Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

OpenSpiel Anbindung an Prolog

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Philipp Klotzsche

Beginn der Arbeit: 21. September 2022
Abgabe der Arbeit: 30. Dezember 2022

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Dr. C. Bolz-Tereick

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30. Dezember 2022

Philipp Klotzsche

Zusammenfassung

Prolog war einst eine zentrale Programmiersprache in der Entwicklung künstlicher Intelligenz. Über die Jahre ist die Sprache aber in Beliebtheit, Umfang und Ausbau in diesem Bereich durch andere Entwickler überholt worden. Python ist eine dieser Sprachen, die vor allem durch ihre leichte Handhabung, einsteigefreundliches Design und Besitz starker mathematischer Bibliotheken über die Jahre viele Entwickler angezogen hat und zu einer Explosion in der Begeisterung für Machine Learning mitgeholfen hat.

Das Framework OpenSpiel spielt dabei für die Entwicklung von Reinforcement Learning Algorithmen eine bedeutende Rolle, um verschiedene Ideen und Programmiersprachen zusammenzuführen. Die auf C++ und Python basierende Ansammlung an Machine Learning Techniken bietet eine besondere Grundlage für eine Erweiterung auf zusätzliche Ansätze und Programmiersprachen.

In dieser Arbeit präsentiere ich die Verwendung von Reinforement Learning Algorithmen aus OpenSpiel auf Definitionen von Spielen innerhalb Prologs. Diese Anbindung erlaubt es, einfach strukturierte Spielabläufe mit den starken Werkzeugen in Python auszuwerten.

Ich habe die Schnittstelle auf Funktionalität und Richtigkeit im Vergleich zum ursprünglichen Framework getestet und eine zeitliche Performance Analyse durchgeführt. Dabei habe ich berücksichtigt, was es bedeutet, ein Spiel zu spielen und welche Voraussetzungen gegeben sein müssen, um die Algorithmen richtig zu verwenden, sowie Brücken für die zukünftige Erweiterung gebaut. Innerhalb dieser Arbeit konnte ich die Ergebnisse erfolgreich validieren.

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. Leuschel für die Überlassung des interessanten Themas dieser Bachelorarbeit und seiner Betreuung.
Bedanken möchte ich mich ebenfalls bei meinem Zweitgutachter Herrn Dr. Bolz.
Ich möchte mich bei allen Mitarbeitern vom Lehrstuhl "Softwaretechnik und Programmiersprachen" bedanken, die mir während meines Studiums mit Rat und Tat zur Seite standen.

Besonders möchte ich mich für die finanzielle Unterstützung meines Studiums bei meinem Eltern, Frau Liane Klotzsche und Herrn Steffen Kirbach bedanken.

Inhaltsverzeichnis

1 Motivation **1**

2 Grundlagen **1**

2.1 Prolog 1
2.1.1 Design 2
2.1.2 Datentypen 2
2.1.3 Queries 4
2.1.4 Fehler 4
2.2 Python 5
2.2.1 Design 5
2.2.2 Datentypen 6
2.3 Exceptions 7
2.4 PySwip 7
2.4.1 Installation 7
2.4.2 SWI-Prolog Engine finden 7
2.4.3 Query von Python an Prolog 8
2.5 Machine Learning 9
2.5.1 RL: Verstärktes Lernen 9
2.5.2 OpenSpiel 9
2.5.3 OpenSpiel Voraussetzungen 10
2.5.4 Umgebung 10
2.5.5 Spiele 12
2.5.6 Algorithmen 12

3 Umsetzung **14**

3.1 Spiel in Prolog 14
3.1.1 Spielelogik 15
3.1.2 Brücke für PySwip 16
3.2 Wrapper Klasse 18
3.2.1 Spiel Objekt 19
3.2.2 Spielzustand Objekt 19

3.2.3 Hilfsmethoden 20
3.3 Observerklasse 21

4 Fallstudien **22**

4.1 Funktionstest 22
4.1.1 Minimax mit Alpha-Beta-Pruning 22
4.1.2 MCTS: Monte Carlo Tree Search 24
4.2 Lerntest 26
4.2.1 Umgebung 26
4.2.2 Q-Learner 26
4.3 Performancetest 28
4.4 Erweitert auf weitere unterstützte Spiele 29

