

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

BACHELORARBEIT

Entwicklung einer Simulationsumgebung und eines Deep-Reinforcement-Learning Agenten für das Brettspiel Aqualin basierend auf Proximal Policy Optimization

Development of a Simulation Environment and of a Deep Reinforcement Learning Agent for the Board Game Aqualin based on Proximal Policy Optimization

vorgelegt am 24.08.2026
Felix Schubert
Matrikelnummer: [redacted]

Erstprüfer/in: Prof. Dr. Larissa Putzar
Zweitprüfer/in: Sune Timon Maute

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Studiengang Media Systems
Finkenau 35
20081 Hamburg

Vorwort

Ich bedanke mich herzlich bei allen, die mich bei der Anfertigung dieser Bachelorarbeit unterstützt haben. Ein ganz besonderer Dank gilt meiner Betreuerin Prof. Dr. Larissa Putzar für die großartige und wertvolle Begleitung während des gesamten Projekts.

Ebenso möchte ich mich bei allen Korrekturlesern für ihre investierte Zeit, ihre Mühe und das hilfreiche Feedback bedanken. Nicht zuletzt gilt mein Dank natürlich auch den zahlreichen Studienteilnehmern, ohne deren Bereitschaft und Spielfreude die abschließende Mensch-Maschine-Evaluierung nicht möglich gewesen wäre.

Zusammenfassung

Künstliche Intelligenz hat in den letzten Jahren gelernt, viele klassische Brettspiele auf höchstem Niveau zu spielen. Diese Bachelorarbeit widmet sich einer neuen Herausforderung: dem asymmetrischen Zwei-Spieler-Spiel *Aqualin*. Ziel der Arbeit war es, ein neuronales Netz zu entwickeln, das die Regeln und komplexen Strategien des Spiels völlig selbstständig erlernt, ohne auf menschliches Vorwissen zurückzugreifen.

Die wesentlichen Hürden von *Aqualin* (gegensätzliche Spielziele sowie ein enormer Aktionsraum durch den zweiphasigen Spielzug) wurden durch eine maßgeschneiderte Simulationsumgebung bewältigt. Die Asymmetrie wurde durch eine rollenspezifische Sichtweise (*Semantic Vision*) gelöst, während der komplexe Spielzug für das Netz in handhabbare Einzelschritte zerlegt wurde.

Der Lernprozess basierte auf dem *Proximal Policy Optimization* (PPO)-Algorithmus. In Millionen von Trainingspartien gegen einen vielfältigen Gegner-Pool wurde der Agent schrittweise (*Curriculum Learning*) an komplexe Situationen herangeführt. Die abschließenden Evaluierungen und eine Mensch-Maschine-Studie zeigen, dass das System das Spielprinzip erfolgreich erlernt und ein kompetitives Leistungsniveau erreicht hat, mit dem es menschliche Spieler besiegen kann.

Abstract

In recent years, artificial intelligence has learned to play many classic board games at the highest level. This bachelor thesis focuses on a new challenge: the asymmetric two-player game *Aqualin*. The goal of this work was to develop a neural network that independently learns the rules and complex strategies of this game, without relying on human prior knowledge.

The main hurdles of *Aqualin* (opposing player objectives and an enormous action space caused by the two-phase turn) were overcome through a custom-built simulation environment. The asymmetry was resolved by providing the agent with a role-specific perspective (*Semantic Vision*), while the complex turn was internally divided into manageable steps.

The learning process was based on the *Proximal Policy Optimization* (PPO) algorithm. Through millions of training matches against a diverse pool of opponents, the agent was gradually introduced to increasingly complex situations (*Curriculum Learning*). The final evaluations, including a human-machine study, demonstrate that the developed system has successfully learned the game mechanics and achieved a competitive level of play, capable of defeating human players.

Inhaltsverzeichnis

Vorwort iii

Zusammenfassung v

Abstract v

Abkürzungsverzeichnis xv

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Problemstellung . . . 2
- 1.3 Zielsetzung . . . 2
- 1.4 Aufbau der Arbeit . . . 3

2 Grundlagen 5
- 2.1 Das Spiel Aqualin . . . 5
  - 2.1.1 Grundidee des Spiels . . . 5
  - 2.1.2 Spielmaterial und Spielfeld . . . 5
  - 2.1.3 Spielablauf . . . 5
  - 2.1.4 Wertungssystem . . . 6
- 2.2 Neuronale Netze . . . 7
  - 2.2.1 Künstliche Neuronen und Aktivierungsfunktionen . . . 7
  - 2.2.2 Feedforward-Architektur und Backpropagation . . . 7
- 2.3 Reinforcement Learning . . . 8
  - 2.3.1 Der Interaktionskreislauf: Agent, Environment und Action . . . 9
  - 2.3.2 Zustandsraum und Beobachtbarkeit (State) . . . 9
  - 2.3.3 Zielerreichung: Rewards und Return . . . 10
  - 2.3.4 Exploration-Exploitation-Dilemma . . . 10
- 2.4 Convolutional Neural Networks (CNNs) . . . 10
  - 2.4.1 Faltungsparameter: Padding und Stride . . . 13
  - 2.4.2 Batch Normalization . . . 13
- 2.5 Proximal Policy Optimization (PPO) . . . 14
  - 2.5.1 Ablauf: Datensammlung und Training . . . 14
  - 2.5.2 Actor-Critic-Architektur und die Advantage-Funktion . . . 15
  - 2.5.3 Das Herzstück: Die Clipped Surrogate Objective . . . 15
  - 2.5.4 Dateneffizienz durch Mini-Batches und Epochen . . . 16
  - 2.5.5 Exploration und Entropie-Bonus . . . 16
  - 2.5.6 Umgang mit verzögerten Belohnungen (Sparse Rewards) . . . 16
- 2.6 Metriken zur Leistungsbewertung . . . 17
  - 2.6.1 Performance-Metriken . . . 17

2.6.2 Trainings-Metriken 17

3 Stand der Technik **19**
3.1 Historische Entwicklung der KI in Brettspielen 19
3.2 Das AlphaZero-Paradigma 21
3.3 Repräsentation durch Semantic Vision 24
3.4 Invalid Action Masking 24
3.5 Multi-Agent Trainingsmethodik 25
3.6 Curriculum Learning 25

4 Konzept **27**
4.1 Systemübersicht 27
4.2 Formalisierung als Markov-Entscheidungsprozess 28
4.3 Entwurf der Zustandsrepräsentation 29
4.4 Aktionsraum und Invalid Action Masking 32
4.5 Architektur des neuronalen Netzes 32
4.6 Reward-Design 34
4.7 Trainingsstrategie & Curriculum Learning 34
4.8 Heuristische Basis-Agenten 36

5 Implementierung **37**
5.1 Die Architektur des *Aqualin*-Environments 37
5.1.1 Schicht 1: Die Core Engine 38
5.1.2 Schicht 2: Das Base Environment 38
5.1.3 Schicht 3: Der Observation Wrapper 40
5.1.4 Schicht 4: Der Self-Play-Wrapper 42
5.2 Die Neuronale Netzarchitektur: NeptuneCNN 42
5.3 Trainingsinfrastruktur und Parallelisierung 43
5.4 Steuerung des Curriculums: Benchmark-Callbacks 46

6 Evaluierung **47**
6.1 Versuchsaufbau und Methodik 47
6.2 Analyse des Trainingsverlaufs und der Modellarchitektur 48
6.2.1 Einfluss der KL-Divergenz auf das Curriculum Learning 48
6.2.2 Trainingsphasen von *Poseidon* (I und II) 49
6.2.3 Konvergenz der PPO-Metriken 53
6.3 Quantitative Spielstärke: Benchmark-Ergebnisse 56
6.4 Asymmetrie-Analyse: Farben vs. Tierarten 56
6.5 Mensch-Maschine-Evaluierung 57

7 Diskussion **63**
7.1 Interpretation der Ergebnisse 63
7.2 Kritische Reflexion und Limitierungen 63
7.3 Ausblick 64

8 Fazit **65**

Eidesstattliche Erklärung **69**

Erklärung zur Nutzung von Künstlicher Intelligenz **71**

Abbildungsverzeichnis

2.1 Spielfeld und Spielsteine von *Aqualin* (Eigene Darstellung) 6
2.2 Schematische Darstellung eines künstlichen Neurons 8
2.3 Der Interaktionskreislauf im Reinforcement Learning 9
2.4 Isometrische Darstellung einer Convolution (3×3 Kernel) 11
2.5 Typische Architektur eines CNNs 12
2.6 Systematik des Actor-Critic PPO-Algorithmus 14

3.1 Kupferstich des Schachtürken von Wolfgang von Kempelen 20
3.2 Die „Two-Headed“ Architektur von AlphaZero 23
3.3 Visualisierung des MCTS-Prozesses 23

4.1 Konzeptionelle Übersicht des Gesamtsystems 27
4.2 Beispielhafter Spielzustand von *Aqualin* 30
4.3 Dekomposition des Spielzustands in binäre Kanäle 30
4.4 Konzeptionelle Architektur des neuronalen Netzes 33

5.1 Verlauf der Squashed Margin Reward Funktion 41
5.2 Architektur des NeptuneCNN 44

6.1 Curriculum-Fortschritt: Orion vs. *Poseidon* I 48
6.2 Lernkurve und Curriculum in Phase 1 50
6.3 Point Margin in Phase 1 51
6.4 Lernkurve und Curriculum in Phase 2 52
6.5 Point Margin in Phase 2 52
6.6 Konvergenz der PPO-Metriken in Phase 1 53
6.7 Konvergenz der PPO-Metriken in Phase 2 55
6.8 Benchmark Win Rates (1.000 Spiele) 56
6.9 Gewinnraten nach Spieler-Rolle 57
6.10 Grafische Testumgebung der Studie 58
6.11 Siegquote der menschlichen Spieler in der Studie 59
6.12 Punktedifferenz in der Mensch-Maschine-Studie 60
6.13 Menschliche Siegquoten nach Rolle 62

Tabellenverzeichnis

2.1 Punktwerte für Gruppen gleicher Farbe bzw. Tierart 7

5.1 Hierarchische Struktur der Software-Komponenten 37
5.2 Durchschnittlich legale Aktionen im Spielverlauf 39
5.3 Hyperparameter des *Poseidon*-Trainings 45

Abkürzungsverzeichnis

AEC Agent-Environment-Cycle

API Application Programming Interface

BFS Breadth-First Search

BN Batch Normalization

CNN Convolutional Neural Network

DRL Deep Reinforcement Learning

GAE Generalized Advantage Estimation

GUI Graphical User Interface

KI Künstliche Intelligenz

KL Kullback-Leibler

KNN Künstliches Neuronales Netz

MCTS Monte Carlo Tree Search

MDP Markov Decision Process

MLP Multi-Layer Perceptron

PPO Proximal Policy Optimization

ReLU Rectified Linear Unit

RL Reinforcement Learning

TD Temporal Difference

1 Einleitung

1.1 Motivation

Seit dem Sieg von AlphaGo über den südkoreanischen Go-Großmeister Lee Sedol im Jahr 2016 hat Deep Reinforcement Learning (DRL) seine Leistungsfähigkeit bei der Lösung strategischer Entscheidungsprobleme eindrucksvoll demonstriert (Silver et al., 2016). Während klassische Verfahren wie Minimax meist auf explizit kodiertes Spielwissen (Heuristiken) angewiesen sind, lernt ein DRL-Agent seine Strategie ausschließlich durch Interaktion mit der Umgebung, ohne menschliches Vorwissen über optimale Züge.

Die vorliegende Arbeit überträgt diesen Ansatz auf *Aqualin*, ein kombinatorisches Zwei-Spieler-Brettspiel, das trotz seines einfachen Regelwerks eine bemerkenswerte taktische Spieltiefe aufweist. Die Struktur des Spiels führt zu einem großen diskreten Aktionsraum und stellt damit eine interessante Herausforderung für lernbasierte Entscheidungsverfahren dar.

Darüber hinaus existiert bislang kein veröffentlichter Agent für *Aqualin*, der Künstliche Intelligenz (KI) nutzt. Ziel dieser Arbeit ist es daher, zunächst eine geeignete Spielumgebung zu implementieren, die das Regelwerk von *Aqualin* formal abbildet und als Trainingsumgebung für Reinforcement-Learning-Verfahren dient. Auf dieser Grundlage wird anschließend ein lernfähiger Agent entwickelt, trainiert und experimentell evaluiert.

Hierzu wird insbesondere der Algorithmus der Proximal Policy Optimization (PPO) in Verbindung mit einer semantischen Zustandsrepräsentation, die auf einem Convolutional Neural Network (CNN) basiert, eingesetzt. Der Schwerpunkt der Untersuchung liegt auf der quantitativen Analyse des Trainingsprozesses sowie der daraus resultierenden Spielstärke des Agenten.

1.2 Problemstellung

Brettspiele dienen seit vielen Jahren als wichtige Testumgebungen für Künstliche Intelligenz (Sutton and Barto, 2018), da sie klar definierte Regeln, einen strukturierten Zustandsraum und strategische Entscheidungsprozesse bieten. Während für viele klassische Spiele wie Schach oder Go bereits leistungsfähige KI-Systeme entwickelt wurden, existieren für zahlreiche modernere Brettspiele bislang keine vergleichbaren Ansätze.

Das Spiel *Aqualin* weist mehrere Eigenschaften auf, die es aus Sicht der Künstlichen Intelligenz zu einem interessanten Untersuchungsobjekt machen. Trotz seines vergleichsweise einfachen Regelwerks entsteht durch die Kombination aus Verschiebe- und Platzierungsaktionen ein großer diskreter Aktionsraum. Gleichzeitig sind viele der theoretisch möglichen Aktionen in einer gegebenen Spielsituation nicht zulässig, da sie durch die aktuelle Belegung des Spielfelds eingeschränkt werden.

Darüber hinaus können einzelne Spielzüge langfristige Auswirkungen auf die Struktur des Spielfelds und die spätere Gruppenbildung haben. Entscheidungen in frühen Spielphasen beeinflussen somit häufig die strategischen Möglichkeiten in späteren Zügen. Diese langfristigen Abhängigkeiten stellen eine zusätzliche Herausforderung für lernbasierte Entscheidungsverfahren dar.

Trotz dieser interessanten Eigenschaften existiert bislang kein veröffentlichter KI-Agent, der speziell für *Aqualin* entwickelt und untersucht wurde. Insbesondere fehlen experimentelle Untersuchungen dazu, inwieweit moderne Reinforcement-Learning-Verfahren in der Lage sind, Strategien für dieses Spiel eigenständig zu erlernen.

1.3 Zielsetzung

Ziel dieser Arbeit ist die Konzeption, Implementierung und Evaluierung eines DRL-Agenten für das Brettspiel *Aqualin*. Da moderne DRL-Verfahren ihre Leistungsfähigkeit in komplexen Brettspielen bereits grundsätzlich bewiesen haben, liegt der Fokus dieser Arbeit auf der praktischen Anwendbarkeit: Es soll untersucht werden, wie das Spiel effizient als Markov Decision Process (MDP) modelliert werden kann und mit welchen architektonischen Ansätzen (z. B. Action Masking) ein Agent das Spiel unter moderatem Rechenaufwand erfolgreich erlernen kann.

Hierfür wird eine Spielumgebung entwickelt, die das Regelwerk des Spiels formal abbildet und als Trainingsumgebung für einen lernbasierten Agenten dient. Auf dieser Grundlage wird ein Reinforcement-Learning-Agent implementiert und trainiert. Ziel dieses Agenten ist es, zunehmend effektivere Spielentscheidungen zu treffen.

Ein weiterer Schwerpunkt der Arbeit liegt auf der Analyse des Trainingsprozesses. Dabei wird untersucht, ob sich im Verlauf des Trainings ein Lernfortschritt erkennen lässt und welche Schwierigkeiten während des Trainings auftreten können. Abschließend wird die Spielstärke des trainierten Agenten anhand von Benchmark-Experimenten gegen einfachere Referenzgegner bewertet.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in acht Kapitel. Nach diesem ersten Kapitel *Einleitung* ist der weitere Verlauf wie folgt strukturiert:

- **Kapitel 2 – Grundlagen:** Zunächst werden die theoretischen Grundlagen geschaffen. Dabei geht es vor allem um das Brettspiel *Aqualin* und die wichtigsten Konzepte des *Reinforcement Learnings*.
- **Kapitel 3 – Stand der Technik:** Hier wird ein Blick auf bestehende Literatur geworfen und untersucht, welche KI-Ansätze bereits für ähnliche Brettspiele genutzt wurden.
- **Kapitel 4 – Konzept:** Auf Basis der bisherigen Erkenntnisse wird das theoretische Konzept für den Agenten entwickelt. Insbesondere wird gezeigt, wie sich das Spiel als MDP modellieren lässt.
- **Kapitel 5 – Implementierung:** Dieses Kapitel beschreibt die praktische Umsetzung. Es wird detailliert erklärt, wie die Spielumgebung programmiert und der PPO-Algorithmus integriert wurde.
- **Kapitel 6 – Evaluierung:** In diesem Teil werden die durchgeführten Trainingsexperimente ausgewertet. Dabei wird der Trainingsverlauf betrachtet und geprüft, wie spielstark der Agent geworden ist.
- **Kapitel 7 – Diskussion:** Anschließend werden die Ergebnisse kritisch hinterfragt. Es wird diskutiert, welche Probleme aufgetreten sind und wo die Grenzen des gewählten Ansatzes liegen.
- **Kapitel 8 – Fazit:** Den Abschluss bildet eine Zusammenfassung der wichtigsten Erkenntnisse sowie ein kurzer Ausblick auf mögliche Weiterentwicklungen.

2 Grundlagen

2.1 Das Spiel Aqualin

2.1.1 Grundidee des Spiels

Aqualin ist ein abstraktes Zwei-Spieler-Brettspiel (Franckh-Kosmos Verlags-GmbH & Co. KG, 2020), bei dem beide Spieler versuchen, zusammenhängende Gruppen von Spielsteinen auf einem 6×6 Felder großen Spielfeld zu bilden. Ziel des Spiels ist es, möglichst große Gruppen zu erzeugen, da die Punktzahl einer Gruppe mit deren Größe zunimmt.

Eine besondere Eigenschaft von *Aqualin* ist die asymmetrische Spielmechanik: Beide Spieler nutzen zwar dasselbe Spielfeld und identische Steine, verfolgen am Ende jedoch ganz unterschiedliche Ziele. Während ein Spieler versucht, möglichst große Farbgruppen zu bilden, konzentriert sich der andere Spieler rein auf das Verbinden gleicher Tierarten.

Daraus entsteht ein ständiges strategisches Spannungsfeld. Das Platzieren oder Verschieben eines Steins unterstützt nicht nur den eigenen Spielplan, sondern beeinflusst oft auch direkt die Möglichkeiten des Gegners.

2.1.2 Spielmaterial und Spielfeld

Das Spielmaterial von *Aqualin* besteht aus einem quadratischen Spielfeld mit 6×6 Feldern sowie insgesamt 36 Spielsteinen. Das Spielfeld umfasst damit insgesamt 36 Positionen, auf denen im Verlauf des Spiels die Steine platziert werden.

Die Spielsteine sind durch zwei unabhängige Eigenschaften charakterisiert: eine Farbe und eine Tierart. Insgesamt existieren sechs verschiedene Farben sowie sechs unterschiedliche Tierarten. Jeder Spielstein stellt dabei eine eindeutige Kombination aus genau einer Farbe und genau einer Tierart dar. Da jede mögliche Kombination genau einmal vorkommt, ergibt sich eine Gesamtzahl von $6 \times 6 = 36$ unterschiedlichen Steinen (siehe Abbildung 2.1).

2.1.3 Spielablauf

Das Spiel verläuft rundenbasiert und die beiden Spieler führen ihre Züge abwechselnd aus. Ein Zug besteht dabei stets aus zwei aufeinanderfolgenden Aktionen.

Zunächst darf der aktive Spieler optional einen bereits auf dem Spielfeld liegenden Spielstein verschieben. Diese Aktion, die im weiteren Verlauf der Arbeit als *Slide*-Schritt bezeichnet wird, erlaubt es, einen Stein horizontal oder vertikal um eine beliebige Anzahl freier Felder zu bewegen. Dabei darf der bewegte Stein weder über andere Steine hinweg springen noch auf einem bereits belegten Feld landen. Der Bewegungsweg muss daher vollständig frei sein.



Abbildung 2.1: Spielfeld und Spielsteine von *Aqualin* (Eigene Darstellung)

Im Anschluss an diesen optionalen Schritt muss der Spieler einen neuen Spielstein auf ein freies Feld des Spielfelds platzieren. Beide Spieler ziehen dabei aus derselben gemeinsamen, offenen Auslage von sechs Steinen (im Folgenden als *Pool* bezeichnet). Dieser Pool repräsentiert eine Teilmenge der insgesamt 36 im Spiel vorhandenen Steine. Wählt ein Spieler einen Stein aus dem Pool und platziert ihn auf dem Spielfeld, wird anschließend ein neuer Stein aus dem verbleibenden Vorrat aufgedeckt, sofern noch nicht alle Steine verwendet wurden.

Nachdem ein Spieler beide Aktionen seines Zuges abgeschlossen hat, ist der andere Spieler am Zug. Das Spiel endet, sobald alle 36 Spielsteine auf dem Spielfeld platziert wurden.

2.1.4 Wertungssystem

Am Ende des Spiels werden die zusammenhängenden Gruppen von Spielsteinen auf dem Spielfeld bewertet. Eine Gruppe besteht aus mehreren orthogonal benachbarten Steinen, die entweder die gleiche Farbe oder die gleiche Tierart besitzen. Diagonale Verbindungen werden dabei nicht berücksichtigt.

Die beiden Spieler verfolgen unterschiedliche Bewertungsziele. Während ein Spieler versucht, Gruppen gleicher Farben zu bilden, konzentriert sich der andere Spieler darauf, Gruppen gleicher Tierarten zu verbinden. Beide Spieler bewerten somit dieselbe Spielfeldkonfiguration anhand unterschiedlicher Eigenschaften der Spielsteine.

Die Punktzahl einer Gruppe hängt von der Größe der Gruppe ab. In *Aqualin* werden Gruppen ab einer Größe von zwei Steinen gewertet. Tabelle 2.1 zeigt die entsprechenden Punktwerte.

Die Gesamtpunktzahl eines Spielers ergibt sich aus der Summe der Punkte aller von ihm gebildeten Gruppen. Dabei wird jede Gruppe separat bewertet, auch wenn mehrere Gruppen derselben Farbe beziehungsweise Tierart existieren. Der Spieler mit der höheren Gesamtpunktzahl gewinnt die Partie.

Gruppengröße	Punkte
2	1
3	3
4	6
5	10
6	15

Tabelle 2.1: Punktwerte für Gruppen gleicher Farbe bzw. Tierart

2.2 Neuronale Netze

Künstliche Neuronale Netze (KNN) bilden die technologische Grundlage für moderne Reinforcement-Learning-Systeme. Sie sind lose von der Funktionsweise biologischer Gehirne inspiriert und zeichnen sich durch ihre Fähigkeit aus, komplexe Muster in Daten zu erkennen und mathematische Funktionen beliebiger Komplexität zu approximieren (Goodfellow et al., 2016). Im Kontext dieser Arbeit dienen sie dazu, die Handlungsstrategie (*Policy*) und die Bewertung von Spielzuständen (*Value*) des *Aqualin*-Agenten zu modellieren.

2.2.1 Künstliche Neuronen und Aktivierungsfunktionen

Ein künstliches Neuron ist der grundlegende Baustein eines neuronalen Netzes (siehe Abbildung 2.2). Mathematisch gesehen verrechnet es verschiedene Eingabewerte, indem es sie gewichtet aufsummiert und anschließend einen Schwellenwert *Bias* addiert (Nielsen, 2015, Kap. 1). Diese Gewichte und der Bias sind die eigentlichen Lernparameter des Netzwerks. Sie werden während des Trainings kontinuierlich angepasst, damit das Neuron lernt, welche Eingaben für das Ergebnis besonders wichtig sind.

Damit das Netzwerk nicht nur einfache, lineare Zusammenhänge erlernen kann, wird das Ergebnis dieser Rechnung noch durch eine Aktivierungsfunktion transformiert. Ohne diesen nichtlinearen Zwischenschritt wäre selbst ein Netzwerk mit vielen Schichten am Ende mathematisch gesehen nur eine einzige lineare Funktion, wodurch es keine komplexen Probleme lösen könnte (Goodfellow et al., 2016, S. 168 ff.). Eine der heutzutage wichtigsten Aktivierungsfunktionen ist die Rectified Linear Unit (ReLU), die schlicht definiert ist als $f(x) = \max(0, x)$. Sie hat sich durchgesetzt, weil sie das Problem der verschwindenden Gradienten (*Vanishing Gradient Problem*) effektiv umgeht und das Training von tiefen Netzwerken dadurch deutlich stabiler macht (Glorot et al., 2011).

2.2.2 Feedforward-Architektur und Backpropagation

In einer *Feedforward*-Architektur sind die Neuronen in mehreren Schichten hintereinander angeordnet. Die Informationen fließen dabei immer nur in eine einzige Richtung. Sie starten in der Eingabeschicht, durchlaufen eine oder mehrere verborgene Schichten (*Hidden Layers*) und kommen schließlich in der Ausgabeschicht an. Auf diesem Weg filtert jede Schicht bestimmte Merkmale heraus und baut so Schritt für Schritt ein immer abstrakteres Verständnis der ursprünglichen Daten auf.

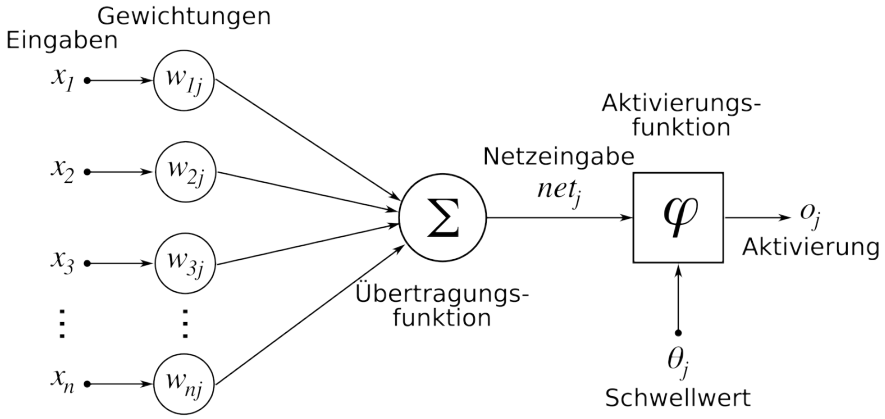


Abbildung 2.2: Schematische Darstellung eines künstlichen Neurons mit Gewichtung, Summation und Aktivierungsfunktion (entnommen aus Chrislb (2005)).

