.1: Yahtzee: Ein Würfelbecher, eine Punktekarte und fünf Würfel

2. Erster Wiederholungswurf beliebiger Würfel (optional).
3. Zweiter Wiederholungswurf beliebiger Würfel (optional).
4. Auswahl der Punktekategorie und Vermerken der resultierenden Punkte.

Die Person die an der Reihe ist, würfelt zunächst mit fünf Würfeln. Ziel ist es, am Ende eines Zuges eine Würfelkombination zu erreichen, die einer der Punktekategorien zugeordnet werden kann, wobei jede Kategorie nur einmal pro Spiel genutzt werden darf. Tabelle 2.1 zeigt exemplarisch die in den Kategorien möglichen Kombinationen und die daraus resultierenden Punkte.

Pro Zug besteht zweimal die Möglichkeit, eine beliebige Auswahl der liegenden Würfel erneut zu werfen. Somit könnte z. B. aus einem doppelten Paar (Wurf: ⚃ ⚃ ⚂ ⚂ ⚅) durch Neuwürfeln des verbleibenden Würfels (⚅) ein Full House (⚃ ⚃ ⚂ ⚂ ⚂) gebildet werden. Kommt am Ende eines Zuges keine der Kombinationen zustande, muss die würfelnde Person eins ihrer Punktefelder mit null Punkten versehen. Innerhalb eines Spiels kann jede der dreizehn Kombination nur einmal auf dem eigenen Punkteblock eingetragen werden, somit endet jedes Spiel nach genau dreizehn Runden. Eine Besonderheit bei der Punkteabrechnung besteht darin, dass bei Erreichen von mindestens 63 Punkten im oberen Block, 35 Punkte zusätzlich als Bonus verrechnet werden. Am Ende des Spiels werden die Punkte aus dem oberen und unteren Block summiert. Die Person mit der höchsten Gesamtpunktzahl gewinnt das Spiel.

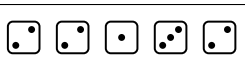
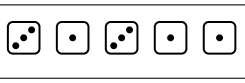
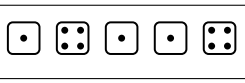
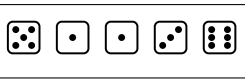
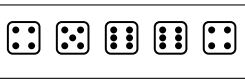
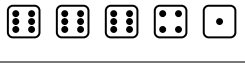
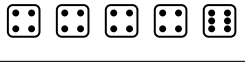
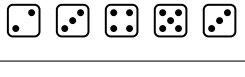
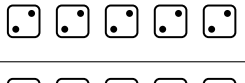
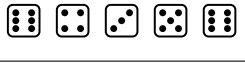
Kombination	Bedingung	Wertung	Beispiel	Punkte
Oberer Block				
Einser	Keine	Augensumme der Einser		3
Zweier	Keine	Augensumme der Zweier		6
Dreier	Keine	Augensumme der Dreier		6
Vierer	Keine	Augensumme der Vierer		8
Fünfer	Keine	Augensumme der Fünfer		5
Sechser	Keine	Augensumme der Sechser		12
Bonus	63 Punkte im oberen Block	35 Punkte		35
Unterer Block				
Dreier-Pasch (Drilling)	Drei gleiche Würfel	Augensumme aller Würfel		23
Vierer-Pasch (Vierling)	Vier gleiche Würfel	Augensumme aller Würfel		22
Full House	Ein Paar und ein Drilling	25 Punkte		25
Kleine Straße	Vier Würfel aufsteigend	30 Punkte		30
Große Straße	Fünf Würfel aufsteigend	40 Punkte		40
Yahtzee (Fünfling)	Fünf gleiche Würfel	50 Punkte		50
Chance	Keine	Augensumme aller Würfel		24

Tabelle 2.1: Exemplarische Würfelkombinationen und Punkte beim Yahtzee-Spiel

2.2 Maschinelles Lernen

Um zu einem besseren Verständnis beizutragen, wird zunächst der Begriff des maschinellen Lernens (ML) im Zusammenhang mit künstlicher Intelligenz (KI) eingeordnet und auf die Grundlagen des ML eingegangen. Neben den englischen Begriffen, wie sie in der Regel auch in Übersetzungen der Fachliteratur zu finden sind, werden in dieser Arbeit teilweise eingedeutschte Bezeichnungen genutzt, sofern sie zu einem klareren Verständnis beitragen.

2.2.1 Begriffseinordnung

Maschinelles Lernen stellt nach dem heutigen Sprachgebrauch einen Teilbereich der künstlichen Intelligenz dar, wobei eine genaue Einordnung nur bedingt möglich ist, da es schon keine einheitliche Definition für Intelligenz gibt. Der Sammelbegriff KI fasst aus heutiger Sicht computerbasierte Systeme zusammen, die Informationen aus Beobachtungen abstrahieren, auf deren Basis Entscheidungen treffen und ihr Verhalten anpassen, um ein bestimmtes Ziel zu erreichen. Maschinelles Lernen beschäftigt sich mit statistischen Lernalgorithmen und mathematischen Modellen, die teilweise auf sehr großen Datenmengen trainiert werden. Die resultierenden Algorithmen folgen nicht mehr einer explizit programmierten Logik, sondern finden sich in abstrakter Form in einem trainierten Modell wieder, welches aus eingegebenen Daten eine Ausgabe errechnet. Innerhalb des Bereichs des maschinellen Lernens wird zwischen verschiedenen Arten des Lernens unterschieden. Die geläufigsten Lernparadigmen sind das Supervised Learning, Unsupervised Learning und das Reinforcement Learning, die jeweils in Abschnitt 2.2.3 beleuchtet werden. Des Weiteren kommen in allen Bereichen, in denen auf großen Datenmengen gelernt wird, mittlerweile künstliche neuronale Netzwerke zum Einsatz, die der Funktionsweise „echter“ neuronaler Netzwerke im Gehirn nachempfunden sind. Diese werden in Abschnitt 2.3 behandelt.

Die Abbildung 2.2 zeigt eine Beispielübersicht zur Einordnung der genannten Bereiche.

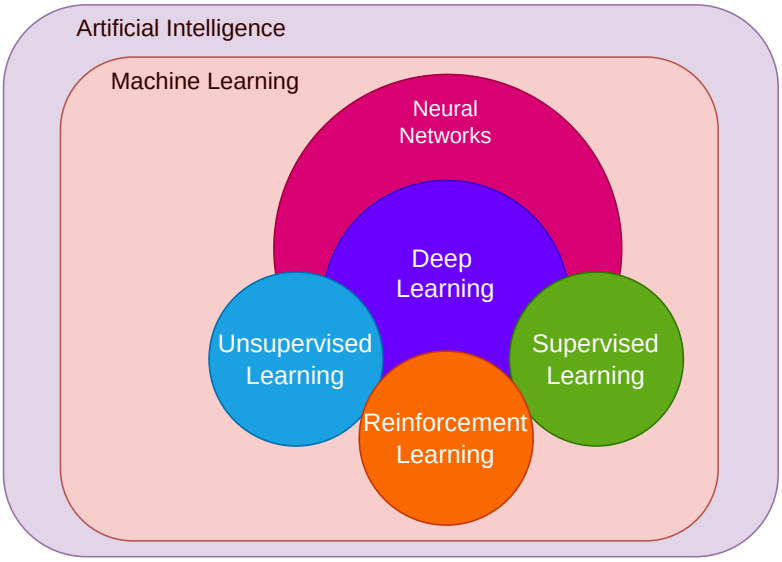


Abbildung 2.2: Bereiche der künstlichen Intelligenz in der Übersicht

2.2.2 Grundlagen des maschinellen Lernens

Markov-Kette

Eine Markov-Kette ist ein stochastischer Prozess, der Anfang des zwanzigsten Jahrhunderts durch den Mathematiker Andrey Markov beschrieben wurde. Sie besteht aus einer endlichen Anzahl an Zuständen, die ausgehend von einem Ausgangszustand in beliebiger Reihenfolge und Schrittzahl durchlaufen werden können. Zustandsübergänge $s \to s'$ sind dabei jeweils durch eine Wahrscheinlichkeit $p_{s,s'}$ definiert, wobei die Summe aller von einem Zustand ausgehenden Übergänge stets $p = 1$ beträgt. Beispielsweise könnte ein Zustand s_0 mit Übergängen in s_1 und s_2 als

$$s_1 \xleftarrow{p=0{,}6} s_0 \xrightarrow{p=0{,}4} s_2$$

dargestellt werden. Es ist ebenfalls möglich, dass ein Zustand auf sich selbst zurückgeführt wird, wobei im Fall $s = s'$ mit $p_{s,s'} = 1$ von einem Endzustand gesprochen wird. Bei der Markov-Kette spricht man von einem Prozess „ohne Gedächtnis“, da vergangene Zustände für die Übergangswahrscheinlichkeiten unerheblich sind.[8, 7]

Markov-Entscheidungsproblem

Das 1957 von Richard Bellman entwickelte Markov-Entscheidungsproblem (engl. *markov decision process*, MDP) basiert auf dem Prinzip der zuvor beschriebenen Markov-Kette und erweitert diese um einige entscheidende Änderungen [5]. Beim MDP sind die Wahrscheinlichkeiten eines Zustandsübergangs nicht inhärent an einen Zustand gebunden, sondern an wählbare Aktionen. Ein Agent wählt eine von mehreren zustandsabhängigen Aktionen aus, wobei jede Aktion zu unterschiedlich wahrscheinlichen Zustandsübergängen führt. Des Weiteren können Zustandsübergänge durch Vergabe oder Abzug von Punkten belohnt oder bestraft werden. Der Agent verfolgt das Ziel, eine Strategie zu entwickeln, nach deren Aktionsauswahl die Belohnung mit der Zeit maximiert wird. Bellman stellte fest, dass sich für einen Zustand s ein erwartbarer optimaler Zustandswert $V^*(s)$ annähern lässt, vorausgesetzt ein Agent handelt optimal. Dies wird als das Bellman'sche Optimalitätsprinzip bezeichnet. Die daraus formulierte Bellman-Gleichung 2.1 beschreibt den Zustandswert $V^*(s)$ als Summe aller folgenden diskontierten Belohnungen, wobei $T\big(s,a,s'\big)$ die Übergangswahrscheinlichkeit und $R(s,a,s')$ die Belohnung, jeweils für Aktion a, darstellen. Über den Diskontierungsfaktor γ lässt sich steuern, wie zeitlich verzögerte Belohnungen gewichtet werden[8].

$$V^*(s) = max_a \sum\nolimits_s T\big(s,a,s'\big)\big[R(s,a,s') + \gamma \cdot V^*(s')\big] \tag{2.1}$$

Q-Value-Iteration-Algorithmus

Der *Q-Value-Iteration*-Algorithmus ist eine von Richard Bellman auf Grundlage der zuvor beschriebenen Bellman-Gleichung entwickelte Berechnungsvorschrift, die den Wert eines Zustand-Aktions-Paares (s,a), zusammengefasst als Qualitäts- oder Q-Wert, annähert.

$$Q_{t+1}(s,a) \leftarrow \sum_{s'} T\big(s,a,s'\big)\Big[R(s,a,s') + \gamma \cdot max_{a'}\ Q_t(s',a')\Big] \tag{2.2}$$

Q-Learning

Das Q-Learning ist eine Lernmethode aus dem Bereich des Reinforcement Learnings (Abs. 2.4) und eine Weiterentwicklung des in Gleichung 2.2 dargestellten *Q-Value-Iteration*-Algorithmus. Q-Learning lässt sich dabei entgegen des vorangegangenen Algorithmus

auch bei Umgebungen mit anfänglich unbekannten Belohnungen und Übergangswahrscheinlichkeiten benutzen. Der Q-Learning-Algorithmus macht sich Bellmans Optimalitätsprinzip (Abs. 2.2.2) zunutze um Q-Werte über die Summe aller zukünftigen diskontierten Belohnungen anzunähern. Schon 1992 konnte mathematisch bewiesen werden, das dieser Algorithmus beim Vorliegen eines endlichen Markov-Entscheidungsproblems (MDP) gegen optimale Q-Werte konvergiert [18]. Für die Anwendung von Q-Learning ist kein Modell der Umgebung notwendig, weswegen auch von einem modellfreien Algorithmus die Rede ist. Stattdessen wird eine Q-Tabelle (englisch *Q-table*) zur Speicherung der berechneten Q-Werte genutzt. Diese bildet üblicherweise zeilenweise die möglichen Zustände S und spaltenweise die Aktionen A ab und wird dabei initial mit Zufallswerten befüllt. Um die Tabelle zu aktualisieren führt ein Agent z. B. zufällige Aktionen an einer Umgebung aus. Die Q-Werte werden dann durch die erhaltenen Rewards nach folgender Gleichung berechnet [1]:

$$Q\big(S_t, A_t\big) = Q\big(S_t, A_t\big) + \alpha\Big[R_{t+1} + \gamma \cdot \max Q\big(S_{t+1}, a\big) - Q\big(S_t, A_t\big)\Big] \tag{2.3}$$

Die Variablen dieser Gleichung werden im Folgenden erläutert:

- **Lernrate $\boldsymbol{\alpha}$** : Die Lernrate $\alpha \in [0, 1] \subset \mathbb{R}$ definiert zu welchem Grad ein neuer Q-Wert den alten Q-Wert anpasst. Ein Wert $\alpha = 0$ führt hierbei zu keinem Lerneffekt, während $\alpha = 1$ den alten Wert komplett überschreibt.

  In der Praxis wird für die Lernrate in nicht-deterministischen Umgebungen ein fixer Wert von $\alpha = 0,1$ gewählt. Somit erhalten Aktionen, die wiederholt gut belohnt werden, mit der Zeit eine bessere Bewertung [1]. In deterministischen Umgebungen kann folglich $\alpha = 1$ gesetzt werden.

- **Reward $\boldsymbol{R}$**: Der Reward $R \in \mathbb{R}$(hier R_{t+1}) gibt die Belohnung für die Ausführung von Aktion A_t in Zustand S_t an.

- **Diskontierungsfaktor $\boldsymbol{\gamma}$**: Mit dem Diskontierungsfaktor $\gamma \in [0, 1] \subset \mathbb{R}$ lässt sich das Verhalten eines Agenten in Bezug auf das Streben nach kurz- oder langfristigen Belohnungen steuern. Ein Wert von $\gamma = 1$ lässt den Agenten Belohnungen bevorzugen, die unmittelbar auf eine Aktion folgen. Werte $0 \leq \gamma < 1$ sorgen abnehmend für eine Priorisierung künftiger Rewards.

- **$\mathbf{maxQ\big(S_{t+1}, a\big)}$**: Dieser Wert ist der maximale Q-Wert aller in Folgezustand S_{t+1} möglichen Aktionen.

Da der Algorithmus auf zukünftige Q-Werte zugreift, pflanzt sich eine Änderung des Q-Werte mit jeder Iteration von hinten nach vorne durch die Tabelle durch. Durch viele Iterationen ist es somit möglich, die optimalen Q-Werte anzunähern. Diese Lernmethode eignet sich damit für präzise Vorhersagen in Umgebungen mit einer überschaubaren Anzahl an Zuständen [8].

Deep-Q Learning

Deep-Q-Learning überträgt das Konzept des Q-Learnings auf den Bereich des Deep Learnings (Abs. 2.3.4), indem tiefe, vielschichtige neuronale Netze, sogenannte Deep-Q-Netzwerke (DQN), eingesetzt werden. Durch einen angepassten Trainingsalgorithmus ist es damit möglich, die Q-Werte als Ausgangsfunktion des DQN anzunähern [1].

Epsilon-Greedy-Strategie

Die Epsilon-Greedy-Strategie ist ein über einen Parameter Epsilon (ϵ) gewichtetes Auswahlverfahren. Es wird genutzt um das Verhältnis zwischen „Ausprobieren“ unbekannter und „Ausnutzen“ bereits bekannter Aktionen (engl. *exploration and exploitation*) einzustellen. Dabei wird der Parameter $\epsilon \in [0, 1] \subseteq \mathbb{R}$ mit einem Zufallswert $x \in [0, 1] \subseteq \mathbb{R}$ verglichen. Abhängig vom Ergebnis lässt sich eine der beiden Aktionskategorien auswählen, bspw. eine Zufallsaktion (Ausprobieren) im Fall $\epsilon > x$.
In der Regel wird ein Modell anfänglich mit Zufallsaktionen trainiert, da noch keine Erfahrungen zu guten oder schlechten Aktionen existieren. Mit voranschreitendem Training können bessere Vorhersagen gemacht werden, sodass eine Zufallsauswahl nicht mehr nötig ist. Wird die Epsilon-Greedy-Strategie im Trainingsprozess verwendet, so wird häufig ein initialer Wert nahe 1 für ϵ verwendet und dieser dann schrittweise dekrementiert.

Softmax-Auswahl

Softmax ist ein Auswahlverfahren mit dem sich Vektoren an Aktions-Werten, respektive Q-Werten, in eine Wahrscheinlichkeitsverteilung im Wertebereich $(0, 1)$ überführen lassen. Die Summe der resultierenden Wahrscheinlichkeiten beträgt dabei stets 1. Die

folgende Gleichung mit Vektor A und Anzahl an Elementen n verdeutlicht die Funktionsweise:

$$\text{softmax}(A_i) = \frac{e^{A_i}}{\sum_{j=1}^{n} e^{A_j}} \tag{2.4}$$

Durch die Exponentialfunktion werden höhere Aktions-Werte stärker betont, während sie über die Summe aller Exponenten wieder in Verhältnis zueinander gebracht werden. Das Softmax-Verfahren wird häufig auf die Ausgangswerte (*Logits*) neuronaler Netze angewandt. Dadurch bietet sich die Möglichkeit, die Wahrscheinlichkeiten der besten Aktionen gegeneinander abzuwägen und ggfls. zwischen mehreren gut bewerteten Aktionen eine engere Auswahl zu treffen.

Monte-Carlo-Algorithmen

Die Monte-Carlo-Algorithmen fassen eine Reihe an Algorithmen zusammen, die im weitesten Sinne Zufallsergebnisse produzieren. Demnach liefert ein solcher Algorithmus sowohl richtige, als auch falsche Ergebnisse. Durch das Gesetz der großen Zahlen ist es dennoch möglich, eine Näherung eines korrekten Ergebnisses zu erreichen, indem eine große Anzahl Iterationen und ein kontinuierlicher Vergleich zwischen Vorhersage und Zielergebnis durchgeführt wird [7, 1].

2.2.3 Arten des maschinellen Lernens

Im Folgenden werden einige typische Lernparadigmen des Machine Learnings behandelt.

Überwachtes Lernen (Supervised Learning)

Supervised Learning (SL) ist ein Lernparadigma, welches das Lernen anhand beschrifteter Datensätze ermöglicht. Ein SL-Algorithmus wird trainiert, indem er für Eingabedaten eine Vorhersage erzeugen soll. Diese Eingabedaten haben eine Beschriftung (engl. *label*), das heißt, ihnen ist ihre Lösung angehängt. Bei einem Bild z. B würde das Label angeben, was auf dem Bild zu sehen ist. Durch einen Vergleich zwischen Vorhersage und Lösung, kann der Algorithmus seine zukünftigen Vorhersagen anpassen und verbessern. Diese Modelle eignen sich unter anderem für die Bilderkennung und Kategorisierung, da sie prädestiniert sind, wiederkehrende Muster in den Trainingsdaten zu identifizieren [8].

Unüberwachtes Lernen (Unsupervised Learning)

Beim Unsupervised Learning werden, im Gegensatz zum Supervised Learning, keine bereits beschrifteten Daten für das Lernen verwendet. Stattdessen versucht der Algorithmus, sich wiederholende Muster in den Daten zu erkennen und diese zu gruppieren. Eine der häufigsten Aufgabenbereiche für Unsupervised Learning ist daher die Clusteranalyse oder das Finden von Anomalien auf unstrukturierten Datensätzen [8].

Reinforcement Learning

Reinforcement Learning (RL) ist das eigenständige Erlernen einer Umgebung auf Basis von Beobachtungen und Belohnungen für ausgeführte Aktionen. Als Schwerpunkt dieser Arbeit wird RL in Abschnitt 2.4 detaillierter behandelt.

Weitere Arten

Neben dem überwachten und unüberwachten Lernen gibt es noch hybride Formen wie das semiüberwachte Lernen (engl. *weak supervision*), bei dem unüberwachtes Lernen genutzt wird um die Daten zu clustern, während überwachtes Lernen die entstehenden Cluster nach bestehenden Labels klassifiziert. Eine weitere Mischform ist das selbstüberwachte Lernen (engl. *self-supervised learning, SSL*), das ohne Beispieldaten lernt und eigenständig Labels für Trainingsdaten erstellt. Diese Form findet unter anderem Anwendung beim Training von Sprachmodellen wie OpenAIs *GPT-3* und *GPT-3.5* [19] und bei einem Spracherkennungsmodell der Firma Facebook(Meta) [4].

