HAW HAMBURG

Masterarbeit
Jan Erich Klaus Wolter

Deep Reinforcement Learning zur Maximierung des Gewinns eines Würfelspiels

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

Jan Erich Klaus Wolter

Deep Reinforcement Learning zur Maximierung des Gewinns eines Würfelspiels

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Automatisierung*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Marc Hensel
Zweitgutachter: Prof. Dr. Florian Wenck

Eingereicht am: 11. Juli 2025

Jan Erich Klaus Wolter

Thema der Arbeit

Deep Reinforcement Learning zur Maximierung des Gewinns eines Würfelspiels

Stichworte

Reinforcement Learning, Deep Learning, Deep Reinforcement Learning

Kurzzusammenfassung

In dieser Arbeit wird das Prinzip des Deep Reinforcement Learnings untersucht. Hierzu soll am Beispiel eines Würfelspiels eine Lernumgebung aufgebaut und zwei verschiedene Algorithmen implementiert und evaluiert werden. Am Ende soll ein Demonstrator für das Deep Reinforcement Learning entstehen, der als Grundlage für weiterführende Arbeiten in diesem Bereich genutzt werden kann.

Jan Erich Klaus Wolter

Title of Thesis

Deep reinforcement learning to maximize the win of a dice game

Keywords

Reinforcement Learning, Deep Learning, Deep Reinforcement Learning

Abstract

This thesis examines the principle of deep reinforcement learning. For this purpose, a learning environment will be set up using the example of a dice game and two different algorithms will be implemented and evaluated. In the end, a demonstrator for deep reinforcement learning will be created that can be used as a basis for further work in this area.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Folgenden werden die Motivation und das Ziel der Arbeit beschrieben, um einen Überblick zugeben.

1.1 Motivation

Künstliche Intelligenz (KI) zählt heute zu den einflussreichsten Technologien der digitalen Transformation. Ihre Einsatzmöglichkeiten reichen von der Bild- und Spracherkennung über automatisierte Entscheidungsprozesse bis hin zur Optimierung komplexer Systeme in Wirtschaft und Gesellschaft. In jüngerer Vergangenheit konnte insbesondere der Bereich der industriellen Prozessoptimierung durch KI signifikante Fortschritte erzielen. So gelang es beispielsweise Google, mithilfe intelligenter KI-Systeme den Energieverbrauch seiner Rechenzentren um bis zu 40% zu reduzieren – durch eine automatisierte, adaptive Steuerung der Kühlsysteme auf Basis lernender Algorithmen [7]. Auch in Bereichen wie der Verkehrssteuerung, der Logistik oder der dynamischen Preisgestaltung hat KI bereits messbaren Einfluss auf Effizienz, Kosten und Nutzererfahrung. [1, 33, 18]

Ein besonders aktives Forschungsfeld innerhalb der KI ist das sogenannte Deep Reinforcement Learning (DRL). DRL kombiniert das klassische Reinforcement Learning (RL), bei dem Agenten durch Rückmeldung aus der Umgebung lernen, optimale Handlungen zu wählen, mit Deep Learning (DL). Dieses ermöglicht es auch komplexe, hochdimensionale Zustände effizient zu verarbeiten. Die Verbindung macht DRL besonders geeignet für anspruchsvolle Aufgaben mit großen Zustandsräumen und unsicheren Handlungskonsequenzen. In der Praxis findet DRL heute Anwendung in so unterschiedlichen Feldern wie dem autonomen Fahren, der Robotik oder dem algorithmischen Handel. Ein prominentes Beispiel für den Erfolg dieser Methode ist der historische Sieg des DRL-Systems AlphaGo von Google DeepMind über den damaligen Weltmeister Lee Sedol im Jahr 2016. [2, 32]

Angesichts der zunehmenden Bedeutung von DRL in Forschung und Praxis stellt sich die Frage, wie diese Methode im Detail funktioniert, welche Konzepte ihr zugrunde liegen und wie sich ihr Lernverhalten anhand praktischer Beispiele untersuchen lässt. Ziel dieser Arbeit ist es daher, die Methodik des Deep Reinforcement Learning systematisch zu analysieren und am Beispiel des Würfelspiels Yahtzee experimentell zu erproben.

Das Spiel Yahtzee bietet sich aus mehreren Gründen als geeignetes Testfeld für DRL-Ansätze an. Zum einen sind die Spielregeln vergleichsweise einfach und gut formal beschreibbar. Zum anderen lässt sich das Spiel in digitaler Form problemlos simulieren, was eine effiziente Datengewinnung für das Training von Agenten ermöglicht. Besonders relevant ist darüber hinaus die Balance zwischen Zufall und Strategie, die das Spiel auszeichnet: Während Spiele wie Schach oder Go stark deterministisch sind, spielt beim Würfeln das Zufallselement eine zentrale Rolle. Dies stellt spezielle Anforderungen an die Modellierung des Lernverhaltens eines Agenten, insbesondere in Bezug auf Unsicherheit, Exploration und langfristige Planung.

Im Rahmen dieser Arbeit soll zunächst ein einfacher Reinforcement-Learning-Algorithmus entwickelt und auf das Spiel Yahtzee angewendet werden. Anschließend wird dieser Ansatz um Deep Learning-Komponenten erweitert, um auch komplexere Zustandsräume verarbeiten zu können. Ein besonderer Fokus liegt auf dem Vergleich zwischen kurzfristigem und langfristigem Lernen sowie auf der Untersuchung des Einflusses der Zufallskomponente auf das Lernverhalten. Frühere Studien [5, 14, 35] bieten dabei eine Grundlage, auf die in der vorliegenden Arbeit aufgebaut wird.

Die Ergebnisse dieser Arbeit sollen sowohl einen praxisnahen Einstieg in die Thematik des Deep Reinforcement Learnings ermöglichen als auch ein neuartiges, reproduzierbares Beispiel für weiterführende Forschungsarbeiten in diesem Bereich liefern.

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit ist die Untersuchung und Anwendung von Deep Reinforcement Learning (DRL) am Beispiel des Würfelspiels Yahtzee. Zu diesem Zweck soll ein DRL-Algorithmus entwickelt, implementiert und hinsichtlich seiner Leistungsfähigkeit im Hinblick auf die Erreichung einer möglichst hohen Punktzahl evaluiert werden. Ein besonderer Fokus liegt auf der Analyse des Lernverhaltens auf dem Spielergebnis, was durch

geeignete Visualisierungen veranschaulicht werden soll.

Darüber hinaus wird angestrebt, einen interaktiven Demonstrator zu entwickeln, der es ermöglicht, die untersuchten Algorithmen praktisch zu erproben und deren Eigenschaften nachvollziehbar zu machen. Ein weiterer Aspekt der Arbeit ist der Vergleich der Leistungsfähigkeit der Algorithmen in Abhängigkeit von der Trainingsdauer (kurzes vs. langes Training). Zusätzlich wird die Möglichkeit untersucht, das Spiel mithilfe einer Kamera und realen Würfeln zu realisieren, um eine Mensch-Maschine-Interaktion zu ermöglichen und von der rein digitalen Zufallszahlengenerierung unabhängig zu werden.

Ziel ist es, durch dieses praxisnahe Projekt vertiefte Erkenntnisse im Bereich des Deep Reinforcement Learnings zu gewinnen und Erfahrungen im Umgang mit entsprechenden Methoden und Technologien zu sammeln.

2 Grundlagen

Nachdem die Motivation und das Ziel der Arbeit erörtert wurden, wird im Folgenden die Grundlagen dieser Arbeit beschrieben. Hierzu wird auf alle notwendigen Grundlagen für das Reinforcement Learning (RL) und das Deep Learning (DL) eingegangen und ein paar für diese Arbeit interessante Algorithmen vorgestellt. Zuerst werden die Spielregeln von Yahtzee erörtert.

2.1 Spielregeln

Die Spielregeln von Yahtzee werden nach [10] und [9] beschrieben. Yahtzee ist für einen oder mehrere Spieler. Es werden fünf Würfel, ein Würfelbecher und eine oder mehrere Punktetabellen benötigt. Die Punktetabelle ist in der Abbildung 2.1 dargestellt und setzt sich aus einem oberen und unteren Tabellenteil zusammen.

Yahtzee ist ein rundenbasiertes Spiel und besteht aus 13 Runden. Ziel ist es in jeder Runde die fünf Würfel zu würfeln, um verschiedene Würfelkombinationen zu erzielen. In

Upper Section	What to Score
Aces (Ones)	Total of Aces only
Twos	Total of Twos only
Threes	Total of Threes only
Fours	Total of Fours only
Fives	Total of Fives only
Sixes	Total of Sixes only

Lower Section	What to Score
3 of a Kind	Total of all 5 dice
4 of a Kind	Total of all 5 dice
Full House	25 points
Small Straight	30 points
Large Straight	40 points
YAHTZEE (5 of a Kind)	50 points
Chance	Total of all 5 dice

Abbildung 2.1: Punktetabelle Yahtzee aus [10].

jeder Runde stehen dem Spieler nach dem ersten Wurf bis zu zwei weitere Würfe zu. Nach jedem Wurf kann die Person entscheiden, ob sie den Wurf in die Punktetabelle eintragen oder noch einmal würfeln möchte. Die Person darf nach jedem Wurf entscheiden, welche der fünf Würfel sie noch einmal würfeln möchte. Die Würfel, die sie behalten möchte, legt sie beiseite. Spätestens nach dem dritten Wurf muss die Person sich für ein Feld auf der Punktetabelle entscheiden, in der sie das Ergebnis einträgt und die entsprechenden Punkte dafür erhält. Im oberen Tabellenteil werden immer Würfel mit der gleichen Augenzahl zusammengezählt und die Summe in das entsprechende Feld eingetragen. Des Weiteren wird Einem ein Bonus von 35 Punkten gewährt, wenn die gesamte Punktzahl für den oberen Tabellenteil mindestens 63 Punkte beträgt. Im unteren Tabellenteil sind nach Abbildung 2.1 verschiedene Kombinationen möglich. Bei einem Dreierpasch werden mindestens drei Würfel mit der gleichen Augenzahl benötigt und beim Viererpasch entsprechend vier. Eine Eintragung in das Feld Full House ist möglich, wenn drei Würfel die gleiche Zahl und zwei Würfel eine gleiche andere Zahl zeigen. Die kleine Straße setzt sich aus vier aufeinander folgende Zahlen zusammen und die große Straße aus fünf. Bei einem Yahtzee müssen alle fünf Würfel dieselbe Zahl aufweisen. Zum Schluss steht jedem eine Chance zur Verfügung, in der die Summe der fünf Würfel eingetragen werden kann. In dieser Arbeit sollen die Regeln bzgl. der Anwendung des Jokers und des Yahtzee Bonus nicht weiter berücksichtigt werden.

Gewonnen hat der Spieler, der zum Schluss die meisten Punkte besitzt. Deshalb ist neben dem Würfelglück eine entsprechende Strategie nicht unbedeutend, um die Gesamtpunktzahl zu maximieren.

2.2 Reinforcement Learning

Das Reinforcement Learning (RL, dt. das bestärkende Lernen) gehört zu den drei großen Bereichen des Machine Learnings (ML). Die beiden anderen Bereiche sind das überwachte und unüberwachte Lernen. Das Ziel von RL ist es mit der Zeit zu erlernen optimale Entscheidungen zu treffen [16]. Im Gegensatz zum überwachten oder unüberwachten Lernen, wo Datensätze für das Training eines Algorithmus notwendig sind, befindet sich beim bestärkenden Lernen der Algorithmus oder auch der Agent in einer definierten Umgebung. In dieser Umgebung kann der Agent festgelegte Aktionen ausführen. Das Ausführen dieser Aktionen führt zu einer Veränderung des Ursprungszustands innerhalb der Umgebung und gibt dem Agent eine unmittelbare Belohnung. Die Belohnung kann positiv als auch

negativ sein. Die Veränderung des Zustands innerhalb der Umgebung bewertet der Agent neu und führt eine neue Aktion aus. Hier gilt es zu beachten, dass immer nur eine Aktion zu einem bestimmten Zeitpunkt ausgeführt werden kann, also ein sequenzieller Ablauf stattfindet. Der Agent besitzt keinerlei Vorwissen und muss zu Beginn die Umgebung erst einmal erkunden und Erfahrung sammeln. Ist der Erkundungsprozess abgeschlossen, kann der Agent die Umgebung zu seinen Gunsten nutzen. Interessant ist, dass Entscheidungen, die der Agent am Anfang trifft, die Belohnung auch zu einem späteren Zeit Punkt beeinflussen kann und somit sich auf die maximal mögliche Belohnung auswirkt. Ein bekanntes Beispiel aus der Praxis ist unter anderem das Trainieren von Hunden, um das gewünschte Verhalten zu bestärken oder schlechtes Verhalten zu bestrafen. Ein weiteres Beispiel ist das Entkommen einer Maus aus dem Labyrinth, die mit einem Stück Käse belohnt wird. Im Folgenden sind die wichtigsten Begriffe noch einmal zusammengefasst: [34, 7, 16]

- **Der Agent** *(engl. agent)* trifft die Entscheidung, welche Aktion als Nächstes ausgeführt werden soll.
- **Die Umgebung** *(engl. environment)* ist der Raum, in dem der Agent interagieren darf, um die Aufgabe oder das Problem zu lösen.
- **Der Zustand** *(engl. state)* ist die Position des Agenten in der Umgebung und spiegelt die momentan beobachtete Situation wider.
- **Die Aktion** *(engl. action)* ist eine Handlung, die ausgeführt wird und zu einer Zustandsveränderung innerhalb der Umgebung führt.
- **Die Belohnung** *(engl. reward)* ist eine Zahl, welche die Belohnung oder die Bestrafung (bei negativem Vorzeichen) für die ausgeführte Aktion symbolisiert, die der Agent erhält. Die Belohnung kann je nach Aufgabe regelmäßig oder einmalig vergeben werden.

Die Abbildung 2.2 veranschaulicht das Prinzip des Reinforcement Learnings.

2.2.1 Markov-Entscheidungsprozess

Markov-Entscheidungsprozesse (*engl. Markov decision process; MDP*) bilden die Grundlage jeder Reinforcement Learning Aufgabe. Der MDP dient der Modellierung, wofür

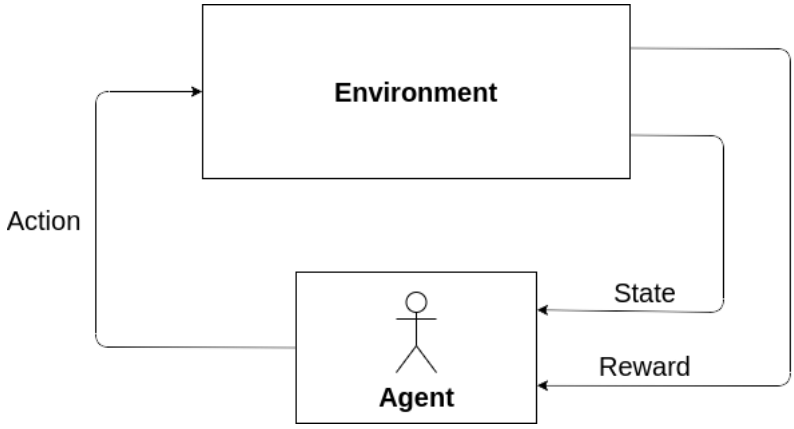


Abbildung 2.2: Prinzip des Reinforcement Learnings aus [7].

das Problem oder die Aufgabe die Markov-Eigenschaft besitzen muss. Die Markov-Eigenschaft beschreibt, dass die beste Aktion für einen bestimmten Zustand ohne direkten Bezug auf andere vorherige Zustände gewählt werden kann. Der Agent befindet sich in einem Zustand und gibt die erwartete Belohnung für die entsprechende Aktion zurück. Für den MDP gilt somit, dass in einem bestimmten Zustand die beste Aktion gewählt werden kann, weil der Zustand alle Informationen aus der Vergangenheit besitzt, die einen Unterschied auf zukünftige Belohnungen haben. Der Agent kann somit durch die Auswahl der besten Aktion seine Belohnung maximieren. [34, 39]

Die Markov-Kette (S, P) setzt sich aus allen Zuständen S zusammen, wobei die Wechsel zwischen den Zuständen durch die Übergangswahrscheinlichkeiten P definiert sind. Wird die Markov-Kette um die Belohnung R erweitert, ist das der Markov-Belohnungsprozess (S, P, R). Wird wiederum diesem Prozess die Aktion A hinzugefügt, entspricht dies dem Markov-Entscheidungsprozess (S, P, R, A). Der Markov-Belohnungsprozess und die Markov-Kette können die Umgebung nicht beeinflussen, sondern nur beobachten. Erst durch das Hinzufügen der Aktion kann das Modell beeinflusst werden und der Agent kann mit dem Modell interagieren. Des Weiteren gilt es zu beachten, dass nicht für jedes Problem die Übergangswahrscheinlichkeiten einfach oder überhaupt zu ermitteln sind. Deshalb kann zwischen zwei Kategorien unterschieden werden - dem modellbasierten und dem modellfreien Lernen. Beim modellbasierten Lernen sind die Gesetzmäßigkeiten der Umgebung bekannt und der Agent versucht daraus die bestmöglichen Handlungen abzuleiten. Beim modellfreien Lernen ist kein Modell notwendig und dem Agenten sind die Gesetzmäßigkeiten der Umgebung nicht bekannt. Dadurch entfällt die komplexe

Modellierung der Umgebung. Zu berücksichtigen gilt auch, dass es sich bei Markov-Entscheidungsprozesse oft um endliche (*engl. finite*) Zustandsräume handelt. Endliche Zustandsräume besitzen einen terminierenden, finalen Zustand. Weitere Abbruchbedingungen sind möglich. [7, 16]

2.2.2 Policy

Eine Policy oder auch Strategie ist eine Funktion $\pi(a|s)$, die einen Zustand auf eine Wahrscheinlichkeitsverteilung über die Menge der möglichen Aktionen in diesem Zustand abbildet [39, 16]. Eine andere Formulierung ist, dass die Wahrscheinlichkeit jeder möglichen Aktion in der Verteilung gleich der Wahrscheinlichkeit ist, dass die gewählte Aktion die größte Belohnung zurückgibt. Das bedeutet, dass eine Aktion aus einer Menge von Aktionen $a \in A$ abhängig vom Zustand s die Wahrscheinlichkeit mit der größten Belohnung G_t verfolgt. Zum Beispiel steht der Agent auf einer Plattform und vor ihm sowie links und rechts befindet sich ein sehr tiefer Abgrund. Er hat vier Richtungen (vorwärts, nach links, nach rechts und rückwärts), in denen er sich bewegen kann. Die Aktion Rückwärtsbewegen besitzt die Wahrscheinlichkeit eins, wohingegen die anderen Drei null sind. Demzufolge ist das Ziel die Strategie zu optimieren, sodass auf lange Sicht die Belohnung maximal wird. Um festzustellen, wie zielführend eine Aktion a oder Zustand s ist, wird eine Werte-Funktion verwendet. Der zu erwartende Gewinn G_t dient hierzu als Vergleichswert. [34]

Policy- und Wert-Funktionen

Die Werte-Funktionen werden unterschieden in die Zustandswertfunktion (engl. *State-Value-Function*) und die Aktionswertfunktion (engl. *Action-Value-Function*). Die Zustandswertfunktion eines Zustandes s, die die Policy π verfolgt, wird als $v_\pi(s)$ bezeichnet und ist die *erwartete Belohnung*, wenn im Zustand s begonnen und danach π weiterverfolgt wird. Für MDPs wird $v_\pi(s)$ formal wie folgt definiert: [34]

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \forall s \in S \tag{2.1}$$

$\mathbb{E}_\pi[\cdot]$ bezeichnet den Erwartungswert einer Zufallsvariablen, wenn der Agent die Policy π verfolgt und t ein beliebiger Zeitschritt ist. v_π ist die Zustandswertfunktion für die Policy π. Ähnlich wird die Aktionswertfunktion q_π für die Policy π definiert. Sie beschreibt den

Erwartungswert den der Agent erhält, wenn dieser beginnend vom Zustand s die Aktion a ausführt und anschließend die Policy π verfolgt. [34]

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s,\ A_t = a\right],\ \forall s \in S,\ \forall a \in A \end{aligned} \tag{2.2}$$

Die Wert-Funkionen v_π und q_π können durch Erfahrung geschätzt werden [34].

Optimale Policy- und Werte-Funktion

Es ist nach wie vor das Ziel, dass der Agent eine Policy findet, die am meisten Belohnungen einbringt. Ist eine Policy π gefunden, wo der Gewinn maximal ist, wird diese als optimale Policy bezeichnet. Hierfür reicht es aus, wenn der zu erwartende Gewinn für alle möglichen Zustände mindestens genauso gut oder besser ist, wie von jeder anderen Policy π'. Demzufolge gilt für die Zustandswertfunktionen $v_\pi(s)$ und $v_{\pi'}(s)$ sowie die Aktionswertfunktionen $q_\pi(s,a)$ und $q_{\pi'}(s,a)$ [34]:

$$\pi \geq \pi',\ \text{wenn}\ v_\pi(s) \geq v_{\pi'}(s),\ \forall s \in S \tag{2.3}$$

$$\pi \geq \pi',\ \text{wenn}\ q_\pi(s,a) \geq q_{\pi'}(s,a),\ \forall s \in S\ \text{und}\ \forall a \in A \tag{2.4}$$

Die optimale Policy wird als π_* bezeichnet, was äquivalent für die optimale Zustandswertfunktion v_* sowie die optimale Aktionswertfunktion q_* gilt. Diese geben immer den maximalen Erwartungswert bzw. die maximale Belohnung zurück und werden wie folgt definiert [34]:

$$v_*(s) \doteq \max_\pi v_\pi(s),\ \forall s \in S \tag{2.5}$$

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a),\ \forall s \in S\ \text{und}\ \forall a \in A \tag{2.6}$$

Zum Schluss soll noch der erwartete Gewinn eines Zustandsaktionspaares (s,a) für das Ausführen einer Aktion a im Zustand s und dem anschließenden Verfolgen der optimalen

Policy in Abhängigkeit von v_* beschrieben werden.

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})\ S_T = s,\ A_t = a] \tag{2.7}$$

Die Grundlage dafür ermöglicht die Optimalitätsgleichung von Bellman, welche in Sutton [34] nachgelesen werden kann.

Epsilon-Greedy-Policy

Eine optimale Policy gibt immer den bestmöglichen Ertrag aus dem aktuellen Zustand zurück. Bis eine bessere Strategie gefunden wird, kann der Agent mit einer optimalen Policy die Umgebung ausbeuten. Demzufolge hat der Agent schon Erfahrung in der Umgebung gesammelt oder ihm wurde diese von Anfang an mitgegeben. Im letzteren Fall hat der Agent nicht selber gelernt, sondern befolgt eine vorgegebene Handlungsdirektive. In diesem Fall würde der Agent immer wieder gleich handeln und keine neuen Erfahrungen machen. Die Umgebung wird somit von dem Agenten nicht weiter erkundet. Das Ziel ist jedoch, dass der Agent selbständig lernt, also die Umgebung erkundet und ausbeutet. Darüber hinaus ist es sinnvoll, wenn der Agent auf lange Sicht gelegentlich etwas Neues in der Umgebung ausprobiert und somit die Erkundung nicht gänzlich einstellt. So kann gewährleistet werden, dass der Agent auch zu einem späteren Zeitpunkt neue Erkenntnisse gewinnen und die Policy optimieren kann. Wie schon Einstein formulierte, können keine neuen Erkenntnisse erlangt werden ohne neue Wege auszuprobieren.

> *"Die Definition von Wahnsinn ist: immer wieder das Gleiche zu tun und andere Ergebnisse zu erwarten."* *Albert Einstein*

Eine der bekanntesten Policies, die die Erkundung und Ausbeutung miteinander vereint, ist die ϵ-Greedy-Policy. Bei dieser Strategie wird die Erkundungsrate (Exploration Rate) ϵ eingeführt, die sich im Bereich von null bis eins bewegt. Die ϵ-Greedy-Strategie wählt mit einer Wahrscheinlichkeit von $1 - \epsilon$ die Aktion, die die höchste bisher erlernte Belohnung verspricht. Eine andere Aktion wird zufällig mit einer Wahrscheinlichkeit von ϵ ausgewählt, um die Erkundung zu fördern. Ein ϵ-Wert von Null bedeutet, dass nur auf das bereits erlernte Wissen zurückgegriffen wird (Ausbeutung der Umgebung), während bei einem Wert von Eins ausschließlich zufällige Aktionen zur Erkundung durchgeführt werden. Die Entscheidung, welche Aktion gewählt wird, basiert auf einer zufällig generierten Zahl zwischen null und eins. Ist diese Zahl kleiner als ϵ, wird eine zufällige Aktion

durchgeführt, andernfalls wird die Aktion mit der höchsten erwarteten Belohnung ausgewählt. Zu Beginn des Trainings wird der ϵ-Wert oft hoch angesetzt, um eine umfassende Exploration zu ermöglichen. Mit fortschreitendem Training wird der Wert schrittweise reduziert, um die Strategie zu verfeinern und eine Konvergenz zu einer optimalen Lösung basierend auf den gesammelten Erfahrungen zu erreichen. [34, 39, 7]

2.3 Deep Learning

Bevor auf die verschiedenen Algorithmen eingegangen wird, soll zuerst das Deep Learning vorgestellt werden. Dies ist ein weiteres Teilgebiet des Machine Learnings. Das Reinforcement Learning in Kombination mit Deep Learning ermöglicht noch mächtigere Lösungsverfahren, weil die Ergebnisse geschätzt werden können. Dadurch wird die ursprüngliche Q-Tabelle, die die Ergebnisse bisher gespeichert hat, obsolet. Das Deep Learning kann im verallgemeinerten Sinne als die Nachbildung der Funktionsweise eines Gehirns beschrieben werden. Dazu werden neuronale Netze (NN) oder auch künstliche neuronale Netze (KNN) konstruiert bzw. implementiert. Diese neuronalen Netze setzen sich wiederum aus verschiedenen Schichten zusammen. In diesen Schichten befinden sich die kleinsten Einheiten eines neuronalen Netzes, die Knoten oder Neuronen genannt werden. [24]

2.3.1 Das Perzeptron

Das Perzeptron beruht auf der Arbeit [28] von Frank Rosenblatt. Es besitzt *Eingabevariablen* von x_1 bis x_n (siehe hellgrau dargestellt in 2.3). Des Weiteren wird der Bias b definiert. Dieser gehört zu den Eingangsvariablen, weshalb er auch aufgrund der Beziehung $x_0 = b$ als x_0 bezeichnet werden kann (siehe dunkelgrau dargestellt in 2.3). Anschließend wird die gewichtete Summe z der Eingabevariablen gebildet und an eine Aktivierungsfunktion übergeben. In der Abbildung 2.3 ist die gewichteten Summe mit einem gelblich gefärbten Rechteck und die Aktivierungsfunktion $\sigma(z)$ mit einem hellblau gefärbten Rechteck symbolisiert. Das Ergebnis oder die Ausgabe nach der Aktivierungsfunktion ist $\hat{y}$.

$$z = \sum_j w_j x_j + b \tag{2.8}$$

$$\hat{y} = \sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2.9}$$

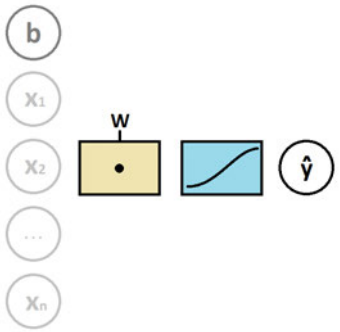


Abbildung 2.3: Darstellung eines Perzeptrons nach Perrotta [24].

Die hier verwendete Aktivierungsfunktion $\sigma(z)$ ist eine Sigmoidfunktion. Die ursprüngliche Form des Perzeptrons kann jedoch nur entweder eine Null oder eine Eins zurückgeben. Sie besitzt also als Aktivierungsfunktion eine Sprungfunktion, die über einen Schwellenwert (Bias) b zwischen eins und null unterscheidet. [22, 24]

$$\hat{y} = \begin{cases} 0 & \text{, wenn } z = \sum_j w_j x_j \leq b \\ 1 & \text{, wenn } z = \sum_j w_j x_j > b \end{cases} \tag{2.10}$$

Bei Betrachtung der zuvor beschriebenen Gleichungen ohne $\sigma(z)$ wird ersichtlich, dass je nach Änderung eines Gewichts oder Bias das Perzeptron andere Ergebnisse als mit den zuvor eingestellten Größen produzieren kann. Die Aktivierungsfunktion kann je nach Bedarf und abhängig von der Aufgabenstellung angepasst werden. Dadurch ist es möglich mehr als nur zwischen null und eins zu unterscheiden. Kleine Änderungen in den Gewichten oder dem Bias verursachen nicht allzu große Änderungen am Ausgang. Dies bringt jedoch auch neue Herausforderungen mit sich. Dazu betrachten wir die oben genannte Aktivierungsfunktion $\sigma(z)$. Diese Funktion eröffnet den Wertebereich zwischen null und eins. Damit jedoch zum Beispiel die Funktion gegen null tangiert, muss $e^{-z} \to \infty$ verlaufen und für eins muss $e^{-z} \approx 0$ sein. [22, 24]

Bisher wurde nur ein einzelner Knoten oder ein einzelnes Neuron betrachtet. Die Stärke des Perzeptrons liegt darin, dass es parallelisiert und serialisiert werden kann. Im Ergebnis lässt sich damit ein neuronales Netz aufspannen.

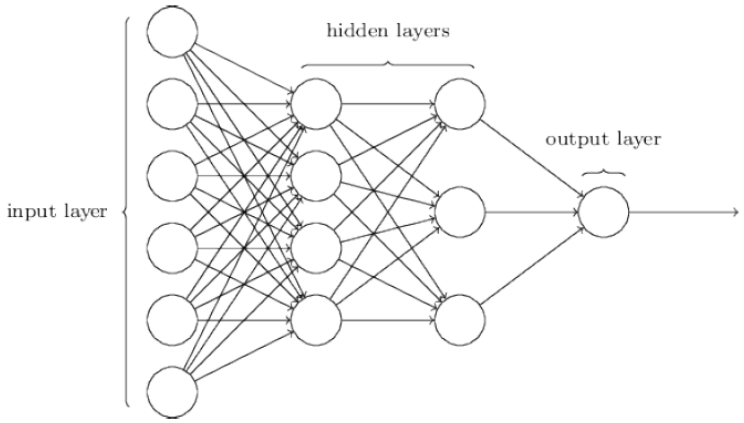


Abbildung 2.4: Aufbau eines tiefen neuronalen Feedforward Netzes [22].

2.3.2 Neuronales Netz

Ein neuronales Netz setzt sich aus mehreren Schichten zusammen. Eine Schicht kann wiederum mehrere Neuronen besitzen. Es besteht aus einer Eingabeschicht (input layer), einer Ausgabeschicht (output layer) und mindestens einer verborgenen Schicht (hidden layer). Wenn ein solches neuronales Netz nur eine verborgene Schicht besitzt, wird es auch als flaches Netz bezeichnet. Die Abbildung 2.4 veranschaulicht dies. Das Deep in Deep Learning steht dafür, dass das neuronale Netz mehr als eine verborgene Schicht besitzt. [22]

Feedforward Netze

Es gibt verschiedene Arten, wie ein neuronales Netz aufgebaut sein kann. Die einfachste Form sind die Feedforward Netze (siehe Abbildung 2.4). Feedforward Netze sind Netze, bei denen der Ausgang der einen Schicht als Eingang für die nächste Schicht verwendet wird. Sie geben die Information immer nur in Vorwärtsrichtung weiter. Das bedeutet, dass es keine Schleifen in dem Netz gibt. Neuronale Netze, die Rückkopplungsschleifen enthalten, werden rekurrente Netze genannt. [22]

Rekurrente Netze

Das Konzept hinter rekurrente Netze ist, dass Neuronen für eine bestimmte Zeitspanne immer wieder aktiviert werden, bevor sie in einen inaktiven Zustand übergehen. Diese wiederholte Aktivierung kann andere Neuronen anregen, die nach einer kurzen Verzögerung ebenfalls erneut aktiviert werden. Auf diese Weise entsteht eine Kaskade von Neuronen, die über einen längeren Zeitraum hinweg immer wieder aktiviert werden. Das Ziel ist es, die Funktionsweise des menschlichen Gehirns noch realistischer nachzubilden, da auch dort Informationen nicht nur direkt von einem Eingang zu einem Ausgang weitergegeben werden. Aus diesem Grund eignet sich das rekurrente neuronale Netzwerk besonders gut für Aufgaben, die die Analyse von Daten oder Prozessen umfassen, die sich im Laufe der Zeit verändern. Das einfachste Beispiel dafür ist, wenn der Ausgang eines Neurons wieder auf einen seiner Eingänge gelegt wird. [22]

Faltungsnetze

Zum Schluss sollen noch die Faltungsnetze (*Convolutional Neural Network, CNN*) erwähnt werden. Sie kommen bei der Bild- und Videoanalyse zum Einsatz. Im Gegensatz zu Feedforward Netzen nutzt das Faltungsnetz keine vollständig verbundenen Schichten. Demzufolge steht nicht jedes Neuron einer Schicht in Verbindung mit jedem Neuron in der nächsten Schicht. Faltungsnetze können dadurch schneller trainiert werden und ermöglichen eine tiefere Netzwerkstruktur. Die Funktionsweise orientiert sich dabei sehr stark an die der Sehrinde des menschlichen Gehirns für die Erkennung von Objekten [25]. Dafür werden die drei Konzepte lokales Rezeptivfeld, geteilte Gewichte und Pooling verwendet. Zum Beispiel bei der Analyse eines Bildes von 100x100 Pixeln wird mit einem kleineren Fenster (lokales Rezeptivfeld) der Größe 5x5 Pixel Stück für Stück betrachtet und je mit einem verborgenem Neuron übergeben. Das Pixelfenster kann auch als Filter der Größe 5x5 betrachtet werden, welches Pixel für Pixel über das Bild geschoben wird und anschließend mit konstanten Gewichten (geteilte Gewichte) multipliziert wird. Geteilte Gewichte bedeuten in diesem Fall, dass im Gegensatz zum Feedforward Netz im Faltungsnetz nur eine Teilmenge an konstanten Gewichten benötigt wird. Das Pooling trägt dazu bei eine gewisse lokale Invarianz zu gewährleisten, was bedeutet, dass geringe Veränderungen in der unmittelbaren Umgebung eines Bildbereichs nicht zu einem veränderten Ergebnis führen. [22]

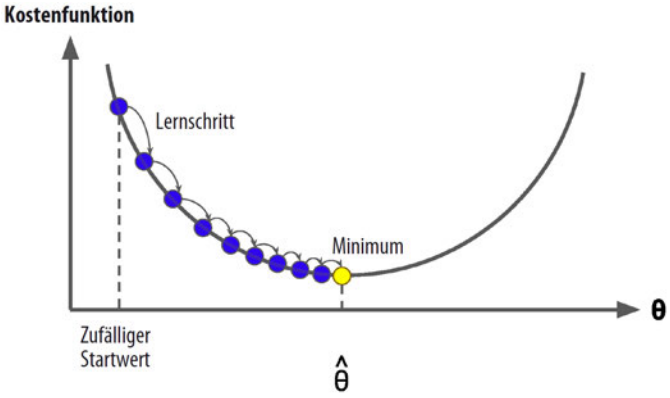
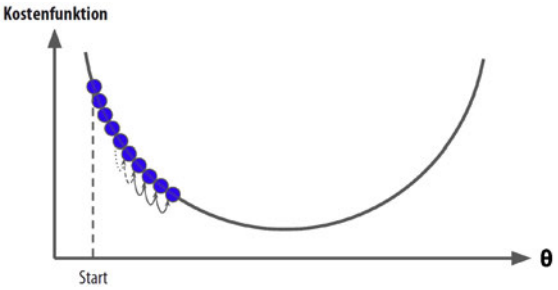


Abbildung 2.5: Gradientenverfahren [8].