Das eigentliche Lernen geschieht durch den sogenannten *Backpropagation*-Algorithmus, also die Fehlerrückführung (Nielsen, 2015, Kap. 2). Dabei verarbeitet das Netzwerk zunächst die Eingaben in einem Vorwärtsdurchlauf (*Forward Pass*) und trifft eine Vorhersage. Anschließend vergleicht eine Verlustfunktion (*Loss Function*), wie stark diese Vorhersage vom gewünschten Ziel abweicht. Dieser Fehler wird dann von der Ausgabeschicht aus wieder rückwärts durch das gesamte Netzwerk gereicht (*propagiert*). Mit Hilfe der mathematischen Kettenregel lässt sich so für jeden einzelnen Parameter berechnen, wie er angepasst werden muss (Rumelhart et al., 1986). Ein Optimierer verändert die Gewichte daraufhin gezielt, sodass sich der Fehler beim nächsten Durchlauf weiter verringert.

Zusammenfassend lässt sich festhalten, dass neuronale Netze Informationen schrittweise über mehrere Schichten hinweg verarbeiten. Durch den ständigen Abgleich ihrer Vorhersagen mit den tatsächlichen Zielwerten lernen sie kontinuierlich aus ihren Fehlern und passen ihre internen Parameter so an, dass sie im Laufe der Zeit immer genauer werden.

2.3 Reinforcement Learning

Reinforcement Learning beschreibt einen Ansatz des maschinellen Lernens, bei dem ein *Agent* (ein eigenständig lernendes System oder Programm) durch Interaktion mit einem *Environment* (seiner Umgebung) eine Strategie erlernt (Sutton and Barto, 2018). Der Begriff „Reinforcement“ (Verstärkung) bezieht sich dabei auf den Lernmechanismus, bei dem der *Agent* für seine *Actions* (Handlungen) Rückmeldungen in Form von *Rewards* (Belohnungen) erhält.

Führt eine *Action* zu einem vorteilhaften Ergebnis, wird ein positiver *Reward* vergeben, während ungünstige *Actions* entsprechend negative Belohnungen nach sich ziehen können. Durch diese wiederholten Interaktionen sammelt der *Agent* Erfahrungen und passt sein Verhalten schrittweise an, um die Summe der Belohnungen über die Zeit zu maximieren.

2.3.1 Der Interaktionskreislauf: Agent, Environment und Action

Das Grundprinzip des *Reinforcement Learning* lässt sich als ein fortlaufender Dialog zwischen zwei Akteuren beschreiben: dem *Agenten* und seiner Umgebung (*Environment*). Der *Agent* übernimmt dabei die Rolle des Lernenden, der eigenständig Entscheidungen trifft, um ein bestimmtes Ziel zu erreichen. Das *Environment* hingegen stellt die Welt dar, in der sich der *Agent* bewegt. Es umfasst alle äußeren Gegebenheiten und Regeln, auf die der *Agent* keinen direkten Einfluss hat.

Dieser Austausch findet in aufeinanderfolgenden Schritten statt. In jedem Schritt betrachtet der *Agent* die aktuelle Situation und wählt eine Handlung (*Action*) aus den verfügbaren Möglichkeiten, dem sogenannten *Action Space*, aus. Das *Environment* reagiert auf diese Wahl, verändert den Zustand und liefert dem *Agenten* eine Rückmeldung (*Reward*) über den Erfolg der Aktion zurück. Formal lässt sich dieser Zyklus als Sequenz aus Zustand S, Aktion A und Belohnung R beschreiben (siehe Abbildung 2.3):

$$S_t \xrightarrow{A_t} R_{t+1}, S_{t+1} \tag{2.1}$$

Die formale Beschreibung dieses Kreislaufs kann auf eine Vielzahl von Problemen angewendet werden. In gitterbasierten Brettspielen beispielsweise übernimmt der *Agent* die Rolle eines virtuellen Spielers, während das *Environment* das Spielfeld, die Regeln und die Aktionen des Gegners umfasst.

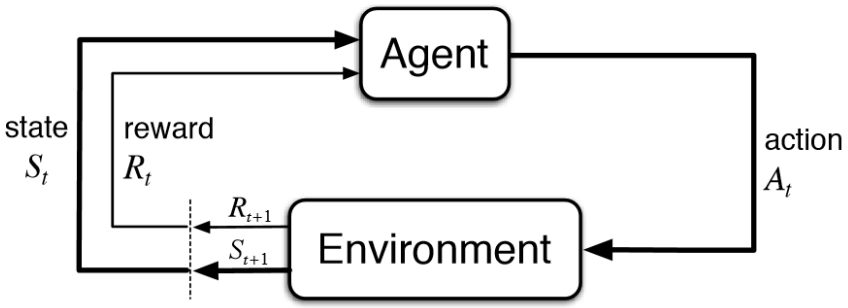


Abbildung 2.3: Der Interaktionskreislauf im Reinforcement Learning: Der Agent wählt basierend auf dem Zustand S_t eine Aktion A_t, woraufhin das Environment mit einer Belohnung R_{t+1} und einem Folgezustand S_{t+1} antwortet (entnommen aus (Sutton and Barto, 2018, S. 48)).

2.3.2 Zustandsraum und Beobachtbarkeit (State)

Der *State* (Zustand) stellt das aktuelle Abbild dar, welches der *Agent* von seiner Umgebung wahrnimmt. Er dient als Informationsgrundlage, auf deren Basis in jedem Schritt entschieden werden muss, welcher Zug am sinnvollsten ist.

In der Spieltheorie umfasst der *State* typischerweise die aktuelle Belegung des Spielfelds sowie zusätzliche Informationen über verfügbare Ressourcen (z. B. Spielsteine im Vorrat). Je nach Spieltyp verfügt der *Agent* dabei über vollständige oder unvollständige Informationen über den Gesamtzustand der Umgebung.

2.3.3 Zielerreichung: Rewards und Return

Der *Reward* (Belohnung) fungiert als zentrales Steuerungssignal, das dem *Agenten* eine unmittelbare Rückmeldung über die Qualität seiner gewählten *Action* liefert. Das fundamentale Ziel des Lernprozesses besteht darin, das Verhalten so zu optimieren, dass die Summe dieser Belohnungen über den gesamten Spielverlauf maximiert wird. Diese kumulierte Summe wird in der Fachsprache als *Return* bezeichnet.

In vielen Anwendungsfällen gibt es jedoch nicht für jeden Einzelschritt einen *Reward*. Dies ist unter anderem bei Brettspielen der Fall, bei denen ein sogenannter *Sparse Reward*-Ansatz (spärliche Belohnung) verfolgt wird. Ein *Agent* erhält hierbei erst nach Beendigung einer Partie eine finale Rückmeldung darüber, ob das angestrebte Ziel (der Sieg) erreicht wurde. Diese verzögerte Form der Belohnung erschwert den Lernprozess (*Credit Assignment Problem*), stellt aber sicher, dass der *Agent* eine robuste und unverfälschte Gewinnstrategie entwickelt.

2.3.4 Exploration-Exploitation-Dilemma

Ein zentrales Problem beim Reinforcement Learning ist das Abwägen zwischen dem Nutzen bereits gelernter Strategien und dem Entdecken neuer Handlungsweisen. Dieser Interessenkonflikt wird als Exploration-Exploitation-Dilemma bezeichnet (Sutton and Barto, 2018, Kap. 2). Falls der *Agent* ausschließlich auf bekanntes Wissen zurückgreift, lernt er nichts Neues hinzu und läuft Gefahr, auf einer zwar funktionierenden, aber nicht optimalen Taktik (lokales Optimum) zu verharren. Wenn er hingegen fortlaufend neue Aktionen ausprobiert, ohne sein gelerntes Wissen anzuwenden, können sich keine stabilen Verhaltensmuster festigen und der Lernprozess kommt nicht zum Abschluss.

In komplexen Umgebungen ist dieses Gleichgewicht besonders kritisch. Mit zunehmender Größe des Zustandsraums wächst die Anzahl der möglichen Handlungssequenzen exponentiell an. Bleiben zudem Teile der Information vor dem *Agenten* verborgen (unvollständige Information), entsteht eine ständige Unsicherheit, die eine kontinuierliche und strategisch geführte Exploration notwendig macht.

Idealerweise sollte die Bereitschaft zur Exploration im Laufe des Trainings abnehmen, damit der *Agent* seine gelernten Taktiken konsequent ausführen und verfeinern kann. Dennoch muss das System flexibel bleiben. Wenn eine Taktik, die lange gut funktioniert hat, gegen stärkere Gegner im *Self-Play* plötzlich nicht mehr ausreicht, hilft oft eine gezielte Exploration. Indem der *Agent* wieder Neues ausprobiert, kann er eingefahrene Muster aufbrechen und seine Strategie an das höhere Spielniveau anpassen (Schulman et al., 2017).

2.4 Convolutional Neural Networks (CNNs)

Während klassische, vollvernetzte Schichten (*Fully Connected Layers*) jeden Eingabewert isoliert betrachten und so die räumliche Struktur der Daten ignorieren, behalten CNNs diese zweidimensionale Anordnung bei. Das Herzstück dieser Netzwerke ist die diskrete Faltung, die in der Praxis meist als Kreuzkorrelation umgesetzt wird.

Wie in Abbildung 2.4 zu sehen ist, wandert dabei ein Filter (der sogenannte Kernel K) schrittweise über die zweidimensionalen Eingabedaten I. Bei jedem Schritt betrachtet

der Kernel einen kleinen Ausschnitt der Daten. In der Praxis handelt es sich dabei häufig um ein 3×3 großes Fenster, da dieses genau ein zentrales Feld sowie dessen acht direkte Nachbarn abdeckt und so lokale Beziehungen optimal erfasst. Die Werte innerhalb dieses Fensters werden mit den Gewichten des Kernels multipliziert und aufsummiert. Das Ergebnis dieser Rechnung wird dann als einzelner Wert in der daraus entstehenden *Feature Map* S an der Position (i, j) gespeichert.

Klassische Methoden der digitalen Bildverarbeitung verwenden oft manuell entworfene Filter (z. B. zur Kantendetektion). In modernen Reinforcement Learning (RL)-Agenten sind die Gewichte der Kernel hingegen frei lernbare Parameter. Während des Trainings werden diese Gewichte durch Backpropagation kontinuierlich optimiert. Das Netzwerk lernt dadurch völlig selbstständig, welche räumlichen Merkmale für eine erfolgreiche Spielstrategie wichtig sind. Um noch komplexere Zusammenhänge zu erfassen, lässt sich dieser Prozess auf eine dreidimensionale Faltung erweitern. Dadurch kann das System Informationen aus mehreren semantischen Kanälen gleichzeitig verarbeiten.

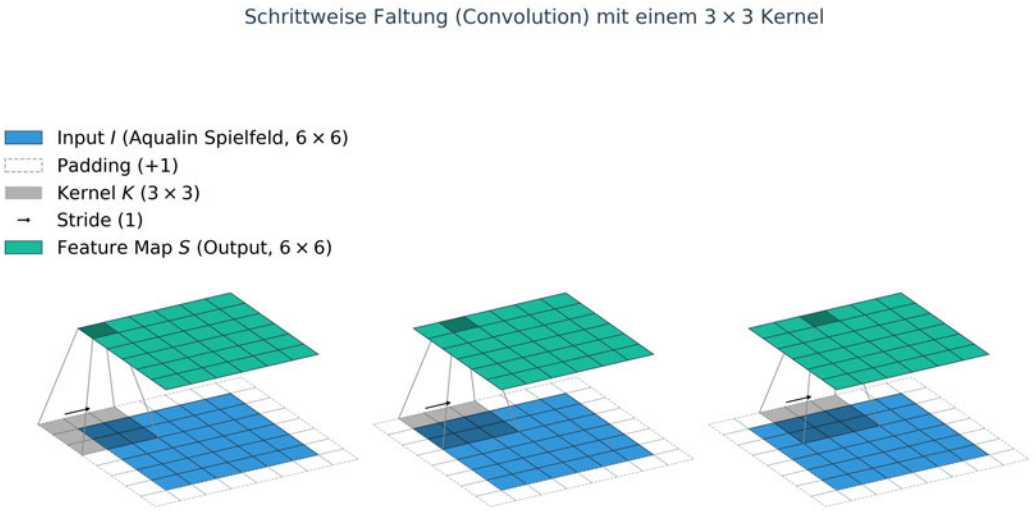


Abbildung 2.4: Isometrische Darstellung einer Convolution (3×3 Kernel, Stride 1) auf dem 6×6 *Aqualin*-Spielfeld. Durch das Padding von $+1$ behält die resultierende Feature Map exakt die geometrische Dimension des Inputs bei. (Eigene Darstellung)

Mathematisch lässt sich dieser Vorgang durch folgende Formel ausdrücken:

$$S(i,j) = (I * K)(i,j) = \sum_{m=-h}^{h} \sum_{n=-h}^{h} I(i+m, j+n) K(m,n) \tag{2.2}$$

Dabei beschreibt $h = \lfloor k/2 \rfloor$ den halben Radius des Filters der Größe $k \times k$ (bei einem typischen 3×3-Kernel ist $h = 1$). Die Indizes m und n sorgen in der Doppelsumme dafür, dass genau dieses direkte Umfeld des Punktes (i, j) Schritt für Schritt abgetastet wird.

Ein technisches Detail, das bei dieser Operation beachtet werden muss, ist das Verhalten am Rand des Spielfelds: Liegt der zentrale Punkt (i, j) direkt an der äußeren Kante, ragt das Filterfenster (wie man in Abbildung 2.4 gut an den gestrichelten Randfeldern erkennen kann) über die tatsächlichen Daten hinaus. Um zu verhindern, dass das Netzwerk versucht, auf nicht existierende Werte zuzugreifen, wird daher oft ein künstlicher Rand aus Nullen (*Padding*) um die Eingabedaten gelegt.

In der Praxis wird auf das Ergebnis dieser Rechnung meist noch ein lernbarer Bias-Term b addiert.

Um komplexe Muster zu erkennen, werden in einem CNN typischerweise mehrere solcher Faltungsschichten hintereinandergeschaltet. Abbildung 2.5 zeigt die generelle Architektur eines solchen Netzwerks.

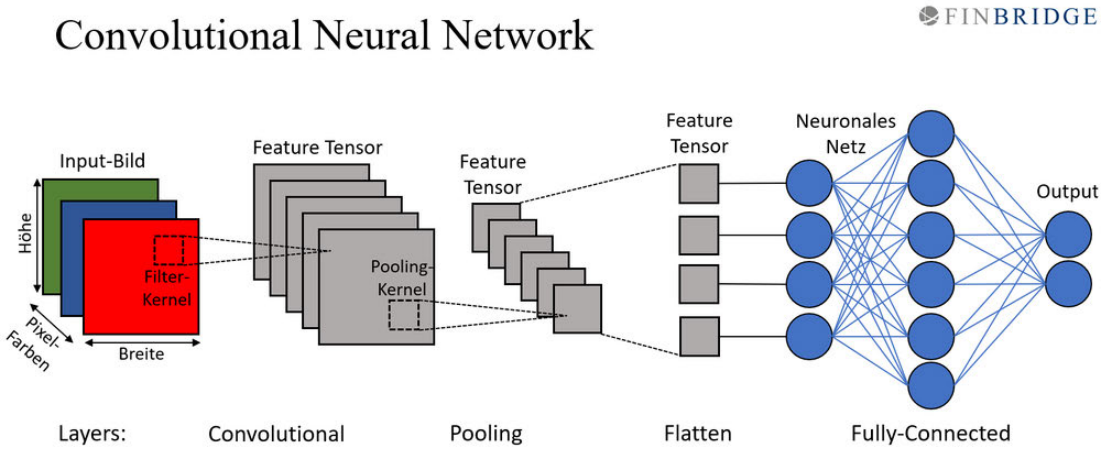


Abbildung 2.5: Typische Architektur eines Convolutional Neural Networks, bestehend aus Faltungs- (Convolutional), Pooling- und Fully-Connected-Schichten (entnommen aus Finbridge GmbH & Co. KG (2022)).

Während bei klassischen Aufgaben der Bildverarbeitung (wie in der Abbildung dargestellt) häufig *Pooling*-Schichten zur räumlichen Datenkompression zwischen die Faltungen geschaltet werden, wird bei abstrakten Brettspielen wie *Aqualin* bewusst darauf verzichtet. Pooling würde die exakten Koordinaten der Spielsteine „verwischen“, was für die Bewertung der exakten Spielsituation fatal wäre. Stattdessen wird die räumliche Auflösung über alle Convolutional-Schichten hinweg beibehalten. Erst am Ende des Netzwerks erfolgt das *Flattening* – das Umwandeln der mehrdimensionalen Matrizen in einen eindimensionalen Vektor – um die extrahierten Merkmale an die klassifizierenden *Fully-Connected Layers* zu übergeben.

Speziell bei gitterförmigen Strukturen, wie einem Spielbrett, bietet der Einsatz von Convolutional-Schichten drei entscheidende Vorteile (Goodfellow et al., 2016, Kap. 9):

1. **Lokale Konnektivität**: Jedes Neuron verarbeitet ausschließlich einen eng begrenzten Bereich des Spielfelds, das sogenannte *Receptive Field*. Dadurch erfasst das Netzwerk lokale Zusammenhänge, wie direkt benachbarte Spielsteine oder die Ränder einer Gruppe, äußerst präzise.

2. **Gewichtsteilung (*Parameter Sharing*)**: Das Netzwerk wendet denselben Filter auf alle Positionen des Spielfelds an. Dies reduziert die Anzahl der zu optimierenden Parameter drastisch, da nicht für jede Position neue Gewichte berechnet werden. Folglich beschleunigt sich das Training spürbar und die statistische Effizienz steigt.

3. **Translationsinvarianz**: Aufgrund des einheitlichen Filters erkennt das Netzwerk räumliche Muster unabhängig von ihrer absoluten Position. Eine zusammenhängende Gruppe von Steinen identifiziert das System somit stets korrekt, unabhängig davon, ob sie am Rand oder im Zentrum des Spielfelds liegt.

2.4.1 Faltungsparameter: Padding und Stride

Die Dimensionierung der resultierenden *Feature Map* S wird maßgeblich durch zwei Hyperparameter gesteuert: *Padding* und *Stride*. Das *Padding* beschreibt das Hinzufügen künstlicher Randwerte (meist Nullen) um den Input. Dies ist notwendig, da das Zentrum eines Kernels ohne Padding die Randpixel des Spielfelds nicht direkt erreichen könnte, was zu einer Reduktion der Dimensionen führen würde.

In dieser Arbeit wird *Half-Padding* (auch *Same-Padding* genannt) verwendet. Dabei wird bei einem Kernel der Größe $k = 3$ ein Rand von $p = 1$ Nullen hinzugefügt. Dies stellt sicher, dass die Feature Map exakt dieselbe räumliche Dimension wie der Input behält, wodurch die geometrische Struktur der Daten über alle Netzwerkschichten hinweg konsistent abgebildet bleibt. Der *Stride* (Schrittweite) definiert den Versatz des Filters zwischen zwei Operationen. Ein Stride von 1 stellt dabei sicher, dass der Filter pixelweise über das Feld gleitet und somit keine Informationen übersprungen werden (siehe Abbildung 2.4).

2.4.2 Batch Normalization

Ein zentrales Problem beim Training tiefer neuronaler Netze ist der sogenannte *Internal Covariate Shift*. Dieser beschreibt den Effekt, dass sich während des Lernprozesses die Verteilung der Eingangswerte (Aktivierungen) in den tieferen Schichten ständig verändert, da sich die Gewichte der davorliegenden Schichten laufend anpassen. Für die darauffolgenden Schichten stellt dies ein „bewegliches Ziel“ dar: Sie müssen ihre eigenen Parameter ständig an eine neue Datenverteilung anpassen, was den gesamten Lernprozess instabil macht und die Konvergenz (das Annähern an ein optimales Fehler-Minimum) massiv verlangsamt.

Um dieses Problem zu lösen, wird Batch Normalization (BN) eingesetzt (Ioffe and Szegedy, 2015). BN fungiert als stabilisierende Zwischenschicht, die die Aktivierungen innerhalb eines Trainings-Batches auf einen Mittelwert von Null und eine Varianz von Eins normiert. Im Anschluss erlaubt eine lernbare lineare Transformation dem Netzwerk jedoch, diese Werte bei Bedarf wieder gezielt zu verschieben und zu skalieren.

Dieser Mechanismus bewirkt zwei entscheidende Vorteile:

1. **Höhere Lerngeschwindigkeit**: Da die Schichten mit stabileren Datenverteilungen arbeiten, kann eine deutlich höhere Lernrate gewählt werden, ohne dass das Training divergiert.

2. **Robustheit gegen Gradientenprobleme**: Durch die Normierung wird sichergestellt, dass die Werte nicht in Bereiche der Aktivierungsfunktionen (wie dem Sättigungsbereich von Sigmoid-Funktionen) abrutschen, in denen die Gradienten (Fehlersignale) verschwinden würden.

Für komplexe Lernprobleme ist BN essentiell, um trotz tiefer Architekturen und potenziell schwacher Gradientensignale eine stabile und effiziente Optimierung zu gewährleisten.

2.5 Proximal Policy Optimization (PPO)

Um Agenten in komplexen Umgebungen stabil zu trainieren, hat sich der PPO Algorithmus als Standard etabliert. PPO gehört zur Klasse der *Policy Gradient* Methoden und wurde mit dem Ziel entwickelt, die Trainingsstabilität und Dateneffizienz gegenüber klassischen Verfahren deutlich zu verbessern (Schulman et al., 2017).

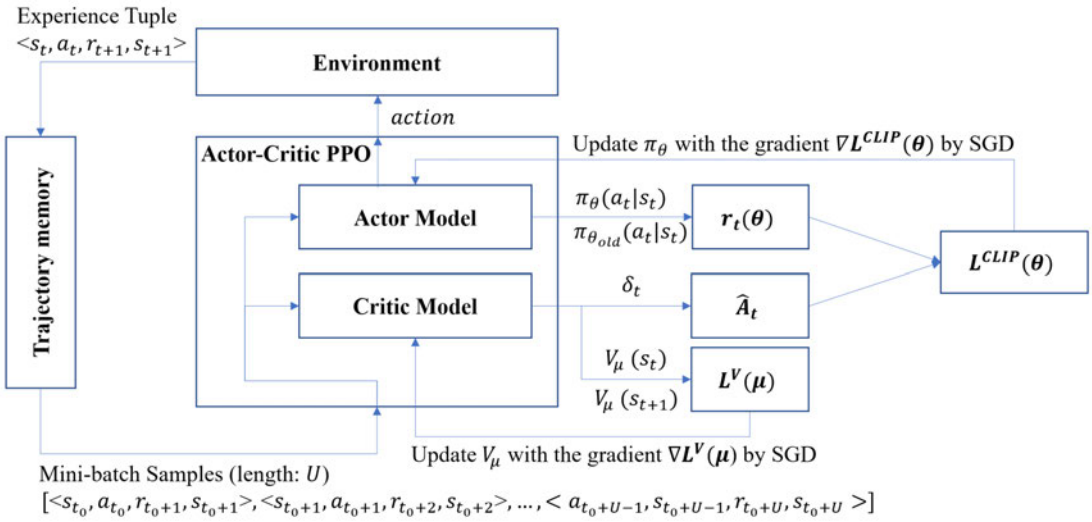


Abbildung 2.6: Systematik des Actor-Critic PPO-Algorithmus: Visualisierung des Zusammenspiels zwischen dem Actor (Policy π) und dem Critic (Value-Funktion V) sowie der Berechnung des Clipped Surrogate Objectives L^{CLIP} (entnommen aus (Lim et al., 2020, Fig. 3)).

2.5.1 Ablauf: Datensammlung und Training

Im Gegensatz zu sehr einfachen Algorithmen, die unmittelbar nach jedem ausgeführten Schritt lernen, arbeitet PPO sequenziell in zwei strikt getrennten Phasen (wie in Abbildung 2.6 ersichtlich). Zunächst interagiert der *Agent* mit der Umgebung (*Environment*) und sammelt Erfahrungen in Form von sogenannten *Experience Tuples* $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$. Diese werden in einem Zwischenspeicher (*Trajectory Memory*) abgelegt. In dieser Phase findet noch kein Lernen statt. Erst wenn der Speicher vollständig gefüllt ist, stoppt die Datensammlung und die Trainingsphase beginnt. Die gespeicherten Erfahrungen

werden nun genutzt, um die neuronalen Netze zu aktualisieren. Da PPO ein On-Policy-Algorithmus ist, wird der Speicher nach dem Training komplett geleert, um im nächsten Durchlauf wieder ausschließlich frische, mit der neuen Strategie generierte Erfahrungen zu sammeln. Dies ist zwingend notwendig, da das Netzwerk sonst aus veralteten Daten lernen würde, die nicht mehr seiner aktuellen Spielstärke entsprechen. Durch das konsequente Verwerfen alter Erfahrungen wird verhindert, dass die Schätzungen des Critic-Netzwerks verfälscht werden, was eine wesentliche Voraussetzung für den stabilen und zielgerichteten Lernfortschritt von PPO darstellt.

2.5.2 Actor-Critic-Architektur und die Advantage-Funktion

Der Algorithmus nutzt intern zwei separate Modelle (oft als zwei Köpfe desselben neuronalen Netzwerks realisiert), deren Aufgaben sich anschaulich mit einem Spieler und einem Kommentator vergleichen lassen (Mnih et al., 2016):

- **Actor Model (Der Spieler):** Dieser Teil wählt die Aktionen aus. Für einen Zustand s_t berechnet das Modell die Policy $\pi_\theta(a_t|s_t)$ – also die Wahrscheinlichkeit für die verschiedenen möglichen Züge.
- **Critic Model (Der Kommentator):** Dieser Teil beobachtet das Geschehen und schätzt den Wert $V(s_t)$ eines Zustands ab. Er bewertet fortlaufend, wie vorteilhaft die aktuelle Situation für den *Agenten* insgesamt ist.

Nachdem der Actor gehandelt hat, kommt die **Advantage-Funktion** $\hat{A}_t$ (der Vorteil) ins Spiel. Sie vergleicht die tatsächlich erhaltene Belohnung aus der Aktion mit der Vorhersage des Critics. Die Advantage gibt exakt an, ob und wie stark eine Aktion besser (oder schlechter) war, als der Critic es ursprünglich erwartet hatte. War die Aktion besser als vorhergesagt ($\hat{A}_t > 0$), wird sie vom Algorithmus künftig stärker priorisiert.