5 Zukünftige Arbeiten **30**

5.1 Erweiterung 30
5.2 Speichern in Prolog 30
5.3 Error Handling 31

6 Fazit **31**

Abbildungsverzeichnis **32**

Tabellenverzeichnis **32**

Algorithmenverzeichnis **32**

Quellcodeverzeichnis **32**

Literatur **33**

1 Motivation

Die Entwicklung von künstlicher Intelligenz (englisch *Artificial Intelligence*, `AI`) hat in den letzten Jahren bedeutende Fortschritte [LBH15] erzielt, mit einer weiten Reihe an Hilfsmitteln und Algorithmen, die in der Lage sind, zu intelligenten Systemen anhand von Datensätzen trainiert zu werden. Ein Kern-Konzept dabei ist, gelernte Ergebnisse aus einer vorgegebenen Umgebung auf ein neues, unbekanntes Umfeld anzuwenden und positive Resultate zu erzielen.

Verstärktes Lernen (`Reinforcement Learning`) ist ein leistungsfähiger Teil des Machine Learnings und findet in vielen Bereichen der heutigen Zeit Gebrauch. OpenSpiel [LLL+19] ist eine Open Source Bibliothek, die eine Vielzahl an Werkzeugen und Algorithmen anbietet, um Systeme im verstärkten Lernen zu trainieren.

Prolog [SS94] ist eine Programmiersprache, die gut darauf ausgelegt ist, übergebene Probleme anhand eines gespeicherten Umfeldes zu lösen. Anstelle von verfahrensorienterten Befehlen werden in Prolog ein Set an möglichen Lösungen definiert. Das System bekommt daraufhin Probleme vorgelegt, für die es aus der eigenen Umgebung nach möglichen Lösungen sucht.

Das Ziel dieser Bachelorarbeit ist es, eine API (`Application Programming Interface`) zu entwickeln, welche die Anbindung von in Prolog definierten Spielen an die Algorithmen aus OpenSpiel ermöglicht. Das Ergebnis ermöglicht es zukünftig, Entwicklern, Prolog für die Ausbau weiterer AI-Anwendungen über Python zu verwenden.

2 Grundlagen

Im Folgenden stelle ich das Hintergrundwissen dar, welches zur Anwendung der Anbindung und zum Verständnis dieser Bachelorarbeit benötigt wird. Dies beinhaltet die Sprachen Python und Prolog, die Brücke PySwip[Tc20] zur Abfrage von Prolog-Regeln und -Fakten sowie das Framework OpenSpiel[LLL+19].

2.1 Prolog

Die Programmiersprache Prolog [CR96] wurde ursprünglich zur Verarbeitung menschlicher Sprache entwickelt. Sie wurde 1972 veröffentlicht und existiert heute in verschiedenen Ausführungen. Seit 1987 besteht die vielseitige Umsetzung in SWI-Prolog [WSTL12], die bis heute weiter entwickelt wird und bereits viele High-Level Interfaces zu anderen Programmiersprachen unterstützt wie z.B. Java, Python und Rust. Die Bezeichnung kommt von dem Namen der niederländischen Gruppe der Entwickler “ Sociaal-Wetenschappelijke Informatica“ (“Social Science Informatics“).

2.1.1 Design

Prolog ist eine logische Programmiersprache [Bra90], die im Unterschied zu vielen bekannteren Sprachen nicht das imperative Muster zur Bewältigung von Problemen verwendet, sondern ein deklaratives Paradigma. Im imperativen Ansatz werden Problemlösungen in Algorithmen Schritt für Schritt angegeben, bis eine Lösung erzielt wird; in Prolog werden Fakten und Regeln beschrieben, welche die Lösungen für ein Problem darstellen. Um Lösungen zu erhalten, muss der Prolog-Engine ein Problem beschrieben werden, woraufhin die Engine in der Prolog-Datenbank sucht, ob eine passende Lösung existiert. Dies geschieht mit der in der Sprache eingebauten und effizienten Vereinigung (`unification`), um gefragte Variablen und Terme zu vereinen . Es wird dabei versucht, die Anfrage anhand der gegebenen Datenlage in Prolog auf ihre Richtigkeit zu überprüfen. Sollten sich in der Anfrage auch Variablen befinden, versucht die Prolog Engine, diese mit einem Wert gleichzusetzen, damit die Aussage wahr wird.