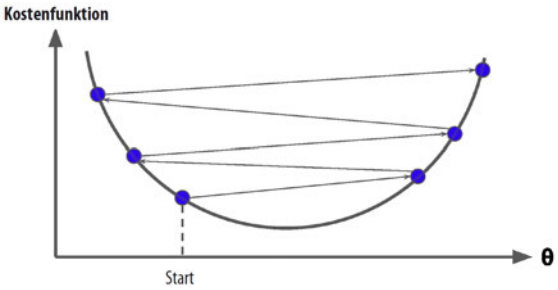
2.3.3 Gradientenverfahren

Eine Möglichkeit, wie neuronale Netze lernen und ihre Ergebnisse optimieren, bietet das Gradientenverfahren oder Verfahren des steilsten Abstiegs. Dieses Verfahren lässt sich am besten an einem Beispiel erklären. Eine Bergsteigerin, die sich gerade an einem Hang befindet, versucht ins Tal zu ihrem Basislager zu kommen. In diesem Tal befinden sich keine Klippen oder Löcher. Der Weg ist also stetig. Erschwerend kommt hinzu, dass es schon so dunkel ist, dass sie nur den Boden unmittelbar um ihre Füße sehen kann. Sie nimmt dafür den Weg des steilsten Abstiegs. Auf kurz oder lang führt sie so der Weg in ihr Lager. Betrachten wir den Querschnitt dieses Tals, lässt sich der Verlauf als Normalparabel beschreiben, bei welchem sich das Basislager im globalen Tiefpunkt befindet. Die Normalparabel wird auch als Kosten- oder Verlustfunktion bezeichnet. Um die Steigung in dem Punkt zu bestimmen, an dem sich die Wanderin befindet, muss der Gradient gebildet werden. Dieser zeigt jedoch in die entgegengesetzte Richtung, in die die Wanderin gehen muss. Das Ziel des Trainings ist es die Kostenfunktion $C(w, b)$ zu minimieren. Im Optimalfall stimmt am Ende des Trainings das tatsächliche Ergebnis mit dem geschätzten Ergebnis überein. Dann sind die besten Gewichte w und der Bias b gefunden worden. Es ist zu berücksichtigen, dass eine Anpassung der Schrittweite erfolgt je tiefer wir ins Tal wandern. Sie wird also kleiner, weil die Steigung abnimmt (siehe Abbildung 2.5). [22, 24]

$$\nabla C = \begin{pmatrix} \frac{\partial C(w,b)}{\partial w} \\ \frac{\partial C(w,b)}{\partial b} \end{pmatrix} \tag{2.11}$$



(a) Gradient mit zu kleiner Lernrate [8].



(b) Gradient mit zu großer Lernrate [8].

Eine Aktualisierung der Parameter w und b kann wie folgt aussehen:

$$b \to b' = b - \alpha \cdot \frac{\partial C(w,b)}{\partial b} = b + \Delta b \tag{2.12}$$

$$w \to w' = w - \alpha \cdot \frac{\partial C(w,b)}{\partial w} = w + \Delta w \tag{2.13}$$

α ist die Lernrate, welche die Schrittgröße beeinflusst. Fällt sie zu groß aus, kann das System oszillieren, d.h. es springt über das Tal. Fällt sie zu klein aus, kann die Schrittweite zu kurz sein und der Algorithmus muss viele Iterationen durchführen, bis er konvergiert. Die Abbildung 2.3.3 (a) veranschaulicht das Gradientenverfahren bei zu kleiner Lernrate und die Abbildung 2.3.3 (b) bei zu großer Lernrate. [8, 24]

Eine weitere Herausforderung gibt es bei Verlustfunktionen mit lokalem Minimum oder Sattelpunkten, z. B. bei einem Plateau. Die Abbildung 2.6 veranschaulicht dies. Hier besteht die Gefahr, dass das globale Minimum nicht erreicht wird. Es gilt zu berücksichtigen, dass bei einer dreidimensionalen Betrachtung der Verlauf ins Tal zickzackförmig und nicht direkt, wie im Diagrammen dargestellt ist (siehe Abbildung 2.5). Perrotta fügt dem Gradienten ein Momentum hinzu. Dieses Momentum sorgt für mehr Dynamik beim Lernen. Dadurch wird der Pfad glatter, den die Bergsteigerin nimmt. Gleichzeitig wird das Training beschleunigt und es ist sogar möglich lokale Minima zu überwinden. [8, 24]

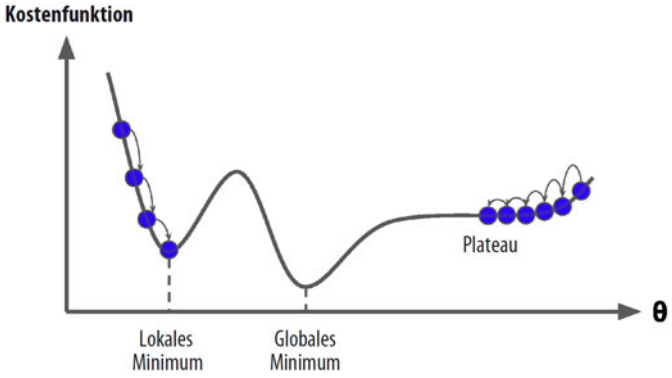


Abbildung 2.6: Fallstricke des Gradienten [8].

Mini-Batch-Gradientenverfahren

Das Hinzufügen des Momentums ist eine Möglichkeit das Training zu beschleunigen. Eine Weitere ist das Mini-Batch-Gradientenverfahren. Die Vorteile dieses Verfahrens sind im Allgemeinen, dass es schneller konvergiert und weniger Speicherplatz benötigt. Es besteht sogar die Möglichkeit, dass es einen geringeren Verlust findet, weil lokale Minima überwunden werden. Im zuvor beschriebenen Gradientenverfahren wurden alle Trainingsdaten als ein Batch gesammelt und für das Training des neuronalen Netzes verwendet. Wie der Name schon vermuten lässt, wird die Menge der Batches verkleinert und kleinere Teilmengen m (Mini-)Batches dem neuronalen Netz hinzugefügt. Anschließend werden zufällige Punkte x berechnet. Dadurch wird der durchschnittliche Gradient geschätzt und nur eine kleine Gradientenmenge ∇C_x berechnet. So kann der Gesamtgradient ∇C angenähert werden. [22, 24]

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{x_j} \tag{2.14}$$

2.3.4 Aktivierungsfunktionen

Anhand der Abbildung 2.3 lässt sich eine Linearität erkennen. Wird dies zu einem Netz von Neuronen aufgespannt, bleibt die Linearität bestehen und die Summe aller Gewichte kann als Matrix $\mathbf{W}$ dargestellt werden. Demzufolge werden auch die Eingabeschicht x

und die Ausgabeschicht $\mathbf{y}$ zu Vektoren. Die folgende Formel veranschaulicht dies:

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} \tag{2.15}$$

Die Aktivierungsfunktion bringt die notwendige Nichtlinearität mit in die Gleichung ein, die der Dreh- und Angelpunkt eines neuronalen Netzes sind. Normalerweise wird in den verborgenen Schichten nur eine Art von Aktivierungsfunktion verwendet, wobei es keinen Wechsel innerhalb dieser Schichten gibt. Für die Ausgabe hingegen sollte die Aktivierungsfunktion je nach Aufgabe angepasst werden und kann sich von derjenigen der verborgenen Schichten unterscheiden. [24]

Sigmoid

Die Sigmoid-Funktion wurde vorher schon einmal in Kapitel 2.3.1 erwähnt. Im Gegensatz zur Stufenfunktion kann sie abgeleitet werden. Demzufolge können Gradienten gebildet und das zuvor beschriebene Gradientenverfahren angewendet werden. Die Sigmoid-Funktion kommt in der Regel nicht in den verborgenen Schichten zum Einsatz, sondern findet hauptsächlich in der Ausgabeschicht Anwendung. Sie ist besonders vorteilhaft für Aufgaben, bei denen Wahrscheinlichkeiten vorhergesagt werden, da ihr Wertebereich zwischen null und eins liegt.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.16}$$

Wird $\sigma(z)$ nach z abgeleitet, ergibt sich

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z)). \tag{2.17}$$

Neben den zuvor genannten Vorteilen ergeben sich aus der Sigmoid-Funktion auch neue Herausforderungen wie zum Beispiel, dass z gegen $-\infty$ oder $+\infty$ strebt. In diesem Fällen treten entweder tote Neuronen oder verschwindende Gradienten auf. Tote Neuronen entstehen, wenn der Gradient gegen null strebt. Dies führt zu einer Verlangsamung oder Stillstand beim Lernen. Bei verschwindenden Gradienten dagegen kann durch die Steigerung der Anzahl der Schichten der Gesamtgradient abnehmen und ebenfalls null werden, weil die Steigerung der Anzahl der Schichten ab einem bestimmten Punkt nichts mehr bringt. Neben dem verschwindenden Gradient, bei dem der Gesamtgradient null wird, kann der Gesamtgradient auch explodieren. Dies bedeutet, dass der Absolutwert des Ge-

samtgradienten zunimmt. Explodierende Gradienten können auch tote Neuronen oder Überläufe verursachen. [24]

Softmax

Ähnlich wie die Sigmoid-Funktion gibt es auch die Softmax-Funktion einen Vektor zurück, dessen Werte im Bereich zwischen null und eins liegen. Zusätzlich ist die Summe aller Ausgaben immer gleich eins. Diese Eigenschaft ist besonders nützlich, da wir die Werte als Wahrscheinlichkeiten interpretieren können, wenn sie sich zu eins addieren. Dies entspricht einer Normierung. Die Softmax-Funktion wird meist als letzte Aktivierungsfunktion in der Ausgabeschicht des neuronalen Netzes eingesetzt, wenn mehr als zwei Klassen zu unterscheiden sind. [24, 25]

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum e^z} \tag{2.18}$$

Weitere Aktivierungsfunktionen neben der Sigmoid- und der Softmax-Funktion sind die Tangens hyperbolicus- [8, 25], die ReLU- [8, 24] und die Leaky ReLU-Funktion [8, 24].

2.3.5 Backpropagation

Eine der größten Herausforderung beim Lernen ist es den Gradienten zu bestimmen. In der Praxis können neuronale Netze, bestehend aus vielen miteinander verknüpften Schichten und Gewichtsmatrizen, äußerst komplex sein. Bei einem so großen Netzwerk wird es schwierig, die Verlustfunktion zu definieren sowie ihre Ableitung zu berechnen. Das Ziel ist es, den Verlustgradienten für beliebige neuronale Netze zu bestimmen. Die Ableitungen sind nur für die einfachsten und weniger leistungsfähigen Netze berechenbar. Hier kommt die Backpropagation ins Spiel. Sie nutzt im Kern die Kettenregel (siehe Formel 2.19), um verschachtelte Funktionen effizient abzuleiten und das Problem zu lösen. [24]

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx} \tag{2.19}$$

Der Algorithmus führt zunächst jeden Trainingsdatenpunkt durch das Netzwerk und berechnet die Ausgabe jedes Neurons in den verschiedenen Schichten. Dies entspricht einem Vorwärtsdurchlauf wie bei der Vorhersage. Danach wird der Fehler der Netzwerk-Ausgabe

gemessen. Dies entspricht der Differenz zwischen der gewünschten und der tatsächlichen Ausgabe. Für jedes Neuron in der letzten verborgenen Schicht wird ermittelt, wie stark es zum Fehler beigetragen hat. Anschließend wird zurückverfolgt, welcher Anteil des Fehlerbeitrags auf jedes Neuron in der vorherigen Schicht entfällt. Dieser Prozess wird fortgesetzt, bis die Eingabeschicht erreicht ist. Im Rückwärtsdurchlauf wird der Fehlergradient für alle Gewichte im Netzwerk berechnet (daher der Begriff Backpropagation). Der letzte Schritt des Backpropagation-Algorithmus besteht darin mittels Gradientenverfahren die Gewichte im Netzwerk basierend auf dem zuvor berechneten Gradienten zu aktualisieren. Kurz gesagt: bei jedem Trainingsdatenpunkt macht der Backpropagation-Algorithmus zunächst eine Vorhersage (Vorwärtsdurchlauf), berechnet den Fehler, ermittelt dann rückwärts den Fehlerbeitrag jeder Verbindung (Rückwärtsdurchlauf) und passt schließlich die Gewichte an, um den Fehler zu verringern. Der letztes Schritt findet im Gradientenverfahren statt. [8, 25, 22]

2.3.6 Gewichtsinitialisierung

Die Initialisierung der Gewichte hat einen ein Einfluss auf die Lerndynamik des neuronalen Netzes. Je nachdem wie es initialisiert wird, können zum Beispiel verschwindende oder explodierende Gradienten verstärkt, minimiert oder wenn nicht sogar verhindert werden. Werden Sättigungen und somit tote Neuronen vermieden, bleibt das neuronale Netz leistungsfähig. Hierzu werden im Folgenden mögliche Gewichtsinitialisierungen und ihre Auswirkungen auf die Lerndynamik und demzufolge auf das neuronale Netz betrachtet. [24]

Symmetrische Initialisierung

Perrotta erklärt in seinem Buch [24], dass es niemals gut ist alle Gewichte mit dem gleichen Wert, d.h. weder mit null, eins oder einer anderen Konstante, zu initialisieren. Bei identischen werten lernen alle Gewichte eines Neurons in der Backpropagation gleich und werden immer identische Werte behalten. Die Verlustfunktion beginnt beim Training eines neuronalen Netzes entweder gegen null zu sinken oder zu divergieren. Das Netzwerk liefert infolgedessen keine sinnvollen Ergebnisse mehr. Diese Symmetrie kann aufgebrochen werden, indem es mit zufälligen Werten initialisiert wird. Die zufällige Initialisierung kann zum Beispiel nach der Normalverteilung passieren. Es gilt zu berücksichtigen, dass die Gewichte ebenfalls nicht mit großen Werten initialisiert werden dürfen, damit die

Neuronen nicht in die Sättigung kommen. Ob dies große negative oder positive Werte umfasst, ist für die Sättigung unerheblich. Zusammenfassend lässt sich sagen, dass die Werte klein und zufällig sein müssen, um das Training zu beschleunigen und tote Neuronen zu vermeiden. [24]

Xavier-Initialisierung

Eine weitere Möglichkeit die Gewichte zu initialisieren, beschreiben Xavier Glorot und Yoshua Bengio in ihrer Arbeit [6]. Ziel der Arbeit war es das Problem der verschwindenden und explodierenden Gradienten in tiefen neuronalen Netzen zu minimieren, was die Stabilität und Effizienz des Trainingsprozesses verbessert. Bei dieser Technik wird die Varianz σ^2 der Gradienten gleichmäßig auf die verschiedenen Schichten verteilt, sodass keine Schicht übermäßig gewichtet und andere Schichten vernachlässigt werden. Die Gewichtswerte werden unter Berücksichtigung der Anzahl n der Eingangs- und Ausgangsneuronen einer Schicht initialisiert. Es kann zwischen einer Normal- oder einer Gleichverteilung gewählt werden. Bei der Normalverteilung ergibt sich mit einem Mittelwert $\mu = 0$ eine Standardabweichung σ

$$\sigma = \sqrt{\frac{2}{n_{Eingang} + n_{Ausgang}}} \tag{2.20}$$

oder für die Gleichverteilung im Intervall $[-r, r]$ für r

$$r = \sqrt{\frac{6}{n_{Eingang} + n_{Ausgang}}}. \tag{2.21}$$

Mit ihrer Arbeit konnten Glorot und Bengio zeigen, dass die Initialisierung der Gewichte für das Training eine nicht unwesentliche Bedeutung hat und die Lerndynamik abhängig von der Aktivierungsfunktion, in diesem Fall die Sigmoid-Funktion oder die tanh-Funktion, verbessern kann. [8, 6]

He-Initialisierung

Die Arbeit von Glorot und Bengio wurde 2015 von einer anderen Forschergruppe mit der He-Initialisierung weiterentwickelt. Sie wurde speziell für die ReLU-Aktivierungsfunktion und deren Varianten mit dem gleichen Ziel wie bei der Xavier-Initialisierung entwickelt.

Da sie eine Weiterentwicklung ist, wird über den Mittelwert $\mu = 0$ die Standardabweichung σ in der Normalverteilung über die Formel

$$\sigma = \sqrt{2} \sqrt{\frac{2}{n_{Eingang} + n_{Ausgang}}} \tag{2.22}$$

und für die Gleichverteilung im Intervall $[-r, r]$ für r mit

$$r = \sqrt{2} \sqrt{\frac{6}{n_{Eingang} + n_{Ausgang}}} \tag{2.23}$$

angegeben. [8, 11]

2.3.7 Verlustfunktion

Die Verlustfunktion ist auch als Kosten-, Fehler- oder Straffunktion bekannt. Die richtige Wahl hängt von der Aufgabenstellung ab. Die Verwendung des mittleren quadratischen Fehlers (MSE) für Regressionsaufgaben und die Kreuzentropie-Kostenfunktionen eignet sich für Klassifikationsaufgaben besser [25]. Sie bildet ein Qualitätsmaß dafür, wie stark der Fehler, also die Differenz zwischen vorhergesagtem und erwartetem Wert, bei der Anpassung der Gewichte berücksichtigt wird. [8]

Der **RMSE (Root Mean Square Error)** wird auch für Regressionsaufgaben verwendet und entspricht der Größe des Fehlers, den das System im Mittel bei Vorhersagen macht. Zu beachten ist, dass großen Fehlern ein höheres Gewicht gegenüber kleinen Fehlern beigemessen wird. Sie kann auch als mittlere quadratische Abweichung (MSE, Mean Sqaure Error) ohne Wurzel verwendet werden [25]. Der RMSE ist definiert als

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2} \,, \tag{2.24}$$

wobei m die Anzahl der Datenpunkte im Datensatz, y der tatsächliche und $\hat{y}$ der vorhergesagte Wert ist. [8]

Der **MAE (Mean Absolute Error)** bietet sich an, wenn es viele Ausreißer gibt, weil es die durchschnittliche absolute Differenz zwischen zwei Punkten misst. Diese werden

genauso stark gewichtet wie die anderen Fehler. [8]

$$MAE = \frac{1}{m} \sum_{i=1}^{m} |y_i - \hat{y}_i| \tag{2.25}$$

Die **Kreuzentropie** ist eine weit verbreitete Verlustfunktion, die in Klassifikationsaufgaben zum Einsatz kommt. Ein höherer Wert der Kreuzentropie bedeutet, dass die Differenz zwischen der vorhergesagten und der tatsächlichen Klasse größer ist. Bei Vorhersagen für mehrere Klassen spricht man von kategorischer Kreuzentropie während bei nur zwei Klassen die binäre Kreuzentropie verwendet wird. [25, 8]

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_{ik} \cdot log(\hat{y}_{ik}) \tag{2.26}$$

2.3.8 Einstellung Hyperparameter

Die Flexibilität neuronaler Netze ist gleichzeitig auch einer ihrer größten Nachteile. Es gibt viele Hyperparameter, die angepasst werden können. Zusätzlich können verschiedene Netzwerktopologien, d.h. unterschiedliche Arten wie die Neuronen miteinander verbunden sind, verwendet werden. Bereits bei einem einfachen neuronalen Netzwerk können zahlreiche Parameter wie etwa die Anzahl der Trainingsepochen, die Anzahl der Schichten, die Anzahl der Neuronen in jeder Schicht, die Wahl der Aktivierungsfunktion, die Methode zur Initialisierung der Gewichte, die Lernrate und die Batchgröße eingestellt werden [8]. Es ist sinnvoll immer nur einen Parameter auf einmal anzupassen und sich die Änderungen anzuschauen [24]. Im Folgenden sollen einige Hinweise und Ratschläge für das Finden geeigneter Hyperparameter gegeben werden.

Anzahl der Epochen

Die Anzahl der Epochen ist der Hyperparameter, der am einfachsten angepasst werden kann. Es ist bekannt, dass das System mit zunehmendem Training immer genauer wird, je länger es trainiert wird. Dies passiert jedoch nur bis zu einem gewissen Punkt. Nach einer bestimmten Anzahl an Epochen erreicht die Genauigkeit ihren Höchstwert, sodass weiteres Training nur noch ineffizient wäre. Ein Nachteil von zu viel Training kann die

Überanpassung sein, welche die Genauigkeit sogar verschlechtern könnte. Auf das Thema Überanpassung wird in Kapitel 2.3.9 näher eingegangen. Perrotta [24] empfiehlt, mit einer großen Anzahl an Epochen zu starten und zu beobachten, nach welcher Anzahl absolvierter Epochen die Genauigkeit nicht mehr steigt. [24]

Anzahl der verdeckten Schichten

Mit ausreichend Neuronen kann ein flaches neuronales Netz mit nur einer verborgenen Schicht auch komplexeste Funktionen modellieren [24]. Tiefere Netze besitzen jedoch eine höhere Parametereffizienz, wodurch komplexe Funktionen mit exponentiell weniger Neuronen als flache Netze modelliert werden können. Hieraus resultiert auch eine kürzere Trainingszeit. Dies liegt an der hierarchischen Architektur. Durch das schnellere Training konvergiert das neuronale Netz schneller gegen annehmbare Lösungen. Für den Beginn wird meist empfohlen mit einer oder zwei verborgenen Schichten zu starten und diese je nach Komplexität der Aufgabe schrittweise zu erhöhen bis die Trainingsdaten überangepasst sind. In der Praxis ist es gängig Teile eines sehr gut vortrainierten neuronalen Netzes für bekannte Probleme wiederzuverwenden. [8]

Zum Schluss soll noch erwähnt werden, dass Géron [8] explizit beschreibt die Trainingsdaten in diesem Fall überanzupassen. Perrotta [24] weist dagegen darauf hin, dass dies kontraproduktiv für den Testdatensatz sein kann und das Netz dort schlechter performt, weil es überangepasst ist [24].

Anzahl der verdeckten Neuronen pro Schicht

Hier gilt das gleiche Prinzip wie im Abschnitt zuvor. Am Anfang sollte mit wenigen Neuronen gestartet und diese schrittweise bis zur Überanpassung erhöht werden. Hier kann je nach Problem viel Aufwand notwendig sein, um die richtige Anzahl an Neuronen zu finden. Es gilt dabei zu beachten, dass zu viele Knoten das Training verlangsamen, was das Netz zu intelligent macht und dadurch zu einer Überanpassung führen kann [24]. Géron beschreibt in seinem Buch [8], dass es ein einfacher Ansatz der *Stretch Pants* Ansatz ist. Hier wird ein neuronales Netz mit mehr Schichten und Neuronen ausgewählt, als tatsächlich benötigt werden. Durch ein frühes Beenden wird versucht (*Early Stopping*) die Gefahr einer Überanpassung zu verhindern. [8]

Lernrate

Die Lernrate ist ein entscheidender Hyperparameter im Gradientenabstiegsverfahren. In jedem Schritt des Verfahrens wird der Gradient mit der Lernrate multipliziert, um die Gewichte anzupassen. Je höher die Lernrate gewählt wird, desto größere Anpassungen werden erreicht. Daraus resultiert, dass bei einer zu kleinen Lernrate das Training verlangsamt wird. Bei einer zu großen Lernrate besteht dagegen die Möglichkeit sich vom globalen Minimum zu entfernen. Laut Perrotta [24] kann mathematisch gezeigt werden, dass das Batch-Gradientenverfahren bei einer glatten Verlustfunktion immer das Minimum erreicht, solange die Lernrate ausreichend klein ist. Bei einer zu großen Lernrate ist dies jedoch nicht garantiert. Perrotta veranschaulicht die Einstellung an einem exponentiellem Verfahren, um schneller eine passende Größenordnung zu finden und diese anschließend nachzujustieren. [8, 24]

Batchgröße

Die Batchgröße legt fest, wie viele Trainingsdaten verarbeitet werden, bevor die Modellparameter angepasst werden. Sie hat Einfluss auf die Trainingsgeschwindigkeit sowie die Genauigkeit des Modells. Eine größere Batchsize beschleunigt das Training des neuronalen Netzwerks, erfordert jedoch mehr Speicherkapazität. Eine kleinere Batchsize führt zwar zu einer längeren Trainingsdauer, ermöglicht dem Netzwerk jedoch eine bessere Generalisierung der Daten, was die Leistung des Modells verbessern kann. [24]

2.3.9 Herausforderung des Trainierens von neuronalen Netzen

Bisher wurden einige Stellschrauben zum Trainieren von neuronalen Netzen vorgestellt. In diesem Abschnitt soll darauf eingegangen werden, wie mögliche Herausforderungen bewältigt werden können und wie diese aussehen. Eine Herausforderung kann zum Beispiel sein, dass die Verlustfunktion nicht konvergiert oder die Leistung des neuronalen Netzes abnimmt. Unvorteilhaft ist es auch, wenn die Verlustfunktion sich zu gut dem Trainingsdatensatz anpasst. Dies kann mit einem Schüler verglichen werden, der die Antworten für einen Test auswendig lernt, aber das Wissen nicht auf neue Aufgaben vom gleichen Typus transferiert. In einem solchen Fall ist das neuronale Netz nicht gut generalisiert und fokussiert sich auf die falschen Dinge. [24]

Verschwindende und explodierende Gradienten

Tote Neuronen wurden vorher schon einmal erwähnt. Dies passiert, wenn die Sigmoid-Funktion in Sättigung geht. Die Folge ist, dass das Training immer langsamer wird. Beim verschwindenden Gradienten werden zum Beispiel während der Backpropagation die Teilgradienten miteinander multipliziert. Sind die Teilgradienten sehr klein, ist der Gesamtgradient am Ende winzig und hat kaum bis keinen Einfluss auf die Gewichtsänderung der ersten Schichten. Das neuronale Netz konvergiert nicht. Bei explodierenden Gradienten verhält sich dies gegensätzlich. Der Gradient wächst während der Backpropagation sehr schnell, was den Namen *explodierender Gradient* begründet und das neuronale Netz divergieren lässt. Beides kann vermieden werden, indem anstatt der Sigmoid-Funktion andere Aktivierungsfunktionen eingesetzt werden, die nicht sättigen. [8, 24]

Über- und Unteranpassung

Eine Unteranpassung entsteht, wenn das ML-System nicht leistungsfähig genug ist. Dies führt zu einer unzureichenden Aussagegenauigkeit. Mögliche Ursachen können die Architektur oder die Einstellung der Hyperparameter sein. Bei der Überanpassung hingegen trifft das System genauere Vorhersagen bei neuen, ihm unbekannten Daten. Dies ist mit dem zu vorgenannten Beispiel des Schülers zu vergleichen, der die Erkenntnisse nicht auf neue Situationen transferieren kann. Im Falle eines ML-Systems kann die Generalisierung der Trainingsdaten nicht ausreichend sein, um sie auf neue Situationen anwenden zu können.[24]

Vermeiden von verschwindenden und explodierenden Gradienten

Die folgenden Möglichkeiten helfen dabei verschwindende / explodierende Gradienten zu reduzieren:

- Anpassung Gewichtsinitialisierung,
- nicht sättigende Aktivierungsfunktionen,
- Batch-Normalisierung, oder
- Gradient-Clipping.

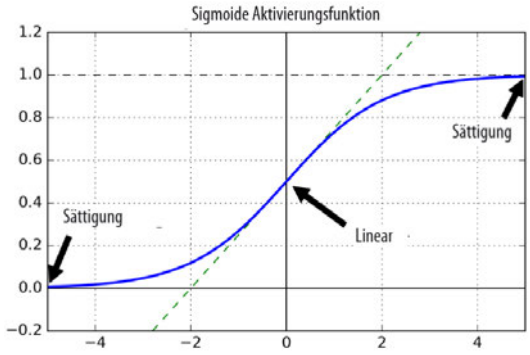


Abbildung 2.7: Sättigung der Sigmoid-Aktivierungsfunktion [8].

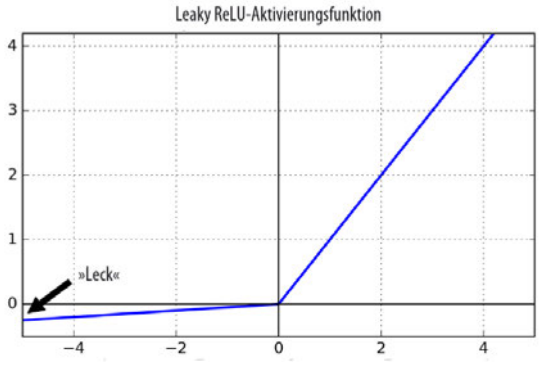


Abbildung 2.8: Leaky ReLU-Aktivierungsfunktion [8].

Auf die Anpassung der Gewichtsinitialisierung wurde bereits im Kapitel 2.3.6 eingegangen. Wichtig ist, dass durch die Anpassung der Initialisierung am Anfang des Trainings, das Auftreten von toten Neuronen verhindert werden kann. Diese kann im späteren Trainingsverlauf immer noch auftreten. [8]

Nicht sättigende Aktivierungsfunktionen: Neben der Sigmoid-Funktion wurden im Kapitel 2.3.4 noch weitere Aktivierungsfunktionen erwähnt. Eine Andere ist die Tangens hyperbolicus-Funktion. Diese beiden können für sehr große positive und negative Werte in Sättigung geraten, wie die folgende Abbildung 2.7 veranschaulicht. Andere Aktivierungsfunktionen hingegen wie zum Beispiel die ELU- oder Leaky RelU-Aktivierungsfunktion wurden speziell dafür entwickelt, dass das nicht passiert. Der Verlauf der Leaky ReLU-Funktion ist in der folgenden Abbildung 2.8 dargestellt. Aus der Abbildung 2.7 wird ersichtlich, dass für unendlich große positive als auch für negative Werte keine Sättigung eintritt. Im Jahr 2015 wurde mit der Exponential Linear Unit (ELU) [3] eine weitere

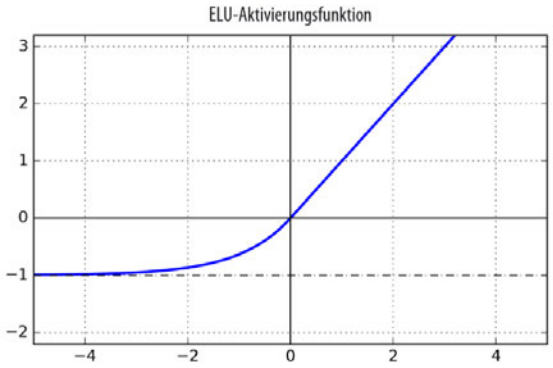


Abbildung 2.9: ELU-AKtivierungsfunktion [8].