2.5.3 Das Herzstück: Die Clipped Surrogate Objective

Ein zentrales Problem im Reinforcement Learning ist das sogenannte *Catastrophic Forgetting* (katastrophales Vergessen). Wenn ein Netzwerk seine Gewichte nach einer Episode zu drastisch anpasst, verlernt es oft bereits mühsam erlerntes, sinnvolles Verhalten. Um dies zu verhindern, darf die neue Policy π_θ nicht zu stark von der alten Policy $\pi_{\theta_{old}}$ abweichen. PPO misst diese Abweichung durch das Wahrscheinlichkeits-Verhältnis (Ratio) $r_t(\theta)$:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.3}$$

Die Verlustfunktion des Actors wird so konstruiert, dass sie zu große Strategie-Sprünge blockiert. Dies wird durch eine Clipping-Funktion erreicht, die das Verhältnis in einem sicheren Intervall $[1-\epsilon, 1+\epsilon]$ (typischerweise $\epsilon = 0,2$) abschneidet:

$$L^{CLIP}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.4}$$

Die min-Funktion wählt stets den konservativeren Schätzwert. Weicht die neue Strategie zu weit ab und stößt an das vorgegebene Limit, liefert das Netzwerk keinen Lern-Gradienten mehr für diese Aktion. Das Update wird somit begrenzt, und der *Agent* verändert sein Verhalten stets nur in kleinen, sicheren Schritten.

2.5.4 Dateneffizienz durch Mini-Batches und Epochen

Dank der Sicherheitsvorrichtung durch die Clip-Funktion bietet PPO einen massiven Vorteil bei der Dateneffizienz: Die gesammelten Erfahrungen aus dem Trajectory Memory können mehrfach zum Lernen verwendet werden, ohne dass die Policy instabil wird. Der Algorithmus unterteilt die Daten in kleine Päckchen (*Mini-batch Samples*) und trainiert in mehreren Durchläufen (Epochen) mit denselben Daten. Anstatt einen Datensatz nach einem Update direkt zu verwerfen, lernt der *Agent* also mehrfach aus der gleichen Situation. Hierbei ist jedoch ein Balanceakt erforderlich: Trainiert der *Agent* zu häufig mit denselben Daten, droht *Overfitting* (Überanpassung). Der *Agent* fixiert sich dann zu stark auf spezifische Verhaltensmuster aus dem aktuellen Batch und verliert die Fähigkeit, in neuen Situationen das allgemeine Optimum zu finden.

2.5.5 Exploration und Entropie-Bonus

In strategisch komplexen Umgebungen wie *Aqualin* besteht oft die Gefahr, dass der *Agent* sehr früh eine mittelmäßige Strategie findet (z. B. immer nur eine bestimmte Farbe sammeln) und fortan aufhört, nach besseren Alternativen zu suchen. Er konvergiert vorzeitig auf ein lokales Optimum. Um den *Agenten* neugierig zu halten, addiert PPO zur Verlustfunktion einen sogenannten *Entropie-Bonus* (L^{ENT}). Die Entropie ist in diesem Kontext ein Maß für die Unsicherheit oder Unberechenbarkeit des *Agenten*. Ein hoher Entropie-Bonus belohnt den *Agenten* dafür, dass er eine breitere Wahrscheinlichkeitsverteilung beibehält. Er legt sich also nicht zu 100 % auf einen bestimmten Zug fest, sondern probiert bewusst verschiedene Handlungsweisen aus (*Exploration*). Erst wenn er im späten Trainingsverlauf absolut sicher ist, dass ein Zug der beste ist, darf die Entropie sinken.

Dennoch muss das System flexibel bleiben. Wenn eine Taktik, die lange gut funktioniert hat, gegen stärkere Gegner im Self-Play plötzlich nicht mehr ausreicht, hilft oft eine gezielte Exploration. Indem der Agent wieder Neues ausprobiert, kann er eingefahrene Muster aufbrechen und seine Strategie an das höhere Spielniveau anpassen (Silver et al., 2017).

2.5.6 Umgang mit verzögerten Belohnungen (Sparse Rewards)

Wie in Abschnitt 2.5.2 zur Advantage-Funktion beschrieben, benötigt der Critic nach jeder Aktion ein Feedback in Form einer Belohnung (r_t), um beurteilen zu können, ob der Zug gut war. Bei abstrakten Brettspielen wie *Aqualin* tritt hierbei jedoch ein massives Problem auf: Die Belohnung wird im Environment bewusst erst ganz am Ende einer Partie (basierend auf dem Endergebnis) ausgeschüttet. Solche *verzögerten Belohnungen* (Sparse Rewards) bedeuten, dass der *Agent* während des Spiels völlig im Dunkeln tappt und erst nach Dutzenden von Zügen erfährt, ob er gewonnen hat oder nicht.

Um diesem Problem zu begegnen, nutzt PPO Techniken wie die Generalized Advantage Estimation (GAE) (Schulman et al., 2015). GAE ermöglicht es, dem Actor auch dann schon fundiertes Feedback zu geben, wenn die echte Belohnung noch aussteht. Hierfür wird als zentraler Zwischenschritt zunächst der sogenannte Temporal Difference (TD)-Fehler δ_t (Temporal-Difference-Error) ermittelt:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.5}$$

Hierbei ist r_t die (oft ausbleibende) Belohnung aus der Umgebung und γ der Diskontierungsfaktor. Der entscheidende mathematische Kniff liegt in den Termen des Critics ($V(s_{t+1})$ und $V(s_t)$): Fehlt die echte Belohnung r_t, stützt sich die Formel einfach auf die eigenen *Schätzungen* des Critics für den aktuellen und den darauffolgenden Zustand. Der Critic behilft sich also mit seiner eigenen Prognose darüber, ob der durch die Aktion neu erreichte Zustand vorteilhafter ist als der alte.

Die tatsächliche Advantage-Schätzung $\hat{A}_t$ aggregiert diese Schätzfehler anschließend nicht nur für den nächsten Schritt, sondern blickt (exponentiell abgewichtet) weit in die Zukunft l:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{2.6}$$

Der Hyperparameter $\lambda \in [0, 1]$ steuert dabei die Weitsicht und Glättung dieser Schätzung. Durch diesen vorausschauenden Ansatz erhält der Actor nach jedem Zug eine sofortige Bewertung, selbst wenn das eigentliche Spielende noch in weiter Ferne liegt.

2.6 Metriken zur Leistungsbewertung

Um den Erfolg und die Lernqualität eines RL-*Agenten* mathematisch beurteilen zu können, kommen verschiedene Kennzahlen zum Einsatz.

2.6.1 Performance-Metriken

Um die absolute Spielstärke zu messen, werden typischerweise Metriken herangezogen, die aus den finalen Spielergebnissen resultieren:

- **Gewinnrate (Win Rate)**: Die primäre Metrik, die den prozentualen Anteil der gewonnenen Spiele gegen definierte Benchmarks (wie Zufallsgegner oder Heuristiken) angibt.
- **Point Margin (Gewinnabstand)**: Die absolute Punktdifferenz zwischen dem *Agenten* und seinem Gegner am Ende einer Partie. Eine dauerhaft positive Margin signalisiert eine klare Spieldominanz.
- **Mean Reward (Durchschnittliche Belohnung)**: Die durchschnittlich akkumulierte Belohnung, die der *Agent* pro Episode erhält. Bei *Sparse Rewards* steht dieser Wert in direktem Zusammenhang mit der Gewinnrate, erlaubt aber oft eine differenziertere Einschätzung der Spielstärke.

2.6.2 Trainings-Metriken

Neben dem Spielerfolg geben interne Netzwerkvariablen Auskunft über die Stabilität des Lernprozesses und das Verhalten des PPO-Algorithmus:

- **Policy Entropy (Entropie)**: Ein Maß für die Unberechenbarkeit der gewählten Aktionen (Mnih et al., 2016). Eine hohe Entropie bedeutet, dass der *Agent* stark exploriert und verschiedene Züge ausprobiert. Im Laufe des Trainings sollte die Entropie idealerweise sinken, was einen Übergang zur *Exploitation* (gezielte Ausnutzung gelernter Strategien) signalisiert.
- **Value Loss**: Beschreibt den Vorhersagefehler des Critic-Netzwerks, typischerweise berechnet als quadratischer Fehler (Mean Squared Error) zwischen der Vorhersage und dem tatsächlichen Return (Sutton and Barto, 2018). Ein sinkender Value Loss belegt, dass der Critic ein immer tieferes Verständnis für die Qualität von Spielsituationen entwickelt und den tatsächlichen Ausgang der Partie besser prognostizieren kann.
- **Explained Variance**: Diese Kennzahl gibt an, wie viel der Varianz der tatsächlichen Returns durch die Vorhersagen des Critics erklärt werden kann. Ein Wert nahe 1 deutet auf einen hochpräzisen Critic hin.
- **Kullback-Leibler (KL)-Divergenz**: Die Kullback-Leibler-Divergenz misst die Änderung der Policy zwischen zwei Updates. Eine zu hohe Divergenz signalisiert Instabilität, weshalb PPO diese durch sein *Clipping* in Schach hält (Schulman et al., 2017).

Diese theoretischen Grundlagen bilden die Basis für den im folgenden Kapitel beschriebenen Entwurf des *Aqualin-Agenten*.

3 Stand der Technik

Dieses Kapitel beleuchtet den aktuellen Forschungsstand im Bereich des Deep Reinforcement Learning (DRL) bei kompetitiven Brettspielen. Die hier vorgestellten Systeme und Architekturen zeigen auf, wie moderne KIs heute aufgebaut sind, und dienen als praktisches Vorbild für die Entwicklung des *Aqualin-Agenten*.

3.1 Historische Entwicklung der KI in Brettspielen

Die Faszination für Maschinen, die uns Menschen in strategischen Spielen übertreffen, reicht weit über die heutige Ära der Computer hinaus. Bereits im Jahr 1770 erregte der sogenannte **Schachtürke** (*Mechanical Turk*) weltweites Aufsehen. Dabei handelte es sich um einen Apparat, der scheinbar völlig autonom Schach auf hohem Niveau spielte (Standage, 2002). Wie in Abbildung 3.1 zu sehen ist, offenbarten geöffnete Schranktüren damals ein komplexes mechanisches Innenleben. Ein Lineal in der unteren rechten Ecke des Kupferstichs verdeutlicht die realen Dimensionen der Konstruktion.

In Wahrheit war das Ganze jedoch eine geschickte Täuschung. Im Inneren des Gehäuses saß ein verborgener menschlicher Spieler, der die Arme der Figur steuerte. Diese frühe Illusion unterstreicht wunderbar die historische Bedeutung des Problems: Die Erschaffung einer echten künstlichen Intelligenz galt lange Zeit als geradezu magische, unlösbare Herausforderung. Erst durch die rasante Entwicklung digitaler Rechensysteme konnte dieser Traum schließlich von einer mechanischen Illusion in die algorithmische Realität umgesetzt werden.

Ein früher Pionier des Reinforcement Learning: TD-Gammon. Bereits im Jahr 1992, also noch vor den großen Erfolgen der klassischen Suchalgorithmen im Schach, demonstrierte Gerald Tesauro mit *TD-Gammon*, dass neuronale Netze das Brettspiel Backgammon auf Weltklasseniveau spielen können. Das Bemerkenswerte daran: Das System lernte die Strategien nicht durch menschliche Vorgaben, sondern trainierte durch sogenanntes *Temporal Difference Learning* ausschließlich gegen sich selbst (Tesauro, 1995). Zwar funktionierte dieser frühe Reinforcement-Learning-Ansatz bei Backgammon hervorragend, dies hing jedoch stark mit der stochastischen Natur des Spiels (durch das Würfeln) zusammen. Bei deterministischen Spielen wie Schach oder Go scheiterten ähnliche Ansätze damals kläglich. Die neuronalen Netze blieben immer wieder in Schleifen stecken oder scheiterten schlichtweg an der gigantischen Tiefe der Suchbäume.

Die Blütezeit der Heuristiken: Deep Blue. Weil selbstlernende Netze bei deterministischen Spielen lange Zeit auf der Stelle traten, dominierten stattdessen klassische, hart kodierte Algorithmen die Forschung. Der absolute Meilenstein dieser Ära war das Jahr

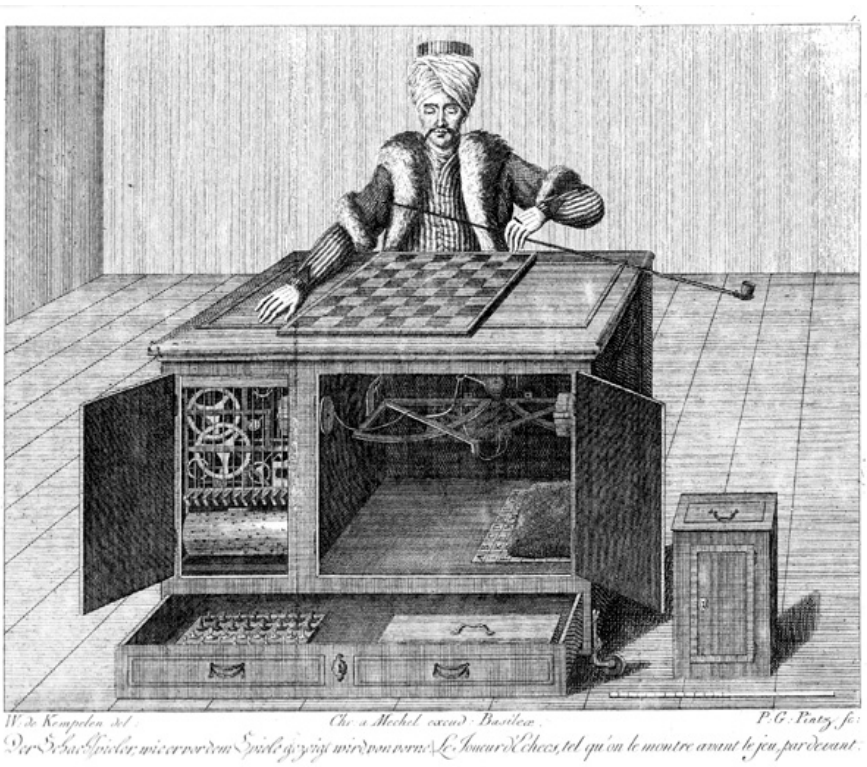


Abbildung 3.1: Kupferstich des Schachtürken von Wolfgang von Kempelen. Das Bild zeigt die geöffneten Schranktüren sowie die Mechanik im Inneren. Ein Lineal in der unteren rechten Ecke dient als Maßstabangabe (Quelle: Wikimedia Commons).

1997. Damals gelang es dem IBM-Computer *Deep Blue*, den amtierenden Schachweltmeister Garri Kasparow in einem regulären Turnier zu besiegen (Campbell et al., 2002). Das System verließ sich dabei primär auf rohe Rechengewalt. Deep Blue durchsuchte die möglichen Spielverläufe mit dem *Minimax*-Algorithmus, welcher stets versucht, den eigenen Gewinn zu maximieren, während er davon ausgeht, dass der Gegner optimal dagegenhält. Um bei dieser gigantischen Suche nicht zu langsam zu werden, half das *Alpha-Beta-Pruning* dabei, offensichtlich schlechte Züge frühzeitig abzuschneiden und gar nicht erst weiter zu berechnen. Die eigentliche Beurteilung der Brettstellungen lernte die Maschine jedoch nicht selbst. Stattdessen bewertete sie das Spielfeld anhand komplexer Regeln (sogenannten Heuristiken), die menschliche Großmeister zuvor manuell festgelegt hatten. Die „Intelligenz“ des Systems war somit hart kodiert und verließ sich vollständig auf menschliches Expertenwissen.

Der Wendepunkt: AlphaGo. Im Jahr 2016 gelang DeepMind mit *AlphaGo* der Durchbruch im Spiel Go (Silver et al., 2016). Im Gegensatz zu *Deep Blue* nutzte *AlphaGo* tiefe neuronale Netze zur Mustererkennung. In dieser ersten Version wurde das System jedoch noch mit Millionen von Zügen aus menschlichen Partien vortrainiert (*Supervised Learning*), bevor es durch Self-Play weiter optimiert wurde. Dies bildete die Brücke zum

heutigen Standard des reinen Reinforcement Learning.

3.2 Das AlphaZero-Paradigma

Der bedeutendste Meilenstein in der modernen Spiel-KI wurde durch das System **AlphaZero** von DeepMind gesetzt (Silver et al., 2017). Im Gegensatz zu klassischen Systemen wie *Deep Blue*, die auf handgefertigten Bewertungsfunktionen und massiver Rechenpower für die Suche basierten, erlernte *AlphaZero* Spiele wie Schach, Shogi und Go ausschließlich durch Self-Play (*Tabula Rasa*).

Zentral für diesen Ansatz ist die Nutzung eines **Deep Residual Neural Network**, das als „Two-Headed Architecture“ bekannt ist. Ein gemeinsamer Feature-Extraktor (Backbone) verarbeitet den Spielzustand und teilt sich anschließend in zwei spezialisierte Ausgabeschichten auf:

1. **Policy Head**: Berechnet eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen (π).
2. **Value Head**: Schätzt die Gewinnwahrscheinlichkeit des aktuellen Zustands (v).

Die technologische Basis: Residual Networks. Der Erfolg dieses Backbones basiert maßgeblich auf der Architektur der *Residual Networks* (ResNet) (He et al., 2016). Vor deren Einführung stießen tiefe neuronale Netze an eine Optimierungsgrenze: Mit zunehmender Anzahl der Schichten sank die Genauigkeit (*Degradation Problem*), da die Gradienten während des Trainings gegen Null strebten (*Vanishing Gradient Problem*). ResNets lösen dies durch sogenannte *Skip-Connections* (oder *Shortcuts*). Dabei wird der Eingangswert einer Schicht am eigentlichen Faltungsvorgang vorbeigeschleust und direkt zum Ergebnis addiert. Mathematisch lernt das Netzwerk somit nur noch die Differenz (*Residuum*) zum vorherigen Zustand. Dies erlaubt das Training von extrem tiefen Architekturen mit hunderten von Schichten, was für die komplexe Mustererkennung in Spielen wie Go oder Schach unerlässlich ist.

Abbildung 3.2 illustriert die genannte *Two-Headed Architecture*. In der oberen Hälfte (a) ist der Self-Play-Prozess zu sehen, bei dem das System gegen sich selbst spielt und so kontinuierlich neue Zustände (s_t) sowie Zielwerte für die Wahrscheinlichkeiten (π) und den Spielausgang (z) generiert. Die untere Hälfte (b) zeigt das eigentliche Training: Das neuronale Netz nimmt einen Zustand s_t als Eingabe, verarbeitet ihn durch tief gestaffelte Schichten (das Residual-Backbone) und spaltet sich am Ende in zwei Köpfe auf, um Vorhersagen für die Policy ($\boldsymbol{p}$) und den Value (v) zu treffen. Diese Ausgaben werden wiederum direkt mit den im Self-Play generierten Werten abgeglichen, um das Netz zu optimieren.

Wie diese Vorhersagen in der Praxis genutzt werden, um Spielzüge zu planen, verdeutlicht Abbildung 3.3 anhand des Monte Carlo Tree Search (MCTS)-Suchprozesses. Während der Auswahlphase (a) wandert der Algorithmus den bisher erkundeten Spielbaum hinab, geleitet von den Policy-Vorhersagen des Netzes. Sobald ein neuer Blattknoten erreicht wird (b), bewertet das neuronale Netz diesen Zustand (V) und berechnet Wahrscheinlichkeiten für weitere mögliche Züge (P). Diese Einschätzung wird anschließend (c) den Baum hinaufpropagiert (*Backup*), um die Bewertungen der darüberliegenden Knoten

(Q) zu aktualisieren. Nach vielen solcher Simulationen wird schließlich (d) der finale Zug im tatsächlichen Spiel ausgeführt. Dieser hybride Ansatz aus tiefer Mustererkennung und vorausschauender Suche ermöglicht es dem System, auch in extrem tiefen Suchbäumen präzise Entscheidungen zu treffen.

Der in dieser Arbeit entwickelte *Aqualin-Agent* orientiert sich exakt an diesem bewährten *Two-Headed*-Netzwerk-Paradigma von *AlphaZero* und adaptiert die strikte Trennung in Strategie (Actor) und Zustandsbewertung (Critic).

Die nächste Stufe: MuZero. Den derzeitigen Stand der Technik markiert der **MuZero**-Algorithmus (Schrittwieser et al., 2020). Während *AlphaZero* noch Kenntnis über die exakten Spielregeln (den Simulator) benötigt, lernt MuZero ein internes Modell der Umgebungsdynamik. Er sagt nicht nur Aktionen und Werte voraus, sondern auch, wie sich der Zustand bei einer Aktion verändern wird. Dies beweist, dass moderne KI-Systeme in der Lage sind, komplexe Logiken wie Brettspiele rein aus der Beobachtung zu meistern, ohne jemals explizit die Regeln gelesen zu haben.

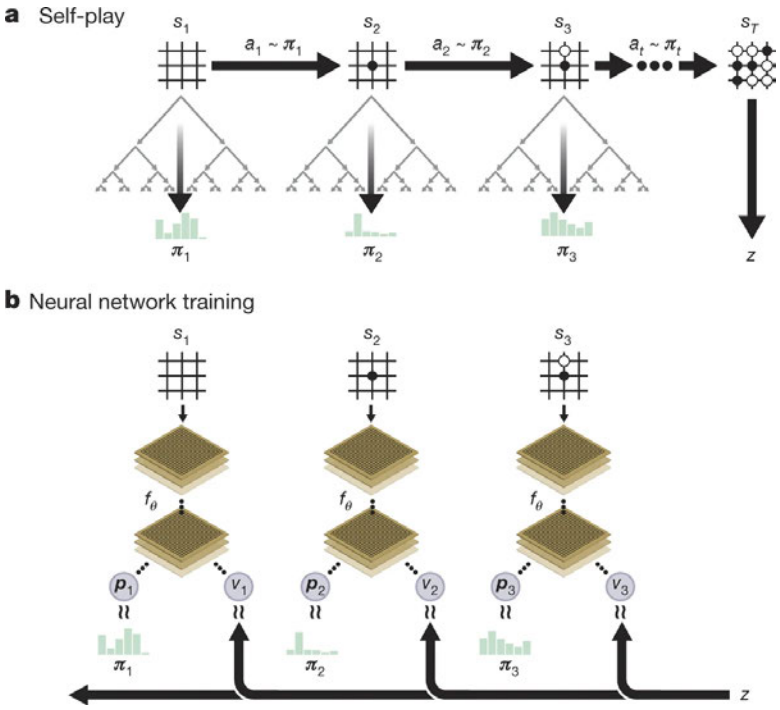


Abbildung 3.2: Die „Two-Headed“ Architektur von AlphaZero: Ein gemeinsamer Merkmalsextraktor verzweigt in eine Policy- und eine Value-Ausgabe (entnommen aus Silver et al. (2017)).

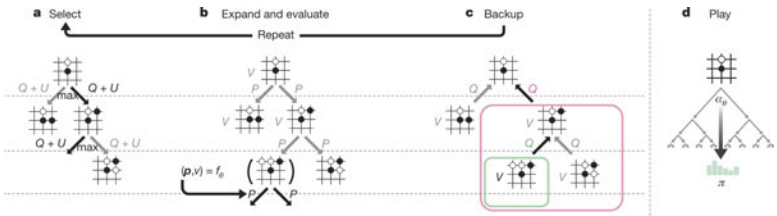


Abbildung 3.3: Visualisierung des MCTS-Prozesses: (a) Selektion basierend auf der Upper Confidence Bound (UCB), (b) Expansion und Evaluation durch das neuronale Netz, (c) Backpropagation der Werte (entnommen aus Silver et al. (2017)).

3.3 Repräsentation durch Semantic Vision

Ein zentraler Paradigmenwechsel im modernen DRL ist der Verzicht auf manuell extrahierte Merkmale (*Feature Engineering*) (Silver et al., 2017). Früher mussten Entwickler der KI wichtige Spielkonzepte wie schwache Bauernstrukturen im Schach oder verbundene Steingruppen im Go mühsam als feste Regeln einprogrammieren. Stattdessen betrachten neuronale Netze heute das Spielfeld direkt wie eine Art digitales Bild. Mathematisch ausgedrückt handelt es sich dabei um einen räumlichen Tensor.

Der Stand der Technik bei gitterbasierten Spielen ist die Nutzung von **semantischen Kanälen** (One-Hot-Encoding auf Gitterebene). Dabei wird für jede Eigenschaft eines Spielsteins (Farbe, Typ) eine eigene binäre Ebene erstellt. Dieser Ansatz wurde prominent in *AlphaGo* genutzt (z. B. separate Ebenen für schwarze und weiße Steine) (Silver et al., 2016) und findet sich in Form von mehrkanaligen Eingabe-Tensoren auch in modernen Engines wie *Leela Chess Zero* (Leela Chess Zero Team, 2021) wieder.

Der Vorteil dieser Repräsentation liegt in der Ausnutzung der **lokalen Konnektivität** von Convolutional Neural Networks (CNNs). Durch die semantische Trennung kann das Netzwerk räumliche Muster (wie Gruppen in *Aqualin*) direkt über Faltungsfilter erkennen, ohne durch numerische Ähnlichkeiten verschiedener Kategorien (Ordinal-Bias) beeinflusst zu werden.

3.4 Invalid Action Masking

Viele strategische Brettspiele besitzen einen riesigen Pool an theoretisch denkbaren Zügen. In der programmiertechnischen Umsetzung wird der Aktionsraum oft absichtlich groß und einheitlich abgebildet, um die Architektur des Netzwerks zu vereinfachen. In der Praxis ist ein beträchtlicher Teil dieser theoretischen Ausgaben jedoch in der jeweiligen Spielsituation physikalisch unmöglich oder nach den Regeln schlichtweg verboten. Das stellt das Lernmodell vor eine Herausforderung. Wenn die KI einfach blind ausprobiert, wählt sie andauernd ungültige Aktionen.

Klassische Ansätze versuchten früher, der KI solche Fehler durch negative Belohnungen abzugewöhnen. Das System erhielt für jeden ungültigen Zug nachträglich Punktabzug. Die Forschung belegt jedoch, dass das sogenannte **Invalid Action Masking** deutlich eleganter und messbar effektiver ist (Huang and Ontañón, 2022). Anstatt das Netzwerk im Nachhinein zu bestrafen, greift der Algorithmus direkt in die Entscheidungsfindung ein. Mathematisch wird die Wahrscheinlichkeit für alle illegalen Züge vor der finalen Berechnung einfach auf null gesetzt. Dafür erhalten die entsprechenden Werte (die Logits) den Wert $-\infty$.

