DIPLOMARBEIT

Erstellung und Evaluierung einer auf neuronalen Netzen basierenden, kognitiven Architektur

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Projektass. Dipl.-Ing. Dr.techn. Alexander Wendt

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Rupert Schneeberger
Matr.Nr. 0426426
Hyrtlgasse 34/4, A-1160 Wien

Wien, am 16. November 2018

Kurzfassung

In den vergangenen Jahren ist weltweit zunehmend größeres Interesse an künstlicher Intelligenz und ihrer Nutzung in Wirtschaft und Alltag zu bemerken. Für die Lösung und Implementierung wurden in der Forschung verschiedene Ansätze entwickelt. Klassischerweise wurden in den vergangenen Jahrzehnten künstliche Intelligenzen zumeist pseudo-intelligent durch fest einprogrammierte Regeln ausgeführt. Künstliche neuronale Netze sind eine Möglichkeit einen Agenten selbständig lernfähig zu machen, mit der sich vorliegende Arbeit beschäftigt.
Daraus ergibt sich die Frage, wie neuronale Netze genutzt werden können, um einem kognitiven Agenten die Fähigkeit zu verleihen, möglichst selbständig und ohne spezielle Anpassungen im Programmcode gegebene Aufgaben zu lösen, wobei der Agent inkrementell von Versuch zu Versuch dazulernt. Ziel dieser Arbeit ist die Schaffung einer kognitiven Architektur, welche eben hierzu neuronale Netze nutzt. Als Anwendungsgebiet für diese Architektur dient ein simples Computerspiel, in welchem Agenten gegeneinander spielen und möglichst viele Punkte erhalten sollen. Ein Agent, der die kognitive Architektur verwendet, soll in diesem Spiel mehr Punkte erreichen, als Agenten mit zur Verfügung gestellten Benchmark-Architekturen.
Die Arbeit umfasst sowohl den Aufbau der kognitiven Architektur, als auch ihre Evaluierung. Die Architektur wurde in verschiedenen Szenarien und mit unterschiedlichen Parametern des neuronalen Netzes und der Spielkonfiguration auf die Fähigkeit des Agenten, Punkte im Spiel lukrieren zu lernen, getestet. Die Ergebnisse illustrieren Stärken und Schwächen der vorgestellten Architektur. Insgesamt waren die Versuche erfolgreich, da Agenten mit dieser Architektur im Durchschnitt höhere Punktestände als die Benchmark-Agenten erzielen. Abgesehen davon wird über Anwendungsgebiete einer solchen Architektur und weitere mögliche Forschungsschritte spekuliert.

Abstract

Recent years have seen a steady rise in the world-wide interest towards artificial intelligence and its economic and everyday benefits. This field of research has seen the development of diverse solutions and implementations. In the past intelligent designs were usually little more than pseudo-intelligent by programming fix if-then rules. Artificial neural networks are one possible means to enable agents to learn independently. This thesis explores how to employ them.
This raises the question how to make use of neural networks in order to enable an agent to learn independently and without significant interference by a programmer to solve given problems, learning incrementally from one instance to another. It's the goal of this thesis to implement a cognitive architecture featuring neural networks in order to achieve this. The architecture is to be used in a provided simple computer game. The game's objective is for agents to compete against each other and reach high scores. An agent using the cognitive architecture should score higher than agents employing benchmark architectures that were provided.
The thesis contains the development and evaluation of the implemented cognitive architecture. It was tested in various scenarios and using different neural network and game parameters to examine the agent's ability to reach high scores. The results show strengths and weaknesses of the proposed architecture. Overall, the tests were successful because agents using the architecture score higher on average than the benchmark ones. Moreover, the thesis discusses the possible fields of application for such an architecture and speculates about possible further research in that area.

Inhaltsverzeichnis

Abkürzungen

ACT-R	Adaptive Control of Thought – Rational
ANN	Artificial Neural Network (dt.: *künstliches neuronales Netz*)
BP	Backpropagation
BPTT	Backpropagation Through Time (dt.: *Backpropagation über Zeit*)
CLARION	Connectionist Learning with Adaptive Rule Induction On-line
DL4J	Deep Learning for Java
DQN	Deep Q-Network (dt.: *tiefes Q-Netz*)
DP	Dynamic Programming (dt.: *dynamisches Programmieren*)
GOFAI	Good Old-Fashioned Artificial Intelligence
GUI	Graphical User Interface (dt.: *graphische Benutzeroberfläche*)
KI	Künstliche Intelligenz
LIDA	Learning Intelligent Distribution Agent
LSTM	Long Short-Term Memory (dt.: *langes Kurzzeitgedächtnis*)
MDP	Markov Decision Process (dt.: *Markov-Entscheidungsprozess*)
MOBA	Multiplayer Online Battle Arena
RL	Reinforcement Learning (dt.: *verstärkendes Lernen*)
RL4J	Reinforcement Learning for Java
SiMA	Simulation of the Mental Apparatus & Applications
SLF4J	Simple Logging Facade for Java
STM	Short-term Memory (dt.: *Kurzzeitgedächtnis*)
TD	Temporal Difference

1 Einführung

Die Forschung auf dem Gebiet der künstlichen Intelligenz (KI) macht große Fortschritte, im Zuge derer bereits verschiedene Ansätze erdacht und verfolgt wurden. Einen solchen Ansatz stellen *künstliche neuronale Netze* (engl.: *Artificial Neural Network (ANN)*) dar, mit deren Verwendung in lernfähigen Agenten in einem noch zu spezifizierenden Computerspiel und ihrer Evaluierung sich diese Arbeit beschäftigt. Genaueres zur Motivation ist in Kap. 1.1 zu finden. Die Problemstellung wird eingehend in Kap. 1.2 definiert. Es folgen die Aufgabenstellung in Kap. 1.3 sowie Ausführungen zur Methodologie in Kap. 1.4.

1.1 Motivation

Klassischerweise handelt es sich bei KIs zumeist um solche, welche nach bestimmten Regeln (wenn-dann-sonst) fest vorprogrammiert und damit in ihrem Verhalten festgelegt sind. In der Literatur wird dies auch als *GOFAI (Good Old-Fashioned Artificial Intelligence)* [Dow15, S. 8f.] bezeichnet. Diese Agenten erhalten also eine auf ihren Wirkbereich angepasste Programmierung, die mehr oder weniger komplex sowie mehr oder weniger dynamisch sein kann. Das Hauptproblem, das diese Methode aufweist, ist, dass die Agenten für eine andere als die Umgebung, für die sie geschaffen und angepasst wurden, oft untauglich sind, und manuell an die veränderten Gegebenheiten ausgerichtet werden müssen.

Daraus folgt der Wunsch, Agenten zu schaffen, welche dazu in der Lage sind, selbständig Lösungswege für ein gegebenes Problem zu finden und danach zu handeln. Mit anderen Worten ist es gewünscht, einen Agenten zu schaffen, welcher in der Lage ist, eigenständig ein passendes Verhalten zu einer gegebenen Situation, in die er gestellt wird, zu erlernen. Eine Möglichkeit, um einen solchen Agenten zu erstellen, die sich zur Lösung an der Biologie bzw. an den Erkenntnissen der Neurologie über das Säugetier- und insbesondere menschliche Gehirn orientiert, sind künstliche neuronale Netze. Wie die eigentliche Problemstellung beschaffen ist, wird im nachfolgenden Kap. 1.2 umrissen.

1.2 Problemstellung

Das Problem, mit dem sich vorliegende Arbeit befasst, besteht darin, wie mithilfe *künstlicher neuronaler Netze* (engl.: *Artificial Neural Networks (ANN)*) ein Agent so entwickelt und trainiert

werden kann, dass dieser in vorgegebenen Situationen und Aufgaben im Vergleich zu statisch programmierten KI-Agenten besser abschneidet. Es handelt sich bei den angedeuteten Situationen um eine Spielwelt, wobei die Aufabe darin besteht, möglichst lange darin zu überleben und z. B. durch Nahrungsaufnahme Punkte zu lukrieren. In der Spielwelt sind auch Gegner enthalten, welche um die Ressourcen (Nahrung) konkurrieren und bei Gelegenheit aggressiv agieren. Diese nutzen jene kognitivistischen Architekturen, welche im folgenden Absatz erläutert werden. Das verwendete Spiel trägt den Namen *Foodchase* und ist in Kap. 5.1 im Detail beschrieben.

Eine am Institut für Computertechnik der TU Wien entwickelte Spieleumgebung namens *Miklas* bietet das Programmgerüst, in welchem das Spiel läuft (vgl. Kap. 3.1). Als Grundlage für die Vergleichs-KI-Agenten dient die ebenfalls am Institut entwickelte *SiMA*-Architektur, welche in Kap. 2.1 vorgestellt wird. Es handelt sich um zwei aufgrund von Wenn-Dann-Regeln programmierte Architekturen. Die eine emuliert das Verhalten von SiMA und wird im folgenden *ESiMA* (emulierte SiMA) genannt. Die andere baut auf der ersten auf, ist allerdings speziell auf das gegebene Spiel *Foodchase* angepasst. Im weiteren wird sie mit dem Namen *OSiMA* (optimierte SiMA) betitelt. Folgende Fragen bedürfen der Klärung:

Wie kann eine kognitive Architektur mit neuronalen Netzen aufgebaut werden, sodass ein damit begabter Agent selbständig lernt, auf im Spiel *Foodchase* auftretbare Situationen angemessen zu reagieren?

Dies beinhaltet einige untergeordnete Fragen:

- *Wonach soll entschieden werden können, ob eine durchgeführte Aktion gut oder schlecht ist?* Damit ein Agent lernen kann, wie er erfolgreich sein kann, muss auf irgendeine Weise klargestellt werden, wann Erfolg — oder im Gegenteil: Misserfolg — eintritt.
- *Wie soll das Gedächtnis aufgebaut sein und welche Informationen bedürfen der Aufzeichnung und Speicherung, damit der Agent ein ausreichendes Bild über den Spielablauf hat?* Eine Voraussetzung von Lernen ist die Merkfähigkeit der Vergangenheit. Denn nur an vergangenen Ereignissen lässt sich erkennen, was in der Welt möglich ist, und daraus ableiten, was auch in Zukunft auftreten könnte.
- *Wie sieht der Aufbau des neuronalen Netzes aus? Welche Eingänge und Ausgänge besitzt es? Genügt eines oder ist eine Kombination aus mehreren Netzen gleichen oder unterschiedlichen Typs sinnvoll?*
- *Wie soll das Training von neuronalen Netzen des Agenten beschaffen sein?* Auftretende Ereignisse im Spiel und ihre Konsequenzen für den Agenten sollen diesen veranlassen, aus dem Erfahrenen zu lernen, und sein künftiges Verhalten entsprechend anzupassen.

Welche Ergebnisse erzielen die Benchmark-Architekturen ESiMA und OSiMA im Spiel?

Dies ist der eine Teil der Evaluierung, um die Benchmark-Architekturen mit der ANN-Architektur vergleichen zu können. Bewertet wird nach dem erreichten Endpunktestand der gegeneinander antretenden Agenten.

Kann ein Agent mit neuronalen Netzen erlernen, in einem gegebenen einfachen Computerspiel im Durchschnitt höhere Punktzahlen zu erzielen, als ein nach starren Regeln arbeitender Benchmark-Agent?

Die Bemessung erfolgt auch hier anhand des erreichten Punktestandes der Agenten im Spiel. Je höher dieser ausfällt, desto besser.

Kommt es bei dem auftretenden (emergenten) Verhalten des Agenten zu unerwarteten Problemlösungsansätzen?

In der Literatur (Kap. 2.4) existieren Beschreibungen von Vorfällen, bei welchen lernfähige Agenten zu nicht vorhergesehenen bzw. ungewöhnlichen Lösungen ihrer Problemstellung gelangt sind.

Wie weit ist es möglich, die Architektur ohne korrigierenden Eingriff von außen trainieren zu lassen?

In Analogie zum Top-down-/Bottom-up-Gegensatz soll die Architektur Verhalten möglichst bottom-up, also ohne menschliches Zutun und Definition von Spezialregeln nur anhand der vorgegebenen Spielregeln, entwickeln. Die sich stellende Frage ist, wie weit dies machbar ist, und wo spezielle Regeln im Algorithmus nötig (top-down) sind.

Welcher Art soll der Einfluss auftretender Ereignisse sein?

Es kann im Spiel zwischen positiven und negativen Ereignissen unterschieden werden, die einem Agenten widerfahren können. Diese können durch eigene Aktionen oder ohne Zutun des Agenten auftreten. Es ist daher fraglich, wie mit der Unterscheidung umzugehen ist. Inwiefern soll der Lernalgorithmus Positives und Negatives verschieden behandeln?

Kann ein Agent mit eingeschränkter Sicht ausreichendes Wissen über seine Umwelt erlangen, um für den Beobachter nachvollziehbar sinnvolle Aktionen zu setzen?

Wie in Kap. 3.1 gezeigt werden wird, erhalten die Agenten im Spiel visuelle Daten nur in einem beschränkten Gesichtsfeld. Damit besitzen sie kein vollständiges Wissen über den Zustand der Spielwelt.

1.3 Aufgabenstellung

Sinn vorliegender Arbeit ist die Kreierung einer auf neuronalen Netzen fußenden kognitiven Architektur, welche einen Agenten in die Lage versetzt, in einem vorgegebenen Spiel durch Erreichen einer höheren Punktzahl besser zu sein als vorgegebene Benchmark-Agenten, wie schon zuvor in Kap. 1.2 beschrieben. Diese Architektur ist in die *Miklas-Umgebung* eingefügt, welche, wie unten noch näher ausgeführt, erweitert werden muss. Die anschließende Evaluierung der vorgestellten Architekturen dient dem Vergleich der ANN-Architektur mit den Benchmark-Architekturen. Schlussendlich soll ein Agent, welcher die ANN-Architektur nutzt, höhere Punktzahlen im Spiel erreichen als die Benchmark-Agenten. Dies geschieht in verschiedenen Spielkonfigurationen des

verwendeten Spieles, welche dazu dienen sollen, die Fragen aus Kap. 1.2 zu beantworten. Die Beschreibungen und Ergebnisse dieser Konfigurationen sind in Kap. 5.3 zu finden.

Bei der Erstellung der kognitiven Architektur soll diese mit möglichst wenig Regulierung durch menschliche Programmierer lernen, sich in ihrer Spielumgebung zu orientieren. In Analogie kann das als *Bottom-Up*-Ansatz gesehen werden. Im Gegensatz dazu steht der *Top-Down*-Ansatz, welcher die Entsprechung der üblichen GOFAI-Architekturen ist, bei welchen sämtliche Verhaltensaspekte vom Entwickler durchprogrammiert sind. Dies ist auch für die verwendeten Benchmark-Architekturen *ESiMA* und *OSiMA* der Fall. Weiters gehört die Weiterentwicklung der bereits bestehenden Spielumgebung *Miklas* zur Aufgabenstellung. Diese umfasst folgende Punkte:

- Erweiterung der graphischen Benutzeroberfläche (engl.: *Graphical User Interface (GUI)*)
  - Erstellung eines Hauptmenüs mit Schaltflächen zum Starten und Konfigurieren des Spiels sowie zum Beenden des Programms
  - Erstellung eines einfachen Konfigurationseditors, um die schnelle Bearbeitung der Konfiguration des aktuellen Spieles aus dem Programm heraus zu ermöglichen
  - Anfügung einer Statusauflistung der am Spiel partizipierenden Agenten im Spielfenster
  - Erweiterung des Spielfensters um Schaltflächen zur Spiellaufsteuerung (Start/Pause, Einzelschritt, Spielabbruch)
- Mitschnitt des Status der Agenten in einer Log-Datei
- Definition einer Spielabbruchbedingung: Spiel endet, wenn ein einziger Agent übrig bleibt
- Erweiterung von Miklas, um Agenten zusätzlich benötigte Daten über die Spielwelt und den Spielverlauf zu übergeben

Nicht Teil der Aufgabenstellung ist die Entwicklung der Benchmark-Architekturen. Diese wurden bereits fertig zur Verfügung gestellt. Auch das verwendete Spiel *Foodchase*, welches in Kap. 5.1 eingehend erklärt wird, wurde in seinem Aufbau übernommen und für die verschiedenen angestellten Versuche lediglich in seiner Konfiguration entsprechend angepasst.

1.4 Methodologie

Es wird aufgezeigt, wie eine kognitive Architektur unter Verwendung von ANNs erstellt werden kann, sodass der Agent in seiner Umgebung — von Personen unbeaufsichtigt und unbeeinflusst — erlernt, einen möglichst hohen Punktestand im bereits erwähnten Spiel *Foodchase* zu erreichen. Punkte können sowohl für wünschenswerte (Pluspunkte) als auch für negative Aktionen (Minuspunkte) vergeben werden. Der Agent kann im Spiel auch auf Gegner treffen, welche eine der Benchmark-Architekturen (ESiMA bzw. OSiMA) verwenden, mit denen er in einem Konkurrenz-Verhältnis steht. Es gilt das Prinzip: jeder gegen jeden. Alle Agenten verfügen über Gesundheit, welche im Lauf des Spiels abnimmt und durch Nahrungsaufnahme wieder aufgefüllt werden kann. Allerdings können Attacken auch zu Gesundheitsabzug führen. Ist die Gesundheit aufgebraucht, stirbt der Agent. Weitere Informationen zum Spiel sind Kap. 5.1 zu entnehmen. Wie ESiMA bzw. OSiMA ihre Aktionen auswählen, ist Gegenstand von Kap. 5.2.

Im Übrigen stellt die Implementierung der geforderten Veränderungen der *Miklas*-Umgebung die Grundlage für alle weiteren Schritte dar. Diese umfasst, wie der Aufgabenstellung (Kap. 1.3)

zu entnehmen, die Erweiterung der Benutzeroberläche um ein Startfenster mit Spielmenü, einen Konfigurationseditor, sowie Steuerschaltflächen und Spielerstatistiken im Spielfenster. Außerdem ist eine Abbruchbedingung in Miklas zu definieren und implementieren, da eine solche bislang nicht existierte, und dementsprechend Spiele bis zum manuellen Beenden des Programms fortdauerten. Die Bedingung ist so definiert, dass das Spiel endet, sobald nur ein lebender Agent übrig ist. Danach wird das Spielfenster geschlossen und das Spielmenüfenster wieder eingeblendet. Wie *Miklas* aufgebaut ist, ist in Kap. 3.1 dargelegt. Ausführungen zum Quellcode von Miklas vor den Anpassungen sind in Kap. 4.1 und nach den Anpassungen in Kap. 4.2 angegeben.

Die Hauptaufgabe besteht darin, eine kognitive Architektur — im weiteren *NNMind* genannt — zu erstellen, welche ANNs zur Aktionsauswahl von Agenten verwendet. Es muss zunächst der Aufbau der Architektur und die von ihr benutzten ANNs bestimmt werden. Dies ist in Kap. 3.2 ausgeführt. Darauf folgt die Implementierung von *NNMind* in *Miklas*, worüber detaillierte Angaben in Kap. 4.3 gemacht werden.

Die anschließende Evaluierung erfolgt mittels Statistiken des Punktestandes und anderer später noch zu spezifizierenden Größen in verschiedenen Spielkonfigurationen. Diese Konfigurationen umfassen sowohl verschiedene Aufbauten des ANN (unterschiedliche Neuronenzahl, Lernrate, Schichtenanzahl) als auch veränderte Ausgangssituationen des Spiels. Für die Evaluierung wurde *NNMind* gegen zwei verschiedene Benchmark-Architekturen getestet. Die eine ist dem Verhalten der *SiMA*-Architektur (Kap. 2.1) nachempfunden, welche am Institut für Computertechnik der Technischen Universität Wien entwickelt wurde. Sie wird im weiteren *ESiMA* genannt. Die andere Benchmark-Architektur ist für das gegebene Spiel (Foodchase), wie es in Kap. 5.1 dargelegt ist, optimiert. Die Benennung dieser Architektur lautet im folgenden *OSiMA*.

Die aus den Versuchsdurchläufen bei der Evaluierung gewonnenen Daten sind statistisch aufbereitet in Kap. 5.3 präsentiert. Auch die Beschreibungen der unterschiedlichen Versuchskonfigurationen sind hier angegeben, sowie Anmerkungen dazu, welche Situationen und daraus resultierendes erlerntes Verhalten von NNMind bei den Versuchen aufgetreten sind. Diese Anmerkungen fußen auf visuellen Beobachtungen während der Spieldurchläufe sowie den für jeden Zeitpunkt bzw. Spielzug ebenfalls aufgezeichneten Positionen der Agenten auf dem Spielfeld. Dadurch ist auch die Darstellung einiger herausstechender Spieldurchläufe zur Untemauerung und Veranschaulichung der Angaben möglich.

2 Stand der Technik

Es hat in den vergangenen Jahren und Jahrzehnten bereits eine Vielzahl an Arbeiten gegeben, die sich mit dem Einsatz von neuronalen Netzen für verschiedene Aufgaben in Spielen oder auch der Robotik beschäftigen. Zuvorderst soll eine Einführung in die Grundlagen der in der Literatur und dieser Arbeit verwendeten Technologien und Methoden gegeben werden. Zunächst befasst sich Kap. 2.1 aber mit *kognitiven Architekturen*, wozu sie dienen und welche verschiedenen Arten es gibt. Auch wird *SiMA* vorgestellt, auf der die Benchmark-Architekturen fußen. Darauf bietet Kap. 2.2 einen Einblick in die Welt der *künstlichen neuronalen Netze*. Dieser umfasst den grundsätzlichen Aufbau, die Art, wie diese Netze lernen, und die Vorstellung einiger ANN-Typen. Danach erfolgt die Diskussion von Methoden *verstärkenden Lernens* (engl.: *Reinforcement Learning (RL)*) in Kap. 2.3, sowie die Erklärung einiger der wichtigsten Algorithmen dieses Gebiets, da die Ideen von RL für diese Arbeit eine wichtige Rolle spielen. Zuletzt werden einige bereits existente Arbeiten aus der Literatur in Kap. 2.4 aus dem gleichen Themengebiet vorgestellt.

2.1 Kognitive Architekturen

Notwendig ist zunächst die Klärung, was eine kognitive Architektur ist. Es handelt sich hierbei um ein allgemein gehalten entwickeltes Modell, welches die grundlegende Struktur und Abläufe im menschlichen Hirn abbilden soll. Dadurch soll die Untersuchung des Verhaltens künstlicher Intelligenzen auf verschiedenen Ebenen und mit unterschiedlichen Plattformen ermöglicht werden. [Sun07, S. 2]

Es sind in der Vergangenheit vielerlei kognitive Architekturen entwickelt worden, bzw. befinden sich auch noch weiterhin in Entwicklung. Grundsätzlich lassen sie sich in drei Gruppen einteilen: *symbolische*, *emergente* und *hybride* Architekturen. *Symbolische Architekturen* zeichnen sich dadurch aus, dass sie anhand abstrakter Symbole, welche aus den Rohdaten abgeleitet werden können, ihre Aktionen planen. Ein bekanntes Beispiel für eine solche Architektur ist *Soar* [Lai08, S. 225ff.]. Emergente Architekturen hingegen verarbeiten die sensorischen Rohdaten direkt, um das Aktionsverhalten zu bestimmen, wie das bei neuronalen Netzen der Fall ist. Weiters gibt es noch *hybride* Ansätze, welche die Eigenschaften von symbolischen und emergenten Architekturen kombinieren. Zu diesen zählen etwa *ACT-R*, *CLARION* [Sun07, S. 11], *LIDA* [FMDS14, S. 1ff.] und auch *SiMA*, die nachfolgend genauer erläutert wird. [SWK+15, S. 515]

SiMA steht für *Simulation of the Mental Apparatus & Applications* und wurde am Institut für Computertechnik der Technischen Universität Wien entwickelt. Es handelt sich dabei um eine

kognitive Architektur, welche sich im Grunde aus drei Abstraktionsschichten zusammensetzt. Die erste Schicht, die *neuronale Schicht*, stellt die neuronalen Aktivitäten dar, inklusive Sensor- und Aktoranbindung. In der zweiten Schicht, der *neuro-symbolischen Schicht* sind Neurosymbole enthalten, welche die Daten aus der ersten Schicht abstrahieren. Bei der dritten und obersten Schicht handelt es sich um die *Psyche* bzw. *mentale Schicht*, welche für die Verarbeitung dieser Neurosymbole verantwortlich ist. Zwischen diesen Schichten bestehen Schnittstellen. [SWK+15, S. 516][Wen16, S. 28f.]

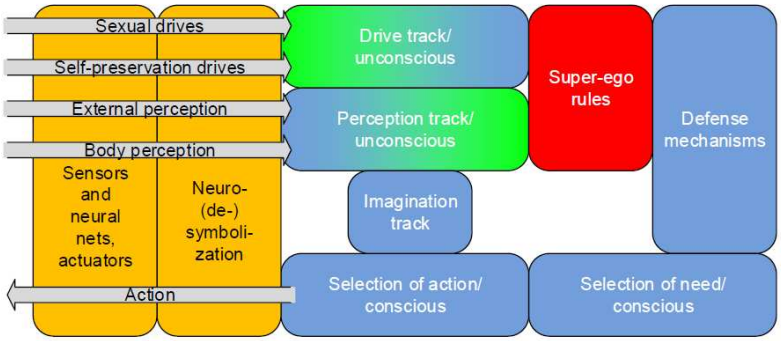


Abbildung 2.1: Schema d. SiMA-Modells [Wen16, S. 4]

Die Grundidee von SiMA ist die Anwendung psychoanalytischer Theorien auf eine künstliche kognitive Architektur. Es handelt sich, anders als bei vielen anderen Architekturen, um einen *bionischen* Ansatz, bei dem auch Gefühlsentwicklung Raum geboten wird. Freuds Einteilung des menschlichen Geistes in *Es*, *Ich* und *Über-Ich* stellt einen Grundpfeiler für die Überlegungen zu SiMA dar. Dabei bedeutet *Es* die äußeren und inneren sensorischen Zustände, also die aktuelle Umgebungssituation. Das *Ich* ist das Bewusstsein, und vermittelt zwischen Es und Über-Ich. Das *Über-Ich* besteht im Grunde aus den moralischen Vorstellungen und Regeln, die sich durch Erlernen einstellen, und so das Ich in seinen Wirkmöglichkeiten einschränken. Über auszuführende Aktionen wird diesem Muster folgend anhand der verschiedenen Triebe sowie der äußeren und inneren sensorischen Empfindungen entschieden (Abb. 2.1). [SWK+15, S. 516][Wen16, S. 27ff.]

2.2 Künstliche neuronale Netze

ANNs sind eine maschinelle, rein mathematische Repräsentation, aufbauend auf den Erkenntnissen über biologische Nervensysteme [Kon05, S. 169]. Als Startpunkt ihrer Entwicklung gilt die Arbeit von Donald Hebb aus dem Jahr 1949 [Dow15, S. 106]. Darin legte er den Zusammenhang dar, dass, wenn ein Neuron häufig ein anderes zum Feuern anregt, sich ihre Bindung aneinander ändert.

Natürliche Neuronen im Säugetiernervensystem bestehen aus einem *Zellkörper* (Nukleus) mit einem *Zellkern* (Soma), sowie einer Schar *Dendriten* und einem *Axon*. An den Enden der Dendriten und des Axons — welche stark verästelt sein können — befinden sich Synapsen, Verbindungselemente, über welche die Kommunikation zwischen den Neuronen abläuft. Durch elektrochemische Vorgänge an den Synapsen verändert sich das Potenzial an der Zellmembran. Die dadurch hervorgerufene elektrische Spannung kommt hauptsächlich durch ein Ungleichgewicht an Natrium-, Kalium-, Calcium- und Chlorionen zustande, und beträgt in Ruhe etwa -75mV von außen nach innen. Steigt das Potenzial — verringert sich also der negative Spannungsbetrag — auf einen

bestimmten Schwellenwert durch Ionenaustausch an den Synapsen, kommt es zu einem *Aktionspotenzial* von −20mV. Das Neuron feuert also. Das Aktionspotenzial hält etwa 1ms an, bevor die Spannung wieder auf Ruheniveau absinkt, und wird über das Axon an andere Neuronen weitergegeben. Der Anteil des Axons an der synaptischen Verbindung heißt auch *Präsynapse*. Ob die Rezeptoren an der *Postsynapse* — der empfangenden Synapsenseite — den Ionenaustausch verstärken oder hemmen, hängt von der Anwesenheit der Neurotransmitter *Glutamat* bzw. *Glyzin* ab. [Rä10, S. 1ff.][GRS00, S. 75ff.][Dow15, S. 102ff.]

Die folgenden Unterkapitel beschäftigen sich mit dem Funktionieren und Aufbau *künstlicher neuronaler Netze*. Eine allgemeine Einführung in den Themenkomplex gibt Kap. 2.2.1. Darauf wird maschinelles Lernen (Kap. 2.2.2) sowie verschiedene Typen von ANN (Kap. 2.2.3) erläutert. Im speziellen wird der Typ der *Backpropagation*-Netze (BP-Netze) in Kap. 2.2.4 behandelt. Abschließend findet sich die Erläuterung einiger ANN-Typen, deren Kenntnis zum Verständnis der in Kap. 2.4 angeführten Arbeiten von Vorteil ist, in Kap. 2.2.5.

2.2.1 Einführung zu Künstlichen Neuronalen Netzen

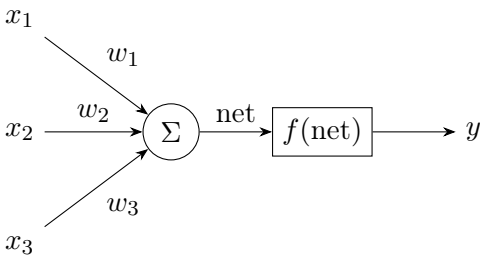


Abbildung 2.2: Aufbau eines künstlichen Neurons

Vom natürlichen Neuron lässt sich das künstliche Neuron ableiten, was je nach Anwendungsfall (z. B. Technik, Neurophysiologie) mit stark unterschiedlichem Detailgehalt geschehen kann. So wollen Biologen bzw. Neurophysiologen die natürlichen Vorgänge im Gehirn möglichst exakt darstellen, was für eine technische Nutzung nicht sinnvoll ist [Dow15, S. 111ff.]. Das üblicherweise in der Technik verwendete künstliche Neuron ist eine mathematische Einheit bestehend aus zwei Elementen: einer linearen Aufsummierung der gewichteten Eingangswerte und einer nichtlinearen Ausgangsfunktion. Anhand Abb. 2.2 lassen sich die Gleichungen 2.1 und 2.2 aufstellen.

$$\text{net} = \sum_i w_i x_i \tag{2.1}$$

$$y = f(\text{net}) = f\left(\sum_i w_i x_i\right) \tag{2.2}$$

Die Ausgangsfunktion kann prinziell eine beliebig gewählte Nichtlinearität sein. Zumeist wird eine der folgenden verwendet [Kon05, S. 169f.][Rä10, S. 5f.][GRS00, S. 85f.]:

- *Sprungfunktion*
- *Signumfunktion*

- *McCulloch-Pitts-Funktion* ist eine Sprungfunktion, bei der

$$f(\text{net}) = \begin{cases} +k & ,\text{net} \geq \theta \\ -k & ,\text{net} < \theta \end{cases} \tag{2.3}$$

  gilt, wobei θ den zu überschreitenden Schwellenwert und k eine beliebige Konstante bedeuten.

- *Fermi-Funktion* entspricht der Sigmoidfunktion

$$f(\text{net}) = \frac{1}{1 + \exp(-g \cdot \text{net})}, \tag{2.4}$$

  wobei g ein Verstärkungsfaktor ist.

- *Tangens hyperbolicus*

2.2.2 Maschinelles Lernen

Damit die Neuronen Nutzen bringen, müssen sie zu ANNs verschalten werden. Es gibt verschiedene Netztopologien, sowie, um diese lernfähig zu machen, zahlreiche Lernparadigmen, derer einige im folgenden diskutiert werden sollen, und die zusammen verschiedene Typen von ANN ergeben. Diese unterschiedlichen Typen eignen sich aufgrund ihrer jeweiligen Eigenschaften zum Einsatz in diversen Bereichen. Beim Lernen werden prinzipiell immer die Eingangsgewichte der Neuronen im Netz einer bestimmten Regel folgend angepasst, sodass letztendlich bestimmte sensorische Eingänge zu einem bestimmten Ergebnis am Netzausgang führen.