2.1.2 Datentypen

In Prolog bezeichnet ein Term einen Ausdruck, der einen Wert oder eine Operation darstellt. Ein Term kann die Form eines Atoms, einer Variable, einer Struktur (`compound term`) oder einer Liste annehmen.

Ein `Atom` ist eine Zeichenkette, beginnend mit einem kleinen Buchstaben, die nicht weiter unterteilt werden können. Sie werden als Bezeichner für Predicates oder Funktionen sowie für Konstanten verwendet.

`Variablen` sind noch nicht festgelegte Placeholder für Terme, die bei der Unification gesetzt werden können, um eine wahre Aussage bei Abfragen zu erreichen. Sie beginnen mit einem Großbuchstaben oder einem Unterstrich. Bei letzterem handelt es sich um eine anonyme Variable, deren Ergebnis zwar bei Berechnungen berücksichtigt wird, aber im Endergebnis nicht angegeben werden.

Ein `compound term` ist entweder ein Predicate oder eine Funktion, die aus dem Bezeichner als Funktionssymbol und einer Liste an Argumenten besteht. Sie werden in Prolog, beginnend mit dem Bezeichner, gefolgt von der Liste der Argumente in Klammern dargestellt. Eine solche Struktur ohne Argumente ist ein Atom. In der Dokumentation werden die Strukturen in der Form `Bezeichner/AnzahlDerArgumente` erwähnt.

Eine `Liste` ist eine Sequenz von Termen und Werten, die in verschiedenen Formen angesprochen werden kann. Die ersichtlichste Form bildet die Auflistung der Inhalte, getrennt mit einem Komma und komplett in eckige Klammern gesetzt. Die Prolog-Engine unterstützt aber auch die Unterteilung in `Head` und `Tail`: die ersten / das erste Element(e) und die Restliste, getrennt mit dem Atom `"`. Die Restliste kann entweder eine Liste von Werten und Termen oder die leere Liste verkörpern. Die beschriebenen Darstellungen der Liste sind im Quellcode 1 zu sehen.

Quellcode 1: Liste Beispiel

```
1: []                % leere Liste
2: [1,a,"Text"]      % Liste mit möglichen Termen
3: [1|[a,"Text"]]    % die gleiche Liste mit Head-Tail
4: [1,a|T]           % die gleiche Liste mit T = ["Text"]
```

2.1.3 Queries

Eine Abfrage (`Query`) ist die Möglichkeit, der Prolog-Datenbank Fragen zu stellen. Die Anfrage beginnt mit einem Fragezeichen und besteht aus einer oder mehreren mit Kommas getrennten Predicates und endet mit einem Punkt. Die Aufgabe der Prolog-Engine ist es hier, die Predicates in der Datenbank zu finden, gegebenenfalls mit gesetzten Variablen. Werden zutreffende Regeln gefunden, gibt die Engine "Yes" oder "True" sowie die gesetzten Variablenwerte aus; wenn mindestens eines der Predicates nicht gefunden wurde, ist die Antwort "No" oder "False".
Die Prolog Engine verarbeitet den Code Top-Down, d.h. bei einer Abfrage werden die passenden Datensätze von oben nach unten ausgewertet. In Quellcode 2 ist ein einfaches Beispiel der Syntax zu sehen.

Quellcode 2: einfaches Prolog Beispiel

```
1: % Fakten. parent/2, male/1, female/1
2: parent(james, harry).
3: parent(lily, harry).
4: male(james).
5: female(lily).
6:
7: % Regeln. mother/2, father/2
8: mother(Mother, Child) :-
9:     parent(Mother, Child),
10:    female(Mother).
11: father(Father, Child) :-
12:    parent(Father, Child),
13:    male(Father).
14:
15: % Abfragen
16: ?- parent(P, harry).
17: % = Wer ist alles ein Elternteil von Harry?
18: %  Yes, P = james,
19: %  Yes  P = lily.
20:
21: ?- father(X, harry).
22: % = Wer ist der Vater von Harry?
23: % Lösung: F = james.
```