Dieser Ansatz bietet zwei entscheidende technische Vorteile:

- **Recheneffizienz:** Das Netzwerk verschwendet keine Kapazitäten damit, offensichtlich ungültige Züge zu erforschen.
- **Trainingsstabilität:** Die mathematische Fehlerberechnung beim Lernen wird nicht durch ständige Regelverstöße verfälscht.

Als Resultat dieser beiden Faktoren lernt die KI insgesamt wesentlich schneller und stabiler, da sie sich ab der ersten Sekunde voll auf das Erlernen der eigentlichen Spieltaktik

konzentrieren kann.

3.5 Multi-Agent Trainingsmethodik

Eine der größten Herausforderungen im kompetitiven RL ist die Instabilität beim Lernen. Trainiert ein *Agent* nur gegen einen festen Gegner, nutzt er oft gezielt dessen Schwachstellen aus (*Overfitting*). Er entwickelt auf diese Weise keine robuste Generalstrategie.

Die moderne Lösung für dieses Problem ist das **League Training** oder **Self-Play** (Silver et al., 2017). Dabei spielt der *Agent* gegen alte Versionen von sich selbst. Das erzeugt ein stetiges Wettrüsten. Der *Agent* muss kontinuierlich neue Taktiken lernen, um seine eigenen, immer stärker werdenden Vorversionen zu besiegen. Frameworks wie *PettingZoo* (Terry et al., 2021) haben diese Methodik für die Wissenschaft standardisiert. Sie bilden auch die Basis für das Training in dieser Arbeit.

3.6 Curriculum Learning

Ein ergänzender Standard zur Effizienzsteigerung im Reinforcement Learning ist das **Curriculum Learning** (Bengio et al., 2009). Das Konzept orientiert sich am menschlichen Lernprozess, bei dem komplexe Aufgaben nicht sofort in ihrer Gesamtheit, sondern über eine Abfolge von ansteigenden Schwierigkeitsgraden vermittelt werden.

Im Kontext von kompetitiven Spielen bedeutet dies, dass ein *Agent* nicht unmittelbar gegen einen perfekten Gegner oder in der vollen Komplexität des Spiels trainiert. Stattdessen wird die Umgebung so modifiziert, dass der *Agent* zunächst einfache Teilprobleme löst (z. B. verkürzte Partien oder Spiele gegen stark explorierende Gegner). Sobald eine definierte Erfolgsquote erreicht wird, erhöht das Curriculum den Schwierigkeitsgrad automatisch. Dies verhindert, dass der *Agent* in frühen Trainingsphasen durch ausbleibende Belohnungen (*Sparse Rewards*) stagniert, und führt zu einer robusteren Generalisierung der erlernten Strategien.

4 Konzept

In diesem Kapitel wird das theoretische Konzept des *Agenten* beschrieben. Dabei liegt der Fokus auf der Modellierung des Problems sowie den Design-Entscheidungen, die unabhängig von der konkreten softwaretechnischen Umsetzung getroffen wurden.

4.1 Systemübersicht

Bevor die einzelnen Komponenten im Detail beschrieben werden, gibt Abbildung 4.1 einen Überblick über das Gesamtsystem. Der Lernprozess wird durch eine zyklische Interaktion zwischen dem *Agenten* (*Poseidon*) und der Spielumgebung bestimmt. Übergeordnete Steuerungseinheiten wie das Curriculum Learning und das League Training (Self-Play) beeinflussen dabei die Konfiguration der Umgebung, um einen kontinuierlichen Lernfortschritt sicherzustellen.

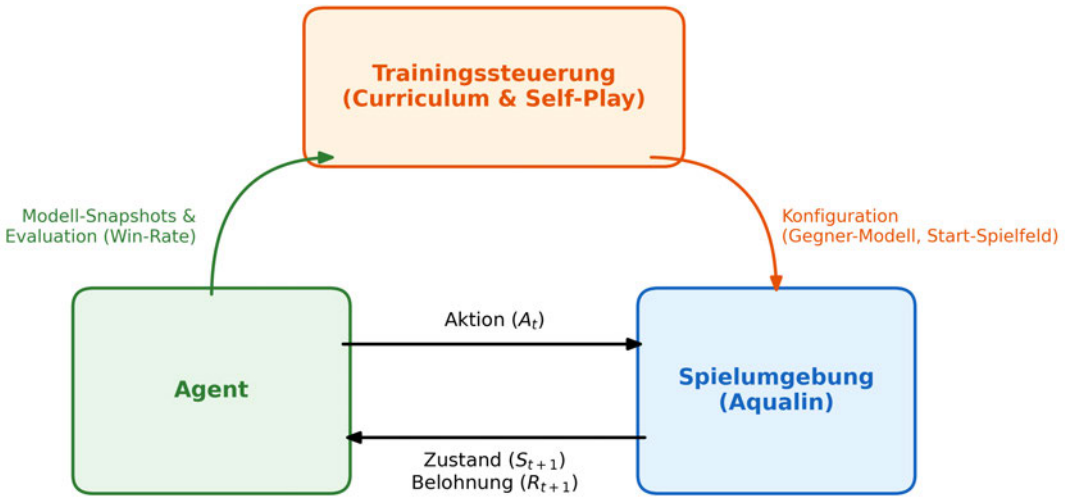


Abbildung 4.1: Konzeptionelle Übersicht des Gesamtsystems: Zusammenspiel von Agent, Umgebung und Trainingssteuerung. (Eigene Darstellung)

4.2 Formalisierung als Markov-Entscheidungsprozess

Um *Aqualin* für den Agenten greifbar zu machen, wird das Spiel als klassischer MDP abgebildet. Mathematisch entspricht dies dem bekannten Tupel $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma)$.

Der **Zustandsraum** $\mathcal{S}$ enthält sämtliche Informationen, die der *Agent* in einem bestimmten Moment über die Partie wissen muss. Dazu gehören im Detail:

- das 6×6 große Spielfeld (die Brett-Matrix),
- die bis zu sechs verfügbaren Steine im aktuellen Pool,
- die Information, welcher Spieler gerade am Zug ist,
- sowie die aktuelle Spielphase (*Slide* oder *Place*).

Das jeweilige Spielziel (Farben oder Tiere sammeln) muss nicht als eigene Variable übergeben werden. Durch die gezielte Sortierung der Eingabedaten (siehe agentenzentrische Normalisierung in Abschnitt 4.3) erkennt das neuronale Netz seinen Fokus ganz automatisch.

Für das neuronale Netz wird der **Aktionsraum** $\mathcal{A}$ in einen flachen Vektor aus exakt 1.513 diskreten Handlungsmöglichkeiten übersetzt. Diese teilen sich in drei Kategorien auf:

- 1.296 Slide-Aktionen (Startfeld $\times$ Zielfeld, 36×36),
- 1 NoSlide-Aktion (den Stein liegen lassen),
- 216 Place-Aktionen (Stein aus dem Pool $\times$ Zielfeld, 6×36).

Wie in den Spielregeln (siehe Abschnitt 2.1.3) erläutert, dürfen Steine auf dem Brett nur waagerecht oder senkrecht verschoben werden. Dies ergibt physikalisch betrachtet lediglich 360 echte Slide-Möglichkeiten. Der Aktionsraum wird im Code dennoch bewusst als volles Raster (36×36) aufgebaut, da dies die Netzwerkarchitektur und die spätere Rückübersetzung der Ausgaben in die eigentliche Spiellogik deutlich vereinfacht. Die verbleibenden 936 Züge, die auf dem echten Brett völlig unmöglich sind (zum Beispiel diagonal über das halbe Brett), filtert das *Invalid Action Masking* (siehe Abschnitt 4.4) einfach heraus. Deren Auswahlwahrscheinlichkeit wird permanent auf null gesetzt, sodass sie das eigentliche Training nicht stören.

Aqualin ist ein deterministisches Nullsummenspiel für zwei Personen. Das bedeutet für die **Übergangsfunktion** $\mathcal{T}$, dass auf eine Aktion in einem bestimmten Zustand mit absoluter Sicherheit ein fester Folgezustand eintritt. Es gibt keine Zufallselemente bei der Zugausführung. Jeder Spieler reagiert direkt auf den exakten Brettzustand, den der Gegner mit seinem letzten Zug hinterlassen hat.

Die **Belohnungsfunktion** R liefert dem *Agenten* das Feedback über die Qualität seiner Züge. Wie genau diese Belohnung im Rahmen des *Reward Engineerings* berechnet wird, beleuchtet Abschnitt 4.6 im Detail. Der **Diskontierungsfaktor** γ bestimmt, wie stark zukünftige Belohnungen im Vergleich zu sofortigen Erfolgen gewichtet werden. Dieser Hyperparameter wird später bei der Implementierung festgelegt.

Zweigeteilter Aktionsraum und Kausalität. Ein naiver Ansatz für die Modellierung des Aktionsraums wäre es, einen offiziellen Spielzug als einen einzigen, kombinierten Ausgabevektor (Schieben + Setzen) zu definieren. Dies führt jedoch zu einer enormen kombinatorischen Explosion der Handlungsmöglichkeiten. In der Spitze entstünden so über 8.000 Kombinationen pro Spielzug. Durch die Aufteilung wird diese multiplikative Komplexität auf eine einfache, additive Auswahl von maximal 216 Optionen pro Schritt reduziert. Vor allem aber entsteht durch die Kombination ein unlösbares Kausalitätsproblem für das *Invalid Action Masking* (siehe Abschnitt 4.4): Die legalen Felder für das Setzen eines neuen Steins hängen direkt davon ab, welches Feld durch die vorherige Schiebe-Aktion frei und welches neu belegt wird. Ein simultanes, präzises Maskieren beider Teilaktionen in einem einzigen Schritt ist logisch unmöglich, da die Entscheidung der ersten Phase die Legalität der zweiten Phase bedingt. Um diese harte Abhängigkeit aufzulösen, wird ein Spielzug in der Umgebung intern zwingend in zwei sequentielle Einzelschritte zerlegt: erst Schieben, dann den resultierenden Zwischenzustand des Brettes evaluieren, und abschließend Setzen. Nur durch diese Zweiteilung wird ein dynamisches, fehlerfreies Action Masking technisch überhaupt erst realisierbar.

4.3 Entwurf der Zustandsrepräsentation

Ein entscheidender Faktor für die Performance des *Agenten* ist die Aufbereitung der Eingangsdaten. Statt einem neuronalen Netz ein einfaches Bild zu zeigen, wird der Spielzustand in eine **12-Kanal Semantic Vision** übersetzt. Dabei wird das Spielfeld in 12 Schichten (ein $12 \times 6 \times 6$ Tensor) aufgeteilt. Jede Schicht konzentriert sich isoliert auf eine einzige Eigenschaft wie eine Farbe oder Tierart und kodiert deren Position auf dem Gitter. Abbildung 4.2 zeigt einen beispielhaften Spielzustand, während Abbildung 4.3 die entsprechende Dekomposition in die einzelnen Kanäle visualisiert.

Diese Form der Repräsentation bietet zwei entscheidende theoretische Vorteile: Erstens verhindert sie den sogenannten *Ordinal-Bias* (Bishop, 2006). Würde man Farben einfach durchnummerieren ($1 = Rot, 2 = Blau, \dots$), würde das neuronale Netz fälschlicherweise eine mathematische Nähe zwischen Farben vermuten, die im Spiel nicht existiert (z. B. dass Rot näher an Blau als an Gelb sei). In der binären 12-Kanal-Darstellung sind alle Eigenschaften mathematisch gleich weit voneinander entfernt (orthogonal). Zweitens liefert diese Struktur dem CNN einen starken *Inductive Bias*: Da das Netz alle Kanäle gleichzeitig betrachtet, kann es räumliche Zusammenhänge (z. B. benachbarte Steine gleicher Farbe) direkt über Faltungsoperationen erfassen, was die Lernkomplexität massiv reduziert.

Ein wesentliches Merkmal dieses Konzepts ist die **agenten-zentrische Normalisierung**. Die Kanäle werden so angeordnet, dass der *Agent* stets seine eigene Punkte-Eigenschaft in den ersten sechs Kanälen wahrnimmt:

- **Kanäle 1–6** (Primäre Eigenschaft): Diese Schichten repräsentieren die Stein-Eigenschaft, die für den aktuell aktiven Spieler punktrelevant ist.

- **Kanäle 7–12** (Sekundäre Eigenschaft): Diese Schichten kodieren die jeweils andere Eigenschaft der Spielsteine (die für den Gegner relevant ist).

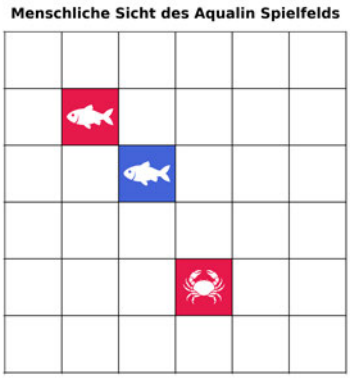


Abbildung 4.2: Beispielhafter Spielzustand von *Aqualin* mit drei Steinen (Roter Fisch, Blauer Fisch, Rote Krabbe). (Eigene Darstellung)

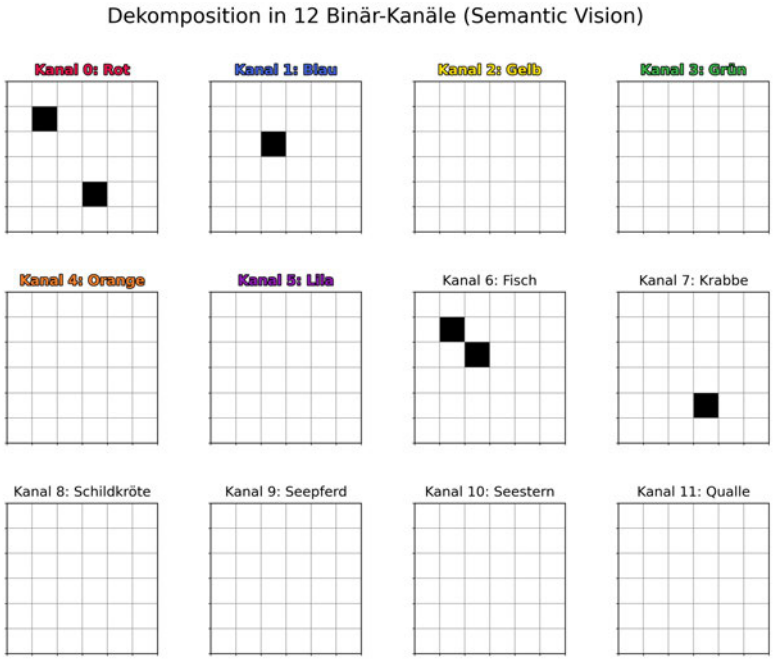


Abbildung 4.3: Dekomposition des obigen Zustands in 12 binäre Kanäle. Jede Eigenschaft wird in eine eigene räumliche Ebene extrahiert. (Eigene Darstellung)

Diese Normalisierung vereinfacht das Lernen, da die Bedeutung der Kanäle für beide Spielerrollen gleich bleibt.

Ganz im Sinne der *AlphaZero*-Philosophie wird zudem auf explizite Punktestände in der Observation verzichtet. Das zwingt das neuronale Netz dazu, ein eigenes Verständnis für die Spielregeln zu entwickeln und Stellungen allein auf Basis ihrer visuellen Anordnung zu bewerten. Dies stellt sicher, dass der Agent das Gelernte sofort auf symmetrische

Situationen übertragen kann.

4.4 Aktionsraum und Invalid Action Masking

Eine besondere Herausforderung beim Training ist die **hohe Sparsität** des Aktionsraums. Zwar umfasst der Raum insgesamt 1.513 theoretisch mögliche Züge, doch zu jedem Zeitpunkt ist nur ein winziger Bruchteil davon tatsächlich erlaubt.

Um zu verhindern, dass der *Agent* wertvolle Rechenzeit mit der Exploration illegaler Aktionen verliert, wird das Konzept des **Invalid Action Masking** eingesetzt. Dabei werden die Auswahlwahrscheinlichkeiten aller ungültigen Aktionen im Entscheidungsprozess faktisch auf Null gesetzt. So lernt der Algorithmus ausschließlich aus Zügen, die den Spielregeln entsprechen, was die Konvergenz in diesem riesigen Suchraum überhaupt erst ermöglicht.

4.5 Architektur des neuronalen Netzes

Die Architektur des *Agenten* (siehe Abbildung 4.4) folgt einem modularen Ansatz, der die unterschiedlichen Datenquellen des Spiels berücksichtigt. Der Feature-Extraktor verarbeitet den Spielzustand in zwei spezialisierten Zweigen:

Board-Branch (CNN). Die Aufgabe des Board-Branch ist es, Muster und räumliche Zusammenhänge auf dem Spielfeld zu erkennen. Die Wahl einer CNN-Architektur begründet sich durch die räumliche Natur des Spielbretts: Da Punkte in *Aqualin* durch benachbarte Steine entstehen, ermöglicht der lokale Fokus der Faltungsfilter die effiziente Erkennung von Gruppen. Da der *Agent* keinerlei Information über den aktuellen Punktestand erhält, muss das Netzwerk allein aus der visuellen Anordnung der Steine lernen, die Wertigkeit von Stellungen zu bewerten.

Pool-Branch (Analyse des Vorrats). Während es auf dem Spielfeld wichtig ist, wo genau ein Stein liegt (räumliche Zusammenhänge), geht es beim Vorrat (Pool) rein um Mengen und Kategorien. Das Netzwerk muss hier lediglich wissen, welche und wie viele Steine einem Spieler noch für zukünftige Züge zur Verfügung stehen.

Da diese Informationen als einfache Liste und nicht als Raster vorliegen, benötigt dieser Zweig des Netzwerks keine aufwendige Mustererkennung (wie die Faltungsoperationen beim Board-Branch). Stattdessen werden hier klassische, vollständig verbundene Schichten (*Dense Layers*) verwendet.

Die Aufgabe des Pool-Branches ist es, die Liste der übrigen Steine auszuwerten und zu lernen, wie wertvoll oder nützlich diese verbleibenden Ressourcen für den weiteren Spielverlauf sind. Das Ergebnis ist eine kompakte mathematische Zusammenfassung der aktuellen Vorratssituation, die bereit ist, mit den Informationen des Spielfelds kombiniert zu werden.

Fusion (Zusammenführung). Nachdem das Netzwerk das Spielfeld (Board) und den Vorrat der Steine (Pool) getrennt voneinander analysiert hat, müssen diese Informationen zu einem Gesamtbild zusammengefügt werden.

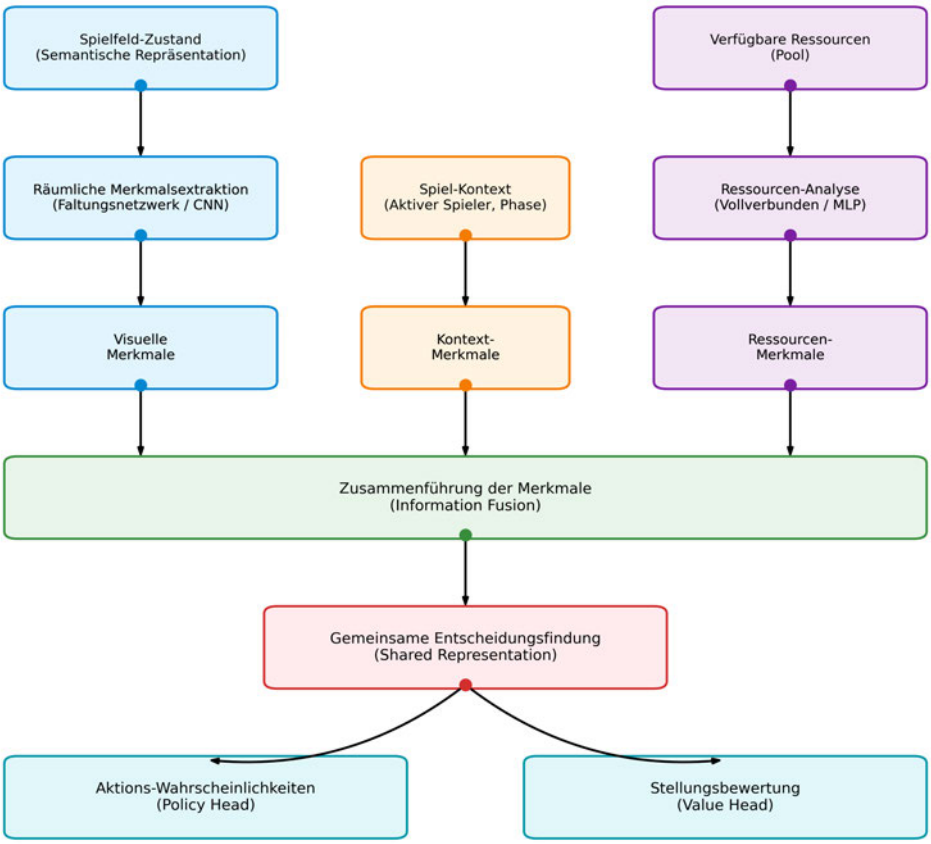


Abbildung 4.4: Konzeptionelle Architektur des neuronalen Netzes von *Poseidon* mit den zwei Eingabezweigen (Board-Branch und Pool-Branch), der Fusion, dem Shared Fully Connected Layer sowie den aufgespaltenen Policy- und Value-Köpfen (eigene Abbildung).

Da das Spielfeld als Raster und der Pool als einfache Liste verarbeitet wird, wird das Ergebnis des Spielfelds zunächst abgeflacht (in eine Reihe umgewandelt). Danach werden die Ergebnisse beider Bereiche einfach aneinandergehängt (Konkatenation).

Durch diese Zusammenführung kennt der *Agent* nun die exakte Position aller Steine auf dem Brett und weiß gleichzeitig, welche Steine er noch zur Verfügung hat. Diese Kombination bildet die direkte Grundlage für die Policy- und Value-Köpfe, mit denen der *Agent* schließlich entscheidet, wie gut die aktuelle Situation ist und welcher Zug als Nächstes gemacht werden soll.

4.6 Reward-Design

Das Belohnungssystem ist darauf ausgelegt, langfristigen Erfolg über kurzfristige Gewinne zu priorisieren.

Terminal Rewards. Es wird bewusst auf dichte, schrittweise Belohnungen (*Intermediate Rewards*) verzichtet. Dies orientiert sich an der Philosophie von *AlphaZero* (Silver et al., 2017). Das Modell lernt, den langfristigen Spielausgang zu optimieren, anstatt gierig kurzfristige Punktgruppen zu bilden, die im späteren Spielverlauf eventuell blockiert werden könnten.

Squashed Margin Reward. Um die Stabilität des Trainings zu erhöhen, dient am Spielende nicht die exakte Punktedifferenz als direkte Belohnung (*Reward*). Stattdessen wird der erspielte Vorsprung durch eine Sättigungsfunktion (*Squashing*) nach oben hin begrenzt.

Diese Entscheidung verfolgt ein klares Ziel: Der *Agent* soll einen signifikanten Unterschied zwischen einem Unentschieden und einem knappen Sieg wahrnehmen. Gleichzeitig verhindert die Abflachung der Kurve, dass extrem hohe Siege überproportional belohnt werden.

Dies schützt den *Agenten* vor einem zu riskanten Spielverhalten. Würde die Belohnung mit jedem weiteren Punkt linear ansteigen, entstünde ein Anreiz, einen bereits gesicherten Sieg durch riskante Züge zu gefährden, um die eigene Punktzahl weiter zu maximieren. Die exakte mathematische Funktion sowie deren grafische Darstellung werden im Rahmen der technischen Umsetzung (siehe Kapitel 5.1.2) behandelt.

4.7 Trainingsstrategie & Curriculum Learning

Um den Lernfortschritt des *Agenten* objektiv zu steuern und die enormen strategischen Herausforderungen von *Aqualin* zu bewältigen, kombiniert das Konzept den Ansatz des *Self-Play* mit einem adaptiven Curriculum.

Entscheidungsfindung: Model-Free statt Baumsuche. Obwohl der *Agent* das *Two-Headed*-Paradigma übernimmt, verzichtet er beim Spielen gänzlich auf eine vorausschauende Baumsuche (wie das von *AlphaZero* genutzte MCTS). Der Verzicht ist eine bewusste Entscheidung, da eine solche Suche für das lokale Training zu rechenintensiv wäre. Stattdessen wird das System direkt mittels PPO trainiert, sodass das Netzwerk

selbst in der Lage ist, Züge ohne externe Suchalgorithmen präzise zu bewerten. *Poseidon* agiert somit als reiner *Model-Free*-Agent: Anstatt wie *AlphaZero* oder *MuZero* mithilfe eines Simulators (*Model-Based*) aktiv in die Zukunft zu planen, wählt das Netzwerk seine Aktionen rein reaktiv auf Basis der im Training erlernten Muster und Intuition aus.

Bootstrapping durch Self-Play. Anstatt das Netzwerk durch überwachtes Lernen mit menschlichen Partien zu trainieren, wird ein reiner Self-Play-Ansatz gewählt. Der *Agent* spielt kontinuierlich gegen Versionen seiner selbst. Dies ermöglicht es dem System, völlig eigenständig Strategien zu entwickeln und ein Verständnis für die komplexen Wechselwirkungen von Board-State und Punktevergabe aufzubauen, ohne durch menschliche (möglicherweise suboptimale) Heuristiken limitiert zu werden.

Komplexitätsreduktion durch Curriculum Learning. Da der *Agent* durch die rein terminale Belohnung am Ende des Spiels (*Sparse Rewards*) anfangs keinerlei Signal erhält, welche Züge gut oder schlecht waren, wäre es äußerst zeitintensiv und rechenaufwendig, das Spiel direkt ab einem leeren Brett zu erlernen. Um dieses Problem zu lösen, wird die Schwierigkeit über ein Curriculum gesteuert: Die Anzahl der bereits platzierten Steine zu Beginn einer Episode wird manipuliert.

Das Training beginnt im extrem späten Endgame (wenn das Spielfeld bereits fast voll ist). Dadurch sieht sich der *Agent* mit einem deutlich kleineren Zustandsbaum konfrontiert und die Zeitspanne zwischen Aktion und Belohnung ist extrem kurz. Erst bei Erreichen einer objektiv messbaren Erfolgsschwelle T gegen Referenzgegner wird die Komplexität erhöht, indem der Startzeitpunkt des Spiels schrittweise weiter in Richtung Spielbeginn verschoben wird. So lernt das neuronale Netz die fundamentale Korrelation zwischen visuellen Mustern und der Punktegenerierung von hinten nach vorne, bis es fähig ist, vollständige Partien zu bestreiten.