Donald Hebb hat eine Lernregel aufgestellt, nach der die Gewichte des ANN modifiziert werden, die für alle nachfolgenden Lernregeln als Vorbild dient:

$$\Delta w_{ij} = \eta s_i x_{ij} \tag{2.5}$$

$$w_{ij}^{k+1} = w_{ij}^{k} + \Delta w_{ij} \tag{2.6}$$

Es werden also die Gewichte mit einem um Δw_{ij} veränderten Wert überschrieben, welcher dem Produkt aus der Lernrate η im Bereich $0 < \eta < 1$, dem Sollwert s_i am Ausgang des Neurons i und x_{ij} dem j-ten Eingangswert ebendieses Neurons entspricht. Aufgrund ihrer Unflexibilität, und da sie kein Abbruchkriterium kennt, wurde der Ansatz der Hebb-Regel mit der Δ- bzw. *Widrow-Hoff*-Lernregel verbessert. Es gilt:

$$\Delta w_{ij} = \eta(s_i - y_i)x_{ij}, \tag{2.7}$$

wobei y_i den wahren Ausgangswert des Neurons i repräsentiert. Die neuen Gewichtswerte werden, wie in Glg. 2.6 angegeben, berechnet. Der Term $(s_i - y_i)$ entspricht dem Fehler zwischen Soll und Ist am Ausgang. Erreicht dieser Term den Wert 0, wird das Lernen für dieses Neuron abgebrochen. Die Δ-Lernregel findet bei BP-Netzen (Kap. 2.2.4) Anwendung. Es gibt noch weitere Lernregeln, deren Diskussion den Rahmen dieser Arbeit sprengen würde. Die Grundidee ist bei allen die gleiche. [Rä10, S. 65f.][GRS00, S. 94f.]

2.2.3 Einteilung der ANN-Typen

Über die Jahre wurden vielerlei Typen von ANN entwickelt, welche sich in Topologie und der Art, wie diese Netze lernen, unterscheiden. An dieser Stelle sollen einige ANN-Typen nach Lernart gruppiert vorgestellt werden. Bei den Lernarten unterscheidet man:

- *Überwachtes Lernen* (engl.: *Supervised Learning*) ist jene Art zu lernen, bei der ein Trainer einen Satz von zu erlernenden Mustern in den Lernalgorithmus des ANN einspeist. Netze dieses Typs haben im allgemeinen eine *Feed-Forward*-Topologie, d. h. eine solche, bei der die Neuronen in Schichten organisiert sind. Die Datenlaufrichtung geht lediglich vorwärts von einer Schicht zur nächsten. Abb. 2.5 auf S. 12 zeigt ein Beispiel für ein Netz dieser Topologie. BP-Netze (vgl. Kap. 2.2.4) sind ein Beispiel für überwachtes Lernen. [Kon05, S. 171f.][Dow15, S. 231ff.]

- *Unüberwachtes Lernen* (engl.: *Unsupervised Learning*) benötigt keinen Trainer. Anders als beim überwachten Lernen, wo Eingangsmuster einem Resultat zugewiesen werden, kategorisiert das ANN beim unüberwachten Lernen die gelernten Eingangsmuster in sich ergebende Klassen. Im allgemeinen braucht es bestimmte Regeln, nach welchen Gesichtspunkten die Klassifizierung erfolgen soll. Ebenfalls anders als beim überwachten Lernen ist die Verschaltung der Neuronen, die als Topologie mit Rückkopplungen zwischen den Schichten oder auch gänzlich beliebig sein kann. [Kon05, S. 172ff.][Dow15, S. 239ff.]

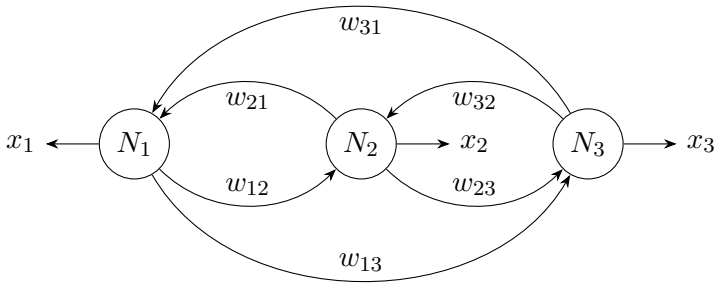


Abbildung 2.3: Beispiel eines *Hopfield*-Netzes mit 3 Neuronen

Zu ANN-Typen, welche unüberwacht lernen, gehören:

  – *Hopfield-Netze*, welche, wie Abb. 2.3 zeigt, Netze mit Rückkopplungen sind. Dabei kommen aber keine Rückkopplungen auf dasselbe Neuron vor, d. h.: $w_{ij} = 0$, wenn $i = j$. Das Hopfield-Netz wurde gemäß der Spinglas-Theorie, also einer sich mit magnetischen Festkörpern beschäftigenden Theorie, entwickelt. Mit Hopfield-Netzen ist es möglich, Daten verteilt zu speichern sowie wieder auszulesen. Es wird dabei zwischen zwei Betriebsmodi unterschieden: dem Encodieren und dem Erinnern. In Analogie zum Vorbild, der Spinglas-Theorie, kann eine Energiefunktion mit

$$E = -\frac{1}{2}\sum_{i}^{n}\sum_{j}^{n} w_{ij}x_i x_j \tag{2.8}$$

  angegeben werden, wobei n die Anzahl der Neuronen im Netz bedeutet. x_i bzw. x_j sind die Ausgangsgrößen, w_{ij} die Gewichte zwischen den Neuronen, wie Abb. 2.3 zu entneh-

men ist. Gemäß der Hopfield-Regel werden die Gewichte folgendermaßen eingestellt:

$$w_{ij} = \mathbf{M}_i \cdot \mathbf{M}_j. \tag{2.9}$$

Hier bedeuten $\mathbf{M}_i$ und $\mathbf{M}_j$ die Mustervektoren für Neuronen i und j, um Gewicht w_{ij} zwischen diesen beiden zu berechnen. In den Mustervektoren sind sämtliche zu speichernden Muster abgebildet. Wird nun ein beliebiges Muster zum Abruf angegeben, gibt das Netz nach einigen Durchläufen das nächstwahrscheinliche gespeicherte Muster zurück. Es wird dabei solange vorgegangen, bis die Energiefunktion ein Minimum erreicht hat. Hopfield-Netze eignen sich also zum Entstören verrauschter Eingangsdaten. Es gibt sie in diskreter und kontinuierlicher Form, wobei letztgenannte im Grunde ein Differentialgleichungssystem darstellt. [Kon05, S. 239ff.][Dow15, S. 239ff.][Rä10, S. 113]

- *Boltzmann-Maschinen* stellen eine Verbesserung der Hopfield-Netze dar, indem lokale Minima der Energiefunktionen durch hinzugefügte Zufälligkeit vermieden werden. Diese Technik wird in Analogie zum Normalglühen bei der Metallbearbeitung als *Simulated Annealing* bezeichnet. [Kon05, S. 252]
- *Autoencoder* ist ein Netzwerktyp, bei welchem ein Eingabevektor $\mathbf{x}$ auf eine Repräsentation $\mathbf{c}(\mathbf{x})$ abgebildet wird, sodass aus dieser $\mathbf{x}$ bei der Decodierung wiederhergestellt werden kann. Hat der Vektor $\mathbf{c}(\mathbf{x})$ eine geringere Länge als $\mathbf{x}$, kommt es somit zu Datenkomprimierung. Durch Lernen soll für gegebene Eingabedaten die Kompression fehlerfrei werden. Autoencoder können auch zum Entrauschen von Signalen genutzt werden. [Ben09, S. 45ff.]

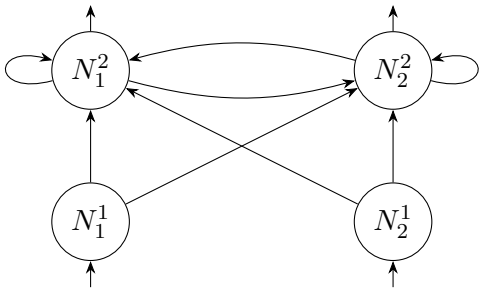


Abbildung 2.4: Beispiel für ein kompetitives neuronales Netz mit Feed-forward-Verbindungen von der Eingangs- zur Ausgangsschicht

- *Kompetitives Lernen* wird allgemein in rückgekoppelten Netzen durchgeführt, bei denen die Verbindungen zu den Nachbarneuronen inhibitiv (abschwächend), die Eigenverbindungen hingegen exzitativ (verstärkend) sind. Ein Beispielnetz ist in Abb. 2.4 zu sehen. [Kon05, S. 267ff.][Dow15, S. 247ff.]

  Es gibt zwei sinnvolle Konfigurationen, um ein kompetitives Netz einzusetzen:

  - Neuronen in der kompetitiven Schicht konkurrieren miteinander. Welches Neuron zuletzt einen Wert ungleich 0 liefert, gewinnt.
  - *Selbstorganisierende Karten* (engl.: *Self-organizing Maps (SOM)*) werden nach ihrem Entwickler auch *Kohonen-Netze* genannt. Bei ihnen handelt es sich um zweidimensionale Neuronennetze, bei denen die Verbindungsgewichte — alle Neuronen sind mit sämtlichen anderen Neuronen verbunden — sich dynamisch so einstellen, dass sie dort

maximal sind, wo die geometrische Entfernung am geringsten ist. Diese Netzart kann z. B. für Gesichtserkennung eingesetzt werden. [Kon05, S. 282ff.][Dow15, S. 251ff.][Rä10, S. 8ff.]

- *Verstärkendes Lernen* wird in Kap. 2.3 eingehend behandelt.

2.2.4 Backpropagation-Netze

Es gibt noch viele weitere Typen von ANN, deren Aufzählung den Rahmen dieser Arbeit allerdings sprengen würde. Da sie für vorliegende Arbeit aber essentiell sind, wie in Kap. 3 dargelegt, sollen im Folgenden noch BP-Netze im Detail erläutert werden. Hierbei handelt es sich um ein Mehrschichten-*Perceptron*, das über den *Backpropagation*-Algorithmus lernt. [Rä10, S. 58]

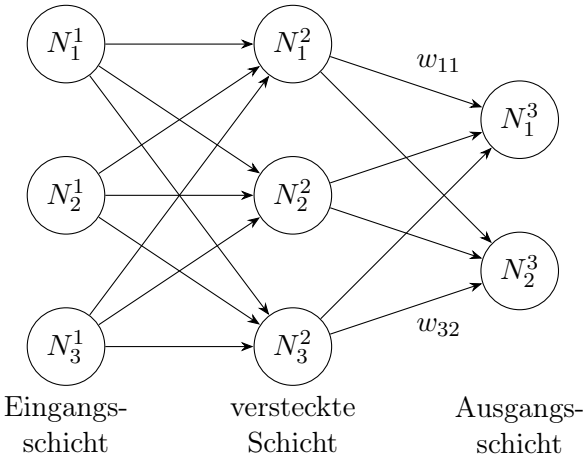


Abbildung 2.5: Beispiel eines *Backpropagation*-Netzes mit einer versteckten Schicht

Ein *Perceptron* ist klassisch ein Feed-Forward-Netzwerk mit lediglich einer Eingangs- und Ausgangsschicht. Es wird für die Neuronen das McCulloch-Pitts-Modell benützt. Dieses Modell definiert Neuronen nach dem früher bereits angegebenen Muster (siehe Abb. 2.2). Das Neuron besteht aus einem gewichteten Summierer und einer nichtlinearen Ausgangsfunktion. [Kon05, S. 198ff.]

Mit Perceptronen lassen sich ohne weiteres logische UND- und ODER-Operationen durchführen. Für die UND-Operation wird lediglich ein Neuron in der Ausgangsschicht gebraucht mit den Eingangsgewichten $w_1 = w_2 = 1$ und einer einfachen Sprung-Ausgangsfunktion, sodass $y = 1$, wenn net ≥ 2. Wenn an beiden Neuroneneingängen 1 anliegt, liefert der Summierer Glg. 2.1 folgend den Wert net $= 2$, der zum Sprung auf $y = 1$ am Ausgang des Neurons führt. Ähnlich aufbauen lässt sich die ODER-Operation mit zwei Neuronen, wobei lediglich die Sprungschwelle auf 1 herabgesetzt werden braucht. Allerdings lässt sich auf diese Art und Weise bereits die XOR-Operation nicht durchführen. Hierfür ist es notwendig, eine weitere Schicht zwischenzuschalten, um den Ausdruck

$$x_1 \oplus x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \tag{2.10}$$

für die Eingangsgrößen x_1 und x_2 zu erhalten. Es werden also in der zwischengeschalteten, sogenannten versteckten Schicht zwei gegengleiche Neuronen für die UND-Operationen mit den Gewichtswerten 1 für die unnegierten Eingänge bzw. -1 für die negierten eingesetzt. Diese liefern die Eingangsgrößen für das ODER-Neuron in der Ausgangsschicht, welches das Endergebnis der Operation liefert. [Rä10, S. 58ff.][Kon05, S. 203ff.]

BP-Netze sind also Mehrschichten-Perceptrone mit einer oder mehreren versteckten Schichten, wie eines beispielhaft in Abb. 2.5 dargestellt ist. Verfügt das Netz über mehr als eine versteckte Schicht, wird es als *tief* bezeichnet. Höhere Netztiefe erlaubt die Darstellung komplexerer Funktionen, welche sich auch hierarchisch einteilen lassen [Ben09, S. 5ff.]. Als Ausgangsfunktion für die Neuronen eines BP-Netzes ist die Fermi-Funktion — eine Sigmoid-Funktion — in Verwendung, da sie stetig und damit differenzierbar ist. Damit können sich die Werte der Ausgangsneuronen allerdings auch nur an 0 und 1 asymptotisch annähern. Optional kann man zum Beschleunigen des Lernens für jedes Neuron einen Bias-Wert einführen, welcher im Schaubild als eigener Eingang, welcher ständig auf 1 liegt und ein Gewicht θ aufweist, modelliert werden kann, wie Abb. 2.6 veranschaulicht. [Rä10, S. 5f.]

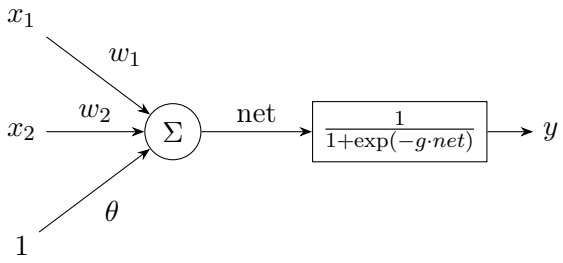


Abbildung 2.6: Neuron mit Bias-Wert zur Verwendung in einem Backpropagation-Netz

Wie bereits angesprochen, findet für BP die Δ-Lernregel (vgl. Glg. 2.7) Anwendung. Der Algorithmus führt das *Gradientenabstiegsverfahren* für die Fehlerterme aus der Δ-Regel durch. Da die Lernregel abbricht, sobald der Fehlerterm $(s_i - y_i) = 0$ erreicht, liegt dort ein lokales Minimum der Fehlerfunktion $E(\mathbf{w})$. Diese lässt sich in Vektorschreibweise für alle Neuronen einer Schicht einführen, wobei $\mathbf{w}$ der Gewichtsvektor ist. Somit kann der Ansatz für den Gradientenabstieg als

$$\Delta \mathbf{w} = -K \frac{dE(\mathbf{w})}{d\mathbf{w}} \tag{2.11}$$

angegeben werden. Daraus folgt dann unter Berücksichtigung der Glg. 2.7 sowie 2.4 nach einigem Umformen:

$$\Delta w_{ij} = \eta \sum_{m=1}^{n} (s_m - f(\text{net}_m)) \frac{\partial f(\text{net}_m)}{\partial \text{net}_m} \frac{\partial \text{net}_m}{\partial w_{ij}} = g\eta(s_i - y_i)(1 - y_i)x_{ij}y_i \tag{2.12}$$

Hierbei bedeutet n die Neuronenanzahl am Ausgang. Die anderen Größen haben die gleichen Bedeutungen wie in ihren Ursprungsgleichungen. Dabei wird von der Ausgangs- zur Eingangsschicht hin vorgegangen, weswegen dieses Verfahren *Backpropagation* heißt. Der BP-Algorithmus leidet an zwei Hauptmakeln. Der eine ist, dass bei zu hoher Lernrate das Netz Gefahr läuft, paralysiert zu werden, wenn die Gewichte hohe Werte erhalten, und aufgrund der Eigenschaften der Fermifunktion ihre Ableitung bei den daraus resultierenden hohen gewichteten Summenwerten gegen 0 geht ($\frac{\partial f(\text{net})}{\partial \text{net}} \to 0$). Dadurch kommt der Lernprozess praktisch zum Stillstand. Dies lässt sich verhindern, indem die Lernrate heruntergesetzt wird, was allerdings den Lernprozess insgesamt verlängert. Das andere Problem ist, dass der Lernprozess bei lokalen Minima steckenbleiben kann. Es handelt sich hierbei um Minima der Fehlerfunktion, welche nicht das absolut mögliche Minimum markieren, sondern lediglich eine Nebentalsohle. Dies kann durch Wahl einer hohen Lernrate vermieden werden. Daran zeigt sich, dass es notwendig ist, einen geeigneten Kompromiss für die Lernrate zu finden. [Rä10, S. 64ff.][Kon05, S. 217ff.][Dow15, S. 231]

2.2.5 Weitere ANN-Typen

Ein spezieller Anwendungsfall für rückgekoppelte Netze ist das *Long Short-Term Memory (LSTM)*, bei welchem die Neuronen in der versteckten Schicht durch *Speicherblöcke* ersetzt sind. Ein solcher Block kann aus mehreren Zellen, welche selbstexhibitiv sind, bestehen. Jede Zelle besitzt einen Eingang und Ausgang. Außerdem teilen sich alle Zellen eines Speicherblocks jeweils ein Tor an Ein- und Ausgang, um Schreib- bzw. Lesezugriff zu steuern. Es kann noch ein Tor hinzugefügt werden, um Vergessen zu ermöglichen. Als Lernalgorithmus wird klassisch *Backpropagation Through Time (BPTT)* verwendet. BPTT passt die Gewichte an, die an den Speicherzellen- und Toreingängen gelten. Damit erlauben LSTMs die Speicherung von zeitlich verteilten Ereignisfolgen, auch wenn diese weit auseinander liegen. [Ger01, S. 6ff.]

Zuletzt soll noch ein besonderer Typ von ANNs vorgestellt werden: *neuronale Faltungsnetze*. Es handelt sich bei ihnen um eine spezielle Art tiefes Feed-Forward-Netz. Sie sind dem visuellen Cortex nachempfunden. Entsprechend werden sie häufig für Bildverarbeitungsaufgaben verwendet. Ein besonders prominentes Beispiel befasst sich mit Handschrifterkennung, das Faltungsnetz von *LeCun*. Faltungsnetze bestehen aus Schichten, worin die Neuronen als zweidimensionales Netz organisiert sind. Dabei erfüllen die Schichtübergänge unterschiedliche Aufgaben (Faltung, Bündelung). [SB18, S. 227f.][Ben09, S. 43ff.]

2.3 Verstärkendes Lernen

Beim *verstärkenden Lernen* (engl.: *Reinforcement Learning (RL)*) geht es im Prinzip darum, Aktionen auftretenden Situationen so zuzuweisen, dass der Agent die höchstmögliche Belohnung erhalten kann. Dazu bedarf ein RL-System jedenfalls einer *Policy*, einer passenden Art *Belohnungssignal* — welches umgekehrt auch Bestrafung umfasst — und einer *Wertefunktion*. Ein Agent, der RL einsetzt, versucht neue Ansätze und lernt aus Erfolgen wie aus Fehlschlägen, mit anderen Worten: aus Erfahrung. Er kann bei seiner Arbeit bereits gelerntes Verhalten anwenden (*Exploitation*) oder Neues ausprobieren (*Exploration*). Ohne Exploration wird der Agent aber auf seinem bisherigen Verhalten (Policy) beharren und sie nicht inkrementell verbessern. [SB18, S. 1ff.][Kon05, S. 295ff.][Dow15, S. 278ff.]

Es wird die *Belohnung* (engl.: *Reward*) für jeden Zeitschritt als reelle Zahl R_t definiert. Das Ziel eines Agenten ist es, die insgesamt erhaltene Belohnung zu maximieren, was sicherstellen soll, dass er Grund hat, lediglich Aktionen mit positivem Effekt zu setzen, solche mit negativem hingegen zu meiden. Ist das Ergebnis einer Aktion negativ, gilt $R_t < 0$. Falls eine Aktion weder als positiv noch als negativ gesehen werden kann, wird die Belohnung zu diesem Zeitpunkt den Wert $R_t = 0$ annehmen. Die R_t summieren sich zur total erlangten Belohnung. Es kann die erwartete Belohnung also als

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.13}$$

angeschrieben werden, wenn sich der Lernvorgang in Episoden mit jeweils einem Endzustand zum Zeitpunkt T einteilen lässt. Diese Art Aufgabe wird dempentsprechend *episodisch* genannt. Im Gegensatz hierzu gibt es auch *sich fortsetzende Aufgaben*, bei denen ein Agent — beispielsweise ein Roboter — mit seiner Umgebung laufend interagiert. Hier gibt es demnach auch keinen Endzustand, womit sich

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.14}$$

definieren lässt, wobei $\gamma \in [0, 1]$ ein *Diskontfaktor* ist. Dieser Faktor γ sorgt für die Beschränktheit von G_t, wenn $\gamma < 1$, und weiter in der Zukunft liegende Belohnungen haben vom Zeitpunkt t aus abnehmenden Einfluss auf die erwartete Belohnung. [SB18, S. 54f.][Kul12, S. 62f.]

Aktions-Wertefunktionen können unterschiedlich definiert werden, wobei der Wert einer Aktion der durchschnittlichen Belohnung bei Auftritt dieser Aktion entspricht. Es kann also der Wert einer Aktion einfach als

$$Q_t(a) \doteq \frac{\text{Belohnungssumme mit Aktion } a \text{ vor } t}{\text{Auswahlhäufigkeit von } a \text{ vor } t} \tag{2.15}$$

angegeben werden. Im Laufe der Zeit konvergiert $Q_t(a)$ zum Wert der für die Aktion a erwarteten Belohnung

$$q_*(a) \doteq E\left\{R_t | A_t = a\right\}, \tag{2.16}$$

wobei R_t die Belohnung und A_t die gewählte Aktion zum Zeitpunkt t angeben. Wird rein exploitativ vorgegangen, wird die nächste Aktion *gierig* (engl.: *greedy*) nach der Gesetzmäßigkeit

$$A_t \doteq \max_a Q_t(a) \tag{2.17}$$

ausgewählt. [SB18, S. 25ff.][Kul12, S. 69f.]

Gierig bedeutet in diesem Zusammenhang, dass nur Aktionen gewählt werden, die der aktuellen *Policy* entsprechen. Es kann also bei gieriger Aktionswahl die für RL so wichtige Exploration nicht auftreten. Wird ein bestimmter Anteil der Aktionen nicht gierig gewählt, nennt man die Aktionsauswahl *ε-greedy* bzw. *ε-gierig*. ε entspricht hier der Auftrittswahrscheinlichkeit eines Explorationsschrittes. Soll also z. B. jede zehnte Aktion explorativ sein, ergibt sich $\varepsilon = 0{,}1$. Versuche zeigen, dass ε-gierige Agenten wesentlich bessere Ergebnisse erzielen, da vollkommen gierige Methoden weniger wahrscheinlich neuen, noch nicht aufgetretenen Situationen ausgesetzt werden, um ihre Policy an diese anzupassen und so zu verbessern. [SB18, S. 27ff.]

Ist das zu lösende Problem nicht-stationär — sind die Wahrscheinlichkeiten der Belohnung von Aktionen in bestimmten Situationen über die Zeit also veränderlich —, macht es Sinn, kürzer zurückliegende Belohnungen zu bevorzugen. Im Umkehrschluss bedeutet dies, dass, je weiter eine Belohnung zurückliegt, diese umso geringeren Einfluss auf die Wertefunktion ausübt. Dies lässt sich einfach über die Einführung des Schrittweitenparameters $\alpha \in (0, 1]$ lösen, sodass

$$Q_{n+1} \doteq Q_n + \alpha\left(R_n - Q_n\right) \tag{2.18}$$

gilt. [SB18, S. 32f.]

Es gibt verschiedene Typen von Algorithmen für RL. Üblicherweise bauen sie auf *Markov-Entscheidungsprozessen* (engl.: *Markov Decision Processes (MDP)*) auf, welche eine klassische Formulierung sequenzieller Entscheidungsfindung darstellen. Dabei werden nicht nur unmittelbare Belohnungen und Zustände, sondern auch nachfolgende, berücksichtigt. Anders als zuvor erwähnt, soll die Wertefunktion nicht zu $q_*(a)$, sondern zu $q_*(s, a)$ oder $v_*(s)$ konvergieren. $v_*(s)$ wird dabei für jeden Zustand s unter der Annahme, dass die Aktionen a optimal gewählt werden, geschätzt. Bei MDPs geht man davon aus, dass Agent und Umwelt — alles, was außerhalb des Agenten vorgeht — unablässig miteinander interagieren. Dabei hat die Umwelt zu allen Zeiten einen bestimmten Zustand und bewertet auch das Verhalten des Agenten über Belohnungen. Demgemäß entscheidet sich der Agent sodann für die beste Aktion, welche wiederum Einfluss auf die Umwelt nimmt, und ihren Zustand abändert, wofür der Agent eventuell belohnt oder bestraft wird. Damit schließt

sich der Kreislauf. Enthält ein Zustand sämtliche Informationen über die Auswirkungen einer Interaktion zwischen Agent und Umwelt, so besitzt dieser Zustand die sog. *Markov-Eigenschaft*. Im speziellen gibt es *finite* MDPs, die sich dadurch auszeichnen, dass es eine endliche Anzahl an Zuständen, Aktionen sowie Belohnungen gibt. [SB18, S. 47ff.][Kul12, S. 68f.]

Ein weiteres Konzept ist *dynamisches Programmieren* (*DP*; engl.: *Dynamic Programming*). Dabei handelt es sich um eine Gruppe von Algorithmen, welche, ein perfektes Modell der Umwelt vorausgesetzt, optimale Policies als MDPs berechnen. Dabei wird zwischen zwei Hauptschritten unterschieden: *Policy-Evaluierung*, welche die iterative Berechnung der Wertefunktionen für eine Policy meint, und *Policy-Verbesserung*, wobei mithilfe gegebener Wertefunktionen eine angepasste, verbesserte Policy errechnet wird. Üblicherweise lassen sich alle RL-Methoden als *Generalized Policy Iteration* ansehen. Dabei arbeiten Evaluierung und Verbesserung der Policy anhand geschätzter Wertefunktionen bzw. Policies abwechselnd zusammen, indem der eine Schritt die Schätzung für den jeweils anderen durchführt. [SB18, S. 73ff.][Kul12, S. 71ff.]

Außerdem sollen hier noch kurz *Monte-Carlo*-Methoden behandelt werden. Sie unterscheiden sich von DP dadurch, dass sie kein bekanntes Modell voraussetzen, sondern lediglich gesammelte *Erfahrung* benötigen. Damit ist auch kein Vorwissen über die Umwelt zum Lernen notwendig. Hier werden die Gesamt-Belohnungen über ganze Episoden, auf welche die Erfahrung aufgeteilt wird, gemittelt, und darüber Wertefunktionen für alle Zustands-Aktions-Paare durch Mittelung geschätzt. Dadurch funktioniert aber Lernen mit Monte-Carlo nicht schrittweise, sondern nur für ganze Episoden. [SB18, S. 91ff.]

Monte-Carlo-Methoden und DP sind Ansätze, die in den folgenden Unterkapiteln Anwendung finden. Darin wird der bekannte *Temporal-Difference (TD)*-Lernalgorithmus erläutert (Kap. 2.3.1). Danach werden die darauf aufbauenden Algorithmen des *Sarsa-* bzw. *Q-Lernens* für Steueraufgaben in Kap. 2.3.2 behandelt.

2.3.1 TD(0)- und TD(λ)-Lernen

TD-Lernen ist eine RL-Methode, welche in der Literatur hohe Aufmerksamkeit besitzt. Sie nutzt Elemente von Monte-Carlo- sowie DP-Methoden. So gleicht TD-Lernen DP insofern, als nicht auf Vollendung von Episoden gewartet werden muss, wie dies bei Monte-Carlo der Fall ist. Andererseits lernt TD wie Monte-Carlo rein von Erfahrungswerten. TD dient zunächst der Schätzung der Wertefunktion v_π bei gegebener Policy π (*Vorhersageproblem*). Methoden, um die Policy selbst zu optimieren (*Steuerproblem*), werden im nächsten Unterkapitel 2.3.2 vorgestellt.

Im einfachsten Fall kann der von v_π zurückgegebene Wert $V(S_t)$ sofort bei Erhalt der Belohnung R_{t+1} mit

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.19}$$

iterativ angepasst werden, sobald sich der Zustand S_{t+1} im nächsten Zeitabschnitt einstellt. Dies erfolgt in Analogie zu Monte-Carlo. Allerdings wird nicht die gemittelte Gesamtbelohnung sondern die nächste Einzelbelohnung zwischen zwei Zeitabschnitten genutzt. α und γ sind die bereits bekannten Schrittweiten- bzw. Diskontkonstanten. Bei $V(S_{t+1})$ handelt es sich um eine Schätzung der Wertfunktion zum Zeitpunkt $t+1$.

Diese simpelste Anpassungsregel macht den *TD(0)*-Algorithmus aus. Er kann auch als *Einschritt-TD* bezeichnet werden. Dieser steht *n-Schritt-TD* sowie *TD(λ)* gegenüber, wobei TD(0) ein Spezialfall dieser beiden ist. [SB18, S. 119ff.][Kul12, S. 71ff.]

Bei *n-Schritt-TD* umfasst die Anpassung nicht nur den unmittelbar nächsten, sondern die nächsten n Zeitabschnitte. Entsprechend kann die Schätzung der Wertefunktion erst nach n Schritten erfolgen. Ein Nachteil davon ist, dass mehr Rechenschritte pro Zeitabschnitt gebraucht werden. Es handelt sich um eine Mischform aus TD(0) und Monte-Carlo. [SB18, S. 141ff.]

TD(λ) stellt eine weitere Zwischenform aus TD(0) und Monte-Carlo dar, wobei sog. *Eligibility Traces* (dt.: *Eignungsspuren*) eingesetzt werden. Solche Mischformen liefern oftmals bessere Ergebnisse als die Reinformen. Im Fall von *TD(λ)* bedeutet $\lambda = 0$ reines TD — daher die Bezeichnung TD(0) — bzw. $\lambda = 1$ Monte-Carlo. Bei einer Eligibility Trace handelt es sich im Prinzip um einen Kurzgedächtnis-Vektor $\mathbf{z}_t$. Er existiert neben dem die Wertefunktion bestimmenden Gewichtsvektor $\mathbf{w}_t$. Ist nun eine Komponente von $\mathbf{w}_t$ an der Schätzung eines Wertes beteiligt, wird die entsprechende Komponente in $\mathbf{z}_t$ laut

$$\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t) \tag{2.20}$$

erhöht, welche sodann nach und nach verfällt. Hierbei bedeutet $\nabla\hat{v}(S_t, \mathbf{w}_t)$ den geschätzten Wert von $v_\pi(s)$. Es kann dann der Gewichtsvektor mit

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\mathbf{z}_t \left[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \tag{2.21}$$

angepasst werden, wobei sich nur jene Komponenten ändern, deren Äquivalente in $\mathbf{z}_t$ ungleich 0 sind. $\lambda \in [0, 1]$ ist der *Spurverfallsparameter* und bestimmt, wie schnell $\mathbf{z}_t$ verfällt. Es zeigt sich an der Darstellung mit dem Gewichtsvektor $\mathbf{w}_t$ bereits, dass sich ANNs für die Approximation der Wertefunktionen eignen. [SB18, S. 197, 223ff., 289ff.]