2.1.4 Fehler

Wenn die Prolog Engine auf eine Situation trifft, in der es den Code nicht mehr ausführen kann, wirft das System eine Exception [CWA+22]. Erhält beispielsweise ein eingebautes Predicate ein Argument des falschen Datentyps, wirft es einen `Type_Error`. Der Standardfall besteht in der Unterbrechung des Programmablaufs, das Schreiben der Fehlermeldung auf der Konsole und das Beenden des Programms. Danach ist das System wieder auf der

höchsten Ebene ebenso wie vor dem letzten Aufruf des Programms.
Prolog unterstützt dafür mehrere Wege, den Programmablauf nicht komplett abstürzen zu lassen. Eine dieser Möglichkeiten stellt das Umschließen von aufgerufenen Predicates mit einem Exception Handler wie `catch/3` dar.
Sollte dabei kein Fehler auftreten, wird das beschützte Predicate normal ausgeführt. Tritt dagegen ein erwarteter Fehler während der Ausführung auf, wird das gesamte Ziel des Predicates aufgegeben und gesetzte Variablen wieder zurückgesetzt. Anstelle dessen wird nun ein Ersatzpredicate ausgeführt, welches dem Programmierer erlaubt, ein genaueres Verhalten des Programmablaufs zu bestimmen. Somit kann z.B. eine Fehlermeldung angepasst werden oder der Abbruch des Programmablaufs konkret verhindert werden.
Ein einfaches Beispiel wäre das Abfangen einer Benutzereingabe. Wenn ein Predicate eine Eingabe einer Zahl erwartet, der Benutzer aber eine Zeichenkette aus Buchstaben übergibt, dann kann ein Exception Handler den Input abfangen, auf der Konsole eine Nachricht ausgeben, dass eine Zahl erwartet wird und dann neuen User-Input ermöglichen.

2.2 Python

Python[VR+07] ist eine allgemeine interpretierte Programmiersprache, die 1991 von Guido van Rossum erfunden und entwickelt wurde. Sein Ziel war es, eine Sprache zu entwickeln, die leicht lesbar und leicht zu lernen ist. Sie sollte außerdem für eine Vielzahl an Anwendungen anwendbar sein.

2.2.1 Design

Einrückungen sind in vielen anderen Programmiersprachen optional und werden als gute Codepraxis angesehen, da es den Code leichter für andere Programmierer lesbar macht, in Python ist sie jedoch Teil der Syntax. Damit werden zueinander gehörende Codeblöcke definiert [VRD10] und vom Interpreter als solche ausgewertet. Das reduziert Programmierfehler und es wird an Semicolons und Klammern gespart, die in anderen Sprachen semantisch notwendig sind, aber den Code schwerer verständlich machen können.

Python unterstützt mehrere Programmierparadigmen. Ähnlich zu den Sprachen C und Pascal lassen sich Methoden und Funktionen als wiederverwendbare Codeblöcke schreiben. Diese haben ein in sich geschlossenen Ablauf mit Eingabe von Werten, Durchführen von Aktionen und eine optionale Produktion einer Ausgabe. In Python können Methoden mit dem Schlüsselwort `def` definiert werden und Kontrollstrukturen wie Schleifen mit `while` und `for` begonnen werden. Das beschreibt das prozedurale Prinzip.

Daneben verfolgt Python auch Ansätze der funktionalen Programmierung, welche das Verarbeiten von Daten mit solchen Funktionen ermöglicht. Bei funktionalen Sprachen

können Funktionen erster Klasse als Variablen genutzt werden. Werte werden ausschließlich durch Transformation anderer Werte mit Funktionen erzeugt.

Im Vergleich zu Prolog ist Python eine imperative Sprache, in der Anweisungen gegeben werden, welche Aktionen in welcher Reihenfolge ausgeführt werden sollen, um den Zustand des Programms zu verändern. Der Name leitet sich aus dem Imperativ ab: Es wird ein Befehl gegeben, der ausgeführt werden soll. In Python wird dadurch die Struktur des Programmablaufs definiert: wann und wie Variablen spezifiziert werden, Berechnungen durchgeführt werden und welche Programmzweige durch Kontrollstrukturen wie `if-else` ausgewählt werden.

Numpy[HMW+20] ist eine essenzielle Bibliothek innerhalb Pythons, um mit großen Datenmengen in sortierter Form zu rechnen. Deren Werkzeuge werden vermehrt in wissenschaftlichen Berechnungen und Datenanalyse verwendet und bilden eine Grundlage für größere Berechnungen und Frameworks wie OpenSpiel[LLL+19]. Die Kombination hat dazu beigetragen, Python als beliebte Wahl für Machine Learning Auswertungen auszuwählen.