2.3 Künstliche neuronale Netzwerke

Künstliche neuronale Netze (KNN, engl. *artificial neural networks*) stellen einen der Hauptbestandteile heutiger Algorithmen aus dem Bereich des maschinellen Lernens dar. Ihre Funktion basiert auf untereinander verknüpften künstlichen Neuronen, die im weitesten Sinne den Neuronen (Nervenzellen) eines Gehirns nachempfunden sind. Im Gehirn sind Neuronen dafür zuständig, gemäß ihres Erregungszustands elektrische Signale an das zentrale Nervensystem weiterzuleiten. Die Kommunikation zwischen Neuronen, von denen es im menschlichen Gehirn schätzungsweise 86 Milliarden [3] gibt, ermöglicht es

äußere Reize zu verarbeiten und aus ihnen zu lernen.
Im Rahmen dieser Arbeit werden neben der Abkürzung KNN die Formulierungen „künstliches neuronales Netz“ sowie „neuronales Netz“ und „Netz“ synonym verwendet. Gleiches gilt für die Bezeichnungen „künstliches Neuron“ und „Neuron“. Im englischen Sprachgebrauch hat sich teilweise auch der Begriff *unit* (zu deutsch: „Einheit“) anstelle von *neuron* etabliert, da heutige Modelle faktisch nur die Grundzüge echter Nervenzellen imitieren [8].

2.3.1 Geschichte

Das erste künstliche Neuron wurde 1943 von den US-amerikanischen Wissenschaftlern McCulloch und Pitts vorgestellt und ist als *threshold logic unit* oder *TLU* bekannt [12]. Die TLU besteht aus vier Komponenten, deren Funktionsweise in ähnlicher Form auch in heutigen KNNs zu finden ist:

- Eingangs-Signale: x_n
- Gewichtungen: w_n
- Einheit mit Schwellenwert-Logik: step($\mathbf{x}^T\mathbf{w}$)
- Ausgangs-Signal: h_w

Abbildung 2.3 veranschaulicht die durch Gleichung 2.5 beschriebene TLU:
Zuerst werden die Eingangssignale x_n mit den zugehörigen Gewichten w_n multipliziert. Aus den so gewichteten Eingangssignalen wird die Summe $z = \mathbf{x}^T\mathbf{w} = (\sum_{n=1}^{n_{max}} x_n w_n)$ gebildet. Die Schwellenwertfunktion setzt abschließend den Ausgang $h_w(x) = 1$ für $z \geq 0$, oder $h_w(x) = 0$ für $z < 0$ [8].

$$\text{TLU:} \quad h_w(x) = step(\mathbf{x}^T\mathbf{w}) \tag{2.5}$$

Durch diese Architektur stellt das TLU ein primitives künstliches Neuron dar, welches in Abhängigkeit der gewichteten Eingangssignale aktiv oder inaktiv geschaltet wird.
Der US-amerikanische Psychologe und Informatiker Frank Rosenblatt entwickelte die Idee der TLU 1957 zum *Perzeptron* weiter [16]. Ein Perzeptron besteht aus einer Ausgabeschicht aus mehreren TLUs und einer Eingabeschicht mit mehreren gewichteten Eingängen. Die Besonderheit hierbei ist die Verknüpfung jedes Eingangssignals mit jeder

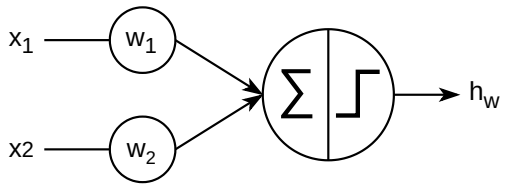


Abbildung 2.3: Darstellung der threshold logic unit (TLU)

einzelnen TLU und die damit erreichte Komplexität. Als zusätzliche Erweiterung verfügt eine Schicht über ein *Bias*-Neuron, mit welchem die gewichteten Eingabewerte linear angepasst und somit das Verhalten gegenüber dem Schwellenwert verbessert werden kann, beispielsweise bei schwachen Eingangssignalen.

Die Grundzüge des Perzeptrons, z. B. die volle Vernetzung der Neuronen zweier Schichten, oder die Verwendung von Bias-Neuronen, sind noch bis heute in neuronalen Netzen erkennbar. Mit der Zeit wurde die Funktionalität jedoch stark erweitert und spezialisiert.

2.3.2 Funktionsweise

Den Kern neuronaler Netze bilden künstliche Neuronen (KN). Wie im menschlichen Gehirn sind diese ebenfalls miteinander verbunden, KNs werden aber explizit definierten Schichten zugeordnet. In einer einfachen Form besteht ein Netz mindestens aus einer Ein- und Ausgabeschicht, die jeweils mit einer festen Anzahl KNs initialisiert werden. In der Regel ist jedes Neuron mit jedem Neuron der nachfolgenden Schicht verbunden, wobei es je nach Umsetzung (siehe 2.3.3) auch Abweichungen geben kann. Neben den regulären Neuronen gibt es, ähnlich wie beim Perzeptron (Abs. 2.3.1), Bias-Neuronen, die eine lineare Anpassung der Signale ermöglichen. Komplexität und Leistungsfähigkeit eines Netzes können durch zusätzliche Schichten (siehe Abs. 2.3.4) oder eine größere Anzahl an Neuronen gesteigert werden. Die Verbindungen zwischen den Neuronen werden als Gewichte (engl. *weights*) bezeichnet. Sie stellen den veränderlichen Teil des Netzes dar, der durch kontinuierliches Training angepasst werden kann und dadurch das Lernen ermöglicht. Abbildung 2.4 zeigt den schematischen Aufbau eines Netzes mit drei Neuronen in der Eingabe- und zwei Neuronen in der Ausgabeschicht, wobei die Bias-Neuronen der Schichten nicht mit dargestellt sind.

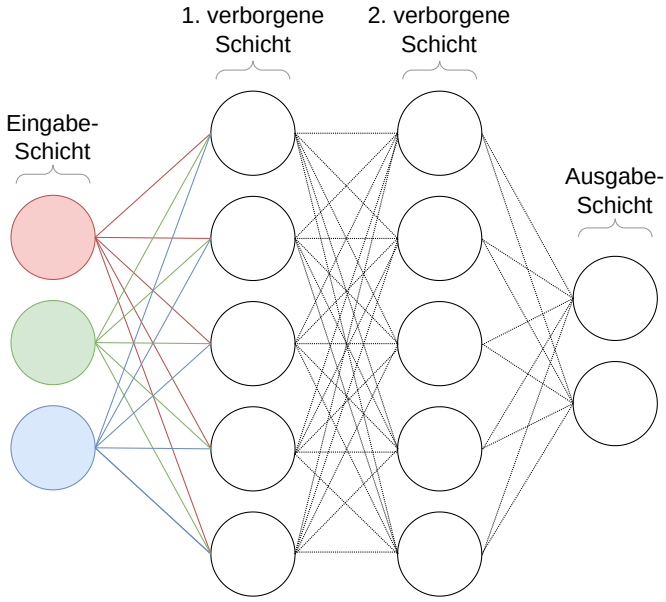


Abbildung 2.4: Schema eines neuronalen Netzes mit zwei verborgenen Schichten

Ein initialisiertes neuronales Netz kann auf zwei Arten genutzt werden:

1. **Training**: Die Gewichte des Netzes sind veränderlich. Sie werden im besten Fall so angepasst, dass das Netz mit jedem Trainingsschritt ein besseres Ergebnis für eine Eingabe liefert.

2. **Vorhersage**: Die Gewichte des Netzes sind fix. Eingegebene Daten werden durch das Netz geführt und erzeugen eine Vorhersage in Form der Ausgabeschicht.

2.3.3 Arten künstlich neuronaler Netzwerke

Grundsätzlich kann zwischen verschiedenen Arten von neuronalen Netzen unterschieden werden, wobei sich diese maßgeblich in der Art und Weise unterscheiden, wie die Daten durch das Netz geführt und verarbeitet werden. Beliebte Vertreter, die sich für je unterschiedliche Einsatzzwecke eignen und im Folgenden kurz beleuchtet werden sind z. B. Netze vom Typus *vorwärtsgerichtet*, *rekurrent*, oder *faltend*.

Vorwärtsgerichtete Netzwerke (FNN)

Vorwärtsgerichtete Netzwerke (engl. *feedforward neural network*, FNN) sind die klassische Form eines künstlichen neuronalen Netzes. Eine Eingabe wird auf direktem Weg durch alle Schichten des Netzes geführt, jedes Neuron nimmt jeweils nur Einfluss auf Neuronen der nachfolgenden Schicht.

Rekurrente Netzwerke (RNN)

Rekurrente Netzwerke (engl. *recurrent neural network*, RNN) haben die Besonderheit, dass KNs nicht Schichtweise miteinander verbunden sein müssen. Ein KN kann beispielsweise eine Verbindung zu einem KN einer vorangegangenen Schicht (indirekte Rückkopplung), zu einem anderen KN der gleichen Schicht (seitliche Rückkopplung), oder oder zu sich selbst (direkte Rückkopplung) haben. Netze von diesem Typ werden in der Praxis häufig verwendet, wenn sequentielle Daten ausgewertet werden sollen, zum Beispiel bei der Spracherkennung oder dem Erkennung von Handschrift [6].

Faltende Netzwerke (CNN)

Faltende- oder Faltungsnetzwerke (engl. *convolutional neural network*, CNN) nutzen auf oberster Ebene sogenannte Faltungsschichten. In diesen werden kleinerer Faltungsmatritzen schrittweise über die Eingabedaten bewegt und mittels diskreter Faltung die Ausgabe jedes Neurons berechnet. Der Vorteil bei dieser Technik ist die daraus entstehende Überlappung zwischen den Eingangsdaten aufeinanderfolgender Schritte, da hierdurch die Aktivität benachbarter Neuronen teilweise aneinander gekoppelt wird. Das Prinzip ähnelt der Bildverarbeitung im Gehirn von Menschen und Tieren, die dadurch in der Lage sind Muster zu erkennen. In der Praxis werden CNNs deshalb vorzugsweise zur Bild- uns Gesichtserkennung eingesetzt [22].

2.3.4 Begriffsdefinitionen

Vorwärtspropagierung (*Forward propagation*)

Die Vorwärtspropagierung bezeichnet die vorwärts gerichtete Führung von Informationen durch ein neuronales Netz. Sie beschreibt den Prozess, in dem jede Information von der

Eingabeschicht des Netzes ausgehend eine Gewichtung erfährt, von der Aktivierungsfunktion einer Schicht verarbeitet wird und anschließend den Ausgang eines Neurons bildet (vgl. *TLU* Abs. 2.3). Je nach Anzahl der Schichten wird dieser Prozess wiederholt fortgesetzt, bis die Ausgangsschicht erreicht wird. Die Vorwärtspropagierung ist der Hauptmechanismus, durch den ein Netz aus einer Eingabe eine Vorhersage erzeugt.

Rückpropagierung (*Back propagation*)

Die Rückpropagierung (auch *Fehlerrückführung*) ist ein wichtiger Teil des Trainingsprozesses eines neuronalen Netzes, da sie für die Adaptierung der Gewichte zwischen den Schichten benötigt wird. Damit ein Netz lernfähig ist, muss nach einer Vorwärtspropagierung der mittels Verlustfunktion ermittelte Fehler zwischen Zielwert und Vorhersage auf die Gewichte zurückgeführt werden. Die Rückpropagierung ermittelt den Einfluss eines Gewichts auf den entstandenen Fehler und passt das Gewicht mittels Gradientenverfahren an [15].

Verlustfunktion (Fehlerfunktion)

Die Verlustfunktion (engl. *loss function* oder *cost function*), auch Fehlerfunktion genannt, wird genutzt, um den Fehler zwischen einem vom Modell vorhergesagten Wert und dem erwarteten Wert zu ermitteln, bzw. zu quantifizieren. Durch den aktuellen durchschnittlichen Fehler lässt sich auch eine Aussage über die Genauigkeit des Modells bezüglich seiner Vorhersagen treffen. Der Fehlerwert wird bei der Rückpropagierung genutzt, um die Gewichte des Netzes so anzupassen, dass der Fehler mit der Zeit minimiert wird. Bei Regressionsproblemen, also bei kontinuierlichen Wertespektren, wird als Verlustfunktion meist die mittlere quadratische Abweichung (engl. *mean squared error*, MSE) genutzt, da sie einen richtungsunabhängigen Fehler berechnet [8].
Weitere bekannte Verlustfunktionen für Regressionsprobleme sind:

- MAE: *mean absolute error*
- RMSE: *root mean squared error*
- MSLE: *mean squared logarithmic error*

Bei Klassifizierungsaufgaben werden hingegen Kreuzentropie-Verlustfunktionen wie

- BCE: binary cross-entropy loss

- CCE: categorical cross-entropy loss

genutzt, wenn das Modell eine Vorhersage zur Zugehörigkeit zu einer von zwei Klassen (BCE), oder zu einer von mehreren Klassen (CCE) ausgibt [15]. In der Regel ist die Wahl der optimalen Verlustfunktion stark vom Fehlerverhalten der Umgebung und vom eingesetzten Modell abhängig.

Aktivierungsfunktion

Die Aktivierungsfunktion ist Bestandteil jedes Neurons und bestimmt sein Ausgangssignal im Verhältnis zum gewichteten Eingangssignal. Die beim TLU und Perzeptron (Abs. 2.3.1) genutzte Schwellenwertfunktion (Abb. 2.5) ist eine simple Aktivierungsfunktion, hat heutzutage aber keinen großen Stellenwert mehr im Kontext neuronaler Netze. Zum einen kann sie nur zwei mögliche Zustände abbilden und ist damit sehr unflexibel, zum anderen ist sie nicht differenzierbar und damit ungeeignet für das Gradientenabstiegsverfahren (Abs. 2.3.4), welches eine der wichtigsten Werkzeuge zum Trainieren moderner neuronaler Netze ist. Dieser Umstand hat eine ganze Reihe heutiger Aktivierungsfunktionen mit verschiedenen Vor- und Nachteilen hervorgebracht, von denen im Folgenden die Funktionen *Sigmoid*, *ReLu*, *Leaky-ReLu* und *ELU* (jeweils dargestellt in Abb. 2.6) genauer betrachtet werden [15, 8].

- **Sigmoid**: Wird bei einfachen neuronalen Netzen eingesetzt, da sie durchgehend differenzierbar ist. Bei extremen Eingangswerten nimmt die Differenzierbarkeit aufgrund des flachen Verlaufs jedoch stark ab, wodurch es zu einem Verschwinden der Gradienten (Abs. 2.3.4) kommen kann.
  Die Sigmoidfunktion wird berechnet durch $f(x) = \left(1 + e^{-x}\right)^{-1}$.

- **ReLu** (Rectified Linear Unit): Wird für große neuronale Netze genutzt, da sie durch ihre sehr simple Berechnung Rechenzeit spart. Durch ihren linearen Anstieg hegt sie nicht die Gefahr für verschwindende Gradienten. Ein möglicher Nachteil von ReLu ist, dass Werte $x < 0$ immer Null als Ausgabe liefern, was in manchen Konstellationen zu „toten“ Neuronen führen kann, Neuronen die auf einem Wert steckenbleiben und keine Änderung mehr erfahren.
  Die ReLu-Funktion wird berechnet durch $f(x) = max(0, x)$.

- **Leaky-Relu**: Leaky ReLu ist eine optimierte Form der ReLu-Funktion, die das Problem toter Neuronen verhindern soll. Durch einen einstellbaren Parameter α,

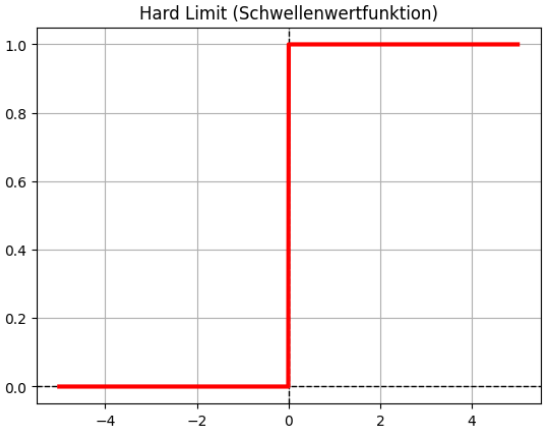


Abbildung 2.5: Die Schwellenwertfunktion

der die Steigung für Eingangswerte $x < 0$ bestimmt, können auch negative Funktionswerte genutzt werden.
Die Leaky-ReLu-Funktion wird berechnet durch $f(x) = max(\alpha x, x)$.

- **ELU** (Exponential Linear Unit): Die ELU-Funktion ist eine weitere Optimierung der (Leaky-)ReLu-Funktion. Wie Leaky-ReLu soll ELU ebenfalls die Entstehung toter Neuronen verhindern. Infolge des exponentiellen Verlaufs für für $x < 0$ ist die ELU-Funktion etwas rechenintensiver, soll aber das Lernen in einigen Fällen verbessern [1].
Die ELU-Funktion wird berechnet durch $f(x) = max(\alpha(e^x - 1), x)$.

Optimierung mittels Gradientenverfahren

Das Gradientenverfahren, auch Gradienten*abstiegs*verfahren, wird im Trainingsprozess dazu eingesetzt die Fehlerfunktion eines neuronalen Netzen zu minimieren und damit zukünftige Vorhersagen zu verbessern. Die Funktion differenziert die Verluste, ermittelt also den Anstieg der Fehlerfunktion an der aktuellen Stelle. Anschließend wird ein Schritt entgegen des Anstiegs unternommen, also in Richtung des Minimums. Die Schrittweite lässt sich dabei durch die Lernrate (engl. *learning rate*) einstellen. Wird eine zu große Lernrate gewählt, kann es sein, dass das anvisierte Minimum immer wieder übersprungen wird. Die Folge ist ein instabiles Lernverhalten [15].

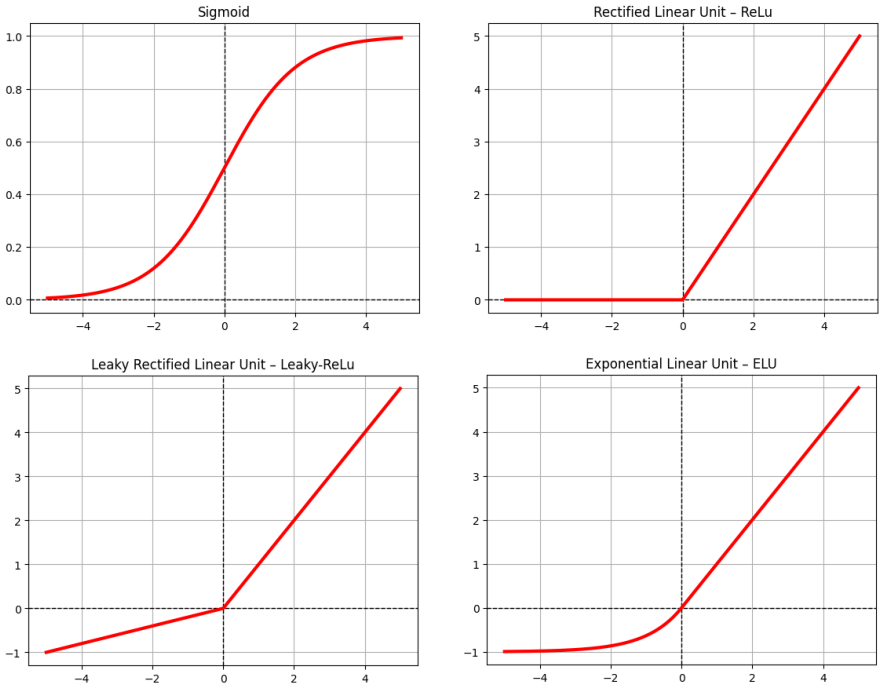


Abbildung 2.6: Die Aktivierungsfunktionen Sigmoid, ReLu, Leaky-ReLu und ELU

Verschwindende Gradienten

Ein häufiges Problem das bei neuronalen Netzen auftreten kann, sind die verschwindenden Gradienten (engl. *vanishing gradients*). Darunter versteht man die Abnahme von Gradienten, beispielsweise aufgrund falsch gewählter Aktivierungsfunktionen. Die Folge sind Gradienten, die kaum noch einen Einfluss auf das Netz haben, wodurch das Lernen ins Stocken kommen kann [15].

Über- und Unteranpassung

Bei einer Überanpassung (engl. *overfitting*) eines neuronalen Netzes findet eine zu starke Anpassung an die Trainingsdaten statt. Statt relevanten Details fängt das Netz an, Muster im Rauschen zu erkennen, wodurch der Lernfortschritt wieder abnimmt. Überanpassung kann auf verschiedene Faktoren zurückzuführen sein, z. B. einen zu kleinen Trainingsdatensatz, eine zu lange Trainingsdauer oder ein Modell mit einer zu großen Anzahl an Parametern. Eine Unteranpassung (engl. *underfitting*) tritt auf, wenn über

eine zu kurze Zeit trainiert wurde, oder ein Modell über zu wenige Parameter verfügt um zu lernen [15].