2.3.2 Sarsa- und Q-Lernen

Da es sich bei *Sarsa-* und *Q-Lernen* um TD-Steueralgorithmen handelt, soll $q_\pi(s, a)$ für Policy π für alle Zustände s und Aktionen a geschätzt werden. Es kann also

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{2.22}$$

für die Gleichung zum iterativen Ermitteln des Schätzwertes von Q angeschrieben werden. Dies ist der *Sarsa*-Algorithmus, der wegen der involvierten Parameter in der eckigen Klammer seinen Namen hat. Um die bestmögliche Aktion zu wählen, braucht dann nur $Q(s, a)$ für alle möglichen Aktionen bei Zustand s berechnet zu werden, woraus dann jene Aktion gewählt wird, welche den höchsten Wert zum Resultat hat. [SB18, S. 129ff.]

Bei Sarsa handelt es sich um einen *On-policy*-Algorithmus. D. h., dass die Policy, nach der vorgegangen wird, auch während des Laufs angepasst wird. Daneben gibt es auch den *Off-policy*-Ansatz, bei dem es zwei Policies nebeneinander gibt: eine Zielpolicy, welche gelernt werden soll, und eine, die das aktuelle Verhalten vorgibt. *Q-Lernen* funktioniert off-policy, woraus sich

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.23}$$

angeben lässt. Damit schätzt der Algorithmus die optimale Aktions-Wertefunktion q_* unabhängig vom aktuell verwendeten Verhalten. [SB18, S. 131f.][Kul12, S. 74]

Bei den oben behandelten Algorithmen kann es zum sogenannten *Maximization Bias* kommen. Da die Q-Werte für die aktuelle Situation bzw. den aktuellen Zustand und die verschiedenen möglichen

Aktionen geschätzt werden, kann es vorkommen, dass eine nicht optimale Aktion den maximalen Wert liefert, und dementsprechend fälschlich ausgewählt wird. Dem kann mittels *Double Learning* begegnet werden. Hierbei wird der Q-Wert nicht ein- sondern zweimal geschätzt, wobei der erste zur Schätzung des zweiten herangezogen wird. Im Falle des Q-Lernens wird Glg. 2.23 zum iterativen Ermitteln von Q_1 entsprechend zu

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2 \left(S_{t+1}, \operatorname{argmax}_a Q_1(S_{t+1}, a)\right) - Q_1(S_t, A_t)\right] \tag{2.24}$$

modifiziert. Q_2 lässt sich symmetrisch auf die gleiche Weise ermitteln. Der vollständige Algorithmus für doppeltes Q-Lernen wählt dann die Aktion zunächst ε-gierig aus den Ergebnissen von $Q_1 + Q_2$. Danach wird mit gleicher Wahrscheinlichkeit entweder Q_1 oder Q_2 lt. Glg. 2.24 angepasst. [SB18, S. 134ff.]

2.4 Existente Arbeiten

Im folgenden sollen einige bereits existierende Arbeiten vorgestellt werden, welche im gegebenen Feld der Nutzung von ANNs zur Erstellung lernfähiger Agenten Vorarbeit geleistet haben. Sie dienen als Referenzen dafür, wie kognitive Architekturen mit ANNs funktionieren können. Viele der angeführten Beispiele beschäftigen sich mit reinen Spielagenten (Kap. 2.4.1, 2.4.2 u. 2.4.3), wohingegen Kap. 2.4.4 Arbeiten auf dem Gebiet der Robotik aufgreift.

2.4.1 TD-Gammon

Vielfach in der Literatur diskutiert findet sich die Arbeit *TD-Gammon* von *G. Tesauro*, in welcher ein ANN Backgammon erlernt, indem es gegen sich selbst spielt. Zur Anwendung kommt dabei ein *Mehrschichten-Perceptron* (vgl. Kap. 2.2), welches *TD(λ)-Lernen* (vgl. Kap. 2.3.1) betreibt. Der Netzaufbau sieht so aus, dass in der Eingangsschicht die Positionen am Spielfeld sequenziell angegeben werden, woraus sich am Ausgang 4-dimensionale Vektoren ergeben, deren Komponenten für die geschätzten Wahrscheinlichkeiten eines normalen bzw. Gammon-Gewinns von Weiß und Schwarz stehen. Die Eingangsvektoren bestehen aus den Anzahlen an Spielsteinen auf den Brettpositionen für den jeweils aktuellen Spielzug. In späteren Versionen wurden noch zusätzliche Informationen mitkodiert, welche die Spielsituation weiter bewerten. Nach jedem Spielzug werden die Gewichte $\mathbf{w}$ des ANN dem TD(λ)-Algorithmus folgend mit

$$\mathbf{w}_{t+1} - \mathbf{w}_t = \alpha(\mathbf{Y}_{t+1} - \mathbf{Y}_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}} Y_k \tag{2.25}$$

angepasst, wobei $\mathbf{Y}_t$ den Ausgangsvektor zum Spielzug t bezeichnet. Bei Spielende wird dann mit der Endbelohnung — der das tatsächliche Spielergebnis im Format von $\mathbf{Y}_t$ angibt — dieser Vorgang noch einmal durchgeführt. Für die Wahl des nächsten Spielzugs werden die Ergebnisse für alle möglichen Züge geschätzt und das Maximum gewählt. Beim Training hat das Netz dies für beide Seiten durchgeführt. Für das ANN wurde in verschiedenen Versuchen eine versteckte Schicht mit 40 bzw. 80 Neuronen eingesetzt. Auch wurden Versuche zunächst mit einer Suchtiefe von einem Spielzug, später mehreren gemacht. Die Ergebnisse zeigen, dass das Netz in der Lage ist, nach einigen tausend Spielen die Spielgrundlagen und wichtigsten Strategien zu erlernen. Nach tausenden weiteren Spielen hat sich das Netz auch fähig gezeigt, auch fortgeschrittene Methoden zu entwickeln. Auch gegen menschliche Gegner — einige der besten der Welt — ist das System in seinen verschiedenen Versionen getestet worden, wobei sehr gute Ergebnisse erzielt wurden. [Tes95, S. 58ff.][Tes02, S. 181ff.]

2.4.2 AlphaGo

Ein weiterer prominenter Versuch, neuronale Netze für ein Brettspiel einzusetzen, betrifft *Go* mit *AlphaGo*. Hier wurde zunächst ein Policy-Netz mit Zügen menschlicher Experten aus historischen Daten trainiert. Das neuronale Netz besitzt 13 Faltungsschichten, wobei eine Schicht aus 19×19 Einheiten besteht, und wurde danach auf ein weiteres Policy-Netz übertragen, um die Policy über RL weiter zu verfeinern. Dabei kamen Monte-Carlo-Methoden zum Einsatz. Zuletzt wurde das Policy-Netz benützt, um ein Werte-Netz ebenfalls mithilfe von RL zu trainieren, zum Schätzen der Wertefunktion. Das Programm hat gegen andere Go-Programme sowie menschliche Gegner gespielt. Dabei gewann es gegen andere Programme nahezu immer und auch sämtliche der fünf Spiele gegen den europäischen Go-Meister. [SHM+16, S. 484ff.]

2.4.3 Diverse Versuche an Videospielen

In der Literatur finden sich ebenfalls verschiedene Bemühungen KI-Agenten Videospiele erlernen zu lassen. Eine Gruppe um Mnih beschäftigt sich mit der Verwendung von ANNs zum Erlernen von *Atari2600*-Spielen. Es wurden hierfür tiefe Q-Netze eingesetzt, die Q-Lernen anwenden. Diese Art Netze wird entsprechend als *DQN* (*Deep Q-Networks*) bezeichnet. Als Eingänge für die Q-Funktion (dargestellt durch das ANN) dienen das Videosignal von 210×160 Pixel (RGB) und die Belohnung, welche aufgrund der hohen Unterschiede in der Punktevergabe der verschiedenen Spiele aber auf $+1$ für Positives und -1 für Negatives normiert ist. So kann analog zu Glg. 2.23 die Gewichtsmatrix des ANN an den neuen Q-Wert angepasst werden. Die Autoren benützen eine Technik namens *Experience Replay*, um die vom Agenten gemachten Erfahrungen für jeden Zeitpunkt zu speichern. Diese können dann zur Laufzeit neben der aktuellen Erfahrung zufällig zur Anpassung der Q-Funktion herangezogen werden. Der Agent folgt bei der Aktionswahl einer ε-gierigen Policy. Dabei wurde beim Lernen ε von 1 bis 0,1 verringert. Dieser und andere Algorithmen wurden mit mehreren Atari-Spielen (z. B.: Space Invaders, Seaquest) getestet. Es zeigt sich nach etlichen Trainingsepochen, in welche die Spieldurchläufe eingeteilt werden, ein Anstieg der durchschnittlich erreichten Punktezahl bzw. die Anzahl der erhaltenen Belohnungen pro Episode. Vergleiche mit anderen Methoden zeigen die hier vorgestellte in den meisten der verschiedenen getesteten Spiele als klar dominant. Auch menschlichen Testspielern ist sie in vielen Spielen überlegen. [MKS+15, S. 529ff.][MKS+13, S. 1ff.]

Ausgehend von den eben vorgestellten Arbeiten [MKS+15, S. 529ff.] [MKS+13, S. 1ff.] wurde eine Methode vorgeschlagen, um den Replay-Speicher ökonomischer zu befüllen und verwalten. Dabei wird die euklidische Distanz zwischen Erfahrungsdatensätzen genutzt, um sie zu filtern. Auf diese Art werden sehr ähnliche Erfahrungen nicht mehrfach gespeichert, wodurch die Daten im Speicher vielfältigere Erfahrungen repräsentieren, und der Speicher nicht mehrmals mit sich ähnelnden Erfahrungen aufgefüllt wird. Dies ist besonders dann ein Problem, wenn der zur Verfügung stehende Speicher relativ gering ist. Der Speicher muss in diesem Fall als FIFO funktionieren, worauf die Filterung angewendet werden kann. Zum Testen ihrer Methode dient den Autoren das Szenario eines Roboters, der auf einer hindernisgespickten Karte auf eine Person zufahren soll, und belohnt wird, wenn er sich dieser auf weniger als einen Meter nähert. Ebenso wird bestraft, wenn der Roboter ein Hindernis — genauer: eine Wand — trifft. In diesem Szenario konnte der vorgestellte Algorithmus eine höhere durchschnittliche Belohnung pro Episode lukrieren, besonders bei geringer Speichergröße. Diese Arbeit zeigt deutlich die Wichtigkeit eines sinnvollen Managements der gespeicherten Erfahrungen, welche für die Erlernung einer Policy herangezogen werden. [NAO18, S. 243ff.]

Ebenfalls mit RL beschäftigt sich das *RL4J*-Projekt [1], welches ein Subprojekt von *Deeplearning4j* (DL4J) [7] ist. Auch hier geht es um die Implementierung kognitiver Architekturen. Dafür gibt es auf der Webseite auch etliche Implementierungsbeispiele, so auch Spiele wie z. B. *Doom*. Auch hier werden DQN und andere Techniken eingesetzt, welche hier nicht weiter Erwähung finden sollen. Auch in MOBA[1]-Spielen wie *Dota 2* wurden bereits sehr erfolgreich Agenten, welche mit ANNs arbeiten, eingesetzt. Dies ermöglicht die Architektur *OpenAI Five* [2], die nach Angaben der Entwickler bereits Amateurspieler in Dota 2 besiegt hat. Die Architektur nutzt für jede Spielfigur ein eigenes *LSTM* (Long Short-Term Memory) [Ger01, S. 11ff.] und RL, um Strategien zu erlernen und zu entwickeln.

2.4.4 Robotik

Es finden sich in der Literatur aber auch Arbeiten, welche sich mit Aufgaben aus der Robotik befassen. So beschäftigt sich eine Arbeit mit der Implementierung eines BP-Netzes, welches mithilfe von RL und unter Verwendung eines Kurzzeitgedächtnisses einen Roboter in die Lage versetzen soll, zu lernen, einen bestimmten Punkt in einem Labyrinth anzufahren. Das Kurzzeitgedächtnis wird mit Vektoren, bestehend aus der Verschmelzung der Eingangs- und Ausgangsvektoren des Netzes, befüllt und für den Lernvorgang online — d. h., während des Durchlaufes — nach dem Prinzip der BP herangezogen. Der Roboter wird bei Zusammenstoß mit einem Hindernis bestraft. Hingegen erhält er eine Belohnung, wenn er in Sichtweite eines ausgesetzten Zielobjektes gelangt. Dabei ist die Fehlerfunktion für die BP je nach Vorzeichen des Belohnungssignals — positiv für Belohnung, negativ für Bestrafung — unterschiedlich. Die Autoren haben ihren Algorithmus in einer Testumgebung im Mobile Robot Simulator getestet, wobei das ANN des Roboters über 15 Eingangs- und 2 Ausgangsneuronen verfügt. Die Eingangsdaten umfassen 12 in alle Richtungen verteilte Sensoren zur Distanzmessung zu Hindernissen sowie die eigenen Richtungs- und Positionsdaten. Die Wertdifferenz an den beiden Ausgangsneuronen bestimmt die Richtungsvorgabe für den nächsten Schritt. Durch das Training gelangt der Roboter tatsächlich in die Lage, seinen Weg durch das Labyrinth zum Zielobjekt zu finden. [GL12, S. 210ff.]

Eine ganz ähnliche Aufgabenstellung findet sich beim nächsten Beispiel, das hier angeführt wird. Im Gegensatz zur zuvor beschriebenen Arbeit wurde allerdings die Zielerreichung nicht bloß simuliert, sondern mit einem real existierenden Roboter getestet. Es handelt sich dabei um eine verbesserte Variante des Lego Mindstorm mit Kamera- und Rechneraufbau. Auch die Herangehensweise unterscheidet sich dahingehend, dass der Lernprozess nicht nur online sondern auch offline — also im Nachhinein — die gespeicherten Schritte bewertet und in den verwendeten ANNs verarbeitet. Aufgrund der höheren Komplexität (Kamerabilddaten) verwendet die Architektur ein tiefes ANN, da die Rohdaten der Kamerabilder erst zu einer internen Repräsentation bzw. Erkennung der sichtbaren Gegenstände führen müssen. Dies wird durch ein tiefes Faltungsnetz erreicht, welches mithilfe eines Autoencoders arbeitet. Dessen Ergebnisvektor wird dann an ein Feed-Forward-Netz übergeben, das kombiniert für die Rolle des Agenten als Aktor als auch als Beobachter Verwendung findet. Dabei folgt direkt auf die Eingangsschicht eine Aktor-Schicht, welche aus den Eingangswerten drei Schätzungen der Q-Funktion durchführt — für jede mögliche Aktion eine —, worauf an der Ausgangsschicht die Ausgabe des Ergebnisses der Wertfunktion V und das Ergebnis der Maximumssuche für die Aktionswahl stehen. Die Ausgangsschicht repräsentiert dabei den Beobachteraspekt. Damit ist also der Roboter ein selbstbeobachtender, selbstkritisierender Agent. Zur Evaluierung des Lernerfolges haben die Autoren die Anzahl der Schritte gezählt,

[1] Multiple Online Battle Arena

welche der Roboter in den verschiedenen Episoden bzw. Durchgängen gebraucht hat, um sein Ziel zu erreichen. Es zeigt sich dabei, dass die Anzahl der benötigten Schritte signifikant sinkt, wobei sich drei Abschnitte des Lernprozesses unterscheiden lassen: der initiale, in dem der Agent keine Selbstreflexion durchführt, der im Gegensatz zum vorigen selbstreflektive und jener Abschnitt, in welchem der Agent bloß seiner erlernten Policy folgt. Die durchschnittliche Schrittzahl sinkt vom einen Abschnitt zum nächsten. [Alt16, S. 4565ff.]

3 Kognitive Architektur

Für diese Arbeit wurde eine eigene kognitive Architektur mit dem Namen *NNMind* entwickelt, deren Kernstück ein BP-Netz ist, welches sich RL zunutze macht, um bei Einlangen einer Belohnung bzw. Bestrafung nach entsprechend dynamisch erstellten Trainingsdatensätzen zu lernen. Um die Wirkungsweise zu zeigen (Kap. 3.2), soll zunächst gezeigt werden, wie die Umgebung, in welcher der Agent eingesetzt werden soll, definiert ist. Es handelt sich um die Miklas-Spielumgebung, welche zusammen mit dem verwendeten Spiel *Foodchase* in Kap. 3.1 vorgestellt wird.

3.1 Miklas-Spielumgebung

Die am Institut für Computertechnik der Technischen Universität Wien entwickelte Spieleumgebung *Miklas* erlaubt die Definition von Spielen für verschiedene Spielfiguren (Agenten), wobei deren Steuerung sowohl von KIs als auch von menschlichen Spielern übernommen werden kann. In den von Miklas unterstützten Spielen können diese Agenten unterschiedliche Aktionen durchführen, um dafür gegebenenfalls Punkte zu erhalten oder zu verlieren. Sie bewegen sich dabei auf einem Spielraster, auf dem sich Objekte (Hindernisse, Nahrung) und die Figuren befinden. Die Spieldefinition erfolgt mithilfe einer Konfigurationsdatei. Nähere Informationen hierzu sind Kap. 4.1.1 zu entnehmen.

Das für diese Arbeit benützte Spiel — der Einfachheit halber *Foodchase* genannt — lässt die Agenten gegeneinander in einer mit Nahrungsmitteln gefüllten Welt antreten. Abb. 3.1 zeigt *Miklas* im ursprünglichen, unbearbeiteten Zustand während der Ausführung des besagten Spieles *Foodchase*, welches in Kap. 5.1 eingehend beschrieben wird. Bei den Agenten handelt es sich um die abgebildeten Alien-Wesen. Führen sie bestimmte Aktionen aus, wird knapp über ihnen ein entsprechendes Hinweissymbol angezeigt. Das Spielziel ist, eine möglichst hohe Punktzahl zu erreichen und die Gegner nach Möglichkeit zu überleben. Die Agenten können sich durch die Welt bewegen, Nahrung zu sich nehmen und ihre Gegner angreifen. Diese Aktionen, zum richtigen Zeitpunkt und Ort angewendet, werden durch Vergabe von Pluspunkten belohnt. Gegnerische Angriffe und fehlgegangene Aktionen können durch Minuspunkte bestraft werden.

Die Agenten besitzen Sensorik, um ein Bild ihrer Umgebung und ihres eigenen körperlichen Zustandes zu erhalten, sowie eine Reihe an möglichen durchführbaren Aktionen. Es soll zunächst abgeklärt werden, welche diese sind, sowie ihre Organisation. Die Sensorneuronen stellen die Daten des visuellen Systems und Hunger — definiert als

$$\text{Hunger} = 1 - \frac{\text{Gesundheit}}{100} \tag{3.1}$$

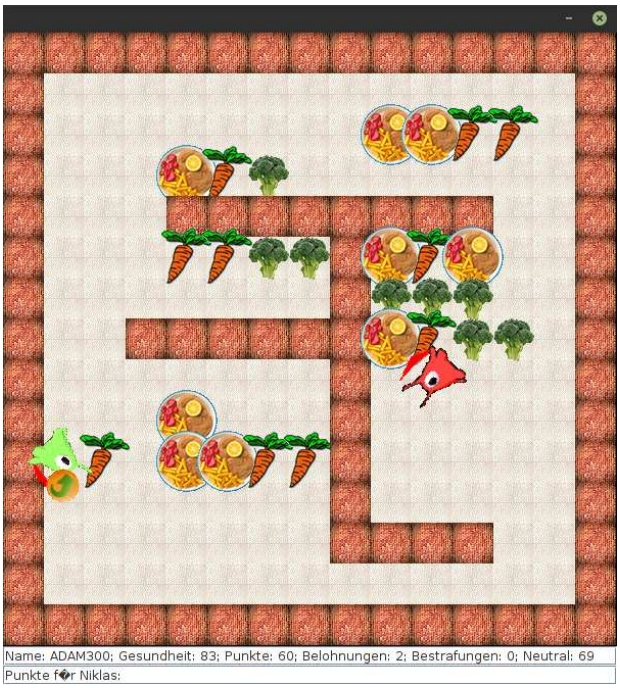
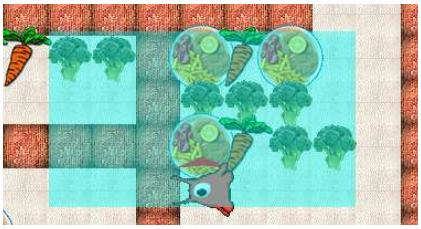


Abbildung 3.1: Spielfenster von *Miklas* im ursprünglichen Zustand

im Wertebereich 0–1 — dar, wobei die Gesundheit im Spiel im Bereich von 0 (tot) bis 100 (volle Gesundheit) definiert ist. Das Sichtfeld des Agenten weist — wie in Abb. 3.2a durch den türkisen Bereich veranschaulicht — einen Radius von drei Feldern und einen Öffnungswinkel von 180° auf. Es ergeben sich somit insgesamt 27 Kacheln im Sichtbereich des Agenten, welche lt. dem Schema in Abb. 3.2b zur Adressierung indexiert sind. Die Position des Agenten selbst ist hier durch ein Sternchen angegeben. Die durchlaufende Nummerierung beginnt auf dem Feld direkt vor dem Agenten und setzt sich weiter vor bis zum Rand des sichtbaren Bereichs fort. Danach geht es abwechselnd auf der linken und rechten Seite weiter.



(a) Beispiel auf dem Spielfeld

22	14	6	2	10	18	26
21	13	5	1	9	17	25
20	12	4	0	8	16	24
19	11	3	⋆	7	15	23

(b) Feldindizes d. Sichtbereichs

Abbildung 3.2: Sichtfeld des Agenten

Auf den Kacheln des Spielfeldes können sich verschiedene Objekte befinden, welche in die folgenden Klassen eingeteilt sind:

- Hindernisse (hier nur Mauern)
- Essbares (Schnitzel, Karotten, Brokkoli)
- Gegner (angreifbar)

Damit werden also zur Inhaltsdarstellung jeder Kachel 3 Bit benötigt. Ist eines der Objekte auf einer Kachel vorhanden, wird das entsprechende Bit — welches ein Neuron in der Eingangsschicht des ANN beansprucht — auf 1 gesetzt. Im Falle, dass die Kachel leer ist, besitzen alle Neuronen den Wert 0. Es handelt sich also insgesamt um 27 Felder à 3 Klassen, woraus sich mitsamt dem Hungersensorwert 82 Sensorneuronen ergeben.

Neben der Sensorik des Agenten muss auch dessen Aktorik angegeben werden. Er kann per Definition die gleichen Aktionen wie die Benchmark-Agenten für das Versuchsspiel (*Foodchase*) vollführen. Tab. 3.1 gibt die möglichen Aktionen an.

Aktion	*Beschreibung*
NONE	keine Aktion
MOVE_FORWARD	Bewegung einen Schritt vorwärts
MOVE_BACKWARD	Bewegung einen Schritt zurück
TURN_LEFT	Drehung 45° gegen den Uhrzeigersinn
TURN_RIGHT	Drehung 45° im Uhrzeigersinn
EAT	Nahrungsaufnahme
ATTACK	Angriff auf Gegner

Tabelle 3.1: Liste der unterstützten Agenten-Aktionen in *Foodchase*

3.2 Konzept

Abb. 3.3 zeigt die Grundschritte, welche der Agent während jedes Spielzuges durchläuft. Zunächst besitzt der Agent eine bestimmte Policy, welche durch die Gewichtswerte des ANN determiniert ist. Nach dieser Policy wählt er für seine aktuelle Situation die beste Aktion aus, legt die Paarung aus Zustand (Situation) und gewählter Aktion im Speicher (*Kurzzeitgedächtnis*) ab, und führt dann diese Aktion durch. Diese Aktion hat eine Veränderung der Situation in der Spielwelt zur Folge, welche gleichzeitig auch als Beobachter agiert und den Vorgang durch eventuelle Punktevergabe beurteilt. Sollte der Spielzug als positiv oder negativ anzusehen sein — mit anderen Worten: wurden Punkte vergeben — wird der Lernvorgang gestartet, welcher zur Anpassung der ANN-Gewichte des Agenten führt. Dies entspricht der Grundidee von RL (vgl. Kap. 2.3). Das verwendete ANN ist ein einfaches Feed-Forward-Netz, das mit dem BP-Algorithmus (vgl. Kap. 2.2.4) trainiert wird. Dabei dient das ANN als Funktion, um aus den Werten der Neuronen an der Eingangsschicht am Ausgang Q abzuschätzen. Die Eingangsneuronen umfassen sowohl die Sensordaten (visuelle Daten und Hunger) als auch die kodierte Entsprechung der aktuell untersuchten Aktion. Dies wird in Kap. 3.2.1 genauer erläutert.

Für jede Aktion außer NONE (keine Aktion) steht ein Neuron bereit, welches 1 gesetzt ist, sofern es sich um die aktuell untersuchte handelt. Andernfalls ist sein Wert 0. Dies entspricht 6 Neuronen, um alle Aktionen außer NONE zu kodieren. NONE selbst wird durch Nullsetzen aller Aktionsneuronen dargestellt. Zusätzlich enthalten ist ein Neuron, welches den Hungerwert als Fließkommazahl enthält. Sensor- (vgl. 3.1) und Aktionsneuronen ergeben gemeinsam die Eingangsschicht des ANN. Diese hat dementsprechend eine Breite von $27 \times 3 + 1 + 6 = 88$ Neuronen. Jedem Neuron des Eingangsneuronenworts kann somit ein Index zugewiesen werden, wobei 0 bis 80 nach der Formel $(n-1) \times 3 + k$ — n ist der Feldindex lt. Abb. 3.2b, $k \in \{0, 1, 2\}$ der Objektklassenindex[1] — die

[1] Hindernisse (0), Nahrung (1), Gegner (2)

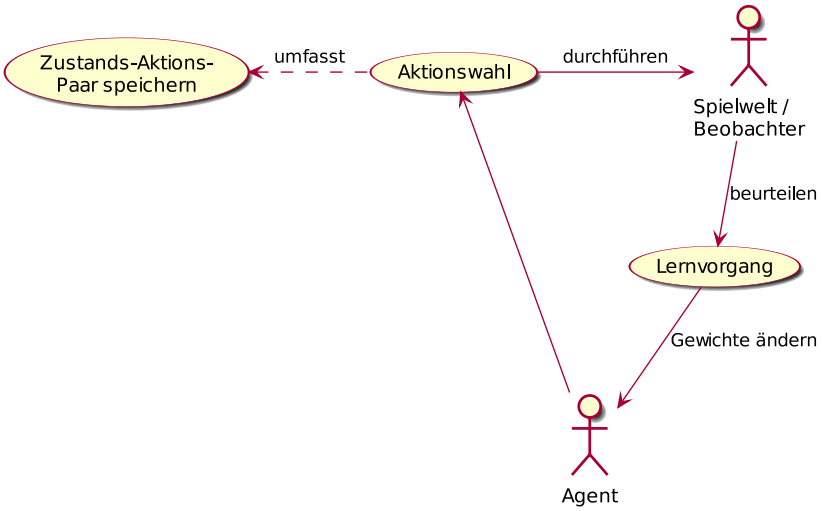


Abbildung 3.3: Anwendungsfalldiagramm des Ablaufes eines Spielzuges

visuelle Sensorik adressiert. Index 81 bezeichnet das Hungerneuron und die restlichen Indizes (82 bis 87) die Aktionsneuronen.

Die beiden Funktionen *Aktionswahl* und *Lernvorgang* werden in den nachfolgenden Kap. 3.2.1 bzw. 3.2.2 im Detail behandelt. Zuletzt wird noch eine Spezialform von NNMind für ein einzelnes Experiment (Verringerung der benötigten Sensorneuronen) in Kap. 3.2.3 vorgestellt. NNMind kann in drei Modi betrieben werden. Bei diesen handelt es sich um:

1. die *Vorprägephase*, welche der Initialisierung der Gewichte des ANN dient. In diesem Modus erfolgt die Aktionswahl rein stochastisch mittels Zufallsgenerator. Um eine schnelle Anpassung der Gewichte zu gewährleisten, ist die Lernrate vergleichsweise hoch. Gegebenenfalls wird dieser Modus anfangs zur Vorprägung eines neuen Agenten für 10 Spieldurchgänge angewandt. Der Agent betreibt in diesem Modus bloß Exploration, keine Exploitation.

2. den *Normalbetrieb*, bei welchem Modus die Aktionswahl, wie zuvor beschrieben, ε-gierig durchgeführt wird. Es wird die gewählte Aktion also üblicherweise vom ANN determiniert, mit einer Wahrscheinlichkeit von ε aber zufällig bestimmt (Exploration).

3. den *lernfreien Modus*, bei welchem die Aktionswahl *gierig* erfolgt. D. h., dass sämtliche Aktionen durch das ANN ausgewählt sind. Es wird in diesem Modus das ANN auch nicht weiter bei Eintritt von positiven oder negativen Ereignissen trainiert. Somit dient er einzig zur Prüfung des bisher gelernten Lösungspfades. Ist die Welt statisch, kann der Agent, sobald er zufriedenstellend trainiert ist, auch in diesem Modus betrieben werden.

3.2.1 Aktionswahl

Bei der *Aktionswahl* — dargestellt in Abb. 3.4 werden durch das ANN zunächst die Q-Werte für die aktuellen Zustands-Aktions-Paare geschätzt. Die Wortlängen sind jeweils in eckigen Klammern angegeben. Zustands-Aktions-Paare setzen sich aus dem Sensorwort und den für jede mögliche Aktion kodierten Aktionswörtern zusammen. Die Q-Werte werden zwischengespeichert und

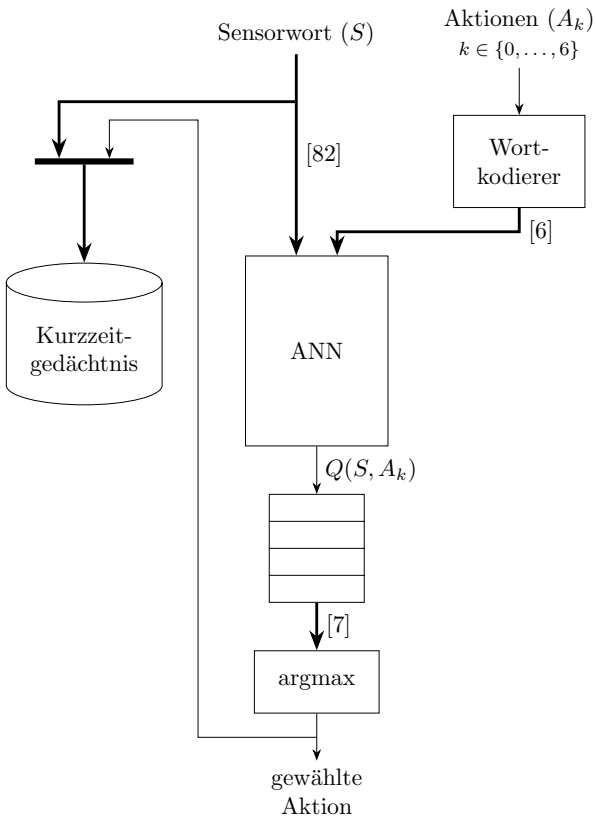


Abbildung 3.4: Aktionsauswahl

schließlich der *argmax*-Funktion zugeführt. Diese gibt jene Aktion zurück, für welche der ermittelte Q-Wert maximal ist.

Zur besseren Veranschaulichung sei dies hier anhand eines vereinfachten Beispiels durchgeführt. Angenommen, das ANN des Agenten verfüge über fünf Sensor- und drei Aktionsneuronen für insgesamt vier mögliche Aktionen inkl. der Aktion NONE, welche für Untätigkeit steht. Die Sensordaten ergeben im gegebenen Beispiel aktuell das Teilwort $(1, 0, 1, 1, 0)$. Dann könnten sich anhand des Zustands der Gewichte des ANN beispielsweise die Q-Werte wie in Tab. 3.2 ergeben. Das Eingangswort ist die Neuronenbelegung an der ANN-Eingangsschicht und setzt sich aus den Sensor- und Aktionsteilwörtern zusammen. In diesem Fall würde der Agent als nächstes eine Drehung nach links vollführen, da Q dafür den höchsten Wert erreicht.