Aktivierungsfunktion vorgestellt. In der Abbildung 2.9 ist ersichtlich, dass sie der ReLU-Funktion sehr ähnelt. Die Funktion wird durch die folgende Formel beschrieben:

$$ELU_\alpha(z) = \begin{cases} \alpha(exp(z) - 1) & \text{, wenn } z < 0 \\ z & \text{, wenn } z \geq 0 \end{cases} \tag{2.27}$$

Die ELU-Funktion weist gegenüber der ReLU-Funktion einige große Unterschiede auf. Zum Einen ist es möglich, dass sie für $z < 0$ negative Werte annehmen kann. α ist ein Hyperparameter, der eingestellt werden kann. Ihm nähert sich die ELU-Funktion an, wenn die Werte von z stark negativ werden. Dadurch kann das Neuron eine durchschnittliche Ausgabe um null haben. Zum Anderen ist die Ableitung für negative Argumente $z < 0$ ungleich null, was das Problem sterbender Neuronen umgeht. Ein weiterer Unterschied ist, dass die Funktion glatt ist. Dadurch springt die Funktion weniger links und rechts von $z = 0$ umher, was das Gradientenverfahren beschleunigt. [8]

Ein Nachteil der ELU- gegenüber der ReLU-Funktion ist, dass sie sich langsamer berechnen lässt. Im Allgemeinen empfiehlt Géron in seinem Buch [8] die Aktivierungsfunktionen aufsteigend von der Sigmoid < tanh < ReLU < Leaky ReLU < ELU zu verwenden. [8]

Die **Batch-Normalisierung** ist eine weitere Möglichkeit um verschwindende Gradienten zu vermeiden. Die Xavier- und He-Initialisierung zusammen mit der ELU- oder einer anderen Aktivierungsfunktion können zwar verschwindende oder explodierende Gradienten zu Beginn des Trainings sehr reduzieren, sind aber keine Garantie dafür, dass diese nicht im späteren Trainingsverlauf wieder zurückkehren können. Die Methode besteht darin in jeder Schicht des Modells eine Operation unmittelbar vor der Aktivierungsfunk-

tion hinzuzufügen. Diese Operation zentriert die Eingaben auf null und normalisiert sie. Danach werden die normalisierten Werte mithilfe von zwei neuen Parametern pro Schicht skaliert bzw. verschoben. Dadurch kann das Modell die ideale Skalierung und den Mittelwert für die Eingaben jeder Schicht selbst erlernen. Um die Eingaben zu zentrieren und zu normalisieren, schätzt der Algorithmus den Mittelwert und die Standardabweichung der Eingaben. Diese Werte werden aus dem aktuellen Mini-Batch berechnet. Der Algorithmus ist in der folgenden Formel zusammengefasst: [8]

$$1.\ \mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)} \tag{2.28}$$

$$2.\ \sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}(\mathbf{x}^{(i)} - \mu_B)^2 \tag{2.29}$$

$$3.\ \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.30}$$

$$4.\ \mathbf{z}^{(i)} = \gamma\hat{\mathbf{x}}^{(i)} + \beta \tag{2.31}$$

mit

- μ_B ist der empirische Mittelwert für den gesamten Mini-Batch B.
- σ_B ist die empirische Standardabweichung, ebenfalls für den gesamten Mini-Batch bestimmt.
- m_B ist die Anzahl Datenpunkte im Mini-Batch.
- $\hat{\mathbf{x}}^{(i)}$ ist die auf null zentrierte und normalisierte Eingabe.
- γ ist der Parameter zum Skalieren der Schicht.
- β ist der Parameter zum Verschieben der Schicht (Offset).
- ϵ ist eine kleine Zahl, zum Vermeiden einer Division durch null (normalerweise 10^{-5}). Dies wird als *Smoothing-Term* bezeichnet.
- $\mathbf{z}^{(i)}$ ist die Ausgabe der BN-Operation: Sie ist eine skalierte und verschobene Version der Eingaben.

Das **Gradient Clipping** ist eine gängige Methode, um das Problem der explodierenden Gradienten zu mildern. Es besteht darin, die Gradienten während der Backpropagation zu

begrenzen, sodass sie niemals einen bestimmten Schwellenwert überschreiten. Heutzutage bevorzugen jedoch viele die Anwendung der Batch-Normalisierung. [8, 41]

Vermeiden von Unter- und Überanpassung

Wenn das ML-System nicht leistungsfähig genug und unterangepasst ist, kann dies ausschließlich durch mehr Leistung behoben werden. Das ML-System ist dann am besten in die Überanpassung (Overfitting) zu bringen und die folgenden Methoden anzuwenden, um dieses zu reduzieren. [24]

Early-Stopping: Um eine Überanpassung an die Trainingsdaten zu verhindern, ist Early Stopping eine effektive Methode. Sobald die Leistung des Modells auf den Validierungsdaten zu sinken beginnt, wird das Training gestoppt. In TensorFlow kann dies durch regelmäßige Evaluierung des Modells auf einem Validierungsdatensatz umgesetzt werden (z. B. alle 50 Schritte). Ein Gewinnermodell wird gespeichert, wenn es das bisher beste Modell übertrifft. Es wird verfolgt wie viele Schritte seit dem letzten Speichern des Gewinnermodells vergangen sind und das Training beendet, wenn diese Zahl einen festgelegten Schwellenwert überschritten hat. Anschließend wird das gespeicherte Gewinnermodell wieder hergestellt. [8]

l1-und l2-Regularisierung: Ähnlich wie bei einfachen linearen Modellen können auch bei neuronalen Netzen mithilfe von l1- und l2-Regularisierung Einschränkungen auf die Verbindungsgewichte (jedoch nicht auf die Bias-Terme) anwenden [8]. Beide Methoden integrieren die Gewichte in die Verlustfunktion des neuronalen Netzes. Das Gradientenverfahren versucht dann, den Verlust zu minimieren, indem es die Gewichte kleinhält. Kleinere Gewichte tragen zu einem glatteren Modell bei. [24]

Weitere Regularisierungstechniken sind Drop-Out, Max-Norm-Regularisierung und Data Augmentation [8].

Optimierer

Im Kapitel 2.3.3 wurde das Gradientenverfahren sowie das Mini-Batch-Gradientenverfahren vorgestellt. Damit sich die Gewichte und der Bias in die gewünschte Richtung verschieben und die Verlustfunktion abnimmt, muss eine entsprechende Optimierungsfunktion

definiert werden. Diese wird als Optimierer bezeichnet und steuert die Anpassung der Gewichte und des Bias. Durch die Wahl eines guten Optimierers wird das Training beschleunigt. In dem Paper [29] von Sebastian Ruder von 2017 gibt dieser einen Überblick darüber mit welchen Optimierern das Gradientenverfahren verbessert werden kann. [8]

Der **stochastische Gradienten Abstieg (SGD)** beruht auf der von Robbins und Monro vorgestellten stochastischen Approximationsmethode [27, 29]. Das Hauptproblem des Batch-Gradientenverfahrens liegt darin, dass es für jeden Berechnungsschritt der Gradienten den gesamten Trainingsdatensatz benötigt. Das macht es bei sehr großen Datensätzen extrem langsam. Auf der anderen Seite steht das SGD, das bei jedem Schritt nur einen zufällig ausgewählten Datenpunkt verwendet, um die Gradienten zu berechnen. Dies beschleunigt den Algorithmus erheblich, da in jeder Iteration nur ein minimaler Teil des Datensatzes verarbeitet werden muss. Daher ist es besonders geeignet, um mit sehr großen Datensätzen zu arbeiten, da nur ein einzelner Datenpunkt pro Iteration geändert wird. Allerdings führt diese stochastische Herangehensweise dazu, dass der Algorithmus viel unregelmäßiger als das Batch-Gradientenverfahren ist. Anstatt gleichmäßig zum Minimum zu konvergieren, „hüpft“ die Kostenfunktion auf und ab und sinkt nur im Durchschnitt. Im Laufe der Zeit erreicht sie zwar ein Minimum, bleibt dort jedoch nie stabil und schwankt weiter. Am Ende des Trainings liefert der Algorithmus somit gute, aber nicht optimale Parameter. Bei einer stark unregelmäßigen Kostenfunktion kann diese Schwankung jedoch dabei helfen, aus lokalen Minima herauszukommen. Aus diesem Grund hat das stochastische Gradientenverfahren im Vergleich zum Batch-Gradientenverfahren eine größere Chance, das globale Minimum zu finden. Die Zufälligkeit hilft, in lokale Minima zu entkommen, verhindert jedoch, dass der Algorithmus im globalen Minimum zur Ruhe kommt. Eine mögliche Lösung für dieses Problem besteht darin, die Lernrate nach und nach zu verringern. Zu Beginn sind die Schritte groß, um schnell voranzukommen. In der Folge werden die Schritte immer kleiner, sodass der Algorithmus letztlich im globalen Minimum stabil wird. [8]

Das **AdaGrad (Adaptive Gradient Algorithm)** passt die Lernrate an die einzelnen Modellparameter an, indem es die bisherigen Gradientenwerte berücksichtigt. Es summiert die quadrierten Gradienten für jeden Parameter während des Trainings auf und verfolgt so, wie stark sich der Gradient in der Vergangenheit verändert hat. Auf dieser Grundlage wird die Lernrate für jeden Parameter angepasst, indem die ursprünglich festgelegte Lernrate durch die berechnete Summe der quadrierten Gradienten geteilt wird. Die Lernrate wird antiproportional zu der Größe des Gradienten gewählt. Dadurch

wird eine individuelle Skalierung für die Parameter erreicht. Parameter mit seltenen oder großen Gradienten bekommen eine niedrigere Lernrate, während Parameter mit häufigen oder kleinen Gradienten eine höhere Lernrate erhalten. Aufgrund dieser Eigenschaft eignet sich AdaGrad besonders gut für spärliche Datensätze und erspart das manuelle Anpassen der Lernrate. Ein wesentlicher Nachteil von AdaGrad ist jedoch, dass die Lernrate mit der Zeit immer weiter sinkt. Wird sie so klein, dass keine weiteren Aktualisierungen mehr stattfinden, kommt der Lernprozess schließlich zum Stillstand. [4, 29]

Die **RMSProp (Root Means Square Propagation)** ist eine adaptive Methode, welche die Lernrate während des Trainings anpasst. Es verwendet eine kumulierte Summe der quadrierten Gradienten, die mit einem gleitenden Durchschnitt gewichtet wird. Auf diese Weise werden jedoch nur die früheren Gradienten bei der Berechnung der Aktualisierung berücksichtigt. Eine weitere Methode, die zur selben Zeit wie RMSProp entwickelt wurde und die auch die vergangenen Aktualisierungsschritte berücksichtigt, ist AdaDelta [40]. Dadurch wird die Anpassung genauer, benötigt jedoch mehr Rechenaufwand. [12, 29]

Die **Adam (Adaptive Moment Estimation)** vereint zwei Prinzipien - den RMSprop-Algorithmus mit einer dynamischen Lernrate und den stochastischen Gradientenabstieg (SGD) mit Momentum. Dabei werden zwei verschiedene Momente verwendet. Das erste Momentum hilft dabei, die Richtung des nächsten Schrittes im Parameterraum festzulegen. Das zweite Momentum passt hingegen die Schrittgröße. Adam hat sich als besonders effektiv, insbesondere bei komplexen Netzwerkstrukturen oder großen Datensätzen, erwiesen. Die dynamische Anpassung der Lernrate sorgt für eine schnelle und stabile Konvergenz der Verlustfunktion zum Minimum, was das Training beschleunigt und stabiler macht. Zudem trägt die Momentschätzung dazu bei, Probleme mit verschwindenden oder explodierenden Gradienten zu reduzieren, die beim Training von neuronalen Netzen häufig auftreten. [15, 8, 29]

Weitere Optimierer sind beschleunigter Gradient nach Nesterov und Momentum Opimization [8].

2.4 Algorithmen

Nachdem die grundlegenden RL- und DL-Formalismen beschrieben wurden, wird im Folgenden auf verschiedene Methoden eingegangen, die für eine Lösung in Frage kommen.

2.4.1 Tabellenbasierte Lösungsverfahren

Die Aufteilung von RL-Methoden kann je nach Quelle leicht unterschiedlich sein. Sutton und Barto [34] unterscheiden in erster Instanz zwischen tabellarischen und approximierten Lösungsverfahren. Gridin [7] unterscheidet hingegen zuerst zwischen dem Bandits-Problem und dem Markov Entscheidungsprozess. Das Bandits-Problem ist eine Sonderform des MDP, bei welchem es nur einen einzigen Zustand gibt. Diese Aufteilung kann auf den ersten Blick verwirrend sein. Trotz dessen die Anordnungen voneinander abweichen, werden sie dennoch in die gleichen Kategorien aufgeteilt. So unterscheiden beide im Wesentlichen zwischen modellfreien, modellbasierten, strategiebasierte (nach [34] approximierte Lösungsverfahren) und wertebasierte Methoden (nach [34] tabellarische Lösungsverfahren) sowie Off-Policy und On-Policy Ansätze. Nach Gridin [7] lernen modellbasierte Verfahren indirekt das optimale Verhalten durch das Erlernen eines Modells der Umgebung. Die in der Umgebung ausgeführten Handlungen und dessen Ergebnisse, also der neue Zustand sowie die sofortige Belohnung, werden beobachtet. Bei den modellfreien Verfahren ist ein Modell nicht notwendig. Hier wird jedoch zuerst zwischen den strategiebasierten (Policy-Based) und den wertebasierten (Value-Based) Methoden unterschieden. Letzteres wird zusätzlich zwischen Off-Policy und On-Policy differenziert. [34, 7]

Monte-Carlo

Die Monte-Carlo-Methode ist eine breite Klasse von Algorithmen, die auf der wiederholten Durchführung von Zufallsstichproben basiert. Ihr Hauptmerkmal ist, dass es nach einer ausreichend großen Anzahl an Zufallsexperimenten möglich wird, die Eigenschaften und Merkmale eines bestimmten Prozesses oder einer Umgebung zu ermitteln. Diese Methode findet Anwendung in vielen Berechnungsproblemen, insbesondere dann, wenn es schwierig ist, analytische Ergebnisse über die betreffende Umgebung zu erzielen. Die Ergebnisse werden in der Q-Tabelle gesammelt. Diese Tabelle beinhaltet zu jedem Zustand die entsprechende Aktion. Das ist die zuvor beschrieben Aktionswertfunktion q und gehört zu den wertebasierten Lösungsverfahren. Ein Nachteil der Methode ist, dass entweder nur trainiert oder nur bewertet werden kann. Ein Update der Q-Tabelle ist erst am Ende des Trainings möglich. Der Q-Wert eines Zustandsaktionspaares ergibt sich aus der Summe aller Gewinne in der Episode n, der durch die Gesamtanzahl des Zustandsaktionspaares im jeweiligen Trainingsverlauf geteilt wird. Dies wird in der folgende Formel

beschrieben:

$$Q(s,a) = \frac{\sum_n G(s,a;n)}{N(s,a)} \tag{2.32}$$

Die Summer aller Gewinne beinhaltet auch die zukünftigen diskontierten Belohnungen, wie durch die Formel

$$G(s_k,a_k;n) = \sum_{i=k}^{t} \gamma^{i-k}\, r_k \tag{2.33}$$

ausgedrückt wird. [7]

Q-Learning

Wie bei der Monte-Carlo Methode wird beim Q-Learning eine Tabelle mit den Q-Werten befüllt. Jedoch im Fall des Q-Learnings nähert sich die gelernte Aktionswertefunktion q direkt der optimalen Aktionswertefunktion q_* an, unabhängig von der verfolgten Policy. Anders ausgedrückt wird nach dem Ausführen einer Aktion und dem Erreichen des nächsten Zustands die Werte in der Q-Tabelle aktualisiert. Dies vereinfacht die Analyse des Algorithmus erheblich und ermöglicht frühzeitige Nachweise von Konvergenzen. Die Policy bleibt weiterhin von Bedeutung, da sie festlegt, welche Zustandsaktionspaare besucht und aktualisiert werden. Für eine korrekte Konvergenz ist jedoch lediglich erforderlich, dass alle Paare weiterhin aktualisiert werden. Je deterministischer die Umgebung ist, in welcher der Q-Learning Algorithmus lernen soll, desto besser. Eine Herausforderung beim Q-Learning ist, dass dieselben Stichproben zur Bestimmung der maximierenden Aktion als auch zur Schätzung ihres Wertes verwendet werden. Dies kann zu einer erheblichen Maximierungsverzerrung führen, was mit dem Double Q-Learning unterbunden werden kann. Die Formel 2.34 veranschaulicht den Algorithmus. Das aktuelle Zustandsaktionspaar $Q(S_t, A_t)$ wird mit der Multiplikation der temporalen Differenz (engl. Temporale Difference) und der Lernschrittweite (α) addiert sowie anschließend in der Q-Tabelle aktualisiert. Die Lernschrittweite α und der Diskontierungsfaktor γ liegen beide im Bereich zwischen null und eins. Die Summe innerhalb der Klammer wird auch als Temporale Difference bezeichnet und drückt die erwartete Differenz in verschiedenen Momenten der Untersuchung bzw. des Trainings aus. Sie bildet sich aus der Summe der Belohnung des nächsten Zeitschritts R_{t+1} und der Differenz zwischen dem diskontierten (γ) maximalen Q-Werts $\gamma \max_a Q(s_{t+1}, a)$ des nächsten Zustands, nach der gewählten Aktion, minus des

aktuellen Q-Werts $Q(S_t, A_t)$ des Zustandsaktionspaar. [7, 34, 37]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.34}$$

SARSA

Die SARSA-Methode (State-Action-Reward-State-Action) nimmt hier als On-Policy und Value-Based Methode eine Sonderstellung ein. Der Einsatz von SARSA empfiehlt sich immer dann, wenn die Erfüllung der Aufgabe wichtiger ist, anstatt das optimale Ergebnis zu erzielen. Demzufolge eignet sich SARSA für Probleme in der realen Welt, wo ein Fehler sehr teuer und zum Verlust einer wichtigen Ressource führen kann. Die SARSA-Methode funktioniert analog des Q-Learnings, wie aus der Formel 2.35 ersichtlich wird. Der Unterschied besteht lediglich darin, dass nicht nach der maximal möglichen Belohnungen gesucht wird. [7]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.35}$$

2.4.2 Approximierte Lösungsverfahren

In diesem Abschnitt wird das Beste aus zwei Welten vereint und die approximierten oder geschätzten Lösungsverfahren beschrieben. Die Vereinigung des Reinforcement Learnings mit dem Deep Learning hat im Bereich des Machine Learnings neue Tore geöffnet und leistungsfähigere Algorithmen hervorgebracht.

Deep Q-Learning

Das Deep Q-Learning oder auch Deep Q-Network (DQN) ist bei vielen Problemstellungen ein vielversprechender Ansatz. Er wurde im Jahr 2013 von Google DeepMind vorgestellt [21]. Deep Q-Learning setzt sich aus dem Q-Learning und dem Deep Learning zusammen. Ein möglicher Aufbau wird in der Abbildung 2.10 dargestellt. Bei den wertebasierten Lösungsverfahren ist die Pflege und Aktualisierung einer Q-Tabelle in Umgebungen mit großen Zustandsräumen wenig praktikabel. Dies führt zur Verwendung des Deep Q-Learnings. Statt einer Q-Tabelle nutzt das Deep Q-Learning ein neuronales Netzwerk, das einen Zustand als Eingabe erhält und die Q-Werte für jede mögliche Aktion in diesem Zustand schätzt. Das macht diese Methode attraktiv für den Einsatz

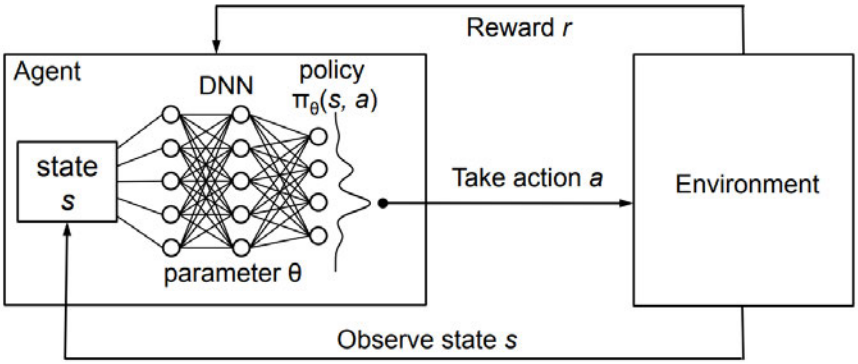


Abbildung 2.10: Reinforcement Learning in Kombination mit einem neuronalen Netzwerk [19].

in großen Zustandsräumen, wie zum Beispiel beim Spiel Yahtzee. Hinzu kommt, dass durch die Verwendung von Erfahrungsspeicher (Replay Buffer) der Agent stabiler und effizienter lernt. [7, 21, 34]

Die Formeln 2.36 - 2.39 zeigen, wie die beiden Methoden mathematisch zusammenhängen. Die Aktionswertfunktion $Q^*(s, a)$ wird um den Parameter θ erweitert und angenommen, dass diese Erweiterung ungefähr gleich der ursprünglichen Aktionswertfunktion $Q^*(s, a) \approx Q(s, a; \theta_i)$ ist. Der Parameter θ beinhaltet die Gewichte $\mathbf{W}$ und Bias $\mathbf{b}$, die bei der Minimierung der Verlustfunktion $L_i(\theta_i)$ in jeder Iteration optimiert werden. Ziel ist es, wie beim Q-Learning die erwartete Belohnung in Formel 2.36 zu maximieren. Dies passiert analog des Q-Learning Algorithmus, indem die Differenz zwischen dem aktuellen und dem nächsten Zustandsaktionspaar gebildet und die Differenz quadriert wird. Dabei werden die Parameter θ_{i-1} der vorherigen Iteration für die Optimierung der Verlustfunktion festgehalten. Zum Schluss wird wie in Formel 2.39 gezeigt, das Gradientenverfahren für die Optimierung des Parameters θ angewendet. [21]

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \tag{2.36}$$

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim p(\cdot)}[(y_i - Q(s, a; \theta_i))^2] \tag{2.37}$$

$$y_i = \mathbb{E}_{s' \sim \epsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a] \tag{2.38}$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot);s'\sim\epsilon}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{2.39}$$

Alle wichtigen Informationen werden nun im neuronalen Netz und nicht mehr in einer Q-Tabelle gespeichert. Dadurch kann die Methode größere Zustandsräume besser generalisieren, weil sie die passenden Aktionen in den jeweiligen Zuständen approximiert. Das spart Speicherplatz.

Policy Gradient-Methode

Die Policy Gradient-Methode ist eine Reinforcement Learning-Methode, bei der eine stochastische Policy direkt optimiert wird. Sie ist mit dem Deep Learning kombinierbar und macht die Methode besonders effektiv, wenn ein neuronales Netz für die Parametrisierung der Policy eingesetzt wird [7]. Das Hauptziel dieser Methode besteht darin, den Parameter zu bestimmen, bei dem der Agent die höchste Belohnung gemäß der Policy π_θ erzielt. Anhand der Formel 2.40 maximiert die Methode die erwartete Gesamtbelohnung $\mathbb{E}$ durch iterative Schätzung des Gradienten von g.

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta log \pi_\theta(a_t|s_t)\right] \tag{2.40}$$

Ψ_t kann dabei eine der folgenden Möglichkeiten annehmen:

1. $\sum_{t=0}^{\infty} r_t$: Gesamtbelohnung der Trajektorie,
2. $\sum_{t'=t}^{\infty} r_{t'}$: Belohnung bei Verfolgung der Aktion a_t,
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: Baseline Version der Formel zuvor,
4. $Q^\pi(s_t, a_t)$ Zustands-Aktionswertfunktion,
5. $A^\pi(s_t, a_t)$ Advantage-Funktion, oder
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$ Temporal Difference Residual. [30]

Die Variante drei mit der Baseline ist eine Möglichkeit, dass die Varianz reduziert wird. Die zusätzliche Verwendung eines neuronalen Netzes macht diesen Algorithmus noch leistungsfähiger. Eine Möglichkeit der Implementierung als REINFORCE oder auch Monte-Carlo Policy Gradient-Methode beschreiben Sutton und Barto in ihrem Buch [34] und

wurde von Williams 1992 in seiner Arbeit [38] vorgestellt. [30, 7]

Klassische Methoden wie Q-Learning versagen bei kontinuierlichen oder stochastischen Aktionsräumen. Die Policy Gradient-Methode ist eine mächtige Technik, die sich gut für solche Reinforcement Learning-Probleme eignet. Trotz der hohen Varianz und der langsamen Konvergenz gibt es viele Weiterentwicklungen wie zum Beispiel die Actor-Critic-Methoden, die diese Methoden stabiler und effizienter machen. [34, 7]

Actor-Critc-Methoden

Die Actor-Critic-Methode vereinigen sowohl die Policy-Based als auch die Value-Based Methoden, im Speziellen die Policy Gradient-Methode und das Q-Learning. Es gibt einen Akteur (Actor), der zum Beispiel nach der Policy-Gradienten Methode eine Aktion auswählt und einen Kritiker (Critic), der diese Aktion anschließend bewertet. Durch die Zusammenarbeit dieser beiden Methoden wird direkt die Policy des Akteurs als auch des Kritikers optimiert und verbessert. Des Weiteren lassen sich beide Methoden mit neuronalen Netzen erweitern. [7]

Die Actor-Critic-Methode und die Policy Gradient-Methode sind in Ψ_t über die Variante 5 durch die Advantage-Funktion $A^\pi(s_t, a_t)$ miteinander verbunden [30]. Diese setzt sich wie folgt zusammen

$$A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{2.41}$$

wobei $V^\pi(s_t)$ die Wertefunktion bzw. der Kritiker und $Q^\pi(s_t, a_t)$ die Aktionswertfunktion bzw. der Akteur ist. Somit ergibt sich für g die folgende Formel für die grundlegende Actor-Critic-Methode:

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \nabla_\theta log \pi_\theta(a_t|s_t)(Q^\pi(s_t, a_t) - V^\pi(s_t))\right] \tag{2.42}$$

Die Vorteile dieser Methode sind, dass die Varianz geringer, der Lernprozess durch die Wertefunktion effizienter ist, die Methode auch in kontinuierlichen Aktionsräumen eingesetzt werden kann und die Methode flexibel ist, was ihre vielen Erweiterungen zeigen. Zusätzliche Erweiterungen sind zum Beispiel A2C [7], A3C [20] und PPO [31]. Nachteile dieser Methode und seiner Varianten sind unter anderem die komplexere Implementierung und dass die Kombination aus zwei Netzwerken zu Instabilitäten führen kann. [7, 34]

3 Stand der Wissenschaft

Nachdem in dem Kapitel zuvor auf die Grundlagen eingegangen wurde, wird in diesem Kapitel der Stand der Technik beschrieben. Dazu wird erörtert welche Untersuchungen es mit dem Spiel Yahtzee als Problemstellung gibt, welche Lösungsansätze für eine Punktmaximierung verfolgt wurden sowie deren Ergebnisse.

3.1 Der Ursprung

Den Startschuss legte 1999-2000 Tom Verhoeff von der Eindhoven University of Technology mit der Arbeit *Optimal Solitaire Yahtzee Strategies* [36]. In dieser Arbeit entwickelte er eine optimale Strategie mit Hilfe der Modellierung von Yahtzee als Markov-Entscheidungsprozess und dem Lösungsansatz der dynamischen Programmierung. Dazu wurde ein Yahtzee Spielbaum aufgestellt und die möglichen Zustände minimiert, die während eines Spiel erreicht werden können. In seiner Arbeit definierte er seine optimalen Kriterien wie folgt:

- Maximierung der erwarteten Gesamtpunktzahl,
- Minimierung der Varianz der Gesamtpunktzahl,
- Maximierung der Wahrscheinlichkeit, die Höchstpunktzahl zu übertreffen,
- Maximierung der Wahrscheinlichkeit gegen einen Gegenspieler zu gewinnen, und
- Maximierung der minimalen Gesamtpunktzahl.

Auch Verhoeff stand vor dem Dilemma, welche Entscheidung bei Würfen getroffen werden soll, die in mehrere Kategorien passen. Ein schönes Beispiel dafür ist die folgende Würfelkombination $Wurf = [1\ 1\ 6\ 6\ 6]$.

Kategorie	Erwartete Punktzahl	Standardabweichung
Aces	1,82	1,14
Twos	5,25	1,95
Threes	8,57	2,65
Fours	12,19	3,24
Fives	15,74	3,81
Sixes	19,29	4,61
Bonus Upper Part	24,14	16,19
Three of a Kind	22,23	5,5
Four of a Kind	13,04	11,44
Full House	22,86	6,99
Small Straight	29,53	3,71
Large Straight	33,04	15,16
Yahtzee	15,89	23,28
Chance	22,26	2,44
Grand Total	**245,87**	**39,82**

Tabelle 3.1: Punktzahl und Standardabweichung pro Kategorie ohne extra Yahtzee Bonus und Joker [36].

Mit diesem Wurf können verschiedene Pfade, abhängig davon wie viele Wiederholungswürfe noch offen stehen, verfolgt werden. Sollen die Würfel mit der Augenzahl sechs behalten werden und die beiden Einsen erneut gewürfelt oder bereits eine Kategorie ausgewählt werden? In seiner Arbeit konnte Verhoeff zeigen, dass es besser war die drei Sechsen zu behalten, um mehr Punkte zu erzielen. Des Weiteren simulierte er seinen Lösungsansatz auch an einem Yahtzee in dem es keinen Bonus oder Joker gab. In der Tabelle 3.1 sind die Ergebnisse von Verhoeff dargestellt, von der Yahtzee-Simulation ohne Bonus und Joker. [36]

2006 wurde die Arbeit von Tom Verhoeff von James Glenn am Loyola College in Maryland, Baltimore aufgegriffen. In seiner Arbeit [5] versucht auch er mittels elementarer Kombinatorik und Graphentheorie eine optimale Strategie zu bestimmen. Diese optimale Strategie verglich er mit anderen Strategien, die zum Beispiel gezielt auf Yahtzees und / oder Straßen gehen, weil diese im Schnitt die meisten Punkte bringen. Es gilt zu berücksichtigen, dass Glenn den Bonus und Joker berücksichtigt und somit die erwartete Gesamtpunktzahl mit $254,59$ Punkten und einer Standardabweichung von $59,61$ Punkten auf den ersten Blick etwas größer ausfällt. Jedoch mit der von Tom Verhoeff unter der gleichen Fallbetrachtung identisch ist. [5, 36]

Eine weitere Arbeit, welche die Entwicklung einer optimalen Strategie für Yahtzee mit der Verwendung von Graphentheorie und dynamischer Programmierung versucht, ist die Arbeit von Marcus Larsson und Andreas Sjöberg von 2012 [17]. Sie erreichten in ihrer Publikation ein durchschnittliches Gesamtergebnis von 248, 63 Punkten. Es gilt zu berücksichtigen, dass sich die Rechenleistung von 2000 zu 2006 und zu 2012 jedes mal weiterentwickelt hat. Der Vollständigkeit halber soll die KTH Royal Institute of Technology in Schweden erwähnt werden, weil auch hier verschiedene Abschlussarbeiten für optimales Yahtzee veröffentlicht wurden, welche die Nutzung verschiedener Algorithmen behandelt haben [13, 23].

Zusammenfassend lässt sich sagen, dass diese Arbeiten einen guten Einblick über mögliche Untersuchungen mit Yahtzee geben. Alle greifen jedoch auf ein Modell zurück, dass die Wahrscheinlichkeiten von einem Zustand in den nächsten benötigt. In der vorliegenden Arbeit wird versucht einen modellfreien Ansatz zu verfolgen.

3.2 Arbeiten mit modellfreien Lösungsansätzen

In den Arbeiten von Minhyung Kang und Luca Schroeder [14] sowie Philip Vasseur [35] wurde ein modellfreier Ansatz verfolgt. In beiden Publikationen wurden verschiedene Algorithmen des ML angewendet. In der Arbeit von Kang und Schroeder [14] wurden speziell Algorithmen des Reinforcement Learnings priorisiert. Philip Vasseur hat dagegen in seiner Arbeit [35] das Deep Q-Learning für den Vergleich verschiedener Strategie-Leitern verwendet, um zu untersuchen, wie die Leistung der KI in Abhängigkeit der Regelsätze variiert.

Kang und Schroeder [14] entwickelten einen Simulator, der es erlaubt, die Spieleranzahl einzustellen und Turniere zwischen den entwickelten Algorithmen auszutragen. Eine Mensch-Maschinen-Interaktion mit dem Simulator wurde nicht implementiert. Demzufolge wurden die Agenten ausschließlich miteinander verglichen. Des Weiteren war es ein Ziel dieser Arbeit herauszufinden, welche RL-Algorithmen wie gut gegen einfachere Algorithmen bestehen können. Für die einfacheren Algorithmen wurden ein Random-Agent (Zufallsagent) und drei Greedy-Agenten (gierige Agenten) entwickelt. Der Random-Agent wählte jede Aktion zufällig aus. Die Greedy-Agent wurden in drei Stufen unterteilt, in Level eins bis drei. Der Greedy-Agent Level-1 nahm den Initialwurf und ordnete den Wurf

der Punktetabelle zu, wo er die meisten Punkte für bekam. Der Greedy-Agent Level-2 hatte einen weiteren Wurf zur Verfügung und der Greedy-Agent Level-3 entsprechend alle drei Wurfversuche. Dadurch besaßen die höherstufigen Greedy-Agenten bessere Möglichkeiten, um eine höhere Endpunktzahl zu erreichen. Die verwendeten RL-Agenten waren unter anderem das Perceptron Q(Λ) und das Hierarchical Learning (HRL). Dem HRL wurde im Verlauf der Arbeit ein Greedy-Element hinzugefügt und somit zum HRL-G1-Agent. Dies sollte dafür sorgen, dass der HRL-G1-Agent mehr Beispiele sieht und so seine Endpunktzahl erhöhen kann. Neben der durchschnittlichen Endpunktzahl wurde die durchschnittliche Zeit pro Zug ermittelt. Hier ergab sich, dass der Greedy-Agent Level-3 mehr als drei Sekunden pro Zug braucht. Dies macht ihn sehr langsam, obwohl er damit die durchschnittlich höchste Endpunktzahl mit $203,88$ Punkten erreicht hatte. Dieser kam dem Greedy-Algorithmus von Glenn von der Gesamtpunktzahl am Nächsten. Der Greedy Level-3 wurde nicht weiter für das Turnier berücksichtigt und demzufolge auch nicht in der Auswertung. Es stellte sich heraus, dass der Perceptron Q(Λ)-Agent die meisten Spiele gegen den Random-Agenten gewann. Die Ergebnisse gegen den Greedy-Agenten Level-1 und Level-2 waren dagegen eher bescheiden. Die beiden anderen RL-Agenten schnitten insgesamt gegen den Greedy-Agenten Level-1 mit einer Gewinnwahrscheinlichkeit von mehr als 50% besser ab. Der Greedy-Agent Level-2 hatte in dieser Arbeit die höchste Gewinnwahrscheinlichkeit. Es gilt zu beachten, dass Untersuchungen wie in Arbeit [14] der Agent oder auch menschliche Spieler zu einem anderen Lernverhalten neigen, wenn das Siegen gegenüber der Maximierung der Gesamtpunktzahl in Vordergrund steht. Zum Beispiel kann der verlierende Spieler zu einem risikoreichen Verhalten und Entscheidungen neigen, solange weiterhin eine Chance auf den Sieg besteht. [14]

Aus den Ergebnissen der Arbeiten ergeben sich interessante Fragestellungen inwiefern Deep Reinforcement Learning unter Verwendung von tiefen neuronalen Netzen die Wahrscheinlichkeiten und strukturellen Eigenschaften von Yahtzee besser erfassen können. Aus dem Abstract von Kan und Schroeder [14] geht hervor, dass einfache Benchmarks übertroffen werden können, aber insgesamt suboptimal sind. Wenn wir uns die Ergebnisse in der Tabelle 3.2 anschauen, wird ersichtlich, dass die Gesamtpunktzahl der optimalen Yahtzee Strategien ausbaufähig ist.