2.2.2 Datentypen

Python ist ebenfalls eine objektorientierte Sprache. Das bedeutet, dass jedes Objekt eine Schablone (die Klasse) hat, wie solch ein Objekt auszusehen und zu handeln hat und welche Methoden darauf anwendbar sind. Es existieren Mechanismen wie Vererbung (Tochterklassen können mindestens so viel wie die Elternklasse) und Polymorphismus (verschiedene Formen für Objekte der gleichen Klasse). Ein Beispiel dafür ist das Konzept der virtuellen Methoden in C++. Dabei können abgeleitete Klassen Methoden der Basisklasse überladen, um eine genauere Implementation zu erstellen (=Vererbung). Wenn es zwei abgeleitete Klassen der gleichen Basisklasse gibt, handelt es sich dabei um Polymorphismus.
Das trifft auch auf alle Datentypen zu. Einige Datentypen sind in Tabelle 1 dargestellt.

Datentyp	Nutzen	Beispiel
Integer	Ganzzahl	42
Float	Kommazahl	3.14154
String	Zeichenkette	"Text"
Boolean	Wahrheitswert	True
List	sortierte veränderbare Sammlung	[1,2,'e']
Dictionary	wie Liste mit Schlüssel-Wert-Paaren	{'key' : 'value'}
bytes	unveränderbare Folge an Bytes	b'Text'
Generator	Iterable zum Sammeln an Werten	<Python Generator Objekt>

Tabelle 1: Überblick über einige Python Datentypen

2.3 Exceptions

Fehler, die in Python geworfen werden können, führen nicht direkt zum Unterbrechen des Programms. Python versucht über den Call-Stack die aufgerufenen Methoden aufzurollen und eine mögliche Handhabung eines Fehlers über einen `try` und `catch`-Block zu finden. Sobald ein passender Block gefunden wird, läuft das Programm an diesem Punkt im Stack weiter. Nur wenn keine Handhabung gefunden wird, terminiert das Programm und Python wirft den Fehler auf der Konsole mit einer passenden Fehlermeldung aus. Dieser Umgang mit Fehlern an beliebigen Stellen in Python ermöglicht eine weitere Kontrolle des Programmablaufs. Diese Handhabung ähnelt dabei sehr der Handhabung in Prolog.

2.4 PySwip

PySwip[Tc20] ist eine Brücke zwischen Python und SWI-Prolog, die es ermöglicht, Prolog Abfragen in Pythonklassen zu stellen. Diese Anbindung ist der Schlüssel für diese Bachelorarbeit: eine Anbindung von Prologdefinitionen an Algorithmen in Python zu bewerkstelligen. PySwip unterstützt Python ab der Version 3.6, SWI-Prolog ab Version 8.2 und benötigt `libswipl` als eine geteilte Bibliothek. Letzte wird meist bei der Installation von SWI-Prolog eingestellt. Wichtig ist, dass die installierte Architektur von SWI-Prolog und von Python übereinstimmen (z.B. 64bit build).

2.4.1 Installation

Die Schritte zur Installation sind aus dem Github Repository[1] für das entsprechende Betriebssystem zu entnehmen.
Als Beispiel für Linux muss zuerst versichert werden, dass SWI-Prolog installiert ist. Danach muss über den Package Manager pip pyswip installiert werden, siehe Quellcode 3.

Quellcode 3: PySwip installieren

```
1: $ sudo apt install swi-prolog
2: $ pip install pyswip
```

2.4.2 SWI-Prolog Engine finden

PySwip versucht, die Engine schrittweise auf dem Betriebssystem ausfindig zu machen: zuerst über die von SWI-Prolog benannte Umgebungsvariablen, dann über die `PATH`-Umgebungsvariable und zuletzt über ein paar Standardinstallationspfade wie `/usr/bin/swipl` oder `/usr/local/bin/swipl`. Erst wenn diese Möglichkeiten ausgeschöpft sind, wirft

[1] https://github.com/yuce/pyswip/blob/master/INSTALL.md, 21. September 2022

PySwip einen Error.