Hyperparameter

Als Hyperparameter werden Parameter bezeichnet die vor dem Training eines Lernalgorithmus definiert werden und diesen steuern. Bei einem neuronalen Netz sind das beispielsweise die Anzahl an Schichten und zugehörigen Neuronen, oder auch die Lernrate. Die Optimierung dieser Parameter, die Hyperparameteroptimierung, ist ein Vorgehen um die beste Kombination an Hyperparametern zu finden. Hierfür wird das Training mit verschiedenen Parameterkombinationen durchgeführt um die performanteste Einstellung zu ermitteln.

Deep Learning

Unter Deep Learning wird ein Teilgebiet des maschinellen Lernens bezeichnet, bei dem vielschichtige neuronale Netze zum Einsatz kommen. Das „Tief“ (engl. *deep*) in der Bezeichnung bezieht sich auf die Verwendung verdeckter Schichten (engl. *hidden layers*), zwischen Ein- und Ausgabeschicht des Netzes und die dadurch entstehenden tiefen Verzweigungen. Die dadurch entstehenden Strukturen ermöglichen eine weitaus komplexere Interpretation der eingegebenen Daten, als es mit einem konventionellen Netz möglich ist. Ab welcher Anzahl verdeckter Schichten ein Netz in den Bereich des Deep Learnings eingeordnet werden kann ist jedoch nicht einheitlich definiert.

On- und Offline-Lernen

Beim Training von neuronalen Netzen wird grundsätzlich zwischen On- und Offline-Lernen unterschieden. Findet ein Lernen „Online“ statt, so ist damit gemeint, dass ein Netz Vorhersagen produziert, anhand derer es im gleichen Prozess trainiert wird. Beim Offline-Lernen wird das Netz entweder anhand vorgefertigter Daten trainiert, oder es wird eine Kopie des Netzes genutzt, welche die Vorhersagen produziert während nur das Ursprungs-Netz angepasst wird[1]. Letztere Vorgehensweise findet sich z. B. bei Target-Networks, bei denen zwei Netze zu festen Intervallen synchronisiert werden. Der Vorteil beim Offline-Learning ist die geringere Anfälligkeit zu Schwankungen und dadurch ein

stabileres Training. Gleichzeitig stellt der doppelte Speicherbedarf im Fall von Target-Networks einen großen Nachteil dar, was besonders bei Systemen mit geringer Hardwareleistung zum Tragen kommt [8].

2.4 Reinforcement Learning

Der Begriff Reinforcement Learning (RL), zu deutsch be- oder verstärkendes Lernen, fasst Methoden aus dem Bereich des maschinellen Lernens zusammen, die sich das Prinzip der operanten Konditionierung zunutze machen, um eine Aufgabe zu lösen. Demnach wird ein Verhalten, ähnlich wie es bei Tieren zu beobachten ist, durch positive Reize verstärkt oder durch negative Reize abgeschwächt. RL-Algorithmen eignen sich damit für dynamische Umgebungen, da sie im Vergleich zu anderen Lernparadigmen, wie dem Supervised Learning, durch eigenes Handeln ein gewünschtes Verhalten erlernen können.

2.4.1 Geschichte

Zwar wurde Reinforcement Learning schon seit den 1950er Jahren diskutiert und erforscht, hat aber mit dem Aufkommen potenter Hardware und der öffentlichen Verfügbarkeit von Frameworks zur Arbeit mit neuronalen Netzen, deutlich an Bedeutung gewonnen. Mitarbeitende der 2010 gegründeten Firma DeepMind, veröffentlichten 2013 eine Publikation, in der sie einen Reinforcement-Learning-Algorithmus präsentierten, der die Vorteile von Deep-Learning und Q-Learning kombinierte [13]. Dieser als *Deep-Q-Learning* bezeichnete Ansatz ermöglichte es dem Algorithmus, 49 bekannte Videospiele für den Atari 2600 eigenständig spielen zu lernen und darüber hinaus sogar menschliche Spieler zu übertreffen. Ein weiterer Meilenstein von DeepMind, die 2014 von Google LLC aufgekauft wurden, stellt die Entwicklung des Programms AlphaGo dar. AlphaGo, welches dafür konzipiert wurde das chinesische Brettspiel Go zu spielen, schlug 2016 in vier aus fünf Partien den südkoreanischen Profispieler Lee Sedol, der bis dahin als einer der besten Spieler weltweit galt [1]. Das Programm wurde durch eine Kombination aus RL, überwachtem Lernen und eine Monte-Carlo-Baumsuche trainiert. Auf Basis dieses Ansatzes folgten die Weiterentwicklungen AlphaZero (2017), welches in der Lage ist Schach, Go und Shōgi zu spielen sowie AlphaStar (2019), das das Videospiel Starcraft II

[1] Mensch gegen Maschine 1:4 – AlphaGo gewinnt auch das letzte Spiel: `https://www.heise.de/news/Mensch-gegen-Maschine-1-4-AlphaGo-gewinnt-auch-das-letzte-Spiel-3135188.html`, Aufgerufen: 27.04.2025

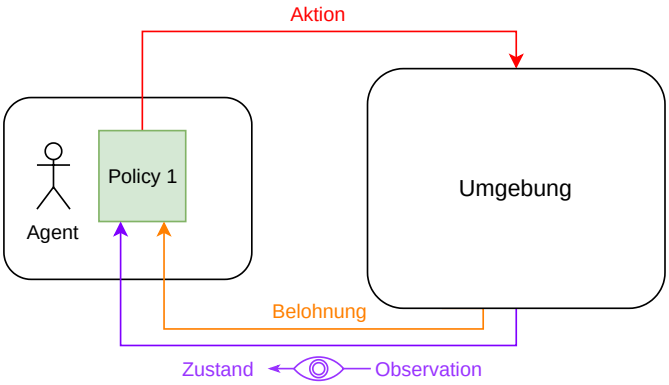


Abbildung 2.7: Reinforcement Learning in einer schematische Darstellung

beherrscht. Anfang 2025 veröffentlichte das chinesische Startup *DeepSeek* einen Bericht zu ihrem Large Language Model *DeepSeek-R1* [2], Teile dessen ebenfalls via RL trainiert wurden.

2.4.2 Funktionsweise

Die Anwendung von RL-Algorithmen erfordert die Bereitstellung einer Umgebung, möglicher darin ausführbarer Aktionen und erhaltbarer Belohnungen für das Erreichen eines oder mehrerer Ziele. Der Algorithmus hat die Aufgabe Aktionen an der Umgebung auszuführen und diese nach einer Zustandsänderung zu beobachten. Führen Aktionen wiederholt zu einem gewünschten Verhalten, z. B. zum Erreichen eines Ziels, wird eine Belohnung in Form von Punkten vergeben. Durch das übergeordnete Ziel, die mögliche Punktzahl zu maximieren, lässt sich so durch kontinuierliche Wiederholung dieses Ablaufs ein eigenständig gesteuertes Lernverhalten realisieren [8]. In Abbildung 2.7 wird das Schema dieses Prozesses noch einmal verdeutlicht. Im Folgenden wird ein einfaches Anwendungsbeispiel beschrieben. Anschließend werden die wichtigen Komponenten des Reinforcement Learnings hervorgehoben.

Anwendungsbeispiel

Als anschauliches Beispiel für die Anwendung von RL kann ein Roboter betrachtet werden, welcher Kisten in ein Regal einsortieren soll. Der Roboter hat anfangs keine Informa-

[2] DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning: https://github.com/deepseek-ai/DeepSeek-R1/blob/main/DeepSeek_R1.pdf

tion darüber, nach welchen Kriterien dieser Sortiervorgang erfolgen soll. Um die Kisten zu bewegen, hat er eine vordefinierte Reihe an möglichen Bewegungsabläufen bzw. Aktionen. Der Roboter beginnt seine Aufgabe damit, die Kisten in willkürlicher Anordnung im Regal abzulegen. Wird eine Kiste zufällig richtig einsortiert, erhält der Roboter eine Belohnung. Da der Roboter das Ziel verfolgt, möglichst viele Belohnungen zu sammeln, bevor ein Sortiervorgang beendet ist, lernt er indirekt, nach welcher Strategie die Kisten korrekt einzusortieren sind.

Environment (Umgebung)

Die Environment bezeichnet eine reale oder virtuelle Umgebung, die als ein geschlossenes System betrachtet werden kann, welches Ein- und Ausgänge zur Interaktion zur Verfügung stellt. Über die Eingänge können Aktionen ausgeführt werden, die den Inhalt oder das Verhalten der Environment beeinflussen. Eine Analogie wäre ein Kasten, dessen Innenleben über außenliegende Hebel oder Knöpfe gesteuert werden kann. Über die Ausgänge der Environment werden deren aktuelle Zustände, beziehungsweise Beobachtungen (engl. *observations*) abgerufen. Die Analogie hierzu wäre z. B. eine Kamera, deren Sensor eine Umgebung erfasst und das entstehende Bild zur Auswertung weitergibt. Führt eine Aktion nun zu einem gewünschten Verhalten oder Ergebnis, kann die Environment dafür Belohnungen (englisch *Rewards*) in Form von Punkten vergeben.

Rewards (Belohnungen)

Rewards stellen einen der wichtigsten Bestandteile des Reinforcement Learnings dar. Sie geben dem Agenten eine Rückmeldung darüber, ob die von ihm durchgeführten Aktionen erwünscht sind oder vermieden werden sollten. Dabei kann nicht nur für das Erreichen eines bestimmten Ziels ein Reward ausgegeben werden, sondern auch für Aktionen, die dazu beitragen, das Ziel schneller, besser oder verlässlicher zu erreichen, je nach Konfiguration. Möglich sind auch negative Rewards bzw. Bestrafungen bei kontraproduktivem Verhalten. Die Dimensionierung der Rewards sollte daher individuell angepasst an die Environment erfolgen. Ist die Environment beispielsweise ein von einem Agenten zu erlernendes Videospiel, dann beinhaltet dieses bestenfalls bereits ein Punktesystem, aus dem sich Rewards direkt ableiten lassen. Häufig sind aber zusätzliche Anpassungen notwendig, gerade dann, wenn Rewards zu weit auseinander liegen oder zu selten vergeben werden. Dieser im Englischen als *credit assignment problem* beschriebene Umstand führt

im schlechtesten Fall dazu, dass der Lernalgorithmus keinen Zusammenhang zwischen einer Aktion und einem Reward erkennt. Zur korrekten Verteilung von Rewards gibt es verschiedenste Methoden. Um beispielsweise das Verhalten des Agenten dahingehend zu steuern, unmittelbare oder zeitlich verzögerte Belohnungen zu priorisieren wird häufig ein Diskontierungsfaktor γ (engl. *discount*) eingesetzt. Damit lässt sich die Gewichtung der Rewards in Bezug auf die benötigten Schritte steuern [8].

Agent

Ein Agent, in in Bezug auf KI häufig *KI-Agent*, oder im Englischen *intelligent agent* ist die Komponente des Lernsystems, die für das eigentliche Handeln in einer Environment verantwortlich ist. Der Agent erhält eine Observation, bzw. einen Zustand, und einen Reward von der Environment, wertet diese anhand einer vorgegebenen Policy aus und wählt darüber die nächste auszuführende Aktion. Diese wird dann an der Environment ausgeführt und der Vorgang wiederholt sich bis ein Endzustand erreicht wird. Ein Agent verfolgt dabei stets das Ziel, den kumulierten Reward zu maximieren und lernt dadurch indirekt Strategien, die zu einer optimalen Lösung führen.

Policy (Richtlinie)

Als Policy wird ein Algorithmus bezeichnet, der das Verhalten eines Agenten steuert. Eine primitive Form einer Policy ist beispielsweise durch eine konditionelle Logik („Wenn-Dann-Beziehung“) gegeben. In diesem Fall eines statischen Algorithmus existiert jedoch keine Komponente die ein Lernen ermöglicht. Um auf bekannte Zustände zu reagieren, könnte z. B. eine sogenannte *Brute-Force*--Policy, die sämtliche Zustände einer Environment mit allen verfügbaren Aktionen testet und die Ergebnisse speichert, genutzt werden. Auch in diesem Fall findet kein generelles Lernen, sondern viel eher ein Auswendiglernen statt. Zudem wächst die Größe des Aktionsraums und die Rechenzeit quadratisch mit der Anzahl an Parametern, weswegen sich Policies dieser Art nur für Environments mit wenigen Parametern eignen. Für umfassendere Environments dienen daher fast immer neuronale Netze als Policy [1].

2.5 Stand der Technik

Das Lösen von Yahtzee mittels Computern wurde in der Vergangenheit schon in verschiedenen Arbeiten thematisiert. Der Mathematiker Phil Woodward veröffentlichte 2003 ein Paper mit dem Titel *The Solution*[21] in dem er Strategien für das Lösen von Yahtzee zeigte. Woodward hatte dafür ein Programm geschrieben, dass alle 1.279.054.096.320 Wege, die durch das Spiel führen, berechnet und auswertet. Ein Vorgehen das für damalige Computer eine Rechenzeit mehrerer Tage bedeutete. Nach eigener Aussage würde das Programm einen menschlichen Spieler auf Dauer schlagen können. Trotzdem gibt es durch die Zufallskomponente keine Garantie für einen Sieg. In einer Arbeit von Marcus Larsson und Andreas Sjöberg [11] aus dem Jahr 2012 wird ein Zustandsgraph für die Abbildung möglicher Spielzustände genutzt. Ein von ihnen entwickelter Algorithmus durchläuft dabei rekursiv den Graphen vom Endergebnis zum Start, um den optimalen Weg durch das Spiel zu finden. Mit dieser Strategie konnten sie bei 20.000 Durchläufen eine durchschnittliche Punktzahl von 248,92 erreichen. Eine Punktzahl die relativ nah am Schnitt eines menschlichen Spielers, 250 bis 270 Punkte[3], liegt. Umsetzungen mittels Methoden des Reinforcement Learnings finden sich dagegen erst im letzten Jahrzehnt, nicht zuletzt durch das Aufkommen von weitaus leistungsstärkeren Ansätzen wie dem Deep-Q-Learning [14]. Zwei Arbeiten aus den letzten Jahren mit Themenschwerpunkt Q-Learning sind zum einen die Bachelorarbeit von Klejda Alushi [2] von 2022 und die Masterarbeit von Jan Wolter [20] aus dem Jahr 2025, bei der auch Deep-Q-Learning untersucht wurde.

[3]The Yahtzee Manifesto: `https://www.yahtzeemanifesto.com/yahtzee-odds.php`

3 Anforderungsanalyse

Eine strukturierte Anforderungsanalyse stellt den grundlegenden Baustein für die erfolgreiche Durchführung und die Kontrolle eines Projekts dar. Durch die Ausarbeitung zentraler Anforderungen können Herausforderungen im Vorfeld identifiziert und Fallstricke frühzeitig erkannt werden. Die Interessengruppen, auch *Stakeholder* genannt, sind Ausgangspunkt für die Formulierung der Anforderungen und können damit sowohl direkt als auch indirekt Einfluss auf das Projekt nehmen. Dieser Zusammenhang ist entscheidend, da somit jederzeit überprüft werden kann, ob auf die Bedürfnisse der Stakeholder hingearbeitet wird. Damit bietet die Anforderungsanalyse die Möglichkeit den Erfolg eines Projekts messbar zu machen, da klare Ziele im späteren Verlauf evaluiert werden können.

Im Folgenden wird zunächst auf die Systemumgebung eingegangen, um einen Überblick über die wichtigsten Komponenten und Schnittstellen zu geben, anhand derer sich die Anforderungen dieses Projekts ableiten lassen. Anschließend werden die Stakeholder und ihre Rollen im Projekt erklärt und die Anwendungsfälle beschrieben. Am Ende des Kapitels werden die Anforderungen herausgearbeitet und erläutert.

3.1 Systembeschreibung

Die Systemumgebung wird im Rahmen dieser Arbeit als reines Softwareprojekt umgesetzt. Die Bestandteile des Systems sind auf der einen Seite ein Programm, welches die Logik des Würfelspiels Yahtzee implementiert, auf der anderen Seite ein Algorithmus, der das Lernen des Spiels ermöglicht. Diese Subsysteme sollen jeweils unabhängig voneinander funktionieren, bieten aber Schnittstellen an, über die sie miteinander verbunden werden können. Das übergeordnete System stellt diese Verbindungen her. Die Übersicht in Abb. 3.1 stellt diesen Zusammenhang noch einmal dar.

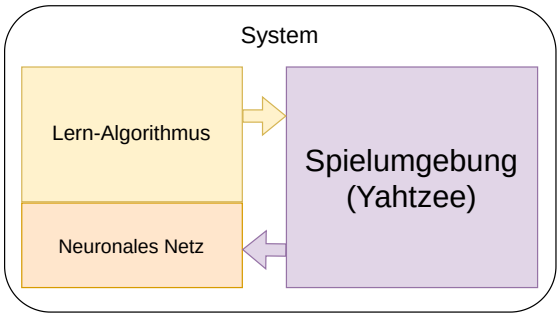


Abbildung 3.1: Die Systemumgebung

3.2 Stakeholder

An dieser Stelle wird die Rolle der Stakeholder erläutert. Sie haben einen maßgeblichen Einfluss auf das Projekt und profitieren auf unterschiedliche Weise von einer erfolgreichen Durchführung.

- **Auftraggeber**: Als Auftraggeber der Arbeit ist Prof. Dr. Hensel daran interessiert, neue Erkenntnisse im Zusammenhang mit Reinforcement Learning zu gewinnen. Diese lassen sich als Basis für zukünftige Abschlussarbeiten nutzen. Auch können im Rahmen dieser Arbeit implementierte Schnittstellen ggfls. für spätere Projekte wiederverwendet werden oder zu einem besseren Verständnis beitragen.
- **Entwickler** Der Entwickler ist der Ersteller dieser Arbeit. Sein Interesse besteht darin, eine funktionsfähige Software nach dem zeitlichen und qualitativen Anspruch einer Bachelorarbeit zu entwickeln. Durch die Einarbeitung in mehrere Themenbereiche wie das maschinelle Lernen und das Entwickeln von Software, damit verbundene Problemlösungen sowie die Interaktion mehrerer vorhandener und eigener Schnittstellen, können die eigenen Fähigkeiten und das eigene Wissen erweitert werden. Zudem verbessert der Entwickler seine Fertigkeiten bei der kontinuierlichen Dokumentation des Projekts sowie im Zeitmanagement und beim Erstellen eines Projektplans.
- **Studierende**: Die Studierenden können Erkenntnisse aus dieser Arbeit nutzen, um darauf aufbauend eigene Projekte im Rahmen ihres Studiums durchzuführen. Auch wird es ihnen ermöglicht, Vergleiche innerhalb von Arbeiten zu verwandten Themenbereichen anzustellen. Eine saubere und strukturierte Dokumentation des

Quellcodes sowie der Herangehensweisen und der Problemlösungen ist für diese Interessensgruppe essenziell.

- **Reinforcement-Learning-Interessenten**: Personen mit einem grundlegenden Interesse an Reinforcement Learning können die Erkenntnisse aus dieser Arbeit nutzen, um eigene Algorithmen zu entwickeln oder zu testen und Vergleiche anzustellen.
- **Spielende**: Yahtzee-Spielende haben zum einen die Möglichkeit, ihre eigenen Fähigkeiten gegen den Computer zu testen, zum anderen lassen sich durch Beobachtung des Algorithmus eventuell neue Strategien für das eigene Spiel entwickeln.

3.3 Virtuelle Umgebung

In diesem Abschnitt werden mit Hilfe eines Anwendungsfalldiagramms die Anwendungsfälle und die daraus abgeleiteten Anforderungen an die Virtuelle Umgebung, folgend als *VU* bezeichnet, erschlossen.

Anwendungsfalldiagramm

Das Anwendungsfalldiagramm (Abb. 3.2) zeigt den Zusammenhang der Akteure und der Anwendungsfälle. Hierbei zeigt der Zusammenhang *«include»* an, dass ein Anwendungsfall obligatorisch einen anderen Anwendungsfall aufruft. Bei einem Aufruf, der nur unter bestimmten Bedingungen auftritt, wird dieser Zusammenhang durch *«extends»* dargestellt.