In [35] wurde ein Deep Q-Learning Algorithmus entwickelt und verwendet, der Yahtzee spielen lernt. Ziel dieser Arbeit ist nicht die Untersuchung, ob der Algorithmus eine erfolgreiche Strategie zur Maximierung der Endpunktzahl entwickelt, sondern ob es eine

Agent	**Gesamtpunktzahl**
Random	45,635
Greedy Level-1	112,541
Greedy Level-2	171,166
Greedy Level-3	203,882
Perceptron Q(λ)	77,772
HRL	120,299
HRL-G1	129,580

Tabelle 3.2: Durchschnittliche Gesamtpunktzahl verschiedener Agenten [14].

Korrelation zwischen den Strategieleitern von Mensch und Maschine gibt. Hieraus soll eine Beurteilung erfolgen, ob ein Spiel für einen Menschen gut und herausfordernd ist. Dabei wird untersucht wie eine Änderung der Regelsätze die Leistung des Algorithmus variiert und sich auf das Lernverhalten des Agenten auswirkt. Dies wird unter anderem erreicht, indem die Schwelle für die Mindestpunktzahl des Bonus zwischen 53 bis 75 Punkten variiert. Das Ergebnis von [35] zeigt, dass eine niedrigere Punkteschwelle zwischen 53-57 Punkten zu einer interessanteren Variante von Yahtzee führen könnte.

Zusammenfassend lässt sich sagen, dass Vasseur dem Agenten nur die Kategorie übergibt und eine untergeordnete Hilfsfunktion die Auswahl der Würfel und die Wiederholungswürfe übernimmt. Bei der Arbeit von Kang und Schroeder entscheidet der Agent selbständig zwischen der Auswahl einer Kategorie oder einem Wiederholungswurf mit allen oder nur bestimmten Würfeln. Dies gibt zwar mehr Freiheiten, gestaltet allerdings das (Er-)Lernen des Spiels herausfordernder.

4 Anforderungsanalyse

Bevor es an die konzeptionelle Ausarbeitung und Umsetzung geht, wird zuerst das zu entwickelnde System beschrieben und analysiert. Des Weiteren werden die Bedürfnisse und Wünsche verschiedener Stakeholder berücksichtigt, die zu unterschiedlichen Anwendungsfällen und Anforderungen führen. Zusammen bilden sie die Grundlage für eine systematische Umsetzung der Arbeit und dokumentieren das Projektziel. Ein weiterer Vorteil ist, dass sich die Ergebnisse durch die verschiedenen Anforderungen am Ende der Arbeit messen lassen. Hierdurch kann der Erfolg überprüft werden sowie verfolgt werden, welche Erkenntnisse aus der Umsetzung bestimmter Lösungswege gewonnen wurden.

4.1 Systembeschreibung

In der Abbildung 4.1 ist eine Übersicht des Gesamtsystems mit all seinen Komponenten und deren Zusammenhänge zu finden. Das System kann in zwei Teile unterschieden werden:

- den KI-Agenten und der Simulationsumgebung, und
- dem physischen Demonstrator der in einer ersten Betrachtung ausschließlich aus einem Bildverarbeitungssystem bestehen kann.

Zum Einen soll es möglich sein, dass die KI-Agenten und die Simulationsumgebung unabhängig vom Bildverarbeitungssystem interagieren. Zum Anderen soll es möglich sein, in den genannten Systemen unabhängig die Entwicklung vorantreiben zu können. Somit lässt sich Teil eins unabhängig von Teil zwei entwickeln. Dies ist möglich, weil die Zustandsaktualisierung aus der Simulationsumgebung erfolgt. Das Bildverarbeitungssystem kann als Schnittstelle für das Spielen mit echten Würfeln verwendet werden. Die Simulationsumgebung ist notwendig, damit der KI-Agent schnell und einfach trainieren kann.

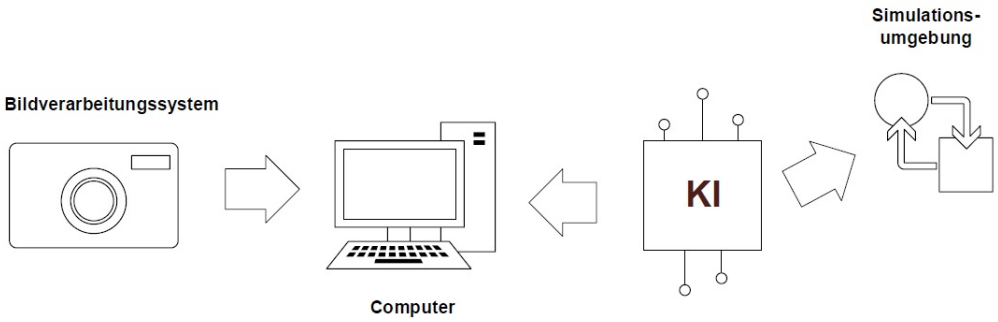


Abbildung 4.1: Systemumgebung.

4.2 Zielgruppen

Der Erfolg des Projekts hängt nicht nur von der technischen Machbarkeit, der Umsetzung und den funktionalen Anforderungen ab. Ebenso ist es entscheidend, die Bedürfnisse und Erwartungen aller beteiligten Interessensgruppen zu berücksichtigen. Die Stakeholder haben einen direkten oder indirekten Einfluss auf den Verlauf und den Erfolg der Arbeit. Die nachfolgend betrachteten Stakeholder haben ein besonderes Interesse am Erfolg dieser Abschlussarbeit.

4.2.1 Auftraggeber

Als Erstprüfer, Betreuer und Auftraggeber der Abschlussarbeit nimmt Herr Prof. Dr. Hensel eine zentrale Rolle ein. Er verfolgt dabei verschiedene Ziele und Interessen. Mit dem Demonstrator soll einerseits das Interesse von anderen Studierenden an RL geweckt werden. Andererseits soll damit die Möglichkeit geschaffen werden das Spiel Yahtzee mit Hilfe von KI zu lösen, da es einen nicht deterministischen Anteil besitzt. Nicht jedes Problem, das mittels Einsatz von KI gelöst werden soll, erfüllt die Anforderungen und Qualitäten für den Praxiseinsatz. Die Arbeit kann zudem als Anwendungsbeispiel für zukünftige studentische Arbeiten genutzt werden, um Weiterentwicklungen verschiedener KI-Methoden und Algorithmen zu erforschen. Aus Sicht des Prüfers ist nicht nur die fachliche Qualität der Lösung von Bedeutung, sondern auch der Erkenntnisgewinn und die methodische Bearbeitung der Aufgabenstellung. Dies schließt sowohl theoretische

Einblicke in die Funktionsweise von KI-Systemen als auch praktische Erfahrungen in deren Implementierung und Anwendung ein.

4.2.2 Autor der Arbeit

Mit der Erstellung der Arbeit liegt der Schwerpunkt auf der persönlichen Weiterentwicklung im Bereich der Künstlichen Intelligenz. Hierbei ist es es wichtig, umfassendes Grundlagenwissen zu erwerben und neue Fähigkeiten zu erlernen. Das Hauptziel besteht jedoch darin, am Ende der Arbeit einen Prototypen zu erstellen, der als Referenz dienen und von anderen Studierenden genutzt werden kann, um diese oder ähnliche Projekte fortzuführen oder weiterzuentwickeln.

4.2.3 Interessierte an Künstlicher Intelligenz

Als weitere Nutzergruppen lassen sich die Interessierten identifizieren. Dies könnten Personen sein, die sich einen greifbaren Eindruck von den Ergebnissen studentischer Arbeiten verschaffen möchten oder an der Art und Weise der Umsetzung interessiert sind. Das Projekt bietet eine ideale Plattform, um das Interesse an Künstlicher Intelligenz und technischer Innovation zu fördern. Es ermöglicht einen Einblick in die Potenziale der Technologie und zeigt, wie theoretische Konzepte praktisch umgesetzt werden.

4.2.4 Spieler

Des Weiteren haben Spieler die Möglichkeit, ihre strategischen und problemlösenden Fähigkeiten zu schärfen, während sie gleichzeitig Freude daran haben, sich mit einer KI zu messen. Das Projekt dient somit nicht nur als Bildungs- und Forschungsinstrument, sondern bietet auch eine unterhaltsame Freizeitbeschäftigung.

4.2.5 Weiterentwickler

Das System und die Ergebnisse dieser Arbeit bieten insbesondere für Studierende eine solide Grundlage für weitere Entwicklungen und Forschungsarbeiten. Vor allem MINT-Studierenden stellt das automatisierte Yahtzee-Spiel sowie die Simulation eine interessante Möglichkeit dar, theoretisches Wissen in die Praxis umzusetzen. Es gibt zudem zahlrei-

che Ansätze, um verschiedene Konzepte der Bildverarbeitung zu testen und miteinander zu vergleichen. Auch der physische Demonstrator eröffnet viel Raum für kreative Automatisierungslösungen bis hin zur Robotik. Neben den Studierenden könnte diese Arbeit auch Menschen aus der allgemeinen Öffentlichkeit als Inspirationsquelle für Selbststudien oder Optimierungen dienen.

4.3 Die KI-Agenten und die Simulationsumgebung

Die folgenden beiden Unterkapitel beschreiben und analysieren die Anwendungsfälle und Anforderungen an den KI-Agenten sowie der Simulationsumgebung. Sie stellen eine Anleitung für die Systementwicklung sowie die Erstellung von Testszenarien zur Bewertung bereit.

4.3.1 Anwendungsfälle

Dieses Unterkapitel betrachtet die Anwendungsfälle, die sich für die KI-Agenten und der Simulationsumgebung ergeben. Die Abbildung 4.2 visualisiert die Anwendungsfälle in einem Diagramm und stellt deren Beziehung zueinander dar. Es werden für eine Lösungsfindung verschiedene KI-Agenten ausprobiert. Für eine spätere Verwendung wäre es ebenfalls von Interesse die mit den entsprechenden Parametereinstellungen erzielte Ergebnisse zu reproduzieren oder das Lernverhalten der Agenten zu untersuchen. Dazu muss der Anwender den Agenten nicht nur trainieren sondern auch evaluieren können. Des Weiteren kann über die Lernparameter die Anzahl der Durchläufe, die der Agent trainieren soll, oder die Parameter, die das Lernverhalten beeinflussen, eingestellt werden. Mit der Möglichkeit, das Spiel konfigurieren zu können, kann die Größe der Punktetabelle, d.h. die Berücksichtigung des oberen und / oder unteren Tabellenabschnitts, angepasst und somit die Komplexität des Spiels verändert werden. Aus der Sicht des Agenten ist es wichtig mit der Simulationsumgebung interagieren zu können. Dazu wählt er eine Aktion aus, die die Simulationsumgebung durchführt und anschließend den neuen Zustand des Agenten aktualisiert. Je nach Agent stehen verschiedene Modelle zur Verfügung. Zudem soll die Möglichkeit der Speicherung des trainierten Modells aufgenommen werden, so dass es zu Beginn eines neuen Trainingslaufs geladen werden kann, um die Trainingszeiten zu verkleinern. Auch soll es möglich sein verschieden Trainingsstände laden oder

speichern zu können.

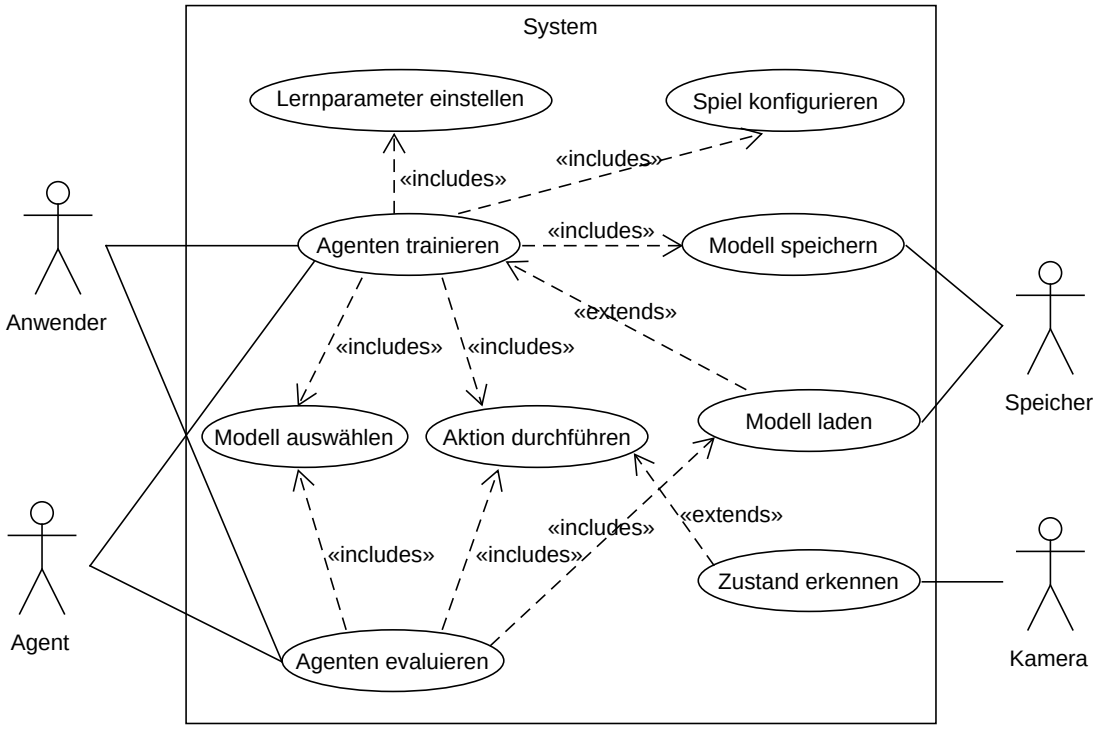


Abbildung 4.2: Anwendungsfalldiagramm der Software.

4.3.2 Anforderungen

Die Anforderungen werden in funktionale und nicht funktionale Anforderungen unterschieden. **Funktionale Anforderungen (F)** legen fest, welche Aufgaben und Funktionen das System erfüllen muss und welchen Zweck es verfolgt. Sie definieren demnach, was das System leisten soll. Im Gegensatz dazu beschreiben **nicht-funktionale Anforderungen (NF)** die Eigenschaften, Qualitätsmerkmale oder Einschränkungen des Systems. Sie beziehen sich darauf, wie das System seine Funktionen unter bestimmten Bedingungen ausführen soll. Dazu wird im Folgenden die Anforderungen der KI-Agenten **(KI)** ausformuliert.

KI-F1: Aufbau einer Simulationsumgebung, mit der die KI-Agenten interagieren können und in welcher die ausgewählten Aktionen durchgeführt werden.
Die Simulationsumgebung repräsentiert das Yahtzee-Spiel mit allen notwendigen Funktionen und Bedingungen, die für die Durchführung der Aktion notwendig sind. Sie gibt für das Training und die Evaluation die Zustandsaktualisierung an den Agenten zurück.

KI-F2: Die Lernparameter des Agenten müssen vor dem Training anpassbar sein.

KI-F3: Das Spiel muss konfigurierbar sein, sodass die Komplexität einstellbar ist.
Die Anforderung gilt als erfüllt, wenn die Punktetabelle des Yahtzee-Spiels angepasst werden kann. Hierzu soll die Möglichkeit bestehen, dass entweder nur mit der oberen Punktetabelle, der oberen Punktetabelle inklusive des Bonus oder der oberen und unteren Punktetabelle trainiert werden kann.

KI-F4: Der Trainingsfortschritt muss bewertbar visualisierbar sein.
Die Anforderung gilt als erfüllt, wenn für die Visualisierung ein Diagramm am Ende des Trainings generiert wird. Die Messbarkeit des Lernfortschritts ergibt sich aus der durchschnittlich erreichten Gesamtpunktzahl sowie der durchschnittlichen Punktzahl pro gewählter Kategorie.

KI-F5: Bei Agenten, die verschiedene Modelle besitzen, soll das Modell anpassbar sein.
Ein Modell kann zum Beispiel ein neuronales Netz sein. Dies kann aus mehreren Schichten bestehen. Die Anzahl der Schichten soll bis zu einem Maximum von drei Schichten eingestellt werden können.

KI-F6: Sind bereits Trainingsdaten vorhanden, sollen diese bei Bedarf geladen werden können.
Das Laden der vorhandenen Trainingsdaten ist für zwei Fälle besonders interessant. Zum Einen kann hiermit das Training bis zu einem bestimmten Punkt evaluiert werden oder es kann zum Anderen für die Fortführung des Trainings von einem bestimmten Punkt aus weiterverwendet werden.

KI-F7: Trainingsdaten sollen speicherbar sein.
Diese Anforderung gilt als erfüllt, wenn abhängig von den zuvor eingestellten Parametern die Trainingsdaten gespeichert werden können.

KI-F8: Nachdem alle notwendigen Einstellungen getätigt wurden, kann der Agent trainiert werden.
Der Agent soll selbstständig trainieren können und es soll während des Trainings keine weitere Aktion von außerhalb notwendig sein.

KI-NF1: Der Code soll in der Programmiersprache Python implementiert werden.

KI-NF2: Der Programmcode soll strukturiert und übersichtlich implementiert sein. Zudem soll eine Kommentierung erfolgen, die den Code gut lesbar gestaltet und eine einfache Wartung ermöglicht.
Es soll mit Klassen gearbeitet werden, um eine klare Struktur in zusammenhängende Bereiche zu schaffen. Außerdem sollen aussagekräftige Namen für Variablen, Methoden und Klassen gewählt werden.

4.4 Physischer Demonstrator

Der physische Demonstrator besteht unter anderem aus dem Bildverarbeitungssystem. Auch für die Bildverarbeitung sollen die Anwendungsfälle und Anforderungen beleuchtet werden.

4.4.1 Anwendungsfälle

Der wichtigste Anwendungsfall ist, dass die Kamera das Bild aufnimmt und daraus den Zustand mit Bildverarbeitungsmethoden ermittelt werden kann. Anschließend kann der Agent aus dem neu ermittelten Zustand eine Entscheidung über die Aktion ableiten.

4.4.2 Anforderungen

Daraus ergeben sich die Anforderungen für das Bildverarbeitungssystem **BV**.

BV-F1: Die fünf Würfel und deren entsprechende Augenzahl soll mit einer Sicherheit von 80% erkannt werden.
Die Anforderung gilt als erfüllt, wenn die Würfelaugen auf den Würfeln innerhalb der definierten Grenzen erkannt werden.

BV-F2: Für den Fall, dass die Augenzahl auf den Würfeln nicht richtig erkannt wird, muss es eine Korrekturmöglichkeit geben.

BV-F3: Nur der menschliche Spieler soll mit echten Würfeln spielen können.

BV-F4: Das System erkennt eigenständig, wenn ein neuer Wurf vorliegt bzw. übermittelt wurde.

BV-F5: Das System erlaubt keine Übermittlung des Bildes nach dem zweiten Wiederholungswurf.

BV-F6: Nach dem zweiten Wiederholungswurf steht nur die Auswahl eines freien Feldes auf der Punktetabelle zur Verfügung.

5 Konzept

Nachdem die Anforderungen an das System definiert sind, wird in diesem Kapitel auf die Vorgehensweise eingegangen. Dazu wird im Folgenden auf die notwendigen Werkzeuge, der konzeptionelle Aufbau der Software und mögliche Methoden, die in diesem Projekt untersucht werden, beschrieben. Die Software setzt sich grundlegend aus der Simulationsumgebung und dem Agenten zusammen.

5.1 Software

5.1.1 Entwicklungsumgebung und Programmiersprache

Die richtige Entwicklungsumgebung (IDE = Intergated Development Environment) und Programmiersprache sind nicht unerheblich für die Entwicklung eines Projektes. Es stehen viele Entwicklungsumgebungen zur Verfügung: Visual Studio Code, PyCharm, Spyder, Jupyter Notebook, PyDev etc. Einige dieser IDEs können über die Distribution Anaconda Navigator installiert werden, unter anderem Spyder. Die IDE Spyder bietet alle notwendigen Anwendungen, wie einen integrierten Debugger, einen Variableninspektor und eine interaktive Konsolenschnittstelle. Des Weiteren hilft Anaconda als Distribution dabei den Überblick über die installierten Bibliotheken zu behalten und bietet die Möglichkeit verschiedene Versionen einer Programmiersprache zu verwenden. Bei der Programmiersprache ist es von Relevanz eine solche zu verwenden, die eine große Beliebtheit besitzt, sodass das Ergebnis für eine große Interessengruppe zugänglich ist. Gleichzeitig sollte sie leicht zu verstehen sein und viele Bibliotheken für das Machine Learning besitzen. Die Wahl fiel hier auf Python. Sie bietet eine einfache Syntax und die Möglichkeit für eine strukturierte und objektorientierte Programmierung. Des Weiteren gibt es viele verschiedene Bibliotheken für das Thema Machine Learning. Zudem erfreut sich Python einer großen Anwenderzahl in Wissenschaft und Praxis[1]. [39]

[1] https://pypl.github.io/PYPL.html

5.1.2 Machine Learning Framework

Für das klassische Reinforcement Learning gibt es keine speziellen Machine Learning Frameworks, die die Handhabung vereinfachen. Beim Deep Learning hingegen sind die am häufigsten genutzten Frameworks TensorFlow und PyTorch [7]. Diese Frameworks sind in der Programmiersprache Python verfügbar und bieten eine breite Palette an Funktionen zur Erstellung von Machine-Learning- und Deep Learning-Modellen. TensorFlow wurde im Jahr 2016 von Google veröffentlicht, während PyTorch 2017 durch Facebook AI Research entwickelt wurde. In der Forschung gilt PyTorch heute als das bevorzugte Framework, während TensorFlow in der Industrie häufiger verwendet wird.[26] Die Kombination des klassischen Reinforcement Learnings mit dem Deep Learning und dessen Frameworks eröffnete die Möglichkeit neue starke Algorithmen für komplexere Problemstellungen zu entwickeln. Je nach Gegebenheit bietet das eine Framework Vorteile gegenüber dem Anderen. Somit wurde PyTorch für eine erste Implementierung gewählt, weil es eine bessere Nachvollziehbarkeit bietet. Ein späterer Umstieg auf TensorFlow mit Keras ist nicht ausgeschlossen. Weitere Bibliotheken, die verwendet werden, sind Pandas zur Daten Analyse in Python und Matplotlib für die Visualisierung von Diagrammen.

5.1.3 Simulationsumgebung

Für den Aufbau einer Simulationsumgebung und das Training von KI-Modellen ist **OpenAI Gym** ein äußerst wertvolles Werkzeug. Seit seiner Veröffentlichung im April 2016 durch die Firma OpenAI wird es kontinuierlich weiterentwickelt. Die Plattform bietet eine breite Auswahl an standardisierten Umgebungen, Algorithmen und verschiedenste Methoden, was die Entwicklung und den Vergleich von KI-Modellen deutlich vereinfacht. Als standardisierte Schnittstelle ermöglicht OpenAI Gym dem KI-Agenten sowohl mit vordefinierten als auch benutzerdefinierten Umgebungen zu interagieren. Im Kern dieser Python-Bibliothek steht die Fähigkeit eines Agenten mit seiner Umgebung auf flexible Weise zu kommunizieren. Dadurch besteht die Möglichkeit eine Simulationsumgebung aufzubauen, die alle KI-Agenten verwenden können. Dies ist auch für weiterführende Arbeiten interessant, wenn eine Erweiterung mit anderen Algorithmen gewünscht ist.

5.2 Lernstrategie des Spiels Yahtzee

Yahtzee birgt beim Spielen verschiedene Herausforderungen. Zum Beispiel kann keine Vorhersage über den nächsten Spielzug gemacht werden. Der Spieler weiß nicht was er als nächstes würfelt. Er kann daher nicht vorher entscheiden, wann die Wahl eines Felds sinnvoll ist. Daher besitzt das Spiel eine große Abhängigkeit vom Zufall. Die einzige Möglichkeit den Zufall zu reduzieren, geht ausschließlich durch das Auswählen der Würfeln, die der Spieler erneut im Rahmen seiner drei Versuche würfeln möchte. Allein der erste Wurf (Initialwurf) eines Zuges besitzt bei Berücksichtigung aller Würfelanordnungen $6^5 = 7.776$ Möglichkeiten. Danach darf der Spieler noch zweimal würfeln und sich bei jedem Wurf entscheiden, welche der fünf Würfel er erneut würfeln möchte. Um die Anzahl an Möglichkeiten schon beim Initialwurf gering zu halten, ist es sinnvoll die Würfel zu sortieren, zum Beispiel von links nach rechts aufsteigend. Dadurch wird die Anordnung von 7.776 Möglichkeiten auf $\binom{10}{5} = 252$ Möglichkeiten reduziert. Hinzu kommen dann noch die beiden Wiederholungswürfe und die Auswahl des Felds. [14]

Um auch hier die Komplexität des Spiels weiter zu minimieren, gibt es verschiedene Ansätze. Die Punktetabelle wird dazu wie in der Abbildung 5.1 veranschaulicht in unterschiedliche Segmente unterteilt. Das erste Segment (grün) des oberen Tabellenteils wird immer trainiert. Die weiteren beide Segmente (in orange und rot) können unabhängig voneinander zusätzlich ausgewählt werden. So kann zum Beispiel nur mit dem oberen Tabellenteil trainiert werden und der Untere bei Bedarf mit dazugenommen werden. Dies bringt verschiedene Vorteile mit sich. Der Zustandsraum wird verkleinert. Die Zeit pro Spiel und somit auch die Trainingszeit des Agenten werden verkürzt. Dieser befasst sich hierdurch erst einmal mit kleineren Herausforderungen und nicht von Beginn mit dem gesamten Spiel. Des Weiteren ist es eine gute Option den Bonus für den oberen Tabellenteil analog dem unteren Tabellenteil auswählbar zu machen. Der Bonus kann für den Agenten für den Anfang eine zu abstrakte Belohnung sein, die er erst bei Erreichen von mehr als 63 Punkten im oberen Tabellenteil bekommt. Dies wird im schlimmsten Fall nie erreicht. Zusätzlich ist es gut für den Agenten am Anfang ohne Wiederholungswürfe zu starten, sodass er den Initialwurf nur den Abschnitten der Punkttabelle zuordnen muss. Mit diesen Aufteilungen kann die Komplexität schrittweise gesteigert und geschaut werden, ab wann der Agent an seine Grenzen stößt, sowohl beim Lernen als auch beim Spielen.

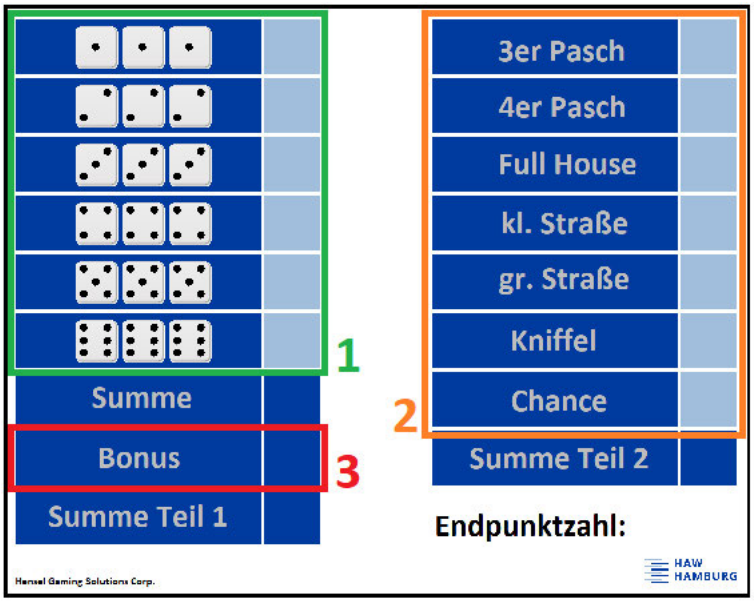


Abbildung 5.1: Beispiel einer Yahtzee Punktetabelle mit den einstellbaren Trainingsbereichen.

5.3 Der Agent

Neben den verschiedenen Methoden, die für die Umsetzung in Betracht kommen, stellt sich auch die Frage, ob das System mit einem oder zwei Agenten aufgebaut werden soll. So kann zum Beispiel ein Agent entscheiden, ob er einen Wiederholungswurf tätigen oder ein leeres Feld auswählen möchte. Andererseits können die Aufgaben auch aufgeteilt werden. Zum Beispiel lernt ein Agent, welche Würfel er neu würfelt und ein zweiter Agent, welche Felder er auswählt. Die Abbildung 5.2 veranschaulicht dies, indem links das System mit einem Agenten und rechts das System mit zwei Agenten dargestellt ist.

5.3.1 System mit einem Agenten

Das System mit einem Agenten bietet den Vorteil, dass nur ein Modell trainiert werden muss. Der Nachteil kann unter anderem sein, dass sich das Modell nur für die Auswahl der Felder eignet, aber zum Beispiel bei der hohen Komplexität der Auswahl der Würfel scheitern könnte. Hier könnte bei einem System mit zwei Agenten für die jeweilige Aufgabe das entsprechend beste Modell verwendet werden. Ein Vorteil bei einem System

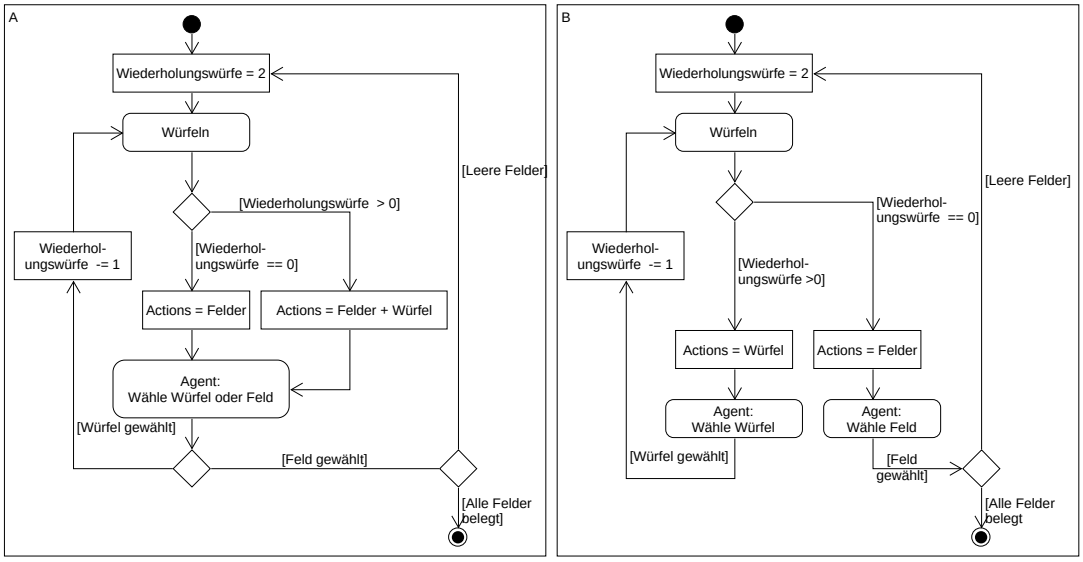


Abbildung 5.2: Aufbau des System mit links einem Agenten und rechts zwei Agenten.

mit nur einem Agenten ist, dass bei der Durchführung der Aufgaben nur ein Belohnungssystem berücksichtigt werden muss. Die Belohnung kann zum Beispiel die gewonnenen Punkte sein. Dies stellt gleichzeitig auch das Hauptziel des Spiels dar so viele Punkte wie möglich zu erzielen. Hierdurch muss nur ein Modell kontrolliert werden, ob dieses richtig lernt. Dies vereinfacht die Entwicklung. Bei einem System mit zwei Agenten hingegen ist dies ein Nachteil, da zwei Agenten auf ihren Lernfortschritt hin überprüft werden müssen. Des Weiteren stellt sich die Frage, ob beide das gleiche Belohnungssystem nutzen können. Für die Auswahl der Felder können zum Beispiel, wie auch beim System mit einem Agenten, die gewonnenen Punkte genutzt werden. Doch wie sieht das beim Agent aus, der die Würfel auswählt? Wie wird die Wahl der Felder belohnt und was ist eine gute Wahl und was nicht? Nach der Abbildung 5.2 wird ersichtlich, dass eine Kommunikation der Agenten nicht zwangsweise notwendig ist. Dies würde zwar die Umsetzung erleichtern, jedoch nicht wesentlich vereinfachen. Werden die Vor- und Nachteile gegeneinander abgewogen, stellt sich eine Implementierung mit einem Agenten als vorteilhaft dar. Die Tabelle 5.1 fasst die zuvor beschriebenen Kriterien in einer Tabelle zusammen, wobei + positiv gegenüber dem anderen Agentensystem und − negativ gegenüber dem anderen Agentensystem zu bewerten ist.

Kriterien	1 **Agent**	2 **Agenten**
Implementierungsaufwand	+	-
Belohnungssystem	+	-
Aufgabenverteilung	-	+
Training und Validierung	+	-
Extraaufwand	+	-

Tabelle 5.1: Kriterien für ein ein Agenten- oder zwei Agentensystem.