Um zu verhindern, dass das Netz im frühen Spielstadium (bei dem viel Zufall im Spiel ist) an einer unrealistisch hohen Gewinnanforderung scheitert, wird die Erfolgsschwelle T dynamisch an den aktuellen Startzug t_{start} angepasst:

$$T(t_{start}) = 0{,}98 - \frac{t_{start}}{30} \cdot (0{,}98 - 0{,}58) \tag{4.1}$$

Im extrem späten Endgame (z. B. Zug 30) muss der *Agent* lediglich eine Win-Rate von 58 % gegen den Zufalls-Bot erreichen, um die Stufe zu meistern. In dieser späten Spielphase ist das Brett bereits so stark gefüllt, dass der *Agent* nur noch begrenzte Möglichkeiten hat, eigene Strategien aufzubauen, und das Ergebnis stark vom Zufall der initialen Steine abhängt. Je weiter das Curriculum jedoch in Richtung Spielbeginn rückt ($t_{start} \to 0$), desto mehr strategischen Einfluss gewinnt der *Agent*. Entsprechend steigt die geforderte Schwelle linear an. Da der berechnete Wert die Hürde zum *Aufstieg* definiert, liegt die höchste praktische Anforderung beim Übergang auf das komplett leere Startbrett: Um von Zug 1 auf Zug 0 aufzusteigen, muss der *Agent* eine Win-Rate von ca. 96,7 % erreichen.

4.8 Heuristische Basis-Agenten

Als feste Basislinien (Baselines) für die Curriculum-Steuerung und die abschließende Evaluierung der Spielstärke dienen zwei hartcodierte, heuristische Gegner:

- **Zufalls-Bot (Random-Bot):** Wählt in jedem Zug aus der Menge der durch das Action-Masking freigegebenen, legalen Züge eine rein zufällige Aktion. Er agiert völlig strategielos und dient als absoluter Nullpunkt (*Minimum Viable Opponent*). Er eignet sich ideal, um zu messen, ob der RL-*Agent* fundamentale Spielmechaniken und das Konzept der Punktegenerierung verstanden hat.
- **Greedy-Bot (Bounded):** Ein deterministischer, kurzsichtiger *Agent*, der den Spielbaum für exakt einen Zug im Voraus (Tiefe 1) durchsucht. Um die Rechenzeit während des massiv-parallelen Trainings gering zu halten, evaluiert er dabei nicht zwangsläufig alle legalen Züge, sondern zieht eine zufällige Stichprobe von maximal 150 legalen Aktionen heran. Aus diesem reduzierten Suchraum wählt er stets diejenige Aktion, die ihm in diesem Moment die meisten direkten Punkte einbringt. Er opfert langfristige strukturelle Vorteile oft für kurzfristige Punktgewinne, stellt aber insbesondere im Endspiel einen deutlich stärkeren und strafenden Gegner als der Zufalls-Bot dar.

5 Implementierung

Hinweis zur Namensgebung: Um die Orientierung für Leser zu erleichtern, die parallel zur Arbeit den beiliegenden Quellcode betrachten, werden in diesem Kapitel die originalen Klassennamen aus dem Code referenziert. Die Klasse des neuronalen Netzes trägt dort den historischen Namen `NeptuneCNN`. Das finale, erfolgreich trainierte Agenten-Modell wird im Text dieser Arbeit jedoch als *Poseidon* (Phase I und II) bezeichnet. *Poseidon* ist iterativ aus den frühen Entwicklungsstufen von *Neptune* hervorgegangen und nutzt exakt dieselbe zugrundeliegende Netzarchitektur.

5.1 Die Architektur des *Aqualin*-Environments (Die Zwiebel)

Um den Code übersichtlich zu halten, wurde das *Aqualin*-Environment in mehrere Schichten aufgeteilt – ähnlich wie bei einer Zwiebel. Anstatt die kompletten Spielregeln, die Belohnungen und das Netzwerk-Training in eine riesige Datei zu quetschen, übernimmt jede Schicht genau eine konkrete Aufgabe.

Ganz innen liegt die Kern-Logik (die reinen Spielregeln). Darum herum legen sich weitere Schichten (sogenannte Wrapper), die das Spielfeld für das neuronale Netz aufbereiten und die Gegnerzüge steuern. Ganz außen entsteht so eine einfache Schnittstelle, an die der PPO-Algorithmus direkt angeschlossen werden kann, ohne sich um die komplexen *Aqualin*-Regeln kümmern zu müssen.

Tabelle 5.1 zeigt, wie diese Python-Dateien ineinandergreifen.

Tabelle 5.1: Hierarchische Struktur der Software-Komponenten (Zwiebel-Prinzip)

Schicht	**Komponente (Python-Modul)**	**Zentrale Aufgabe**
1 (Kern)	Core Engine (`game.py`)	Simulation der Spielregeln und Zustandsverwaltung.
2	Base Environment (`aqualin_base_env.py`)	Steuerung der Zugreihenfolge und Belohnungen.
3	Observation Wrapper (`semantic_observation_wrapper.py`)	Umwandlung des Spielfelds in die 12-Kanal-Ansicht.
4 (Außen)	Self-Play Wrapper (in `train_poseidon_phase1.py`)	Automatisierte Steuerung des Gegners im Training.

Im Folgenden werden diese Schichten von innen nach außen detailliert beschrieben. Die theoretischen Grundlagen des zugrundeliegenden MDP wurden bereits in Kapitel 4.2 definiert.

5.1.1 Schicht 1: Die Core Engine (`game.py`)

Wie in Tabelle 5.1 dargestellt, bildet diese Schicht die innerste Komponente.

Das Herzstück bildet die Klasse `AqualinGame2`. Sie beinhaltet ausschließlich das rohe Regelwerk und den physikalischen Spielzustand, völlig isoliert von jeglicher Reinforcement-Learning-Logik. Die wichtigsten Aufgaben dieser Engine sind:

- **Zustandsverwaltung**: Die Engine speichert das 6×6 Spielfeld sowie den aktuellen Pool (die bis zu sechs offenen Steine). Das Brett wird als einfaches `numpy`-Array abgebildet, bei dem jeder der 36 möglichen Spielsteine durch eine eindeutige Zahl (Integer) kodiert ist.
- **Zugabwicklung**: Die Engine stellt für die Zugabwicklung sowohl die atomaren Methoden `slide()` und `place()` für isolierte Brettveränderungen als auch die Methode `step()` zur Verfügung, die einen kompletten Spielzug zusammenfasst.
- **Punkteberechnung**: Um die aktuelle Punktzahl zu ermitteln, nutzt die Methode `_count_groups()` eine klassische Breitensuche (Breadth-First Search (BFS)) (Cormen et al., 2022). Sie rastert das Feld ab und sucht zusammenhängende Gruppen gleicher Farben oder Tierarten.

Um sicherzustellen, dass der *Agent* fehlerfrei trainiert und keine Regellücken ausnutzt, wurde die Anwendung umfassend durch Unit-Tests abgesichert. Dabei wurde die Programm-Logik gezielt auf verschiedene Grenzfälle getestet – wie beispielsweise sehr lange Verschiebungen von Steinen oder das korrekte Verhalten bei einem nahezu vollen Spielfeld.

5.1.2 Schicht 2: Das Base Environment (`aqualin_base_env.py`)

Als zweite Schicht der Zwiebelarchitektur (vgl. Tabelle 5.1) steuert dieses Modul den Ablauf.

Um den extremen Durchsatzanforderungen moderner Deep-RL-Algorithmen gerecht zu werden, wird die Core Engine in einem PettingZoo-Environment (`AqualinEnv_Neptune`) gekapselt (Terry et al., 2021). PettingZoo setzt hier das Konzept des Agent-Environment-Cycle (AEC) konsequent um: Da *Aqualin* ein strikt abwechselndes Spiel ist, modelliert der AEC-Cycle die Interaktion sequenziell, wodurch garantiert wird, dass die *Agenten* immer auf dem neuesten Board-Status agieren.

Darüber hinaus übernimmt das Base Environment entscheidende Performance-Optimierungen und tiefgreifende RL-Mechaniken, die den *Agenten* bei der Lösungsfindung unterstützen. Dazu gehören primär die Optimierung der Zugabwicklung, der Ausschluss illegaler Züge sowie die Berechnung einer stabilen Belohnung am Spielende.

Speed-Optimierung (Bypass der BFS)

Während in älteren Versuchsmodellen noch die Methode `step()` verwendet wurde, welche nach jedem Zug eine automatische Punkteberechnung auslöst, verzichtet der Wrapper in *Poseidon* aus Performance- und Konzeptgründen bewusst darauf. Stattdessen ruft er

direkt die atomaren Methoden `slide()` und `place()` auf. Die rechenintensive Breitensuche zur Punkteberechnung wird dadurch während des normalen Spielverlaufs komplett übersprungen. Die `step()`-Methode kommt stattdessen nur noch in der Graphical User Interface (GUI) zum Einsatz.

Implementierung des Action Maskings

Das in Kapitel 4.4 hergeleitete Action Masking ist zwingend erforderlich, da von den zahlreichen theoretisch möglichen Zügen im Aktionsraum meist nur ein winziger Bruchteil tatsächlich erlaubt ist. Dafür berechnet das Base Environment in der `observe()`-Methode bei jedem Aufruf eine 1.513 Dimensionen große binäre Maske (1 für legale Züge, 0 für illegale).

Hierbei erfüllt das Masking **zwei zentrale Aufgaben gleichzeitig**: Zum einen filtert es **strukturell unmögliche** Züge heraus. Da der Aktionsraum für Verschiebungen aus Implementierungsgründen als volles 36×36 Raster definiert wurde (siehe Abschnitt 4.4), entsprechen allein 936 dieser 1.513 Dimensionen physikalisch unmöglichen Zügen (wie etwa diagonalen Verschiebungen über das Brett). Zum anderen blockiert es **zustandsabhängige** Züge, die zwar physikalisch existieren, aber in der aktuellen Brettsituation illegal sind.

Wie dünn besetzt dieser Raum in der Praxis ist, zeigt eine empirische Messreihe über 200 Testspiele in Tabelle 5.2. Im Durchschnitt stehen dem *Agenten* pro Schritt lediglich etwa 63 legale Aktionen zur Verfügung. Die Anzahl der legalen Optionen verändert sich dabei im Spielverlauf deutlich: Während die Platzwahl mit jedem Stein auf dem Brett linear abnimmt, folgt der Spielraum für Verschiebungen (*Slides*) einer Glockenkurve. Insgesamt führt die Kombination aus strukturellen und situativen Einschränkungen dazu, dass im Mittel über 95 % der 1.513 Netzwerk-Ausgaben permanent blockiert sind.

Tabelle 5.2: Durchschnittlich legale Aktionen im Spielverlauf (Eigene Messreihe)

Steinanzahl	**Slide-Phase**	**Place-Phase**	**$\sum$ Legal**
0	0,0	216,0	108,0
10	53,8	156,0	104,9
20	46,9	96,0	71,4
30	20,4	36,0	28,2
35	4,3	1,0	2,7

Technisch werden dabei (nach Huang and Ontañón (2022)) die Logit-Werte (die unnormierten Rohausgaben des Netzwerks) aller ungültigen Aktionen im Netzwerk auf einen extrem niedrigen Wert (negativ unendlich) gesetzt. Nach der Softmax-Berechnung sinkt deren Auswahlwahrscheinlichkeit dadurch faktisch auf Null.

Umsetzung des Rewards: Squashed Margin

In der Neptune-Umgebung wird die in Kapitel 4.6 begründete Philosophie der **Terminal Rewards** umgesetzt. Dies bietet zwei wesentliche Vorteile für die Implementierung:

1. **Recheneffizienz:** Da die Punktstände mittels einer Breitensuche (BFS) ermittelt werden müssen, wird die Berechnung erst am Spielende durchgeführt, was die Simulationsrate maximiert.

2. **Zielorientierung:** Das Modell optimiert den langfristigen Spielausgang.

Um ein stabiles Training zu gewährleisten, wird der am Spielende erzielte Punkt-Vorsprung (*Margin*) nicht eins zu eins als Belohnung übernommen. Stattdessen wird der Wert über eine Sättigungsfunktion (hyperbolischer Tangens) geglättet und auf das Intervall $(-1, 1)$ skaliert:

$$r = \tanh\left(\frac{\text{Margin}}{10}\right)$$

Listing 5.1 zeigt die technische Umsetzung innerhalb der `step()`-Methode der Umgebung. Hierbei wird sichergestellt, dass die aufwendige Punkteberechnung mittels Breitensuche nur ein einziges Mal pro Episode aufgerufen wird.

```
# Nur am Spielende (Terminal State) ausfuehren
c_pts, t_pts = self.game.get_scores() # \ac{BFS} Score Berechnung

# Punkt-Differenz fuer beide Spielerrollen
margin_0 = c_pts - t_pts
margin_1 = t_pts - c_pts

# Hyperbolische Tangens-Transformation zur Skalierung
self.rewards["player_0"] = np.tanh(margin_0 / 10.0)
self.rewards["player_1"] = np.tanh(margin_1 / 10.0)
```

Listing 5.1: Technische Umsetzung des Squashed Margin Rewards (Auszug aus `env_neptune.py`)

Hierbei berechnet sich die *Margin* für Spieler 0 (Farben) als $P_{\text{Farben}} - P_{\text{Tiere}}$ und für Spieler 1 (Tierarten) als $P_{\text{Tiere}} - P_{\text{Farben}}$. Diese Skalierung gewichtet knappe Siege (z. B. 2 Punkte Vorsprung $\rightarrow r \approx 0{,}20$) geringer als dominante Siege (z. B. 15 Punkte Vorsprung $\rightarrow r \approx 0{,}90$), wobei die Kurve ab einer Differenz von ca. 10 Punkten in die Sättigung übergeht. Dieser Sättigungseffekt ist entscheidend für die Stabilität des Modells: Dadurch wird verhindert, dass extreme Punktedifferenzen das Training destabilisieren und der Agent nur noch versucht, den Punkteabstand zu maximieren.

5.1.3 Schicht 3: Der Observation Wrapper (`semantic_observation_wrapper.py`)

In der dritten Architekturschicht (Tabelle 5.1) wird die Wahrnehmung des *Agenten* definiert.

Basierend auf dem in Kapitel 4.3 beschriebenen Konzept der **12-Kanal Semantic Vision** erfolgt die technische Umsetzung über den `NeptuneObservationWrapper`. Dieser übersetzt das Spielfeld in einen $12 \times 6 \times 6$ Tensor.

```
def observe(self, agent):
    # 12 Kanaele: 6 fuer Primaereigenschaft, 6 fuer Sekundaereigenschaft
    obs = np.zeros((12, 6, 6), dtype=np.float32)

    for r in range(6):
```

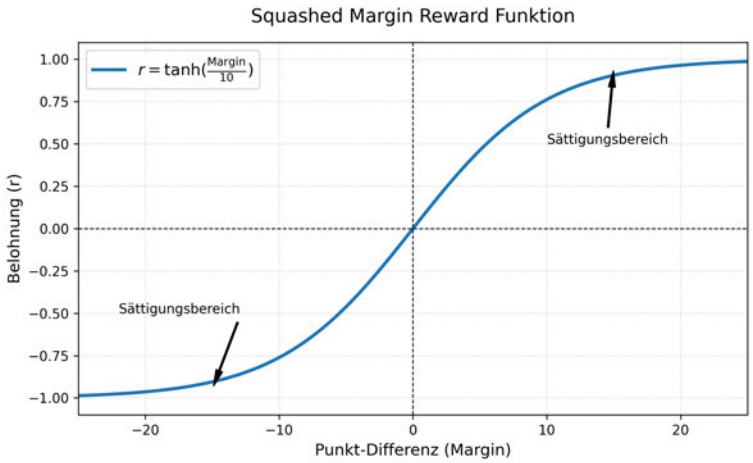


Abbildung 5.1: Verlauf der Squashed Margin Reward Funktion. Die Sättigung ab einer Differenz von ca. 10 Punkten stabilisiert den Lernprozess gegenüber extremen Ausreißern. (Eigene Darstellung)

```
        for c in range(6):
            idx = game.board[r, c]
            if idx != -1:
                color, kind = game.all_pieces[idx]
                # Symmetriebrechung: Der aktive Agent sieht seine
                # Ziel-Eigenschaft immer in den ersten 6 Kanaelen.
                if agent == "player_0": # Sammler von Farben
                    obs[color, r, c] = 1.0      # Kanaele 0-5: Farbe
                    obs[6 + kind, r, c] = 1.0   # Kanaele 6-11: Tierart
                else: # player_1: Sammler von Tierarten
                    obs[kind, r, c] = 1.0       # Kanaele 0-5: Tierart
                    obs[6 + color, r, c] = 1.0  # Kanaele 6-11: Farbe
    return obs
```

Listing 5.2: Agenten-zentrische Normalisierung der 12-Kanal-Observation (Auszug aus `semantic_observation_wrapper.py`)

Diese Normalisierung vereinfacht das Lernen, da die Bedeutung der Kanäle für beide Spielerrollen gleich bleibt. Parallel dazu wird auch der Pool (die sechs verfügbaren Steine) in einen 12×6 Tensor übersetzt. So kann der *Agent* räumliche Muster auf dem Brett sofort mit den aktuell verfügbaren Ressourcen im Pool abgleichen.

Dieser Aufbau bietet zwei wesentliche Vorteile: Erstens verhindert die Trennung der Eigenschaften in binäre Kanäle einen sogenannten *Ordinal-Bias* (Bishop, 2006). Bei einer einfachen Nummerierung (z. B. $1 = \text{rot}, 2 = \text{blau}, \ldots$) würde das Netzwerk fälschlicherweise eine mathematische Nähe zwischen Farben vermuten, nur weil deren Zahlenwerte nah beieinander liegen. In *Aqualin* sind jedoch alle Farben kategorial gleichwertig. Zweitens liefert diese Struktur dem CNN einen starken *Inductive Bias* (Goodfellow et al., 2016, Kap. 9.2). Da das Netzwerk alle Kanäle gleichzeitig betrachtet, kann es räumliche Zusammenhänge zwischen Farben und Tierarten direkt erfassen, ohne diese erst mühsam aus einem flachen Vektor ableiten zu müssen.

Ganz im Sinne der ***AlphaZero***-Philosophie verzichtet Neptune zudem auf explizite

Punktestände in der Observation. Der *Agent* erhält keine Information über den aktuellen Wert seiner Gruppen. Das zwingt das tiefe CNN dazu, ein eigenes Verständnis für die Spielregeln zu entwickeln und Stellungen allein auf Basis ihrer visuellen Anordnung zu bewerten.

5.1.4 Schicht 4: Der Self-Play-Wrapper (in `train_poseidon_phase1.py`)

Die äußerste Schicht (Schicht 4 in Tabelle 5.1) bildet das Interface für den RL-Algorithmus.

Um den PPO-Algorithmus (Schicht 4) in einer Multi-Agenten-Umgebung trainieren zu können, muss die Umgebung für den *Agenten* wie ein Single-Player-Szenario wirken. Die Klasse `NeptuneSelfPlayWrapper` kapselt diesen Prozess, indem sie Gegnerzüge automatisch ausführt und so die Komplexität des AEC-Cycles (Schicht 2) abstrahiert.

Der zentrale Mechanismus ist die Methode `_sync_to_rl_turn` (Listing 5.3), die so lange Schritte in der Core-Engine ausführt, bis entweder das Spiel endet oder der lernende Agent wieder am Zug ist.

```
def _sync_to_rl_turn(self, accumulated_reward):
    while True:
        agent = self.env.unwrapped.agent_selection

        # Spielende: Terminale Belohnung addieren und Step beenden
        if self.env.unwrapped.terminations[agent]:
            rew = self.env.unwrapped.rewards[self.rl_player]
            return obs, accumulated_reward + rew, True, info

        # \ac{RL}-Agent ist am Zug: Kontrolle zurueckgeben
        if agent == self.rl_player:
            return self.env.observe(self.rl_player), accumulated_reward,
                False, info

        # Gegnerzug: Aktion vom Opponent-Model waehlen und ausfuehren
        obs = self.env.observe(agent)
        action, _ = self.opponent_model.predict(obs, action_masks=obs["
            action_mask"])
        self.env.step(action)
```

Listing 5.3: Synchronisations-Loop zur automatisierten Abhandlung von Gegnerzügen im Self-Play.

Dabei bleibt der Gegner über den gesamten Verlauf einer Episode konstant. Zur Steigerung der Trainingseffizienz und zur Stabilisierung der Lernkurve wird zudem ein neues Modell aus dem League-Pool lediglich alle 50 Episoden ausgewählt. Dies verhindert ein übermäßiges Laden von Modellen (*Cache Thrashing*) und ermöglicht es dem lernenden Agenten, sich über mehrere Partien hinweg auf die spezifische Spielweise eines Gegners einzustellen.

5.2 Die Neuronale Netzarchitektur: NeptuneCNN

Die technische Umsetzung des Feature-Extraktors `NeptuneCNN` basiert auf der in Kapitel 4.5 beschriebenen Zwei-Zweig-Struktur:

Board-Branch. Dieser Zweig wird durch drei hintereinandergeschaltete Blöcke der Form Conv2d → BatchNorm2d → ReLU realisiert (siehe Abbildung 2.5 in den Grundlagen). Durch den Einsatz von 3×3-Filtern mit Padding „sieht“ das Netz jeden Stein immer im Kontext seiner direkten Nachbarn, wobei die ursprüngliche Größe des Spielfelds über alle Schichten hinweg erhalten bleibt. Die Anzahl der Kanäle steigt dabei schrittweise von 64 über 128 auf 256 an. Am Ende wird der verarbeitete Zustand zu einem Merkmalsvektor mit 9.216 Elementen umgeformt (*Flattening*).

Pool-Branch. Dieser zweite Zweig verarbeitet den Markt-Pool über ein klassisches Multi-Layer Perceptron (MLP). Der Pool-Tensor (12×6) wird dazu in einen flachen Vektor umgewandelt und auf 128 Dimensionen verdichtet.

Fusion. Die Ergebnisse beider Zweige werden anschließend (siehe Abbildung 5.2) mit den Metadaten des Spielzugs (`current_player`, `phase`) zusammengeführt. Dadurch entsteht ein kombinierter Vektor aus $9.216 + 128 + 2 + 2 = 9.348$ Elementen. Zwei weitere vollverbundene Schichten ($1.024 \rightarrow 512$) berechnen daraus den finalen Feature-Vektor für die Policy- und Value-Köpfe.

Um das Lernen zu stabilisieren, kommt **Batch Normalization** zum Einsatz (siehe Kap. 2.4.2). Diese Technik verhindert zu starke Schwankungen im Netz (*Internal Covariate Shift* (Ioffe and Szegedy, 2015)). Das ist bei *Poseidon* besonders wichtig, da durch den Wegfall der direkten Punktbelohnungen anfangs nur sehr schwache und unruhige Lernsignale zur Verfügung stehen.

5.3 Trainingsinfrastruktur und Parallelisierung

Das Training des *Poseidon*-Agenten wird auf vier parallelen CPU-Prozessen mittels `SubprocVecEnv` durchgeführt. Da die Spielumgebung in dieser Ausbaustufe bereits hochgradig optimiert ist, können deutlich größere Rollout-Fenster verwendet werden als in früheren Experimenten. Die gesammelten Erfahrungen werden nach je $n_steps = 8.192$ Schritten pro Umgebung synchronisiert. Die effektive Batch-Größe pro PPO-Update beträgt damit $4 \times 8.192 = 32.768$ Transitions. Die Rechenlast wurde auf einer **NVIDIA GeForce RTX 30-Serie** unter Nutzung der CUDA-Schnittstelle verarbeitet. Ein vollständiger Trainingszyklus über beide Phasen (*Poseidon* I und II) beanspruchte dabei eine Rechenzeit von über **569 Stunden** (ca. 23,7 Tage), in denen insgesamt **390,4 Millionen** Interaktionen mit der Umgebung verarbeitet wurden.

Zur Echtzeit-Überwachung des Lernfortschritts wurde **TensorBoard** eingesetzt. Dabei standen zwei Metriken im Vordergrund: Die *Policy Entropy* wurde überwacht, damit sich der Agent nicht zu früh auf eine einzelne Strategie festfährt. Gleichzeitig diente der *Value Loss* dazu zu prüfen, wie zuverlässig das Kritik-Netzwerk die aktuelle Spielstellung bewerten kann.