Aktion	Eingangswort	Q
keine Aktion	$(1, 0, 1, 1, 0, 0, 0, 0)$	0,24
Bewegung vor	$(1, 0, 1, 1, 0, 1, 0, 0)$	0,31
Drehung links	$(1, 0, 1, 1, 0, 0, 1, 0)$	0,73
Drehung rechts	$(1, 0, 1, 1, 0, 0, 0, 1)$	0,42

Tabelle 3.2: Beispielergebnisse der $Q(S, A_k)$-Schätzung

Um die gewählten Zustands-Aktions-Paare für den späteren Lernvorgang (Kap. 3.2.2) zur Verfügung zu haben, müssen sie in den Speicher — gleichsam das *Kurzzeitgedächtnis* (engl.: *Short-term Memory (STM)*) — geschrieben werden. Das STM wird nach jedem Entscheidungsprozess mit

dem vollständigen Eingangswort für die gewählte Aktion befüllt, außer bei Auswahl der Aktion *NONE* oder wenn das visuelle Sensorbild beim vorigen Spielzug bereits zuvor exakt gleich war. Bei erstgenannter Ausnahme wird das Tupel einfach nicht gespeichert, im anderen Fall wird das letzte Tupel nicht gespeichert, im anderen Fall wird das letzte Tupel überschrieben. Dies soll verhindern, dass der Agent für wiederholt auftretende Situationen diese unnötig häufig trainiert. Auch soll das Erlernen von Untätigkeit verhindert werden, da ein Agent aus diesem Zustand nur durch eingefügte Exploration oder durch eine eventuelle Änderung der Umgebung wieder herausfinden kann. Um den Lernvorgang, der zur Laufzeit stattfindet, nicht zu zeitaufwändig zu machen, ist das Gedächtnis auf 10 Einträge beschränkt, was ein brauchbarer Kompromiss zwischen Rechenaufwand und der Fähigkeit, komplexere Aktionsfolgen zu lernen, ist.

Da die Architektur ε-gierig ist, findet die Aktionswahl nicht ausnahmslos wie oben beschrieben statt. Stattdessen wird alle 25 Spielzüge ein Explorationsschritt eingefügt, was $\varepsilon = 0{,}04$ entspricht. Es handelt sich hierbei um einen Kompromiss, da der Agent dazu angehalten ist, möglichst exploitativ seine Aktionen zu setzen. Andererseits soll in sinnvollen Intervallen etwas Neues ausprobiert werden, also Exploration stattfinden, um Fortentwicklung des Verhaltens des Agenten zu gewährleisten. ε ist bei NNMind allerdings nicht konstant. Wenn ein *Deadlock* erkannt wird, oder bei zu häufiger Auswahl der Aktion NONE (mehr als 5-mal), wird die Exploration vorgezogen. Unter Deadlock ist hier zu verstehen, wenn der Agent sich permanent um die eigene Achse dreht, oder abwechselnd Vor-Zurück-Bewegungen bzw. Links-Rechts-Drehungen vollführt. Die Exploration zeichnet sich dadurch aus, dass der Agent seine Aktion rein zufällig auswählt, wovon die Aktion NONE allerdings ausgenommen ist. Das Zustands-Aktions-Paar wird wie gewohnt auch ins STM übernommen.

3.2.2 Lernvorgang

Der Lernvorgang geschieht nur direkt nach Auftreten einer Punktevergabe durch das Spiel. Die Spielumgebung selbst dient als Beobachter, beurteilt die Aktionen der Agenten und belohnt oder bestraft jene nach einem Punktesystem. Erhält der Agent Punkte, wird im nächsten Denkzyklus zur Spiellaufzeit der Lernvorgang gestartet. Dabei werden, wie Abb. 3.5 zeigt, aus dem gesamten Inhalt des STM die Trainingsdaten abgeleitet. Die Lernrate des BP-Algorithmus nimmt von einem Anfangswert beginnend ab, je weiter das behandelte Zustands-Aktions-Paar zurückliegt. Dabei errechnet sich die Anfangslernrate zu

$$\mathrm{LR}_0 = C \cdot \mathrm{abs}(R), \tag{3.2}$$

wobei C eine Konstante und R die zuletzt erhaltenen Punkte sind. D. h., dass die Lernrate umso höher ist, je größer die Belohnung bzw. Strafe ausfällt. Die Lernraten der weiter zurückliegenden Schritte ergeben sich rekursiv zu

$$\mathrm{LR}_{k+1} = 0{,}9 \cdot \mathrm{LR}_k. \tag{3.3}$$

Der Wert 0,9 dient hier also als Diskontfaktor für die Lernrate. Die Wahl von C unterscheidet sich in den verschiedenen durchgeführten Versuchen. Die Aufstellung der verwendeten Werte von C findet sich in Kap. 5.3.2. Es muss noch geklärt werden, wie die Trainingsdaten aus den gespeicherten Daten erstellt werden sollen. Dazu ist zu unterscheiden, ob Auslöser des Lernvorganges eine Belohnung oder eine Bestrafung ist.

Für jeden Eintrag im STM werden sämtliche möglichen Zustands-Aktions-Kombinationen mit entsprechenden Q-Werten in die Trainingsdaten übernommen. Den Zusammenhang zeigt Tab. 3.3.

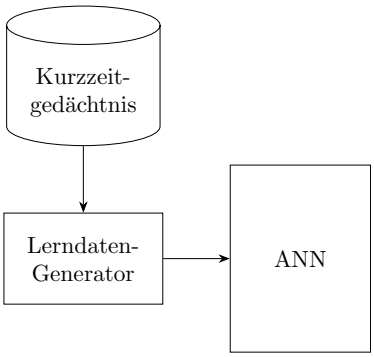


Abbildung 3.5: Lernvorgang

Es wird also zwischen Belohnung ($R > 0$) und Bestrafung ($R < 0$) unterschieden. Die andere Unterscheidung prüft, ob die Aktion, für welche der Trainingsdatensatz gerade erstellt wird, die im STM verzeichnete ist. Dabei wird bei positivem R für die historisch belegte Aktion $Q = 1$ gesetzt, bei negativem $Q = 0$. Im einen Fall soll also die gewählte Aktion gefördert, im anderen für die Zukunft unterdrückt werden. Für die Datensätze, die die nicht gewählte Aktionen abbilden, wird $Q = 0$ bzw. $Q = {}^1/_2$ gesetzt. Damit sollen die restlichen Aktionen im positiven Fall verhindert werden. Im negativen Fall jedoch sollen die übrigen Aktionen eine etwa gleich hohe Auftrittswahrscheinlichkeit haben.

historisch durchgeführte Aktion	ja		nein	
Belohnung (R)	$R > 0$	$R < 0$	$R > 0$	$R < 0$
Soll-Q-Wert	1,0	0,0	0,0	0,5

Tabelle 3.3: Q-Werte beim Training des Backpropagation-Netzwerks

Zum besseren Verständnis möge ein Beispiel dienen. Gehen wir von dem vereinfachten Beispiel aus Kap. 3.2.1 aus, wo lt. Tab. 3.2 die Drehung nach links gewählt wurde, worauf das zugehörige Eingangswort in das STM geschrieben wurde. Der Trainingsdaten-Generator würde nun folgende Datenfolge für den BP-Algorithmus vorbereiten, falls $R > 0$:

$$\begin{aligned}(1,0,1,1,0,0,0,0) &\to 0,\\ (1,0,1,1,0,1,0,0) &\to 0,\\ (1,0,1,1,0,0,1,0) &\to 1,\\ (1,0,1,1,0,0,0,1) &\to 0.\end{aligned}$$

Es wird also dem dritten Tupel, welches den Zustand in Kombination mit der gewählten Aktion (Drehung nach links) repräsentiert, $Q = 1$ zugewiesen, allen anderen $Q = 0$. Für den gegenteiligen Fall ($R < 0$) gilt:

$$\begin{aligned}(1,0,1,1,0,1,0,0) &\to {}^1/_2,\\ (1,0,1,1,0,0,1,0) &\to 0,\\ (1,0,1,1,0,0,0,1) &\to {}^1/_2\end{aligned}$$

Zu beachten ist, dass die Aktion NONE nur bei positiven Ereignissen berücksichtigt wird, bei negativen hingegen nicht. Dadurch soll die Wahl von NONE möglichst unwahrscheinlich werden.

Die eben dargelegte Methode zur Erstellung der Trainingsdaten wird in dieser Arbeit in weiterer Folge als *nicht diskrminierend* bezeichnet. Ursprünglich sah der Entwurf vor, dass die Trainingsdatensätze für positive und negative Ereignisse unterschiedlich behandelt — also *diskriminiert* — werden sollten, weswegen dieser Methode diese Bezeichnung verliehen wurde. Sie zeichnet sich dadurch aus, dass bei Eintritt eines positiven Ereignisses die Daten wie bereits erläutert erstellt werden. Hingegen bei einem negativen Ereignis wird nur das unmittelbare, neueste Zustands-Aktions-Paar zum Lernen herangezogen. Erste Versuche mit der diskriminierenden Lernmethode waren allerdings nicht zufriedenstellend. Das liegt daran, dass nicht, wie ursprünglich angenommen, nur die letzte Aktion zum Misserfolg geführt hat, sondern potentiell auch die Schritte zuvor. Durch ihre Modifikation zur nunmehr geltenden nicht diskriminierenden Methode konnte die Lernleistung des Agenten wesentlich verbessert werden.

3.2.3 Anpassung für Klassenkodierung

Es sei an dieser Stelle noch angemerkt, dass NNMind für eines der Experimente angepasst wurde. Bei dieser Versuchsreihe sollte getestet werden, ob die Anzahl der Sensorneuronen verringert werden kann, indem, statt für jeden der drei Objekttypen (Hindernis, Essbares, Gegner) ein eigenes Neuron je Feld zu benötigen, die Typen binär auf 2 Neuronen zusammengefasst werden können, wie in Tab. 3.4 angegeben. Dadurch lässt sich die Neuronenzahl an der Eingangsschicht von 88 auf $27 \times 2 + 1 + 6 = 61$ Neuronen verringern.

Bitfolge	Objekt
00	nichts
10	Hindernis
01	Essbares
11	Gegner

Tabelle 3.4: 2-Bit-Folgen f. Objekte auf sichtbaren Feldern bei der senorkodierten Variante von NNMind

Abgesehen von der Verringerung der Sensorneuronenanzahl und der damit einhergehenden Änderung der Sensorwortindexierung funktioniert die so angepasste Variante von NNMind exakt wie zuvor beschrieben. Die entsprechenden Details und Ergebnisse zu dem durchgeführten Versuch sind bei *Konfiguration 9* in Kap. 5.3.2.9 ab S. 60 zu finden.

4 Implementierung

Die Implementierung der in der Aufgabenstellung (Kap. 1.3) geforderten Punkte umfasst einerseits die Erweiterung der Miklas-Umgebung. Diese beinhaltet Verbesserungen am GUI, den Mitschnitt des Agenten-Evaluierungssystems von Miklas, eine Spielabbruchbedingung, sowie vorbereitende Änderungen in Hinblick auf die kognitive Architektur. Kap. 4.2 liefert die Details hierzu. Andererseits ist die Implementierung von NNMind Teil der Ausführungen, welche in Kap. 4.3 aufgeführt sind. Die gesamte Implementierung ist in *Java* ausgeführt. Eine Einführung in den zu Beginn bestehenden Quellcode von *Miklas* findet sich in Kap. 4.1. Kap. 4.4 macht Angaben über die für *NNMind* verwendete ANN-API: *Neuroph*.

4.1 Bestehender Quellcode von Miklas

Miklas ist vollständig in Java programmiert und verwendet das *JGameGrid*-Framework [4], das es erlaubt, 2D-Spiele mit gitterartigen Spielfeldern zu erstellen. Des weiteren wird die API *SLF4J* (Simple Logging Facade for Java) [5] fürs Logging genützt. Für *JGameGrid* wird in Kap. 4.1.2 noch eine genauere Beschreibung geliefert.

Die Methode `main()` befindet sich in einer *Launcher*-Klasse, welche als Ausgangspunkt für das Programm dient. Ihre Aufgabe besteht darin, die Konfiguration (vgl. Kap. 5.1) zu lesen und eine Spielinstanz zu starten. Zum Spielstart instantiiert der Launcher die Klasse `Visualization`, eine Erweiterung der Swing-Klasse `JFrame`, welche das Spielfeld und zusätzliche Informationen und Kontrollen enthält, sowie die für Soundeffekte (`SoundManagerImpl`), Musik (`MusicManagerImpl`) und die Punktevergabe (`EvaluationManager`) zuständigen Klassen. Ihre Instanzen werden der ebenfalls initialisierten `GameEngineImpl`-Klasse zugewiesen, welche wiederum von `Visualization` referenziert wird. `GameEngineImpl` dient dabei als Sammelpunkt aller für den Spielablauf benötigten Komponenten-Instanzen, auch der `GameGrid`-Instanz (vgl. Kap. 4.1.2), welche im von `Visualization` aufgebauten Fenster als GUI-Komponente eingefügt ist. Das Klassendiagramm in Abb. 4.4 auf S. 35 zeigt, wie die oben genannten Klassen organisiert sind. Das Schaubild zeigt allerdings den Status nach der Bearbeitung.

Die Objekte auf dem Spielfeld werden durch Instanzen der Klasse `Entity` beschrieben. Alle Objekte, egal ob belebt oder unbelebt (Nahrung, Hindernisse), werden als *Entitäten* angesehen. `Entity` enthält auch eine Referenz auf die zugehörige Körperklasse `Body`. Darin befinden sich alle Informationen, die die Rolle und das Aussehen der Entitäten in der Spielwelt beschreiben: ob es sich um eine (un)belebte Entität handelt, Sprites und Soundeffekte, mögliche bemerkbare Ereignisse und

durchführbare Aktionen, sowie den möglichen Verweis auf eine *Mind*. Die Mind stellt die Intelligenz eines Agenten bereit. Für dieses Projekt werden für die Benchmark-Agenten die ESiMA- und OSiMA-Architektur verwendet. Im Gegensatz dazu nutzen Agenten in verschiedenen Versuchen eine Version der für diese Arbeit kreierten Architektur *NNMind*.

4.1.1 Spielekonfiguration

Die Konfiguration von *Miklas* erfolgt über *INI*-Dateien im Unterverzeichnis `conf` des Programmhauptverzeichnisses. Aufgerufen wird sie vom *Launcher* beim Programmstart durch Instantiierung der Klasse `ConfigLoader` mithilfe der statischen Methode `getConfig()`. Die Inhalte des `ConfigLoader`s werden nach Ausführung von `init()` gesetzt und über Getter-Methoden allgemein abrufbar. Dabei wird zunächst die Datei `conf/game.ini` geladen, welche lediglich den Verweis auf eine Spielkonfigurationsdatei über die Variable `modfilepath` enthält. Damit lautet der Inhalt dieser Datei im gegebenen Fall zum Beispiel:

```
modfilepath = conf/foodchase.ini
```

Jene Datei enthält wiederum sämtliche Informationen, die für das korrekte Funktionieren des Spiels unerlässlich sind. Diese Informationen umfassen allgemeine Angaben zum Spiel (Titel, Simulationsperiodendauer, Zellengröße des Spielfeldes in Pixel), den Aufbau des Spielfeldes, Hintergrundmusik sowie die komplette Konfiguration der Agenten. Deren Konfiguration beinhaltet die jeweils verwendeten *Sprites* (bildliche Repräsentionen der Figur für verschiedene Situationen), die möglichen Aktionen, welche sie ausführen, und die möglichen Ereignisse, auf welche sie reagieren können sollen, sowie die Angabe einer *Mind*. Bei der Mind kann es sich um eine KI-Architektur oder um die Steuerung für einen menschlichen Spieler handeln. Letztere Möglichkeit wird allerdings für diese Arbeit nicht genutzt. Sämtliche Eigenschaften werden im Format

```
<Gruppe>[.<ID>][.<Subgruppe> [...]].<Eigenschaft>=<Wert>
```

angegeben. Einige Egenschaftengruppen (u. a. Agentendefinition, Ereignisse) lassen die Definition mehrerer Objekte zu, indem für diese jeweils eine eigene *ID* angegeben wird.

Alle diese Gruppen besitzen auch eine Eigenschaft, welche das Objekt durch einen eindeutigen Namen identifizert. Andere Eigenschaften können auf andere definierte Objekte verweisen. Dies trifft etwa auf Agenten zu, deren Körpern jeweils eine Mind zugewiesen wird. Kap. 5.1 gibt die Konfiguration des für diese Arbeit verwendeten Spiels *Foodchase* wieder.

Das Konfigurationsdateiformat unterstützt auch die Erstellung von Kommentaren. Sie werden durch Eingabe einer Raute (#) begonnen und gelten bis zum Zeilenende.

4.1.2 JGameGrid

Entwickelt von Aegidius Plüss bietet *JGameGrid* [4] ein einfaches Framework zur Kreierung gitterbasierter 2D-Spiele in Java. Dabei wird sowohl die Möglichkeit geboten, ein Spielfeld als eigenes Fenster zu erstellen, als auch, es als Komponente in einem AWT- oder Swing-Fenster einzufügen. Die letzte Variante ist für *Miklas* in Verwendung. Weiters ist die Zellengröße des Spielfeldes beliebig konfigurierbar. Sämtliche Objekte werden als *Aktoren* angesehen, deren Sprites sich auf dem

Spielfeld platzieren und rotieren lassen. *JGameGrid* kümmert sich um die Zeichnung des Spielfeldes und aller darauf befindlichen Aktoren. Es steht auch Kollisionserkennung zwischen Aktoren zur Verfügung. Außerdem gibt es spezielle Aktorklassen, um die Platzierung von z. B. Schaltflächen und Checkboxen auf dem Spielfeld zu ermöglichen. Zusätzlich unterstützt *JGameGrid* Tonausgabe im WAV- und MP3-Format mitsamt Steuermethoden (Wiedergabe, auch endlos; Pause). Auch die Simulationsperiode — das Zeitintervall zwischen zwei Spielzügen — lässt sich einstellen. Die wichtigsten Klassen für die Entwicklung in Miklas und die darin enthaltenen Elemente sind im UML-Klassendiagramm in Abb. 4.1 zu finden.

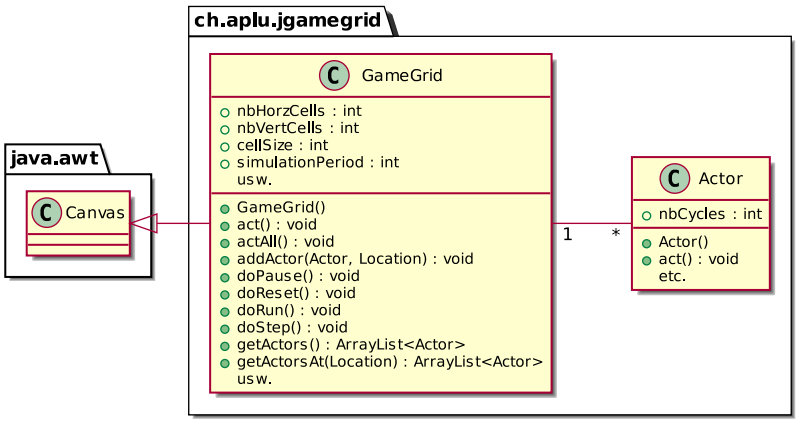


Abbildung 4.1: UML-Klassendiagramm der wichtigsten für Miklas nötigen Klassen und zugehörigen Elemente von *JGameGrid* von Aegidius Plüss

Ein Spielfeld kann durch Instanzierung der Klasse `GameGrid` erstellt werden. Die Klasse ist von `Canvas` abgeleitet. Dadurch ist es einerseits möglich, eine eigene Klasse davon abzuleiten, um das Spielfeld in einem eigenen Fenster automatisch zu kreiern. Alternativ kann die Instanz auch einem Fremdfenster als Objekt hinzugefügt werden. Dem Spielfeld können dann die gewünschten Aktoren durch `GameGrid.addActor()` eingefügt werden, welche durch Ableitung der Klasse `Actor` deklariert werden. Die eigenen Agenten müssen die Methode `Actor.act()` neu definieren, welche die Anweisungen eines Aktors während eines Zeitschrittes enthält. In *Miklas* ist die Klasse `Entity` die Erweiterung der JGameGrid-Klasse `Actor`. Für weitere Informationen zu *JGameGrid* gibt es einige Tutorials unter [4], die volle Dokumentation der API ist unter [3] abrufbar.

4.2 Erweiterung von Miklas

Die lt. Kap. 1.3 zu implementierenden Neuerungen an der Miklas-Spielumgebung sind die Erweiterung des GUI mit einem Startmenü, verbesserten Kontrollen und Statistik-Anzeigen im Spielfenster sowie der Möglichkeit zur Anpassung der Spielkonfiguration aus Miklas heraus. Außerdem dazu gehörig ist die Definition der Spielabbruchbedingung solcherart, dass das Spiel endet, sobald nur noch ein Agent lebend auf dem Spielfeld verbleibt. Letztendlich umfasst dieses Kapitel auch die Erweiterung des *Evaluators*, sodass dieser die benötigten Daten dem GUI zur Verfügung stellt und das Führen eines Spiellogs ermöglicht.

Für Miklas dient der sog. *Launcher* (eine eigene Java-Klasse) als Startpunkt des Programms. Der bisherige Launcher startet lediglich ein neues Spiel, welches dann bis zur manuellen Schließung läuft. Die erste Neuerung an Miklas ist eine neue Launcher-Klasse namens `GUILauncher`, welche

(a) Miklas-Startfenster für gewähltes Spiel *Foodchase*

(b) Konfigurationseditor

Abbildung 4.2: Neu hinzugekommene GUI-Elemente der Miklas-Spielumgebung

das GUI durch ein Startfenster, wie in Abb. 4.2a abgebildet, erweitert. Dieses Startfenster zeigt ein Menü mit den drei Optionen: *New Game*, *Settings* und *Quit*. Bei Klick auf *Settings* wird die Konfigurationsdatei des aktuell ausgewählten Spieles angezeigt (Abb. 4.2b) und kann bearbeitet und abgespeichert werden, bevor das nächste Spiel gestartet wird. Das Banner oberhalb der Schaltflächen ist in der Spielkonfiguration als Dateipfad zu einer Bilddatei einstellbar. Nur muss das gewünschte Spiel — bzw. die zugehörige Konfigurationsdatei — wie bisher durch Editierung der Datei `game.ini` ausgewählt werden.

Bei Spielstart durch Klick auf *New Game* im Hauptmenü werden vom Launcher alle benötigten Klassen initialisiert und das Spielfenster (siehe Abb. 4.3) angezeigt. Die zu initialisierenden Klassen sind `Visualization`, `GameEngineImpl`, `EvaluatorManager`, `SoundManagerImpl` und `MusicManagerImpl`, wobei die letzten drei nur über Schnittstellen referenziert werden. Abb. 4.4 zeigt ein UML-Klassendiagramm der wichtigsten Klassen und Schnittstellen, bzw. jener, welche hinzugefügt oder editiert werden mussten.

Wie Miklas und hieraus ein neues Spiel gestartet wird, zeigt schematisch Abb. 4.5 als Sequenzdiagramm. Dieses gibt den Aufrufsablauf nicht im Detail, sondern zum grundsätzlichen Verständnis wieder. Bei Start der Klasse `GUILauncher` wird zunächst die Klasse selbst instantiiert und die Konfiguration mithilfe des `ConfigLoader`s gelesen. `getConfig()` gibt als statische Methode die Instanz für diesen zurück. Die Methode `init()` aus `ConfigLoader` bzw. `GUILauncher` lädt die eigentliche Spielkonfiguration aus der gewählten Konfigurationsdatei, bzw. baut das Startfenster auf. Bei Klick auf *New Game* werden dann die nötigen Instanzen für Sound-, Musik- und Evaluierungs-Manager erstellt und an die ebenfalls erstellte Instanz von `GameEngineImpl` mittels der Setter-Methoden übergeben und initialisiert. Zuletzt wird das Spielfenster (Klasse `Visualization`) erstellt, konfiguriert und anschließend über `doRun()` das Spiel gestartet. Zu beachten ist hierbei, dass die Klasse `GameGrid` in `GameEngineImpl` instantiiert wird, weswegen auch die Einfügung der Agenten und verschiedenen Objekte (*Aktoren*, vgl. Kap. 4.1.2) durch die in `GameEngineImpl` definierte Methode `addActorsToWorld()` erfolgt.

Das Spielfenster (Klasse `Visualization`) wurde um die Schaltflächen zur Spielsteuerung sowie die Baumauflistung der Agentenstatistiken erweitert. Um die Spielabbruchbedingung, dass das Spiel solange andauert, bis nur noch ein Agent am Leben ist, einzuhalten, wird mithilfe der Methode `countAnimateBodies()` in `Visualization` die Anzahl der noch auf der Spielfläche befindlichen

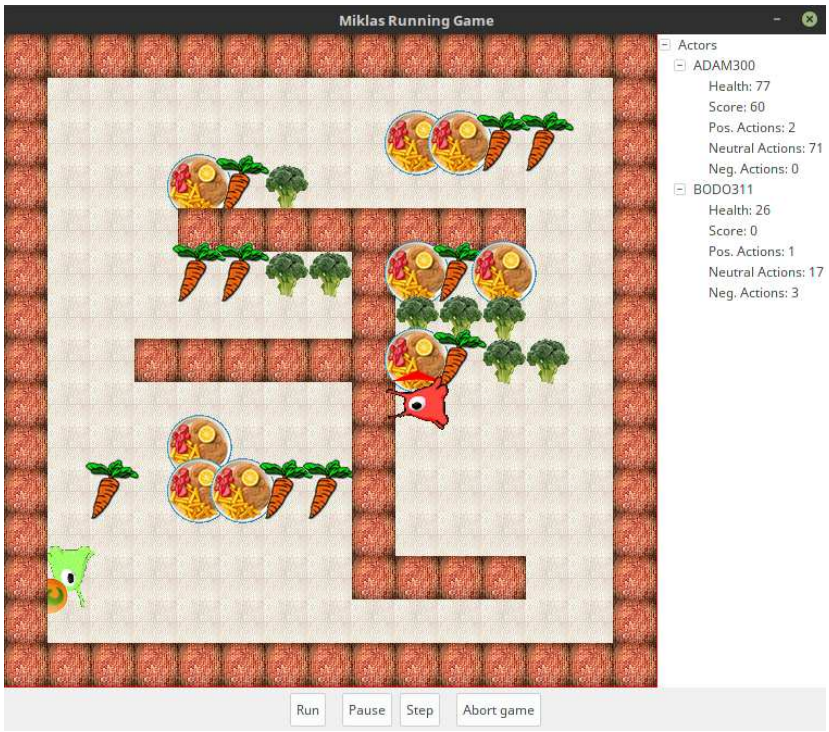


Abbildung 4.3: Spieloberfläche von *Foodchase*

Agenten gezählt. Fällt sie auf 1, endet das Spiel mittels Aufruf von `gameOver()`, wodurch das Spiel abgebrochen, das Spielfenster (`Visualization`) verborgen und das Startfenster wieder angezeigt wird. Versuche, die Instanzen von `Visualization` und `GameGrid` zu zerstören, sind an den vorhandenen Interdependenzen gescheitert. Das hat zur Folge, dass bei Start eines weiteren Spiels ein weiterer Thread erstellt wird, obwohl der alte nach wie vor existiert. Dies macht sich als Quasi-Speicherleck bemerkbar und kann nach mehreren Spielen zur selben Programmlaufzeit zum Absturz führen. Das Spielfenster ist so angepasst, dass es auch die bisherige Launcher-Klasse unterstützt. Ist diese in Verwendung, beendet das Programm bei Spielende vollständig.

Änderungen waren auch am *Evaluator* notwendig, um den Mitschnitt der Statistiken und die Übergabe der Belohnung an die kognitive Architektur zu ermöglichen. Diese Änderungen umfassen die Implementierung der Methode `getLastReward()` in verschiedener Ausführung in der Klasse `EntityEvaluation` sowie der Schnittstelle `EvaluatorManagerMindInterface`. Dabei fragt die Schnittstellen-Variante von `getLastReward()`, die in `EvaluatorManager` definiert ist, die Punktestandänderung für einen bestimmten Agenten aus `EntityEvaluation` ab. Außerdem gehören für die Ermöglichung der Notierung des Zeitstempels und der Agentenpositionen am Spielfeld im Mitschnitt die Methoden `getTimeStamp()` und `updatePerceptionForEvaluation()` zur von `Visualization` implementierten Schnittstelle `VisualizationEvaluationInterface` zu den Anfügungen. Über die Methode `appendLogEntry()` wird bei Update der Statistiken ein Eintrag in die Log-Datei geschrieben, der folgendermaßen im CSV-Format aufgebaut ist:

```
<Agenten-ID>, <Zeitindex>, <Gesundheit>, <+>, <->, <0>, <X>, <Y>
```

Die *Agenten-ID* besteht aus dem Namen des Agenten und einer angehängten generischen Nummer. Der *Zeitindex* ist die seit Spielbeginn ansteigende Spielzugzahl, die mittels `getTimeStamp()`

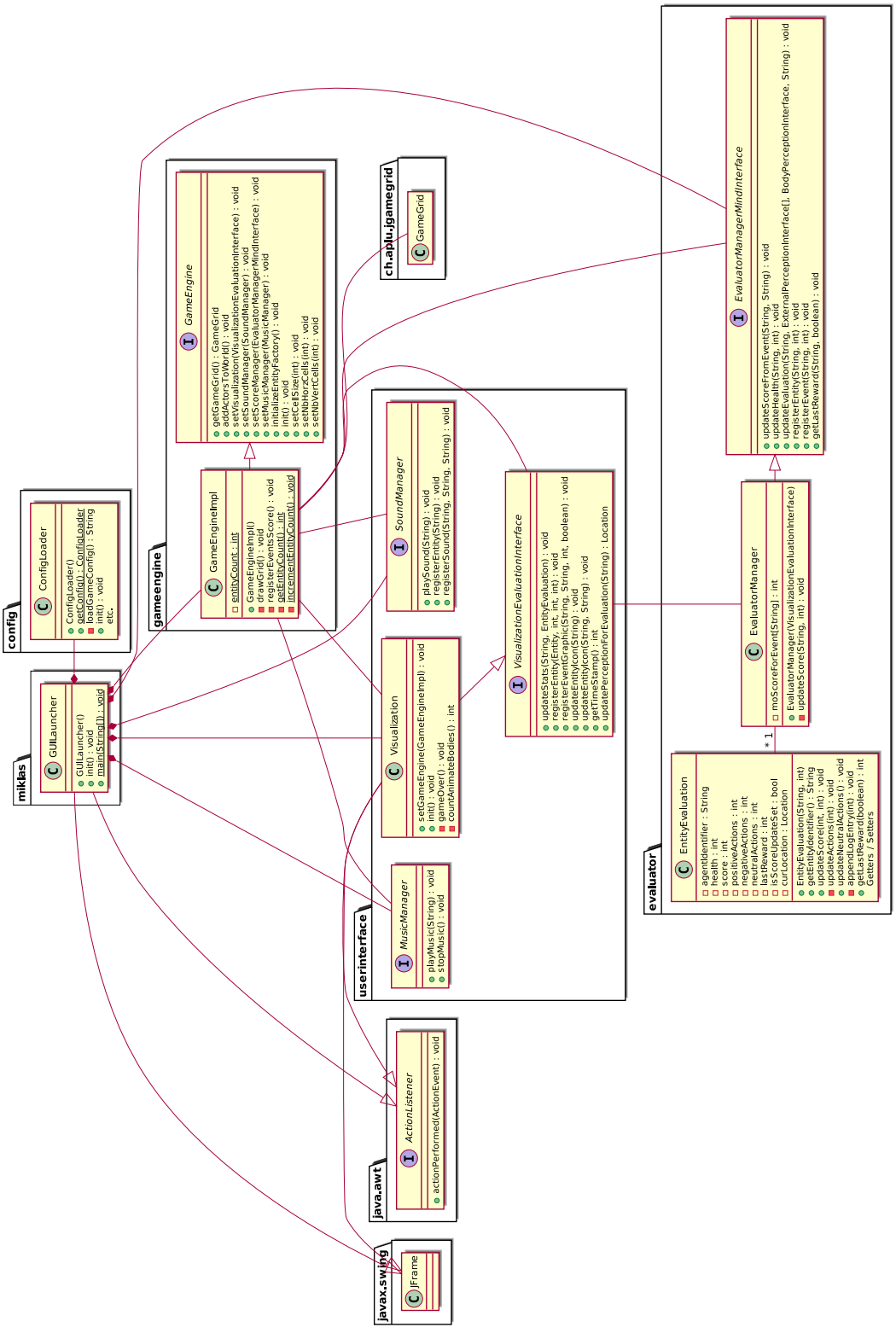


Abbildung 4.4: Klassendiagramm der Änderungen an der Miklas-Benutzeroberfläche mitsamt Attributen und Methoden (*public* [grün], *protected* [gelb], *private* [rot])

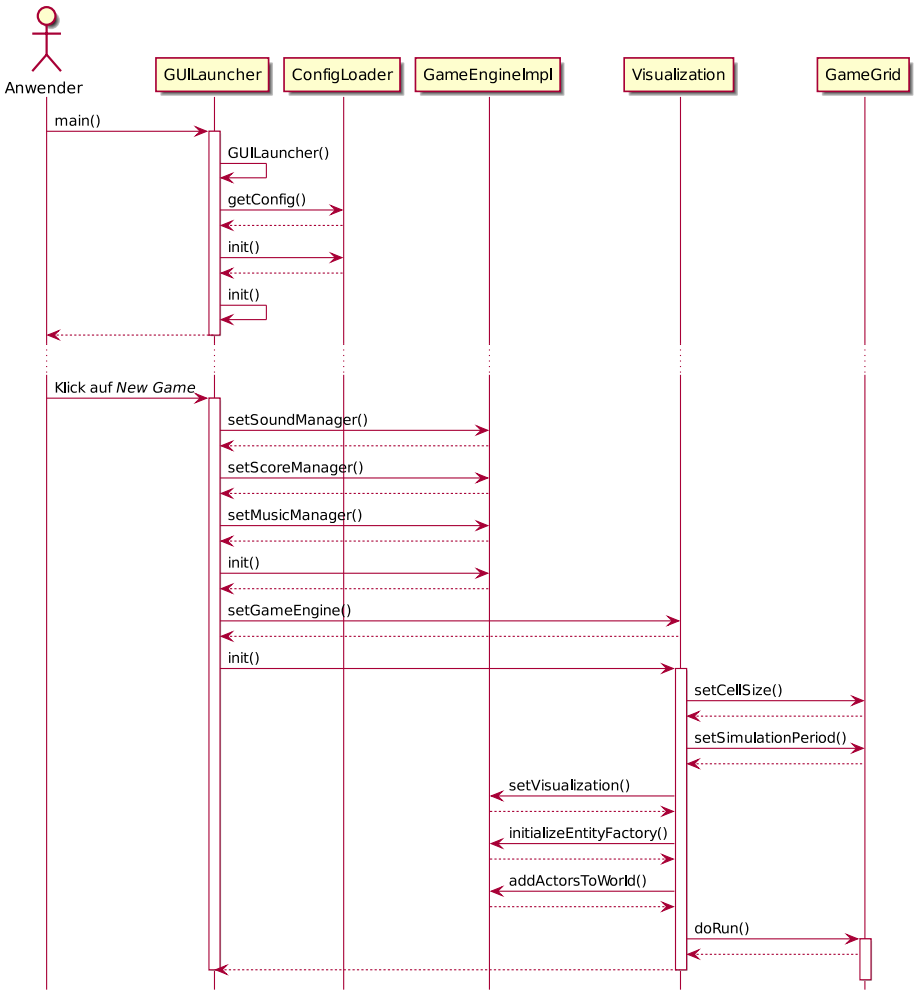


Abbildung 4.5: Sequenzdiagramm für Spielstart mithilfe des *GUILauncher*

abrufbar ist. Außerdem sind die Anzahlen der erfahrenen positiven (+), negativen (-) bzw. neutralen Ereignisse (*0*) angegeben. Zuletzt bedeuten X und Y die Spielfeldkoordinaten der aktuellen Position des Agenten, wobei $X = 0, Y = 0$ auf die Spielfeldkachel in der linken oberen Ecke verweist. Diese können mittels `updatePerceptionForEvaluation()` ermittelt werden.