5.3.2 Methoden

In der vorliegenden Arbeit werden ausschließlich Methoden in Betracht gezogen, die einen Bezug zum Reinforcement Learning haben. Hierzu wurden einige Ansätze im Grundlagenkapitel vorgestellt. Im weiteren Verlauf der Arbeit werden Q-Learning und Deep Q-Learning verwendet. Hierzu wird zuerst das Q-Learning untersucht und entwickelt, weil dies eine Methode ist, die ausschließlich auf dem Prinzip des Reinforcement Learnings beruht. Q-Learning ist gegenüber der Monte-Carlo-Methode effizienter beim Lernen, weil schon während des Trainings die Q-Tabelle aktualisiert und für das weitere Training verwendet wird. Bei der Monte-Carlo-Methode [7] erfolgt dies erst am Ende das Trainings. Generell wird erwartet, dass der Zustandsraum für eine Methode, die ausschließlich auf einen Bereich des Machine Learnings beruht, zu groß ist. Für die tabellarischen Lösungsverfahren bedeutet dies einen sehr großen Speicherbedarf, um die Zustands-Aktions-Wertepaar abzuspeichern. Anschließend wird die natürliche Erweiterung des Q-Learnings, das Deep Q-Learning bzw. die Deep Q-Networks untersucht. Dies ist eine Kombination aus Deep und Q-Learning. Die Vorteile sind ein geringerer Speicherbedarf und weniger Trainingszeit als beim Q-Learning. Es muss jedoch zusätzlich ein neuronales Netz entwickelt, ein Erfahrungsspeicher zur Steigerung der Lerneffizienz implementiert und mehr Parameter eingestellt werden. Dies bedeutet auf den ersten Blick gegenüber dem Q-Learning Algorithmus mehr Aufwand, erzielt bei erfolgreichem Training jedoch wesentlich bessere Ergebnisse. Zunächst wird eine Epsilon-Greedy-Policy verwendet, um das Ziel, eine höchstmögliche Punktzahl zu erreichen, verfolgt wird. Diese Strategie wird für beide Methoden angewendet.

Bei der SARSA-Methode wird nicht nach dem bestmöglichen Ergebnis (maximale Punktzahl) gestrebt, sondern ressourcenschonend versucht, die Aufgabe zu erledigen. Beim Policy Gradient-Methode legt diese zu viel Wert darauf, Aktionen in Zuständen zu verbessern, in denen der Agent bereits gelernt hat mit diesen umzugehen. Dies macht sie

sehr ineffizient und führt zu sehr langen Trainingszeiten. Die Actor-Critic-Methode steht vor dem Dilemma, dass eine Entscheidung des Akteurs sowohl gut als auch schlecht sein kann. Zum Beispiel entscheidet sich der Akteur dafür drei Dreien zu behalten, obwohl die Felder für die Dreien und die Chance schon belegt sind, da dieser mit den verbleibenden Würfen einen Yahtzee erzielen möchte. Dann würde der Kritiker dies als gut bewerten, obwohl das Risiko sehr hoch ist. Wird kein Yahtzee erzielt und ein Feld muss im schlimmsten Fall mit null Punkten bewertet werden, dann wird die Entscheidung schlechter bewertet. Der Zufall kann bei diesem Beispiel eine starke Gewichtung einnehmen. Des Weiteren scheint der Implementierungsaufwand bei der Actor-Critic-Methode größer als beim Deep Q-Learning zu sein. Dies sind weitere Gründe, warum sich für eine Umsetzung der Q-Learning- und der Deep Q-Learning-Algorithmen in der vorliegenden Arbeit entschieden wurde.

6 Entwicklung der Simulationsumgebung

Dieses Kapitel beschreibt den Aufbau einer benutzerdefinierten Simulationsumgebung nach dem Standard von OpenAI Gym, sodass verschiedene Reinforcement Learning Agenten die gleiche Simulationsumgebung nutzen können. Dies eröffnet auch die Möglichkeit des Vergleichs verschiedener Agenten und wie gut diese die Problemstellung lösen. Hierzu wird zuerst auf den allgemeinen Aufbau der benutzerdefinierten Umgebung nach OpenAI Gym eingegangen. Anschließend wird die Umsetzung des Würfelspiels Yahtzee beschrieben. Zum Schluss werden die Schnittstellen erläutert, die einem Agenten zur Verfügung stehen, um mit der erstellten Yahtzee Umgebung zu interagieren.

6.1 OpenAI Gymnasium API

OpenAI bietet zum Einen die Möglichkeit, vorgefertigte Umgebungen zum Trainieren von Agenten zu verwenden. Bekannte Beispiele sind u.a. Cartpole, Pendulum, MountainCar oder Lunar Lander. Zum Anderen können eigene Trainingsumgebungen erstellt und mit diesen interagiert werden. Hierzu empfiehlt es sich die API (Application Programming Interface) nach OpenAI Gym zu verwenden. Die Environment-Klasse besteht im Wesentlichen aus den vier folgenden Methoden:

- __init__(): Die Init-Methode ist der Konstruktor und initialisiert das Environment und definiert den Aktions- und Zustandsraum.
- reset(): Diese Methode wird zu Beginn jeder Episode aufgerufen und versetzt das Environment in seinen Initialzustand zurück. Alle Parameter und Zustandsräume werden zurückgesetzt.
- step(): Bei der Step-Methode wird der Parameter mit der nächsten auszuführenden Aktion (Action) übergeben. Anschließend wird diese ausgeführt und überprüft, ob eine Belohnung (Reward) erhalten wurde. Zum Schluss gibt die Methode den neuen

Zustand (Next State), die Belohnung, ob die Trainingsepisode beendet wurde oder nicht und, falls nötig, weitere Informationen zurück.

- render(): Mit der Render-Methode wird die Möglichkeit gegeben, die Aktionen des Agenten grafisch zu visualisieren. Sie bietet somit eine Möglichkeit das Lernverhalten des Agent zu analysieren. Die Bibliothek PyGame bietet für die Visualisierung weitere Unterstützungen und Möglichkeiten.
- close(): Sie ist eine zusätzliche Methode, die am Ende eines Trainings alle notwendigen Bereinigungen durchführt. Umgebungen, in denen das Programm durchgelaufen ist oder durch Fehler- bzw. Ausnahmebedingungen abgefangen wurde, schließen sich automatisch von alleine.

6.2 Die benutzerdefinierte Umgebung Yahtzee

In der Bibliothek von OpenAI für die Standard-Trainingsumgebungen gibt es keine für das Würfelspiel Yahtzee. Bevor mit der Implementierung begonnen werden kann, müssen einige Vorüberlegungen getroffen werden. Wie sieht der Zustandsraum aus, welche Aktionen kann der Agent auswählen und wie soll das Belohnungssystem aussehen?

Der **Zustandsraum** (*state_space* oder *observation_space*) wurde in Anlehnung an die Arbeit von Kang und Schroeder [14] entwickelt. Dabei werden die fünf Würfelwerte (*dice_values*), die Kategorien der Punktetabelle, ob diese leer sind oder nicht (*fields_empty*) und wie viele Wiederholungswürfe noch zur Verfügung stehen (*remainings_rolls*), zusammengefasst. Der Zustandsraum ist zu Beginn als Dictionary definiert und kann später beliebig in ein Tupel oder Array transformiert werden. Das Listing 6.1 zeigt den verwendeten Zustandsraum. Im Listing 6.2 sind zwei Beispiele des Zustandsraums dargestellt, das Erste für ein Dictionary und das Zweite für die Darstellung eines Arrays. In dieser Darstellung ist nur der Zustandsraum für ein Spiel für den oberen Tabellenteil visualisiert. In einer frühen Phase der Arbeit wurden auch die behaltenen Würfel (*kept_dice*), also die Würfel, die während eines Wiederholungswurfs nicht erneut gewürfelt wurden, mit im Zustandsraum aufgenommen. Diese wurden wieder entfernt, weil sie die Trainingszeit erhöhten und keine Vorteile bei der Punktemaximierung bzw. beim Training des Agenten brachten.

Des Weiteren müssen dem Agenten die gleichen **Aktionen** zur Verfügung stehen wie

```
# Observation space = state space
self.observation_space = gym.spaces.Dict(
    {
        # Values of the dice
        'dice_values': gym.spaces.MultiDiscrete([6] * 5),
        # Score card boxes (is_selected, True or False)
        'fields_empty': gym.spaces.MultiDiscrete([2] * number_boxes),
        # Remaining dice rolls
        'remaining_rolls': gym.spaces.Discrete(2)
    }
)
```

Listing 6.1: Zustandsraum des Würfelspiels.

```
state_space_Dictionary = {'dice_values' = [1, 1, 4, 6, 6],
                          'fields_empty' = [0, 1, 0, 0, 1, 1],
                                           'remaining_rolls' = [2]}

state_space_Array = [1 1 4 6 6 0 1 0 0 1 1 2]
```

Listing 6.2: Beispiel des Zustandsraums als Dictionary und Array für den oberen Tabellenteil mit Zufallswerten.

einem menschlichen Spieler, d.h. er darf zwischen erneutem Würfeln von allen oder nur ausgewählten Würfeln und dem Auswählen eines noch freien Felds entscheiden. Hier ist bei der Entwicklung auf bestimmte Restriktionen zu achten. Wenn der Agent keine Wiederholungswürfe mehr hat, darf er nur noch ein leeres Feld auswählen. Mehrfachbelegung von Feldern ist dem Agenten ebenfalls untersagt (siehe Spielregeln Kapitel 2.1). Deshalb sind Aktionen, die die Auswahl der Felder betreffen, mit zunehmenden Spielverlauf aus dem Aktionsraum zu entnehmen. Auch für den Aktionsraum (*action_space*) wurde sich an der Arbeit von Kang und Schroeder [14] orientiert. Dieser setzt sich aus der Auswahl einer Kategorie der Punktetabelle (*scorecard_action_space*) und der Möglichkeit eines Wiederholungswurfs bestimmter oder aller Würfel (*dice_action_space*) zusammen.

```
self.total_action_space = gym.spaces.Discrete(len(self.
    scorecard_action_space) + len(self.dice_action_space))
```

Wichtig ist, dass die jeweiligen Aktionsräume nicht miteinander vertauscht werden dürfen. Dies hat folgende Ursache: Abhängig davon, ob mit oder ohne dem unteren Tabellenabschnitt gespielt wird, ist der Aktionsraum entweder 37 oder 44 auswählbare Aktionen groß. Hierbei sind die ersten Aktionen entweder von [0, ..., 5] oder [0, ..., 12] für die Auswahl der Felder reserviert und die restlichen Aktionen für die Auswahl der Würfel der

Wiederholungswürfe. Dabei kann der Fall, dass alle Würfel behalten werden, vernachlässigt werden, weil dies der Auswahl einer Kategorie gleichzusetzen ist. Damit die richtigen Würfel behalten werden, muss demzufolge der Aktionsraum für die Auswahl der Felder (*scorecard_action_space*) vom gesamten Aktionsraum abgezogen werden. Folglich wird der richtige Integer Wert der ausgewählten Aktion in eine binäre Darstellung umgewandelt, um die Würfel zu behalten. Das folgende Beispiel soll dies veranschaulichen:

```
Aktionsraum = 37
Wurf = [1, 1, 4, 6, 6]
ausgewaehlte Aktion = 33
ausgewaehlte Aktion - Aktionsraum Auswahl Felder = 27
bin(27) = [1, 1, 0, 1, 1]
```

Aus dem Beispiel geht hervor, dass nur die Vier erneut gewürfelt wird.

Zum Schluss bleibt noch zu entscheiden, wann der Agent eine **Belohnung** (*reward*) erhält. Wird dem Agenten erst ganz zum Schluss die Endpunktzahl pro Spiel mitgeteilt, kann dies zu spät sein. Deshalb wurde zu Beginn der Arbeit entschieden, dass bei der Auswahl eines Feldes die Punktzahl pro Runde als Belohnung zurückgegeben wird. Dies ähnelt dem Spielprinzip, dass die Punkte nach jeder Runde in die Tabelle eingetragen werden. Im Laufe der Arbeit wurde die Rückgabe der Punktzahl um die Rückgabe des Verhältnisses des aktuellen Gewinns einer Kategorie zu der maximal erreichbaren Punktzahl in dieser Kategorie (*reward_ratio*) erweitert. Das sind die einzigen Belohnungen, die der Agent von der Umgebung während des Trainings erhält.

Nach den Vorüberlegungen wird nun auf die Umsetzung eingegangen. Die für das Training der Agenten entwickelte Umgebung besteht aus drei Klassen und ist im Klassendiagramm 6.1 veranschaulicht. Die Klasse YahtzeeEnv ist die Schnittstelle für die Agenten nach dem Vorbild der OpenAI Gym API. Des Weiteren importiert sie die Klassen Dice und Scorecard.

Die **init**-Methode erhält drei Parameter: *seed*, *has_lower_part* und *has_bonus*. Mit dem Integer *seed* kann ein Zahlenwert übergeben werden, der die Reihenfolge der zufällig generierten Würfel beeinflusst. Dies ist für das Trainieren der Agenten und die Suche nach guten Lernparametern wichtig. Mit den Boolean-Parametern *has_lower_part* und *has_bonus* kann eingestellt werden, ob der untere Tabellenteil und / oder der Bonus für das Training berücksichtigt werden. In Abhängigkeit der übergebenden Parameter wird der

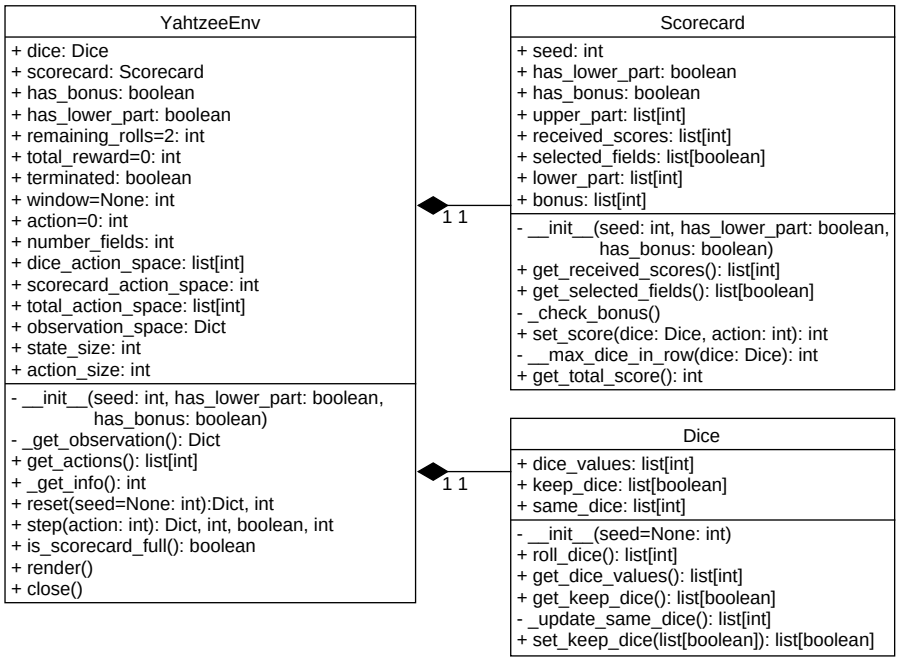


Abbildung 6.1: Klassendiagramm der benutzerdefinierten Umgebung für Yahtzee.

```
def _get_observation(self):
    dice_values = self.dice.get_dice_values()
    open_fields = [1 if item == True else 0 for item in self.
    scorecard.get_selected_fields()]
    return { 'dice_values': dice_values, 'fields_occupied':
    open_fields, 'remaining_rolls': self.remaining_rolls }
```

Listing 6.3: _get_observation-Methode.

Zustands- und Aktionsraum initialisiert. Die Parameter werden an die Klasse Scorecard weitergegeben. Die Klasse Dice hingegen benötigt nur den *seed*. Von der YahtzeeEnv ausgehend werden die Klassen Scorecard und Dice initialisiert.

Die Methode **_get_observation** gibt den aktuellen Zustand zurück. Dies beinhaltet den Zustandsraum, wie er in den Vorüberlegungen dargestellt ist. Sie ist wichtig, damit der Agent seinen aktuellen Zustand erhält. Das Listing 6.3 zeigt den dazugehörigen Code.

Die **reset**-Methode setzt die Umgebung auf ihren Initialzustand zurück. Dies bedeutet, dass die erreichte Gesamtpunktzahl auf null zurückgesetzt wird und die belegten

```
def reset(self, seed = None):
    self.remaining_rolls = 2
    self.total_reward = 0
    self.terminated = False
    self.dice = Dice (seed = seed)
    self.scorecard.__init__(None, self.has_lower_part, self.has_bonus
    )
    self.dice_action_space = list(range(self.number_fields, self.
    number_fields + (2**5 - 1)))
    self.scorecard_action_space = list(range(0, self.number_fields))
    info = None
    return self._get_observation(), info
```

Listing 6.4: Reset-Mehode.

Felder der Scorecard wieder freigegeben werden. Dafür wird die Klasse Scorecard neu initialisiert. Die Methode gibt den zurückgesetzten Zustandsraum zurück und das Spiel kann von Neuem starten. Das Listing 6.4 veranschaulicht die dazugehörige Methode.

Die **step**-Methode führt die vom Agenten ausgewählte Aktion aus. Dies kann entweder die Wahl eines leeren Feldes der Punktetabelle oder das erneute Würfeln bestimmter Würfel sein. Die Methode kontrolliert dabei, ob die Aktion gültig ist. Das bedeutet beispielsweise, dass das Environment unterscheiden muss, ob die gewählte Aktion ein Wiederholungswurf oder die Auswahl eines Feldes ist. Wird ein Feld ausgewählt, wird diese Aktion nach der Ausführung aus dem Aktionsraum entfernt. Bei der Wahl dieser Kategorie wird die Punktzahl (*reward*) und das Verhältnis zur maximal möglichen Punktzahl (*reward_ratio*) zurückgegeben. Die Erweiterung des *rewards* um das *reward_ratio* wird im folgenden Kapitel näher beschrieben. Dynamiken, die das Lernverhalten erschweren, sind zum Beispiel, wenn der Agent gelernt hat, dass viele Sechsen eine hohe Belohnung bringen, aber das Feld schon belegt ist. Versucht er dennoch gezielt die Anzahl der Sechsen zu erhöhen, bringt dies eine niedrigere Gesamtpunktzahl, weil das Feld bereits belegt ist. Des Weiteren wird nach dem Entfernen der Aktion für das ausgewählte Feld die Anzahl der übrigen Wiederholungswürfe für die nächste Runde wieder auf zwei gesetzt und der Initialwurf ausgeführt. Ist die ausgewählte Aktion ein Wiederholungswurf, wird die Länge des Aktionsraums für die Feldauswahl abgezogen und die Dezimalzahl in ihr binäres Äquivalent umgewandelt. Diese wird dann an die entsprechende Methode übergeben, die sich die entsprechenden Würfel merkt und nur die Übrigen würfelt. Sollte die Aktion in keinem der beiden Bereiche liegen, wird eine Ausnahme zurückgegeben. Dies hat sich bei der Entwicklung der Agenten als hilfreich erwiesen, wenn die ausgewählten

```
def step(self, action):
    self.action = action
    reward = 0
    reward_ratio = 0
    if action in self.scorecard_action_space:
        reward, reward_ratio = self.scorecard.set_score(self.dice,
    action)
        self.scorecard_action_space.remove(action)
        # reset all round parameters
        self.remaining_rolls = 2
        self.dice.keep_dice = [False] * 5
        self.dice.roll_dice()
        self.dice._update_same_dice()
    elif action in self.dice_action_space and self.remaining_rolls >
    0:
        binActionValue = [bool((action-self.number_fields) & (1<<n))
    for n in range(5)]
        self.dice.set_keep_dice(binActionValue)
        self.dice.roll_dice()
        self.dice._update_same_dice()
        self.remaining_rolls -= 1
    else:
        raise Exception("sth went wrong! Neither field category nor
    dice were selected!")
    self.observation = self._get_observation()
    self.terminated = self.is_scorecard_full()
    info = self._get_info()
    # contain most of the logical env
    return self.observation, reward, self.terminated, info,
    reward_ratio
```

Listing 6.5: Step-Mehode.

Aktionen außerhalb der Grenzen lagen. Im Normalfall gibt diese Methode den neuen Zustand (*next_state*), die Belohnung (*reward*), die Beendigung der Episode (*terminated*) und eine zusätzliche Information (*info*) zurück. Wie zuvor erwähnt, wurde die Rückgabe um das (*reward_ratio*) erweitert. Die entsprechende Methode ist im Listing 6.5 dargestellt.

In der Klasse YahtzeeEnv sind die folgenden, weiteren Methoden enthalten:

- **get_actions**: Der Agent erhält beim Aufruf dieser Methode die gültigen Aktionen für die aktuelle Runde.

- **is_scorecard_full**: Hier wird überprüft, ob alle leeren Felder der Scorecard belegt sind und der Parameter, der die Beendigung einer Episode definiert (terminated), auf *wahr* gesetzt wurde. Es gibt noch den Parameter truncated, der angewendet wird, wenn zum Beispiel eine Episode zu lange dauert oder andere Abbruchbedingungen erfüllt sind. In diesem Fall wird die Episode vorzeitig beendet, aber nicht das Training. Dieser Parameter findet hier keine Anwendung, da das Spiel deterministisch ist und nach maximal 13 Runden oder 39 Zügen endet.
- **render**: Die Methode bietet die Möglichkeit der visuellen Darstellung, wurde aber nicht weiter berücksichtigt, weil die Trainingszeiten der Agenten dadurch verlängert wird. Bei Bedarf können die Spielzüge über die Ausgabekonsole ausgegeben und nachvollzogen werden.
- **_get_info**: Mit dieser Methode können zusätzliche Informationen bereitgestellt werden. Sie ist implementiert, wird jedoch in dieser Arbeit nicht verwendet.
- **close**: Die close-Methode beendet noch alle offenen Ressourcen, die von dem Environment benötigt wurden, wie das Listing 6.6 zeigt.

```
def close(self):
    if self.window is not None:
        self.showrender = False
        pygame.display.quit()
        pygame.quit()
```

Listing 6.6: Close-Mehode.

Von der Klasse YahtzeeEnv ausgehend werden die Klassen Dice (*Würfel*) und Scorecard (*Punktetabelle*) initialisiert. Sie bilden alle notwendigen Funktionalitäten für die Würfel und die Punktetabelle ab. Für die Klasse Dice bedeutet dies die Ausgabe der Würfelwerte (*dice_values*) und welche Würfel behalten werden sollen (*keep_dice*) und welche Würfel für die Punkteermittlung gleich sind (*same_dice*). Des Weiteren muss der Agent zum Spielen Wiederholungswürfe durchführen können. Dafür muss er die aktuellen Würfelwerte von der Klasse Dice erhalten. Der Agent übergibt diese der Punktetabelle oder wählt für den Folgewurf welche Würfel behalten werden sollen.

Für die Scorecard muss bei der Initialisierung darauf geachtet werden, ob der Bonus und der untere Tabellenteil für das Training berücksichtigt werden. Hinzu kommt, dass der Agent die Punkte setzen bzw. die Liste mit den aktuellen Punkten eventuell auslesen

können muss. Nach der Definition des Zustandsraums braucht der Agent eine Liste der Punktetabelle mit den belegten Feldern. Zum Schluss benötigt der Agent die Endpunktzahl für die Auswertung. Für einen schnellen Test, ob die YahtzeeEnv funktioniert, wurde ein Random-Agent im *main*-Abschnitt der YahtzeeEnv implementiert, der zufällige Aktionen auswählt.

7 Entwicklung der Agenten

In diesem Kapitel wird auf den Aufbau und die Umsetzung der beiden Agenten eingegangen, die das Würfelspiel erlernen sollen. Hierzu wird zuerst der Q-Learning Algorithmus und anschließend der Deep Q-Learning Algorithmus beschrieben. Im Weiteren werden diese Beiden als Q-Agent und DQN-Agent bezeichnet. Der Q-Agent ist, wie im Kapitel 2.4.1 beschrieben, ein reiner Reinforcement Learning Algorithmus. Seine Entwicklung ist verglichen zum DQN-Agent einfacher. Dies spiegelt sich auch in der Funktionsweise bzw. beim Trainieren wieder. Andererseits sind die Einsatzmöglichkeiten nicht so flexibel wie beim DQN-Algorithmus.

7.1 Aufbau der Agenten Klassen

7.1.1 Q-Agent

Der Q-Learning Off-Policy Algorithmus, der in dieser Arbeit verwendet wird, wird in der Abbildung 7.1 in Anlehnung an Sutton [34] beschrieben. Zu Beginn wird die Q-Tabelle $Q(s, a)$ initialisiert. Anschließend wird für die aktuelle Trainingsepisode der Startzustand initialisiert. Danach wird für jeden Schritt der aktuellen Trainingsepisode eine Aktion A im Zustand S nach der gewählten Policy (ϵ-*greedy*) für Q ausgewählt. Nach Ausführung wird beobachtet, was der neue nächste Zustand ist. Danach wird die Q-Tabelle nach der Formel 2.34 aktualisiert. Zum Schluss wird der neue Zustand übergeben. Dies wiederholt sich so lange bis der Terminierungszustand der aktuellen Trainingsepisode erreicht ist. Im Fall von Yahtzee ist der letzte Zustand erreicht, wenn die Punktetabelle voll ist. Aus diesem Algorithmus hat sich für den Q-Agent das Klassendiagramm 7.2 mit den entsprechenden Methoden ergeben.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Abbildung 7.1: Q-Learning Algorithmus aus [34].

Agenteninitialisierung:
Für den Q-Agent ist nur eine Klasse (QAgent) notwendig. Alle notwendigen Parameter werden bei der Initialisierung durch ein Dictionary *parameter* übergeben. Über das Dictionary kann definiert werden, wie die Punktetabelle für das Training aussieht. Des Weiteren können die Lernschrittweite, der Diskontierungsfaktor, die Anzahl der Trainings- und Evaluierungsepisoden sowie die Werte für die Epsilon-Strategie, also wie schnell verkleinert sich Epsilon und sein Minimalwert, eingestellt werden. Zusätzlich kann die Fenstergröße angepasst werden, mit dem die Graphen der Diagramme für die Auswertung lokal gemittelt werden. Dadurch sind diese weniger verrauscht für die Interpretation. Die Abbildung 7.3 zeigt ein verrauschtes und ein lokal gemitteltes Diagramm mit der Fenstergröße von 100. An dieser Stelle wird auch die Q-Tabelle wie nach dem Algorithmus 7.1 initialisiert. Des Weiteren wird eine Liste für die Temporal Difference Werte erzeugt, um zu veranschaulichen, dass diese mit der Zeit abnehmen.

Epsilon-Aktualisierung:
Die Anpassung von Epsilon erfolgt nach jedem Spiel. Würde die Aktualisierung nach jedem Spielzug erfolgen, wäre der Epsilonwert sehr schnell bei seinem Minimalwert. Die Aktualisierung erfolgt in der Methode **decay_epsilon** nach dem folgenden Schema:

$$\epsilon = \epsilon - \epsilon_{decay\_rate}$$
$$\epsilon \leftarrow \max(\epsilon\,, \epsilon_{min})$$

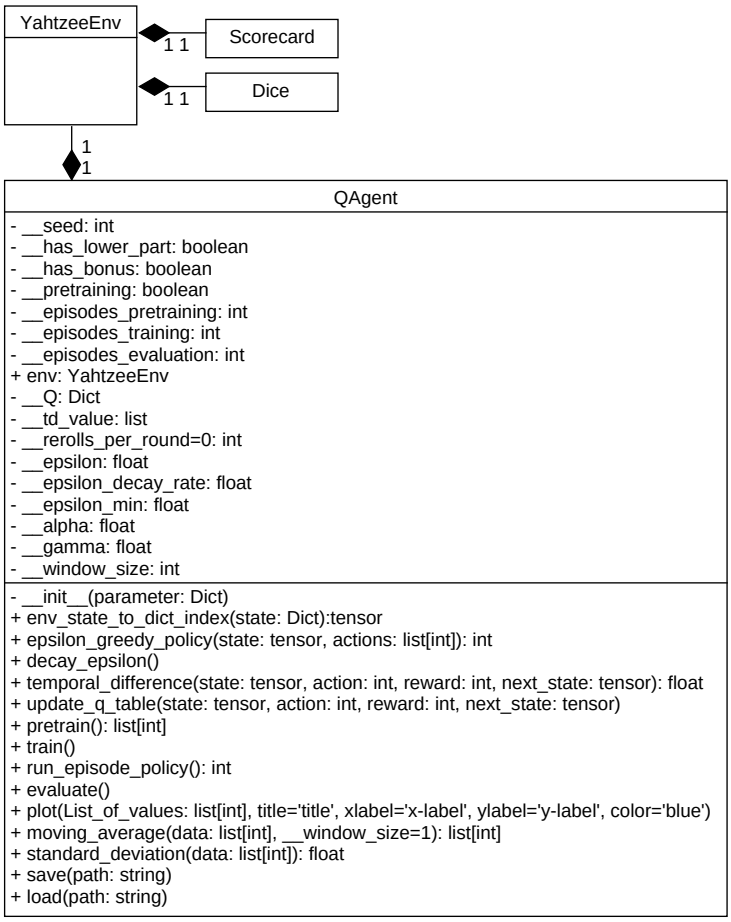
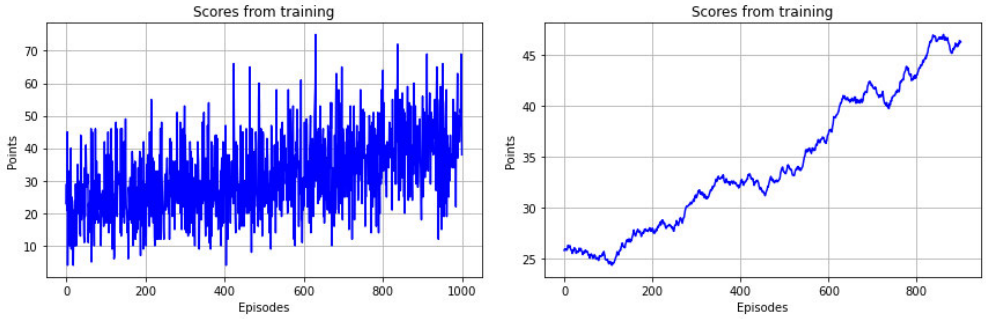


Abbildung 7.2: Klassendiagramm des Q-Agenten.

Epsilon wird dabei, um die ϵ_{decay_rate} verringert und kann nicht kleiner als der Wert ϵ_{min} werden.

Aktionsauswahl:
Die Aktionsauswahl erfolgt in der Methode *epsilon_greedy_policy* nach der im Kapitel 2.2.2 vorgestellten Strategie. Dafür wird der aktuelle Zustand sowie die für diese Runde gültigen Aktionen übergeben. Anschließend wird eine Zufallszahl p zwischen null und eins erzeugt. Ist p kleiner als Epsilon, wird eine zufällige Aktion aus den übergebenen Aktionen ausgewählt. Ansonsten wird für den Zustand die beste Aktion aus der Q-Tabelle ausgewählt und anschließend zurückgegeben.



(a) (b)

Abbildung 7.3: (a) verrauschtes Diagramm (b) lokal gemitteltes Diagramm mit einer Fenstergröße von 100.

```
# Loop for each episode
for e in range(1, self.__episodes_training + 1):
    # Initialize State S
    state, _ = self.env.reset()
    terminated = False
    # Loop for each step of episode until finish
    while not terminated:
        state = self.env._get_observation()
        valid_actions = self.env.get_actions()
        # Choose A from S using epsilon-greedy policy derived from Q
        action = self.epsilon_greedy_policy(state, valid_actions)
        # Take action A, observe R, S'=next_state
        next_state, reward, terminated, _, _ = self.env.step(action)
        # Update Q-table according to Q-Learning formular
        self.update_q_table(state, action, reward, next_state)
        # S <- S'
        state = next_state
    # repeat until S is terminal
    self.decay_epsilon()
```

Listing 7.1: Trainingsschleifen des Q-Agenten.

Trainieren:

Die Methode *train* beinhaltet die Schleifen des Q-Learning Algorithmus 7.1. Das Listing 7.1 veranschaulicht die Methode, die auf das Wesentlichste reduziert ist. Die Q-Tabelle als auch die YahtzeeEnv werden bei der Initialisierung des Agenten mit erzeugt, wie aus dem Klassendiagramm entnommen werden kann. In der YahtzeeEnv werden anschließend der Zustandsraum und der Aktionsraum definiert. Beginnt das Training, wird der Initialzustand mit der *reset*-Methode der YahtzeeEnv hergestellt und die Variable *terminated*

auf *False* gesetzt. Anschließend beginnt mit der zweiten Schleife die erste Runde des Spiels, in welcher der Agent sich zuerst den Zustandsraum (*state*) aus der YahtzeeEnv lädt. Danach werden die für diese Runde gültigen Aktionen *valid_ actions* geladen und beide an die Methode *epsilon_ greedy_ policy* übergeben. Diese gibt die Aktion zurück, die der Agent an die *step*-Methode der Environment übergibt und somit ausführt. Die *step*-Methode gibt, wie in Kapitel 6.2 beschrieben, den nächsten Zustand, die Belohnung, die Terminierungsbedingung, weitere Informationen (hier mit einem Platzhalter versehen) sowie das Belohnungsverhältnis der erreichten zur theoretischen Maximalpunktzahl zurück. Im Listing 7.1 ist das *reward_ ratio* ebenfalls mit einem Platzhalter versehen, weil es für den Q-Agent nicht benötigt wird. Danach erfolgt die Aktualisierung der Q-Tabelle und die Überschreibung des neuen Zustands zum Aktuellen. Die Überschreibung von $state = next\_state$ ist nicht notwendig, weil nach dem Ausführen der *step*-Methode der neue Zustand schon der Aktuelle in der YahtzeeEnv ist und automatisch bei dem nächsten Schleifendurchlauf zu Beginn geladen wird. Des Weiteren ist zu beachten, dass das erste *state* in Listing 7.1 in Zeile vier und das erste *next_ state* in Zeile 13 in ein Numpy Array umgewandelt werden müssen. Dies erfolgt mit der Methode *env_ state_ to_ dict_ index* und wird im Folgenden an einem Beispiel mit zufälligen Werten veranschaulicht.

```
Dict = {dice_values: [1, 5, 2, 6, 4]),
    fields_empty: [0, 1, 0, 0, 0, 1]),
                remaining_rolls: [2]}
```

Das Beispiel-Dictionary *Dict* sieht dann wie folgt aus:

```
[1 5 2 6 4 0 1 0 0 0 1 2]
```

Ist das aktuelle Spiel beendet, wird mit der Methode *decay_ epsilon* der Epsilonwert reduziert und das nächste Trainingsspiel gestartet, bis alle Episoden durchgelaufen sind.