Die Wahl der Hyperparameter orientiert sich an den Best-Practice-Empfehlungen für den PPO-Algorithmus (Schulman et al., 2017), wurde jedoch gezielt an die hohe Kapazität des NeptuneCNN angepasst. Die Lernrate wurde bewusst niedrig auf 10^{-4} angesetzt, damit das neuronale Netz nicht zu hastig lernt und nützliche Muster wieder überschreibt. Zusätzlich wurde der *Entropy Coefficient* (`ent_coef`) angepasst, der den

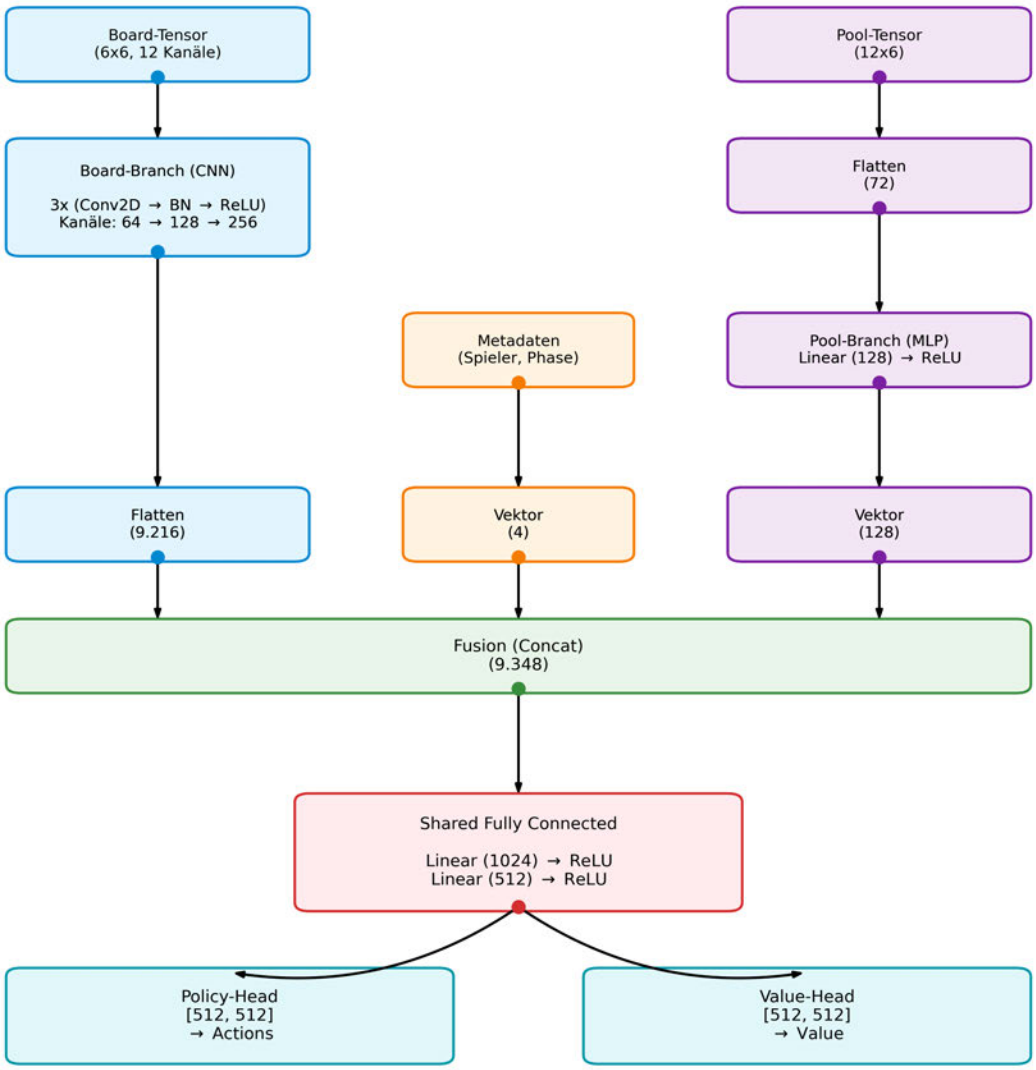


Abbildung 5.2: Architektur des `NeptuneCNN`: Kombination aus Board-Branch (CNN) und Pool-Branch (MLP) mit anschließender Fusion der Merkmalsvektoren. (Eigene Darstellung)

Entdeckerdrang des Agenten steuert. In *Poseidon* I wurde dieser Wert nach etwa 144 Millionen Spielzügen von 0,02 auf 0,005 gesenkt. Das signalisiert dem Agenten, weniger verrückte Züge auszuprobieren und stattdessen seine gefundene Taktik zu perfektionieren. In *Poseidon* II reichte für dieses Gleichgewicht ein fester Wert von 0,01 aus. Zudem greift ein `target_kl` Limit als Early-Stopping-Mechanismus, falls sich die Policy pro

Tabelle 5.3: Hyperparameter des *Poseidon*-Trainings (Basis-Konfiguration für Phase I und II)

Parameter	**Wert**	**Begründung**
`n_envs`	4	Optimale Auslastung der Systemressourcen
`n_steps`	8.192	Erfasst große zusammenhängende Spielverläufe
`batch_size`	1.024	Fördert einen stabilen und gleichmäßigen Lernprozess
`n_epochs`	10	Intensive Wiederholung der gesammelten Erfahrungen
`gamma`	0,999	Starke Gewichtung des langfristigen Spielerfolgs
`clip_range`	0,2	Verhindert sprunghafte Änderungen der Spielweise
`ent_coef`	0,02 $\rightarrow$ 0,005	Schrittweise Reduktion der experimentellen Spielzüge
`learning_rate`	10^{-4}	Behutsame Anpassung bereits erlernter Muster
`target_kl`	0,03	Schutzmechanismus gegen fehlerhafte Strategieanpassungen

Update zu drastisch verändert.

Modell-Checkpoints werden in regelmäßigen Intervallen als Teil des **League Training** gespeichert. Alle 1.000.000 Schritte wird der aktuelle Zustand des *Agenten* als komprimierte Datei auf der Festplatte abgelegt. Bis zum Abschluss des Trainings wuchs dieser Pool auf knapp 400 verschiedene Gegner-Generationen an. Um den Arbeitsspeicher bei dieser gigantischen Menge an Netzwerken nicht zu überlasten, wurde programmiertechnisch ein *Least Recently Used* Cache (`LeagueModelCache`) implementiert. Dieser begrenzt die Anzahl der gleichzeitig im Speicher gehaltenen historischen Modelle strikt auf fünf und tauscht diese dynamisch aus, sobald ein neuer, noch nicht geladener Gegner für eine Episode ausgelost wird. Dies ermöglicht es dem System, im Verlauf des Trainings gegen eine enorme Vielfalt an „Generationen“ seiner selbst zu spielen, ohne an Hardware-Limits zu stoßen, was die Robustheit der Strategie gegenüber einer einseitigen Überoptimierung massiv erhöht.

Die Zusammensetzung des Gegner-Pools variiert dabei gezielt zwischen den beiden Trainingsphasen: Während ***Poseidon* I** als reines Self-Play-Training konzipiert ist und ausschließlich gegen vergangene Checkpoints des League-Pools trainiert, wird der *Agent* in ***Poseidon* II** mit einem gemischten Pool konfrontiert. Hier spielt er zu 65 % gegen die League-Modelle, wird aber zudem in 25 % der Episoden gegen den deterministischen Greedy-Bot und in 10 % der Episoden gegen den Zufalls-Bot gesetzt.

Die genauen Quoten (65 %, 25 % und 10 %) sind das Ergebnis praktischer Überlegungen. Sie wurden nicht durch aufwendige empirische Tests exakt ermittelt, sondern bewusst so gewählt, dass sie zwei wichtige Regeln erfüllen:

Erstens muss der Großteil der Spiele (deutlich über 50 %) gegen die *League* stattfinden. Nur so lernt der *Agent* stetig dazu, indem er sich an immer stärkere Versionen seiner selbst anpasst.

Zweitens muss der Greedy-Bot deutlich öfter zum Einsatz kommen als der Zufalls-Bot. Der Greedy-Bot bestraft grobe Fehler sofort und zwingt das Modell so, solide zu spielen. Sein Anteil darf aber nicht zu groß werden, da das Netzwerk sonst nur noch lernt, genau diesen einen Gegner auszutricksen (*Overfitting*). Die restlichen 10 % gegen den Zufalls-Bot streuen bewusst etwas Chaos ein. Sie verhindern, dass sich das Netz in feste,

immer gleiche Zugmuster verrennt. Ob eine leichte Verschiebung der Verteilung (etwa auf 70-20-10) zu besseren Trainingsergebnissen geführt hätte, wurde nicht im Detail getestet. Die gewählte 65-25-10-Aufteilung erfüllt jedoch genau ihren Zweck: Sie zwingt den *Agenten* dazu, ein robustes und vielseitiges Spielverständnis aufzubauen.

5.4 Steuerung des Curriculums: Benchmark-Callbacks

Die technische Steuerung des Lernfortschritts erfolgt über *Callbacks*, die den in Kapitel 4.7 vorgestellten Konzepten folgen. Die Schwierigkeit des Curriculums wird über die Anzahl der offen liegenden Steine zu Beginn einer Episode gesteuert. Ein *Agent*, der das Spiel erst ab dem 30. Zug (6 verbleibende Steine) lernt, sieht sich mit einem deutlich kleineren Zustandsraum konfrontiert als ein *Agent*, der bei Zug 0 startet.

Ein entscheidendes Detail der Implementierung ist, dass die Evaluationsspiele exakt am aktuellen *start_turn* des Curriculums ansetzen. Das bedeutet, dass das Spiel für den Benchmark nicht bei einem leeren Brett beginnt, sondern analog zum aktuellen Training bei einer zufälligen Konfiguration mit bereits platzierten Steinen. Dies stellt sicher, dass die gemessene Win-Rate direkt mit der für diese spezifische Phase definierten Erfolgsschwelle T vergleichbar ist und der *Agent* nur dann „aufsteigt“, wenn er die aktuelle Komplexitätsstufe sicher beherrscht.

Aufgrund der unterschiedlichen Anforderungen wurden für die beiden Trainingsphasen spezialisierte Callbacks entwickelt:

Phase I: *Poseidon* I (`NeptuneEvalCurriculumCallback`). In der ersten Phase lernt das Netz die grundlegenden Spielmechaniken. Der Benchmark umfasst hierbei jeweils **1.000 Evaluationsspiele** gegen den in Kapitel 4.8 definierten Zufalls-Bot. Erreicht der *Agent* die dynamisch berechnete Win-Rate-Schwelle T (siehe Formel 4.1), wird die Schwierigkeit im nächsten Trainingsblock erhöht (der Startzeitpunkt wird vorverlegt). Der umfangreiche Benchmark von 1.000 Spielen garantiert dabei eine statistisch hochsignifikante Aussage über die Spielstärke.

Phase II: *Poseidon* II (`PoseidonIICurriculumCallback`). Nachdem *Poseidon* I das volle Brett (Zug 0) erreicht hatte, wurde das Curriculum für *Poseidon* II zurück auf Zug 26 (10 freie Felder) gesetzt, um komplexe Endspiel-Taktiken tiefer zu trainieren. In dieser Phase reicht es nicht mehr aus, nur den Zufalls-Bot zu besiegen. Der Callback evaluiert das Modell parallel gegen den Zufalls-Bot *und* den deterministischen *Greedy-Bot*. Um das nächste Curriculum-Level zu erreichen, muss *Poseidon* II nun die Schwelle T gegen den Zufalls-Bot übertreffen **und gleichzeitig** mindestens 40 % der Spiele gegen den *Greedy-Bot* gewinnen. Da die Evaluierung gegen den *Greedy-Bot* deutlich zeitintensiver ist als gegen den Zufalls-Bot, wurde die Anzahl der Evaluationsspiele in dieser Phase auf **50 Runden** pro Benchmark reduziert.

6 Evaluierung

In diesem Kapitel wird der in der Implementierung entwickelte DRL-Agent in seinen beiden finalen Ausbaustufen (*Poseidon* I und II) quantitativ untersucht. Ziel ist es, die Stabilität und Effizienz des Trainingsprozesses nachzuvollziehen sowie die erreichte Spielstärke durch umfangreiche Benchmark-Tests objektiv einzuordnen. Auf Basis der erhobenen Metriken wird evaluiert, inwiefern sich der *Agent* gegen deterministische und zufällige Strategien behaupten kann und wie robust die erlernte *Policy* gegenüber den asymmetrischen Spielregeln von *Aqualin* ist.

6.1 Versuchsaufbau und Methodik

Die Evaluierung des *Agenten* erfolgt in mehreren voneinander getrennten Phasen, um unterschiedliche Aspekte des maschinellen Lernprozesses und der finalen Spielstärke isoliert betrachten zu können.

Als Basis für die Datenerhebung dient die *TensorBoard*-Integration, welche während der Trainingsläufe kontinuierlich Metriken wie den durchschnittlichen Reward, den Verlust der Wertefunktion (Value Loss) und die Vorhersagegenauigkeit (Explained Variance) aufzeichnet. Zur besseren visuellen Signalextraktion und Reduzierung des Rauschens in den Graphen werden die Trainingsmetriken mit einem exponentiell gleitenden Durchschnitt (Exponential Moving Average, EMA) versehen. Hierbei kommt zur besseren Vergleichbarkeit ein einheitlicher Glättungsfaktor zum Einsatz.

Um die absolute Spielstärke des finalen Modells zu evaluieren, wurde eine Benchmark-Umgebung entwickelt. In dieser absolviert der *Agent* vollautomatisierte Turniere über jeweils 1.000 Spiele gegen standardisierte Referenzgegner:

- **RandomBot**: Ein *Agent*, der in jedem Zug eine rein zufällige, gültige Aktion auswählt. Er dient als Baseline, um zu validieren, ob grundlegende Spielregeln und minimale Taktiken erlernt wurden.

- **GreedyBot**: Eine heuristische Strategie, die in jedem Zug eine zufällige Teilmenge von bis zu 150 legalen Zügen simuliert (1-Step-Lookahead). Anstatt das Spielfeld aufwändig mit einer Breitensuche zu analysieren, nutzt der Bot eine schnelle $\mathcal{O}(1)$-Heuristik: Er prüft ausschließlich die direkten Nachbarfelder der platzierten und verschobenen Steine. Ausgewählt wird stets der Zug, der den eigenen Vorteil an passenden Nachbarn gegenüber dem Gegner maximiert.

Alle Benchmark-Spiele werden von Beginn an (leeres Spielfeld) bis zum regulären Spielende durchgeführt. Aufgezeichnet werden dabei die prozentualen Gewinnraten (Win Rates) sowie die absoluten Punktedifferenzen (Point Margins) am Ende der Partie.

6.2 Analyse des Trainingsverlaufs und der Modellarchitektur

Ein zentraler Bestandteil dieser Arbeit war die Identifizierung einer geeigneten Modellarchitektur und stabiler Hyperparameter für das PPO-Verfahren. Da frühe Überlegungen und erste Tests zeigten, dass einfache flache neuronale Netze (MLPs) die räumlichen Beziehungen der Spielsteine auf dem 6×6-Brett nicht ausreichend erfassen können, fiel die Wahl auf ein CNN als Feature Extractor. Diese Architektur wurde im Laufe der Entwicklung iterativ verfeinert, bis das Modell ein konstantes Lernverhalten zeigte.

6.2.1 Einfluss der KL-Divergenz auf das Curriculum Learning

Im frühen Verlauf der Entwicklung zeigte der initiale Modellansatz (*Orion*) erhebliche Schwierigkeiten beim Erlernen der Aqualin-Spielmechanik. Ohne eine Begrenzung der Updates pro Trainingsiteration durch eine Target KL-Divergenz führte das PPO-Verfahren zu unregulierten Policy-Updates. Dies verhinderte, dass sich das Modell stabil an zunehmend komplexe Brettzustände anpassen konnte. Ein niedrigerer Wert auf der Y-Achse (siehe Abbildung 6.1) bedeutet hierbei einen höheren Schwierigkeitsgrad, da das Spiel mit weniger vorgegebenen Steinen auf dem Brett beginnt und somit der Möglichkeitsraum (State Space) kombinatorisch wächst.

Wie in Abbildung 6.1 verdeutlicht wird, stagnierte *Orion* beim Durchlaufen des *Curriculum Learnings* extrem schnell. Innerhalb der ersten 80 Millionen Trainingsschritte schaffte es Orion lediglich, den Start-Turn von 30 auf 24 zu senken (rote Kurve). Durch die Einführung der Target KL-Divergenz in der Architektur von *Poseidon I* wurde das Netzwerk zu konservativeren und gezielteren Updates gezwungen. Dies resultierte in einem drastisch beschleunigten Lernprozess: *Poseidon* I meisterte im selben Zeitraum beinahe das gesamte Curriculum und erreichte Start-Turn 6 (blaue Kurve).

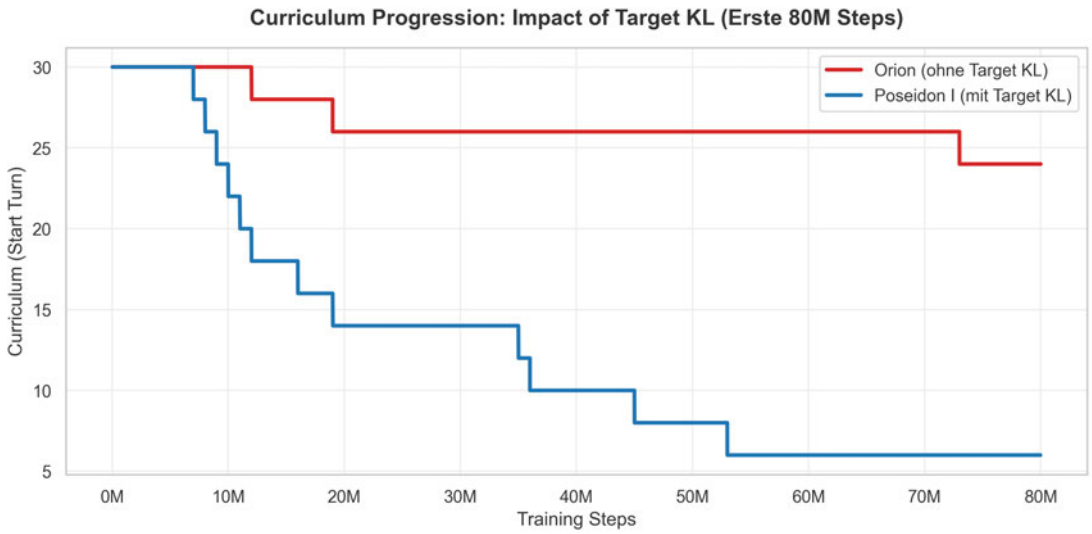


Abbildung 6.1: Vergleich des Curriculum-Fortschritts (erste 80 Mio. Schritte). Die Target KL-Divergenz stabilisiert *Poseidon* I deutlich. (Eigene Darstellung)

6.2.2 Trainingsphasen von *Poseidon* (I und II)

Das Training von *Poseidon* wurde iterativ gestaltet. *Poseidon I* absolvierte 207 Millionen Trainingsschritte in einem reinen Self-Play-Szenario. Hierbei trat der *Agent* kontinuierlich gegen historische Versionen seiner selbst an.

Abbildung 6.2 zeigt den Verlauf der durchschnittlichen Belohnung (Mean Reward) zusammen mit dem Fortschritt im Curriculum. Es ist deutlich zu erkennen, wie der *Agent* das Curriculum (rote gepunktete Kurve) stetig meisterte und den Start-Turn über die Zeit bis auf 0 senkte. Dabei zeigte sich, dass der Reward trotz der regelmäßigen Erhöhung des Schwierigkeitsgrads (Treppenstufen nach unten) keine starken Einbrüche aufwies. Das Modell adaptierte sich so nahtlos an die komplexeren Brettsituationen mit längerer Spieldauer, dass der durchschnittliche Reward über den gesamten Verlauf kontinuierlich anstieg.

Ein detaillierter Blick auf die Lernkurve offenbart die abnehmende Lerngeschwindigkeit bei steigender Brettkomplexität: Während der *Agent* die initialen Stufen des Curriculums (von Zug 30 bis 16) sehr rasant absolvierte, zeigte sich auf Stufe 14 ein erstes deutliches Plateau. Hier benötigte das Modell spürbar mehr Iterationen, um die geforderte Win-Rate zu erreichen. Nach dieser ersten Hürde nahm die Verweildauer auf den darauffolgenden Stufen kontinuierlich zu. Die größte Herausforderung stellte die Stufe ab Zug 4 (Curriculum Start Turn) dar, auf der das Modell annähernd so viel Trainingszeit verbrachte wie für das gesamte restliche Training zuvor.

Die Ursache für diese lange Stagnation lag im Entropie-Koeffizienten (*ent_coef*): Dieser war relativ hoch gewählt, um das Modell zu zwingen, konstant neue, unvorhersehbare Züge auszuprobieren (*Exploration*). Diese anhaltend hohe Stochastik verhinderte jedoch, dass der *Agent* in dieser späten, hochkomplexen Phase seine bereits gefundenen Taktiken verlässlich anwenden konnte, um die strikt geforderte Win-Rate für den nächsten Curriculum-Schritt zu erreichen. Aus diesem Grund wurde der Entropie-Koeffizient ab ca. 140 Millionen Schritten manuell reduziert. Dieser Eingriff befreite den Agenten von dem künstlichen Zwang zur Exploration und erlaubte es ihm stattdessen, sein erlerntes Wissen strategisch auszunutzen (*Exploitation*). Dies führte schließlich zum erfolgreichen Abschluss des Curriculums.

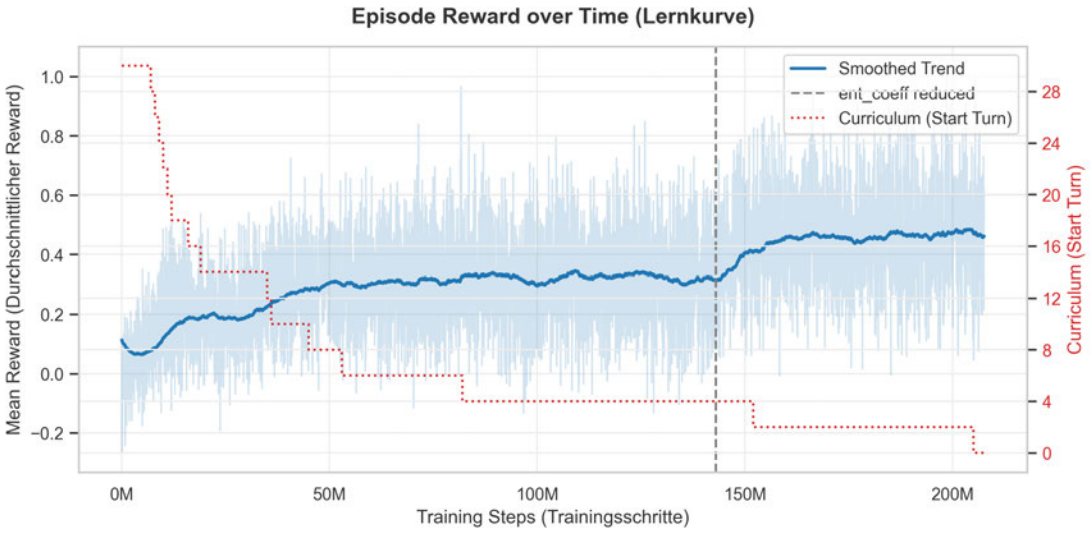


Abbildung 6.2: Mean Reward und Curriculum in Phase 1. Der *Agent* meistert das Curriculum sukzessive, ohne dass der Reward bei den Erhöhungen des Schwierigkeitsgrads signifikant einbricht. (Eigene Darstellung)

Wie Abbildung 6.3 zeigt, etablierte das Modell begleitend dazu eine konsistent positive Spieldominanz (Point Margin > 0). Da der Gegner-Pool im Self-Play laufend mit neueren, stärkeren Netzwerk-Versionen aktualisiert wurde, bedeutete diese dauerhaft positive Punktedifferenz, dass sich der *Agent* nicht auf einer statischen Strategie ausruhte. Vielmehr steigerte er seine Spielstärke fortlaufend, um seine immer kompetenteren Vergangenheits-Ichs zu besiegen. Zusätzlich ist ab der Marke von etwa 140 Millionen Schritten ein leichter Aufwärtstrend der Dominanz zu erkennen. Dieser wurde durch die zuvor beschriebene Reduzierung des Entropie-Koeffizienten ausgelöst, da der *Agent* nun nicht mehr zu zufälligen Experimenten gezwungen wurde und sein Wissen zielgerichteter ausspielen konnte.

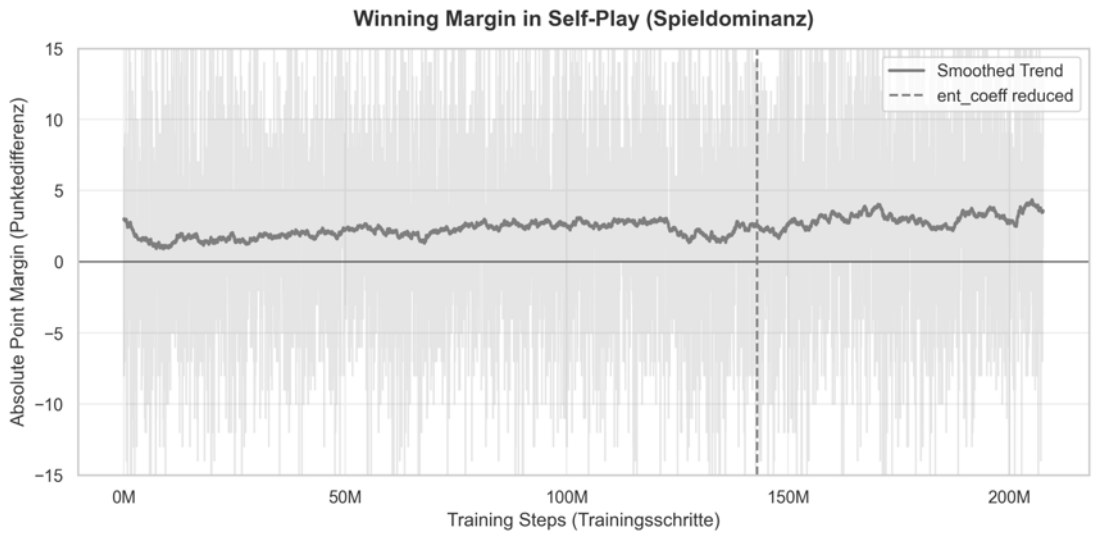


Abbildung 6.3: Point Margin in Phase 1 (Self-Play). *Poseidon* I lernt konstant, eine positive Punktedifferenz aufzubauen. (Eigene Darstellung)

Am Ende der ersten Phase hatte *Poseidon I* erfolgreich bewiesen, dass die zugrunde liegende Modellarchitektur funktionierte und der *Agent* das Spielprinzip erlernen konnte. Erste Testspiele zeigten jedoch eine signifikante Schwäche auf: Während das Modell gegen zufällige Züge extrem hohe Siegquoten (teils über 90 %) erreichte, agierte es gegen deterministische Heuristiken wie den GreedyBot noch äußerst schwach und verlor in der Regel deutlich. Das reine Self-Play hatte den *Agenten* in ein lokales Optimum manövriert, da er nie gelernt hatte, auf fundamental andere, stark nutzenorientierte Strategien zu reagieren.

Um diese Schwäche zu beheben und dem Modell ein tieferes, robusteres Spielverständnis zu vermitteln, wurde eine zweite Trainingsphase (*Poseidon II*) initiiert. Wie bereits in Kapitel 5.3 begründet, wurde hierbei ein asymmetrischer Gegner-Pool eingeführt. *Poseidon* II trainierte über weitere 182 Millionen Schritte gegen die Mischung aus historischen Versionen seiner selbst, dem starken *GreedyBot* sowie dem *RandomBot*. Dieses asymmetrische Umfeld zwang das Modell systematisch dazu, sich aus seinem lokalen Optimum zu lösen und flexible Gegenstrategien zu entwickeln. Zu Beginn dieser Phase wurde das Curriculum bewusst wieder auf Start-Turn 26 zurückgesetzt. Dies gab dem Modell den nötigen Spielraum, um sich an die neuen Gegner anzupassen, hielt den Schwierigkeitsgrad durch noch nicht ausgespielte Steine im Pool aber weiterhin hoch.

Abbildung 6.4 illustriert den Trainingsverlauf in dieser zweiten Phase. Wie bereits in Phase 1 beobachtet, trat auch hier der bekannte Plateau-Effekt auf, jedoch in wesentlich massiverer Ausprägung. Um den Start-Turn von 26 auf 24 zu senken, benötigte das Modell nun knapp 150 Millionen Trainingsschritte. Dieser drastische Anstieg der Verweildauer auf einer einzelnen Curriculum-Stufe resultierte direkt aus der Komplexität des asymmetrischen Gegner-Pools. Da sich der *Agent* nun an stark unterschiedliche Verhaltensmuster anpassen musste, dauerte es entsprechend länger, bis er die für einen Curriculum-Aufstieg nötige Siegesrate konstant erreichen konnte.