Anwendungsfälle

Im Folgenden wird detaillierter auf die einzelnen Anwendungsfälle eingegangen

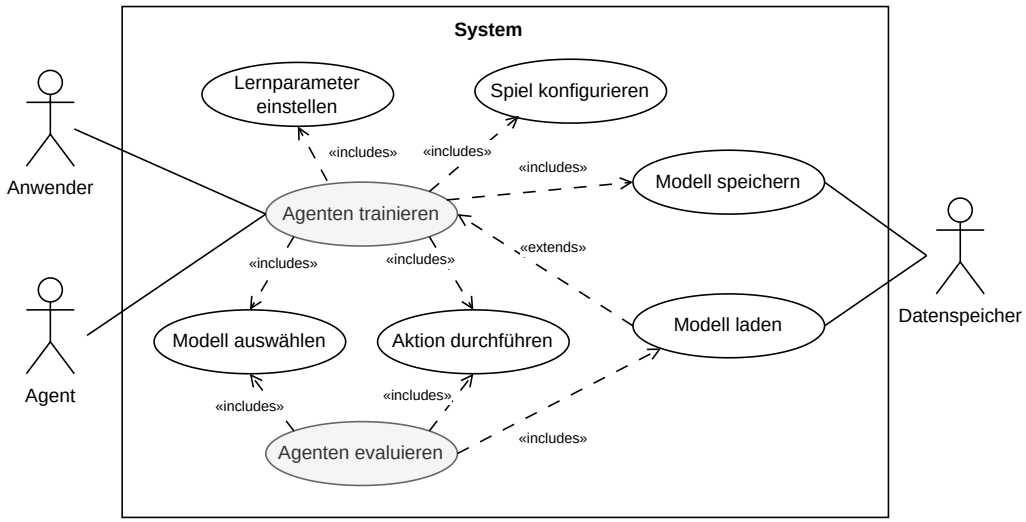


Abbildung 3.2: Anwendungsfalldiagramm

Bezeichner	Spiel konfigurieren
Beschreibung	Der Anwender soll die Möglichkeit haben, das Spiel über die Wahl verschiedener Parameter konfigurieren zu können.
Ergebnis	Umgebung lässt sich anpassen: • Nur oberer Punkteblock (PB) wird bespielt. • Bonuspunkte bei Erreichen von 63 Pkt. im ersten PB. • Oberer und unterer PB werden bespielt. • Spiel ohne Neuwürfeln.
Akteure und Systeme	Anwender

Bezeichner	Agenten trainieren
Beschreibung	Ein Agent soll den Trainingsprozess durchführen, bis ein vorher definierter Endzustand erreicht wurde.
Ergebnis	Der Agent wird durch das Trainieren besser darin, wertvolle Aktionen im Spiel auszuwählen.
Akteure und Systeme	Anwender, Agent

Bezeichner	Modell speichern
Beschreibung	Ein parametrisiertes Modell soll für die spätere Verwendung abrufbar sein.
Ergebnis	Ein Modell wurde in einer Datei gespeichert.
Akteure und Systeme	Datenspeicher

Bezeichner	Modell laden
Beschreibung	Ein parametrisiertes Modell soll aus dem Speicher geladen werden können.
Ergebnis	Ein Modell wurde aus einer Datei geladen.
Akteure und Systeme	Datenspeicher

Bezeichner	Agenten evaluieren
Beschreibung	Ein Agent soll den Ablauf eines Spiels demonstrieren können.
Ergebnis	Ein Agent führt ein Spiel durch und gibt die erreichte Punktzahl aus.
Akteure und Systeme	Anwender, Agent

Bezeichner	Lernparameter einstellen.
Beschreibung	Das Trainingsverhalten soll über Lernparameter eingestellt werden können.
Ergebnis	Lernparameter können vor dem Trainieren eingestellt werden.
Akteure und Systeme	Anwender

Bezeichner	Modell auswählen
Beschreibung	Ein Modell zum Trainieren oder zur Simulation ausgewählt werden.
Ergebnis	Ein Modell wurde ausgewählt.
Akteure und Systeme	Anwender, Agent

Bezeichner	Aktion durchführen
Beschreibung	Ein Agent kann eine Aktion auswählen und durchführen.
Ergebnis	Ein Agent wählt die wertvollste Aktion aus.
Akteure und Systeme	Agent

Anforderungen

Anhand der Anwendungsfälle werden die Anforderungen an die VU formuliert. Hierbei wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden. Funktionale Anforderungen (**F**) bestimmen welche Funktionen das System erfüllen soll. Nichtfunktionale Anforderungen (**NF**) beschreiben darüber hinaus Qualitäts- oder Leistungseigenschaften, d. h. unter welchen Bedingungen eine Aufgabe erfüllt werden soll.

VU-F1: Das Spiel kann im Vorfeld konfiguriert werden.
Es soll möglich sein, das Spiel in seiner Komplexität anzupassen, es können Spiele ohne Bonuspunkte oder ohne Wiederholungswürfe durchgeführt werden. Zudem lässt sich einstellen welche Kategorien für ein Spiel genutzt werden.

VU-F2: Ein Modell kann trainiert werden.
Die Möglichkeit ein Modell zu trainieren ist die Grundlage für seine Funktion.

VU-F3: Modellparameter können in einer Datei gespeichert werden.
Das Speichern der Modellparameter bietet die Möglichkeit, mehrere Modelle zu trainieren und verfügbar zu machen.

VU-F4: Modellparameter können aus einer Datei geladen werden.
Das Laden eines bereits parametrisierten Modells ermöglicht den Vergleich zwischen verschiedenen Modellen, ohne diese mehrmals neu trainieren zu müssen.

VU-F5: Der Trainingsfortschritt kann durch eine geeignete Darstellung evaluiert werden.
Um den Fortschritt des Trainings auswerten zu können, soll seine Leistung in Bezug auf die durchschnittliche Spielpunktzahl visuell dargestellt werden.

VU-F6: Es kann ein Spiel gegen ein Modell durchgeführt werden.
Zu Demonstrationszwecken kann ein Spiel durchgeführt werden, in dem ein trainiertes Modell gegen einen Menschen spielt.

VU-F7: Es gibt eine textbasierte Oberfläche zur Visualisierung der Einstellungen und des Spiels.
Die einfach verständliche Eingabe von Parametern, Auswahl von Modi und Visualisierung eines Spiels soll über ein Kommandozeilenfenster ermöglicht werden.

VU-F8: Es ist möglich, verschiedene Konfigurationen für das Training zu einzustellen.
Das Training soll anhand ausgewählter Lernparameter einstellbar sein.

VU-F9: Es ist möglich, zwischen verschiedenen trainierten Modellen zu wählen.
Zu Simulationszwecken oder zum weiteren Trainieren kann zwischen verschiedenen Modellen ausgewählt werden.

VU-NF1: Die Programmiersprache Python wird für die Implementation genutzt.
Alle Programmteile sollen in Python ausgeführt werden.

VU-NF2: Der Programmcode ist gut strukturiert, dokumentiert und mit sinnvollen Kommentaren versehen.
*Eine gute Dokumentation des Programmcodes dient einer zeiteffizienteren Bearbeitung des Projekts und dem einfacheren Verständnis des/der Leser*innen.*

VU-NF3: Alle Module des Programms sollen über Testfunktionen verfügen.
Es sollen Modultests benutzt werden, um die korrekte Funktionsweise aller Module kontrollieren zu können. Durch automatische Tests lassen sich Fehler zudem früher erkennen.

4 Konzept

Dieses Kapitel behandelt die Auswahl benötigter Softwarekomponenten sowie die Konzeptionierung einer virtuellen Umgebung zur Umsetzung eines Yahtzee-Spiels. Des Weiteren wird die Funktionsweise und das Konzept des RL-Agenten sowie die Wahl des passenden Lernalgorithmus diskutiert. Es wird zunächst die Software thematisiert, mit der die virtuelle Umgebung, die zugrundeliegende Spiellogik und das ML-Modell erstellt werden sollen. Anschließend wird näher darauf eingegangen, wie die einzelnen Komponenten implementiert werden können und wie sie miteinander interagieren, um so eine geeignete Trainingsarchitektur zu bilden.

4.1 Benötigte Software

Im Rahmen dieser Arbeit wird zum Schreiben und Testen des Programmcodes die Entwicklungsumgebung *Visual Studio Code* verwendet, da sich diese problemlos in ihrer Funktionalität erweitern lässt und es bereits Vorerfahrung bei der Benutzung gab.
Weitere benötigte Software wird im Folgenden behandelt.

4.1.1 Python

Der Bereich des maschinellen Lernens hat in den vergangenen Jahren enorm an Momentum dazugewonnen, unter anderem weil freie Software-Bibliotheken und -werkzeuge heutzutage einer breiteren Masse zur Verfügung stehen. Laut eines Blogeintrags [1] von GitHub, dem größten Dienstleister zur Versionsverwaltung von Softwareprojekten, ist (Stand 2024) Python[2] die beliebteste Programmiersprache, die im Bereich künstliche

[1] Octoverse: AI leads Python to top language as the number of global developers surges: `https://github.blog/news-insights/octoverse/octoverse-2024/`, Aufgerufen: 29.04.2025
[2] Offizielle Python Website: `https://www.python.org/`

Intelligenz und Machine Learning zum Einsatz kommt. Neben dem Fokus auf einer einfachen Lesbarkeit und unkomplizierte Syntax wird Python-Code anders als Quellcode kompilierter Sprachen wie C/C++ zur Laufzeit ausgelesen und in ausführbaren Maschinencode umgewandelt. Ein großer Vorteil ist die dadurch gegebene Unabhängigkeit von Betriebssystem und Hardwarearchitektur – das System braucht nur einen Python-Interpreter zur Ausführung. Zusätzlich bleibt der Code eines Python-Skripts für jede Person einseh- und editierbar, die Funktionsweise eines Programms bleibt somit jederzeit nachvollziehbar. Durch mitgelieferte Werkzeuge wie die Paketverwaltung *Pip* lässt sich die Standard-Bibliothek mit rund 300 Modulen[3] zudem beliebig erweitern. Die Paketdatenbank PyPi listet hierzu Stand 2025 über 600.000 Pakete[4] von Drittanbietern. Python ist aus den genannten Gründen die optimale Wahl um ein RL-System im Rahmen dieser Arbeit zu realisieren und zu untersuchen.

4.1.2 OpenAI Gymnasium

Die durch OpenAI entwickelte Open-Source-Bibliothek *Gymnasium* [5] ist eine Schnittstelle die den Trainingsablauf zwischen RL-Algorithmen und Environments vereinheitlicht. Gymnasium bietet ein Framework, in dem Environments mit Aktionsräumen (mögliche Aktionen) und Oberservationsräumen (mögliche beobachtbare Zustände) definiert und anschließend gesteuert werden können. Darüber hinaus ist es auch möglich, vorgefertigte Environments zu importieren, um eigene RL-Modelle an diesen zu trainieren oder zu testen. Über das Gymnasium-Framework lassen sich zudem Schnittstellen zur Visualisierung von Environments einbinden, was bei komplexeren Environments oder bei physikbasierten Simulationen sinnvoll ist. Gymnasium kann somit als Bindeglied zwischen Spiellogik, Reinforcement Learning und Visualisierung gesehen werden. Das Framework kann über Pythons Paketverwaltung installiert werden und lässt sich anschließend im Programmcode verwenden.

4.1.3 TensorFlow & Keras

TensorFlow [6] ist eine der bekanntesten Bibliotheken zur Erstellung und zum Training von Machine-Learning-Modellen und bietet durch eine umfangreiche Dokumentation eine

[3]The Python Standard Library: `https://docs.python.org/3/library/index.html`
[4]Offizielle PyPi Website: `https://pypi.org/`
[5]Gymnasium Documentation: `https://gymnasium.farama.org`
[6]TensorFlow API Documentation: `https://www.tensorflow.org/api_docs/python/tf`

optimale Basis für Softwareprojekte im Bereich des Machine Learnings. Die in Tensorflow integrierte Schnittstelle Keras stellt dazu viele nützliche Funktionen zur Arbeit mit tiefen neuronalen Netzen zur Verfügung. Da neuronale Netze abhängig von ihren Parameteranzahl, Anzahl an verborgenen Schichten und ihrer Komplexität, für ihr Training eine hohe Rechenleistung erfordern, bietet TensorFlow die Möglichkeit, die Berechnungen parallelisiert auf einen Grafikprozessor (GPU) auszulagern und den Trainingsprozess damit stark zu beschleunigen. Bei Grafikprozessoren des Herstellers NVIDIA muss hierfür zusätzlich die Programmierschnittstelle *CUDA*[7] installiert sein. Mit TensorFlow können zudem auch bekannte Datensätze über das Internet importiert und zum Experimentieren genutzt werden. MNIST[8], ein Datensatz bestehend aus 70.000 handgeschriebenen Ziffern, wird beispielsweise in vielen Lehrgängen und in Literatur zum Thema neuronale Netze als Einstieg verwendet [8, 15]. TensorFlow lässt sich mit mehreren Programmiersprachen benutzen, verwendet aber Python als primäre Schnittstelle. Die Installation ist auch hier über Pythons Paketverwaltung möglich.

4.1.4 Software-Übersicht

Um eine volle Kompatibilität zu gewährleisten, wurde darauf geachtet, dass eine bereits ausführlich getestete Kombination der Software-Komponenten eingesetzt wird, die gleichzeitig den Ansprüchen an ein modernes Software-Projekt genügt. Um eine Reproduzierbarkeit zu ermöglichen wurden die genutzten Versionen der wichtigsten Softwarepakete in Tabelle 4.1 dokumentiert.

Software	Versionsnummer
Visual Studio Code	1.102.2
Python	3.12.11
Gymnasium	1.2.0
TensorFlow	2.19.0
Keras	3.10.0
Nvidia CUDA	12.6

Tabelle 4.1: Eingesetzte Software mit Versionsnummern

[7] NVIDIA CUDA Toolkit: `https://developer.nvidia.com/cuda-toolkit`
[8] TensorFlow Dokumentation zum MNIST-Datensatz: `https://www.tensorflow.org/datasets/catalog/mnist`

4.2 Virtuelle Umgebung

Eine virtuelle Umgebung dient dazu, ein reales System nachzubilden und ermöglicht die Erprobung verschiedener Algorithmen in Software. Yahtzee ist dabei sehr gut geeignet für eine Softwareimplementation, denn die Regeln lassen sich unmissverständlich interpretieren und in Programmcode umsetzen. Da davon auszugehen ist, dass die Würfelwahrscheinlichkeiten jedes Würfels identisch sind, ist der pragmatische Ansatz (Pseudo-[9])Zufallszahlen im Bereich $[1, 6]$ für jeden Würfel zu generieren. Neben der zu implementierenden Spiellogik bildet die durch Gymnasium definierte Environment die Schnittstelle der virtuellen Umgebung, über welche Aktionen ausgeführt und Observationen abgerufen werden können. In Abschnitt 4.3 wird dazu zunächst die Umsetzung der Spiellogik diskutiert.

4.3 Konzept der Spiellogik und Environment

Die Spiellogik soll zunächst unabhängig implementiert werden, jedoch so gestaltet sein, dass sich die Funktionalität anschließend in die Gymnasium-Environment einbinden lässt. Zu diesem Zweck lässt sich Paradigma der objektorientierten Programmierung (OOP) nutzen, das zeitgleich eine besser geordnete Projektstruktur ermöglichen kann. Nach der OOP können trennbare Teile einer Logik in einzelne Klassen unterteilt werden.

4.3.1 Spielablauf

Für den Spielablauf wird auf Basis der Yahtzee-Spielregeln ein Flussdiagramm (Abb. 4.1) erstellt, aus dem die wichtigen Teile der Spiellogik abgeleitet werden können. Die Abs. 4.3.2 entworfenen Klassen der Spiellogik müssen in der Lage sein diesen Ablauf abbilden zu können.

[9] Von Computern generierte Zufallszahlen werden auch als Pseudozufallszahlen bezeichnet, da sie i. d. R. deterministisch berechnet werden. Aufgrund der großen Varianz ist diese Differenzierung abseits der Kryptographie aber unerheblich.

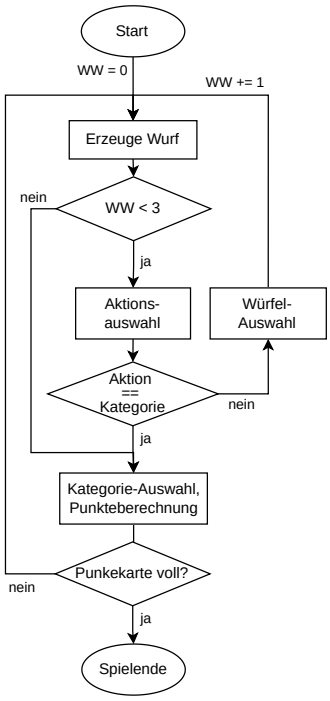


Abbildung 4.1: Flussdiagramm eines Yahtzee-Spiels (WW = Wiederholungswürfe)

4.3.2 Klassen der Spiellogik

Eine Klasse stellt eine Art Bauplan für ein Objekt dar. Objekte können dadurch an anderer Stelle instanziiert und an verschiedene Programmteile weitergereicht werden. Übertragen auf das Spiel Yahtzee lassen sich so etwa die drei trennbare Klassen *Würfel*, *Punktekarte* und *Spieler* festlegen, deren benötigte Funktionalität im Folgenden beschrieben wird.

Würfel-Klasse

Die Würfel-Klasse bildet nicht die Funktionalität eines einzigen Würfels ab, sondern ist vielmehr eine Abstraktion eines gesamten Wurfes, bzw. Wiederholungswurfes. Die Klasse hat primär die Aufgabe Würfe aus fünf Zufallszahlen in den Grenzen $[1, 6]$ zu generieren und soll darüber hinaus auch Wiederholungswürfe korrekt durchführen können, indem nur eine Auswahl aus dem aktuellen Wurf erneut einen Zufallswert zugewiesen bekommt. Der Zustandsraum der bei einer Auswahl aus sechs Würfeln mit $6^5 = 7776$ Zuständen gebildet wird, lässt sich an dieser Stelle reduzieren, indem das Ergebnis eines Wurfes der

Größe nach aufsteigend sortiert wird. Dadurch gibt es $\binom{10}{5} = 252$ mögliche Würfelergebnisse. Gleichzeitig geht keine für das Training benötigte Information verloren, solange der Agent in Zukunft immer bereits sortierte Würfe erhält. Objekte der Würfel-Klasse haben zudem die Aufgabe, den aktuellen Wurf in einer Instanzvariable zu behalten. Abb. 4.2 zeigt den möglichen Aufbau und die Anbindung an andere Klassen.

Punktekarte-Klasse

Die Punktekarte-Klasse stellt die aktuelle Punktekarte dar und verfügt dafür über zwei primäre Instanzvariablen; eine Liste boolescher Werte, die freie oder bereits gewählte Punktekategorien anzeigt sowie eine gleichlange Liste ganzzahliger Werte für die erreichten Punkte je Kategorie. Die Hauptfunktion die von der Punktekarte-Klasse angeboten wird ist die Berechnung der Punkte für eine gewählte Kategorie und den aktuellen Wurf. Des Weiteren bietet sie Hilfsfunktionen zum Ausgeben der Gesamtpunktzahl, zum Anzeigen einer vollen Punktekarte oder um zu Überprüfen ob die Bonuspunkte im oberen Abschnitt der Karte erreicht wurden. Abb. 4.2 zeigt den möglichen Aufbau und die Anbindung an andere Klassen.

Spieler-Klasse

Die Spieler-Klasse stellt das Bindeglied der Würfel- und Punktekarte-Klasse dar und beinhaltet je eine Instanz dieser Klassen. Sie übernimmt zudem übergeordnete Aufgaben, wie das Festhalten der aktuellen Anzahl an Wiederholungswürfen, oder den aktuellen Gesamtpunktestand. Zusätzlich verfügt die Klasse über eine Funktion zum Inkrementieren der Wiederholungswurf-Anzahl sowie eine Funktion zum Beenden eines Zuges. Letztere sollte die Anzahl an Wiederholungswürfen zurücksetzen und den Gesamtpunktestand aktualisieren. Ist mit dem Beenden eines Zuges auch das Spiel zu Ende, wird dies über eine boolesche Variable angezeigt. Abb. 4.2 zeigt den möglichen Aufbau und die Anbindung an andere Klassen.

4.3.3 Gymnasium Environment

Vorgefertigte Environments lassen sich zwar zu einem gewissen Grad anpassen, für das Spiel Yahtzee gibt es jedoch keine geeignete Grundlage, weswegen eine eigene Environment-

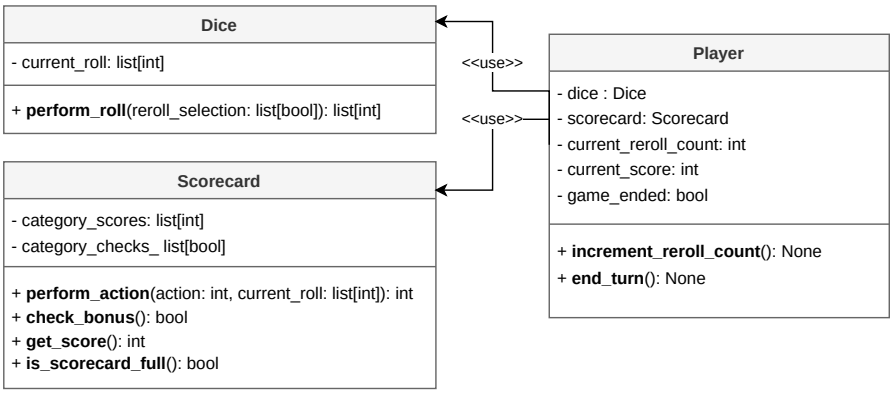


Abbildung 4.2: Prototypen einer Würfel-, Punktekarte- und Spieler-Klasse

Klasse geschrieben werden muss. Gymnasium gibt hierfür eine Reihe an Funktionen vor, die von jeder Environment implementiert werden müssen:

- **reset()** - Setzt die Environment am Anfang eines jeden Spieldurchlaufs auf ihren Ausgangszustand zurück und gibt diesen als Observation zurück.
- **step()** - Empfängt eine Aktionsanweisung, die an der Environment ausgeführt wird und gibt eine neue Observation sowie einen Reward zurück.
- **render()** (optional) - Gibt den aktuellen Zustand an einen optionale Schnittstelle zur Visualisierung weiter.
- **close()** (optional) - Leert alle Variablen der Environment und beendet ggfls. die Visualisierung.