Evaluieren:
Die Evaluierung funktioniert sehr ähnlich wie das Trainieren. Dafür wurde die Methode *evaluate* entwickelt. Der wesentliche Unterschied ist, dass sie zuerst in der Q-Tabelle nach einer optimalen Lösung schaut. Wenn es diese Situation noch nicht gab, wird eine zufällige Entscheidung getroffen. In der Praxis können neue Zustände während der Nutzung in der Q-Tabelle gespeichert werden. Dies wird in dieser Arbeit nicht gemacht, um ausschließlich den bisherigen Trainingsstand zu bewerten.

Speichern und Laden:
Mit den Methoden *load* und *safe* können alte Trainingsergebnisse geladen und Neue gespeichert werden. Dies bietet verschiedene Vorteile. Zum Einen können bereits existierende Q-Tabellen weiter trainiert werden und zum Anderen können alte Trainingsergebnisse reproduziert und erneut evaluiert werden. Die Ergebnisse werden in einer .npy-Datei gespeichert und sollten sich beim Laden im gleichen Verzeichnis wie die Datei des Agenten befinden.

Auswertungen:
Für die Beurteilung der Ergebnisse wurden verschiedene Methoden implementiert, unter anderem die *plot*-Methode. Diese ermöglicht die Ausgabe von Diagrammen im Plot Fenster der Spyder IDE. Ziel der Arbeit ist es zu zeigen, dass der Agent die Gesamtpunktzahl des Würfelspiels maximiert. Dafür werden die Gesamtpunktzahlen pro Spiel, jeweils für das Training und die Evaluation, in einem Diagramm festgehalten. Ein weiterer Indikator für den Lernerfolg des Agenten ist, wenn die Temporale Differenz gegen null konvergiert. Diese wird deshalb auch in einem Diagramm veranschaulicht. Zum Schluss soll noch überprüft werden, ob der Agent seine Chancen zur Punktmaximierung ausnutzt. Deshalb wird für das Training und die Evaluation ein Diagramm mit der Anzahl der Wiederholungswürfe pro Runde erstellt. Hinzu kommt, dass in der Konsole die durchschnittliche Gesamtpunktzahl aller Episoden und die durchschnittlichen Punkte pro Kategorie ausgegeben werden. Die gleichen Konsolenausgaben und Diagramme werden auch für den DQN-Agenten erstellt.

7.1.2 DQN-Agent

In der Abbildung 7.4 ist der Algorithmus nach DeepMind [21] dargestellt. Dort wird, wie auch in dem Buch [7] von Ivan Gridin, ein DQN-Agent mit Erfahrungsspeicher verwendet. Wie aus der Abbildung 7.4 hervorgeht, wird zuerst der Replay Buffer D mit einer Kapazität N initialisiert. Danach wird die Aktionswertefunktion Q oder auch das neuronale Netz mit zufälligen Gewichten initialisiert. Darauf folgend wird für die erste Trainingsepisode die erste Sequenz oder der erste initiale Zustand erzeugt. Anschließend wird für jeden Schritt mit der Wahrscheinlichkeit von ϵ eine zufällige Aktion a_t oder mit der Wahrscheinlichkeit von $1 - \epsilon$ die Aktion a_t mit der größten Erfolgschance ausgewählt, ausgeführt und beobachtet, welcher nächste neue Zustand von dem Environment zurückgegeben wird. Der Übergang (Minibatch) wird dann im Replay Buffer gespeichert.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Abbildung 7.4: DQN-Algorithmus aus [21].

Danach wird, sofern vorhanden, ein zufälliges Minibatch aus dem Replay Buffer geladen. In Abhängigkeit davon, ob der Terminierungszustand erreicht wurde oder nicht, wird y_j gesetzt und das Gradientenverfahren bzw. Gradientenabstiegsverfahren nach Formel 2.39 durchgeführt. Dies wird solange wiederholt, bis alle Trainingsepisoden M durchlaufen sind. Aus dem Algorithmus hat sich das Klassendiagramm 7.5 ergeben. Der DQN-Agent baut sich sehr ähnlich zum Q-Agent auf. Sie unterscheiden sich in der Agenteninitialisierung, der Aktionsauswahl und dem Training des neuronalen Netzes. Daher wird im Wesentlichen auch nur auf die Unterschiede eingegangen.

Agenteninitialisierung:
Bei der Initialisierung des Agenten werden verschiedene Parameter festgelegt, darunter Epsilon, die Größen von Zustands- und Aktionsraum, der Diskontierungsfaktor Gamma, die Batchgröße sowie die Lernrate. Zudem wird der Replay-Buffer eingerichtet und die Methode __init_q_net aufgerufen. Diese Methode ist für die Einrichtung des neuronalen Netzes zuständig und definiert sowohl den Optimierer als auch die Verlustfunktion. Als Optimierer kommt Adam zum Einsatz, der in PyTorch über *optim.Adam* verfügbar ist. Der Adam-Optimierer wird verwendet, weil er fast immer funktioniert und viele Vorteile vereint. Die Verlustfunktion basiert auf dem mittleren quadratischen Fehler (Mean Squared Error) und wird mit $nn.MSELoss()$ implementiert. Die Wahl fiel auf diese Ver-

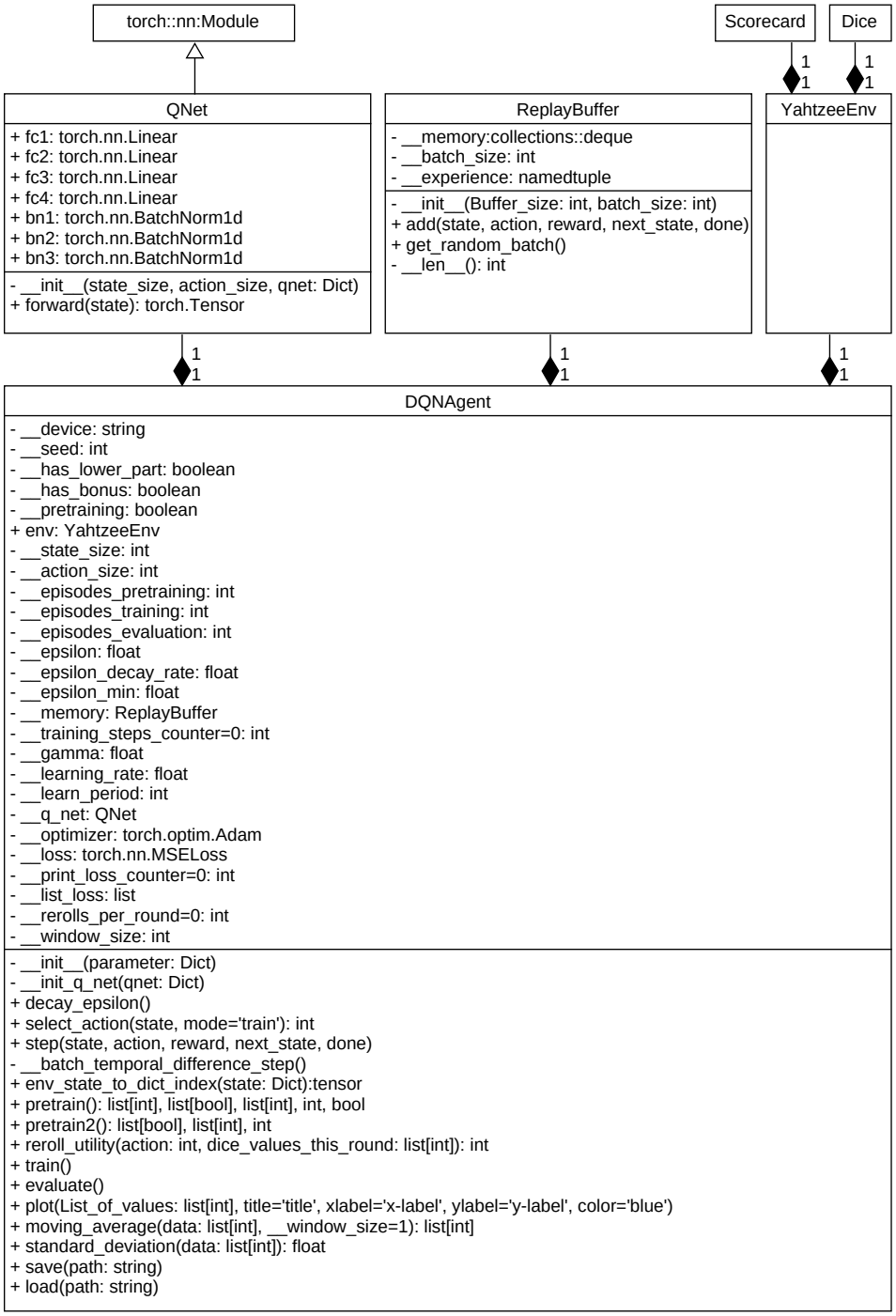


Abbildung 7.5: Klassendiagramm des DQN-Agenten.

lustfunktion, weil sie auch von Gridin in seinem Buch für die Maximierung der Belohnung verwendet wurde. Der Q-Agent konnte in einer Klasse umgesetzt werden. Für den DQN-Agent hat es sich angeboten mehrere Klassen zu erstellen. Wie im Klassendiagramm 7.5 dargestellt, gibt es für das neuronale Netz und den Replay Buffer jeweils eine eigene Klasse.

Aktionsauswahl:
Kommen wir zur Trainings- oder Evaluierungssequenz mit der Aktionsauswahl. Auch hier wird die Epsilon-Greedy-Strategie angewendet, welche in der Methode *select_ action* umgesetzt wird. Um eine Aktion im Greedy-Modus zu wählen, muss der aktuelle Zustand (state) zunächst vom Numpy-Array in ein PyTorch-Tensor umgewandelt werden, damit das neuronale Netz ihn verarbeiten kann. Danach wird das Netzwerk in den Evaluierungsmodus versetzt (self._ _q_net.eval()) und die Berechnung von Gradienten durch torch.no_grad() deaktiviert. Dies spart sowohl Speicher als auch Rechenzeit, da keine Backpropagation erforderlich ist und lediglich eine Vorhersage durchgeführt wird. Bevor jedoch die Aktion mit dem höchsten Q-Wert ausgewählt wird, muss der Aktionsraum nach der Vorhersage kontrolliert werden. Deshalb wird der Methode *select_ action* auch die gültigen Aktionen der aktuellen Runde übergeben, da das neuronale Netz immer die Q-Werte für alle Entscheidungen zurückgibt. Demzufolge sind auch solche Werte für bereits ausgewählte Kategorien dabei. Diese werden vor der Wahl des höchsten Q-Werts entfernt. Nach Abschluss der Vorhersage wird das Netzwerk wieder in den Trainingsmodus mit self._ _q_net.train() zurückgesetzt.

Trainieren:
Auch wenn das Trainieren der beiden Agenten grundlegend gleich abläuft, sind wie beim Q-Agent die beiden Trainingsschleifen im Listing 7.2 dargestellt. Beim Vergleich von Listing 7.2 und Listing 7.1 wird der Vorteil der OpenAI Gym API im Kapitel 6.1 deutlich. Beide Listings unterscheiden sich nicht. Die Aktionsauswahl (*select_ action*) sowie der Lernschritt (*step*) finden an der gleichen Stelle wie beim Listing 7.1 des Q-Agenten statt. Jedoch haben sich beim Trainieren des DQN-Agenten einige Schwierigkeiten aufgetan. Das erste Belohnungssystem hatte nicht den gewünschten Lerneffekt. Daher wurden im Laufe der Arbeit verschiedene Belohnungssysteme ausprobiert. Für die Trainingsstrategie wurde immer zuerst nur mit dem oberen Tabellenteil begonnen zu trainieren. War dies erfolgsversprechend, wurde dies auf den unteren Tabellenteil erweitert. Auf die Ergebnisse wird im Kapitel 7.2 eingegangen. Im Folgenden werden die Belohnungssysteme genauer beschrieben:

```
# Loop for each episode:
for e in range(1, self.__episodes_training + 1):
    # Initialise sequence
    state, _ = self.env.reset()
    terminated = False
    # Loop for each step of episode until finish
    while not terminated:
        state = self.env._get_observation()
        valid_actions = self.env.get_actions()
        # with propability epsilon select a random action
        action = self.select_action(state, valid_actions)
        # execute action and observe
        next_state, reward, terminated, _, reward_ratio = self.env.
    step(action)
        # store transition in replay buffer
        # sample random minibatch of transitions from replay buffer
      # set y = r
      # perform a gradient descent step
        self.step(state, action, reward, next_state, terminated,
    reward_ratio)
    self.decay_epsilon()
```

Listing 7.2: Trainingsschleifen des DQN-Agenten.

1. **Punkte pro Kategorie (PK):** Bei diesem Belohnungssystem werden die von der Umgebung zurückgegebenen Punkte als Belohnung zum Lernen verwendet. Dies bedeutet, der Agent erhält nur Belohnungen, wenn eine Kategorie ausgewählt wird und die Bedingungen für die Punkte erfüllt sind. Wiederholungswürfe werden nicht belohnt. Das NN wird ausschließlich nach der Auswahl einer Kategorie aktualisiert.

2. **Punkte pro Kategorie rückwirkend pro Runde (PKR):** Die Idee dieses Belohnungssystems ist es die Punkte, welche für die Kategorie am Ende vergeben werden, zu teilen. Somit werden auch die Wiederholungswürfe mit der gleichen Punktzahl wie auch die Auswahl der Kategorie am Ende belohnt. Dafür werden alle Züge einer Runde in einer Historie gespeichert. Dies können maximal drei sein. Für den Trainingsschritt werden die vorherigen Züge ausgelesen und bekommen die gleiche Belohnung, wie der letzte Zug. Anschließend wird das NN für jeden Zug aktualisiert.

3. **Minimum Delta (MinΔ):** Mit der Minimum Delta-Belohnung wird zu Beginn der Trainingsrunde die Belohnung auf 63 Punkte gesetzt. Dies entspricht der Mindestpunktzahl, um den Bonus zu erhalten und symbolisiert die erwartete Belohnung

für den oberen Tabellenteil. Jedes Mal wenn eine Kategorie ausgewählt wird, wird überprüft, ob die von der Umgebung zurückgegebene Belohnung drei gleiche Würfel hat. In diesem Fall würde sich die erwartete Belohnung nicht verändern. Liegt die Belohnung unterhalb der Grenze, wird die Summe der fehlenden Würfel abgezogen. Liegt sie dagegen oberhalb der Grenze, wird diese hinzuaddiert. Betrachten wir folgendes Beispiel: die erwartete Punktzahl ist die initiale Belohnung mit 63 Punkten. Nach zwei Wiederholungswürfen wird die Kategorie mit den Einsen ausgewählt. Es wurde jedoch keine eins gewürfelt. Demzufolge ist die neue erwartete Belohnung $63 - 3 = 60$. Bei Wiederholungswürfen war die Runde zuvor die Belohnung gleich der initialen Belohnung (63). Für die nächste Runde werden die Wiederholungswürfe nur noch mit 60 Punkten belohnt. Somit werden im Gegensatz zum ersten Belohnungssystem auch die Wiederholungswürfe belohnt. Die Aktualisierung der Belohnung sieht wie folgt aus:

```
# (action + 1) ist die ausgewaehlte Kategorie
current_reward += (reward-((action + 1)*3))
```

Das NN wird nach jedem Zug aktualisiert.

4. **Maximum Delta (MaxΔ):** Die Maximum Delta-Belohnung folgt einem ähnlichen Prinzip wie die Minimum Delta-Belohnung. Die von der Umgebung zurück erhaltene Belohnung für den oberen Tabellenteil muss mindestens zwei gleiche Würfel haben, sodass null zurückgegeben wird. Die Ermittlung der Gesamtpunktzahl ist abhängig von der theoretisch maximalen Punktzahl für die Kategorie. Werden weniger als zwei gleiche Würfel zurückgegeben, erhält der Agent eine negative Belohnung. Ab drei gleichen Würfeln wird der Agent belohnt. Im Folgenden ist die Aktualisierung dargestellt:

```
# (action + 1) ist die ausgewaehlte Kategorie
reward = reward - ((action+1)*2)
```

Das NN wird nur nach der Auswahl einer Kategorie angepasst.

5. **Belohnung für richtige Würfel- und Kategoriewahl (RWK):** Mit diesem Belohnungssystem wird in Anlehnung an zweite genannte Belohnungssystem versucht, rückwirkend das Auswählen der Würfel zu belohnen oder zu bestrafen. Dafür werden wieder die Züge der Runde sowie die letzte ausgewählte Aktion (die Wahl der Kategorie), die erhaltenen Punkte für die gewählte Kategorie und das

reward_ratio gespeichert. Anschließend werden zusätzliche Belohnungen vor Beginn des Trainings verteilt. Die erste zusätzliche Belohnung des DQN-Agenten ist, wenn das *reward_ratio* über die drei Züge zunimmt. Dafür werden sich für diesen Zustand die Würfelwerte angesehen und mit der letzten Aktion überprüft, welche mögliche Punktzahl (Belohnung) für die Punktekategorie nach der letzten ausgewählten Aktion erzielt worden wäre. Die Klasse Scorecard wurde dafür um die Methode *get_possible_score* erweitert, die die mögliche Belohnung *reward* und das Gewinnverhältnis *reward_ratio* zurückgibt. Die Methode *get_possible_score* ist die gleiche, wie die Methode *set_score* in der Klasse Scorecard mit dem Unterschied, dass nur bei dieser keine Vergabe der Punkte oder andere Aktualisierungen in der Klasse stattfinden. Dadurch ist es möglich für die jeweilige Runde einen Vergleich der beiden *reward_ratios* durchzuführen und eine zusätzliche Belohnung zu vergeben. Nimmt das *reward_ratio* zu, bekommt der DQN-Agent einen zusätzlichen Punkt. Nimmt dieses ab, werden diesem zwei Punkte abgezogen. Des Weiteren wird der DQN-Agent zusätzlich belohnt, wenn er in diesen drei Zügen seine Wiederholungswürfe nutzt. Dafür wird bei der Auslesung der Rundenhistorie geprüft, ob die Aktion ein Wiederholungswurf war. Dafür bekommt der DQN-Agent pauschal 10 Punkte. Die pauschale Vergabe der Punkte wurde mit verschiedenen Werten getestet und durfte nicht zu groß oder zu klein sein. 10 Punkte haben sich als guter Wert durch Probieren ergeben. Nach dieser pauschalen Vergabe wird überprüft, ob die Auswahl der Würfel zielführend zur Punktemaximierung waren. Dafür wird der DQN-Agent mit einem Punkt belohnt, wenn er die richtigen Würfel zum Behalten ausgewählt hat und mit einem Punkt bestraft, wenn er einen falschen Würfel, der nicht für die Kategorie benötigt wird, behalten hat. Auch wird der DQN-Agent dafür bestraft, wenn er einen Würfel vergessen hat zu behalten, der für diese Kategorie gewinnbringend gewesen wäre. Belohnung und Bestrafung bewegen sich hierfür jeweils bei einem Punkt. Nach der Vergabe aller zusätzlichen Belohnungen wird die Trainingsmethode *step* für diesen Zug ausgeführt.

6. **Normalisiertes RWK (NRWK):** Dieses Belohnungssystem ist fast identisch zum RWK, weil das gleiche Prinzip der Anpassung für die Belohnung angewendet wird. Es wurde jedoch eine Normalisierung der Punkte integriert. Dafür wird die Belohnung mit dem *reward_ratio* überschrieben. Die Idee der Normalisierung ist die Vergabe der Punkte zu vereinheitlichen und unabhängig von den Würfelwerten zu machen. Zum Beispiel bekommt der DQN-Agent den Wurf [1, 1, 1, 1, 6]. Sind beide Kategorien noch frei, würde er mit sehr hoher Wahrscheinlichkeit auf

die Sechs gehen, anstatt auf die Einsen, weil die Sechs mehr Punkte bringt. Jeder menschliche Spieler würde auf die Kategorie der Einsen gehen, weil schon vier von fünf Würfel gleich sind. Das entspricht in diesem Fall einem *reward_ratio* von $0,8$, bei der Sechs nur $0,2$. Auch die Vergabe der zusätzlichen Belohnungen wurde dementsprechend angepasst. Für die Zunahme des *reward_ratios*, wird der DQN-Agent um $0,1$ Punkte belohnt oder bei Verkleinerung des *reward_ratio* um $0,1$ Punkte bestraft. Die pauschale Belohnung für einen Wiederholungswurf beträgt $0,5$ Punkte. Die Belohnung und Bestrafung für das richtige Würfelauswählen bewegt sich um $0,1$ Punkte. Die Aktualisierung des NN erfolgt für das RWK und NRWK für jeden Zug nachdem die Belohnungen angepasst wurden.

7. **Die Reroll-Utility-Methode (R-U-M):** Alle vorhergehenden Belohnungssysteme haben grundlegend ein positives Lernverhalten gezeigt. Jedoch waren die Ergebnisse bescheiden, sodass die *reroll_utility*-Methode eingeführt wurde. Mit ihr wurde sich an der Arbeit von Philip Vasseur [35] orientiert. Dieser verwendet eine Methode, die vom DQN-Agenten nur die Kategorie erhält, auf die dieser gehen möchte. Anschließend wählt er die Würfel und führt die Wiederholungswürfe durch. Der gleiche Ansatz wurde mit der *reroll_utility*-Methode verfolgt. Dafür wurde der Aktionsraum, aus dem der DQN-Agent wählen kann, verkleinert. Dieser kann nur noch Aktionen für die Auswahl der Kategorien auswählen. Anschließend wird die Auswahl der *reroll_utility*-Methode übergeben. Besitzt der DQN-Agent für diese Runde noch Wiederholungswürfe, wählt die Methode die neu zu würfelnden Würfel und übergibt dem Environment anschließend die Aktion für einen Wiederholungswurf. Besitzt der DQN-Agent für diese Runde keine Wiederholungswürfe mehr, wird die ausgewählte Kategorie an das Environment übergeben und ausgewählt. Sollte die ausgewählte Kategorie schon optimal sein, z. B. bei einer großen Straße, werden alle Würfel behalten und die Aktion für die Auswahl der Kategorie direkt dem Environment ohne Nutzung von Wiederholungswürfen übergeben. Der DQN-Agent wählt nach jedem Zug die Kategorie erneut. Die Einführung der Methode bietet den Vorteil, dass sich der Aktionsraum des DQN-Agent verkleinert, weil die Methode die Auswahl der Würfel übernimmt, die gewürfelt werden sollen. Dafür muss bei der Initialisierung des Aktionsraums darauf geachtet werden, dass die Aktionen für die Wiederholungswürfe exkludiert sind. Gleichzeitig muss das Environment diese weiterhin ausführen, wenn sie ihm übergeben werden. Das NN wird nach jeder Kategorieauswahl trainiert.

Neuronales Netzwerk:
Das neuronale Netzwerk, welches durch die QNet-Klasse repräsentiert wird, besteht aus mehreren vollständig verbundenen Schichten (Linear Layers), die in PyTorch mit $nn.Linear$ umgesetzt werden. Jede dieser Schichten benötigt zwei Parameter: die Anzahl der Eingabewerte und die Anzahl der Ausgabewerte. Konkret kann das Netzwerk bis zu drei verdeckte Schichten umfassen, benannt als fc1 bis fc3. Die erste Schicht (fc1) transformiert beispielsweise Eingabedaten mit der Dimension state_size (also dem Zustandsraum) in eine verdeckte Repräsentation mit fc1_units Neuronen. Zwischen den einzelnen Schichten wird eine Batch-Normalisierung eingesetzt, um die Verteilung der Eingabedaten zu stabilisieren. Das verbessert die Trainingsgeschwindigkeit und -stabilität. In PyTorch wird dies mit $nn.BatchNorm1d$ umgesetzt, das speziell für eindimensionale Daten, wie sie typischerweise in vollständig verbundenen Schichten vorkommen, geeignet ist . Als Aktivierungsfunktion kommt Leaky ReLU zum Einsatz, diese wird über $nn.functional.leaky\_relu$ bereitgestellt. Diese Funktion erlaubt einen kleinen Gradientenfluss auch für negative Eingabewerte, was das Problem toter Neuronen vermeidet. Die Gewichte des Netzwerks werden mit der He-Initialisierung zufällig gesetzt. Diese Methode ist speziell für ReLU-basierte Aktivierungsfunktionen ausgelegt und sorgt dafür, dass die Gewichtswerte gut verteilt sind, um stabile Trainingsverläufe zu ermöglichen. In PyTorch kann die He-Initialisierung mit $nn.init.kaiming\_uniform\_$ verwendet werden. Der Vorwärtsdurchlauf des Netzwerks wird durch die Forward-Methode definiert. Hier wird der Eingabezustand (state) nacheinander durch die linearen Schichten, die Batch-Normalisierung sowie die Aktivierungsfunktionen geleitet. Die Tiefe des neuronalen Netzes kann eingestellt werden. Demzufolge können zwischen einer bis drei verdeckten Schichten genutzt werden. Die Einstellung erfolgt über der Anzahl der Neuronen, die im Parameter-Dictionary übergeben werden, wie aus dem Listing 7.4 entnommen werden kann. Steht dort für die Schicht fc3 beispielsweise ein Wert (64) wird diese initialisiert. Steht in der Schicht fc3 *None* wird diese Schicht nicht initialisiert und das neuronale Netz besitzt nur zwei verdeckte Schichten. Hinzu kommt, dass die Ausgabe des neuronalen Netzes mit einer Softmax-Funktion normalisiert wird. Diese wird durch $nn.Softmax$ von Pytorch bereitgestellt.

Replay Buffer:
Die Klasse ReplayBuffer bildet das Replay-Memory ab, das im Deep Q-Learning eingesetzt wird. Dieses speichert vergangene Erfahrungen, um die Korrelation zwischen aufeinanderfolgenden Trainingsdaten zu verringern und somit die Stabilität des Lernprozesses zu verbessern. Das Replay-Memory funktioniert nach dem FIFO-Prinzip (First In, First

Out), wodurch ein effizienter Einfüge- und Löschmechanismus ermöglicht wird. Die maximale Speichergröße kann definiert werden. Ist diese erreicht, wird beim Hinzufügen eines neuen Elements automatisch das Älteste entfernt. So wird ein Speicherüberlauf verhindert und gewährleistet, dass stets nur die aktuellsten Erfahrungen erhalten bleiben. Ist der Speicher einmal voll, müssen die neuen Erfahrungen die Bedingung erfüllen, dass ihr *reward_ratio* größer oder gleich $0,6$ sein muss. Dies soll dafür sorgen, dass nur noch interessante Erfahrungen gespeichert werden. Des Weiteren werden Erfahrungen mithilfe der Methode *add* in Form eines benannten Tupels strukturiert im Puffer abgelegt. Für das Training des neuronalen Netzes können mit der Methode *get_random_batch* zufällige Mini-Batches aus dem Replay-Buffer entnommen werden.

main:
Das Ausführen der Programme gestaltet sich bei beiden Agenten gleich. In der jeweiligen .py-Datei des Agenten befindet sich ein *__main__*-Abschnitt. Dort wird als erstes das Dictionary *Parameter* mit allen notwendigen Parametern initialisiert. In diesem Dictionary können alle notwendigen Einstellungen vorgenommen werden, die das Training des Agenten beeinflussen. Anschließend muss in dem *__main__*-Abschnitt der Agent initialisiert und das Dictionary übergeben werden. Ist das Objekt instanziiert, kann auf die Methoden der Klassen QAgent oder DQNAgent zugegriffen werden. Mit der Methode *load("Name.npy")* können gespeicherte Trainingsdaten geladen werden. Anschließend besteht die Möglichkeit von diesem Trainingsstand aus mit der Methode *train* weiter zu trainieren. Sollen die Trainingsergebnisse nach dem Training gespeichert werden, steht dafür die Methode *save("Name.npy")* zur Verfügung. Mit der Methode *evaluate* kann der Agent nach dem Training evaluiert werden. Dafür muss der Agent nicht trainiert werden, wenn bereits ein gespeichertes Modell zuvor geladen wurde. Zum Schluss ist es ratsam die Methode *close* aufzurufen, um alle benutzten Ressourcen wieder freizugeben. Anschließend können die Trainingsparameter über das Dictionary nach Belieben angepasst und das unterschiedliche Trainingsverhalten beobachtet werden. Im Gegensatz zum Q-Agent besitzt der DQN-Agent noch zwei weitere Klassen. Dort können weitere Einstellungen vorgenommen werden, die das Training des DQN-Agent beeinflussen können. Bei der ReplayBuffer-Klasse kann zum Beispiel beeinflusst werden, wann und welche Beispiele für das Training gespeichert werden sollen wohingegen bei der QNet-Klasse die gesamte Architektur des neuronalen Netzes verändert und angepasst werden kann. Hierzu zählen unter Anderem die Anzahl der versteckten Schichten und die Aktivierungsfunktionen für die jeweilige versteckte oder Ausgabeschicht.

7.2 Training der Agenten

Nachdem der Aufbau der Agenten beschrieben wurde, soll auf das Training eingegangen werden. Dabei wird das Trainingsverhalten der unterschiedlichen Belohnungssysteme des DQN-Agent beleuchtet. Der Q-Agent hat keine weiteren Anpassungen erfahren. Hier wurde sich auf die Argumentation von Kang und Schroeder [14] gestützt, dass der Zustandsaktionsraum zu speicherintensiv ist, um jede Möglichkeit in der Q-Tabelle abbilden zu können. Es wurden für beide Agenten die gleiche Vortrainingsfunktionen entwickelt, auf die zuerst eingegangen werden soll.

7.2.1 Vortrainingsfunktionen

In einer frühen Phase der Arbeit wurde versucht beide Agenten mit speziellen Funktionen vorzutrainieren, sodass die Q-Tabelle oder das neuronale Netz schneller gute Zustände erreicht an denen es lernen kann und nicht auf den Zufall angewiesen ist. Dafür wurden zwei unterschiedliche Funktionen (*pretrain* und *pretrain2*) implementiert. Die *pretrain*-Funktion geht verschiedene Würfelzustände und Feldkombinationen durch, indem sie die bereits belegten Felder, die Anzahl der Wiederholungswürfe und Würfelkombinationen manipuliert. Da diese Manipulation auch das Environment betrifft, müssen neben den Würfelwerten, der Anzahl der Wiederholungswürfe und den belegten Feldern auch die verbleibenden gültigen Aktionen für diesen Zug zurückgegeben werden. Der Vorteil dieser Methode ist, dass die Agenten viele verschiedene Zustände sehen, z. B. die Würfelanordnung [1, 1, 1, 1, 1], bei welcher nur die Kategorie der Einsen oder ein Yahtzee auswählbar ist. Jedoch werden sehr viele andere weniger interessante Zustände durchlaufen, was das Vortraining sehr lang macht, weil die Anzahl der Trainingsepisoden statisch ist. Die Trainingsepisoden für den oberen Tabellenteil ergibt sich aus der Multiplikation von der Anzahl der Wiederholungswürfe (3), der möglichen Würfelpunkte eines Würfels, exponiert mit der Anzahl der Schleifen, die die Würfelwerte manipulieren (6^3) und den Möglichkeiten der Kombinationen der auswählbaren Felder minus dem Fall, dass alle Felder belegt sind ($2^6 - 1$). Dies ergibt insgesamt 40.824 Trainingsepisoden. Analog ergeben sich für den unteren Tabellenteil 191.079.648 Trainingsepisoden. Da dies die Trainingszeit sehr stark erhöhte, jedoch keinen Mehrwert für das Training brachte, wurde diese Methode verworfen. Die zweite Methode *pretrain2* funktioniert folgendermaßen. Sie gibt nach und nach die belegten Felder der Punktetabelle frei. Das heißt dem Agenten steht am Anfang nur ein freies Feld zur Verfügung, das er auswählen

kann. Im nächsten Schritt stehen ihm dann zwei Felder zur Verfügung. Die Methode manipuliert nicht die Würfelwerte und auch nicht die Anzahl der Wiederholungswürfe. Jedoch müssen auch hier die gültigen Aktionen zurückgegeben werden, damit nicht ein falsches Feld, das bereits belegt ist, ausgewählt werden kann. Ein Unterschied zur *pretrain*-Funktion ist, dass die Anzahl der Trainingsepisoden im Parametersatz unter *episodes_pre_training* eingestellt werden kann. Dies beeinflusst, wie oft eine Feldkombination durchlaufen werden soll. Gemeinsam haben beide Funktionen, dass sie als Funktionsgenerator implementiert sind und nach jeder Rückgabe der Werte stoppen. In der Methode *train()* können sie dann wieder getriggert werden, um eine Iteration weiterzugehen. Beide Vortrainingsfunktionen haben sich als nicht zielführend herausgestellt, weshalb diese Lösungsstrategie verworfen wurde. Sie sind dennoch weiterhin im Code implementiert, um die Ergebnisse reproduzierbar zu halten. Die Abbildung 7.6 zeigt jeweils die Ergebnisse der *pretrain*- und *pretrain2*-Funktion für den Q-Agent und den DQN-Agent. Zur Erinnerung durchlaufen beide Agenten die Vortrainingsschleife 40.824 Mal für die Methode*pretrain*. Die Anzahl der Wiederholungen für die *pretrain2* kann in den Parametern unter *episodes_pre_training* eingestellt werden. Für die in der Abbildung 7.6 dargestellten Diagramme wurde $episodes\_pre\_training = 100$ eingestellt.