2.4.3 Query von Python an Prolog

Nun wurden alle Voraussetzungen erfüllt, um Anfragen an die SWI-Prolog Engine zu senden und auswerten zu lassen. Zuerst muss der Interpreter geladen werden (`pyswip.Prolog`). Falls eine Prolog Datei vorhanden ist, kann diese mit Angabe des (relativen) Pfades für den Interpreter geladen werden (`pyswip.Prolog.consult`). Ansonsten können auch Regeln über `pyswip.Prolog.asserta` (oben) oder `pyswip.Prolog.assertz` (unten) im leeren Interpreter zur Laufzeit als String definiert werden. Eine Anfrage wird ebenfalls als ein String bei der `pyswip.Prolog.query`-Methode übergeben. Die Syntax entspricht dabei der Syntax von Prolog, jedoch dürfen weder Regeldefinitionen noch Queries mit einem Punkt enden. Dieser wird von PySwip jeweils am Ende angehangen.
Ein Ergebnis der Anfrage wird in einem Generator gesammelt, der am besten direkt zu einer Liste gecastet wird. Wurde kein Ergebnis gefunden, ist diese Liste leer. Wenn ein Ergebnis erfolgreich gefunden wurde, wird dieses als Dictionary eingefügt. Falls Variablen gesetzt wurden, werden diese in den Dictionaries als Schlüssel und die gesetzten Werten als Values gesetzt. Pro erfolgreiches Ergebnis der Query wird ein Dictionary gesetzt.

In 4 ist ein beispielhafter Durchlauf mit dem Prolog Code Beispiel 2 gezeigt:

Quellcode 4: eine Query von Python an Prolog erstellen

```
1:  import pyswip
2:
3:  # Interpreter erstellen
4:  prolog = pyswip.Prolog()
5:
6:  # Predicates einer Prolog Datei laden
7:  prolog.consult("harry_potter.pl")
8:
9:  # Eine Anfrage als String definieren
10: query = "father(X,Y)"
11:
12: # Anfrage an den Prolog Interpreter senden, Ergebnis als Generator
13: query_result = prolog.query(query)
14:
15: # Ergebnis auswerten
16: for solution in query_result:
17:     print(solution)
18: # {'X': 'james', 'Y': 'harry'}
19: # {'X': 'lily', 'Y': 'harry'}
20:
21: # Anfrage ohne Ergebnis
22: failing_query = "father(X, dobby)"
```

```
23: empty = list(prolog.query(failing_query))
24: # empty = []
```

2.5 Machine Learning

Machine Learning [Alp14] beschreibt das Trainieren von Algorithmen (`Artificial Intelligence`), um Entscheidungen und Voraussagen über zukünftige Ereignisse zu treffen, ohne direkt dafür programmiert zu sein. Ziel ist es, dass die AI über den Lernfortschritt anhand der Datenlage eine allgemein zutreffende Lösung für Probleme findet, die sie beim Trainieren noch nicht gelernt hat.
Es gibt viele verschiedene Unterteilungen des Machine Learnings, die sich in der Art des Trainings und der damit angewandten Themengebiete unterscheiden.

2.5.1 RL: Verstärktes Lernen

Reinforcement Learning[SB18] ist ein großer Unterbereich des Machine Learnings. Hierbei wird eine künstliche Intelligenz in eine unbekannte Umgebung geschickt und für gewählte Handlungen belohnt oder bestraft. Ziel der AI ist es, über das Erkunden der Umgebung eine Strategie zu entwickeln, die zu einem positiven Ergebnis führen und schlechte Ergebnisse vermeidet. Dies geschieht über reines Erforschen der Situation und das Lernen aus den Ergebnissen für weitere Durchläufe. Die künstliche Intelligenz muss selbst in dem Umfeld lernen, mithilfe der Erfolge und Niederlagen bessere Entscheidungen zu treffen.
Aus den gelernten Datensätzen soll die AI ab einem gewissen Punkt entscheiden, nicht mehr zu forschen sondern das gelernte Wissen anzuwenden und die bestmögliche Entscheidung für die derzeitige Situation zu finden. Das beschreibt den `Exploration vs. Exploitation`-trade-off. Um eine Belohnung zu maximieren, soll die AI gelernte Aspekte ausnutzen und damit positive Ausgänge versichern (`exploit`). Um aber mögliche bessere Wege zu finden, müssen bisher unbekannte Schritte erforscht werden, um sie bewerten zu können (`explore`).
Anwendungsbereiche von Reinforcement Learning beinhalten Patientenversorgungen im Gesundheitswesen [SSJ18], in Ressourcenverteilung [SOM+20] sowie in der Entwicklung von Spielen.