Jede Environment muss zudem auch die folgenden Felder implementieren:

- **action_space** - Der Aktionsraum stellt alle Aktionen dar, mit denen die Environment aktualisiert werden kann.
- **observation_space** - Der Observationsraum bildet die nach außen sichtbaren Zustände der Environment. Er kann aus diskreten oder kontinuierlichen Werten, Binärmustern oder aus mehreren verschiedenförmigen Räumen gebildet werden.
- **spec** (optional) - Enthält die Spezifikation zur Registrierung einer Environment in der Gymnasium-API. Wird in dieser Umsetzung nicht verwednet.
- **metadata** (optional) - Meta-Daten, beispielsweise für die Render-Software.

- **np_random** (optional) - Generator für (Pseudo-)Zufallszahlen. Wird in dieser Umsetzung von der Würfel-Klasse übernommen.

Bei korrekter Implementierung kann die Environment nach Registrierung in der Gymnasium-API im Programmcode anschließend über `env = gymnasium.make("Yahtzee_Env")` initialisiert werden. Alternativ kann die Environment-Klasse aber auch klassisch im Programmcode importiert und anschließend mit `env = Yahtzee_Env()` instanziiert werden.

Definition des Aktionsraums

Der Aktionsraum `action_space` bildet die von einem Agenten auswählbaren Aktionen ab. Er lässt sich anhand verschiedener von Gymnasium vorgegebener Typen[10] erstellen. So können neben Aktionsräumen aus diskreten Aktionen auch vektorbasierte oder eine simultane Auswahl mehrerer Aktionen genutzt werden. Im Fall von Yahtzee sind mögliche Aktionen zum einen die Kategorieauswahl, zum anderen die Würfelauswahl bei Wiederholungswürfen. Die Kategorieauswahl entspricht der Anzahl wählbarer Kategorien $A_K = 13$. Bei den Würfeln ist aufgrund der Mehrfachauswahl eine Darstellung als Binärmuster (z. B. `10100` für die Wahl des ersten und dritten Würfels) möglich. Eine Option ist die Verknüpfung der beiden Aktionsräume als {`Discrete`,`MultiBinary`}; zur Vereinheitlichung soll aber nur ein Typ genutzt werden. Dafür wird die Auswahl der Würfel in eine Reihe diskreter Aktionen überführt. Bei fünf Würfeln sind das $A_W = (2^5 - 1) = 31$ Aktionen, da die Auswahl keines Würfels, also `00000`, nicht möglich ist. Der Aktionsraum für ein Spiel umfasst somit $A_K + A_W = 44$ Aktionen und lässt sich mit Objekten vom Typ `Discrete` aus Gymnasiums Spaces-Klasse initialisieren:

```
action_space = gymnasium.spaces.Discrete(44)
```

Definition des Observationsraums

Der Observationsraum `observation_space` definiert die beobachtbaren Zustände. Beim Yahtzee wird dieser Zustandsraum gebildet aus dem aktuellen Wurf, der Verfügbarkeit der Kategorien und der Anzahl an Wiederholungswürfen. Da die Observationen vor der Eingabe in das DQN aufbereitet werden ist die Wahl eines einheitlichen Typs hier nicht notwendig. Es kann ein Komposit-Raum verschiedener Typen genutzt werden. Folgende Typen bieten sich dafür an

[10] Gymnasium Spaces: `https://gymnasium.farama.org/api/spaces/`

- `MultiDiscrete`: Um einen Wurf fünf diskreter Zahlen in $[1, 6]$ darstellen.
- `MultiBinary`: Um freie (`0`) und belegte (`1`) Kategorien als Bitmuster darzustellen.
- `Discrete`: Um Wiederholungswürfe in $[1, 3]$ als diskrete Zahl darzustellen.

Mit Objekten vom Typ `Dict` aus Gymnasiums Spaces-Klasse, die analog zu Dictionaries in Python aufgebaut sind, lässt sich dieser Komposit-Raum wie folgt initialisieren:

```
observation_space = gymnasium.spaces.Dict({
                    "dice": gym.spaces.MultiDiscrete([6] * 5),
                    "scorecard" : gym.spaces.MultiBinary(13),
                    "rerolls": gym.spaces.Discrete(3)}
```

Die step-Funktion

Mit der `step()`-Funktion wird ein Schritt in der Environment ausgeführt. Dafür erhält die Funktion eine Aktion von einem Agenten und führt diese aus. In Bezug auf das Yahtzee-Spiel sind Aktionen entweder eine Kategorieauswahl oder ein Wiederholungswurf. In `step()` muss daher die Funktionalität vorhanden sein diese Aktionen unterscheiden zu können und entsprechende Schritte auszuführen. Kategorieauswahlen werden somit an der Punktekarte-Instanz aufgerufen (`player.scorecard.perform_action(action)`). Wiederholungswürfe müssen zunächst in eine Würfelauswahl umgewandelt und dann an der Würfel-Instanz aufgerufen werden (`player.dice.perform_roll(reroll_selection)`). Anschließend ist die zweite wichtige Aufgabe der step-Funktion die Vermittlung der erfolgten Zustandsänderung an den Agenten. Gymnasium sieht dafür als Rückgabe der step-Funktion folgende Felder vor:

1. **observation**: Eine Observation der in `observation_space` definierten Form, z. B. {`"dice": [1,2,4,4,6], "scorecard": [0,1,0,0,(...)], "rerolls": 2`}.
2. **reward**: Eine ganzzahlige Belohnung für eine ausgeführte Aktion, bei Yahtzee zunächst die erzielten Punkte für eine gewählte Kategorie.
3. **terminated**: Boolescher Wert zum Anzeigen eines Endzustands, z. B. am Ende eines Spiels mit `terminated = True`.
4. (**truncated**): Boolescher Wert zum Anzeigen einer Unterbrechung, oder eines nicht auflösbaren Zustands. Wird in dieser Umsetzung nicht verwendet.

5. **info**: Optionaler Rückgabewert für Zusatzinformationen, die für das Training sinnvoll sein können.

Bei der konkreten Umsetzung kann es sinnvoll sein, dass manche Auswertungen in separate Hilfsfunktionen ausgelagert werden, z.B. wenn eine Observation aus mehreren Teilen zusammengesetzt ist, oder anderweitig aufbereitet werden muss.

Die reset-Funktion

Die `reset()`-Funktion ist dafür verantwortlich alle Variablen der Environment auf ihren Ausgangswert zurückzusetzen und wird am Anfang eines Spieldurchlaufs aufgerufen. Als Rückgabe sind die Felder `observation` und `info` vorgesehen, analog zu den Rückgabewerten der `step()`-Funktion. In jeder Trainingsschleife ist der damit erzeugte Ausgangszustand der erste Wert, der zur weiteren Auswertung an den Agenten übergeben wird. Die Rückgabewerte `terminated`, `truncated` und `reward` – wie in der step-Funktion definiert – sind in diesem Fall obsolet, da vor dem ersten Schritt kein Endzustand erreicht und kein Reward erzeugt wird.

Anbindung an die Spielklassen

Um die Anbindung der Environment-Klasse an die Würfel-, Punktekarte- und Spieler-Klasse zu verdeutlichen, wird in Abbildung 4.3 das exemplarische Klassendiagramm dieser vier Klassen dargestellt. Durch die Instanziierung der Würfel und Punktekarte in der Spieler-Klasse kann die Environment die Funktionen und Felder dieser Klassen indirekt aufrufen. Dieser getrennte Aufbau sorgt für eine bessere Übersicht bei der Implementierung.

4.4 Konzept des Agenten

Der Agent ist in Bezug auf das Reinforcement Learning, bzw. Machine Learning im Allgemeinen, diejenige Instanz, die für das Treffen von Entscheidungen und für die Auswahl von Aktionen zuständig ist. Er tut dies anhand einer ihm vorgegebenen Policy – einer Richtlinie – die z. B. durch eine explizite Berechnungsvorschrift gegeben sein kann. Bei komplexeren Umgebungen mit sehr umfassenden Zustandsräumen werden jedoch eher Policies in Form von neuronalen Netzen verwendet [1].

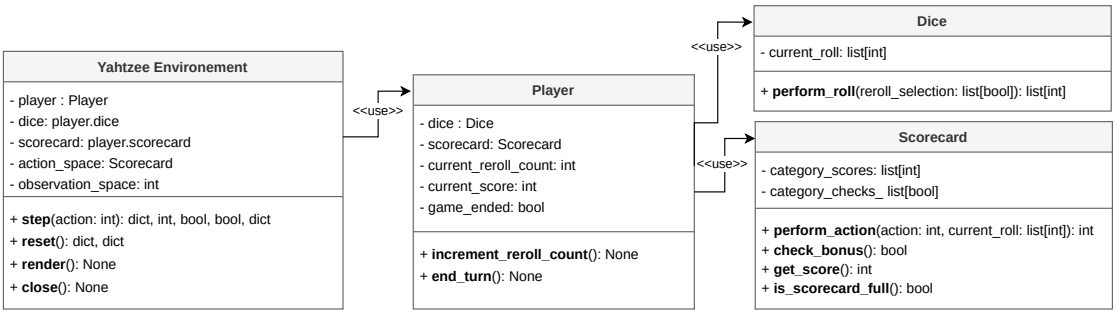


Abbildung 4.3: Prototypischer Aufbau der Environment in Anbindung an die Spielklassen

4.4.1 Wahl des Vorgehens

Die Wahl eines geeigneten Vorgehens zur Entwicklung des Agenten ist stark vom zu lösenden Problem abhängig. Je nach Komplexität der Environment sind verschiedene Ansätze möglich. Dies soll im Folgenden diskutiert werden.

Q-Learning

Bei der Suche nach geeigneten Verfahren um Reinforcement Learning am Beispiel eines Yahtzee-Spiels umzusetzen, wurde zunächst das in Abschnitt 2.2.2 beschriebene Q-Learning ins Auge gefasst. Bei einer Umsetzung mittels Q-Learning würde der Agent nach einem Monte-Carlo Algorithmus (Abs. 2.2.2) spielen, daher so lange zufällige oder teils zufällige Aktionen ausführen bis alle möglichen Q-Werte optimal angenähert wurden. Für die Berechnung eine Q-Werts, der die erwartbare Summe der diskontierten Rewards eines Zustand-Aktions-Paars (s, a) darstellt, ist es notwendig den Reward aller Kombinationen (s, a, s') zu notieren. Um die Größe der notwendigen Q-Tabelle zu bestimmen, soll an dieser Stelle kurz dargestellt werden, wie der Zustandsraum dimensioniert ist. Wie zuvor festgestellt wurde bilden die Kombinationen der Würfel, die freien und belegten Kategorien der Punktekarte sowie die Anzahl der Wiederholungswürfe den Zustandsraum bzw. der Observationsraum (Abs. 4.3.3). Die dabei möglichen Zustände werden noch einmal aufgeschlüsselt:

- **Würfel** (W): 252 mögliche Kombinationen, bei einer sortierten Auswahl.
  $\rightarrow \binom{6+5-1}{5} = \binom{10}{5} = 252$

- **Punktekarte** (P): 8192 mögliche Kombinationen, bei dreizehn Kategorien.
  $\rightarrow 2^{13} = 8192$
- **Wiederholungswürfe** (W_W): Drei mögliche Zustände.

Die Größe des Zustandsraums wird folglich gebildet durch $W \cdot P \cdot W_W = 252 \cdot 8192 \cdot 3 = 6.193.152$ Zustände. Für die Bildung der Q-Werte kommt noch die Menge der Aktionen als zusätzlicher Faktor hinzu. Wie in Abs. 4.3.3 ermittelt gibt es 44 diskrete Aktionen. Die Menge der Q-Werte würde somit $Q_N = 6.193.152 \cdot 44 = 272.498.688$ Werte umfassen. Zur Erfassung der Rewards kämen hier noch die Folgezustände hinzu. Es ist davon auszugehen, dass die Menge der Daten zu sehr hohen Rechenzeiten führt. Q-Learning im klassischen Sinne ist daher nicht zielführend für die Anwendung an einem Yahtzee-Spiel.

Deep-Q-Learning

Die Anwendung des in Abschnitt 2.3.4 beschriebenen Deep-Q-Learnings ist besser für die Zustandsraumgröße eines Yahtzee-Spiels geeignet, da nicht für jeden möglichen Zustand ein durchschnittlicher Q-Wert ermittelt werden muss. Ziel ist es stattdessen eine Funktion $Q_\theta(s, a)$ anzunähern, die für Zustands-Aktions-Paare (s, a) einen Q-Wert approximiert. Die Funktion stellt diesem Fall das Deep-Q-Netzwerk dar. Nach dem Optimalitätsprinzip von Bellman (Abs. 2.2.2) kann ein optimaler Q-Wert (Q_{target}) berechnet werden, indem ein beobachteter Reward r nach Ausführen von Aktion a in Zustand s mit dem diskontierten maximalen Reward aller im Folgezustand s' möglichen Aktionen a' addiert wird. Die folgende Gleichung verdeutlicht wie Q_{target} gebildet wird:

$$Q_{target}(s, a) = r + \gamma \cdot max_{a'}\ Q_\theta(s', a') \tag{4.1}$$

Die Berechnung von Q_{target} findet in jedem Trainingsschritt des Netzes statt. Mit dem Fehler zwischen Q_{target} und dem vorhergesagten Q-Wert wird der Gradient berechnet und die Gewichte des Netzes aktualisiert [8]. Da die Voraussetzungen zur Anwendung dieses Lernverfahrens gegeben sind, wird der Fokus zur Umsetzung auf das Deep-Q-Learning gerichtet.

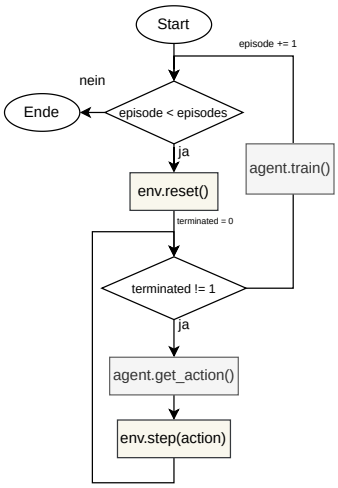


Abbildung 4.4: Flussdiagramm der Trainingsschleife

4.4.2 Entwurf der Trainingsschleife

Das Training von RL-Algorithmen erfordert eine Trainingsschleife, die entweder eine feste Anzahl an Durchgängen (Episoden) durchlaufen wird, oder solange bis ein gewünschter Trainingsfortschritt erreicht wurde. Letzteres ist insbesondere dann wichtig, wenn das Modell ab einem Zeitpunkt zu Overfitting neigt. In diesem Fall können Regularisierungstechniken wie das frühzeitige Stoppen (engl. *Early Stopping)* genutzt werden, um das Modell in festen Intervallen zu evaluieren und die Trainingsschleife bei erkennbarer Verschlechterung der Performance zu unterbrechen. Evaluation bedeutet in diesem Fall, dass die Trainingsschleife kurz pausiert und das aktuelle Modell für mehrere Episoden an der Environment getestet wird, ohne dabei Trainingsschritte auszuführen – die Gewichte des Netzes werden also nicht angepasst. Innerhalb der Trainingsschleife, i. d. R. eine `for`-Schleife mit fester Episodenanzahl, wird eine innere `while`-Schleife genutzt , die durch den Wert `terminated` der Environment begrenzt ist. Zur besseren Unterscheidung wird die innere Schleife im Folgenden als Spielschleife bezeichnet. Es besteht zwar die Möglichkeit, mit jedem Durchlauf der Spielschleife auch einen Trainingsschritt auszuführen, allerdings kann die Anpassung des Netzes inmitten eines Spiels zu einem instabilen Lernen führen. Daher wird erst beim Verlassen der Spielschleife ein Trainingsschritt ausgeführt dafür eine Auswahl an Erfahrungen aus dem Replay-Buffer (Abs. 4.4.4) genutzt [8]. Das Flussdiagramm in Abbildung 4.4 zeigt den exemplarischen Ablauf der Trainingsschleife.

4.4.3 Agent- und DQN-Klasse

In diesem Abschnitt werden die Funktionen beschrieben, die von der Agent-Klasse und der DQN-Klasse angeboten werden.

Funktionen der Agent-Klasse

Die Agent-Klasse hat die Aufgabe Trainingsschritte und Aktionsschritte auszuführen und übernimmt darüber hinaus noch die Vor- und Nachverarbeitung von Observationen und Vorhersagen. Die folgenden Funktionen sollen von der Agent-Klasse implementiert werden:

- **Trainingsfunktion**: In dieser Funktion werden die Trainingsschritte am Deep-Q-Netzwerk ausgeführt. Sie erhält den bisherigen und folgenden Zustand, den Reward, die ausgeführte Aktion und den `terminated`-Wert von der Environment. Mit den eingegebenen Werten können die Q-Werte und Target-Q-Werte (s. Gleichung 4.1) berechnet, der Fehler ermittelt und schließlich ein Anpassungsschritt mittels Gradientenabstieg durchgeführt werden.
- **Epsilon-Greedy-Schrittfunktion**: Diese Funktion erhält die aktuelle Observation und einen Epsilon-Wert (s. Abs. 2.2.2) und gibt je nach Ergebnis entweder eine zufällige Aktion aus, oder ruft mit der Observation eine Vorhersage aus dem DQN ab.
- **Vorverarbeitungfunktion**: Diese Funktion erhält die aktuelle Observation und normalisiert alle Werte auf einen Bereich $[0, 1]$. Dieser Schritt wird durchgeführt um ein stabileres Lernen zu ermöglichen, andernfalls könnten die Aktivierungsfunktionen bei größeren Werten schnell übersättigt werden und das Training behindern.
- **Nachverarbeitungsfunktion**: Diese Funktion führt eine Softmax-Auswahl (s. Abs. 2.2.2) auf den Ausgangswerten (Logits) des DQN aus.

Die Abbildung 4.5 zeigt das exemplarische Klassendiagramm von Agent- und DQN-Klasse.

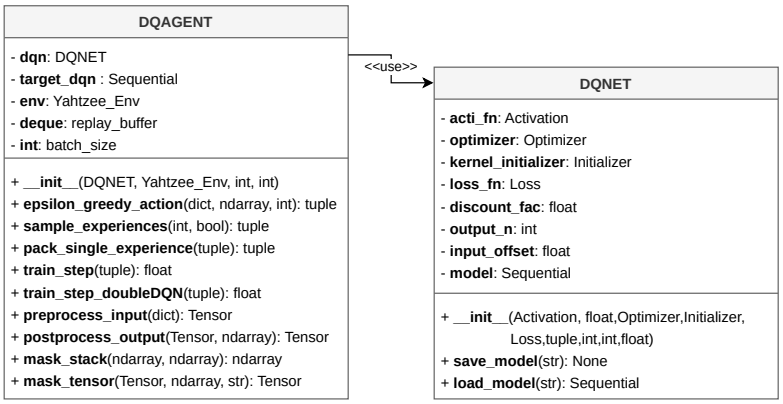


Abbildung 4.5: Klassendiagramm der Klassen DQAGENT und DQNET

Funktionen der DQN-Klasse

Die DQN-Klasse beinhaltet das Deep-Q-Netzwerk. Sie wird mit den Hyperparametern aus Abs. 4.4.4 aufgerufen und initialisiert das Netz, bzw. das Modell. Darüber hinaus bietet die DQN-Klasse noch Funktionen zum Speichern und Laden des Modells. Das Klassendiagramm in Abb. 4.5 zeigt exemplarisch die Klassen.

4.4.4 Aufbau des Trainings

Um schneller ermitteln zu können, welche Trainings- und Hyperparametereinstellungen ein effizienteres Lernen ermöglichen, ist es sinnvoll zunächst mit einem reduzierten Zustandsraum zu trainieren und die Komplexität schrittweise zu erhöhen. Um einen kleineren Zustandsraum beim Yahtzee zu erwirken kann beispielsweise der Aktionsbereich für Wiederholungswürfe und der untere Punkteblock ignoriert werden. Nach diesem Schema wurden insgesamt vier Konfigurationen, die sich zu einer Unterteilung eignen, identifiziert:

- Nur oberer Punkteblock aktiv, ohne Bonuspunkte und ohne Wiederholungswürfe.
- Nur oberer Punkteblock aktiv, ohne Bonuspunkte, aber mit Wiederholungswürfen.
- Nur oberer Punkteblock aktiv, mit Bonuspunkten und mit Wiederholungswürfen.
- Oberer & unterer Punkteblock aktiv, mit Bonuspunkten und mit Wiederholungswürfen.

Dieser Aufbau hin zu einem vollständigen Spiel ermöglicht darüber hinaus auch, dass ein Vergleich zwischen den Konfigurationen angestellt werden kann. Durch die damit einhergehende Änderung des Verhaltens der Environment muss zusätzlich gewährleistet werden, dass alle Klassen mit den unterschiedlichen Konfigurationen umgehen können.