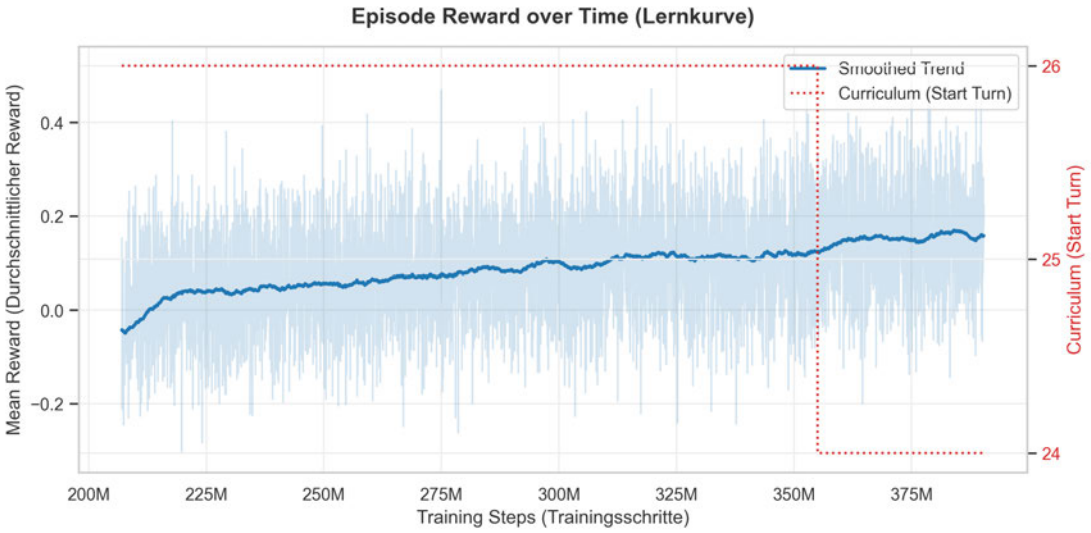


Abbildung 6.4: Mean Reward und Curriculum in Phase 2. Die Dauer bis zum nächsten Curriculum-Aufstieg (von Turn 26 auf 24) verlängert sich durch den komplexen Gegner-Pool drastisch auf knapp 150 Millionen Schritte. (Eigene Darstellung)

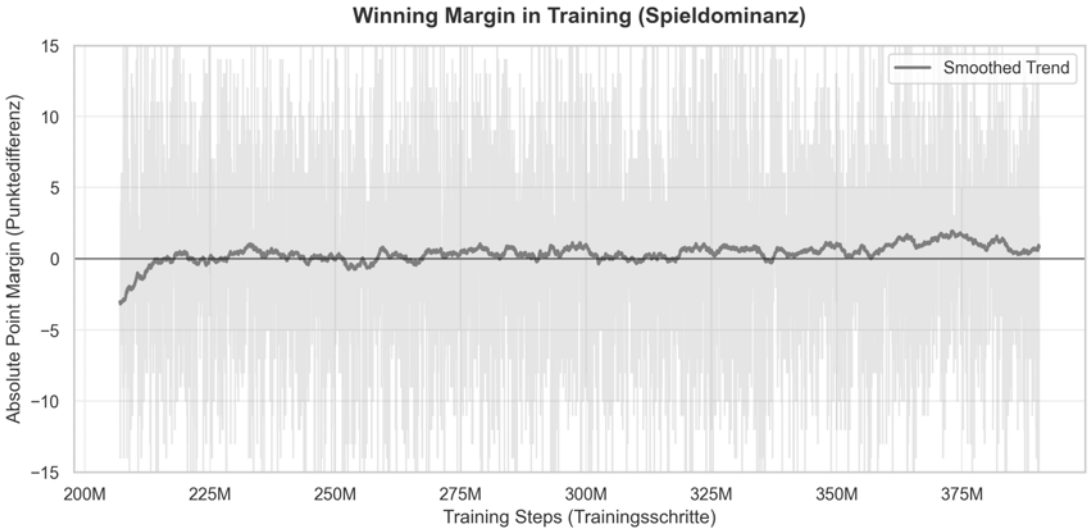


Abbildung 6.5: Point Margin in Phase 2. Die Kurve oszilliert durch den starken Gegner-Pool stärker, stabilisiert sich aber positiv. (Eigene Darstellung)

Der gemischte Gegner-Pool führte zwar zu stärkeren Schwankungen in der Lernkurve (siehe Abbildung 6.5), erzwang aber eine wesentlich robustere Spielweise. Dass sich die *Margin* gegen Ende des Trainings dennoch im positiven Bereich einpendelte, zeigt deutlich: *Poseidon* II kann sich selbst in diesem extrem anspruchsvollen Umfeld behaupten und erzielt im Schnitt mehr Punkte, als er abgibt.

6.2.3 Konvergenz der PPO-Metriken

Zusätzlich zur Betrachtung der Spielstärke (Margin) belegt die Analyse der *Policy Entropy* und des *Value Loss* die stabile Konvergenz des zugrundeliegenden PPO-Algorithmus. Abbildung 6.6 visualisiert diesen Prozess exemplarisch anhand von *Poseidon* I.

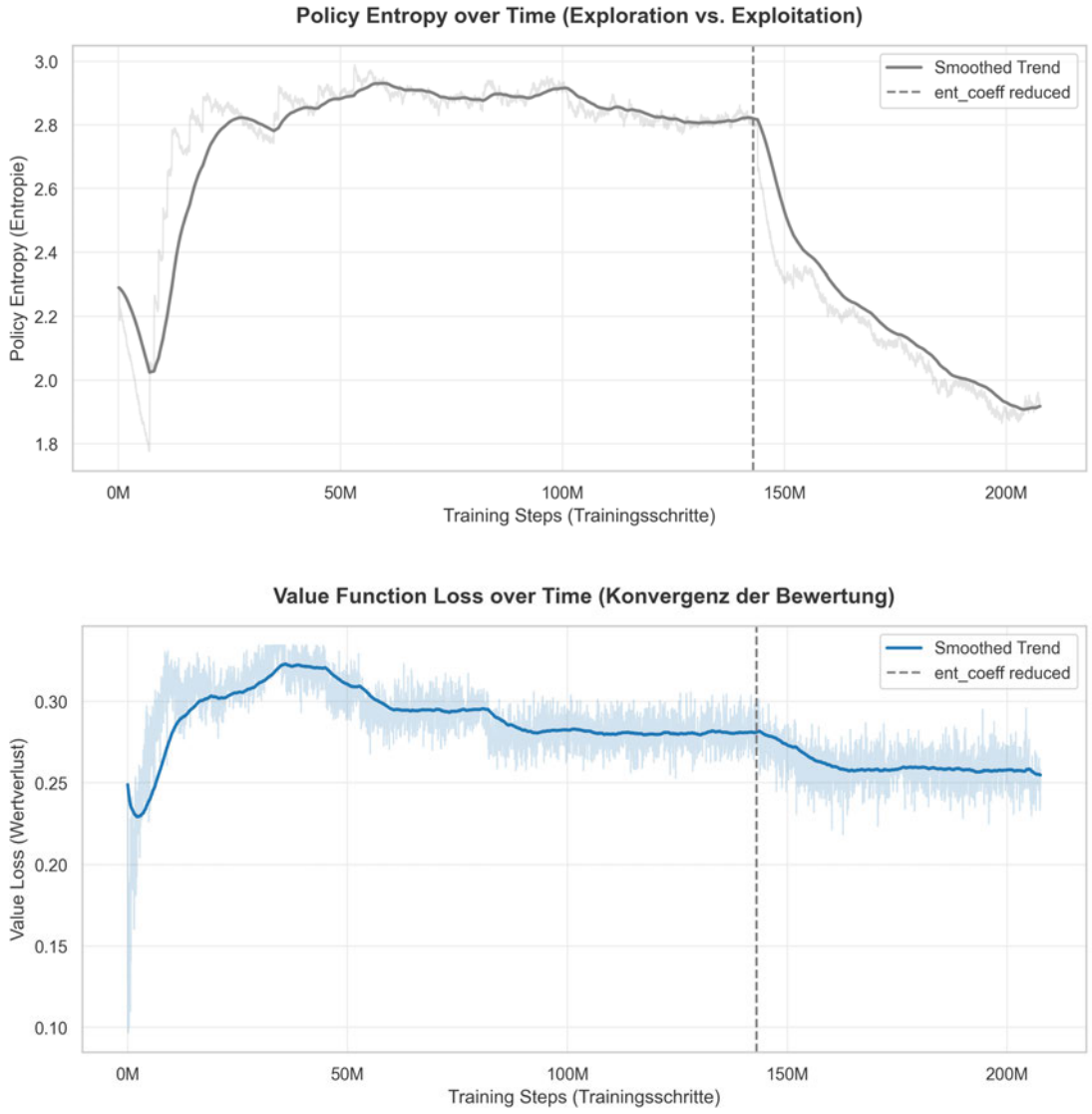


Abbildung 6.6: Konvergenz in Phase 1 (*Poseidon* I). Oben: Sinkende Policy Entropy signalisiert reduzierte Exploration. Unten: Abnehmender Value Loss belegt präzisere Vorhersagen des Critics. (Eigene Darstellung)

Ein detaillierter Blick auf die *Policy Entropy* (siehe Abbildung 6.6, obere Grafik) verdeutlicht, dass das Modell zu Beginn des Trainings intensiv explorierte. Die Entropie stieg initial stark an und verblieb über weite Teile der ersten Trainingshälfte auf einem durchgehend hohen Niveau. Ein zu schnelles Absinken wurde durch den gewählten

Entropie-Koeffizienten gezielt verhindert, um zu gewährleisten, dass sich der *Agent* nicht vorzeitig auf suboptimale Spielweisen fixierte, sondern kontinuierlich alternative Taktiken ausprobierte. Erst ab dem Zeitpunkt der Reduzierung des Koeffizienten (gestrichelte Linie) brach die Exploration signifikant ein. Der *Agent* ging nun in eine ausgeprägte *Exploitation*-Phase über und griff verstärkt auf die bereits erlernten, verlässlichen Strategien zurück.

Diese Dynamik spiegelt sich auch in der Vorhersagegenauigkeit des *Critics* (siehe Abbildung 6.6, untere Grafik) wider. Der *Value Loss* verringerte sich über die Zeit, was belegt, dass der *Critic* die Gewinnwahrscheinlichkeiten des Spielzustands zunehmend fundierter einschätzen konnte. Dabei war zu beobachten, dass der *Critic* bereits während der stark explorativen Phase phasenweise neue, präzisere Niveaus erreichte. Mit der Reduzierung des Entropie-Koeffizienten verzeichnete der *Value Loss* schließlich einen weiteren deutlichen Abfall: Da das Verhalten des *Agenten* nun wesentlich deterministischer und weniger von Zufallszügen geprägt war, wurden auch die Prognosen des *Critics* merklich exakter.

Dieser stabile Konvergenzzustand änderte sich schlagartig zu Beginn von Phase 2 (*Poseidon II*), wie in Abbildung 6.7 zu sehen ist.

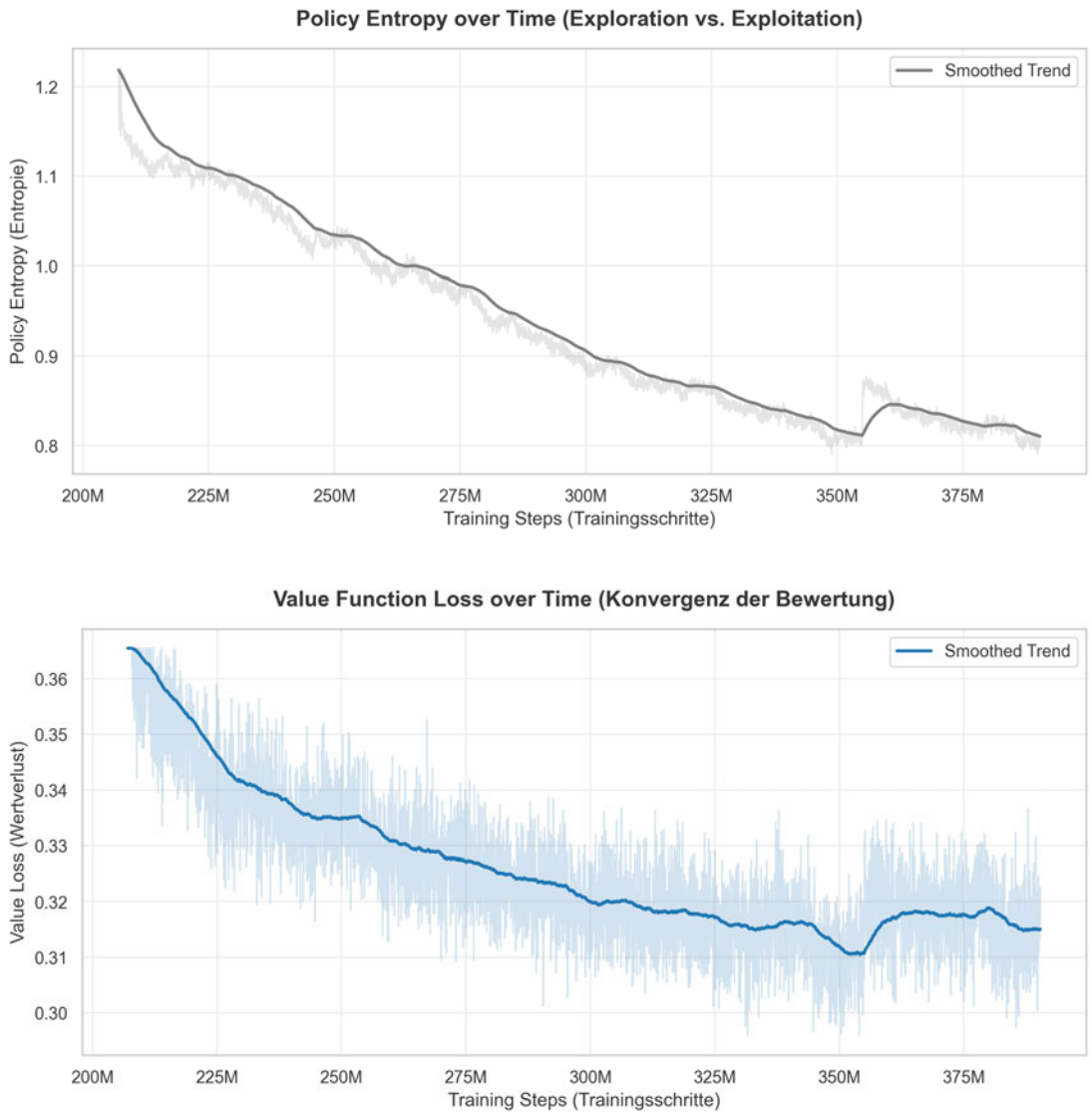


Abbildung 6.7: Konvergenz in Phase 2 (*Poseidon* II). Der gemischte Gegner-Pool sorgt für ein initial höheres Entropie-Niveau. (Eigene Darstellung)

Die Einführung des asymmetrischen Gegner-Pools zwang das Modell, sich an völlig neue, teilweise irrationale Spielweisen (z. B. RandomBot) sowie an lokale, deterministische Heuristiken (GreedyBot) anzupassen. Dies zeigte sich direkt zu Beginn von Phase 2 an einem höheren initialen Entropie-Niveau, da das Modell gezwungen war, seine *Policy* wieder zu öffnen und neu zu explorieren. Bemerkenswert ist zudem der abrupte Ausschlag beider Metriken (rapider Anstieg der Entropie und Verlust der Vorhersagegenauigkeit beim Value Loss) exakt an der Marke von knapp 350 Millionen Schritten. Dieser temporäre „Schock“ im Netzwerk fiel punktgenau mit dem Moment zusammen, in dem der *Agent* die Curriculum-Stufe 26 meisterte und auf Stufe 24 aufstieg (vgl. Abbildung 6.4). Die plötzliche Vergrößerung des Suchraums durch zwei zusätzliche leere Felder zwang den *Agenten* zu einer heftigen, aber kurzen Neuorientierung. Dass die Metriken

im Anschluss wieder kontinuierlich abflachten, belegt die erfolgreiche Adaption von *Poseidon* II an diese komplexere Brettsituation.

6.3 Quantitative Spielstärke: Benchmark-Ergebnisse

Um die finale Leistung objektiv zu bewerten, wurden automatisierte Benchmarks mit jeweils 1.000 Spielen pro Matchup durchgeführt. Abbildung 6.8 zeigt die aggregierten Gewinnraten.

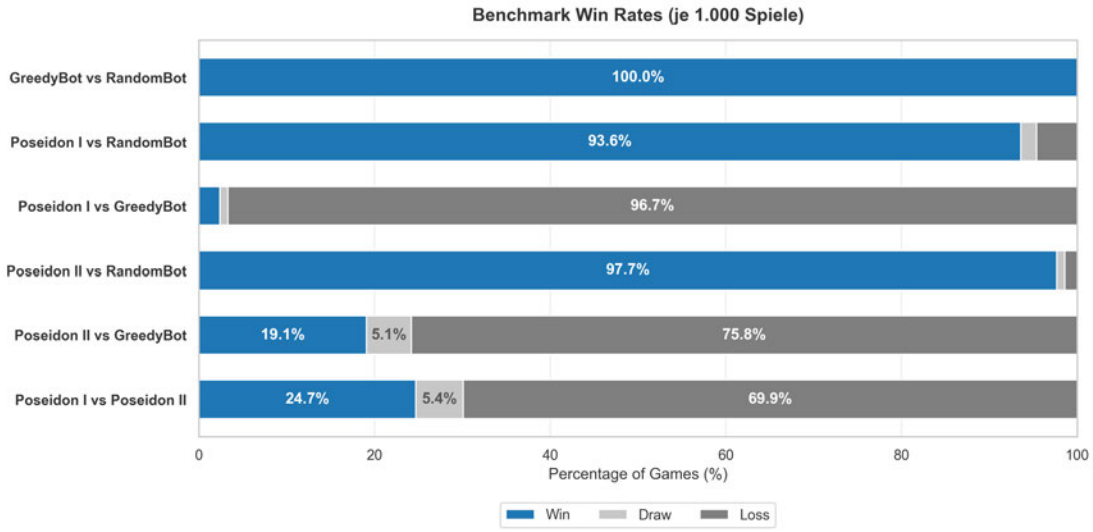


Abbildung 6.8: Benchmark Win Rates über jeweils 1.000 Spiele. Werte $< 5\,\%$ sind aus Platzgründen unbeschriftet. (Eigene Darstellung)

Wie erwartet dominiert der GreedyBot den RandomBot fehlerfrei (100 % Winrate). Der erste Agent, *Poseidon I*, erreicht gegen den RandomBot eine überzeugende Gewinnrate von 93,6 %, scheitert jedoch fast vollständig am GreedyBot (nur 2,4 % Siege).

Der aus Phase 2 hervorgegangene Agent *Poseidon II* belegt den Erfolg des gemischten Gegner-Pools: Gegen den RandomBot steigert er seine Winrate auf 97,7 %. Gegen den GreedyBot gelingt es *Poseidon* II, in knapp jedem fünften Spiel (19,1 %) als Sieger hervorzugehen. Im direkten Duell der beiden Agenten-Generationen schlägt *Poseidon* II seinen Vorgänger überdeutlich mit einer Winrate von rund 70 %.

6.4 Asymmetrie-Analyse: Farben vs. Tierarten

Das Brettspiel *Aqualin* ist strukturell asymmetrisch: Gemäß den festen Spielregeln hat Player 1 stets den ersten Zug und sammelt Punkte ausschließlich durch gleichfarbige Gruppen. Player 2 fungiert als nachziehender Spieler und punktet durch Tierarten. Obwohl *Poseidon* während des Self-Plays beide Rollen im ständigen Wechsel übernahm, wurde untersucht, ob sich das Modell im Laufe des Trainings auf eine der beiden Spielweisen festgefahren hat. Dabei lag der Fokus auf der Frage, ob der *Agent* aufgrund

der strikten Verknüpfung von Startvorteil und Fraktionsart eine der beiden Rollen signifikant bevorzugt und die andere vernachlässigt.

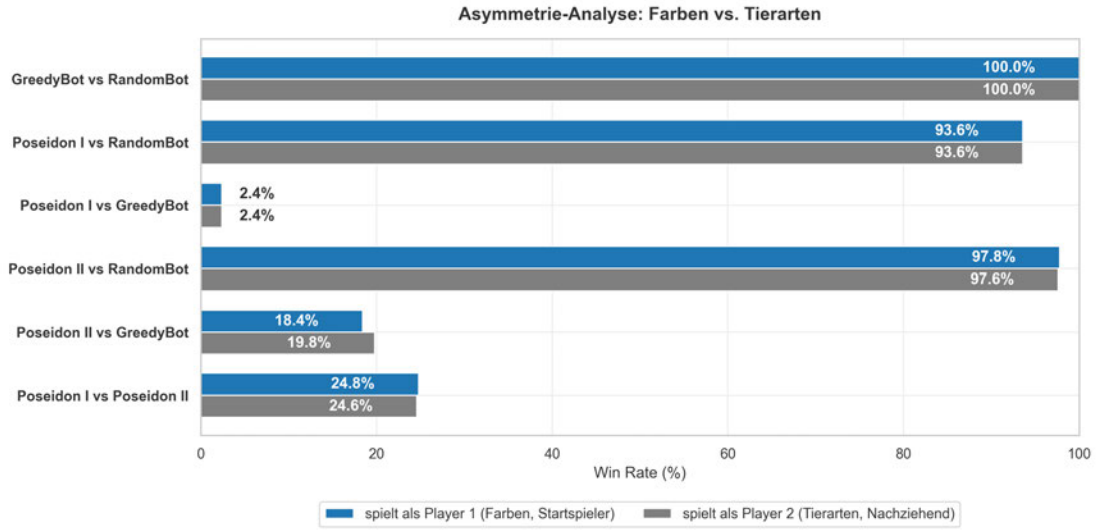


Abbildung 6.9: Gewinnraten getrennt nach Spieler-Rolle (Player 1 / Farben vs. Player 2 / Tierarten). (Eigene Darstellung)

Die Analyse (siehe Abbildung 6.9) zeigt, dass die DRL-Architektur dieses Problem fehlerfrei bewältigt. Die Abweichungen in den Gewinnraten zwischen der Rolle als Farben- oder Tierspieler liegen in sämtlichen Matchups bei unter zwei Prozentpunkten. *Poseidon* (sowohl in Phase I als auch in Phase II) hat das Spiel als Ganzes generalisiert und erzielt vollkommen unabhängig von seiner Startposition oder Fraktion dieselbe statistische Leistung.

6.5 Mensch-Maschine-Evaluierung

Um die quantitativen Metriken der *Agenten* in einen praxisnahen Kontext zu setzen, wurde abschließend eine Studie mit menschlichen Probanden durchgeführt. Ziel dieser Untersuchung war es, die von beiden *Poseidon*-Modellen (Phase I und II) erlernte Spielstärke unter realen Bedingungen gegen menschliche Intuition zu validieren.

Versuchsaufbau und technische Umsetzung

Die methodische Durchführung der Studie setzte zwingend eine grafische Benutzeroberfläche voraus, um den menschlichen Testern eine intuitive Bedienung per Mausklick zu ermöglichen. Da die vollständige Eigenentwicklung einer GUI den inhaltlichen und zeitlichen Rahmen dieser Bachelorarbeit jedoch deutlich überschritten hätte, wurde für deren Erstellung auf die Unterstützung durch Große Sprachmodelle (Large Language Models, siehe Erklärung zur KI-Nutzung) zurückgegriffen. Die Modelle generierten auf Grundlage der eigens entwickelten Kernskripte (siehe Kapitel 5) eine funktionale *Pygame*-Anwendung. Die im Vorfeld geschaffene Application Programming Interface (API) sorgte im Hintergrund dafür, dass alle Spielzüge nahtlos zwischen der GUI, der Aqualin-Umgebung und den neuronalen Netzen synchronisiert wurden.



Abbildung 6.10: Die für die Studie verwendete Testumgebung. Um realistische Testbedingungen zu schaffen, steuerten die Probanden ihre Spielzüge mittels einfacher Klick-Interaktionen über diese Oberfläche. (Eigene Darstellung)

Jeder Proband absolvierte nach einer kurzen Einweisung in die Spielregeln und die Steuerung eine standardisierte Testreihe. Diese umfasste vier aufeinanderfolgende Spiele gegen folgende, dem Spieler vorerst anonymisierte, Gegner:

- **RandomBot**: Zur Überprüfung, ob der menschliche Spieler die Grundmechaniken und die Punktewertung verstanden hat.
- **GreedyBot**: Als Repräsentant einer rein deterministischen, algorithmischen Such-Heuristik.
- ***Poseidon* I**: Das Baseline-Modell aus Phase 1 (reines Self-Play).
- ***Poseidon* II**: Das finale Modell, das im asymmetrischen Gegner-Pool abgehärtet wurde.

Um eine systematische Benachteiligung auszuschließen, wurde den Probanden in jedem Spiel zufällig die Rolle als Startspieler (Farben) oder Nachziehender (Tiere) zugewiesen. Die entwickelte Anwendung protokollierte im Hintergrund automatisiert die relevanten Metadaten der Tester (Alter, Geschlecht, Selbsteinschätzung der Brettspiel-Erfahrung

auf einer Skala von 1 bis 5) sowie die finalen Spielstände als *Comma-Separated Values (CSV)*.

Ergebnisse der Probandenstudie

An der Studie nahmen insgesamt zehn Probanden teil (fünf weiblich, fünf männlich), wobei das Durchschnittsalter bei 40,6 Jahren lag. Die Spielerfahrung wurde auf einer Fünf-Punkte-Skala im Mittel mit einem Wert von 3,2 angegeben, was auf ein solides grundlegendes Verständnis für Gesellschaftsspiele hinweist. Insgesamt wurden in diesem Rahmen 40 verwertbare Spiele absolviert.

Die Ergebnisse (siehe Abbildungen 6.11 und 6.12) zeigen ein deutliches Gefälle in der Spielstärke der verschiedenen Gegner-Versionen. Gegen den *RandomBot* gewannen die menschlichen Tester alle Partien (100 % Siegquote) und dominierten das Spielgeschehen mit einem durchschnittlichen Punktevorsprung (Median) von über 30 Punkten. Dies bestätigt, dass die Probanden die Kernmechaniken von Aqualin schnell adaptieren konnten.