2.5.2 OpenSpiel

OpenSpiel[LLL+19] ist ein OpenSource Framework von DeepMind, einer Tochtergesellschaft von Google, die sich auf die Entwicklung universell anwendbarer künstlicher Intelligenz fokussiert. Das Framework beschäftigt sich mit der Forschung in verstärktem Lernen, Suche und Planungen in Spielen.
Die Bibliothek zielt darauf ab, Entwickler zusammen zu bringen und die Forschung voranzutreiben und reproduzierbaren leichten und verständlichen Code zu erzeugen. Es bildet

eine flexible und starke Grundlage für das Erstellen, Trainieren und Testen von künstlicher Intelligenz.

2.5.3 OpenSpiel Voraussetzungen

Wie auch für PySwip gibt es hier Betriebssystem-abhängige Möglichkeiten, OpenSpiel zu installieren[2]. Wenn man beispielhaft nur die Python API verwenden möchte, reicht es aus, über den Python Package Manager das Modul wie in Quellcode 5 zu installieren:

Quellcode 5: OpenSpiel installieren

```
1: python3 -m pip install open_spiel
```

Die Vollversion 1.0.0 von OpenSpiel verlangt eine x86_64 Computerarchitektur sowie Python Version 3.7 bis 3.10. Für die Ausarbeitung dieser Bachelorarbeit habe ich Features von Python 3.10 genutzt.
Möchte man Spiele oder Algorithmen anfertigen oder erweitern, muss man das Repository clonen und daraufhin wie in Quellcode 6 alle Dependencies installieren und alles builden und testen lassen. Letzteres muss bei Updates zum Framework wiederholt werden, um alles auf dem neusten Stand zu behalten.

Quellcode 6: OpenSpiel zur Entwicklung installieren

```
1: ./install.sh
2: ./open_spiel/scripts/build_and_run_tests.sh
```

2.5.4 Umgebung

Die in OpenSpiel für Python konzipierte Umgebungsklasse **rl_environment.py**[3] unterstützt Spiele mit nacheinander folgenden, zufallsbasierten (`Chance Nodes`) Zügen und Spiele mit gleichzeitigen Zügen. Es benötigt eine Instanz einer unterstützen Spieleklasse, mit der die gespeicherten Spielattribute und -informationen geladen werden. Diese werden bei einem Spielzug eines Spielers sowie bei weiteren Spieler-unabhängigen Zügen abgerufen.

Ein Spielzustand wird in einem `TimeStep`-Tuple repräsentiert, in dem Informationen zum aktuellen Spielezustand abgerufen werden. Das Tuple beinhaltet die Werte wie in Tabelle 2.

Der Observation Datensatz beinhaltet vier Schlüsselinformationen für den aktuellen Zustand. Beispielhaft wird hier ein konkreter Datensatz für einen Zustand in Tic-Tac-Toe beschrieben:

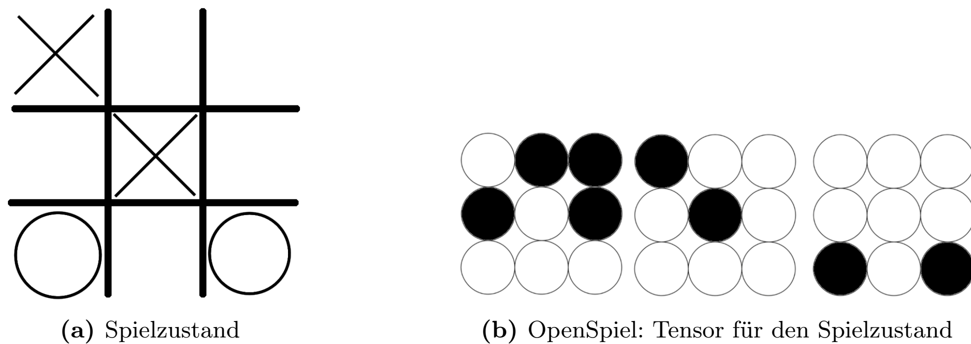
[2] https://github.com/deepmind/open_spiel/blob/master/docs/install.md, 21.September 2022
[3] https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/rl_environment.py, 29. November 2022