Replay-Buffer

Um eine größere Flexibilität zu ermöglichen, wird bei DQNs häufig ein Replay-Buffer als Zwischenspeicher für die in der Spielschleife gemachten Erfahrungen eingesetzt. Eine „Erfahrung“ bezeichnet in diesem Fall ein Datenobjekt, das aus den für einen Trainingsschritt notwendigen Variablen (`observation`, `action`, `reward`, `next_observation`, `terminated`) zusammengesetzt wird. Der Replay-Buffer ist eine Liste nach dem FIFO-Prinzip (*First in – First out*), die eine begrenzte Anzahl an Erfahrungen speichern kann. Bei einem gefüllten Buffer wird der älteste Eintrag entfernt sobald eine neue Erfahrung abgelegt wird. Dieser Vorgehen eignet sich dafür, das Problem des „katastrophalen Vergessens“ eines Netzes zu reduzieren. Katastrophales Vergessen tritt dann auf, wenn das Erkunden eines Teils der Umgebung dazu führt, dass bereits Gelerntes überschrieben wird [8] Durch den Replay-Buffer – insbesondere durch die zufällige Wahl von Erfahrungen aus dem Replay-Buffer – lässt sich ein Problem besser generalisieren.

Belohnungssystem

Das Belohnungssystem stellt einen wichtigen Faktor für den Lernerfolg von RL-Algorithmen dar. Beim Yahtzee ist ein Punktesystem schon vorhanden, die vergebenen Punkte lassen sich unmittelbar als Rewards für den Agenten verwenden. Ein mögliches Problem dabei ist, dass Wiederholungswürfe bei diesem Vorgehen nicht direkt Punkte erzeugen, sondern erst durch die schlussendlich gewählte Kategorie. Es wird zunächst eine Umsetzung nach dem vorhandenen Belohnungssystem vorgenommen, aufgrund der genannten Problematik sind jedoch ggfls. Änderungen bei der Verteilung der Rewards notwendig.

Initiale Dimensionierung der Trainings- und Hyperparameter

Es soll zunächst ein initialer Satz an Parametern gewählt werden, der sich für das Training in der Environment eignet. Dabei werden teils Erfahrungswerte aus der Literatur verwendet, oder Parameter gewählt, die für die Environment sinnvoll erscheinen:

- **Episodenanzahl**: Die Anzahl der Trainingsdurchläufe, hier stellen 1.000 Episoden für die meisten Umgebungen einen guten Kompromiss aus Rechenzeit und Lernerfolg dar [8, 7, 1]. Innerhalb dieses Rahmens sollte eine Tendenz im Lernerfolg erkennbar sein. Bei durchgehend steigender Tendenz kann anschließend das Training mit einer größeren Episodenanzahl durchgeführt werden.
- **Epsilon**: Für Epsilon wird zu Beginn der Trainingsschleife ein Wert nahe $\epsilon = 1$ gewählt damit die Umgebung durch zufällige Aktionen erkundet wird. Dieser Wert kann dann über den Verlauf der Trainings kontinuierlich zu $\epsilon \to 0$ dekrementiert werden [8].
- **Schichten des DQN**: Anfänglich kann eine Schicht gewählt werden, für die Anzahl an Neuronen werden üblicherweise Zweierpotenzen benutzt. Bei geringem Lernerfolg kann eine größere Anzahl Neuronen und/oder Schichten getestet werden. Es wird zunächst mit einer Schicht mit 64 Neuronen begonnen und die Anzahl dann testweise auf bis zu drei Schichten mit je bis zu 256 Neuronen erhöht [15].
- **Lernrate**: Die Lernrate wird i. d. R. sehr klein gewählt, um eine Überanpassung beim Gradientenabstieg zu vermeiden. Ein guter Startwert ist eine Lernrate von 0,001 [1].
- **Optimierer**: Als Optimierer wird Adam (*Adaptive Moment Estimation*) benutzt. Dieser Optimierer hat sich vielfach bewährt um das Steckenbleiben an lokalen Minima beim Gradientenabstieg zu verhindern [15].
- **Aktivierungsfunktion**: Als Aktivierungsfunktion wird Leaky-ReLu verwendet um von Vornherein das Entstehen toter Neuronen zu verhindern. Es kann darüber hinaus getestet werden, ob die Aktivierungsfunktion ELU zu einem besseren Lernverhalten führt.
- **Gewichtsinitialisierung**: Für die Gewichte des DQN wird die He-Initialisierung genutzt, da diese dafür ausgelegt ist, ein stabiles Training bei der Verwendung von (Leaky-)ReLu zu fördern.
- **Verlustfunktion**: Für die Verlustfunktion wird in den meisten Anwendungsfällen die mittlere Quadratische Abweichung (MSE) gewählt [8]. Diese Einstellung wird beibehalten, sofern keine Probleme auftreten.

- **Diskontierungsfaktor**: In der Regel wird ein Diskontierungsfaktor $\gamma > 0.9$ gewählt um verzögerte Rewards zu bevorzugen [8, 1]. Da in der ersten Konfiguration (Abs. 4.4.4) nur die Punktekategorien zur Auswahl stehen und unmittelbare Rewards damit bevorzugt werden sollen, wird zunächst ein kleiner Diskontierungsfaktor $\gamma = 0.05$ gewählt. Bei den weiteren Konfigurationen soll überprüft werden, ob ein höherer Diskontierungsfaktor das Lernverhalten verbessert.
- **Replay-Buffer-Größe**: Für den Replay Buffer wird sich an der Implementierung nach Géron [8] orientiert. Dieser nutzt eine Replay-Buffer-Größe von 2.000.
- **Batch-Größe**: Als Batch-Größe wird ein Wert von 32 gewählt, dieser soll im Verlauf des Trainings nach Möglichkeit vergrößert werden.

4.4.5 Zusätzliche Überlegungen

Ein Aspekt, der für das Training des neuronalen Netzes im Rahmen dieser Arbeit nicht behandelt wird, ist der Einfluss des gegnerischen Punktestands auf das Spielverhalten. Demnach wäre es denkbar, dass eine geringe Punktedifferenz zum Gegner gemäß eines Alles-oder-nichts-Prinzips in einem riskanteren Spielverhalten resultiert. Um dieses Verhalten zu implementieren, müssten die Punktestände anderer Spieler in den Observationsraum integriert und das Belohnungssystem soweit angepasst werden, das eine geringe Punktedifferenz gegen Ende eines Spiels zu größeren Belohnungen führt.

5 Entwicklung

5.1 Entwicklung der virtuellen Umgebung

Im Folgenden wird die Entwicklung der Spiellogik und Environment, sowie eventuelle Schwierigkeiten bei der Implementierung beschrieben.

5.1.1 Entwicklung der Spiellogik

Die Spiellogik wurde wie in Abs. 4.3 implementiert. Die ursprüngliche Spieler-Klasse wurde dabei zum besseren Verständnis als Spiel-Klasse `Yahtzee` angelegt. Durch die Ergänzung verschiedener Spielkonfigurationen war es zudem notwendig eine Variable `game_mode` einzuführen, die einen Wert in den Grenzen $[0, 3]$ annimmt. Der Spielmodus `0` gibt dabei ein Spiel nur mit oberem Punkteblock und ohne Wiederholungswurf oder Bonus an (Vgl. Abs. 4.4.4)). Der Spielmodus wird durch die Environment-Instanz `Yahtzee_Env` an die Spiel-Instanz `Yahtzee` weitergereicht, die bei der Instanziierung die passenden Parameter benutzt, um eine Instanz der Punktekarte-Klasse `Scorecard` zu erzeugen. So erzeugt eine `Scorecard`-Instanz z. B. nur die durch den Spielmodus vorgegebene Anzahl an Kategorien. Für Instanzen der Würfel-Klasse `Dice` ergibt sich keine Änderung zwischen den Spielmodi. Die `Scorecard`-Klasse wurde um einige Hilfsfunktionen wie `calc_field_score (dice_roll, action)` erweitert, mit der die Berechnung der Kategorie-Punkte aus der Funktion `perform_action(dice_roll, action)` ausgelagert wird. Diese feinere Unterteilung in spezialisierte Funktionen hat sich bei der Entwicklung des Programmcodes bewährt, da die korrekte Funktionsweise somit schneller überprüft werden kann.
Das Klassendiagramm in Abb. 5.3 zeigt die implementierten Klassen.

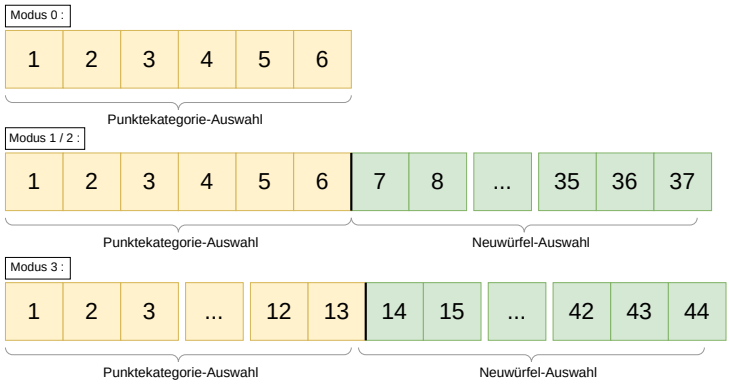


Abbildung 5.1: Der vereinte Aktionsraum der vier Spielmodi (v.o)

Schwierigkeiten

Ein Problem, das früh auffiel, war die einseitig beschränkte Interaktion der Spiel-Klasse mit den untergeordneten Klassen, bedingt durch das geplante Schema. So war es anfangs nicht möglich, dass Funktionen der Würfel- oder Punktekarte-Klasse auf die Spiel-Klasse zugreifen um so z. B. eine Runde zu beenden sobald eine Kategorie ausgewählt wurde. Dies wurde dadurch gelöst, dass die untergeordneten Klassen bei ihrer Instanziierung je eine Referenz auf die Spiel-Klasse erhalten und diese damit direkt verwenden können.

5.1.2 Gymnasium Environment

Die geplanten Funktionen der Environment wurden in der Klasse `Yahtzee_Env` umgesetzt. Um auch hier mehrere Spielmodi zu ermöglichen wird bei der Initialisierung der Modus abgefragt und ein Aktionsraum mit den korrekten Dimensionen angelegt:

```
if game_mode == 0:      action_space = gym.spaces.Discrete(6)
if game_mode in [1,2]:  action_space = gym.spaces.Discrete(37)
if game_mode == 3:      action_space = gym.spaces.Discrete(44)
```

Abbildung 5.1 veranschaulicht den Aufbau des Aktionsraums noch einmal. Der Observationsraum wird ebenso in Abhängigkeit des Spielmodus erstellt. Der generelle Aufbau ist in Abb. 5.2 dargestellt. Des Weiteren wurde die step-Funktion um die unterschiedliche Verarbeitung von Kategorie- und Wiederholungswurf-Aktion ergänzt. Letztere leitet die Aktionsauswahl an die Hilfsfunktion `make_reroll_selection(action)` weiter, wo die diskrete Aktion in eine Würfelauswahl umgewandelt wird. Abbildung 5.3 stellt dazu nochmal ein Klassendiagramm der Environment und Spiel-Klassen dar.

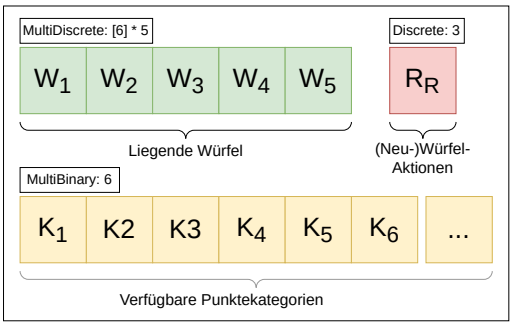


Abbildung 5.2: Der Observationsraum bestehend aus mehreren Sub-Räumen

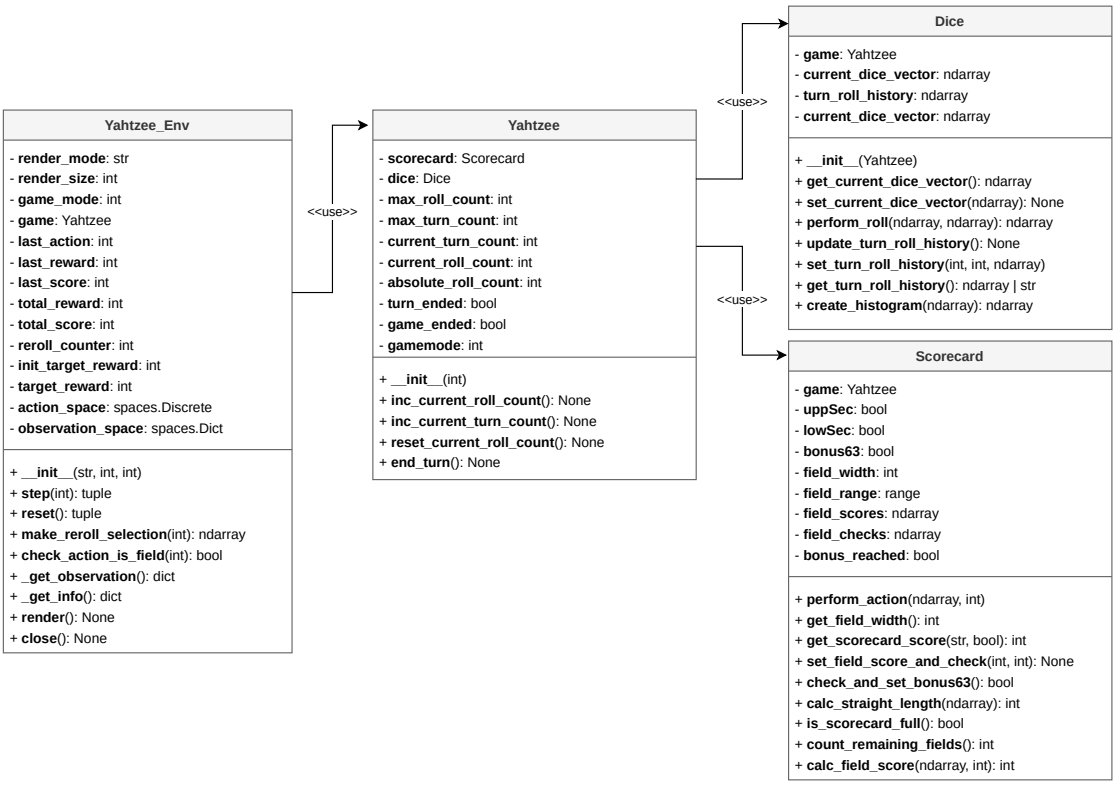


Abbildung 5.3: Klassendiagramm der implementieren Klassen

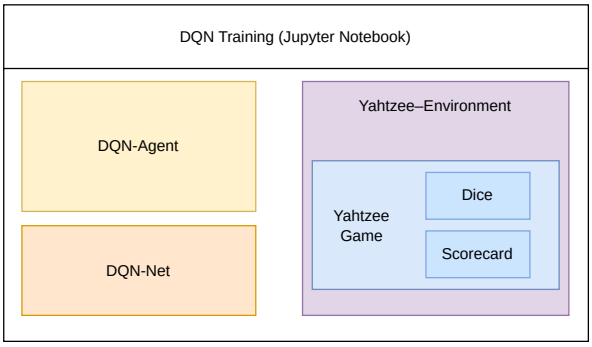


Abbildung 5.4: Übersicht über die Module des Programms

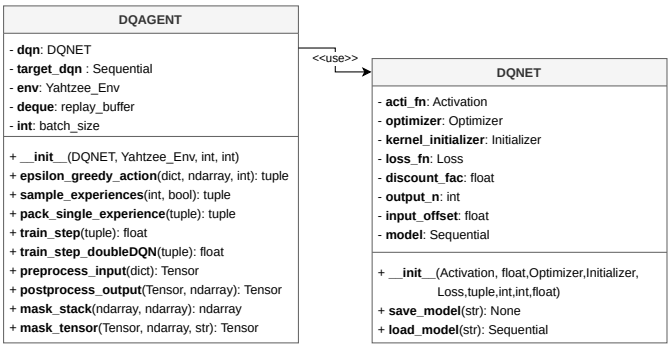


Abbildung 5.5: Klassendiagramm der Agent- und DQN-Klasse

5.2 Entwicklung des Agenten

Die Agent-klasse wurde nach dem in Abs. 4.4.3 beschriebenen Aufbau implementiert. Bei der Trainingsfunktion diente die Implementation nach Géron [8] als Referenz. Die übergeordnete Trainingsschleife wurde in einem Jupyter-Notebook[1] geschrieben, damit die im Training generierten Plots, also der Verlauf der durchschnittlichen Rewards und der Verluste, direkt angezeigt werden können. Abbildung 5.4 zeigt dabei schematisch die finale Projektstruktur. Die Anbindung der Agent- und DQN-Klasse wird im Klassendiagramm in Abb. 5.5 dargestellt.

[1] Project Jupyter `https://jupyter.org/`

5.3 Trainingsdurchführung

Die in Abschnitt 4.4.4 bestimmten Parameter bilden den initialen Satz der Trainings- und Hyperparameter, mit denen erste Trainingsversuche unternommen wurden. Dabei wurden Rewards und Verluste geplottet. Für die Rewards wurde ein gleitender Mittelwert mit einer Fenstergröße von 100 Datenpunkten gebildet, der ebenfalls in allen folgenden Plots dargestellt wird. Die Tabelle 5.2 listet die initialen Parameter noch einmal auf.

5.3.1 Vergleichswerte

Um eine generelle Vergleichbarkeit zu ermöglichen, wurde im Vorfeld getestet, wie viele Punkte ein Agent durchschnittlich erreicht, wenn eine naive Policy, die explizit die gewinnbringendste Kategorie auswählt, genutzt wird. Diese Policy erreichte in Spielmodus 0 durchschnittlich 32,6 Punkte, bei einem vollständigen Spiel (Spielmodus 3) waren es im Schnitt 112,8 Punkte. Diese Werte (Tab. 5.1) sollen im Weiteren als Richtwert dienen. Erreicht der DQN-Agent diese Punktzahlen, wäre das ein Indiz, dass das Netz den Zusammenhang aus Würfeln und Kategorien abstrahieren konnte.

Modus	Durchschnitts-Reward
Spielmodus 0 bis 2	32,6
Spielmodus 3	112,8

Tabelle 5.1: Durchschnittlich erreichte Punkte eines naiven Agenten

5.3.2 Spielmodus 0

Mit den initialen Parametern wurde ein Trainingsdurchlauf für Spielmodus 0 – nur oberer Punkteblock, ohne Bonus oder Wiederholungswurf – durchgeführt. In diesem Spielmodus ist die maximal erreichbare Punktzahl $\sum_{n=1}^{6} 5n = 105$. Wie in Abb. 5.6 zu sehen, lieferte dieser Satz an Parametern keine guten Ergebnisse. Die grundsätzliche Funktion konnte aber durch eine leichte Abnahme in der Verlustkurve bestätigt werden. Dennoch zeigten die durchschnittlichen Rewards kaum eine Änderung im Lernprozess, im Schnitt wurden unter 20 Punkte erreicht.