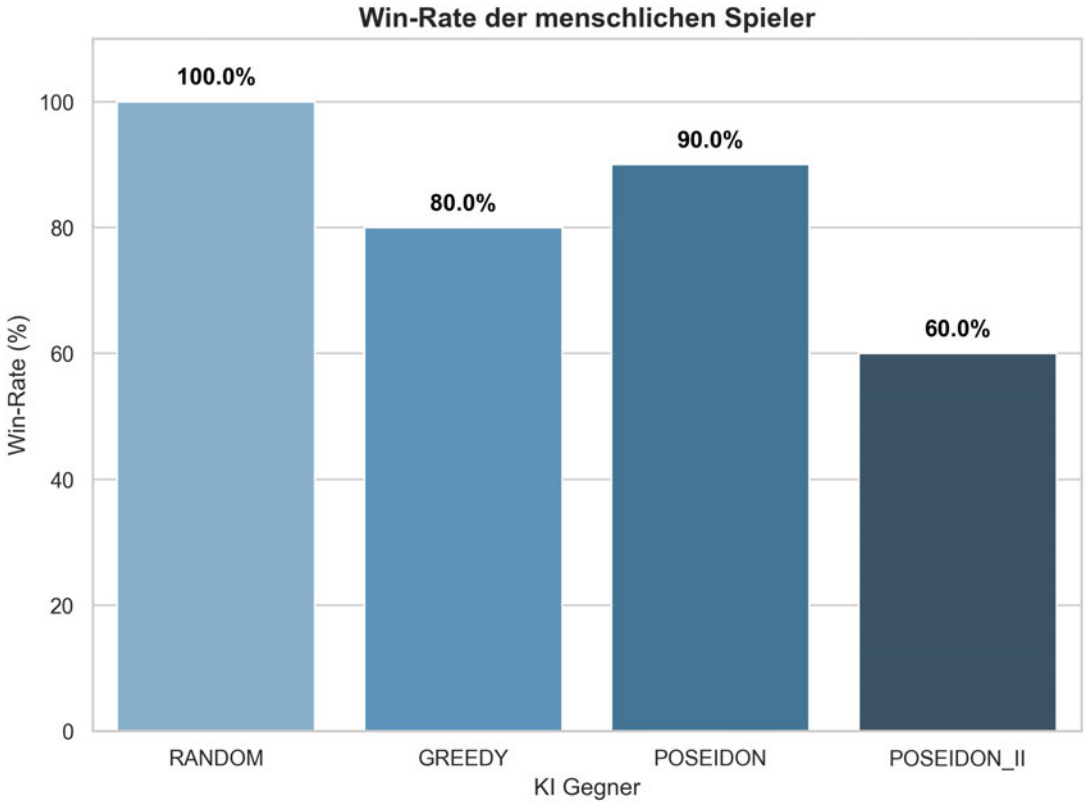


Abbildung 6.11: Siegquote der menschlichen Spieler in der Studie gegen die verschiedenen KI-Gegner. (Eigene Darstellung)

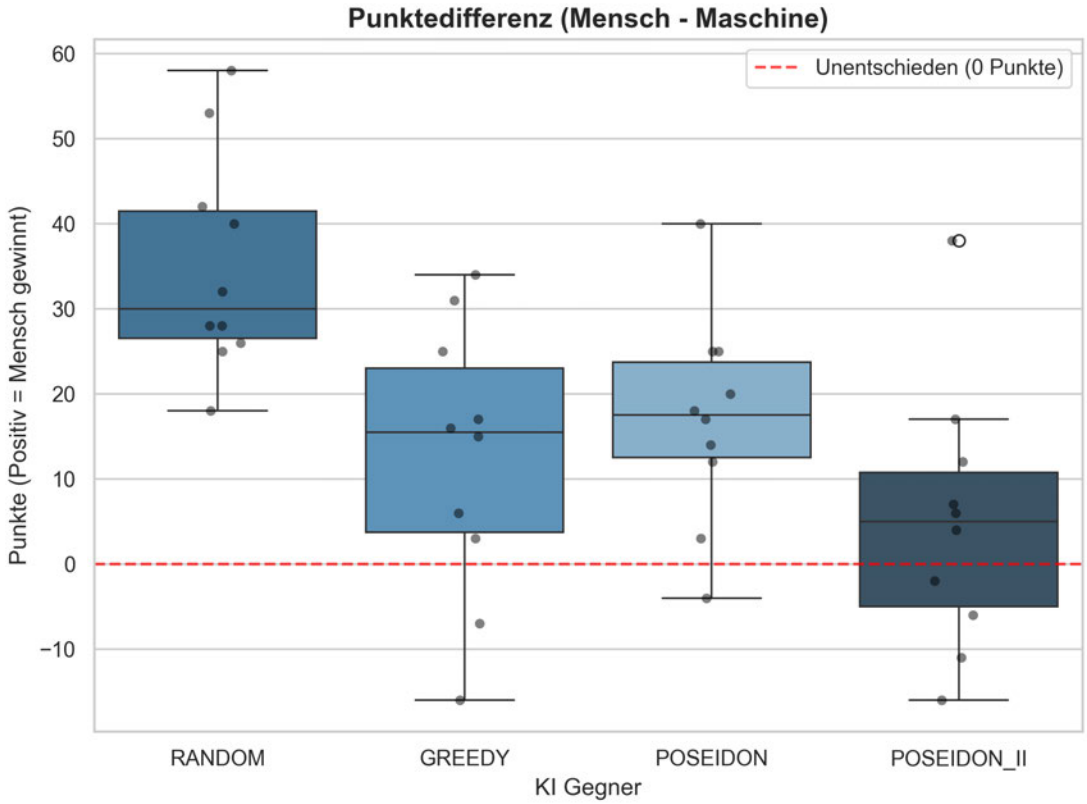


Abbildung 6.12: Punktedifferenz pro Spiel in der Mensch-Maschine-Studie. Werte über der roten Null-Linie kennzeichnen einen Sieg des menschlichen Spielers. (Eigene Darstellung)

Wie bereits aus den automatisierten Benchmarks (vgl. Kapitel 6.3) zu erwarten war, erwies sich die erste RL-Generation (*Poseidon I*) – nach dem *RandomBot* – als zweitschwächster Gegner für die Probanden. Mit einer Siegquote von 90 % zugunsten des Menschen war das *Baseline*-Modell noch leichter zu schlagen als die deterministische Suchheuristik des *GreedyBots* (80 % Siegquote für den Menschen). Zwar konnte *Poseidon I* eine Partie knapp für sich entscheiden, der Median des Punktevorsprungs für den menschlichen Spieler blieb mit rund 17 Punkten jedoch verhältnismäßig hoch. Das bestätigt die bisherige Erkenntnis: Reines Training gegen sich selbst führt zu einseitigen Strategien, die ein Mensch intuitiv leicht durchschauen und ausnutzen kann.

Ein massiver Leistungssprung zeigt sich jedoch bei der Auswertung von *Poseidon II*. Durch das erweiterte Training im asymmetrischen Gegner-Pool fiel die menschliche Siegquote drastisch auf 60 %. Besonders bemerkenswert ist dabei folgendes Detail: Obwohl der *GreedyBot* in den reinen Maschinen-Benchmarks deutlich stärker gegen *Poseidon II* abschnitt (ca. 81 % Siegquote für den Bot), taten sich die menschlichen Spieler gegen *Poseidon II* wesentlich schwerer als gegen den *GreedyBot*. Beobachtungen der Testpartien erklären dieses Phänomen: Der *GreedyBot* agierte stellenweise extrem statisch und wiederholte in lokalen Konflikten (z. B. beim wiederholten Auflösen eines 4er-Blocks durch den Menschen) oft denselben Zug, da dieser für die Heuristik den

höchsten unmittelbaren Punktwert aufwies. Dies ermöglichte den Probanden einfache, repetitive Konter. *Poseidon II* hingegen zeigte eine wesentlich flexiblere und dynamischere Spielweise, durch die menschliche Gegner deutlich mehr gefordert wurden, auch wenn das Modell im direkten Duell der Maschinen gegen den *GreedyBot* häufig den Kürzeren zog.

Diese gestiegene Spielstärke spiegelt sich auch deutlich in den Punktedifferenzen wider: Der Median der menschlichen Siege rutschte auf knapp 5 Punkte herunter, was verdeutlicht, dass die Partien stark umkämpft verliefen. Zudem weist der Boxplot (siehe Abbildung 6.12) nun eine Verschiebung in den negativen Bereich auf (bis zu -15 Punkte), was belegt, dass der *Agent* menschliche Spieler in mehreren Partien mit einem sehr deutlichen Punktevorsprung besiegen konnte. Ein einzelner Ausreißer nach oben (nahe $+40$ Punkte für den Menschen) zeigt zwar auf, dass das Modell in seltenen Sonderfällen noch immer aus dem Konzept gebracht werden kann. Da an der Studie lediglich zehn Probanden teilnahmen, stellen diese quantitativen Ergebnisse zwar keinen absoluten statistischen Beweis dar, sie liefern jedoch eine klare Tendenz: Die algorithmischen Anpassungen in Phase 2 haben erfolgreich zu einem deutlich robusteren *Agenten* geführt, der im direkten Duell mit menschlicher Intuition auf einem sehr kompetitiven Niveau agiert.

Asymmetrie der menschlichen Wahrnehmung

Ein weiteres, interessantes Ergebnis lieferte die Aufschlüsselung der menschlichen Siegquoten nach den jeweils zugewiesenen Spielrollen (Farben vs. Tiere). Die Zuteilung erfolgte in der Studie zufällig und führte zu einer ausgewogenen Verteilung über alle KI-Gegner hinweg. Wie Abbildung 6.13 zeigt, trat insbesondere in den Duellen gegen die RL-Modelle (*Poseidon I* und *II*) eine Diskrepanz auf. Am deutlichsten zeigte sich dieser Effekt gegen den stärksten Gegner (*Poseidon II*): Traten die Probanden in der Rolle der Farben an, gewannen sie 80 % der Spiele. In der Rolle der Tiere fiel ihre Siegquote hingegen auf 40 % (vergleiche hierzu die grafische Benutzeroberfläche in Abbildung 6.10).

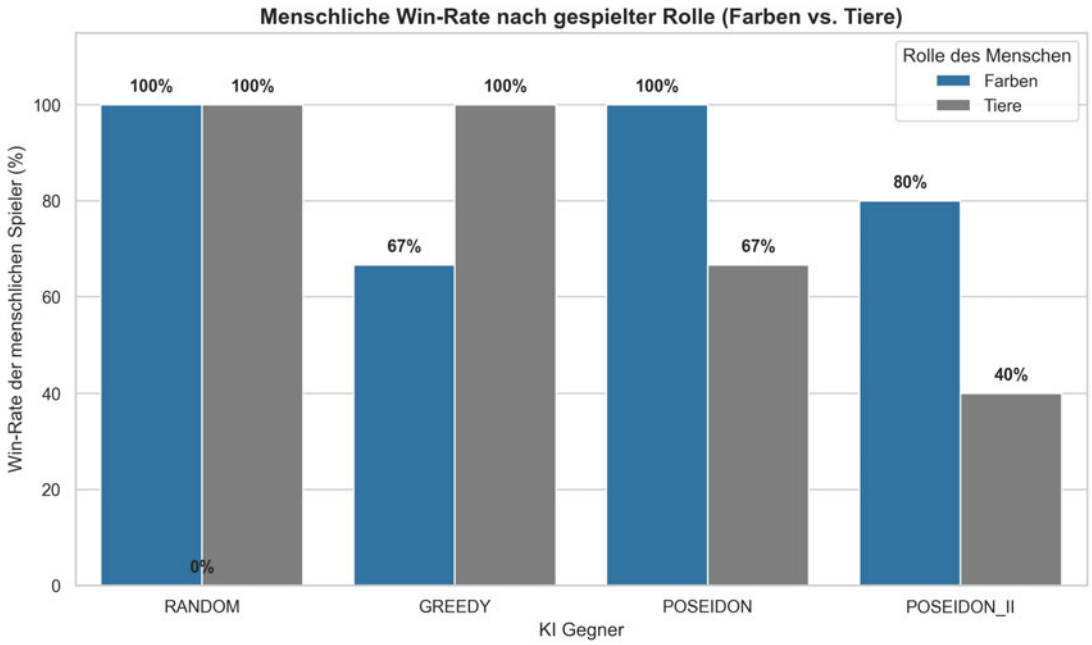


Abbildung 6.13: Menschliche Siegquoten getrennt nach der gespielten Rolle (Farben vs. Tiere). Besonders gegen das stärkste Modell (*Poseidon II*) bricht die Siegquote der Menschen in der Tier-Rolle massiv ein. (Eigene Darstellung)

Das qualitative Feedback der Probanden liefert eine Erklärung für diese Diskrepanz: Zahlreiche Tester gaben an, dass es ihnen deutlich leichter fiel, großflächige Farbblöcke auf dem Brett visuell zu erfassen und zu gruppieren, als auf die feingliedrigeren Tiersymbole zu achten. Während das neuronale Netz (wie in Kapitel 6.4 quantifiziert) das Spiel völlig unvoreingenommen generalisiert hat und keinen Leistungsunterschied zwischen den Rollen aufweist, unterliegen menschliche Spieler offenbar einer visuell-kognitiven Präferenz für Farben. Auch wenn die geringe Stichprobengröße von zehn Probanden keine weitreichenden statistischen Schlüsse zulässt, ist in den erhobenen Daten dennoch eine deutliche Tendenz erkennbar, die sich mit den subjektiven Eindrücken der Tester deckt: Die visuell anspruchsvollere Spielerrolle des Tiersammlers führt zu einer spürbaren Leistungsminderung auf Seiten des Menschen, die insbesondere *Poseidon II* für sich nutzen konnte.

7 Diskussion

Die vorliegende Arbeit hat untersucht, inwiefern sich Methoden des *Deep Reinforcement Learning* (spezifisch der PPO-Algorithmus) eignen, um das asymmetrische Brettspiel *Aqualin* von Grund auf zu erlernen. In diesem Kapitel werden die Ergebnisse aus Kapitel 6 interpretiert, kritisch hinsichtlich ihrer Limitierungen reflektiert und mögliche Ansätze für zukünftige Arbeiten aufgezeigt.

7.1 Interpretation der Ergebnisse

Die Ergebnisse der automatisierten Benchmarks (siehe Abschnitt 6.4) sowie die durchgeführte Probandenstudie belegen eindeutig, dass das Architekturkonzept funktioniert und die Modelle, sowohl *Poseidon I* als auch *Poseidon II*, das Spielprinzip tiefgreifend verstanden haben. Dies spiegelt sich nicht zuletzt darin wider, dass das Modell in den Evaluierungspartien erfolgreich gegen menschliche Spieler bestehen konnte.

Der direkte Vergleich zwischen *Poseidon I* und *Poseidon II* demonstriert jedoch die kritische Relevanz des Gegner-Pools. Reines *Self-Play* (*Poseidon I*) führt den *Agenten* in ein lokales Optimum, in dem er zwar den Zufalls-Agenten dominiert (93,6 % Winrate), dem Suchalgorithmus des GreedyBots aber fast vollständig unterlegen ist (2,4 % Winrate). Ein möglicher Grund hierfür ist eine zu geringe Gegnervarianz: Bedingt durch limitierte Hardwareressourcen (verfügbarer Arbeitsspeicher) kann stets nur eine begrenzte Anzahl an vergangenen Netz-Iterationen gleichzeitig im Speicher gehalten werden. Die daraus resultierende mangelnde Diversität der Spielstile begünstigt das Verfangen in einem Optimum stark.

Die Integration des asymmetrischen Gegner-Pools in Phase 2 erzwang ein Aufbrechen dieser erstarrten Policy. *Poseidon II* konnte seine Leistung gegen den GreedyBot dadurch signifikant auf $19,1\,\%$ steigern und schlägt seinen Vorgänger *Poseidon I* im direkten Duell deutlich (ca. 70 % Winrate). Zieht man in Betracht, dass das gesamte Training auf einem einzigen regulären Arbeitsrechner innerhalb von nur knapp 3 Wochen absolviert wurde, ist die erreichte Spielstärke von *Poseidon II* äußerst beachtlich. Insbesondere unter der Prämisse, dass es sich um ein rein reaktives neuronales Netz ohne vorausschauende Such-Heuristik (wie eine Monte-Carlo Tree Search) handelt, demonstriert das Modell eine erstaunliche Fähigkeit, komplexe Brettzustände intuitiv korrekt zu bewerten.

7.2 Kritische Reflexion und Limitierungen

Obwohl sowohl *Poseidon I* als auch *Poseidon II* das Spiel grundlegend verstanden haben und ein beachtliches Spielniveau aufweisen, unterliegt die entwickelte Architektur gewissen Limitierungen. Der Agent agiert als *Model-Free* Algorithmus rein reaktiv. Die CNN-Architektur schätzt lediglich den Wert des aktuellen Brettzustands (*Value*) und

leitet daraus die beste nächste Aktion (*Policy*) ab, ohne jedoch die zukünftigen Züge explizit vorauszuberechnen. Ein Suchalgorithmus (wie der GreedyBot) kann Züge *im Voraus* simulieren, während *Poseidon* (I und II) vollständig auf die „Intuition“ seiner trainierten Gewichte angewiesen ist.

Darüber hinaus offenbarte das Training die für *Model-Free Reinforcement Learning* typische Sample-Ineffizienz im direkten Vergleich zum menschlichen Lernen. Während menschliche Spieler die Grundmechaniken und Taktiken oft nach wenigen Partien erfassen, erforderte die finale Spielstärke von *Poseidon II* das Simulieren von annähernd 400 Millionen Zeitschritten, was über 10 Millionen absolvierten Partien entspricht. Dieses Phänomen ist in der Literatur wohlbekannt; so benötigten auch Meilensteine wie *AlphaZero* gewaltige, verteilte Rechenzentren und hunderte Millionen an Self-Play-Partien, um zu konvergieren.

Ein weiterer zentraler Aspekt dieser Reflexion betrifft den enormen Entwicklungsaufwand, der im Vorfeld des Trainings oftmals unterschätzt wird. Die Überführung eines für Menschen scheinbar simplen Lege-Brettspiels in ein maschinell lernbares Format erwies sich als hochkomplexes Engineering-Problem. Erst durch die orchestrierte Kombination essenzieller Hilfssysteme (namentlich dem strikten *Invalid Action Masking*, der feingranularen semantischen Separierung des Brettzustands in einzelne Beobachtungskanäle (*Semantic Vision*) sowie der schrittweisen Heranführung über das *Curriculum Learning*) gelang es, das Netzwerk überhaupt in die Lage zu versetzen, die komplexe Spielmechanik zu erfassen und zielführend zu lernen. Dieser Umstand unterstreicht eindrücklich, wie viel domänenspezifisches Design und Feature-Engineering selbst bei modernen DRL-Algorithmen erforderlich ist, um ein künstliches neuronales Netz erfolgreich an ein neues Problemfeld zu adaptieren.

7.3 Ausblick

Aufbauend auf den Ergebnissen dieser Arbeit eröffnen sich spannende Möglichkeiten für zukünftige Entwicklungen. Der vielversprechendste nächste Schritt ist die Integration einer MCTS, um *Poseidon II* von einem reaktiven zu einem planenden Agenten zu transformieren. Das in dieser Arbeit erfolgreich trainierte Value-Netzwerk von *Poseidon II* stellt dafür das perfekte Fundament dar: Anstatt tiefe Rollouts bis zum Spielende spielen zu müssen, kann eine MCTS-Suche das CNN nutzen, um Blattknoten im Suchbaum präzise zu evaluieren. Dies ist exakt der hybride Ansatz, der *AlphaZero* seinen Durchbruch bescherte.

Auch aus softwaretechnischer Sicht bietet das geschaffene Framework viel Potenzial. Die entwickelte Pygame-GUI und die trainierten Modelle ließen sich in eine mobile Applikation überführen, um Aqualin-Spielern einen starken und stets verfügbaren digitalen Trainingspartner zur Seite zu stellen.

8 Fazit

Das Hauptziel dieser Bachelorarbeit war es, eine funktionierende *Reinforcement-Learning*-Architektur zu entwickeln, die das Brettspiel *Aqualin* von Grund auf erlernt. Dafür wurde der *Proximal Policy Optimization* (PPO)-Algorithmus mit neuronalen Netzen (CNNs) kombiniert. Aqualin besitzt eine grundlegende strukturelle Asymmetrie: Ein Spieler sammelt Farben, der andere Tierarten. Genau diese Asymmetrie macht das Spiel für KI-Agenten besonders anspruchsvoll. Die Kernfrage lautete daher: Schafft es ein reaktiver Algorithmus ohne vorausschauende Baumsuche, nur durch reines *Self-Play* eine kompetitive Spielstärke aufzubauen?

Um das herauszufinden, wurde eine maßgeschneiderte Python-Umgebung entwickelt. Durch Ansätze wie *Semantic Vision* und *Invalid Action Masking* gelang es, die räumlichen Muster für neuronale Netze effizient verständlich zu machen und den riesigen Aktionsraum des Spiels gezielt einzugrenzen. Das Training selbst war iterativ und erforderte zur Überwindung anfänglicher Hürden (wie *Catastrophic Forgetting*) ein dediziertes Hyperparameter-Tuning in Kombination mit *Curriculum Learning*.

Das erste lauffähige Modell, *Poseidon I*, wurde rein im *Self-Play* trainiert. Zwar konnte es Zufallsgegner verlässlich besiegen, konvergierte jedoch in einem lokalen Optimum und scheiterte an der abweichenden Spielweise des heuristischen *GreedyBots*. Um dieses Problem zu lösen, wurde *Poseidon II* in einem gemischten Gegner-Pool weitertrainiert. Dieser Ansatz war erfolgreich: *Poseidon II* löste sich aus dem lokalen Optimum, schlug seinen Vorgänger in rund 70 % der Partien und konnte auch dem starken *GreedyBot* wiederholte Siege (knapp 20 %) abringen.

Die abschließende Probandenstudie unterstrich diesen Erfolg eindrücklich. Es zeigte sich, dass die flexible, reaktive Spielweise von *Poseidon II* für menschliche Probanden deutlich schwerer greifbar war als die starre Logik des *GreedyBots*. Das Modell hat das asymmetrische Spielprinzip völlig selbstständig erlernt und ein sichtbares Leistungsniveau erreicht, mit dem es in der Lage ist, auch gegen menschliche Anfänger Siege zu erringen.

Zusammenfassend lässt sich festhalten: PPO-basierte Architekturen eignen sich nachweislich, um komplexe asymmetrische Brettspiele wie Aqualin zu erlernen. Auch wenn für die uneingeschränkte Dominanz gegen deterministische Suchalgorithmen wahrscheinlich eine zusätzliche Baumsuche oder erheblich mehr Rechenkapazität vonnöten wären, liefert diese Arbeit ein vollumfänglich funktionsfähiges Fundament. Die entwickelte Umgebung sowie die trainierten Modelle bilden eine solide Basis, an die künftige Forschungsarbeiten nahtlos anknüpfen können.

Literaturverzeichnis

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer, New York.

Campbell, M., Hoane, J., and Hsu, F.-h. (2002). Deep blue. *Artificial Intelligence*, 134(1-2):57–83.

Chrislb (2005). Schema eines künstlichen neurons. Wikimedia Commons. Abgerufen am 18. April 2026, lizenziert unter GFDL.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms*. MIT press, 4th edition.

Finbridge GmbH & Co. KG (2022). Computer vision für finance - teil 2. https://www.finbridge.de/ml-artikel/2022/09/08/computer-vision-fuer-finance-teil-2. Abgerufen am 27. Juli 2026.

Franckh-Kosmos Verlags-GmbH & Co. KG (2020). Aqualin – schwarmtaktik für zwei. Produktseite und Spielbeschreibung, Zugriff am 12.03.2026.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Huang, S. and Ontañón, S. (2022). A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pages 448–456.

Leela Chess Zero Team (2021). Leela chess zero. urlhttps://lczero.org/.

Lim, H.-K., Oh, S.-W., Kim, J.-B., Lee, D. J., and Han, Y.-H. (2020). Federated reinforcement learning for training control policies on multiple iot devices. *Sensors*, 20(5):1359.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 1928–1937. PMLR.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. Online verfügbar; abgerufen am 12. April 2026.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Standage, T. (2002). *The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine*. Walker & Company, New York.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition.

Terry, J. K., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., Santos, L. S., Dieffendahl, C., Williams, C. L., Lokesh, Y., Sullivan, C., and Morrow, C. (2021). Pettingzoo: Gym for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 15032–15043.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, wurden als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum Unterschrift

Erklärung zur Nutzung von Künstlicher Intelligenz

Bei der Erstellung dieser Bachelorarbeit und der dazugehörigen Software-Implementierung wurde das KI-basierte Assistenzsystem **Gemini 3.1 (Pro)** als unterstützendes Werkzeug eingesetzt. Die Nutzung erfolgte strikt im Rahmen der prüfungsrechtlich zulässigen Verwendungszwecke (Werkzeugfunktion). Die inhaltliche Kontrolle und die geistige Eigenleistung verblieben stets vollständig beim Autor.

Im Detail wurde das genannte System für folgende Zwecke verwendet:

- **Rechtschreib-, Grammatik- und Stiloptimierung:** Prüfung eigener Textentwürfe auf Tippfehler sowie die wissenschaftliche und stilistische Glättung von Formulierungen.
- **Code- und Formatierungshilfe:** Unterstützung bei der Syntax-Formatierung in LaTeX, Hilfestellung bei der Fehlersuche (Debugging) in Python-Skripten sowie Unterstützung beim Refactoring von Code-Strukturen für das Reinforcement-Learning-Framework.
- **Brainstorming und Strukturierung:** Einholen von Impulsen zur inhaltlichen Strukturierung der Kapitel (Gliederung) sowie Prüfung von Argumentationsketten auf logische Konsistenz.
- **Verständnishilfe:** Herunterbrechen komplexer mathematischer Paper und abstrakter Deep-RL-Konzepte zur Förderung des eigenen theoretischen Verständnisses.
- **Generierung peripherer Software-Komponenten:** KI-gestützte Entwicklung der grafischen Benutzeroberfläche (Pygame-GUI) für die Probandenstudie auf Basis der eigens entwickelten Kern-API. Dies diente ausschließlich der Bereitstellung eines Test-Interfaces und stellt bewusst keinen Teil der zu bewertenden Kernarchitektur dar.

Ich versichere, dass sämtliche KI-generierten Outputs vor der Übernahme in die Arbeit kritisch geprüft, in den Kontext der eigenen Forschung eingeordnet und eigenverantwortlich angepasst wurden. Die wissenschaftlichen Kerngedanken, das Architekturdesign der neuronalen Netze, die Durchführung der Trainingsläufe sowie die Interpretation der Ergebnisse stellen meine ungeteilte Eigenleistung dar.

Ort, Datum Unterschrift