4.3 NNMind

Die Implementierung von *NNMind* umfasst sowohl die Klasse `ANNArch2` sowie die *Mind*-Klasse `NNMind2`, welche als Schnittstelle zwischen der Spielumgebung von Miklas und der kognitiven Architektur fungiert. Abb. 4.6 zeigt das zugehörige Klassendiagramm. Aufgabe der Mind (`NNMind2`) ist, die Sensordaten für die Architektur vorzubereiten und zu übergeben, und die resultierende Aktion Miklas mitzuteilen. Dafür wird die Schnittstelle `ExternalMindControlInterface` von `NNMind2` implementiert. Diese stellt die Methode `startCycle()` bereit, welche vom Programm bei jedem Zeitschritt für jeden Agenten aufgerufen wird, und den Denkprozess initiiert. Der Konstruktor von `NNMind2` speichert die Instanz der Klasse `Body` des zugehörigen Agenten als Attribut des Typs `ExternalMindBodyInterface` ab.

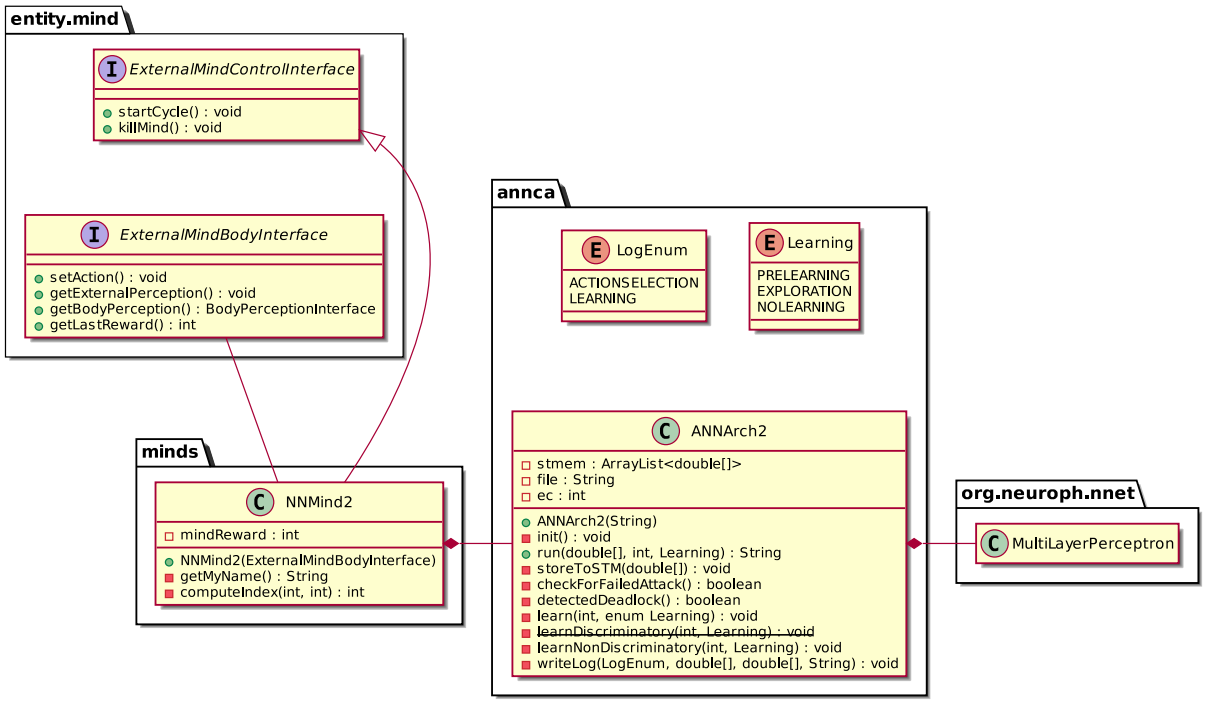


Abbildung 4.6: Klassendiagramm der kognitiven Architektur

Ebenso instantiiert `NNMind2` die Klasse `ANNArch2`, die *Neuroph* [6] für das BP-Netz verwendet. Nähere Informationen zu Neuroph sind in Kap. 4.4 zu finden. Der aktuelle Zustand der kognitiven Architektur wird nach jedem Trainingsvorgang abgespeichert. Als Dateiname wird in diesem Zusammenhang der Name des Agenten verwendet, welcher mittels der Methode `getMyName()` der Mind-Klasse ermittelbar ist. Die Klasse `ANNArch2` enthält im Grunde das *ANN* (Instanz der Klasse `MultiLayerPerceptron` aus *Neuroph*) und das *STM*, welches als einfache `ArrayList` mit den Werten der Eingangsschichtneuronen als Inhalt implementiert ist. Der Konstruktor von `ANNArch2` ruft die Methode `init()` auf, welche versucht, die als Parameter gegebene Datei zu öffnen und das ANN daraus zu laden. Existiert sie noch nicht, wird das ANN laut der gegebenen Konfiguration kreiert.

Die Ziffer 2 in den Klassennamen `NNMind2` bzw. `ANNArch2` bezeichnet die zweite Version dieser Klassen. Bei ihnen handelt es sich um die erste funktionsfähige Version. Daneben gibt es noch die Version 3 (mit Klassen `NNMind3` und `ANNArch3`), welche eine Anpassung für Versuche mit kodiertem Sensorwort darstellt, wie in Kap. 3.2.3 vorgestellt.

Der Betriebsmodus kann einen der drei Werte aus der Enumeration `Learning` annehmen, deren Bedeutung, den Angaben aus Kap. 3.2 folgend, hier angegeben ist:

PRELEARNING entspricht dem *Vorprägemodus*, der zum initialen Setzen der ANN-Gewichte eines frischen Agenten dient.

EXPLORATION ist der ε-gierige *Normalbetrieb*.

NOLEARNING bezeichnet den Operationsmodus, bei dem die Aktionswahl gierig und außerdem kein weiteres Lernen erfolgt. Da das Verhalten hier fixiert ist, dient es nur zum Testen eines gut trainierten ANN.

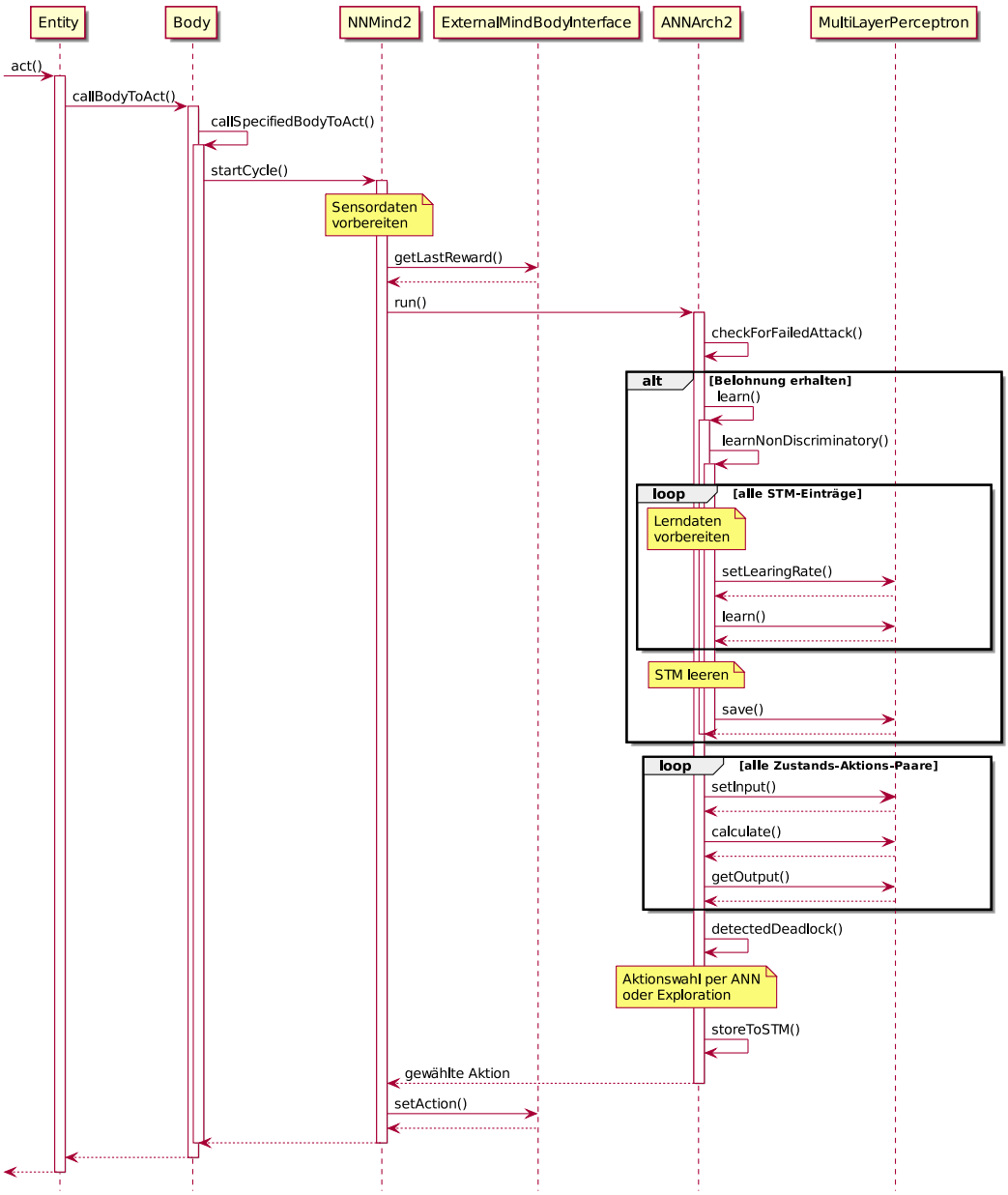


Abbildung 4.7: Schematisches Sequenzdiagramm bei Aufruf von *Entity.act()* durch die Spiel-Klasse *GameGrid*

Wie in Kap. 4.1.2 dargelegt, ruft `GameGrid` aus der *JGameGrid*-API bei jedem Zeitschritt für jeden *Actor* die Methode `act()` aus. Miklas definiert die Klasse `Entity`, welche `Actor` erweitert. Darin enthalten ist auch der Verweis auf den `Body` des Agenten. Abb. 4.7 zeigt verkürzt, wie bei Aufruf von `act()` die *Mind* des Agenten zur Entscheidung gelangt, welche Aktion durchzuführen ist. Es muss darauf hingewiesen werden, dass in Abb. 4.7 `ExternalMindBodyInterface` und `Body` auf dasselbe Objekt verweisen. Da allerdings `NNMind2` auf `Body` über die Schnittstelle zugreift, ist dies so notiert.

Für alle Agenten, welche `NNMind2` verwenden, werden von `startCycle()` zunächst die Sensordaten geholt und für die Übergabe an `ANNArch2` aufbereitet. Die Neuronenwerte der Sensoren werden

zusammen mit der letzten Punktestandänderung — abgerufen über `getLastReward()` — als Parameter der Methode `run()` an die Architektur übergeben. Die Vorbereitung der Sensordaten — als Array von Doubles organisiert — bedarf der Indexierung der Spielfeldkacheln, wie in Abb. 3.2 auf S. 23 angegeben. Der Index der gerade bearbeiteten Kachel wird in `NNMind2` durch die private Methode `computeIndex()` berechnet, welche in Abb. 4.7 keine Erwähnung findet.

`ANNArch2.run()` gibt nach ihrer Abarbeitung die gewählte Aktion als String zurück, sodass dieser sogleich mit `setAction()` dem Agentenkörper übergeben werden kann. Dabei verfährt sie so, dass sie zunächst überprüft, ob die letzte Aktion eine Attacke war, welche fehlgegangen ist. Dies passiert, wenn bei Durchführung einer Attacke kein Gegner auf der Kachel direkt vor dem Agenten steht. In diesem Fall wird der Belohnungswert — welcher als Parameter an `run()` übergeben worden ist — nachträglich intern um 10 verringert. Dieser Schritt ist nötig, da in der Spielkonfiguration für diesen Umstand keine Bestrafung vorgesehen ist. Dennoch soll der Agent aus dem resultierten Fehlschlag lernen, dass Attackieren nur bei Vorhandensein eines Gegners in Reichweite sinnvoll ist.

Die Methode fährt mit Aufruf von `learn()` fort, falls der Belohnungswert ungleich 0 ist. Diese leitet direkt auf `learnNonDiscriminatory()` um. Der Grund ist, dass in der obsoleten Methode `learnDiscriminatory()` der alte Lernalgorithmus (vgl. Kap. 3.2.2) für weitere Experimente konserviert ist. Es werden nun die Lerndaten für alle STM-Einträge vorbereitet und mit lt. Glg. 3.3 abgestuften Lernraten dem ANN eingeprägt. Zuletzt wird das STM geleert und das ANN in seinem neuen Zustand im Dateisystem gesichert.

Es folgt die Aktionsauswahl mithilfe des ANN, indem für sämtliche möglichen Aktionen die Q-Werte berechnet werden, deren Maximum gesucht und so die nächste Aktion ausgewählt wird, wie in Kap. 3.2.1 beschrieben. Anschließend wird mit `detectedDeadlock()` festgestellt, ob ein Deadlock aufgetreten ist. Dies ist der Fall, wenn der Agent sich einmal vollständig im Kreis gedreht hat, oder beständig abwechselnd vor und zurück geht, bzw. sich nach links und rechts wendet. Kommt es zu einem Deadlock oder gebietet es die ε-Gier der Architektur, wird ein Explorationsschritt anberaumt. Das bedeutet, dass die Aktion vom Zufallsgenerator bestimmt wird. Ansonsten wird jene Aktion gewählt, für welche der Q-Wert maximal ist.

Zuletzt wird das aktuell gewählte Zustands-Aktions-Paar mithilfe von `storeToSTM()` ins STM geschrieben. Anzumerken ist, dass, wenn das Sensorbild sich seit dem letzten Zyklus nicht geändert hat, das letzte Wort überschrieben wird. Ansonsten werden die Zustands-Aktions-Paare jeweils am Listenanfang eingefügt. Um den Lernvorgang, der zur Spiellaufzeit immer bei Auftreten einer Punktezahländerung erfolgt, nicht zu rechenaufwändig zu machen — das Training kann mitunter etliche Sekunden dauern, währenddessen das Spiel einfriert —, ist die Größe des STM auf 10 Einträge beschränkt (vgl. Kap. 3.2.1). Es wird daher bei zu großer Anzahl im Speicher das jeweils älteste Wort gelöscht.

Es werden auch mit der Methode `writeLog()` Log-Einträge über die Aktionswahl und den Lernvorgang der Architektur aufgezeichnet, welche dabei helfen, im Nachhinein den Entscheidungsprozess mitsamt den Explorationsschritten nachzuvollziehen. Auch Informationen zum Lernvorgang werden mitgeschrieben. Das Format der Log-Einträge ist dergestalt:

```
<Typ> : <Eingangsneuronenbild> : <weitere Informationen>
```

Abhängig vom Typ des Log-Eintrags unterscheiden sich die dahinter folgenden Angaben. Es folgt eine Aufzählung der möglichen Eintragstypen:

1. `nop` (normale Operation) gibt die Ergebnisse der gierigen Aktionswahl wieder. Als weitere Informationen folgen alle errechneten Q-Werte sowie, getrennt durch einen Doppelpunkt, der Name der resultierenden Aktion. Das Eingangsneuronenbild beinhaltet hier nur die Sensordaten.

2. `learn` wird nach Durchführung eines Lernvorgangs geschrieben. Zusätzlich wird der Wert der Belohnung/Bestrafung angegeben. Log-Einträge werden für sämtliche gelernten STM-Einträge erstellt. Das Eingangsneuronenbild ist gleich dem gespeicherten STM-Eintrag.

3. `check` folgt direkt auf den *learn*-Eintrag. Er unterscheidet sich lediglich durch die Angabe der Schätzung des neuen Q-Wertes nach dem Lernen und dient dazu, den neuen Schätzwert von Q nach dem Lernvorgang zu testen.

4.4 Neuroph

Dieser Abschnitt behandelt die Beschreibung der für die Implementierung der kognitiven Architektur benutzten Programmbibliothek zur Erstellung und Verwendung von ANNs: *Neuroph* [6] in der Version 2.94. Das zugehörige Klassendiagramm mit einer Zusammenstellung der für diese Arbeit wichtigsten Klassen ist Abb. 4.8 zu entnehmen. Die Bibliohtek bietet allerlei Typen neuronaler Netze und verschiedene zum Gebrauch mögliche Lernregeln. Hier ist lediglich das `MultiLayerPerceptron` mit *Backpropagation* als Lernregel von Interesse.

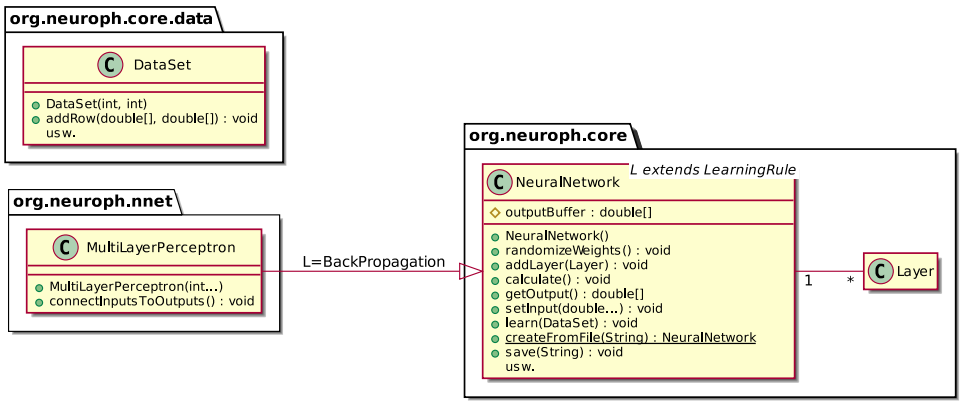


Abbildung 4.8: Klassendiagramm der wichtigsten benötigten Klassen und Methoden der Neuroph-API

Die Klasse `NeuralNetwork`, die `MultiLayerPerceptron` erweitert, enthält, in Schichten (Klasse `Layer`) organisiert, die Neuronen und Neuronalverbindungen des Netzes. Für die Neuronen und Verbindungen gibt es jeweils Klassen, welche hier nicht aufgeführt sind, da die NNMind-Architektur sie nicht direkt verwendet. Das Perceptron kann mit einer Auflistung von Integers, welche die Neuronenzahl je Schicht angibt, initialisiert werden. Es kann auch in eine Datei gesichert und aus solcher wieder geladen werden. Hierfür sind die Methoden `createFromFile()` (statisch) und `save()` verantwortlich. Mit `randomizeWeights()` können sämtliche Gewichte des ANN zufällig neu gesetzt werden.

`setInput()` ermöglicht das Setzen der Neuronenwerte der Eingangsschicht. Darauf kann durch Aufruf von `calculate()` das Ergebnis am Ausgang berechnet werden, um es sodann mithilfe

von `getOutput()` auszulesen. Das Netz kann durch Aufruf von `learn()` trainiert werden. Diese Methode erwartet als Parameter ein Objekt des Typs `DataSet`, welches mit den Trainingsdaten angefüllt werden kann. Der Methode `addRow()` werden dazu die Werte an der Eingangs- und die gewünschten Werte an der Ausgangsschicht als Arrays übergeben.

Neben der bloßen API ist auch das *Neuroph-Studio*, ein GUI-Editor für die Erstellung und das Training von selbst erstellten ANNs, erhältlich. In diesem können ANNs graphisch dargestellt und editiert werden. Für weitere Informationen über *Neuroph* sei auf [6] verwiesen. Es wurde auch die Verwendung anderer Bibliotheken erwogen (z. B. *DL4J* [7]). Letztlich wurde Neuroph aus folgenden Gründen der Vorzug gegeben:

- API für Java
- einfach installier- und erlernbar
- BP-Netze schnell erstellbar bzw. lad- und speicherbar
- einfach mit zeitlich verschiedenen Datensätzen trainierbar

5 Evaluierung

Für die Evaluierung wurden zahlreiche Spieldurchläufe mit den beiden zur Verfügung gestellten Benchmark-Architekturen (*ESiMA* und *OSiMA*) sowie mit der Architektur *NNMind* durchgeführt. Es wird zunächst die Konfiguration des Spieles *Foodchase*, welches für die Testung und Evaluierung der verwendeten Architekturen gewählt wurde, in Kap. 5.1 vorgestellt. Auch die Vorgangsweise bei der Wahl der nächsten Aktion von ESiMA bzw. OSiMA ist in Kap. 5.2 angegeben.

Anschließend werden die Versuchsanordnungen und dabei erhaltenen Ergebnisse in Kap. 5.3 diskutiert. Die Ergebnisse der Benchmark-Tests werden in Kap. 5.3.1 präsentiert. Dadurch soll veranschaulicht werden, zu welchen Leistungen diese im gegebenen Spielumfeld imstande sind. Es wurden hierzu sowohl für die *ESiMA*- als auch die *OSiMA*-Architektur jeweils 50 Durchläufe gespielt und die Statistiken aufgezeichnet. Diese umfassen die Gesamtpunktzahl, die Anzahl der positiven, negativen und neutralen Aktionen der Opponenten, die Spieldauer in Zügen, sowie den jeweiligen Gewinner. Als Gewinner gilt, wer als letzter Überlebender übrig ist.

In Kap. 5.3.2 folgen die Ergebnisse der mit der Architektur *NNMind* durchgeführten Versuche in verschiedenen Versuchskonfigurationen. Diese Konfigurationen unterscheiden sich durch den Aufbau des neuronalen Netzes und die Organisation der Eingangsdaten, sowie durch die äußeren Umstände in der Spielumgebung. Unter letztgenannten sind z. B. die Auswahl der Gegner-Architektur oder die Startposition zu verstehen. Für jede Konfiguration wurden je 10 Agenten trainiert. Die für einen trainierten Agenten aufgewandten Spieldurchläufe bilden eine *Episode*. Zum initialen Setzen der Gewichte wurden in den meisten Konfigurationen 10 Durchgänge im Vorprägemodus (vgl. Kap. 3.2) vorgeschoben. Deren Resultate sind irrelevant, da dieser Modus einzig der Voreinstellung der ANN-Gewichte dient. Anschließend wurden jeweils 50 Durchgänge, die eine Episode ausmachen, im Normalbetrieb gespielt und die Resultate aufgezeichnet.

5.1 Spielkonfiguration

Die Spielkonfigurationsdatei enthält alle für die Definition eines unter *Miklas* laufenden Spieles benötigten Eigenschaften. Dies umfasst allgemeine Simulationsparameter, wie die Größe und Aufbau des Spielfeldes, die Simulationperiode (Zeit zwischen Spielzügen) und Musik. Darauf wird der Aufbau des Spielfeldes angegeben, gefolgt von anderen nötigen Definitionen (Agenten mitsamt ihrer verwendeten Minds, erlebbare Ereignisse und mögliche Aktionen).

Zur Veranschaulichung, wie lt. Kap. 4.1.1 die Spielkonfiguration vonstatten geht, ist der Anfang der verwendeten Konfigurationsdatei für das Spiel *Foodchase* in Listing 1 angegeben. Darin sind

```
#Program config
game.bannerimage=resources/ARS/graphics/banner.png
game.title=Foodchase

#Visualization
visualization.cellsize=40
visualization.simulationperiod=250
visualization.bgimagepath=
visualization.showgrid=false

#Music
music.backgroundmusic=

#World
world.horizontalcells=15
world.verticalcells=15
world.layercount=2
world.layer.0.0=fffffffffffffff
world.layer.0.1=fffffffffffffff
world.layer.0.2=fffffffffffffff
world.layer.0.3=fffffffffffffff
world.layer.0.4=fffffffffffffff
world.layer.0.5=fffffffffffffff
world.layer.0.6=fffffffffffffff
world.layer.0.7=fffffffffffffff
world.layer.0.8=fffffffffffffff
world.layer.0.9=fffffffffffffff
world.layer.0.10=fffffffffffffff
world.layer.0.11=fffffffffffffff
world.layer.0.12=fffffffffffffff
world.layer.0.13=fffffffffffffff
world.layer.0.14=fffffffffffffff
world.layer.1.0=wwwwwwwwwwwwwww
world.layer.1.1=w_____________w
world.layer.1.2=w________aabb_w
world.layer.1.3=w___abc_______w
world.layer.1.4=w___wwwwwwww__w
world.layer.1.5=w___bbccwaba__w
world.layer.1.6=w_______wccc__w
world.layer.1.7=w__wwwwwabcc_w
world.layer.1.8=w_o_____w_____w
world.layer.1.9=w___a___w_____w
world.layer.1.10=w_b_aabbw_____w
world.layer.1.11=w_______w_____w
world.layer.1.12=w__p____wwww__w
world.layer.1.13=w_____________w
world.layer.1.14=wwwwwwwwwwwwwww
```

Listing 1: Anfang der Spielkonfigurationsdatei *foodchase.ini*

die grundsätzlichen Spielparameter sowie das Spielfeld definiert. Die Eigenschaften in Gruppe `visualization` geben die Größe der einzelnen Zellen des Spielfeldes (40 Pixel Seitenlänge) und die Dauer einer Simulationsperiode (250 ms) an. Auch kann angegeben werden, ob das Spielfeldgitter in Form eines roten Rasters dargestellt werden soll. Das Spielfeld ist in Eigenschaften der Gruppe `world` definiert. Es ist 15×15 Kacheln groß und verfügt über zwei Ebenen. Die darauf befindlichen Objekte sind ebenen- und zeilenweise durch Buchstaben repräsentiert. Die erste Ebene besteht zur Gänze aus Boden, was durch f gegeben ist. Die zweite ist mit Wänden, Essbarem und Agenten befüllt. Eine Auflistung der Buchstabenentsprechungen findet sich in Tab. 5.1. Weiters angegeben ist die Banner-Bilddatei und Spieltitel. Optional kann auch der Pfad auf eine Musikdatei zur endlosen Hintergrundwiedergabe angegeben werden.

Zeichen	*Bedeutung*
f	Boden
w	Mauer
a	Schnitzel
b	Karotte
c	Brokkoli
m	Adam*
n	Bodo*
o	Charlie*
p	Dorothy*

* Agentenname

Tabelle 5.1: Entsprechungen d. Buchstaben zur Konfiguration des Spielfeldes

Bei allen Versuchen mit Agenten, die eine der Benchmark-Architekturen *ESiMA* oder *OSiMA* verwenden, deren Verhalten in Kap. 5.2 erläutert wird, befinden sich zwei Agenten dieser Art im Einsatz. Sie tragen die Namen *Adam* und *Bodo* und starten immer von den Koordinaten $(4, 11)$ bzw. $(5, 13)$ aus. Die Koordinaten $(x = 1, y = 1)$ geben die Position der Kachel in der linken oberen Spielfeldecke wieder. x wird nach rechts, y nach unten weiter gezählt.

Der Agent, der für die meisten Versuche mit *NNMind* ausgestattet wurde, trägt den Namen *Charlie*. Für die in Kap. 5.3.2.10 ab S. 60 vorgestellte Versuchskonfiguration 10, bei welcher zwei NNMind-Agenten gegeneinander spielen, heißt Charlies Gegnerin *Dorothy*. Die Startparameter sind für die jeweiligen Versuchskonfigurationen immer angegeben. Weitere Informationen über die verwendeten Agenten und ihre Eigenschaften sind Kap. 5.3 zu entnehmen.

Nach der Definition des Spielfeldes erfolgt die Konfiguration verschiedener Bedingungsklassen (Gruppe `condition`), welche benötigt werden, um mit Ereignissen umgehen zu können. Darauf befinden sich in der Spielkonfiguration die Ereignisdefinitionen (Gruppe `event`). Diese umfassen die Bezeichnung, die verarbeitende Java-Klasse, Bedingungen, zu vergebende Punkte bei Eintritt des Ereignisses und Einflüsse auf den Agenten (Gesundheitsveränderung). Auch Aktionen sind als Ereignisse definiert. Einige der Ereignisse werden bei Auftreten durch Punktevergabe belohnt bzw. bestraft. Dabei handelt es sich neben der erfolgreichen Nahrungsaufnahme und Attacke auf Gegner auch um gegnerische Angriffe sowie Kollisionen mit Hindernissen und den versuchten Verzehr von nicht essbaren Objekten. Letztgenannte umfassen Hindernisse und Leere — wenn vor dem Agenten also kein Objekt vorhanden ist. Die Aufstellung der zu vergebenden Punkte für verschiedene definierte Ereignisse ist in Tab. 5.2 abzulesen.