Parameter	Wert
Trainingsepisoden	1000
Buffer-Episoden	100
Replay-Buffer-Größe	2000
Batch-Größe	32
Verdeckte Schichten: [Neuronen]	1:[64]
Lernrate	0,001
Optimierer	Adam
Aktivierungsfunktion	Leaky-ReLu
Gewichtsinitialisierung	He
Verlustfunktion	Mittlere quadratische Abweichung (MSE)
Diskontierungsfaktor	0,05

Tabelle 5.2: Initialer Satz an Trainings- und Hyperparametern

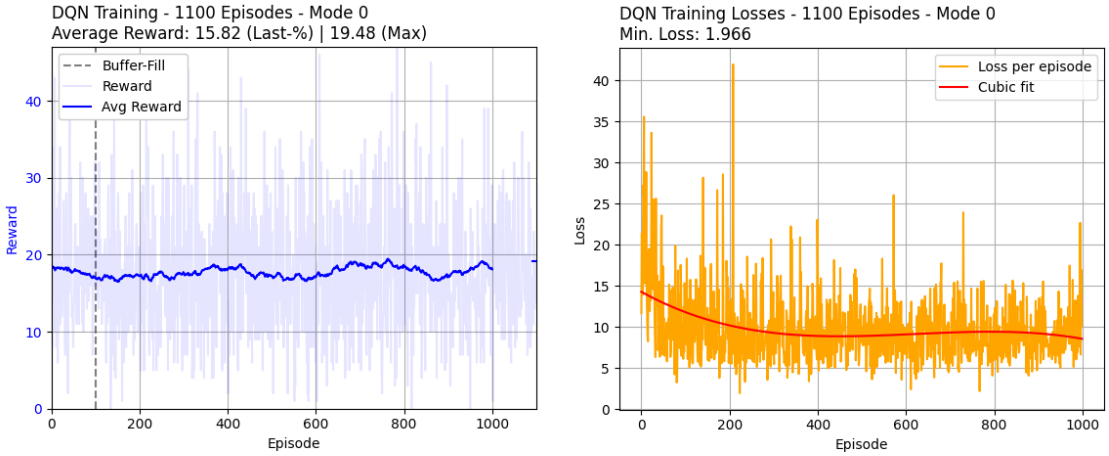


Abbildung 5.6: Trainingsdurchlauf mit initialen Parametern – (Modus 0)

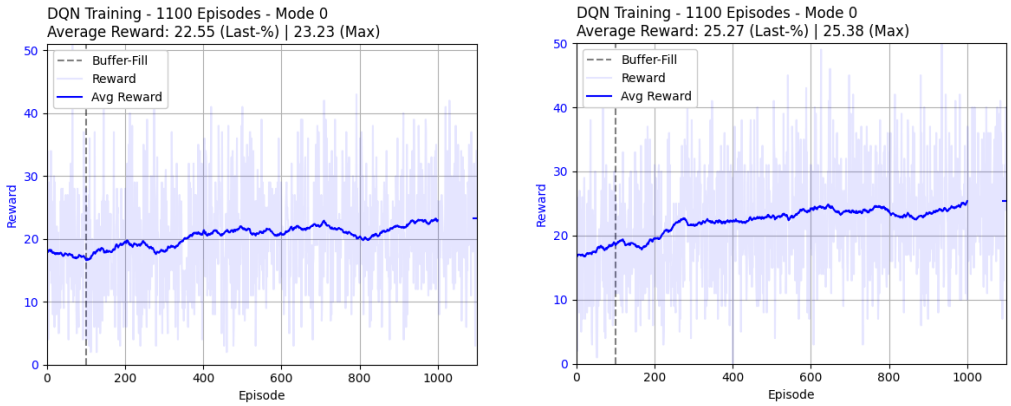


Abbildung 5.7: Angepasste Lernraten lr $= 0,01$ (l.) und lr $= 0,1$ (r.) – (Modus 0)

Spielmodus 0 – Anpassung der Lernrate

Die Suche nach optimalen Hyperparametern kann sich als komplex gestalten, je nach Größe des Netzes und dem Verhalten der Environment. In der Regel wird bei Anpassungen primär versucht ein stabiles Lernverhalten zu erzielen und bei ausreichender Stabilität die Leistung optimiert [15].

Da das vorangegangene Training keinen signifikanten Lernerfolg gezeigt hat, wurde zunächst die Lernrate um eine Zehnerpotenz auf $lr = 0,01$ erhöht. Dadurch zeigte sich eine leicht ansteigende Lernkurve. Im nächsten Schritt wurde die Lernrate noch einmal auf $lr = 0.1$ erhöht (s. Abb. 5.7), womit ein durchschnittliche Reward von 24 Punkten erreicht werden konnte. Höhere Werte $lr > 0,1$ führten nicht zum weiteren Anstieg.

Spielmodus 0 – Anpassung der Batch-Größe

Die Anpassung der Lernrate führte zu einem steigenden Lernverhalten, erreichte in 1.000 Episoden aber kein erkennbares Plateau. Eine Möglichkeit besteht darin, die Anzahl der Episoden zu erhöhen, wodurch aber die Rechenzeit proportional verlängert wird. Effektiver ist die Anpassung der Batch-Größe, da hierdurch für mehrere Erfahrungen gleichzeitig nur ein Gradient berechnet werden muss. Für größere Batches fällt zwar auch mehr Rechenzeit pro Batch an, allerdings sehr viel weniger als für Einzelschritte gebraucht wird. Die Batch-Größe ist grundsätzlich nur durch den verfügbaren Grafik- oder Hauptspeicher begrenzt, größere Batches können aber auch zu einer schlechteren Generalisierung führen [10]. Die Batch-Größe wurde im Folgenden auf 128 Batches erhöht. Nach einer Steigerung

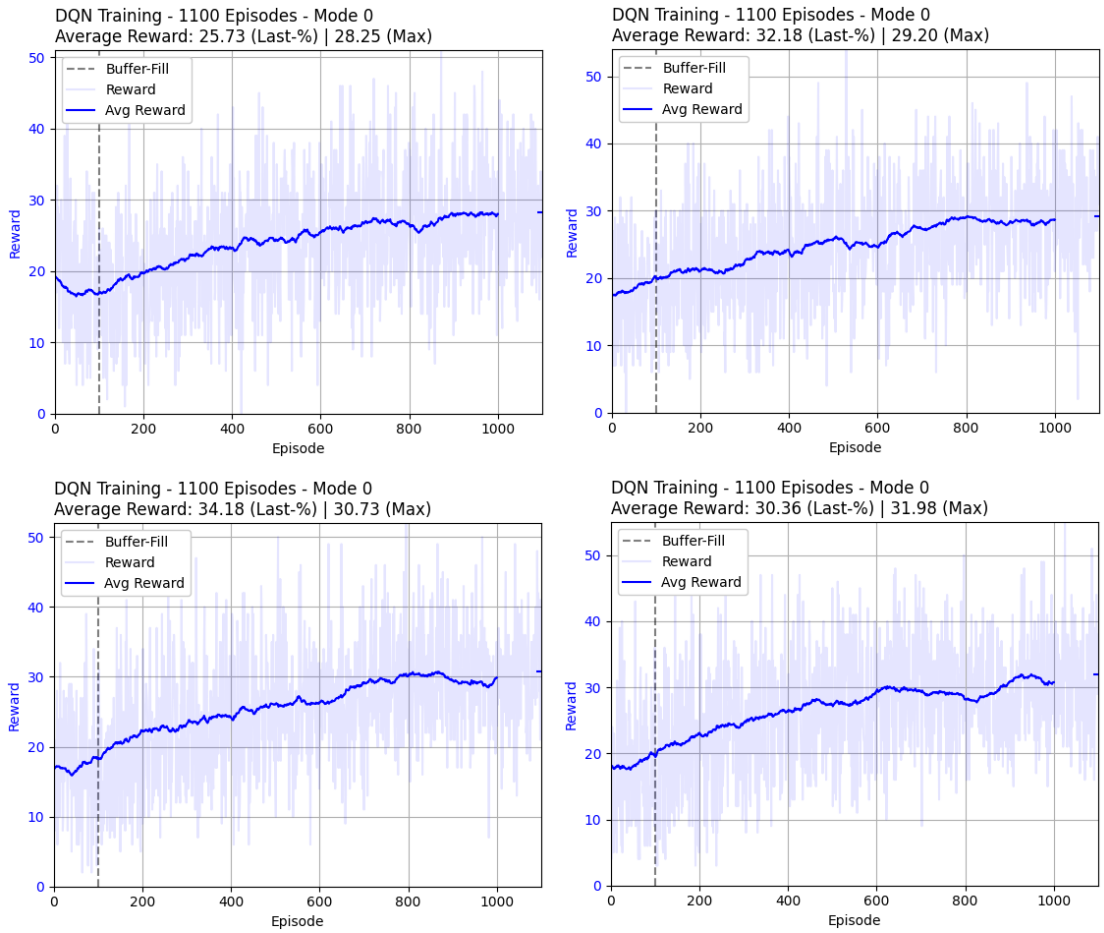


Abbildung 5.8: Training mit Batch-Größen 128 und 256 (o.v.l) sowie 512 und 1024 (u.v.l.) – (Modus 0)

auf durchschnittlich 28 Punkte, wurden auch noch die Batch-Größen 256, 512 und 1024 getestet (Abb. 5.8). Tabelle 5.3 zeigt die erreichten Punkte und die benötigte Rechenzeit. Eine weitere Steigerung der Batch-Größe auf 2.048 brachte keine Verbesserung, da ab einem Reward von 31 Punkten ein Plateau erreicht wurde.

Spielmodus 0 – Weitere Anpassungen

Eine Erhöhung der Neuronenanzahl und die Erweiterung um eine zusätzliche verdeckte Schicht wurden getestet, brachten aber keine Verbesserung. Da der Zustandsraum in Spielmodus 0 noch verhältnismäßig klein ist, ist davon auszugehen, dass diese Optimierungen erst in den anderen Spielmodi zu einem besseren Lernverhalten führen. Da mit

Batch-Größe	Reward	Benötigte Rechenzeit
128	28.2	1:20 min
256	29.1	2:00 min.
512	29.5	2:38 min.
1.024	31.5	4:00 min.

Tabelle 5.3: Batch-Größen und durchschnittlicher Reward

Parameter	Alter Wert	Neuer Wert
Batch-Größe	32	1.014
Lernrate	0,001	0,1

Tabelle 5.4: Angepasste Trainings- und Hyperparameter

dem angepassten Satz an Hyperparametern gute Ergebnisse erzielt wurden, die auch an die durchschnittlichen Rewards des naiven Agenten herankommen, wurde im Folgenden das Training in Spielmodus 1 aufgenommen.

5.3.3 Spielmodus 1

Auf Basis der angepassten Hyperparameter (Tab. 5.4) wurde das Verhalten des Agenten in Spielmodus 1, in dem Wiederholungswürfe möglich sind, untersucht. Aufgrund der gestiegenen Komplexität wurde ein DQN mit zwei verdeckten Schichten getestet, wobei eine Neuronenanzahl von 128 und 64 die beste Stabilität geliefert hat. Der Plot in Abb. 5.9) zeigt vergleichbares Verhalten wie zuvor. Im Schnitt werden Wiederholungswürfe bei dieser Einstellung nur 1,78 mal pro Spiel genutzt. Diskontierungsfaktors

Spielmodus 1 – Anpassung der Diskontierungsfaktors

Um den Agenten zu mehr Wiederholungswürfen zu bewegen, wurde das Training mit verschiedenen Diskontierungsfaktoren γ durchgeführt. Die Plots in Abb. 5.10 zeigen das Training mit $\gamma = 0,5$ und $\gamma = 0,95$, bei denen der Agent 2,5 und 5,57 Wiederholungswürfe pro Spiel eingesetzt hat. Gleichzeitig sinkt der durchschnittliche Reward mit steigendem γ, da zwar häufiger neu gewürfelt wird, die Wiederholungswürfe aber keinen Reward erzeugen und die Rewards weiter auseinanderrücken. In Kombination mit der hohen Lernrate ist die Folge an den Verlustkurven in Abb. 5.11 zu beobachten – die Verluste fangen schon bei $\gamma = 0,5$ leicht an zu oszillieren und schaukeln sich für $\gamma \to 0,95$ immer

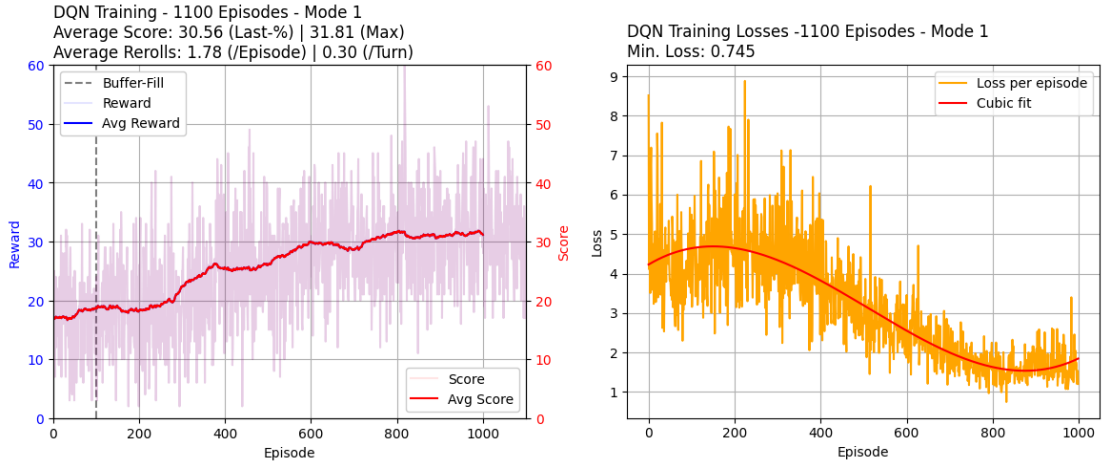


Abbildung 5.9: Training mit angepassten Hyperparametern – (Modus 1)

weiter auf. Um dem vorzubeugen, kann die Lernrate wieder verringert werden, doch das hauptsächliche Problem bleibt weiter bestehen; die Lücken zwischen den Rewards.

5.3.4 Verbesserung des Belohnungssystems

Da die vorangegangenen Anpassungen der Hyperparameter keine signifikanten Fortschritte verzeichnen ließen, wurde der Fokus auf eine bessere Verteilung der Rewards gelegt. Ein lückenhaftes Belohnungssystem stellt eine der größten Herausforderungen beim Reinforcement Learning dar. Der Grund dafür ist, das der Trainingsalgorithmus durch Rewards, die zeitlich versetzt auftreten, oder weiter auseinander liegen (engl. *sparse rewards*), Schwierigkeiten hat, korrekte Rückschlüsse auf einen Zusammenhang zwischen Aktion und Reward zu ziehen [1]. Aus dem pragmatischen Ansatz die Punkte einer Kategorie als Reward an den Agenten zu übergeben, ergibt sich der Nachteil, dass alle Wiederholungswürfe zu einem Reward von Null führen. Um das Lernverhalten zu verbessern, wurden fünf zusätzliche Belohnungssysteme mit dem Ziel entworfen, Wiederholungswürfe direkt oder indirekt mit Rewards zu verbinden:

Cat-Only: Nur Kategorie-Reward

Das bisher genutzte Belohnungssystem. Kategorie-Aktionen führen zu einem direkten Reward R_w in Höhe der erzielten Punkte S_C, während Wiederholungswürfe immer zu

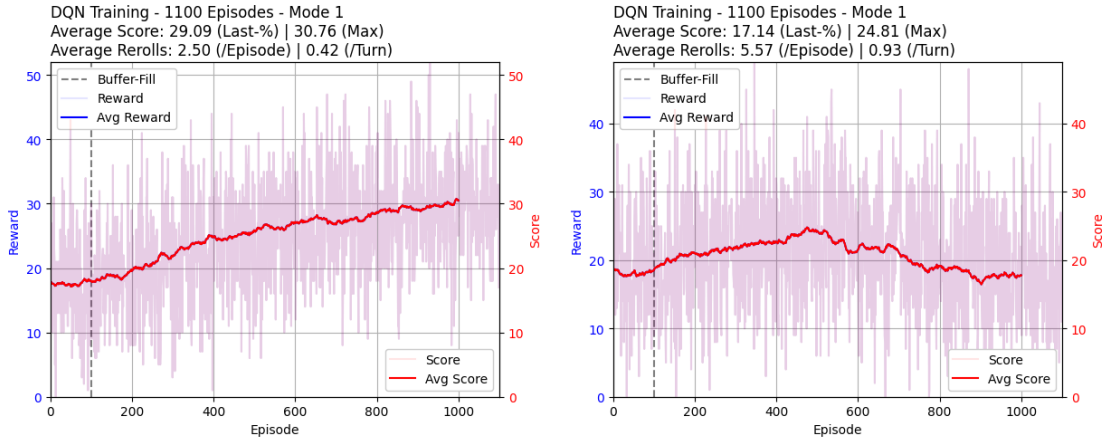


Abbildung 5.10: Rewards bei angepasstem Diskontierungsfaktor $\gamma = 0,5$ und $\gamma = 0,95$ (v.l.) – (Modus 1)

Abbildung 5.11: Verluste bei angepasstem Diskontierungsfaktor $\gamma = 0,5$ und $\gamma = 0,95$ (v.l.) – (Modus 1)

einem Reward von null Punkten führen.

$$R_w = S_c \tag{5.1}$$

Cat-Multi: Kategorie-Reward · Wiederholungswurf

Dieses Belohnungssystem skaliert den Reward R_w anhand der in einem Zug getätigten Wiederholungswürfe RR und einem zusätzlichen Multiplikator X.

$$R_w = S_C \cdot RR \cdot X \ . \tag{5.2}$$

Mit diesem System soll untersucht werden, inwieweit eine zusätzliche indirekte Gewichtung der Kategorie-Aktionen das Lernverhalten beeinflusst. Es bietet weiterhin den Nachteil, Wiederholungswürfe nicht zu belohnen und eignet sich wahrscheinlich nicht, um eine bessere Punktzahl zu erzielen.

Reroll-Inc: Inkrementierender Wiederholungswurf-Reward

Dieses Belohnungssystem gibt für Kategorie-Aktionen die erreichten Punkte als Reward zurück, inkrementiert aber für jeden Wiederholungswurf eines Spiels intern eine Variable RR_{inc} und gibt diese als Reward aus:

$$\text{Kategorie-Reward: } R_w = S_c \quad \Big| \quad \text{Wiederholungswurf-Reward: } R_w = \sum_{RR_{inc}=1}^{steps} 1 \tag{5.3}$$

An diesem System kann untersucht werden, inwieweit sich die ansteigenden Rewards für Wiederholungswürfe auf das Lernverhalten auswirken. Es ist davon auszugehen, dass die Anzahl der Wiederholungswürfe steigt, aber die Gesamtpunktzahl sinkt, da beliebige Wiederholungswürfe zur Inkrementierung des Rewards führen.

Adaptive-Target: Zielabhängiger adaptiver Reward

Dieses Belohnungssystem verwendet einen Ziel-Reward R_T, der ja nach Aktionsauswahl angepasst wird und über die Episode bestehen bleibt. Für Spielmodus 0 wird $R_T = 63$, die Mindestpunktzahl um einen Bonus zu erhalten, gesetzt. Generell sind drei passende

Würfel je Kategorie im oberen Punkteblock dafür notwendig. Wird diese Anzahl über- oder unterschritten, wird das aus Differenz und Kategorie gebildete Produkt zum Ziel-Reward addiert. Dieses Belohnungssystem basiert auf dem Belohnungssystem *Minimum Delta* aus der Masterthesis [20] von Jan Wolter.

Das Listing 5.1 zeigt den vereinfachten Programmcode dieses Algorithmus.

```
if action in [1,2,3,4,5,6]:
    target_dice_count = dice.histogram(observation["dice"])[action]
    delta_count = target_dice_count - 3
    target_reward += delta_count * action
reward = reward + int(target_reward/cat_n) #cat_n = 6
```

Listing 5.1: Adaptive Target Reward

Dieses System bildet ein besseres Verhältnis zwischen wertvollen Wiederholungswürfen und der Kategorie-Auswahl ab, auch wenn Erstere nicht direkt belohnt werden.

Episode-Based-Average: Retroaktiv gemittelter Episoden-Reward

Dieses Belohnungssystem verwendet den durchschnittlichen Reward einer Episode für alle Zwischenschritte n. Mittels eines Pre-Buffers werden die Erfahrungen und Rewards R_n zwischengespeichert. Am Ende einer Episode wird der durchschnittliche Reward jeder Erfahrung des Pre-Buffers als neuer Reward zugewiesen. Der Replay-Buffer erhält den Inhalt des Pre-Buffers und dieser wird anschließend wieder geleert.

$$R_w = \sum_{n=1}^{n_{max}} = \frac{R_n}{n_{max}} \tag{5.4}$$

Turn-Based-Shared: Retroaktiv verteilter Spielzug-Reward

Dieser Belohnungssystem nutzt einen Pre-Buffer als Zwischenspeicher für alle Erfahrungen eines Spielzugs. Sobald eine Kategorie-Aktion auftritt, wird der erzeugte Reward allen im Pre-Buffer befindlichen Erfahrungen zugewiesen. Der Inhalt der Pre-Buffers wird anschließend in den Replay-Buffer geschrieben und geleert. Das folgende Codebeispiel zeigt das Vorgehen:

```
turn_prebuffer.append(exp)
if action <= categories_n:
    for exp in turn_prebuffer:
        replay_buffer.append([exp[0], exp[1], reward, exp[3], exp[4]])
    turn_exp_prebuffer.clear()#Clear prebuffer
```

Listing 5.2: Turn-Based-Shared

Dieses System bildet ein gutes Verhältnis von Kategorie- und Wiederholungswurf-Auswahl ab. Da nicht der durchschnittliche Reward des Zuges verwendet wird, sollten die Vorhersagen des DQN in Richtung der Aktionen geleitet werden, die sowohl zu einer höheren Anzahl an Wiederholungswürfen führen, als auch eine Kategorie erst dann wählen, wenn dadurch eine höhere Punktzahl zu erwarten ist.