7.2.2 Training des Q-Agenten

Beim Q-Agenten ist das Training wesentlich einfacher als beim DQN-Agent, da es weniger Parameter mit einem Einfluss auf das Lernverhalten als beim DQN-Agent gibt. Das Listing 7.3 veranschaulicht das Dictionary mit möglichen Parametereinstellungen. An diesen Stellschrauben wird für die Evaluierung des Q-Agenten gestellt und Anpassungen vorgenommen. Die Einstellung des *seed* sorgt dafür, dass immer die gleiche Sequenz an Zufallszahlen erzeugt wird. Mit den Parametern *has_lower_part* und *has_bonus* kann die Komplexität eingestellt und erweitert werden, wenn diese auf *True* gesetzt werden. Abgesehen von der *window_size*, die die Fenstergröße für eine Verringerung des Rauschens in den Diagrammen einstellt, beeinflussen alle anderen Parameter direkt das Training. Die Belohnung, die der Q-Agent erhält, sind die erreichten Punkte für die jeweils ausgewählte Kategorie. Das heißt es gibt keine zusätzlichen Belohnungen oder Funktionen, die darauf abzielen die Chancen des Q-Agenten während des Lernens zu erhöhen. Für das Training des Q-Agenten wurden der Einfluss der Lernschrittweite (α), der Diskontierungsfaktor (γ) und die Anzahl der Trainingsepisoden genauer untersucht. Die in

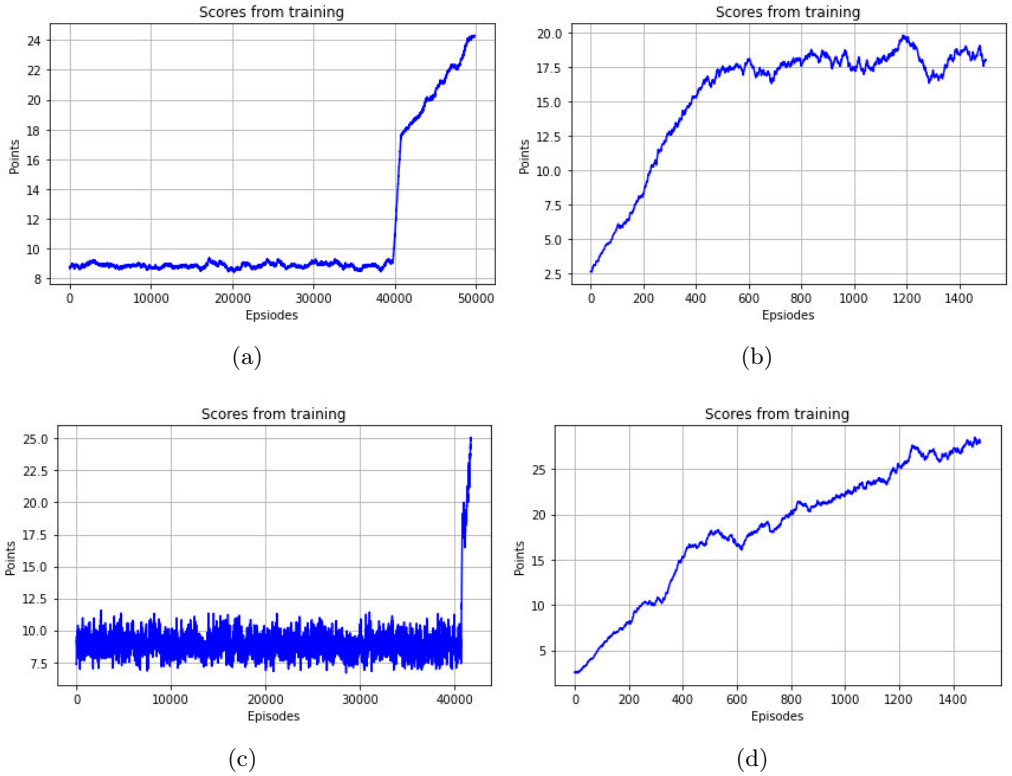


Abbildung 7.6: Ergebnisse der Vortrainingsfunktionen (a) Verlauf der Punktzahl vom Training des Q-Agenten mit der pretrain-Methode, (b) Verlauf der Punktzahl vom Training des Q-Agenten mit der pretrain2-Methode, (c) Verlauf der Punktzahl vom Training des DQN-Agenten mit der pretrain-Methode und (d) Verlauf der Punktzahl vom Training des DQN-Agenten mit der pretrain2-Methode.

den folgenden Untersuchungen dargestellten Ergebnisse sind eine Zusammenfassung und sollen einen Überblick über die Auswirkung der Veränderung der Parameter geben. Es werden die durchschnittlichen Punkte des Trainings sowie der Evaluation mit der Standardabweichung dargestellt.

Lernschrittweitentest:

Für die Untersuchung der Lernschrittweite wurden die Parameter wie im Listing 7.3 eingestellt und α von $0,9$ schrittweise um $0,1$ verkleinert.

```
parameter = {
    'seed': 3,
    'has_lower_part': False,
    'has_bonus': False,
    'pre_training': False,
    'episodes_pre_training': 10,
    'episodes_training': 100000,
    'episodes_evaluation': 2000,
    'epsilon': 1,
    'epsilon_decay_rate': 0.00001,
    'epsilon_min': 0.1,
    'alpha': 0.7,
    'gamma': 0.99,
    'window_size': 1000
    }
```

Listing 7.3: Parameter Q-Agent.

α	**Training**	**Evaluation**
$0,9$	$26,5 \pm 10,67$	$38,5 \pm 7,73$
$0,8$	$27,2 \pm 10,81$	$39,8 \pm 7,46$
$0,7$	$27,5 \pm 10,67$	$40,3 \pm 7,49$
$0,6$	$27,5 \pm 10,41$	$39,2 \pm 7,25$
$0,5$	$27,4 \pm 10,17$	$39,0 \pm 6,82$
$0,4$	$27,1 \pm 9,75$	$37,0 \pm 6,60$
$0,3$	$26,9 \pm 9,40$	$35,7 \pm 6,51$
$0,2$	$26,4 \pm 8,83$	$33,2 \pm 6,19$
$0,1$	$25,5 \pm 8,24$	$30,6 \pm 6,25$

Tabelle 7.1: Die durchschnittliche Gesamtpunktzahl des Lernschrittweitentests.

Fazit:
Aus der Tabelle 7.1 wird ersichtlich, dass eine zu klein gewählte Lernschrittweite das Lernverhalten verschlechtert und die maximale Punktzahl abnimmt. Für die weiteren Untersuchungen wurde die Lernschrittweite von $0,7$ beibehalten, weil die Gesamtpunktzahl beim Training größer und die Standardabweichung kleiner war als im Versuch mit der Lernschrittweite $0,8$. Die Tabelle 7.1 veranschaulicht sehr schön nach Gridin [7] oder Sutton [34], dass die Lernschrittweite nicht zu groß oder zu klein sein darf. Der Q-Agent lernt bei einer zu kleinen Lernschrittweite nicht schnell genug. Bei einer zu großen Lernschrittweite beeinflussen dagegen die schwankenden Punkte (Belohnungen) die Werte der Q-Tabelle stärker.

γ	**Training**	**Evaluation**
$0,99$	$27,3 \pm 10,65$	$40,0 \pm 7,61$
$0,9$	$27,8 \pm 10,18$	$38,0 \pm 6,71$
$0,8$	$27,3 \pm 9,42$	$35,2 \pm 6,29$
$0,7$	$26,9 \pm 8,93$	$33,0 \pm 6,34$
$0,6$	$26,7 \pm 8,64$	$32,5 \pm 6,03$
$0,5$	$26,4 \pm 8,45$	$31,7 \pm 6,12$
$0,4$	$26,2 \pm 8,33$	$31,5 \pm 6,10$
$0,3$	$26,1 \pm 8,29$	$31,3 \pm 6,00$
$0,2$	$26,1 \pm 8,27$	$31,1 \pm 6,05$
$0,1$	$26,0 \pm 8,22$	$31,1 \pm 5,94$

Tabelle 7.2: Die durchschnittliche Gesamtpunktzahl nach dem Diskontierungsfaktortests.

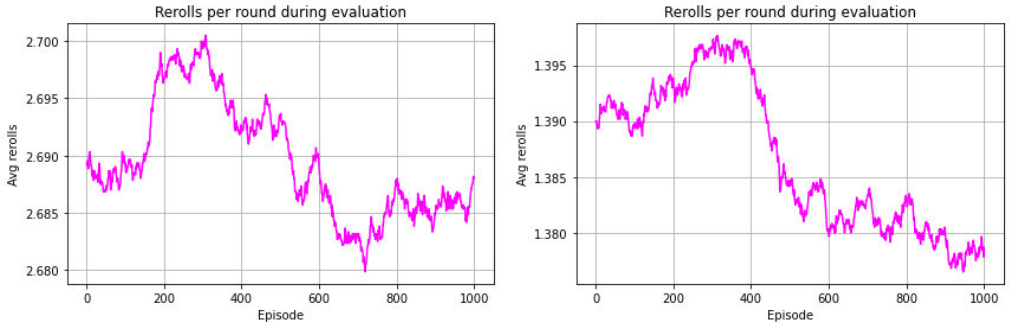
Diskontierungsfaktortest:

Für die Untersuchung des Diskontierungsfaktors wurden die Parameter wie im Listing 7.3 eingestellt und γ von $0,99$ auf $0,9$ und weiter schrittweise um $0,1$ verkleinert.

Fazit:
Aus der Tabelle 7.2 für den Diskontierungsfaktortest wird deutlich, dass die maximale Punktzahl abnimmt sofern dieser zu klein gewählt ist. Dies liegt daran, dass der Q-Agent nur auf kurzfristige Belohnungen aus ist. Die Abbildungen 7.7 (a) und (b) bestätigen dies, weil die Anzahl der Wiederholungswürfe während der Evaluation vom Q-Agenten mit kleinerem Diskontierungsfaktor kleiner ist. Für alle weiteren Tests und Untersuchungen wurde ein γ von $0,99$ beibehalten.

Veränderung der Episodenanzahl:

Die Anzahl der Episoden wurde für die Evaluierung immer auf 2.000 Episoden gelassen, um ein statistisch verwertbares Ergebnis zu bekommen. Die Anzahl der Episoden für die Trainings des Q-Agent haben einen signifikanten Einfluss auf die Qualität der Q-Tabelle. Dafür wurde mit 10.000 Episoden begonnen und diese zweimal um den Faktor 10 erweitert. Die *epsilon_ decay_ rate* wurde dementsprechend angepasst, sodass sie für 10.000 Episoden $0,0001$, für 100.000 Episoden $0,00001$ und für 1.000.000 Episoden $0,000001$ betrug. Demzufolge befindet sich Epsilon in den letzten 10% der Trainingszeit auf dem Minimalwert von $0,1$. Es ist grundsätzlich zu empfehlen die Verzögerungsrate



(a) (b)

Abbildung 7.7: Wiederholungswürfe der Evaluation mit unterschiedlichen Diskontierungsfaktoren (a) 0,99 und (b) 0,1.

Episoden	**Training**	**Evaluation**
10.000	$22,0 \pm 8,28$	$27,8 \pm 6,94$
100.000	$27,5 \pm 10,66$	$39,6 \pm 7,60$
1.000.000	$29,6 \pm 12,23$	$45,1 \pm 7,91$

Tabelle 7.3: Die durchschnittliche Gesamtpunktzahl nach Zunahme der Trainingsepisoden.

von Epsilon so klein wie möglich zu wählen, damit der Agent so viel wie möglich ausprobieren kann. Schauen wir uns zuerst die Ergebnisse der drei Durchläufe an und wie sich die Verlängerung der Trainingszeit auf die Gesamtpunktzahl auswirkt. Wie aus der Tabelle 7.3 hervorgeht, nimmt sowohl beim Training als auch bei der Evaluation die durchschnittliche Gesamtpunktzahl zu. Jedoch ist der Punkteanstieg nicht linear zum Anstieg der Episoden. Gleiches gilt auch für die Tabelle 7.4, in welcher der Anstieg der Punkte pro Kategorie von 100.000 Episoden auf 1.000.000 Episoden nicht mehr so stark ausfällt wie bei der Zunahme von 10.000 Episoden auf 100.000 Episoden. In der Abbildung 7.8 ist die Temporale Difference veranschaulicht, die im Idealfall gegen null konvergiert. Die Diagramme veranschaulichen sehr gut, dass mit der Steigerung der Episodenzahl die Temporale Difference immer besser konvergiert. Das wiederum bedeutet, dass der Agent immer besser wird, was sich auch in der Zunahme der durchschnittlichen Gesamtpunktzahl widerspiegelt. Dass die Temporale Difference am Anfang konvergiert, liegt an dem gewählten Diskontfaktor von $0,99$. Aus der Abbildung 7.9 (a) entnommen werden kann, konvergiert die Temporale Difference bei einem Diskontierungsfaktor von $0,1$ sofort. Des Weiteren kann aus der Abbildung 7.9 entnommen werden, dass ab einem Diskontierungsfaktor von $0,4$ eine Divergenz zu Beginn des Trainings erkennbar ist. Dies

Episoden	Einsen	Zweien	Dreien	Vieren	Fünfen	Sechsen
10.000 Training	0,85	1,79	2,88	4,09	5,48	6,90
2.000 Evaluation	0,92	1,98	3,36	5,23	7,26	9,00
100.000 Training	1,02	2,27	3,67	5,27	6,83	8,42
2.000 Evaluation	1,44	3,32	5,42	7,68	9,97	11,81
1.000.000 Training	1,16	2,65	4,21	5,69	7,20	8,66
2.000 Evaluation	1,50	3,92	6,36	8,64	11,18	13,45

Tabelle 7.4: Punkte pro Kategorie in Abhängigkeit der Trainingsepisoden.

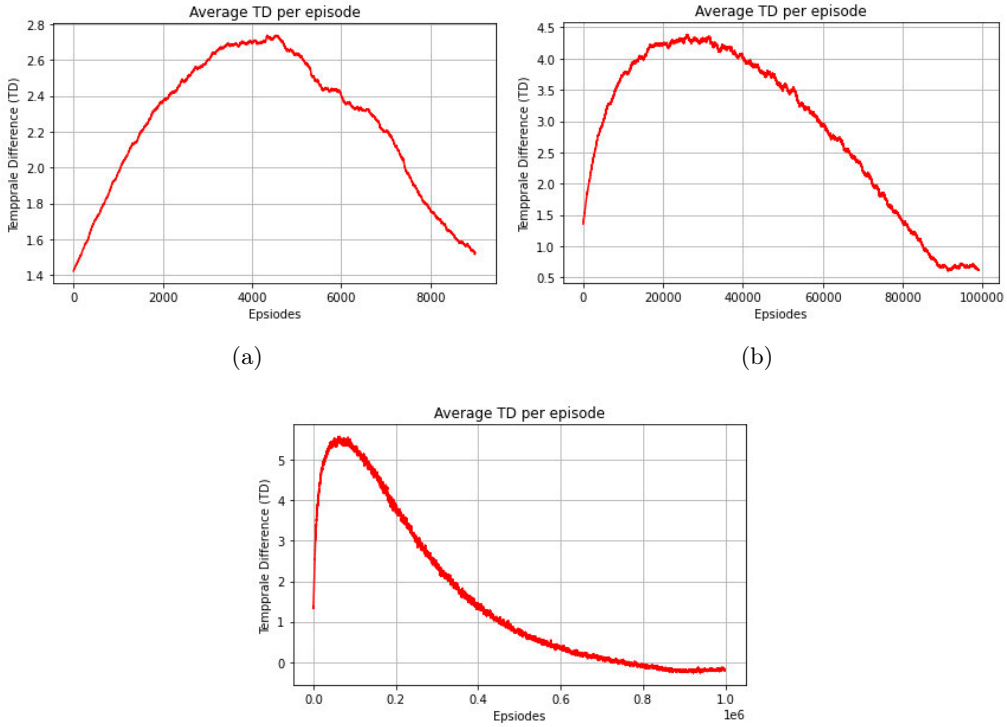
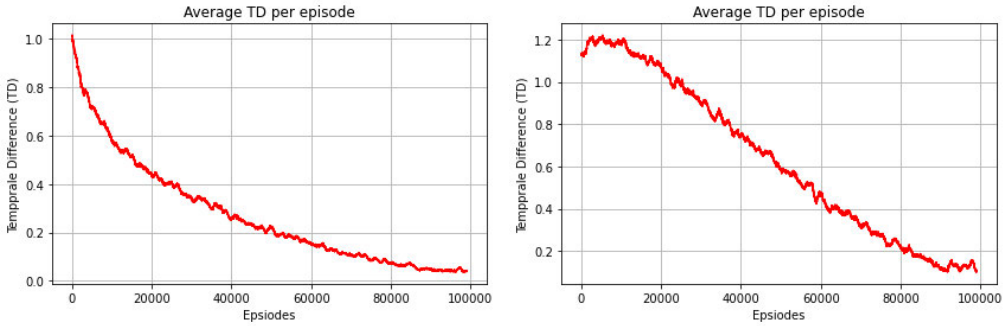


Abbildung 7.8: Temporale Difference von (a) 10.000 Episoden, (b) 100.000 Episoden und (c) 1.000.000 Episoden mit den Parametern aus Listing 7.3.

liegt daran, dass die erzielten Punkte (Belohnungen) sehr schwanken können und der große Diskontierungsfaktor diese entsprechend nach der Formel 2.34 gewichtet.

Nach den zuvor gewonnenen Erkenntnissen wurde der Q-Agent einmal mit 1.000.000 Episoden für das komplette Spiel trainiert. Werden die Ergebnisse der durchschnittli-



(a) (b)

Abbildung 7.9: Temporale Difference unterschiedlicher Diskontierungsfaktoren (a) 0,1 und (b) 0,4.

chen Gesamtpunktzahl des Trainings in Höhe von $74,1$ mit einer Standardabweichung von $\pm 32,66$ Punkte mit der erreichten Punktzahl aus der Evaluation in Höhe von $117,7$ mit einer Standardabweichung von $\pm 28,86$ Punkte verglichen, ist auch hier ein Lerneffekt erkennbar. Jedoch erreicht der Q-Agent gerade einmal das Niveau des Greedy Level-1 Agenten mit einer Punktzahl von $112,541$ von Kang und Schroeder [14]. Dies kann als eine Bestätigung der Annahme interpretiert werden, warum der Q-Learning Algorithmus nicht in der Arbeit der Beiden berücksichtigt wurde. Die Q-Tabelle benötigt ca. $2,7\,\text{GB}$ Speicherplatz, wohingegen die Trainings des oberen Tabellenteils nicht mehr als ca. $32\,\text{MB}$ brauchen.

7.2.3 Training des DQN-Agenten

Zu Beginn der Arbeit wurde angenommen, dass mehr als 1.024 Neuronen für die Bewältigung der Aufgabe benötigt werden. Dafür wurde zusätzlich das Cuda-Toolkit von Nvidia installiert, um die Berechnungen auf einer Grafikkarte vom gleichnamigen Hersteller machen zu können. Dies verkürzt die Trainingszeiten des DQN-Agenten erheblich. Es stellte sich jedoch heraus, dass 64 Neuronen ausreichend waren. Verbesserungen wurden durch die Anpassungen des Belohnungssystems erzielt. Das Listing 7.4 zeigt die Standardparameter des DQN-Agenten. Diese wurden durch Probieren ermittelt und waren der Ausgangspunkt für alle weiteren Untersuchungen. Der Aufbau der Parameter ist sehr ähnlich zu denen des Q-Agenten. Hinzu kommen noch die Parameter, die für den Replay Buffer sowie für das neuronale Netz eingestellt werden können. Dies erschwert das Finden von geeigneten Parametern. Die Werte im Listing 7.4 haben sich als guter Ausgangspunkt

```
parameter = {
    'seed': 3,
    'has_lower_part': False,
    'has_bonus': False,
    'pre_training': False,
    'episodes_pre_training': 10,
    'episodes_training': 1000,
    'episodes_evaluation': 2000,
    'epsilon': 1,
    'epsilon_decay_rate': 0.001,
    'epsilon_min': 0.1,
    'replay_buffer_size': 900,
    'batch_size': 100,
    'gamma': 0.99,
    'learning_rate': 0.0001,
    'learn_period': 1,
    'fc1': 64,
    'fc2': 64,
    'fc3': None,
    'window_size': 100
}
```

Listing 7.4: Parameter DQN-Agent.

für das Training erwiesen, nachdem mit vielen verschiedenen, manuellen Einstellungen getestet wurde. Des Weiteren wurde auch beim DQN-Agenten immer zunächst nur mit dem oberen Tabellenteil trainiert und die Komplexität bei erfolgreichen Ergebnissen erhöht. Im Gegensatz zum Q-Agenten werden beim DQN-Agenten zuerst die verschiedenen Belohnungssysteme mit den im Listing 7.4 angegebenen Parametern verglichen. Anschließend werden analog zum Q-Agent bei Feststellen eines akzeptablen Belohnungssystems verschiedene Parameteranpassungen vorgenommen und näher beleuchtet. Die Tabelle 7.5 fasst die Belohnungssysteme noch einmal zusammen.

1. PK:

Beim Trainieren des DQN-Agenten wurde in einem ersten rudimentären Ansatz das gleiche Prinzip wie beim Q-Agenten verfolgt. Der DQN-Agent lernt nach jeder ausgeführten Aktion. Es konnte keine Konvergenz der Verlustfunktion erzeugt werden. Des Weiteren hat der DQN-Agent mit dieser Strategie seine Chancen mit den Wiederholungswürfen nicht optimal ausgenutzt. Insgesamt konnte ein positives Lernverhalten festgestellt werden, weil die Gesamtpunktzahl über die Trainingszeit von durchschnittlich weniger als 18 Punkten auf durchschnittlich mehr als 27 Punkte zugenommen hat. Die Abbildung 7.10

Belohnungssystem	**Abkürzung**
1. Punkte pro Kategorie	PK
2. Punkte pro Kategorie rückwirkend pro Runde	PKR
3. Minimum Delta	MinΔ
4. Maximum Delta	MaxΔ
5. Richtige Würfel- und Kategoriewahl	RWK
6. Normalisierte richtige Würfel- und Kategoriewahl	NRWK
7. Reroll-Utility-Methode	R-U-M

Tabelle 7.5: Zusammenfassung der Belohnungssysteme.

(b) zeigt den Verlauf der Punktzahl über die Trainingszeit. In der Evaluation konnten im Schnitt $27,2$ Punkte erzielt werden. Aus den in Abbildungen 7.10 (c) und (d) dargestellten Ergebnissen wurde geschlussfolgert, dass dem DQN-Agenten ein Anreiz gegeben werden muss die Wiederholungswürfe besser auszunutzen, um so seine Chancen zu erhöhen. Die Divergenz in Abbildung 7.10 (a) ist dadurch zu begründen, dass das Erhalten der Punktzahl (Belohnung) sehr sprunghaft für den DQN-Agenten ist. Zuerst wurde sich im Folgenden weiter auf die Maximierung der Gesamtpunktzahl fokussiert.

2. PKR:
Eine Überlegung für eine bessere Nutzung der Wiederholungswürfe durch den DQN-Agenten bestand darin die erzielten Punkte pro Kategorie rückwirkend zu teilen. Dafür werden alle Züge in einer Rundenhistorie zwischengespeichert. Wenn der DQN-Agent eine Kategorie auswählt, werden die erzielten Punkte als Belohnung für die Wiederholungswürfe gegeben. Auch mit diesem Belohnungssystem hat sich ein positives Lernverhalten eingestellt. Jedoch hat die Anzahl der Wiederholungswürfe nur leicht zugenommen, wie aus der Abbildung 7.11 ersichtlich wird. Der Verlauf der anderen Graphen verhielt sich sehr ähnlich, wie zu PK.

3. MinΔ und 4. MaxΔ:
Die Ergebnisse von MinΔ und MaxΔ verhalten sich sehr ähnlich zueinander. In der Abbildung 7.12 sind die Verläufe von MinΔ dargestellt. Für beide Δ-Belohnungssysteme hat die Verlustfunktion zum ersten Mal ein Konvergenzverhalten gezeigt (siehe Abbildung 7.12 (a)). Das Konvergenzverhalten kann dadurch begründet werden, dass die Punkte und somit die Belohnungen für den DQN-Agenten weniger sprunghaft sind. Des Weiteren musste wie beim PKR keine Rundenhistorie zur Steigerung der Wiederholungswürfe

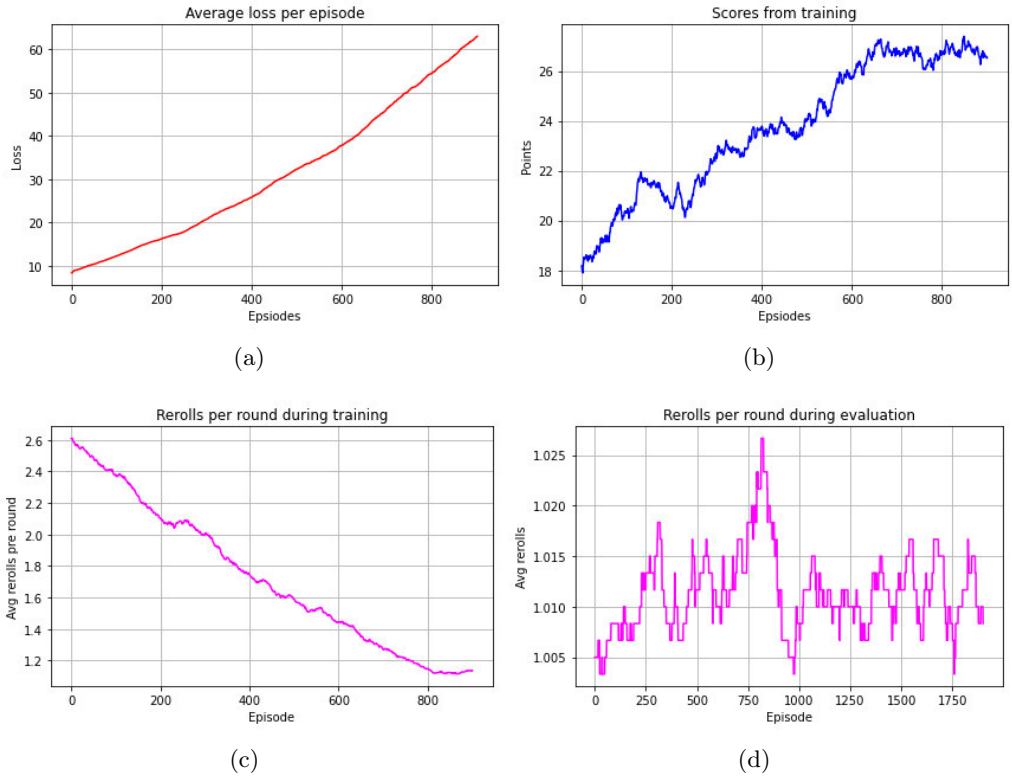


Abbildung 7.10: Diagramme vom PK (a) Verlustfunktion, (b) durchschnittliche Gesamtpunktzahl pro Episode, (c) Anzahl der Wiederholungswürfe pro Episode während des Trainings und (d) Anzahl der Wiederholungswürfe pro Episode während der Evaluation.

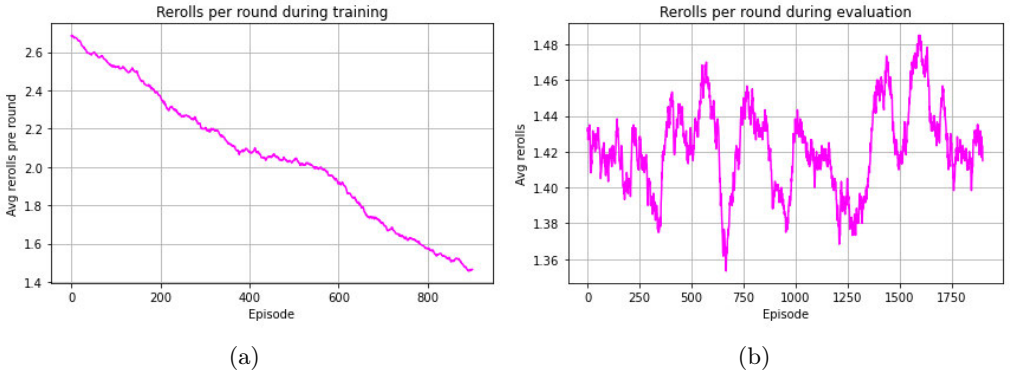


Abbildung 7.11: Diagramme vom PKR (a) Anzahl der Wiederholungswürfe pro Episode während des Trainings und (b) Anzahl der Wiederholungswürfe pro Episode während der Evaluation.

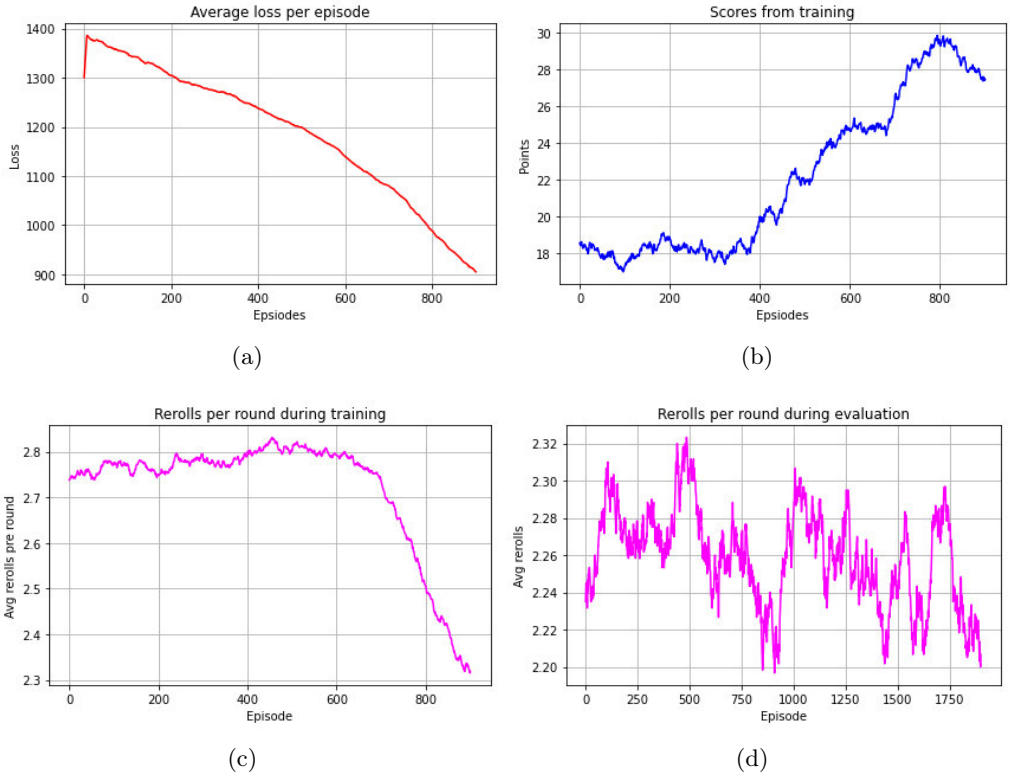


Abbildung 7.12: Diagramme vom MinΔ (a) Verlustfunktion, (b) durchschnittliche Gesamtpunktzahl pro Episode, (c) Anzahl der Wiederholungswürfe pro Episode während des Trainings und (d) Anzahl der Wiederholungswürfe pro Episode während der Evaluation.

genutzt werden. Dies liegt daran, dass die Wiederholungswürfe im Trainingsverlauf nicht so schnell abnehmen, wie bei der PK und PKR. Dies veranschaulichen auch die Abbildungen 7.12 (c) und (d). Des Weiteren wird daraus ersichtlich, dass sich die Anzahl der Wiederholungswürfe bei der Evaluation um den letzten erreichten Wert bewegt, der beim Training für die Anzahl der Wiederholungswürfe erreicht wurde. Bei der Gesamtpunktzahl bewegen sich beide um ca. 30 Punkte bei der Evaluation.

5. RWK:

RWK wurde einmal mit den Standardparametern getestet und einmal wurde die Episodenanzahl auf 10.000 erhöht und die Verzögerungsrate von Epsilon entsprechend auf $0,0001$ angepasst. Der Grund für die Erhöhung war zu überprüfen, ob sich die Anzahl der Wiederholungswürfe pro Runde mit steigender Trainingsanzahl verschlechtert. Wie

aus den Abbildungen 7.13 (a) und (b) ersichtlich wird, ist dies auch der Fall. Des Weiteren nahm nach der Erhöhung der Episodenanzahl die Gesamtpunktzahl nicht weiter zu. Diese bewegte sich nach Abbildungen 7.13 (c) und (d) weiter um die 30 Punkte. Auch zeigen die Abbildungen 7.13 (e) und (f) den Verlauf der beiden Verlustfunktionen, aus denen keine Konvergenz ersichtlich ist. Der Verlauf der Verlustfunktion stieg auch mit der Anzahl der Episoden weiter an. Dies war ein Grund dafür die Punkte zu normalisieren. Dadurch wird die Belohnung für den DQN-Agenten weniger sprunghaft und ein Konvergenzverhalten wahrscheinlicher. Des Weiteren war die Wahl der Höhe der Belohnung für die Wiederholungswürfe schwer zu ermitteln, damit der Agent einen ausreichenden Anreiz bekommt. Werden die möglichen Punkte und somit die Belohnungen nur für den oberen Tabellenteil betrachtet, dann sind die Belohnungen, die der DQN-Agent erhalten kann, 0 bis 30 Punkte in Abhängigkeit der gewählten Kategorie. Die Sprünge bei den Punkten und somit den Belohnungen für den DQN-Agenten können die Divergenz der Verlustfunktion verursachen.

6. NRWK:
Das NRWK wurde wie das RWK einmal mit 1.000 und 10.000 Episoden getestet. Es zeigte bis auf das Konvergenzverhalten, welches mit der steigenden Episodenanzahl zunahm, die gleichen Ergebnisse für die Maximierung der Gesamtpunktzahl.