Aktion	**Punkte**
Kollision mit Hindernis	-10
essen:	
Schnitzel, Karotte	30
Brokkoli	10
Hindernis, Leere	-10
Gegner angreifen	10
angegriffen werden	-10

Tabelle 5.2: Punktevergabe in *Foodchase*

Die Definition der Aktoren beinhaltet die Konfiguration des Körpers (`bodytype`), eventuell der Mind (`mind`) und ihrer selbst mithilfe der beiden ersten Typen (`actor`). Jedem Aktor wird im Zuge dessen ein Buchstabe zugeordnet, der lt. Tab. 5.1 zur Konfiguration des Spielfeldes zur Anwendung kommt. Ebenso werden die verwendeten Sprites (Aktorenbilder bei verschiedenen Aktionen) sowie die anfängliche Position auf der Karte angegeben. Daneben kann für bestimmte Aktoren die Evaluierung aktiviert werden, wodurch deren Daten im Statistikabschnitt des Spielfensters dargestellt werden. Es ist anzumerken, dass sämtliche auf dem Spielfeld liegenden Objekte als Aktoren angesehen werden. Unterschieden wird zwischen belebten (Agenten) und unbelebten Objekten (Boden, Mauern, Nahrung).

5.2 Funktionsbeschreibung der Benchmark-Architekturen

Zum besseren Verständnis, wie Agenten mittels der Benchmark-Architekturen *ESiMA* bzw. *OSiMA* zu Entscheidungen gelangen, sollen jene nachfolgend erklärt werden. Immer gleichbleibend ist der grundsätzliche Ablauf, den visuellen und körperlichen Zustand abzurufen, um daraus zu einer auszuführenden Aktion zu gelangen. Dabei werden immer alle anderen Agenten in der Welt als Feinde wahrgenommen. Für die Reichweite der visuellen Sensorik gelten die Angaben in Kap. 3.1.

5.2.1 Emulierte SiMA (ESiMA)

Agenten, welche mit *ESiMA* arbeiten, entscheiden über die zu wählende Aktion nach den hier genannten Regeln. Die folgenden Regeln sind nach ihrer Prüfreihenfolge sortiert. Sobald eine Bedingung zutrifft, finden die anderen Regeln in demselben Zeitabschnitt keine Beachtung mehr:

1. Gilt Panik > 0 (Attribut der Mind), führt der Agent eine Aktion aus einer vordefinierten Folge panikinduzierter Bewegungen aus. Diese Bewegungsabfolge lautet: Schritt rückwärts, Drehung links um 180° und zwei weitere Schritte vorwärts. Außedem wird der Wert von Panik um 1 verringert.

2. Ist ein Hindernis im Weg, dreht sich der Agent zufällig nach links oder rechts (45°).

3. Wird eine gegnerische Attacke wahrgenommen, hängt die Vorgangsweise unter anderem von der Gesundheit ab. Liegt sie bei weniger als 50, führt der Agent eine Bewegung der in Punkt 1 angegebenen Panikreaktion aus. Andernfalls bereitet der Agent, wenn der Gegner sich im Sichtbereich befindet, den Gegenangriff vor, falls dieser nicht bereits durchführbar ist. Ist der Gegner nicht sichtbar, plant der Agent ein Suchmuster nach Nahrung bestehend aus Drehungen und Vorwärtsbewegungen.

4. Ist die Gesundheit < 80, hängt das weitere Verhalten davon ab, ob ein Gegner und/oder Nahrung im Sichtbereich sind. Falls Nahrung **und** Gegner gesehen werden, gilt:

   - $60 \leq$ Gesundheit < 80: Panikaktionsfolge wird geplant (Panik $= 7$).
   - $50 <$ Gesundheit < 60: Angriff auf einen der sichtbaren Gegner wird versucht.

   Ist kein Gegner sichtbar, wird versucht, die nächstliegende Nahrung zu erreichen. Falls keine Nahrung vorhanden ist, aber Gegner, versucht der Agent einen Gegner anzugreifen, wenn seine Gesundheit > 50 ist. Ist die Gesundheit geringer oder keine Gegner vorhanden, führt der Agent sein Suchmuster nach Nahrung fort.

5.2.2 Optimierte SiMA (OSiMA)

Das Verhalten der Architektur *OSiMA* unterscheidet sich von *ESiMA* in einigen Punkten. Sie wurde speziell für das Spiel *Foodchase* entwickelt, um in diesem Spiel eine möglichst hohe Punktzahl zu erhalten und so lange als möglich zu überleben. Ihre Verhaltensregeln lassen sich analog zu Kap. 5.2.1 wie folgt zusammenfassen:

1. Befinden sich Gegner direkt angrenzend links oder rechts des Agenten, dreht er sich in die entsprechende zum Gegner weisende Richtung.
2. Ist der Gegner direkt vor dem Agenten, entscheidet der Zufall mit gleicher Wahrscheinlichkeit, ob angegriffen oder nichts getan werden soll.
3. Existiert Essbares im sichtbaren Umkreis von zwei Feldern, versucht der Agent es zu erreichen und zu verzehren.
4. Ansonsten führt der Agent eine zufällige Bewegung durch.

Die so ermittelte Aktion kann noch durch weitere Regeln überschrieben werden. Die nun folgenden Bedingungen, verursachen eine entsprechende Neuwahl der auszuführenden Aktion. Dauert das Erreichen der Nahrung nach Punkt 3 obiger Auflistung länger als zwei Spielzüge, verliert der Agent die Geduld und möchte geradeaus gehen. Ist dem Agenten ein Hindernis im Weg, dreht er sich nach der besten Richtung — wo kein weiteres Hindernis liegt — weg. Falls die gewählte Aktion zu diesem Zeitpunkt nicht auf Attacke lautet, wird diese mit einer Wahrscheinlichkeit von 5% durch eine zufällige Auswahl überschrieben. Dabei kann zwischen Vorwärtsbewegung, Drehung, Essen oder Attackieren gewählt werden. Essen und Attackieren können in diesem Fall zu einer Verringerung des Punktestands führen.

5.3 Versuche

Zunächst werden in Kap. 5.3.1 die Resultate der Versuche mit den Benchmark-Architekturen *ESiMA* und *OSiMA* angeführt, um Vergleichswerte für die danach präsentierten Versuchsanordnungen und ihre Ergebnisse mit der Architektur *NNMind* in Kap. 5.3.2 zu erhalten. Die Spielwelt ist dafür wie in Kap. 5.1 definiert, wobei die Konfiguration für die jeweiligen Versuchsanordnungen entsprechend leicht modifiziert sind. Welcher Natur diese Modifikationen sind, ist immer im Vorfeld der einzelnen Versuchskonfigurationen angegeben. Für einige exemplarische Spieldurchgänge ist auch die Verfolgung der zurückgelegten Wege der Agenten dargestellt. In diesen Abbildungen werden die jeweils eingestellten Startpositionen der Agenten als blassfarbene Kreise dargestellt. Die Standpositionen der Agenten während des Spiels markieren kleinere farbige Kreise, wohingegen die Verbindungslinien dazwischen die Laufstrecken der Agenten angeben. Pfeilspitzen auf den Linien geben zu erkennen, in welche Richtung und wie oft ein Agent einen bestimmten Wegabschnitt genommen hat.

Der bei den Spielen erreichte Endpunktestand stellt den wichtigsten Bemessungswert dafür dar, wie gut eine Architektur ist. Da jedes Spiel einem etwas unterschiedlichen Ablauf folgt, sind die Punktestände entsprechend stochastisch verteilt. Allerdings folgen die Benchmark-Architekturen bei der Aktionswahl stets denselben Regeln, welche in Kap. 5.2 erklärt sind. Daher fußen die

Spielendergebnisse immer auf der gleichen Grundlage. Aus diesem Grund ist die Verteilung der in den Durchläufen aufgetretenen Punktestände in Kap. 5.3.1 als Histogramm dargestellt.

Die NNMind-Agenten (Kap. 5.3.2) verändern im Gegensatz dazu im Lauf der Zeit — während und mit jedem Durchlauf — ihr Verhalten, was zu sukzessive höheren Punktesständen führen soll. Infolgedessen wurden für verschiedene Versuchskonfigurationen, bei welchen unterschiedliche Parameter des ANN und der Spielwelt verändert wurden, jeweils 10 Agenten trainiert. Das Training eines solchen Agenten heißt im folgenden *Episode*, welche immer 50 Durchläufe umfasst. Um die Verbesserung der Spielergebnisse mit der Zeit zu illustrieren, ist deren Verlauf über alle Episoden gemittelt für jeden Durchlauf dargestellt. Hierbei ist die fortlaufende Durchlaufnummer auf der Abszisse, der zugehörige mittlere Endspielstand auf der Ordinate aufgetragen. Zusätzlich vermitteln Fehlerbalken, welche die ermittelte Standardabweichung angeben, wie stark die zugrundeliegenden Ergebnisse divergieren.

Zusätzlich werden die Mittelwerte der Überlebensdauern, der Punktestände sowie der Anzahl der positiven, neutralen und negativen Aktionen tabellarisch erfasst. Auch die Überlebensstatistik — wie oft der Agent anteilsmäßig als letzter Überlebender übrig geblieben ist — ist prozentual in den Tabellen angegeben.

5.3.1 Benchmark-Architekturen

Wie bereits erwähnt, wurden für die Benchmark-Versuche zwei verschiedene kognitivistische Architekturen verwendet: *ESiMA* und *OSiMA*. Jeweils 50 Durchgänge wurden mit zwei Agenten namens *Adam* und *Bodo* gespielt, welche von den Positionen $(4, 11)$ bzw. $(5, 13)$ des Spielfeldes starten, wobei die Kachel in der linken oberen Ecke die Koordinaten $(1, 1)$ besitzt. Beide Agenten sind zu Spielbeginn stets nach rechts ausgerichtet. Abb. 5.1 zeichnet einen beispielhaften Spielverlauf anhand eines Durchganges mit der OSiMA-Architektur nach.

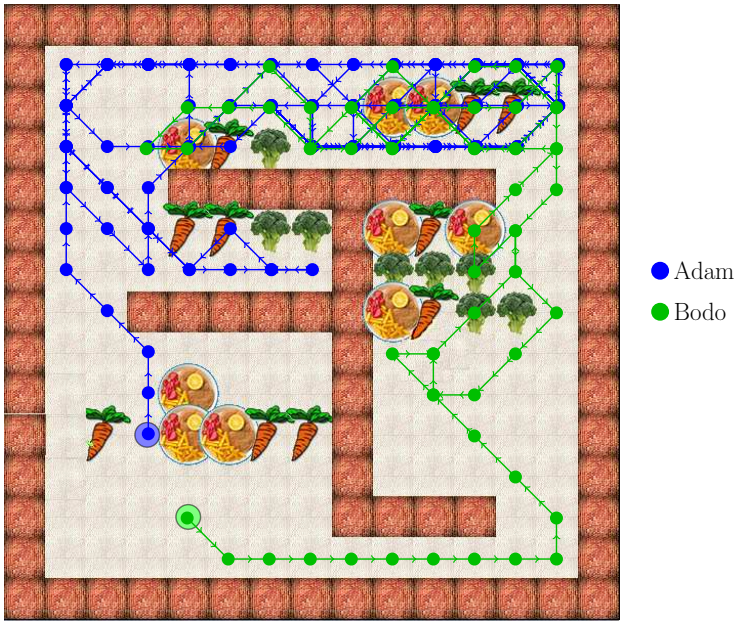
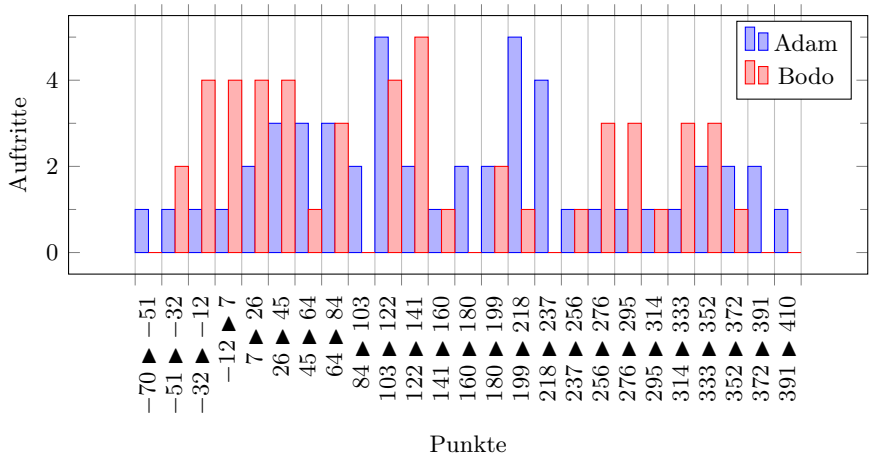


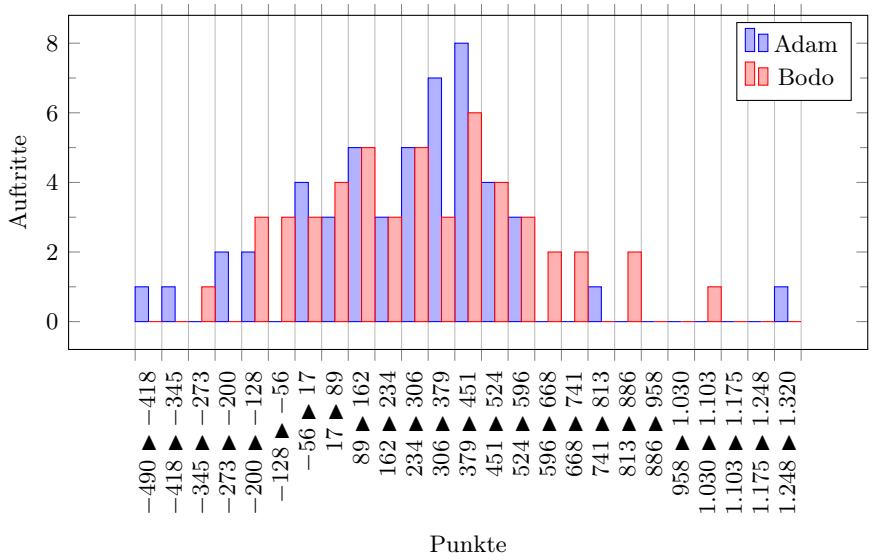
Abbildung 5.1: Spielfeld mit Startpositionen von Adam und Bodo

Anhand der Histogramme der Punktestände bei Spielende in Abb. 5.2 lässt sich die Häufigkeitsverteilung der erreichten Endpunktestände von Adam und Bodo unter Verwendung von jeweils

ESiMA bzw. OSiMA ablesen. Auf der Abszisse sind jeweils die Endpunktestände, in 25 Intervalle unterteilt, vom jeweils aufgetretenen Minimal- bis Maximalwert aufgetragen.



(a) ESiMA



(b) OSiMA

Abbildung 5.2: Histogramme der Benchmark-Agenten

Tab. 5.3 zeigt die Mittelwerte der Spieldauer, Punktestände, sowie der Anzahlen der positiven (*+*), negativen (-) und neutralen (*0*) Spielzüge bzw. Aktionen. Außerdem enthält sie in Prozent den Anteil der gewonnenen Spiele, d. h. den Anteil jener Spiele, in denen der jeweilige Agent den anderen überlebt hat. Generell zeigt sich, dass die OSiMA-Agenten wesentlich länger überleben, wobei dies sich nicht durch ein erhöhtes Auftreten von positiven Ereignissen erklären lässt. Deren Anzahl ist im Schnitt zwar etwa doppelt so hoch und wird von einer annähernd ebenso großen Erhöhung der negativen Ereignisanzahl begleitet. Vielmehr halten sie mit der zur Verfügung stehenden Nahrung besser haus und verhalten sich nicht so aggressiv wie die ESiMA-Agenten. Dies führt zu einem überproportionalen Anstieg der neutralen Aktionen, zu denen Bewegungen und Untätigkeit zählen. Mit der höheren Anzahl an positiven Ereignissen im OSiMA-Fall gehen höhere erreichte Punktestände einher, obwohl auch die Anzahl der negativen Ereignisse in ähnlichem Maß steigt. Dies lässt sich anhand der Angaben in Tab. 5.2 erklären, da die Strafpunkte für negative

Ereignisse oft geringer sind als die Belohnungen für positive.

		Mittelwerte					*über-*
Arch.	*Agent*	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
ESiMA	Adam	186,6	165,0	10,4	6,0	162,2	68%
	Bodo	186,6	135,0	9,5	7,8	162,2	32%
OSiMA	Adam	1.245,0	245,2	18,6	12,3	1.205,3	44%
	Bodo	1.245,0	289,7	23,1	10,1	1.200,6	56%

Tabelle 5.3: Versuchsstatistiken der Benchmark-Agenten

Interessant ist der Umstand, dass beim Test der ESiMA-Architektur *Adam* deutlich über *Bodo* dominiert hat, indem er mehr als zwei Drittel der Spiele für sich entscheiden konnte, wohingegen bei der OSiMA-Architektur sich Gegenteiliges erkennen lässt. Der Grund hierfür liegt zum einen an den Ausgangspositionen der Agenten, zum anderen an ihrem zutage tretenden Verhalten. Mittels ESiMA sind die Agenten relativ aggressiv und Adam befindet sich bei Spielstart in einer strategisch günstigeren Position. Somit ist auch der Fall ziemlich häufig zu beobachten, dass Bodo von Adam zu Tode attackiert wird. Bei der OSiMA-Architektur hingegen lässt sich wesentlich geringere Aggression beobachten. Die durchschnittlich höhere Punktzahl Bodos lässt sich allerdings dadurch erklären, dass Bodo relativ häufig den schmalen Gang am unteren Rand des Spielfeldes durchwandert (vgl. Abb. 5.1) und das dahinter liegende große Nahrungsfeld üblicherweise ungestört bearbeiten kann.

Es bleibt noch zu klären, wodurch die Zahl der negativen Ereignisse zustande kommt. Einerseits wird das Einstecken von Attacken als negatives Ereignis gewertet, andererseits neigen die Agenten dazu, bei Nahrungsmangel in der Umgebung panisch zu werden und ins Leere zu beißen, was bestraft wird. Obwohl die OSiMA-Agenten weniger aggressiv sind, greifen sie dennoch bei sich ergebender Gelegenheit an.

5.3.2 NNMind

Für die Evaluierung der kognitiven Architektur *NNMind* wurden insgesamt 10 verschiedene Versuchskonfigurationen aufgestellt, die im folgenden detailliert beschrieben werden. Zu jeder Konfiguration wurden je 10 Agenten-Exemplare trainiert und 50 Spieldurchläufe im Normalbetrieb durchgeführt. Das Training eines Agenten wird im folgenden als Episode bezeichnet. Die Anzahl von 50 Spielen je trainiertem Agenten genügt, um die Lernkurve darzustellen. Diese Anzahl wurde gewählt, da sich gezeigt hat, dass sich in den meisten Versuchskonfigurationen nach etwa 40 bis 50 Durchläufen der mittlere Punktestand asymptotisch einem Maximum nähert. Dies wird die Präsentation der Ergebnisse zu den jeweiligen Versuchen veranschaulichen.

Alle folgenden Versuchskonfigurationen verwenden zumindest einen Agenten mit NNMind-Architektur. Mit Ausnahme der Konfiguration 10, welche zwei NNMind-Agenten — *Charlie* und *Dorothy* — gegeneinander antreten lässt, dienen überall die aus Kap. 5.3.1 bereits bekannten Benchmark-Agenten *Adam* und *Bodo* als Gegner. Die verschiedenen Versuchskonfigurationen dienen der Untersuchung bestimmter Fragestellungen. Tab. 5.4 gibt die veränderlichen Spielparameter in den unterschiedlichen Versuchskonfigurationen an. Konfigurationen 1–4 haben den Sinn, die ANN-Einstellungen herauszufinden, die die besten Ergebnisse liefern. Konfigurationen 5 und 6 setzen den Agenten Charlie in direkte Konkurrenz zu den Benchmark-Agenten. Mit Konfiguration 7 wird der Einfluss des Hungerneurons auf die Leistung des Agenten geprüft. Die Vorprägung, die der nächste Absatz behandelt, wird in Konfiguration 8 nicht durchgeführt. Dies dient der Testung,

wie gut die Architektur aus einem rein zufälligen Zustand des ANN lernen kann. Konfiguration 9 behandelt die Leistungsfähigkeit von NNMind mit verringerter Eingangsneuronenzahl. Schließlich lässt Konfiguration 10, wie bereits angedeutet, zwei NNMind-Agenten gegeneinander spielen.

Parameter	*Konfiguration*				
	1	2	3	4	5
ANN-Eigenschaften					
Lernratenfaktor C	0,002	0,001	0,001	0,001	0,001
Schichtenkonfig.	88–44–1	88–44–1	88–88–44–1	88–44–22–1	88–44–1
Hungerneuron	ja	ja	ja	ja	ja
Vorprägung	ja	ja	ja	ja	ja
Benchmark-Agenten					
Adam	(4, 11)	(4, 11)	(4, 11)	(4, 11)	(4, 11)
Bodo	(5, 13)	(5, 13)	(5, 13)	(5, 13)	(5, 13)
Attacken aktiv	nein	nein	nein	nein	ja
Mind	OSiMA	OSiMA	OSiMA	OSiMA	OSiMA
NNMind-Agenten					
Charlie	(8, 3)	(8, 3)	(8, 3)	(8, 3)	(3, 10)
Dorothy	—	—	—	—	—
Mind	NNMind2	NNMind2	NNMind2	NNMind2	NNMind2
	6	7	8	9	10
ANN-Eigenschaften					
Lernratenfaktor C	0,001	0,001	0,001	0,001	0,001
Schichtenkonfig.	88–44–1	88–44–1	88–44–1	61–30–1	88–44–1
Hungerneuron	ja	nein	ja	ja	ja
Vorprägung	ja	ja	nein	ja	ja
Benchmark-Agenten					
Adam	(4, 11)	(4, 11)	(4, 11)	(4, 11)	—
Bodo	(5, 13)	(5, 13)	(5, 13)	(5, 13)	—
Attacken aktiv	ja	nein	nein	ja	—
Mind	ESiMA	OSiMA	OSiMA	ESiMA	—
NNMind-Agenten					
Charlie	(3, 10)	(8, 3)	(8, 3)	(3, 10)	(3, 9)
Dorothy	—	—	—	—	(4, 13)
Mind	NNMind2	NNMind2	NNMind2	NNMind3	NNMind2

Tabelle 5.4: Zusammenfassung der Parameter der Versuchskonfigurationen

Die Vorprägungsphase funktioniert so, dass der Agent seine Aktionen allesamt rein zufällig auswählt und aus den so gemachten Erfahrungen lernt. Es handelt sich also um einen reinen Explorationslauf. Die dafür benützte Lernrate ist mit einem Vorfaktor $C = 0{,}003$ (siehe Glg. 3.2 auf S. 27) höher als die im Normalbetrieb verwendete. Da das Verhalten des Agenten in der Vorprägungsphase rein stochastisch ist, sind die Ergebnisse davon nicht aussagekräftig und werden daher nicht weiter erwähnt. Wird Vorprägung verwendet, um das ANN initial zu trainieren, geschieht dies jeweils in 10 Durchläufen.

5.3.2.1 Konfiguration 1: ANN-Test mit hoher Lernrate

Die ersten vier Konfigurationen haben den Sinn, auszutesten, welche Parameter für das ANN sinnvoll sind, um möglichst gute Ergebnisse zu erzielen. Aus diesem Grund wurden für diese Versuchskonfigurationen die Attacken der Gegner deaktiviert. Außerdem liegt Charlies Startposition mit den Koordinaten $(8, 3)$ weit entfernt von seinen Gegnern. Dies soll sicherstellen, dass die Störung durch Adam und Bodo so gering als möglich ausfällt, auch wenn sie nicht ausgeschlossen

werden kann. Diese Vorgehensweise wurde deshalb gewählt, weil per Definition das Spiel endet, sobald nur ein Agent noch am Leben ist. Diese Option wurde gewählt, um eine Umprogrammierung des Miklas-Codes zu vermeiden. Die Gegner Adam und Bodo nutzen OSiMA.

Das Sensorenwort besteht, wie in Kap. 3.2 beschrieben, aus 3 binären Neuronen je Spielfeldkachel im Sichtbereich des Agenten und dem Hunger-Neuron. Dies führt zu einer Sensorwortlänge von $27 \times 3 + 1 = 82$ Neuronen. Die Eingangsschicht enthält zudem 6 Neuronen für die ausführbaren Aktionen. Die gewählte Schichtkonfiguration des BP-Netzes lautet 88–44–1, also 88 Neuronen in der Eingangs-, 44 in der versteckten und 1 Neuron in der Ausgangsschicht. Der Vorfaktor zur Lernrate im Explorationsmodus wurde $C = 0{,}002$ gewählt, und beträgt damit $^2/_3$ des Vorfaktors im Vorprägemodus.

Tab. 5.5 zeigt die Mittelwerte der Spieldauer des Agenten, der Endpunktezahl, sowie der Anzahlen der positiven, negativen und neutralen Aktionen. Auch der Anteil der Spiele ist angegeben, in denen Charlie seine Gegner überlebt hat. Die Statistiken sind über jeweils 10 aufeinanderfolgende Durchläufe und die 10 Episoden gemittelt. Hier lässt sich ablesen, dass Charlie die erreichte Punktzahl von durchschnittlich 168,8 auf 710,8 steigern konnte. Dabei konnte er die Anzahl der Male, wo er belohnt wurde, im Durchschnitt stark erhöhen, wohingegen die Zahl der Bestrafungen tendenziell gesunken ist. Die Lebensdauer ist zunächst im Schnitt gesunken und hat sich dann eingependelt. Dies steht in Zusammenhang mit vermehrter Untätigkeit bzw. ziellosem Umherlaufen. Abb. 5.3 stellt den Verlauf der mittleren erreichten Punktzahlen für jeden Durchlauf der Episoden dar. Der Graph gibt den Mittelwert und die ermittelte Standardabweichung als Fehlerbalken an.

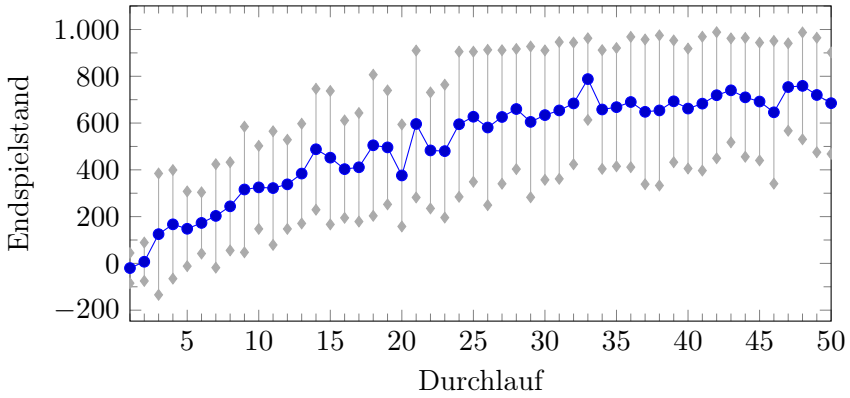


Abbildung 5.3: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 1

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	194,3	168,8	11,5	12,6	167,3	2%
11–20	134,7	417,5	21,0	10,9	100,9	1%
21–30	143,3	588,7	29,2	9,9	102,1	1%
31–40	157,1	679,9	32,9	8,9	112,5	1%
41–50	153,5	710,8	34,7	9,4	106,8	1%

Tabelle 5.5: Statistiken Konfiguration 1

An den Statistiken ist auch ablesbar, dass Charlie seine Gegner sehr selten überlebt. Dies lässt sich durch den Vergleich der durchschnittlichen Lebensdauer von Charlie gegenüber jener der OSiMA-Agenten erklären. Die Angabe von 1% entspricht wegen 100 zugrundeliegenden Datenpunkten für 10 Durchläufe und 10 Episoden einem Mal.

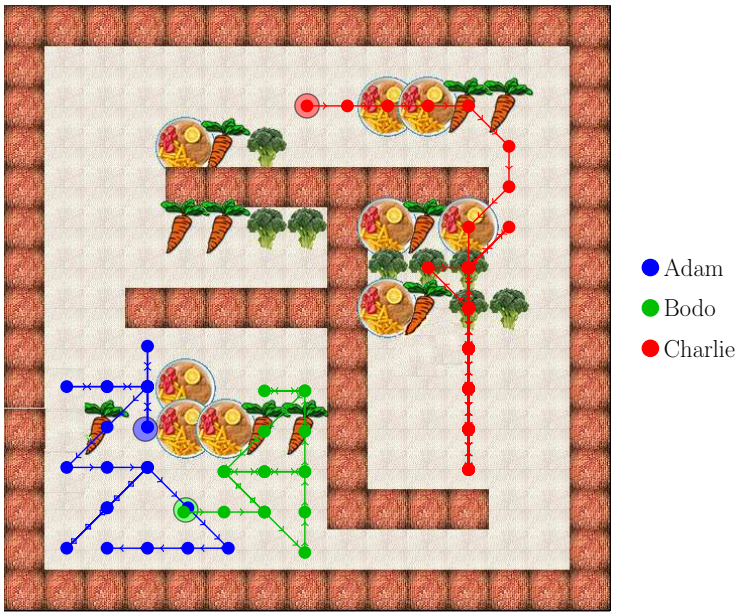


Abbildung 5.4: Darstellung eines Bewegungspfades der Agenten in der Spielwelt (Konfiguration 1, Episode 2, Durchlauf 35)

Abb. 5.4 zeigt beispielhaft einen Spieldurchlauf. Es handelt sich dabei um den 35. Durchlauf der 2. Episode dieser Versuchskonfiguration. Es ist dazu anzumerken, dass Charlie sämtliche Nahrung in der von ihm bearbeiteten Gegend vertilgt hat. Dazu ist es nicht notwendig, die zugehörigen Felder tatsächlich zu befahren. Es genügt, sich auf eine Kachel mit benachbarter Nahrung zu setzen, und ringsum alles abzufressen. Der Abstecher nach unten ist der nachherigen Unwissenheit das weitere Vorgehen betreffend geschuldet. Es ist gut sichtbar, dass im Falle dieses Spieldurchlaufes Charlie von seinen Gegnern abgesondert geblieben ist, da sich diese während des gesamten Spielverlaufs im linken unteren Spielfeldbereich aufgehalten haben.

Es ist bei den Tests mit dieser Konfiguration das Problem aufgetreten, dass oftmals die Anpassung der ANN-Gewichte bereits etabliertes zielführendes Verhalten überschreibt. Dies hat häufig einen Einbruch des Endpunktestands für die nächsten Durchläufe zur Folge. Eventuell stellt sich das verlorene Verhalten in solch einem Fall nach einigen Durchläufen wieder ein. Dies führt zur Annahme, dass die Lernrate zu hoch gewählt war.

5.3.2.2 Konfiguration 2: ANN-Test mit geringerer Lernrate

Diese Versuchskonfiguration stimmt mit Konfiguration 1 überein, mit Ausnahme des Lernraten-Vorfaktors, welcher auf $C = 0{,}001$ halbiert ist. Die Versuchsstatistiken in Tab. 5.6 bzw. Abb. 5.5 sind der Größenordnung nach vergleichbar. Bei Durchsicht der Ergebnisse der einzelnen Durchläufe zeigt sich, dass das Lernen von Verhaltensweisen länger dauert, diese allerdings stabiler im ANN bleiben. Zwar kann es bei neuen Situationen auch weiterhin vorkommen, dass in der Policy bisher Gelerntes überlagert wird. Jedoch wird der Schaden in der Regel in den folgenden Durchläufen schneller wieder behoben.

Abb. 5.6 zeigt als Beispiel einen für Charlie besonders erfolgreichen Spieldurchgang. Der Agent hat von seiner Startposition sich nach rechts bewegend sämtliche auffindbare Nahrung gefressen.