Weitere Belohnungssysteme

Ein Belohnungssystem, das im technischen Bericht *Reinforcement Learning for Solving Yahtzee* [9] von 2018 beschrieben wird, ähnelt der Herangehensweise des zuletzt beschriebenen Spielzug-Rewards. Dort wird jedoch, für Züge in denen keine Punkte erreicht wurden, ein diskontierter Durchschnitts-Reward vergeben. Ein Vorteil ist möglicherweise eine verbesserte Stabilität des Trainings, da die Rewards insgesamt näher beieinander liegen. Wenn die zuvor beschriebenen Belohnungssysteme nicht zu einem optimierten Verhalten führen, soll zusätzlich auch dieser Ansatz untersucht werden.
Des Weiteren wird in der Masterthesis [20] von Jan Wolter aus dem Jahr 2025 ein vielversprechendes Belohnungssystem basierend auf der Vorarbeit [17] von Philip Vasseur genutzt. Nach der dort beschriebenen *Reroll-Utility-Methode* wird die Vorhersage von Kategorie- und Wiederholungswurf-Aktionen getrennt behandelt. Der Agent wählt eine priorisierte Kategorie aus und wird anschließend durch eine Reduzierung des Aktionsraums dazu bewegt Wiederholungswürfe zu wählen. Diese können anschließend ausgewertet und die beste Aktionskette zum Trainieren des DQN genutzt werden. Der Ansatz wird im Rahmen dieser Arbeit nicht untersucht, da durch die direkte Manipulation des Aktionsraums stark vom Konzept eines MDP abgewichen wird. Stattdessen soll der Fokus darauf gelegt werden, das Lernverhalten ausschließlich über Rewards zu steuern.

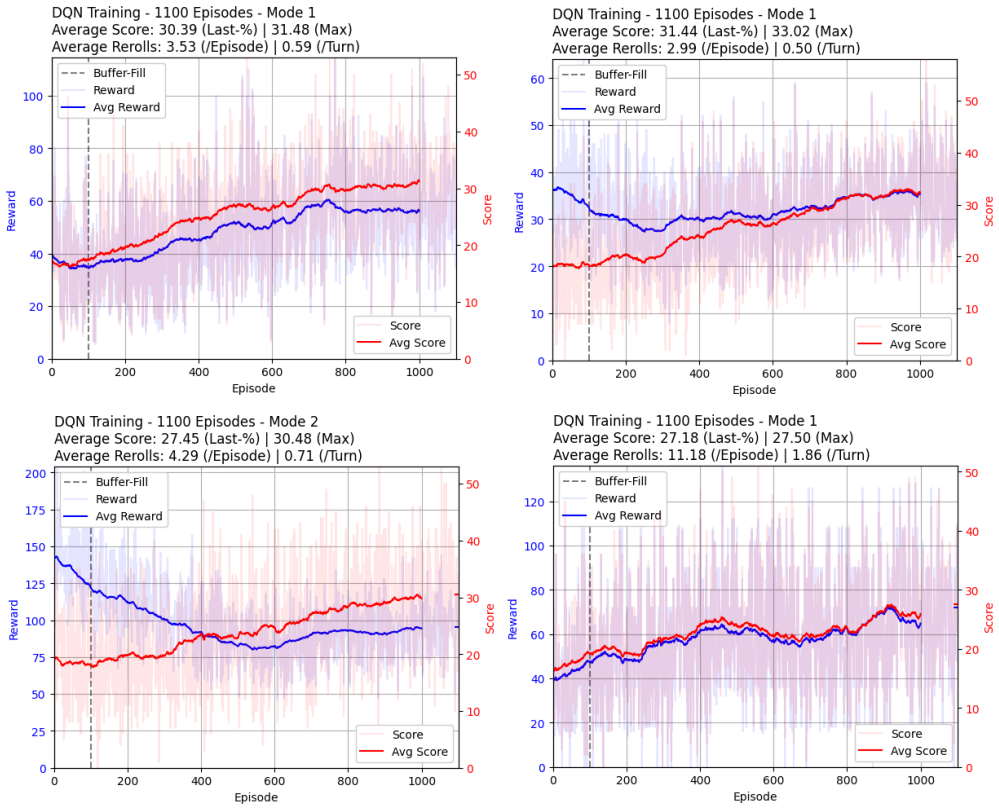


Abbildung 5.12: Belohnungssysteme: Cat-Multi und Reroll-Inc (o.v.l), Adaptive-Target und Episode-Based-Average (u.v.l.) – (Modus 1)

5.3.5 Testen der Belohnungssysteme

Durch das Testen der Belohnungssysteme *Cat-Multi*, *Reroll-Inc*, *Adaptive-Target* und *Episode-Based-Average* konnten sich keine neuen Erkenntnisse gewinnen. Die in Abb. 5.12 dargestellten Plots zeigen, dass das Belohnungssystem *Reroll-Inc* durchschnittlich am besten funktioniert, wobei die Unterschiede geringfügig ausfielen. Hervorzuheben ist die Tatsache, dass der Lernfortschritt in allen Fällen weiterhin positiv verlief, selbst beim *Episode-Based-Average*.

Turn-Based-Shared

Dieses Belohnungssystem konnte sich im Test gegen die vorherigen Systeme behaupten. Letztere erreichten alle ein Plateau ab etwa 32 Punkten. Mit dem *Turn-Based-Shared-*

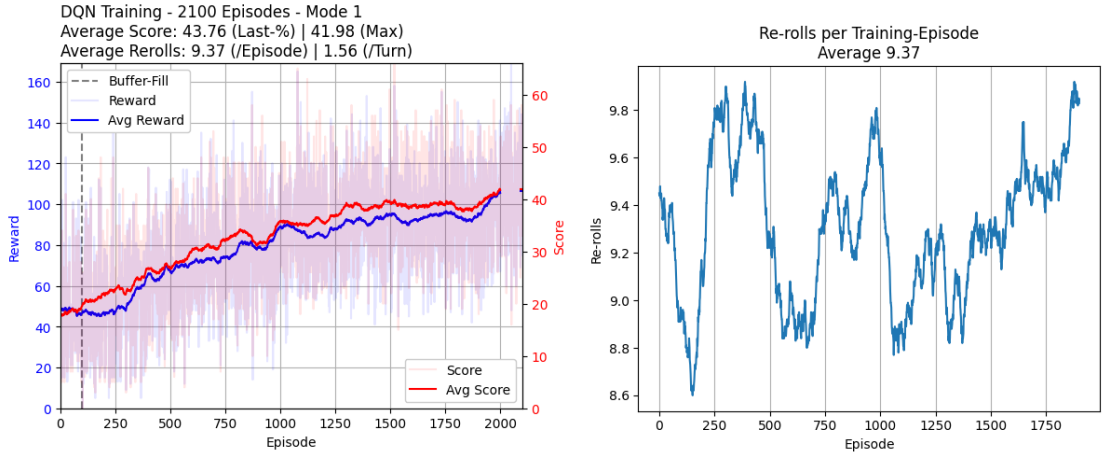


Abbildung 5.13: Turn-Based-Shared Belohnungssystem: Rewards und Wiederholungswürfe je Episode – (Modus 1)

System konnte diese Limitierung überwunden werden. Abbildung 5.13 zeigt, dass die Steigung der Lernkurve zwar nicht verbessert werden konnte, die durchschnittlichen Rewards im Verlauf von 2000 Episoden aber auch nach Erreichen der bisherigen Grenze weiter anstiegen.

5.3.6 Spielmodus 3

Aufgrund der positiven Ergebnisse des in in Abs. 5.3.5 getesteten *Turn-Based-Shared*-Belohnungssystems wurde Spielmodus 2 übersprungen und Tests an Spielmodus 3, welcher das komplette Yahtzee-Spiel abbildet, begonnen. Das Belohnungssystem wurde zuvor erweitert, um die Bonuspunkte in den Rewards abbilden zu können

Erweiterung: *Turn-Based-Bonus*: Spielzug-Reward mit Bonus

Da sich das Belohnungssystem *Turn-Based-Shared* bewährt hat, wird auf dessen Basis eine Erweiterung implementiert, die auch die Bonuspunkte aus dem oberen Punkteblock mit berücksichtigt. Dafür kommt ein zweiter Pre-Buffer zum Einsatz. Die nach dem bisherigen Vorgehen gemittelten Spielzug-Rewards werden dieses Mal bis zum Ende der Episode im zweiten Pre-Buffer abgelegt. Bei Erreichen der Bonuspunkte werden diese auf alle Erfahrungen der Episode verteilt und dann an den Replay-Buffer übergeben.

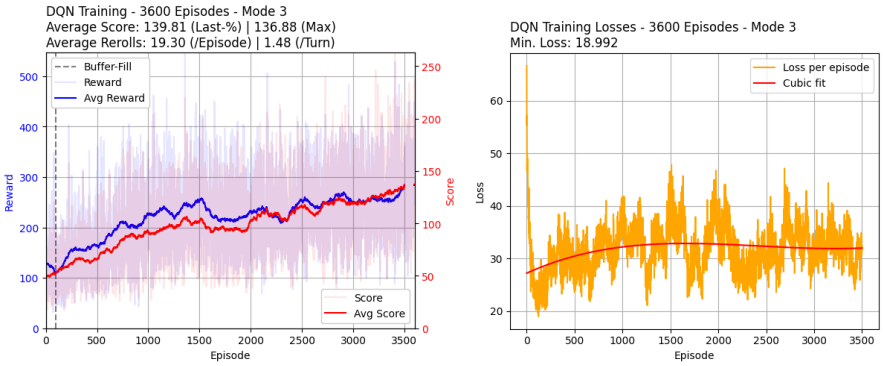


Abbildung 5.14: Durchschnittliche Rewards über 3.500 Episoden – (Modus 3)

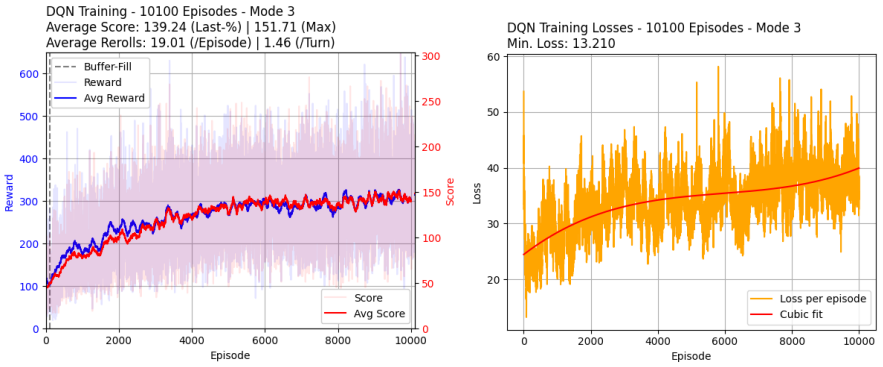


Abbildung 5.15: Durchschnittliche Rewards über 10.000 Episoden – (Modus 3)

Test des *Turn-Based-Bonus*-Belohnungssystems

Die Tests in Spielmodus 3 zeigten eine steigende Lernkurve weit über 1.000 Episoden hinaus. Um das Limit des erreichbaren durchschnittlichen Rewards zu ermitteln, wurde die Batch-Größe auf 2.000 angehoben und die Episodenanzahl auf 3.500 erhöht. In Abb. 5.14 ist erkennbar, dass die durchschnittlichen Rewards auch über die neue Episodenanzahl hinweg steigend bleiben. Erst die Erhöhung auf 10.000 Episoden machte eine neue Limitierung sichtbar. Die Lernkurve in Abb. 5.15 zeigt einen stark abflachenden Verlauf ab 6.000 Episoden und einen durchschnittlichen Reward von 145 Punkten. Damit liegt die durchschnittliche Punktzahl dieses Modells etwa 30 Punkte über dem in Abs. 5.3.1 ermittelten Richtwert. Trotzdem bleibt das Modell damit weit hinter der durchschnittlichen Punktzahl die menschliche Spieler im Schnitt erreichen. Hier werden je nach Quelle[2] zwischen 250 und 270 Punkte als Durchschnitt angegeben.

[2]The Yahtzee Manifesto: `https://www.yahtzeemanifesto.com/yahtzee-odds.php`

6 Auswertung der Ergebnisse

Die in Abschnitt 3.3 definierten Anforderungen sollen an dieser Stelle daraufhin überprüft werden, ob sie erfüllt werden konnten.

VU-F1: [✓] Das Spiel kann im Vorfeld konfiguriert werden.
Diese Anforderung wurde erfüllt, indem verschiedene Spielmodi definiert und implementiert wurden. Eine freie Zusammenstellung der Spielparameter ist nicht vorgesehen, aber es kann eine von vier Konfigurationen gewählt werden.

VU-F2: [✓] Ein Modell kann trainiert werden.
Diese Anforderung wurde erfüllt. Modelle können durch die gegebene Trainingsumgebung trainiert werden.

VU-F3: [✓] Modellparameter können in einer Datei gespeichert werden.
Diese Anforderung wurde erfüllt. Die in TensorFlow/Keras vorhandene Funktion zum Speichern von Modellen wurde erweitert. Trainierte Modelle werden automatisch unter einem eindeutig identifizierbaren Dateinamen mit der aktuellen Konfiguration abgespeichert.

VU-F4: [✘] Modellparameter können aus einer Datei geladen werden.
Diese Anforderung wurde nicht erfüllt. Sie wurde nur schwach priorisiert und und die Funktion nicht fertig implementiert, da das Laden von Modelparametern in der aktuellen Umsetzung nicht gebraucht wurde.

VU-F5: [✓] Der Trainingsfortschritt kann durch eine geeignete Darstellung evaluiert werden.
Diese Anforderung wurde erfüllt. Durch das Plotten der Lernkurven konnte der Trainingsfortschritt ausgewertet werden.

VU-F6: [✘] Es kann ein Spiel gegen ein Modell durchgeführt werden.
Diese Anforderung wurde nicht erfüllt. Durch den zeitlich vorgegebenen Rahmen wurde der Fokus darauf gelegt ein optimales Training des Modells zu ermöglichen.

Abbildung 6.1: Unfertige CLI Anwendung

VU-F7: [–] Es gibt eine textbasierte Oberfläche zur Visualisierung der Einstellungen und des Spiels.
Diese Anforderung wurde teilweise erfüllt. Die Spielklassen besitzen die Funktionalität zur Visualisierung des Spielgeschehens in einem Kommandozeilenfenster, die Logik zur Eingabe besteht nur in Grundzügen (Abb. 6.1).

VU-F8: [✓] Es ist möglich, verschiedene Konfigurationen für das Training zu einzustellen.
Diese Anforderung wurde erfüllt. Durch die Implementation der Trainingsschleife in einem Jupyter-Notebook wurde ein getrennter Bereich für alle Trainingseinstellungen angelegt.

VU-F9: [✘] Es ist möglich, zwischen verschiedenen trainierten Modellen zu wählen.
Diese Anforderung wurde nicht erfüllt. Durch die unfertige Umsetzung der Ladefunktion ist auch die Auswahl zwischen Modellen nicht möglich.

VU-F1: [✓] Die Programmiersprache Python wird für die Implementation genutzt.
Diese Anforderung wurde erfüllt. Die Spiellogik, DQN-Agent und das DQN wurden in Python implementiert.

VU-F1: [✓] Der Programmcode ist gut strukturiert, dokumentiert und mit sinnvollen Kommentaren versehen.
Diese Anforderung wurde erfüllt. Es wurde viel Wert auf eine übersichtliche Implementierung gelegt, die sich an gängigen Standards orientiert.

VU-F1: [–] Alle Module des Programms sollen über Testfunktionen verfügen.
Diese Anforderung wurde teilweise erfüllt. Die Spielklassen verfügen über automatische Testfunktionen, über die eine korrekte Funktionsweise überprüft werden konnte. Für die Klasse des DQN-Agenten wurden nur teilweise Testfunktionen implementiert.

7 Fazit und Ausblick

Die Umsetzung im Rahmen dieser Arbeit hat gezeigt, dass nach wie vor ein großes Verbesserungspotential in der Verteilung der Rewards liegt, was bereits aus anderen Arbeiten [17, 9, 20] hervorgegangen ist. Die Anwendung von Deep-Q-Learning am Beispiel von Yahtzee konnte dennoch überprüft werden, auch wenn die erreichte Leistung weit unter dem menschlichem Niveau bleibt. Auch hat sich gezeigt, dass die Nutzung von (mehrschichtigen) neuronalen Netzen zielführend ist, sobald RL in Umgebungen mit unüberschaubar großen Zustandsräumen eingesetzt wird.

Eine große Hürde bei vielen Softwareprojekten ist die korrekte Einschätzung des Verlaufs im Vorfeld. Dieser Aspekt kam auch bei dieser Arbeit zum Tragen, da manche Bereiche ein vielfaches der zuvor veranschlagten Bearbeitungszeit in Anspruch genommen haben. Insbesondere die Fehlersuche hin zu ersten Lernerfolgen stellte ein großes Hindernis dar. Die Einplanung möglicher Alternativen sollte daher fester Bestandteil jedes Projektplans sein.

Durch weitere Anpassungen am Belohnungssystem und eine optimierte Trainingsstruktur könnte die Leistungsfähigkeit der Modelle weiterhin steigen und auch das Niveau eines menschlichen Spielers erreichen. Ab diesem Punkt gewinnt die die Umsetzung einer grafischen Oberfläche zum Spielen gegen den DQN-Agenten an Relevanz.

Die Anwendung von Deep-Q-Learning bei Atari-Spielen [14] im Jahr 2015 zeigte zwar die große Flexibilität dieser Technik, dennoch sind unterschiedlich strukturierte Umgebungen teilweise mit einem undurchschaubaren Aufwand an Vorverarbeitung verbunden. Zukünftige Arbeiten könnten an dieser Stelle die Möglichkeit untersuchen, ob sich generalisierte Belohnungssysteme entwerfen lassen, welche sich auf ein breiteres Spektrum an Umgebungen bzw. Spielen anwenden lässt.

Literaturverzeichnis

[1] Alexander Zai, Brandon B.: *Einstieg in Deep Reinforcement Learning: KI-Agenten mit Python und PyTorch programmieren.* 1. Auflage. Carl Hanser Verlag GmbH & Company KG, 2020

[2] Alushi, Klejda: *Exploration of Reinforcement Learning Methods when Training a Dice Game Playing Agent*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorthesis, 2022

[3] Azevedo, FA ; Carvalho, LR ; Grinberg, LT u. a.: Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. In: *J. Comp. Neurol.* 513 (2009), Nr. 5

[4] Baevski, Alexei ; Zhou, Henry ; Mohamed, Abdelrahman ; Auli, Michael: *wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations.* 2020. – URL `https://arxiv.org/abs/2006.11477`

[5] Bellman, Richard: A Markovian Decision Process. In: *Journal of Mathematics and Mechanics 6* (1957), Nr. 5

[6] Graves, Alex ; Liwicki, Marcus ; Fernández, Santiago ; Bertolami, Roman ; Bunke, Horst ; Schmidhuber, Jürgen: A Novel Connectionist System for Unconstrained Handwriting Recognition. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31 (2009), Nr. 5, S. 855–868

[7] Gridin, Ivan: *Practical Deep Reinforcement Learning with Python: Concise Implementation of Algorithms, Simplified Maths, and Effective Use of TensorFlow and PyTorch.* 1. Auflage. BPB Publications, 2022

[8] Géron, Aurélien: *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* 2. Auflage. O'Reilly Media, 2019

[9] Kang, Minhyung ; Schroeder, Luca: Reinforcement Learning for Solving Yahtzee / Stanford University. 2018. – Forschungsbericht

[10] Keskar, Nitish S. ; Mudigere, Dheevatsa ; Nocedal, Jorge ; Smelyanskiy, Mikhail ; Tang, Ping Tak P.: *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017

[11] Larsson, Marcus ; Sjöberg, Andreas: *Optimal Yatzy Strategy*, KTH Royal Institute of Technology, Bachelorthesis, 2012

[12] McCulloch, W. ; Pitts, W.: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *The Bulletin of Mathematical Biophysics* (1943)

[13] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: Playing Atari with Deep Reinforcement Learning, 2013

[14] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg ; Petersen, Stig ; Beattie, Charles ; Sadik, Amir ; Antonoglou, Ioannis ; King, Helen ; Kumaran, Dharshan ; Wierstra, Daan ; Legg, Shane ; Hassabis, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), 02, Nr. 7540, S. 529–533

[15] Rashid, Tariq: *Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python*. 2. Auflage. Bookwire GmbH, 2024

[16] Rosenblatt, Frank: The perceptron – A perceiving and recognizing automaton / Cornell Aeronautical Laboratory. 1957 (85-460-1). – Forschungsbericht

[17] Vasseur, Philip: Using Deep Q-Learning to Compare Strategy Ladders of Yahtzee / Yale University. 2019. – Forschungsbericht

[18] Watkins, Christopher J. C. H. ; Dayan, Peter: Q-learning. In: *Machine Learning* 8 (1992)

[19] Wilcox, Ethan ; Qian, Peng ; Futrell, Richard ; Kohita, Ryosuke ; Levy, Roger ; Ballesteros, Miguel: Structural Supervision Improves Few-Shot Learning and Syntactic Generalization in Neural Language Models. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, 2020, S. 4640–4652

[20] Wolter, Jan: *Deep Reinforcement Learning zur Maximierung des Gewinns eines Würfelspiels*, Hochschule für Angewandte Wissenschaften Hamburg, Masterthesis, 2025

[21] Woodward, Phil: Yahtzee®: The Solution. In: *CHANCE* 16 (2003), Nr. 1, S. 18–22

[22] Zhang, Cha ; Zhang, Zhengyou: Improving multiview face detection with multi-task deep convolutional neural networks. In: *IEEE Winter Conference on Applications of Computer Vision*, 2014, S. 1036–1041

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original