7. R-U-M:
Nachdem die vorhergehenden Belohnungssysteme nicht das gewünschte Lernverhalten gezeigt haben, wurde die *reroll_ utility*-Funktion entwickelt. Diese wurde von der Arbeit [35] von Philip Vasseur inspiriert. Wie aus der Abbildung 7.15 für das Training entnommen werden kann, hat sich mithilfe der *reroll_ utility*-Funktion ein gewünschtes Lernverhalten für den oberen Tabellenabschnitt ergeben. Die Ergebnisse tangieren in Richtung der optimalen Strategie von Tom Verhoeff [36]. Wird die Tabelle 7.6 mit der Tabelle 3.1 in Kapitel 3.1 verglichen, schneidet der DQN-Agent in fast allen Kategorien außer für die Einsen schlechter ab. Dies kann daran liegen, dass er eine reine Maximierungsstrategie und keine optimale Strategie verfolgt. Das bedeutet, dass der DQN-Agent bei einem Wurf, welcher gut für die Kategorie der Einsen ist, versucht die maximal möglichen Punkte zu erzielen. Sind noch alle oder die Mehrheit der anderen Kategorien frei, ist das suboptimal, weil mit dem Feld der Einsen die Verluste gering gehalten werden können. Dies ist eine mögliche Ursache, dass die durchschnittliche Gesamtpunktzahl beim DQN-Agenten in der Evaluation in Höhe von $52,2$ mit einer Standardabweichung von $\pm 10,33$ Punkten geringer ausfällt, als bei der optimalen Strategie von Tom Verhoeff nach der Tabelle 3.1.

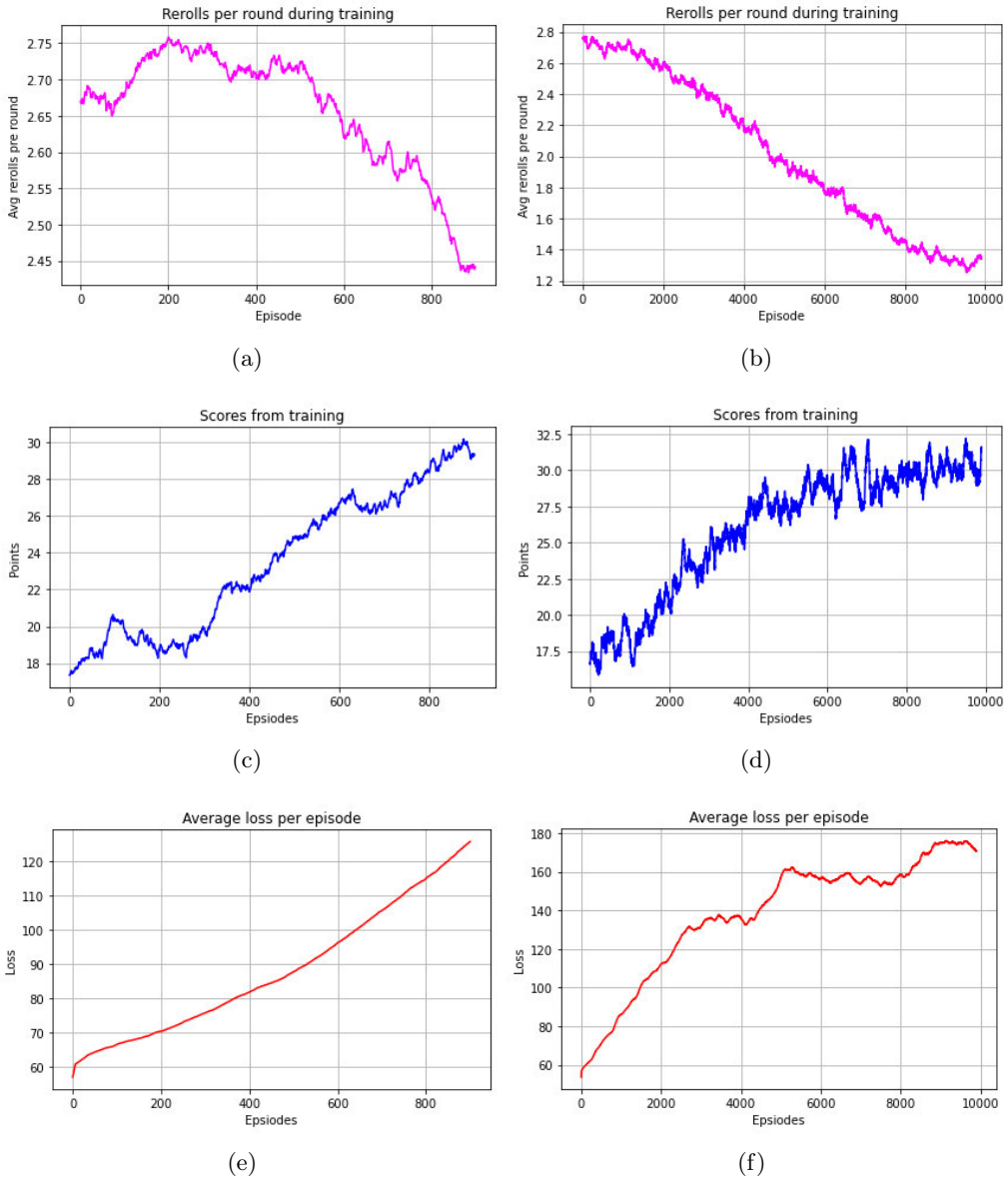
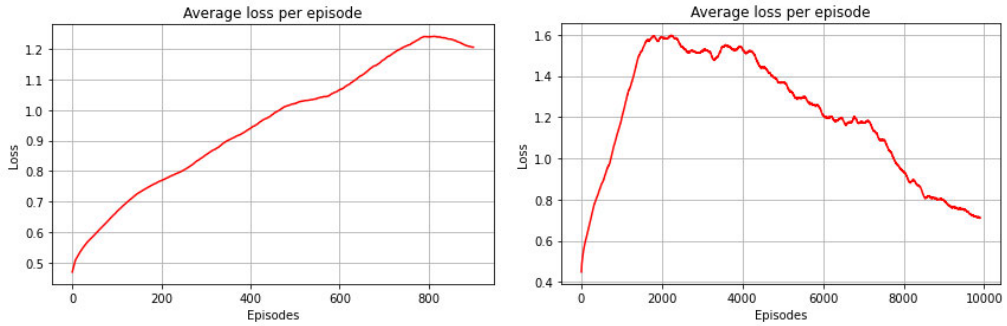


Abbildung 7.13: Diagramme vom RWK (a) Anzahl der Wiederholungswürfe pro Episode während des Trainings für 1.000 Episoden, (b) Anzahl der Wiederholungswürfe pro Episode während des Trainings für 10.000 Episoden, (c) Verlauf der Gesamtpunktzahl für 1.000 Episoden, (d) Verlauf der Gesamtpunktzahl für 10.000 Episoden, (e) Verlauf der Verlustfunktion für 1.000 Episoden und (f) Verlauf der Verlustfunktion für 10.000 Episoden.



(a) (b)

Abbildung 7.14: Diagramme vom NRWK (a) Verlauf der Verlustfunktion für 1.000 Episoden und (b) Verlauf der Verlustfunktion für 10.000 Episoden.

Episoden	**Einsen**	**Zweien**	**Dreien**	**Vieren**	**Fünfen**	**Sechsen**
1.000 Training	1, 49	3, 63	5, 09	6, 42	8, 06	10, 33
2.000 Evaluation	2, 26	5, 16	7, 5	9, 87	11, 39	16, 05

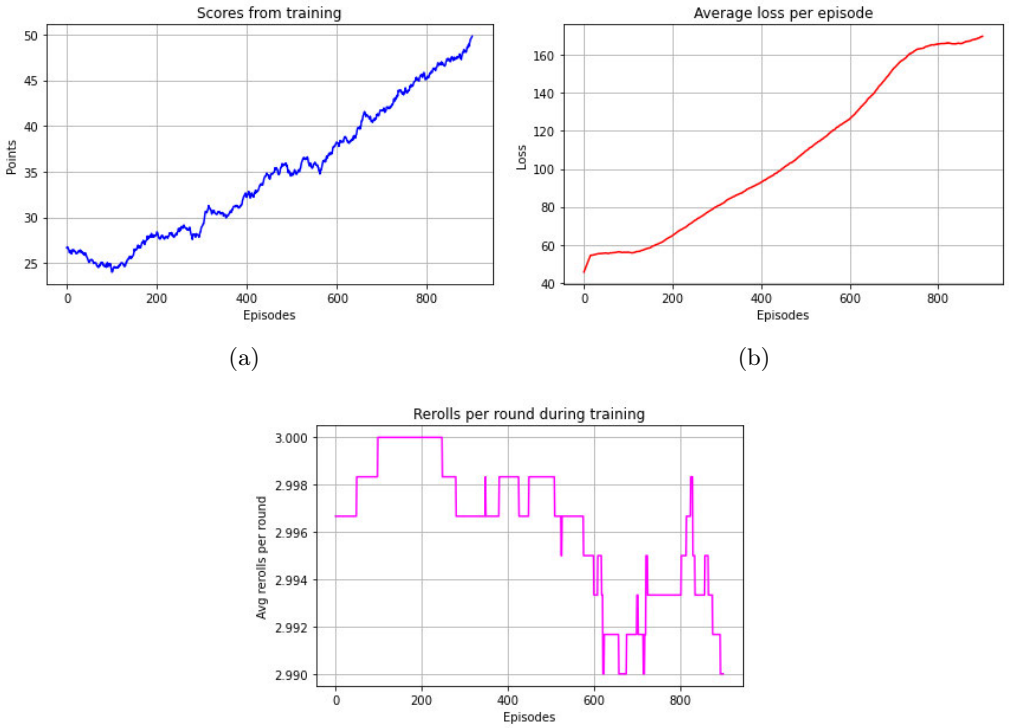
Tabelle 7.6: Erreichte Punkte pro Kategorie mit der R-U-M.

Ein Anreiz durch eine Belohnung für die Wiederholungswürfe muss durch diese Methode nicht gegeben werden. Es werden wie bei der PK die Punkte der ausgewählten Kategorie als Belohnung verwendet und nach der Auswahl einer Kategorie gelernt. Solange der DQN-Agent noch Wiederholungswürfe offen hat und die maximal mögliche Punktzahl für das Feld noch nicht erreicht ist, werden die Züge ausgenutzt. Dies ist in beiden Abbildungen 7.15 (c) und 7.16 (a) zu erkennen. Des Weiteren kann der Abbildung 7.16 (b) entnommen werden, dass das neuronale Netz stabil ist, weil es um die beim Training erreichte maximale Punktzahl pendelt. Obwohl die Gesamtpunktzahl zunimmt, divergiert die Verlustfunktion. Nachdem sich die *reroll_utility*-Funktion als erfolgreich für den oberen Tabellenabschnitt erwiesen hat, wurde diese für den unteren Tabellenabschnitt mit Bonus erweitert. Für das komplette Spiel wuchs die durchschnittliche Gesamtpunktzahl von ca. 70 Punkten auf ca. 130 Punkte an. In der Evaluation wurden im Schnitt 138, 4 Punkte erzielt. Des Weiteren wurde die Episodenanzahl auf 10.000 Episoden erhöht. Dies führte zu einer Steigerung der Gesamtpunktzahl auf über 140 Punkte. Dadurch, dass die Punkte der ausgewählten Kategorie als Belohnung verwendet werden, führt dies zu Divergenz der Verlustfunktion. Die folgenden Optimierungen (O1 - O5) führten schrittweise zu einer Konvergenz und sind in den Abbildungen 7.17 veranschaulicht.

- **O1:** Als Belohnung wird das *reward_ratio* verwendet(siehe Abbildung 7.17 (a)).

- **O2:** Die Episodenanzahl wurde von 1.000 auf 10.000 angehoben und die Epsilonverzögerungsrate von $0,001$ auf $0,0001$ gesetzt (siehe Abbildung 7.17 (a)).
- **O3:** Die Lernrate wurde auf $0,00001$ verkleinert (siehe Abbildung 7.17 (b)).
- **O4:** Die verdeckte Schicht fc2 wird nicht weiterverwendet (siehe Abbildung 7.17 (c)).
- **O5:** Zum Schluss wurde die Größe des Replay Buffers gleich der Batchgröße gesetzt. Diese beträgt für die in der Abbildung 7.17 (d) dargestellten Verlustfunktion für beide 50.

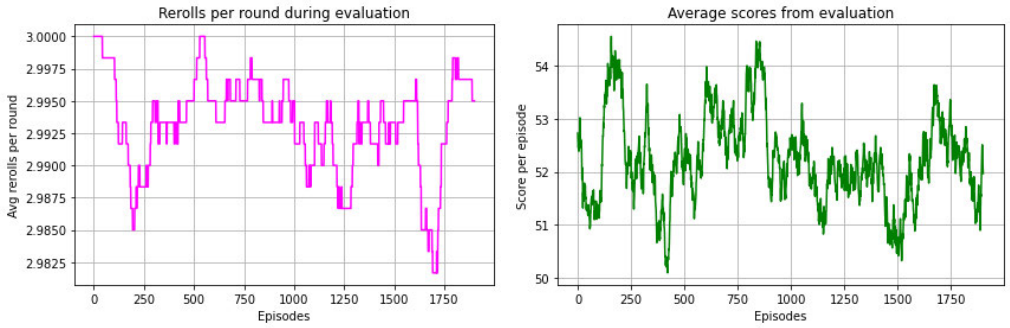
Die Optimierungen gingen jedoch zu Lasten der maximalen Punktzahl.



(a) (b)

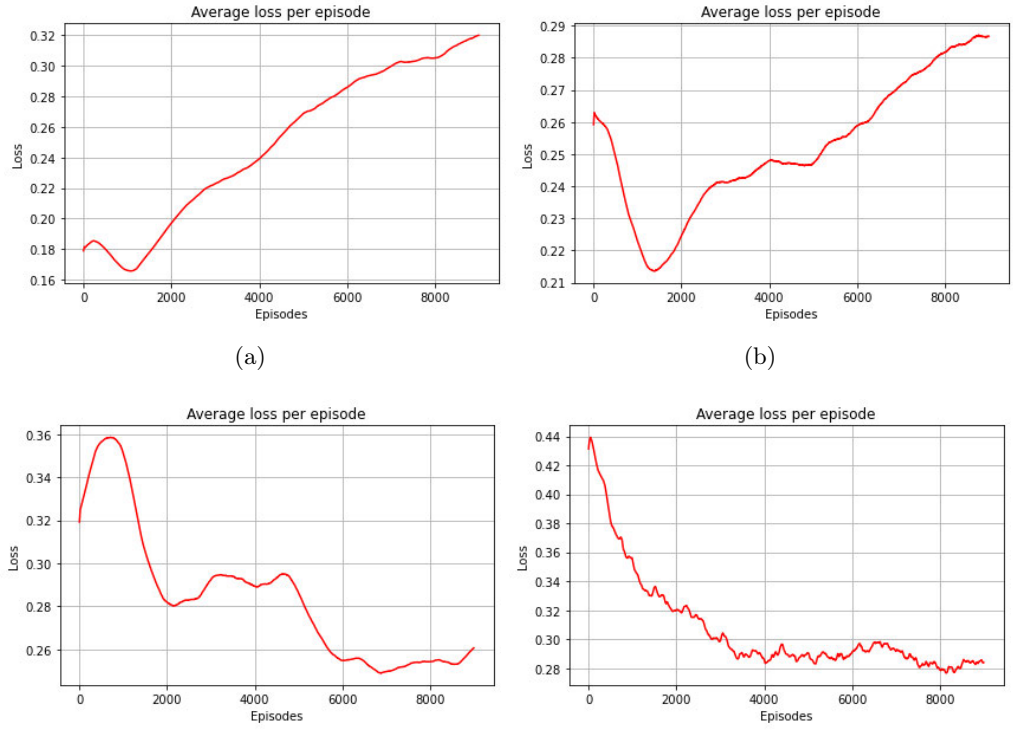
(c)

Abbildung 7.15: Diagramme vom R-U-M-Training. (a) durchschnittliche Gesamtpunktzahl pro Episode, (b) Verlustfunktion und (c) Anzahl der Wiederholungswürfe pro Episode während des Trainings.



(a) (b)

Abbildung 7.16: Diagramme von der R-U-M-Evaluation. (a) durchschnittliche Gesamtpunktzahl pro Episode, (b) Verlustfunktion und (c) Anzahl der Wiederholungswürfe pro Episode während des Trainings.



(a) (b)

(c) (d)

Abbildung 7.17: Optimierte Verlustfunktionen von O1-O5 (a) O1 und O2: Verwendung des reward_ratio und Erhöhung der Episodenanzahl, (b) O3: Anpassung der Lernrate, (c) O3: Reduzierung der Schichten des NN und (d) O5: Größe Replay Buffer = Batchgröße gesetzt.

8 Evaluierung

Zuerst werden die beiden Agenten kurz gegenübergestellt und anschließend eine Prüfung der Anforderungsanalyse durchgeführt.

8.1 Reinforcement- vs Deep Reinforcement-Learning

Ein direkter Vergleich zwischen den beiden Agenten ist aufgrund der *reroll_ utility*-Methode nicht möglich. Beide Agenten haben beim Würfelspiel, bei welcher es nur die Punkte der Kategorien als Belohnung gab, eine Punktmaximierung erreicht. Diese fiel jedoch nicht im ausreichendem Maße aus, sodass sie als Gegner für einen Menschen in Frage kommen würden. Für den Q-Agent wurden keine zusätzlichen Belohnungssysteme entwickelt. Verglichen mit dem DQN-Agenten hat der Q-Agent ohne zusätzliche Belohnungssysteme ein bessere Lernverhalten gezeigt. Es konnte für den Q-Agent die Aussage von Kang und Schroeder [14] bestätigt werden, dass der Q-Learning Algorithmus für das Würfelspiel ungeeignet ist und das Punkteniveaunur ein Level erreichte, wie der von ihnen genutzte Greedy-Level 1 Algorithmus. Der DQN-Agent erzielte dank der *reroll_ utility*-Methode deutliche bessere Ergebnisse als der Q-Agent. Leider ist auch dieser noch nicht auf dem Niveau, um gegen eine Menschen antreten zu können. Dies liegt an der *reroll_ utility*-Methode. Diese bietet noch Verbesserungspotential, um je nach Kategorie die Chancen zu verbessern, besonders für den unteren Tabellenbereich. Des Weiteren hat sich für den DQN-Agent gezeigt, dass eine Veränderung der Architektur, die im Listing 7.4 angegeben ist, nur eine geringe Auswirkung auf die Gesamtpunktzahl hat. Das heißt eine Steigerung der Schichten und Neuronen führt zu keiner signifikanten Steigerung der Gesamtpunktzahl. Das Gleiche gilt auch für die Veränderung des Seeds und damit der Zufallskomponente.

8.2 Prüfung der Anforderungen

In diesem Abschnitt sollen die Anforderungen, die im Kapitel 4.3 aufgestellt wurden, überprüft werden.

KI-F1: (*erfüllt*) Es wurde eine eigene Simulationsumgebung aufgebaut, mit dem die Agenten interagieren können und in welcher die Aktionen ausgeführt werden. Des Weiteren wurde darauf geachtet, dass der Agent die Spielregeln nicht verletzen oder ein Feld zweimal auswählen kann.

KI-F2: (*erfüllt*) Die Lernparameter können im jeweiligen *main*-Abschnitt des Agenten angepasst werden. Dadurch befinden sie sich gebündelt an einem Ort und werden bei der Initialisierung des Agenten übergeben.

KI-F3: (*erfüllt*) Die Komplexität des Spiels kann zusammen mit den Lernparametern konfiguriert werden. Dafür müssen die entsprechenden Parameter auf $True$ gesetzt werden. Die Basiseinstellung ist, dass der obere Tabellenteil immer ausgewählt ist und um den Bonus und / oder den unteren Tabellenteil erweitert werden kann.

KI-F4: (*erfüllt*) Der Trainingsfortschritt wird durch die Ausgabe von Diagrammen visualisiert und durch Ausgaben von Punkten in der Konsole messbar gemacht.

KI-F5: (*erfüllt*) Der DQN-Agent besitzt ein neuronales Netz. Des Architektur kann in der entsprechenden Klasse angepasst werden. Das implementierte neuronale Netz besitzt drei versteckte Schichten, die über die Lernparameter eingestellt werden können.

KI-F6: (*erfüllt*) Gibt es bereits vorhandene Trainingsdateien im Format .npy müssen diese in das gleiche Verzeichnis wie die Agenten geladen werden. Anschließend muss der Agent mit der Methode *load("Name.npy")* geladen werden. Es gilt darauf zu achten, dass die gleichen Parameter eingestellt sind, mit denen der Agent zuvor trainiert wurde.

KI-F7: (*erfüllt*) Die Trainingsdaten können mit der Methode *safe* gespeichert werden. Der Methode muss ein Name übergeben werden, unter welche diese die Daten speichert. Die Daten werden im .npy Format gespeichert.

KI-F8: (*erfüllt*) Der Agent trainiert ohne weitere Eingriffe selbstständig, nachdem al-

le notwendigen Einstellungen bei den Lernparametern getätigt wurden. Des Weiteren steht neben dem Training auch eine Möglichkeit der Evaluierung zur Verfügung. Dies bedeutet, dass bereits gespeicherte Trainingsdaten wieder geladen und deren Ergebnisse reproduziert werden können. Dabei gilt lediglich zu beachten, dass die Lernparameter die Gleichen sein müssen wie diejenigen, die für das Training eingestellt wurden.

KI-NF1: (*erfüllt*) Alle Klassen wurden in der Programmiersprache Python implementiert.

KI-NF2: (*teilweise erfüllt*) Das Programm wurde in verschiedene Klassen aufgeteilt. Die Methoden der Klassen selbst besitzen noch weiteres Verbesserungspotenzial. Zum Beispiel müssen für die Klasse des DQN-Agenten, abhängig vom verwendeten Belohnungssystem oder der Vortrainingsfunktion, umständlich Programmzeilen ein- oder auskommentiert werden. Gleichzeitig tragen weitere Kommentare zu einem noch besseren Programmverständnis bei.

9 Fazit und Ausblick

Es konnte in der vorliegenden Arbeit gezeigt werden, dass der DQN-Algorithmus und Q-Learning-Algorithmus in der Lage sind die Punkte des Würfelspiels Yahtzee zu maximieren. Für den Q-Learning-Algorithmus konnte ein erster praktischer Nachweis erbracht werden, dass dieser in der Arbeit von Kang und Schroeder [14] zurecht nicht berücksichtigt wurde. Interessanterweise war der Q-Learning-Algorithmus im Gegensatz zum verwendeten DQN-Algorithmus in der Lage mit dem Yahtzee-Environment bessere Lernfortschritte zu erzielen. Der DQN-Algorithmus erreichte diese grundlegend auch, jedoch nicht so erfolgreich wie der Q-Learning-Algorithmus. Der Q-Learning-Algorithmus nutzte unter anderem die Chancen durch die Wiederholungswürfe besser aus. Für den DQN-Algorithmus mussten extra Anreize im Belohnungssystem geschaffen werden, damit dieser seine Chancen besser ausnutzt. Schlussendlich wurde sich dafür entschieden die Wiederholungswürfe aus dem Aktionsraum des DQN-Agenten zu entfernen und diesen mit einer Hilfsfunktion zu unterstützen.

Für zukünftige Arbeiten könnte es interessant sein, wie die Problematik der unterschiedlichen Aufgaben in einem neuronalen Netz vereint werden können, da dies die größte Schwierigkeit bereitet hat. Hier stellt sich auch die Frage, ob ein anderer DQN-Ansatz gewählt werden sollte, z. B. mit einem Target-Netzwerk. Eine Untersuchung der Monte-Carlo-Tree-Search-Methode könnte ebenfalls vielversprechend sein. Diese beiden Algorithmen können durchaus genauere Ergebnisse liefern.

Die Entwicklung der beiden Algorithmen hat mehr Zeit in Anspruch genommen als ursprünglich erwartet. Dadurch konnte die ursprüngliche Zielsetzung nicht vollständig erreicht werden. Aus diesem Grund wurde auf die Realisierung eines Demonstrators mit einem Kamerasystem zu Gunsten der erfolgreichen Entwicklung der Agenten verzichtet. Dennoch bietet das erlangte Ergebnis dieser Arbeit einen guten Einstieg in das Thema Reinforcement Learning als auch Deep Reinforcement Learning, von dem aus Weiterentwicklungen durchgeführt werden können. Die Erweiterung eines Kamerasystems oder

einer Benutzeroberfläche bieten dafür die größten Potentiale.

Verbesserungen der Arbeit können unter anderem erzielt werden, in dem die Hilfsfunktion für den DQN-Agent optimiert wird. Die Strukturierung der Agenten-Klassen können für ein besseres Verständnis ebenfalls weiter verbessert werden. Es wurden im Programm Kommentierungen an den Schlüsselstellen vorgenommen, um anderen Benutzern einen leichten Einstieg zu bereiten. Für zukünftige Arbeiten oder andere Benutzer bietet die Arbeit den Vorteil, dass bereits erlangte Trainingsergebnisse zur Verfügung stehen, die reproduziert werden können. Des Weiteren bietet der Aufbau des sehr einfach gehaltenen Hauptabschnittes die Möglichkeit schnell und einfach selbst in die Thematik des Agenten Trainings einzusteigen.

Literaturverzeichnis

[1] Arel, I. ; Liu, C. ; Urbanik, T. ; Kohls, A.G.: *Reinforcement learning-based multi-agent system for network traffic signal control.* S. 8, The University of Tennessee, 2009

[2] Borowiec, Steven: *AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol*, The Guardian, 2016. – URL https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol

[3] Clevert, Djork-Arné ; Unterthiner, Thomas ; Hochreiter, Sepp: *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).* S. 14, ICLR, 2016. – URL https://arxiv.org/pdf/1511.07289v5

[4] Duchi, John ; Hazan, Elad ; Singer, Yoram: *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* S. 13, Journal of machine learning research, 2011. – URL https://web.stanford.edu/~jduchi/projects/DuchiHaSi10_colt.pdf

[5] Glenn, James: *An optimal strategy for Yahtzee.* S. 16, Loyola College in Maryland, 2006

[6] Glorot, Xavier ; Bengio, Yoshua: *Understanding the diffculty of training deep feedforward neural networks.* S. 8. In: *Paper*, Université de Montréal, 2010. – URL https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[7] Gridin, Ivan: *Practical Deep Reinforcement Learning with Python.* bpb online, 2022. – URL www.bpbonline.com. – ISBN 978-93-55512-055

[8] Géron, Aurélien: *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme. 1. Auflage.* dpunkt.verlag GmbH, 2018. – ISBN 978-3-96010-114-7

[9] Hasbro: *Yahtzee Rules*. S. 8, Hasbro, 1996. – URL https://www.hasbro.com/common/instruct/yahtzee.pdf

[10] Hasbro: *Yahtzee Rules*. S. 3, Hasbro, 2003. – URL https://www.hasbro.com/common/instruct/Yahtzee_(2003).pdf

[11] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. S. 11, Microsoft Research, 2015. – URL https://arxiv.org/pdf/1502.01852

[12] Hinton, Geoffrey: *Neuronal Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent*. S. 31, University of Toronto, lecture, 2012. – URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[13] Jendeberg, Daniel ; Wikstén, Louise: *OptimalYahtzeeaComparison*. S. 21, KTH. – URL https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group1Vahid/report/Optimal_Yahtzee_Nils_DN_&_Philip_S.pdf

[14] Kang, Minhyung ; Schroeder, Luca: *Reinforcement Learning for Solving Yahtzee*. S. 7, 2018

[15] Kngma, Diederik P. ; Ba, Jimmy: *Adam: A Method for Stochastic Optimization*. S. 15, conference paper at ICLR, 2015. – URL https://arxiv.org/abs/1412.6980

[16] Lapan, Maxim: *Deep Reinforcement Learning, Das umfassende Praxis-Handbuch*. mitp-verlag, 2020. – ISBN 978-3-7475-0037-8

[17] Larsson, Marcus ; Sjöberg, Andreas: *Optimal Yatzy Strategy*. S. 45, KTH, 2012. – URL https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group89Michael/report/Larsson+Sjoberg.pdf

[18] Lemke, Christian: *Reinforcement Learning kompakt erklärt*, URL https://www.alexanderthamm.com/de/blog/einfach-erklaert-so-funktioniert-reinforcement-learning/, 2023

[19] Mao, Hongzi ; Alizadeh, Mohammad ; Menache, Ishai ; Kandula, Srikanth: *Resource Management with Deep Reinforcement Learning*. S. 7, Microsoft Research, MIT, 2016. – URL https://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf

[20] Mnih, Volodymyr ; Badia, Adrià P. ; Mirza, Mehdi ; Graves, Alex ; Harley, Tim ; Lillicrap, Timothy P. ; Silver, David ; Kavukcuoglu, Koray: *Asynchronous Methods for Deep Reinforcement Learning*. S. 19, Google DeepMind & MILA, 2016. – URL https://arxiv.org/pdf/1602.01783v2

[21] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: *Playing Atari with Deep Reinforcement Learning*. S. 9, DeepMind Technologies, 2013. – URL https://arxiv.org/pdf/1312.5602

[22] Nielsen, Michael A.: *Neural Networks and Deep Learning*. Determination Press, 2015

[23] Norgren, Nils D. ; Svensson, Philip: *Optimal Yahtzee*. S. 45, KTH, 2013. – URL https://www.diva-portal.org/smash/get/diva2:812165/FULLTEXT01.pdf

[24] Perrotta, Paolo: *Machine Learning für Softwareentwickler*. dpunkt Verlag GmbH, 2020. – ISBN 978-3-86490-787-6

[25] Raschka, Sebastian ; Mirjalili, Vahid: *Machine Learning mit Python und Kears, Tensorflow 2 und Scikit-learn: Das umfassende Parxis-Handbuch für Data Science, Deepl Learning und Prediction Analystics. 3. Auflage*. mitp-Verlag, 2021. – ISBN 978-3-7475-0215-0

[26] Raschka, Sebastian ; Patterson, Joshua ; Nolet, Corey: *Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence*. S. 48, URL https://arxiv.org/pdf/2002.04803, 2020

[27] Robbins, Herbert ; Monro, Sutton: *A Stochastic Approximation Method*. S. 8, The Annals of Mathematical Statistics, Ann. Math. Statist. 22(3), 400-407, September, 1951. – URL https://doi.org/10.1214/aoms/1177729586

[28] Rosenblatt, Frank: *The perceptron: A probabilistic model for information storage and organization in the brain*. S. 23, American Psychological Association, 1958. – URL https://doi.org/10.1037/h0042519

[29] Ruder, Sebastian: *An overview of gradient descent optimization algorithms*. S. 14, NUI Galway Aylien Ltd, 2017. – URL https://arxiv.org/pdf/1609.04747

[30] Schulman, John ; Moritz, Philipp ; Levine, Sergey ; Jordan, Micheal I. ; Abbeel, Pieter: *High-Dimensional Continuous Control using Generalized Advantage Estimation.* S. 14, conference paper at ICLR, 2016. – URL https://arxiv.org/pdf/1506.02438

[31] Schulman, John ; Wolski, Filip ; Dhariwal, Prafulla ; Radford, Alec ; Klimov, Oleg: *Proximal Policy Opimization Algorithms.* S. 12, OpenAI, 2017. – URL https://arxiv.org/pdf/1707.06347

[32] Silver, D. ; Schrittwieser, J. ; Simonyan, K. ; Antonoglou, I. ; Huang, A. ; Guez, A. ; Hubert, T. ; Lai, L. Bakerand M. ; Bolton, A. ; Chen, Y. ; Lillicrap, T. ; Hui, F. ; Sifre, L. ; Driessche, G. van den ; Graepel, T. ; Hassabis, D.: *Mastering the game Go without human knowledge.* S. 42, Nature, 2017

[33] Sommer, Matthias: *Resilient Traffic Management from reactive to proactive systems.* S. 200, University of Augsburg, 2018

[34] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning An Introduction, second edition.* The MIT Press, 2018. – ISBN 9780262039246

[35] Vasseur, Philip: *Using Deep Q-Learning to Compare Strategy Ladders of Yahtzee.* S. 12, URL https://scholar.google.com/scholar?hl=de&as_sdt=0%2C5&q=Using+Deep+Q-Learning+to+Compare+Strategy+Ladders+of+Yahtzee&btnG=, 2019

[36] Verhoeff, Tom: *Optimal Solitaire Yahtzee Strategies.* S. 18, Eindhoven University of Technology, 1999-2000. – URL https://www-set.win.tue.nl/~wstomv/misc/yahtzee/slides-2up.pdf

[37] Watkins, Christopher J. ; Dayan, Peter: *Technical Note Q-Learning.* S. 14, Kluwer Academics Publishers, 1992. – URL https://link.springer.com/article/10.1007/BF00992698

[38] Williams, Ronald J.: *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning.* S. 28, Kluwer Academi, 1992. – URL http://link.springer.com/content/pdf/10.1007/BF00992696.pdf

[39] Zai, Alexander ; Brown, Brandon: *Einstieg in Deep Reinforcement Learning KI-Agenten mit Python und Pytorch programmieren.* Hanser-Verlag, 2020. – URL www.hanser-fachbuch.de. – ISBN 978-3-446-45900-7

[40] Zeiler, Matthew D.: *ADADELTA: AN ADAPTIVE LEARNING RATE METHOD*. S. 6, Google Inc. & New York University, 2012. – URL https://arxiv.org/pdf/1212.5701

[41] Zhang, Jingzhao ; He, Tianxing ; Sra, Suvrit ; Jadbabaie, Ali: *Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity*. S. 21, Nassachusetts Institute of Technology, 2020. – URL https://arxiv.org/pdf/1905.11881

A Anhang

Der Anhang zur Arbeit befindet sich auf der beigelegten CD und kann beim Erstgutachter eingesehen werden. Auf dem zugehörigen Datenträger befinden sich folgende Anhänge:

- Die Custom Environment für das Würfelspiel
- Der Code des Q-Agenten
- Der Code des DQN-Agenten
- Die Ergebnisse der verschiedenen Untersuchungen beider Agenten
- Verschiedene trainierte Modelle der Agenten

Erklärung zur selbstständigen 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