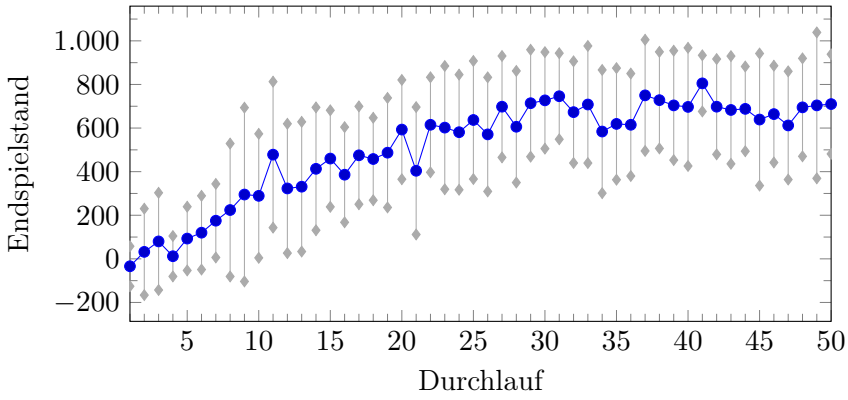


Abbildung 5.5: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 2

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	162,6	128,6	10,1	13,2	136,8	2%
11–20	146,3	440,4	22,4	12,0	109,7	0%
21–30	153,6	615,5	29,9	10,5	111,2	2%
31–40	159,8	682,4	33,3	10,1	114,2	1%
41–50	185,9	689,8	33,5	9,9	139,9	2%

Tabelle 5.6: Statistiken Konfiguration 2

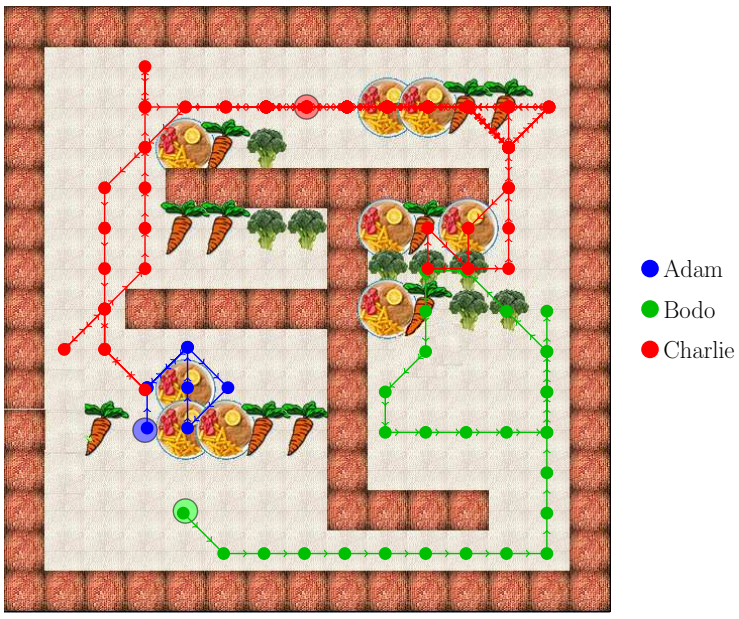


Abbildung 5.6: Darstellung eines Bewegungspfades der Agenten in der Spielwelt (Konfiguration 2, Episode 10, Durchlauf 49)

Danach ist er nach rechts in das Nahrungsdepot darunter abgebogen, das auch von Bodo aufgesucht worden ist, wie an den grünen Spuren zu sehen. Anschließend ist es Charlie gelungen, den Weg zurück nach oben zu finden, ist dem Pfad nach links gefolgt, hat die hier gefundene restliche Nahrung zu sich genommen, und ist dann am linken Rand nach unten abgebogen und schließlich Adam begegnet. Dieser außergewöhnlich erfolgreiche Durchlauf sticht mit einem erreichten

Spielstand von 1250 Punkten heraus. Die Ursache hierfür ist eine Reihe glücklicher Zufälle bei der Exploration. Generell haben die Agenten in so gut wie allen Episoden mehr oder weniger den Weg nach rechts in das große rechte Nahrungsfeld gefunden, und gelernt, die auf dem Weg gefundene Nahrung abzuweiden.

Die Probleme fangen an, sobald in der unmittelbaren Umgebung keine Nahrung mehr aufzufinden ist, noch weitere Weidegründe bzw. den Weg dorthin zu finden. Besonders schwierig erweist sich in diesem Zusammenhang der enge Gang am unteren Spielfeldrand, da hier zum Erreichen und Durchqueren mehr als die 10 Schritte vonnöten sind, welche im Kurzzeitgedächtnis Platz finden. Auch ist die Gefahr sehr hoch, dass der Agent hier gegen eine Wand läuft, und somit eine Bestrafung und damit den Lernvorgang aus dieser Erfahrung auslöst. Damit gehen aber nach dem hier verwendeten Design die Daten der bisher gesammelten Schritte wieder verloren.

5.3.2.3 Konfiguration 3: ANN mit 2 versteckten Schichten

Ziel dieser Versuchsreihe ist, auszutesten, ob die Erweiterung des ANN um eine weitere versteckte Schicht sich vorteilhaft auswirkt. Daher lautet die ANN-Konfiguration hier 88–88–44–1, wohingegen der Rest gleich wie Konfiguration 2 ist. Die in Tab. 5.7 und Abb. 5.7 dargelegten Statistiken der Versuchsresultate zeigen, dass die Architektur mit dieser ANN-Konfiguration deutlich geringere mittlere Spielstände erreicht.

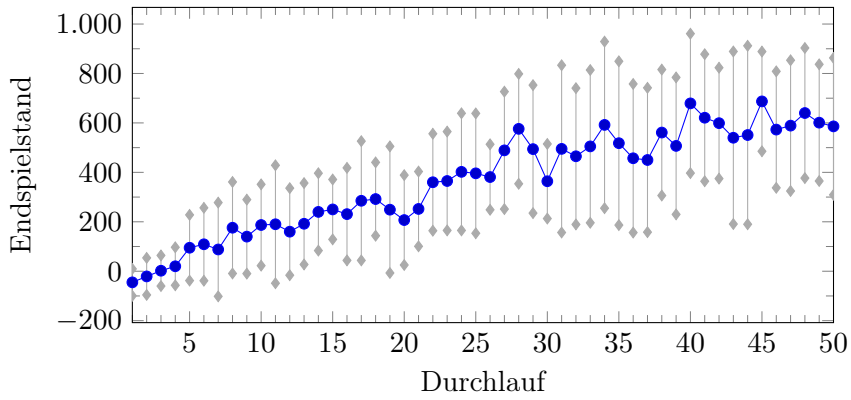


Abbildung 5.7: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 3

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	165,2	75,1	7,1	11,8	143,9	3%
11–20	118,2	229,6	12,4	11,6	92,5	1%
21–30	149,1	407,9	20,4	11,1	115,5	4%
31–40	153,1	522,9	26,1	10,9	113,8	0%
41–50	170,9	598,7	29,3	9,9	129,1	1%

Tabelle 5.7: Statistiken Konfiguration 3

Die Anzahl der negativen Ereignisse ist zwar im Schnitt im Wertebereich der vorigen beiden Versuchskonfigurationen, jedoch ist die durchschnittliche Anzahl der erhaltenen Belohnungen deutlich geringer, was sich auch im geringeren mittleren Punktestand abzeichnet. Die Kurve aus Abb. 5.7 folgt allerdings einem steigenden Trend. Das legt den Schluss nahe, dass das Netz zu viele Zustände abbilden kann, und daher vergleichsweise länger zum Lernen braucht. Daraus folgen die Versuche

der folgenden Konfiguration 4, welche mit kleineren versteckten Schichten arbeitet. Anzumerken ist, dass die Berechnungen, insbesondere der Lernvorgang, mit diesen ANN-Einstellungen deutlich rechenintensiver ist, wodurch sich zwischen den Spielzügen mitunter lange Wartezeiten einstellen.

5.3.2.4 Konfiguration 4: Verkleinerung der 2 versteckten Schichten

Diese Versuchskonfiguration ist eine Neuauflage der Konfiguration 3 mit der Änderung des ANN auf die Schichtengrößen 88–44–22–1. Wie Tab. 5.8 und Abb. 5.8 zeigen, ist die Leistung der damit trainierten Agenten etwas höher als in Konfiguration 3, bringt allerdings keinen Vorteil gegenüber Netzen mit nur einer versteckten Schicht. Damit disqualifizieren sich Netze mit mehr als einer versteckten Schicht.

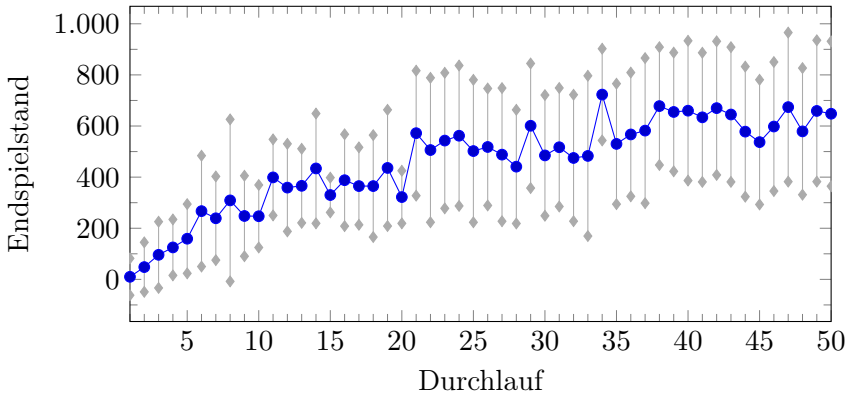


Abbildung 5.8: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 4

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	-	*0*	*lebt*
1–10	165,4	174,8	10,7	11,7	140,6	3%
11–20	119,4	376,4	18,3	10,4	89,0	1%
21–30	142,2	521,8	25,4	10,6	104,0	1%
31–40	159,9	587,0	29,2	11,0	117,3	1%
41–50	153,8	622,2	29,6	8,8	113,3	1%

Tabelle 5.8: Statistiken Konfiguration 4

5.3.2.5 Konfiguration 5: NNMind gegen OSiMA

Die unter dieser Konfiguration durchgeführten Versuche haben den Sinn, Charlie seinen Gegnern Adam und Bodo, die beide OSiMA nutzen, unmittelbar auszusetzen. Zu diesem Zweck ist Charlies Startposition auf $(3, 10)$, in die Nähe seiner Gegner, gesetzt. Deren Attacken sind hier aktiviert. Ansonsten ist die Versuchskonfiguration gleich jener der Konfiguration 2, deren ANN-Parameter für die folgenden Versuchskonfigurationen als Ausgangspunkt dienen.

Die mittleren erreichten Punkte sind, wie an Tab. 5.9 und Abb. 5.9 abzulesen, geringer als in den Konfigurationen 1 und 2, da der Agent hier mehr Konkurrenz hat. Dennoch ist die mittlere Punktzahl in den späten Durchläufen höher im Vergleich zu den Ergebnissen der Benchmark-Agenten (vgl. Kap. 5.3.1). Auch ein Anstieg der Überlebenschance von Charlie lässt sich in den späteren Durchläufen beobachten.

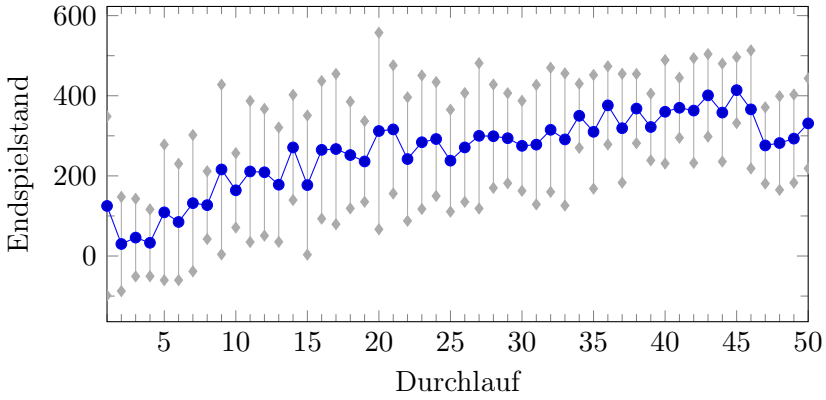


Abbildung 5.9: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 5

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	117,9	106,7	11,7	11,5	94,2	3%
11–20	104,1	237,8	15,3	10,9	77,4	3%
21–30	102,5	281,1	14,4	11,1	76,6	4%
31–40	105,6	328,9	14,9	10,1	80,1	8%
41–50	100,1	345,4	15,9	10,3	74,0	7%

Tabelle 5.9: Statistiken Konfiguration 5

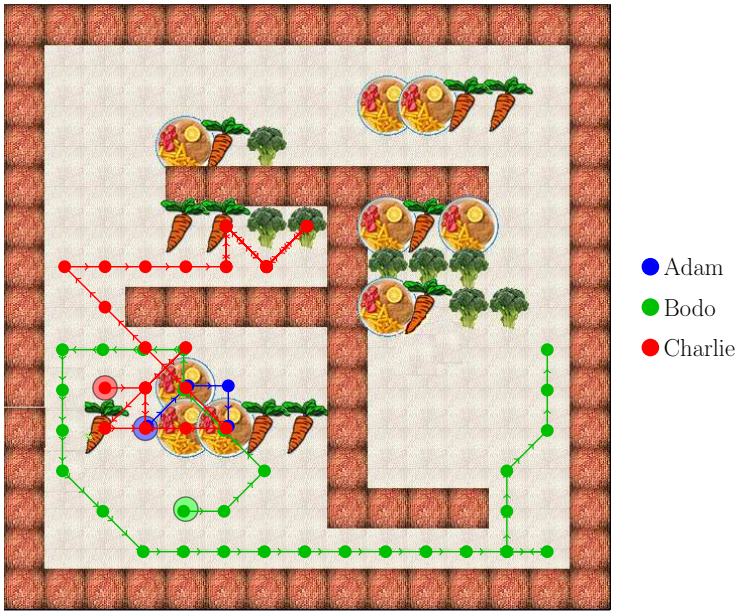


Abbildung 5.10: Darstellung eines Bewegungspfades der Agenten in der Spielwelt (Konfiguration 5, Episode 4, Durchlauf 45)

Eine interessante Beobachtung betrifft das Thema, wie Charlie mit Angriffen auf seine Gegner (und umgekehrt) umgeht. Durch Verfolgen der einzelnen Spieldurchläufe lässt sich hier eine Tendenz erkennen. Wenn nämlich Charlie anfangs lernt, dass der Angriff auf Gegner eine Belohnung einbringt, worauf sein Verhalten manchmal einen aggressiven Ton erhält, verliert sich diese Affinität zum Angriff häufig in späteren Durchläufen zugunsten der Suche nach Nahrung. Dies lässt

sich damit erklären, dass Attacken eine riskante Sache sind, da die Gegner sich üblicherweise zur Wehr setzen, wodurch die Belohnung für den Angriff und die Bestrafung fürs Angegriffenwerden einander aufheben. Auch sonst führen Angriffe der Gegner auf Charlie zu einer höheren Wahrscheinlichkeit, dass er sie zu meiden versuchen wird.

Abb. 5.10 gibt beispielhaft die Bewegungspfade der Agenten aus dem 45. Durchlauf der Episode 4 wieder. Hier zu sehen ist, dass Charlie zunächst die Nahrung in seinem Anfangsumkreis — seine Startposition liegt direkt über der separat liegenden Karotte im linken unteren Spielfeldbereich — frisst, und damit Adam zum Positionswechsel zwingt. Die weiteren Schritte waren das weitere Abarbeiten, soweit möglich, der lokalen Nahrungsansammlung, worauf, als ein Versuch zu fressen fehlging, der Agent rückwärts nach links oben gewandert ist, bis er an der Wand angestoßen ist. Nach einem weiteren Fressfehlversuch war die zweite Option eine Drehung nach links, worauf Charlie mit der Situation wieder zurechtkam, und geradewegs auf die aufgereihte Nahrung zugesteuert hat.

5.3.2.6 Konfiguration 6: NNMind gegen ESiMA

Diese Versuche verlaufen analog zu Konfiguration 5 mit dem Unterschied, dass die Gegner die ESiMA-Architektur verwenden. Die Ergebnisse in Tab. 5.10 und Abb. 5.11 zeigen eine vergleichbare Leistung zu den Versuchen mit der OSiMA-Architektur. Was hier auffällt, ist die überdurchschnittliche Zahl von Malen, an denen Charlie seine Gegner überlebt hat.

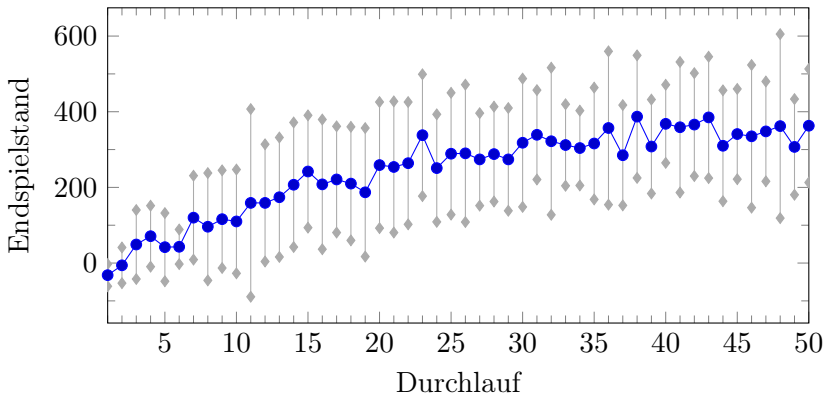


Abbildung 5.11: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 6

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	87,9	60,9	7,6	13,0	69,0	3%
11–20	78,0	202,6	12,6	11,3	55,6	15%
21–30	80,3	284,0	15,7	10,1	55,7	25%
31–40	76,2	329,8	17,4	9,5	50,7	28%
41–50	85,8	347,6	18,4	9,3	59,4	32%

Tabelle 5.10: Statistiken Konfiguration 6

Dies lässt sich hauptsächlich auf den Fall der 10. Episode zurückführen, bei welcher Charlie ab dem 14. Durchlauf eine sehr effiziente Art gefunden hat, beide Gegner schnell durch unablässige Attacken zu töten. Da all dies sich in bloß 23 Spielzügen abspielt, ändert sich diese Policy auch

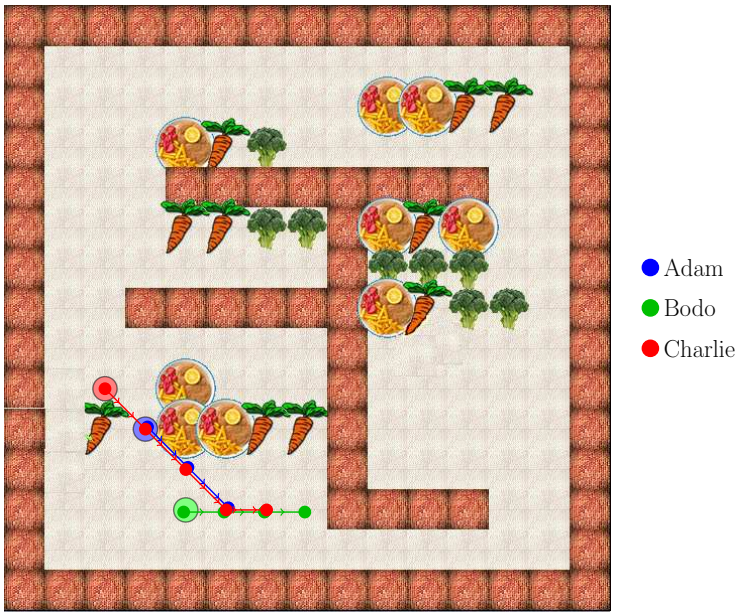


Abbildung 5.12: Darstellung eines Bewegungspfades der Agenten in der Spielwelt (Konfiguration 6, Episode 10, Durchlauf 14)

nicht mehr. Der Grund hierfür ist, dass durch die ε-Gier der kognitiven Architektur ein Explorationsschritt alle 25 Zeitschritte eingefügt wird. Dies geht sich also knapp nicht aus, weswegen das Verhalten in diesem Fall somit statisch bleibt. Die dazugehörige Pfadverfolgung der Agenten kann Abb. 5.12 entnommen werden.

5.3.2.7 Konfiguration 7: Hunger ignoriert

Das in den vorangegangenen Versuchen beobachtete Verhalten des Agenten lässt vermuten, dass das Hungerneuron auf das Verhalten des Agenten keinen großen Einfluss hat. Diese Versuchskonfiguration beschäftigt sich daher mit der Frage, ob Entfernung der Hungerinformation aus dem ANN zu einer Leistungsverbesserung führt. Zu diesem Zweck wurde die Konfiguration 2 so modifiziert, dass das bisherige Hungerneuron konstant auf den Wert 1 gesetzt ist. Damit wirkt dieses wie ein Bias-Neuron (vgl. Kap. 2.2.4).

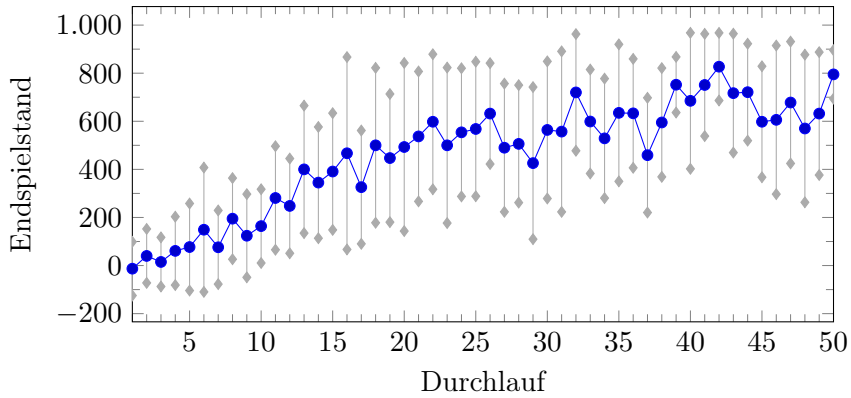


Abbildung 5.13: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 7

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	149,8	88,8	7,7	11,6	128,5	1%
11–20	138,9	389,8	19,7	10,0	106,9	1%
21–30	141,8	537,5	26,6	10,2	103,0	2%
31–40	148,0	616,4	29,6	9,1	107,1	1%
41–50	157,7	689,5	33,6	9,8	112,1	0%

Tabelle 5.11: Statistiken Konfiguration 7

Die Ergebnisse, angegeben in Tab. 5.11 und Abb. 5.13, sind vergleichbar mit jenen aus Versuchskonfiguration 2. Ein möglicher Einfluss des Hungerneurons auf die Leistung des Agenten ist somit nicht beobachtbar.

5.3.2.8 Konfiguration 8: Vorprägephase ausgelassen

Diese Versuchsreihe geht der Frage nach, ob die Vorprägephase ausgelassen werden kann. Bis auf die fehlende Vorprägung entsprechen die Parameter Konfiguration 2. Die dabei gewonnenen Ergebnisse sind Tab. 5.12 und Abb. 5.14 zu entnehmen.

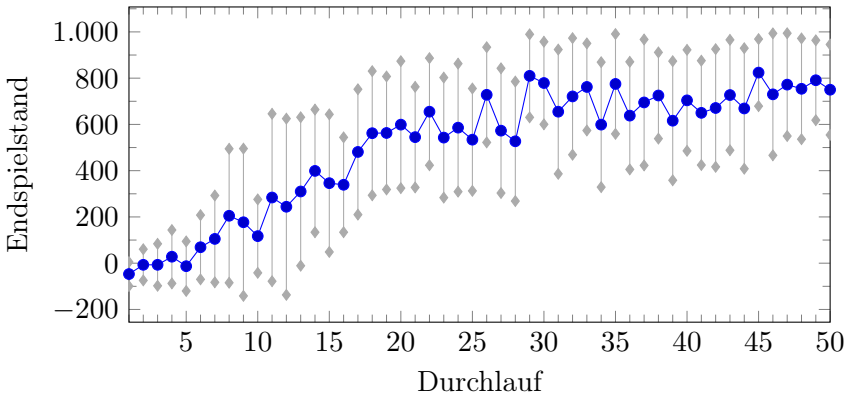


Abbildung 5.14: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 8

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	161,8	62,7	6,3	10,2	143,0	1%
11–20	172,7	412,7	21,8	11,8	136,9	2%
21–30	158,0	628,0	30,7	10,7	114,5	0%
31–40	164,1	689,0	34,1	11,0	116,2	2%
41–50	158,3	733,8	35,3	9,1	111,3	0%

Tabelle 5.12: Statistiken Konfiguration 8

Die Entwicklung der mittleren Punktestände stellt sich ähnlich wie bei Konfiguration 2 dar. Der Lernerfolg ist hier zwischen den Episoden allerdings stärker verschieden. So unterscheidet sich die zur Erreichung vergleichbarer Ergebnisse benötigte Anzahl der Durchläufe zwischen den Episoden teils signifikant, da die Agenten der einzelnen Episoden von stark divergierenden Startpolicies ausgehen.

5.3.2.9 Konfiguration 9: Kodierung der Sensordaten

Das Sensorwort hat eine unökonomische Länge. Es sind je Kachel im Sichtbereich 3 Neuronen für die drei Kategorien von Gegenständen (Hindernis, Essbares, Gegner) reserviert. Daher stellt sich die Frage, ob die Sensorwortlänge durch Kodierung verkürzt werden kann, und der Agent damit dennoch vergleichbare Ergebnisse erzielt wie sonst. Zu diesem Zweck steht die Version 3 von NNMind (vgl. Kap. 3.2.3) zur Verfügung, welche Charlie hier verwendet.

Die Definition des Sensorworts ähnelt dabei dem üblichen Aufbau, was die Indexierung des Sichtbereichs anbelangt. Zur Darstellung eines Objekts an einer bestimmten Position stehen allerdings nur noch 2 Neuronen zur Verfügung. Damit hat das Sensorwort eine Gesamtlänge von $27 \times 2 + 1 = 55$ Neuronen. Die ANN-Schichten wurden daher mit 61–30–1 Neuronen Länge gewählt.

Bis auf die Verwendung von NNMind3 für Charlie entspricht die Versuchsanordnung der Konfiguration 6. Die Benchmark-Agenten nutzen OSiMA. Tab. 5.13 und Abb. 5.15 geben die Ergebnisse dieser Versuche wieder.

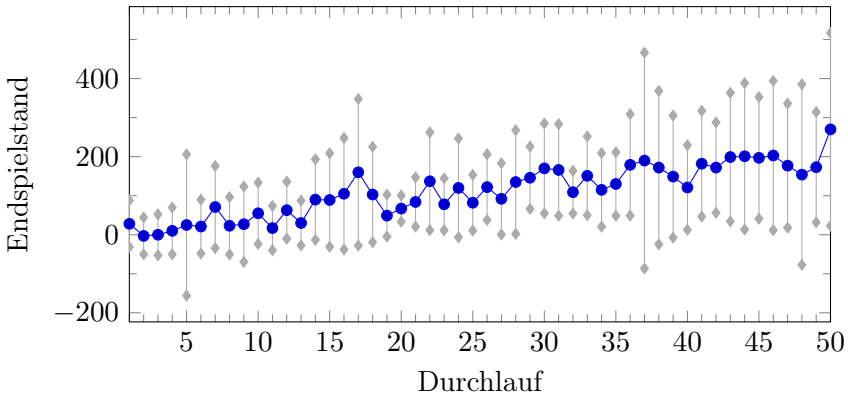


Abbildung 5.15: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 9

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	75,6	25,7	7,8	11,5	58,1	2%
11–20	65,7	77,3	12,1	9,8	46,6	12%
21–30	58,7	116,6	14,1	8,7	39,1	27%
31–40	64,4	148,2	15,6	9,1	43,0	28%
41–50	64,4	192,8	17,3	8,5	41,7	32%

Tabelle 5.13: Statistiken Konfiguration 9

Die Ergebnisse zeigen im Vergleich zu denen aus Konfiguration 6 eine wesentlich geringere mittlere Punktezahl. Dies lässt sich auf eine Tendenz zu angriffsreichen Strategien zurückführen. Die Anzahl der Male, die Charlie seine Gegner überlebt hat, ist wie auch bei Konfiguration 6 sehr hoch. Dies liegt wie auch dort daran, dass der Agent in sogar zwei Episoden gelernt hat, die Gegenspieler schnell und effizient zu töten, wie bei Konfiguration 6 bereits geschehen und erläuert.

5.3.2.10 Konfiguration 10: NNMind gegen NNMind

Es soll noch die Frage beantwortet werden, wie ein Agent mit der NNMind-Architektur auf einen gleichen Gegner reagiert. Charlies Gegnerin wird im folgenden *Dorothy* genannt. Das ANN von

beiden Agenten ist wie in Konfiguration 2 eingestellt. Die Benchmark-Agenten Adam und Bodo sind für diesen Versuch aus dem Spiel entfernt. Die Startpositionen lauten $(3, 9)$ für Charlie und $(4, 13)$ für Dorothy. Sie sind im Vergleich zu jenen der Benchmark-Agenten versetzt, um einen unfairen Vorteil für einen der Agenten zu vermeiden.

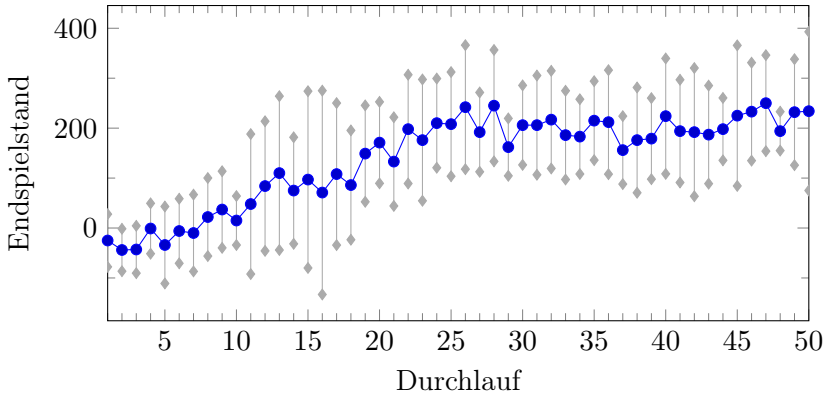


Abbildung 5.16: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 10 (Charlie)

	Mittelwerte					*über-*
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	89,8	−8,9	2,0	6,7	79,6	51%
11–20	96,1	99,9	5,9	7,3	81,3	39%
21–30	83,4	197,2	8,7	6,3	67,0	47%
31–40	75,5	195,4	9,0	6,9	58,4	49%
41–50	74,7	213,9	9,5	6,0	58,2	50%

Tabelle 5.14: Statistiken Konfiguration 10 (Charlie)

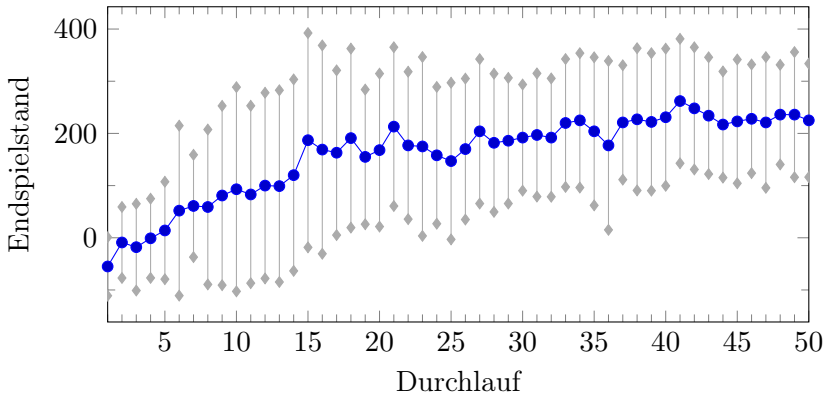


Abbildung 5.17: Erreichter mittlerer Punktestand je Durchlauf von Konfiguration 10 (Dorothy)

Die Versuchsstatistiken, welche für Charlie in Tab. 5.14 und Abb. 5.16 bzw. für Dorothy in Tab. 5.15 und Abb. 5.17 angegeben sind, zeigen, dass beide Agenten zu Episodenende im Mittel vergleichbar hohe Punktzahlen erreichen. Diese liegen von der Größenordnung knapp unter der mittleren Punktzahl der OSiMA-Agenten bei den Benchmark-Tests. Zugleich sind sie höher als die Ergebnisse der ESiMA-Agenten. Die relativ niedrigen Punktestände gehen mit einer sehr geringen

	Mittelwerte					über-
Lauf	*Dauer*	*Punkte*	*+*	*-*	*0*	*lebt*
1–10	90,3	27,7	3,1	6,5	79,4	49%
11–20	96,5	143,5	6,6	5,3	83,6	61%
21–30	83,8	180,4	7,9	5,8	69,2	53%
31–40	76,0	211,6	8,9	5,5	60,6	51%
41–50	75,2	233,0	9,9	6,1	58,3	50%

Tabelle 5.15: Statistiken Konfiguration 10 (Dorothy)

mittleren Spieldauer einher. Diese ist mit hoher Aggressionsbereitschaft auf engem Operationsgebiet erklärbar. Besonders Dorothy konnte gut Gelegenheiten zum Angriff auf Charlie nutzen. Charlie hingegen konnte seine Punktstatistiken verbessern, indem er in einigen Episoden in den Spielfeldabschnitt oben auszuweichen gelernt hat.

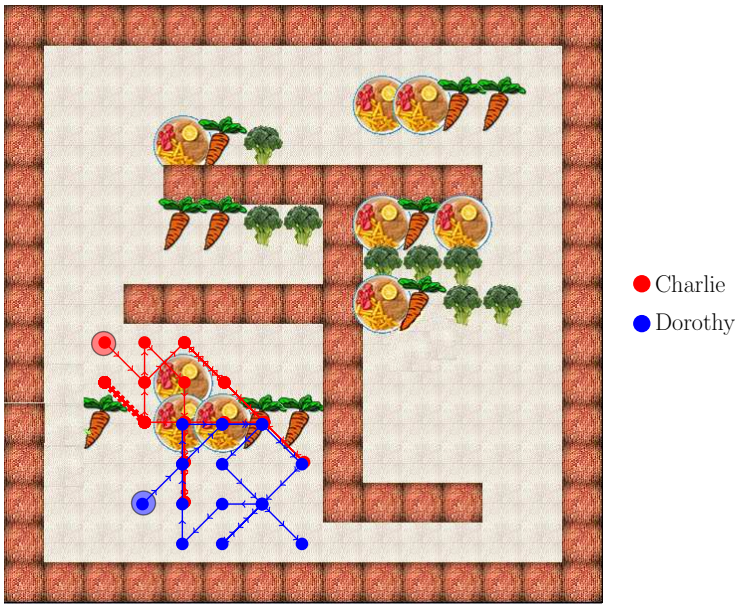


Abbildung 5.18: Darstellung eines Bewegungspfades der Agenten in der Spielwelt (Konfiguration 10, Episode 3, Durchlauf 49)

Abb. 5.18 zeigt hierzu wieder ein Beispiel eines Spieldurchlaufes. Es handelt sich dabei um Durchlauf 49 der 3. Episode. Wie auf dem Bild zu sehen, haben sich Charlie und Dorothy in diesem Fall lediglich im linken unteren Bereich des Spielfeldes aufgehalten. Jeder der beiden Kontrahenten konnte insgesamt neun Aktionen durchführen, welche zu einer Belohnung geführt haben. Diese waren nach nur 15 Spielzügen erreicht. Dies bedeutet, dass zu diesem Zeitpunkt bereits das komplette Feld abgefressen war. Den Rest des Spiels, dessen Dauer insgesamt 77 Züge betrug, sind die beiden ziellos herumgeirrt. Dabei haben die beiden nicht den Versuch unternommen, den jeweils anderen anzugreifen.

5.3.2.11 Anmerkungen und Vergleich der Konfigurationen

Es gibt Aspekte der von den Agenten gefundenen Lösungen, die anhand der Statistiken nicht einwandfrei wiedergegeben werden können. Was hier gemeint ist, ist nicht die quantitave sondern qualitative Beurteilung, welche nur durch visuelle Beobachtung des Spielgeschehens — in

eingeschränktem Maß auch durch Logstudium — möglich wird. Dieser Abschnitt soll einen kurzen Überblick über allgemeine Beobachtungen der Stärken und Schwächen der hier diskutierten kognitiven Architektur geben. Außerdem sind zum direkten Vergleich die Lernkurven der Konfigurationen 1–9 in Abb. 5.19 gegenübergestellt. Es handelt sich jeweils um die über die Episoden gemittelten Endpunktestände für alle Durchläufe.

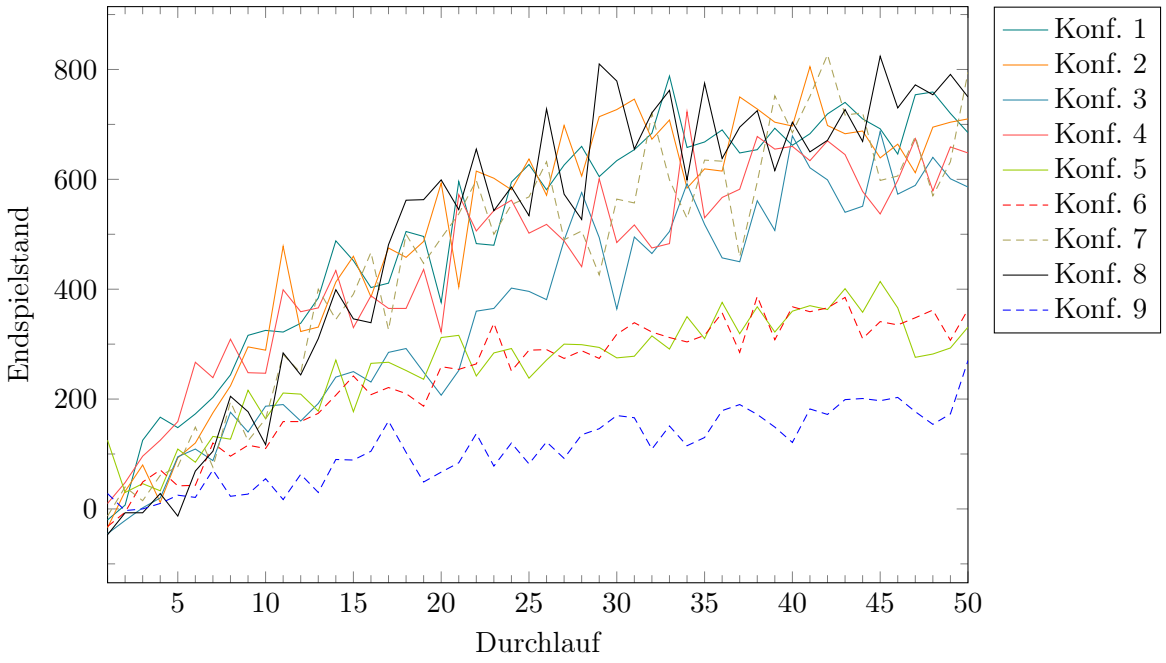


Abbildung 5.19: Vergleich gemittelter Punktestände der Versuchskonfigurationen 1–9

Es hat sich anhand der ersten beiden Versuchsreihen die Wichtigkeit der richtigen Dimensionierung der ANN-Lernrate gezeigt. Die Ergebnisse von Konfiguration 1 und 2 bewegen sich in derselben Größenordnung, jedoch wurde für die weiteren Versuche die geringere Lernrate der Konfiguration 2 beibehalten. Konfiguration 2 verhält sich besser im Zusammenhang mit kontraproduktiven Aktionen durch Exploration. Führt Exploration zur Erlernung einer schlechteren Lösung als bereits gelernt war, gelingt die Reparatur mit geringerer Lernrate in weniger Durchläufen.

Auch haben die Konfigurationen 3 und 4 gezeigt, dass Einführen einer weiteren versteckten Schicht ins ANN keine Vorteile bringt. In diesem Zusammenhang muss auch auf den Rechenaufwand eingegangen werden. Dieser ist unter Verwendung größerer Netze deutlich höher und führt zu einer spürbaren Verlangsamung des Spiels. Tatsächlich können vor allem die Lernvorgänge auch mit den sonst verwendeten dreischichtigen Netzen mitunter sehr lange (mehrere Sekunden) dauern, was den Spielfluss unterbricht. Dies hängt mit dem Online-Lerncharakter der Architektur zusammen. Infolgedessen, und weil sie eine zu komplexe Wertefunktion beschreiben können, eignen sich tiefe ANNs nicht für diese Aufgabenstellung. Erst bei höherer Komplexität der Sensordaten würden sie Vorteile bringen, etwa bei Vorhandensein tatsächlicher Kameradaten.

Mithilfe der Konfiguration 7 konnte gezeigt werden, dass das Hungerneuron als Eingangswert keinen messbaren Einfluss auf die Leistung der Architektur hat. Weiters beweisen die Ergebnisse aus Konfiguration 8, dass der Agent in der Lage ist, ausgehend von beliebigen Gewichtswerten im ANN, ein auf die vorgesetzte Spielsituation angepasstes Verhalten zu entwickeln. Damit ist es auch möglich, einen Agenten, der zuvor für ein anderes Spiel trainiert worden ist, in die gegebene

Spielwelt zu setzen, worauf dieser in der Lage ist, sich nach einigen Durchgängen an die neue Umgebung anzupassen.

Konfigurationen 5 und 6 zeigen, dass die Architektur NNMind Agenten in die Lage versetzt, gegen die Benchmark-Gegner zu bestehen. Der gleiche Versuch wie in Konfiguration 6 wurde in Konfiguration 9 mit verkürztem, codiertem Sensorwort wiederholt. Der Agent lernt so zwar, mit der Situation umzugehen, kann allerdings nicht die Punktestände der anderen Versuchskonfigurationen erreichen. Interessant ist hierbei, dass der Agent hier eine Tendenz zu Angriffen gegenüber Nahrungsverzehr zeigt, was auch die geringen Punktzahlen erklären kann. Tatsächlich haben die trainierten Agenten im Lauf der Versuchsdurchläufe einige Male erfolgreich gegessen, was allerdings nicht zu einer dauerhaften Anpassung des Verhaltens führen konnte.

Allgemein ist zu bemerken, dass sich die gefundenen Lösungswege der Agenten in den einzelnen Versuchsepisoden im Detail immer voneinander unterscheiden. Allerdings kann für bestimmte Ausgangssituationen die Entwicklung ähnlicher Lösungsansätze beobachtet werden. So entwickeln alle NNMind-Agenten, die von Position $(8, 3)$ starten, schließlich das Verhaltensmuster, dass sie zunächst die Nahrung direkt vor ihnen bis in die rechte obere Spielfeldecke abfressen, um sich der Gegend darunter zuzuwenden. Die genaue Vorgangsweise unterscheidet sich zwischen den Episoden.

Es ist anzumerken, dass während der Versuche ein Bug von Miklas identifiziert werden konnte. Dieser manifestiert sich so, dass die Gesundheit des Agenten nicht abnimmt, wenn er eine Vorwärtsbewegung gegen ein Nahrungsobjekt oder einen anderen Agenten vollführt. Da Übertreten von Objekten und Schieben nicht möglich ist, wird bei Aktionswahl nach Policy diese Aktion bis zum nächsten Explorationsschritt durchgeführt. Im Fall der Benchmark-Agenten kann dieser Bug sogar in gewissen Situationen zu einem Deadlock führen. Man stelle sich Adam und Bodo vis-à-vis vor, wobei zwischen ihnen nur ein Feld ist, auf welchem sich ein Nahrungsobjekt befindet. Versuchen beide einen Schritt nach vorne zu gehen, verharren beide auf ewig in diesem Zustand. Dieses Verhalten lässt sich allerdings nur bei ESiMA beobachten.

6 Diskussion

Im Zuge dieser Arbeit sind Erkenntnisse darüber gewonnen worden, wie eine kognitive Architektur mit ANNs ausgestattet entwickelt werden kann, sodass ein in eine Spielumgebung gesetzter Agent sinnvolles Verhalten selbständig erlernen kann. Eine Zusammenfassung dazu wird in Kap. 6.1 geboten. Ein Ausblick auf mögliche zukünftige Entwicklungen auf dem Gebiet der künstlichen Intelligenz, insbesondere der neuronalen Netze, ist in Kap. 6.2 und Kap. 6.3 gegeben.

6.1 Zusammenfassung

In dieser Arbeit wurde eine weitere Möglichkeit aufgezeigt, wie ein Agent anhand RL und ANN erstellt werden kann. Es hat gerade in den vergangenen Jahren eine Vielzahl von Lösungsvorschlägen in verschiedenen Feldern für die Anwendung von RL bei der Schaffung kognitiver Architekturen gegeben. Einige sind in Kap. 2 vorgestellt worden. Als wegweisend gilt Tesauros TD-Gammon [Tes02, S. 181ff.][Tes95, S. 58ff.].

Eine Weise, in welcher sich die in Kap. 3 vorgestellte Architektur von vielen anderen aus der Literatur unterscheidet, ist, dass die Update-Regeln des TD- oder Q-Lernens nicht direkt auf das ANN angewendet wird, sondern die Idee hinter RL genutzt wird, um ein BP-Netz zu trainieren. Es war dazu notwendig, einen Weg zu finden, die Trainingsdaten dafür aus den Aufzeichnungen im STM zu erstellen, sodass das gewünschte Lernverhalten erzielt wird.

Weitere Unterschiede betreffen die Umgebung — die Spielwelt —, in welcher der Agent trainiert und agiert. So trainiert etwa TD-Gammon gegen sich selbst. Andere Arbeiten beschränken sich auf Zielfindungsaufgaben und haben daher keine Gegenspieler [Alt16, S. 4565ff.][GL12, S. 217ff.]. Nur das Lernen von *Atari2600*-Spielen [MKS+13, S. 1ff.] ist in dieser Hinsicht vergleichbar, auch wenn hier tiefe Faltungsnetze verwendet wurden, um die nötigen Aktionen direkt aus dem Bild am Bildschirm abzuleiten. Dahingegen lernt ein Agent, der die für diese Arbeit entwickelte Architektur *NNMind* verwendet, mit zum Teil stochastisch agierenden GOFAI-Gegnern umzugehen, was potentiell eine gewisse Dynamik in das Spielgeschehen bringt. Dabei haben die Agenten in unserem Fall auch ein sehr eingeschränktes Bild ihrer Umgebung. Ein Problem, dem der Agent begegnet, ist, wenn er von hinten attackiert wird. Da der Agent hinten nicht sehen kann, ist die Beobachtung interessant, wie der Agent bei Bestrafung aus unersichtlichem Grund reagiert, und welche Auswirkungen dies in Folge auf seine weitere Verhaltensbildung nimmt. Tatsächlich ist zu bemerken, dass der Agent in solch einem Fall sein Verhalten maßgeblich ändern kann — ob

positiv oder negativ ist von Fall zu Fall verschieden. Eventueller Schaden kann aber in späteren Durchläufen wieder gutgemacht werden.

Wie die in Kap. 5.3.1 präsentierten Versuche zeigen, kann sich ein Agent mit einer kognitiven Architektur, welche ein ANN unter Zuhilfenahme von RL nutzt, dynamisch an seine Umgebung, wie anhand der hier verwendeten Spielwelt präsentiert, anpassen. Es wurde gezeigt, dass die Parameter des ANN wie Neuronenzahl und Lernrate eine gewisse Rolle bei der Leistungsfähigkeit der Architektur spielen. Die Lernrate darf nicht zu klein, nicht zu groß sein. Ist sie zu gering, kann der Agent bei den relativ raren Trainingsdatensätzen nicht ausreichend schnell seine Gewichtsmatrizen so verändern, dass sich dies verhaltensändernd auswirkt. Dadurch würde er unnötig viele Spiele absolvieren müssen. Außerdem sollte die Lernrate hoch genug sein, damit lokalen Minima im BP-Netz entkommen werden kann, welche bei diesem Netztyp oft problematisch sind. Im Gegenteil führt eine zu hohe Lernrate allerdings zu häufigem Überschreiben bereits gelernten erwünschten Verhaltens. Außerdem kann es — wie in Kap. 2.2 angeführt — zur Übersättigung der ANN-Gewichte führen. Der Agent hat somit Probleme, eine stabile Spielstrategie zu entwickeln. Damit ist die Findung eines vernünftigen Kompromisses essentiell.

Auch die Anzahl der versteckten Schichten und der Neuronenanzahl spielt eine Rolle. Höhere Komplexität in ANN durch mehr versteckte Schichten dient eigentlich der besseren Kategorisierung der Sensordaten. Dies macht Sinn, wenn Daten direkt z. B. aus dem visuellen Cortex kommen, rohe Bilddaten also, welche erst verarbeitet, gefiltert und analysiert, werden müssen, bevor sie für Steueraufgaben nützlich sind. Im Fall dieser Arbeit sind die Sensordaten allerdings bereits stark abstrahiert, weswegen mehr als eine versteckte Schicht für diese Aufgabe zu viel Komplexität und zu viele Freiheitsgrade bieten. Durch Halbierung der Neuronenzahl in den versteckten Schichten konnte die Leistung hier zwar verbessert werden, die zusätzliche versteckte Schicht bietet gegenüber den übrigen Konfigurationen dennoch keine Vorteile, da höhere Komplexität beim Lernen auch mehr Rechenzeit erfordert.

Auch hat sich bei anfänglichen Versuchen bald herausgestellt, dass die Architektur eine Deadlock-Erkennung benötigt, um zu vermeiden, dass der Agent sich beständig um die eigene Achse dreht oder Hin-und-her-Bewegungen — also endloses Vor-Zurück bzw. Links-Rechts-Drehungen — ausführt. Da die Aktionsauswahl ε-gierig erfolgt, ist auch eine sinnvolle Wahl von ε erforderlich. Hier wurde stets als Basis $\varepsilon = 0{,}04$ verwendet, was einem Explorationsschritt alle 25 Spielzüge entspricht. Allerdings kann diese Regel bei Erkennung eines Deadlocks oder Untätigkeit über 5 Spielzüge ausgesetzt und sofortige Exploration angesetzt werden. Der gewählte Wert für ε ist ein scheinbar guter Kompromiss. Dennoch hat sich seine Schwäche bei einigen Versuchsreihen gezeigt, in denen der Agent in weniger als 25 Schritten seine Gegner vernichtend geschlagen hat. Dadurch konnte es zu keiner Exploration kommen und das Verhalten des Agenten hat sich festgesetzt. Auch die ESiMA-Gegner haben stets auf dieselbe Weise reagiert, ohne ihr Verhaltensmuster je zu verändern, was zu dieser Fixierung noch beigetragen hat.

Eine interessante Schlussfolgerung aus der Versuchskonfiguration 8 (vgl. Kap. 5.3.2.8) ist, dass die Einführung bzw. Weglassung von Körperinterna — hier nur Hunger — bei der in diesen Versuchen herrschenden Definition der Spielwelt keinen (beobachtbaren) Einfluss auf das Verhalten des Agenten nimmt. Zusätzlich wurde mit Versuchskonfiguration 9 (vgl. Kap. 5.3.2.9) geprüft, ob es möglich ist, das Sensorwort binär zu kodieren, um es kürzer und damit rechen- und speichereffizienter zu gestalten. Mit der Darstellung der drei Objektklassen in nur zwei Bits lt. Tab. 3.4 (S. 29) verkleinert sich die Eingangsschicht immerhin von 88 auf 61 Neuronen. Dies wurde implementiert und ausgetestet. Die Ergebnisse sind im Vergleich zu ähnlich gearteten Versuchskonfigurationen

wenig zufriedenstellend. Vor allem lässt sich eine oftmals verhängnisvolle Änderung des Verhaltens des Agenten bemerken, wonach Aggression der Nahrungssuche vorgezogen wird. So lässt sich das bescheidene Abschneiden dieser Versuchsreihe zum Teil erklären, da Nahrung mehr Punkte bringt und Attacken inhärente Risiken beinhalten. Das im vorherigen Absatz beschriebene Phänomen, dass der Agent zum effektiven Killer wird, ist bei diesen Versuchen übrigens sogar zweimal aufgetreten.

Die Länge des Kurzzeitgedächtnisses sollte aus zwei Gründen beschränkt sein. Der erste ist, dass durch das hier verwendete Training während der Spiellaufzeit, der BP-Algorithmus das Programm nicht zu lange einfriert. Der andere Grund ist, dass weiter zurückliegende Schritte keinen so großen Einfluss auf die unmittelbar gemachten Punkte haben, bzw. besagter Einfluss stark bestritten werden kann. Diesem Umstand wird durch die rekursive Verringerung der Lernrate Rechnung getragen. Bei der verwendeten Gedächtnislänge von 10 Zustands-Aktions-Paaren beträgt die dem ältesten Speichereintrag zugewiesene Lernrate nur noch etwa 39% (vgl. Glg. 3.3) jener des jüngsten Speichereintrags.

Es hat sich auch gezeigt, dass es für den größten Lernerfolg nicht ausreicht, den BP-Algorithmus mit dem gespeicherten Zustands-Aktions-Paar und dem entsprechenden Q-Wert zu beschicken, sondern dass alle anderen möglichen Tupel entsprechend behandelt gehören. Dies bedeutet im Falle einer aufgetretenen Belohnung deren Unterdrückung, wohingegen bei Strafe das gespeicherte Tupel unterdrückt werden muss, dagegen alle anderen ermöglicht. Dies ist der Grund für die Erstellung des *nicht diskriminierenden* Lernalgorithmus, der bei sämtlichen vorgestellten Versuchen zur Anwendung gekommen ist. Allerdings liegt die vorherige *diskriminierende* Lösung für eventuelle nachfolgende Versuche im Quellcode auf.

Agenten mit der hier vorgestellten Architektur können in einem beliebigen inneren Zustand (Gewichtswerte) in eine neue Situation gebracht werden, und sich an diese anzupassen lernen. Die Grundvoraussetzung dafür ist allerdings, dass die nötigen Sensoreingänge und Aktionen verfügbar sind. Der Erfolg kann sich je nach ursprünglichem Zustand nach wenigen Durchgängen zeigen, bei ungünstiger Ausgangslage dauert der Vorgang länger.

Die erste Version von NNMind verfolgt einen Ansatz, der sich dadurch von der gegenwärtigen Architektur unterscheidet, dass das ANN die Sensorwerte in der Eingangsneuronenschicht und für alle möglichen Aktionen je ein Neuron in der Ausgangsschicht hat. Somit hat das ANN in der üblichen Konfiguration 82 Eingangs- und 6 Ausgangsneuronen. Es wird dann jene Aktion gewählt, deren zugeordnetes Neuron den höchsten Ausgabewert aufweist. Diese Maximumsuche könnte mit einem kompetitiven neuronalen Netz durchgeführt werden, welches sich dadurch auszeichnet, dass die Neuronen der Ausgangsschicht dieses ANN-Typs exhibitive Verbindungen mit sich selbst und inhibitive mit allen anderen Neuronen jener Schicht aufweisen. Es hat sich aber im Verlauf anfänglicher Versuche die Formulierung geeigneter Datensätze fürs Training als kompliziert erwiesen, da fürs Lernen eines effizienten Verhaltens eventuell etliche Sonderregeln dem Lernalgorithmus zugefügt werden müssen. Daher wurde dieser Ansatz verworfen, weil es den Voraussetzungen dieser Arbeit zuwiderläuft, wonach Zusammenhänge möglichst ohne Zutun des Programmierers erkannt werden können sollen.

Den Regeln der Aktionswahl und des vorgestellten Lernalgorithmus gemäß lernen Agenten, welche NNMind verwenden, ein bestimmtes auf ihren Erfahrungen fußendes Verhalten. Bemerkenswert ist, dass keine zwei im selben Kontext trainierten Agenten im Verhalten gleich sind. Sie kommen häufig schlussendlich zu ähnlichen, auch annäherungsweise gleichen, Lösungen. Der Lösungsweg unterscheidet sich allerdings. Es treten Verhaltensweisen bei den Agenten individuell auf. Man kann somit von emergentem Verhalten sprechen.

6.2 Zukünftige Arbeiten

Es gibt für die Zukunft etliche Möglichkeiten, wie die Architektur weiterentwickelt und verbessert werden könnte. Einerseits, was sich leicht bewerkstelligen lässt, müssten weitere Aktionen zur Verfügung gestellt werden, welche in Miklas implementiert sind, für das verwendete Spiel *Foodchase* allerdings nicht notwendig und somit weggelassen worden sind. Es handelt sich dabei vor allem um die Aktionen PUSH und PULL, um Objekte über das Spielfeld schieben bzw. ziehen zu können. Infolgedessen könnten dann auch andere Spiele, die unter der Miklas-Spieleumgebung laufen und welche diese Aktionen für ihren Ablauf voraussetzen, gespielt werden.

Weitere Verbesserungsmöglichkeiten sind beim Aufbau und der Organisation des STM gegeben. Dies betrifft die eventuelle Erweiterung der Speichergröße, aber auch die Art, wie und welche Zustands-Aktions-Paare gespeichert werden sollen. Auch der prinzipielle Speicheraufbau kann ersetzt werden. Es kann etwa die Verwendung eines LSTM angedacht werden.

Auch könnte mit Techniken wie *Double Learning* (vgl. Kap. 2.3.2) experimentiert werden. Außerdem kann das ANN hierarchisch um Komponenten erweitert werden, welche weitere verhaltensbeeinflussende Elemente wie Bausteine der SiMA-Architektur beinhalten könnten. Ziel wäre hier, die Aktionswahl nicht rein von der aktuellen äußeren Situation, in der sich der Agent in der Spielwelt befindet, sondern auch von einem mentalen (Gefühls-)Zustand abhängig zu machen. Im Zusammenhang mit dem vorangegangenen Absatz könnte auch versucht werden, verschiedene antrainierte Verhaltensweisen, welche für jeweils bestimmte Spiele gelten, als Modelle abzuspeichern. Damit würde dem Agenten ermöglicht, wenn er das aktuelle Spiel anhand der Umgebung erkennen kann, sein Verhalten sofort entsprechend anzupassen, indem er bereits Gelerntes für diese Umgebung verwendet, und anderes ausblendet.

6.3 Ausblick

Die möglichen Anwendungsgebiete für diese oder ähnliche kognitive Architekturen sind vielfältig. Sie können, wie bereits in Kap. 2.4 aufgezeigt, mit Erweiterungen in der Robotik oder der Gebäudeautomation, aber auch in der Unterhaltungsindustrie (Videospiele) Verwendung finden. Tatsächlich scheint die Menge an Möglichkeiten einer fortgeschritten entwickelten und gut an die Aufgabenstellungen angepassten KI mit einer ähnlichen kognitiven Architektur schier grenzenlos.

Betrachtet man die bereits erfolgten Entwicklungen, wie TD-Gammon [Tes02, S. 181ff.][Tes95, S. 58ff.], das selbst gegen menschliche Meisterspieler gewinnen kann, zeigt sich, dass mit weiterer Entwicklung der KI-bezogenen Technologien lernende Maschinen in Hinkunft auch in anderen Feldern Menschen überlegen sein könnten. Diese Beobachtung beschränkt sich nicht nur auf TD-Gammon, sondern lässt sich auch für andere Spiele, darunter MOBAs (Multiplayer Online Battle Arenas) wie *Dota 2* sagen, bei welchem nach eigenen Angaben von *OpenAI* deren KIs menschliche Teams besiegt haben [2]. Somit ist absehbar, dass KI und damit ANNs in unserer Gesellschaft in Zukunft eine immer größere Rolle spielen werden, mit allen positiven und negativen Implikationen.

Wissenschaftliche Literatur

[Alt16] Altahhan, A.: Self-reflective deep reinforcement learning. In: *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, S. 4565–4570 21, 65

[Ben09] Bengio, Yoshua: Learning Deep Architectures for AI. In: *Found. Trends Mach. Learn.* 2 (2009), Jänner, Nr. 1, S. 1–127. – ISSN 1935–8237 11, 13, 14

[Dow15] Downing, Keith L.: *Intelligence Emerging*. MIT Press, 2015 1, 7, 8, 10, 11, 12, 13, 14

[FMDS14] Franklin, S. ; Madl, T. ; D'Mello, S. ; Snaider, J.: LIDA: A Systems-level Architecture for Cognition, Emotion, and Learning. In: *IEEE Transactions on Autonomous Mental Development* 6 (2014), März, Nr. 1, S. 19–41. – ISSN 1943–0604 6

[Ger01] Gers, Felix: *Long Short-Term Memory in Recurrent Neural Networks*, École Polytechnique Fédérale de Lausanne, Diss., 2001. – `http://www.felixgers.de/papers/phd.pdf` 14, 20

[GL12] Gavrilov, Andrey V. ; Lenskiy, Artem: Mobile Robot Navigation Using Reinforcement Learning Based on Neural Network with Short Term Memory. In: Huang, De-Shuang (Hrsg.) ; Gan, Yong (Hrsg.) ; Bevilacqua, Vitoantonio (Hrsg.) ; Figueroa, Juan C. (Hrsg.): *Advanced Intelligent Computing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–24728–6, S. 210–217 20, 65

[GRS00] Görz, G. ; Rollinger, C.-R. ; Schneeberger, J.: *Handbuch der Künstlichen Intelligenz*. 3. Oldenburg Verlag München Wien, 2000 8, 9

[Kon05] Konar, Amit: *Computational Intelligence*. Springer Berlin, 2005 7, 8, 10, 11, 12, 13, 14

[Kul12] Kulkarni, Parag: *Reinforcement and Systemic Machine Learning for Decision Making*. IEEE Press, 2012 15, 16, 17

[Lai08] Laird, John E.: Extending the Soar Cognitive Architecture. In: *Frontiers in Artificial Intelligence and Applications* Bd. 171, 2008, S. 224–235 6

[MKS+13] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin. *Playing Atari with Deep Reinforcement Learning*. 2013 19, 65

[MKS+15] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg ; Petersen, Stig ; Beattie, Charles ; Sadik, Amir ; Antonoglou, Ioannis ; King, Helen ; Kumaran, Dharshan ; Wierstra, Daan ; Legg, Shane ; Hassabis, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Februar, S. 529–533 19

[NAO18] Nguyen, Phong ; Akiyama, Takayuki ; Ohashi, Hiroki: Experience Filtering for Robot Navigation using Deep Reinforcement Learning. In: *Proceedings of the 10th International Conference on Agents and Artificial Intelligence* Bd. 2, SCITEPRESS — Science and Technology Publications, Lda., 2018, S. 243–249 19

[Rä10] Rädle, Klaus: *Neuronale Netze; eine Einführung mit Programmbeispielen.* Berlin : Pro Business, 2010 8, 9, 11, 12, 13

[SB18] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction.* 2. Edition. MIT Press, 2018. – Draft 14, 15, 16, 17, 18

[SHM+16] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent: Mastering the game of Go with deep neural networks and tree search. In: *Nature* (2016), Jänner, Nr. 529, S. 484–489 19

[Sun07] Sun, Ron: The importance of cognitive architectures: an analysis based on CLARION. In: *Journal of Experimental & Theoretical Artificial Intelligence* 19 (2007), Nr. 2, S. 159–193 6

[SWK+15] Schaat, Samer ; Wendt, Alexander ; Kollmann, Stefan ; Gelbard, Friedrich ; Jakubec, Matthias: Interdisciplinary Development and Evaluation of Cognitive Architectures Exemplified with the SiMA Approach / Institute for Computer Technology, Vienna University of Technology. 2015. – Forschungsbericht 6, 7

[Tes95] Tesauro, Gerald: Temporal Difference Learning and TD-Gammon. In: *Communications of the ACM* 38 (1995), März, Nr. 3, S. 58–67 18, 65, 68

[Tes02] Tesauro, Gerald: Programming backgammon using self-teaching neural nets. In: *Artificial Intelligence* 134 (2002), Nr. 1, S. 181–199. – ISSN 0004–3702 18, 65, 68

[Wen16] Wendt, Alexander: *Experience-Based Decision-Making in a Cognitive Architecture*, TU Wien, Diss., 2016 7

Internet Referenzen

[1] Fiszel, R.: *RL4J*. https://github.com/deeplearning4j/rl4j, 22.3.2018.

[2] OpenAI: *OpenAI Five*. https://blog.openai.com/openai-five, 3.9.2018.

[3] Plüss, A.: *Package ch.aplu.jgamegrid*. http://www.aplu.ch/classdoc/jgamegrid/ch/aplu/jgamegrid/package-summary.html, 1.6.2018.

[4] Plüss, A.: *The JGameGrid Framework*. http://www.aplu.ch/home/apluhomex.jsp?site=45, 26.6.2017.

[5] QOS.ch: *Simple Logging Facade for Java (SLF4J)*. https://www.slf4j.org, 14.6.2018.

[6] Sevarac, Z.: *Neuroph*. http://neuroph.sourceforge.net/index.html, 20.1.2018.

[7] Skymind: *A Beginner's Guide to Deep Reinforcement Learning*. https://deeplearning4j.org/deepreinforcementlearning, 22.3.2018.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 16. November 2018

Rupert Schneeberger, BSc.