Attribut	Datentyp	Inhalt
Observation	Liste von Dicts	Beobachtung des Spielezustands pro Spieler
Reward	Liste von Ints	Punkte für alle Spieler
Discounts	Liste an Ints	Werte der Abzüge pro Spieler
step_type	Enum	Benennung des Spielezustandes

Tabelle 2: Time Step Werte

- Der `InfoState`, wie in Abbildung 1, beinhaltet die Information über den Zustand des Spiels. So ist der Zustand vom 3x3x3 Tensor, reduziert auf eine Liste von 27 Elementen, gespeichert. Die ersten 9 Einträge speichern das Spielfeld, welche Stellen schon gespielt (weiß, 0) beziehungsweise noch frei (schwarz, 1) sind. Die nächsten 9 Einträge stellen die gespielten Felder des ersten Spielers dar. Die letzten 9 Einträge enthalten die gespielten Felder des zweiten Spielers.

  Dieser InfoState wird aus Sicht jedes Spielers gespeichert. Dieser ist in diesem Beispiel für beide Spieler identisch, da alle Spieler alle Informationen (`perfect information`) über das Spielfeld besitzen.



(a) Spielzustand **(b)** OpenSpiel: Tensor für den Spielzustand

Abbildung 1: Tic Tac Toe Spielzustand im InfoState

- Die `LegalActions` halten die erlaubten Züge, sortiert nach den Spielern. Der Spieler, der nicht am Zug ist, hat daher auch keine erlaubten Züge in diesem Zustand.

- Der `CurrentPlayer` speichert den Spieler, der gerade am Zug ist. Bei Tic-Tac-Toe gibt es nur die Spieler für `X` und `O`, aber in anderen Spielen wie z.B. Kuhn-Poker gibt es weitere "Spieler", die keine aktiven Teilnehmer am Spielgeschehen sind. So wird der Spieler beim Ziehen einer zufälligen Karte aus dem Deck als eigener Zufallsspieler behandelt.

- Der `SerializedState` speichert alle Informationen ab, die zum jetzigen Spielzustand geführt haben. Es speichert Spielnamen und den Spielzustand, begonnen bei den bisher

angewendeten Zügen und abschließend die String-Representation des Spielfeldes mit den gesetzen und ungesetzten Feldern. Dieser Zustand wird einerseits zur Darstellung, andererseits auch zum Deserialisieren und der damit verbundenen Weiterverarbeitung für z.B. Entscheidungen anhand der Zughistorie verwendet.

Die `Rewards` sammeln die Punkte für jeden Spieler. Null Punkte werden ausgegeben, wenn das Spiel noch nicht vorbei oder unentschieden ist, ansonsten wird für z.B. Nullsummenspiele 1 Punkt für den Gewinner und -1 Punkt für den Verlierer ausgegeben (oder 2 Punkte für den Gewinner und -2 Punkte für den Verlierer usw.).

Der Abschlag (`Discounts`) behält den Faktor der Reduzierung des gewählten Zugs. Je später der Zug, desto geringer der Abschlagsfaktor und desto geringere Signifikanz des gewählten Zuges.

Der StepType Datentyp ist eine interne Variable zur Bestimmung, ob die Episode gerade erst angefangen hat(StepType.FIRST), noch am Laufen ist (StepType.MID) oder ob die Episode beendet ist(StepType.LAST).

2.5.5 Spiele

OpenSpiel unterstützt eine große Menge an n-Spieler Nullsummenspielen, kooperative Summenspiele, abwechselnd zugbasierte und Spiele mit gleichzeitigen Zügen, Spiele vollständiger und unvollständiger Information und noch viele weitere. Die volle Liste der unterstützten Spiele[4] zeigt auch die kontinuierliche Weiterentwicklung von Tests und die Erweiterung auf zusätzliche Spiele.

Da ein Großteil der Spiele aus OpenSpiel in C++ geschrieben sind, werden sie über pybind11[JRM17] als ein Pythonmodul offengelegt. In der Klasse `pyspiel.cc`[5] wird das Python Äquivalent der pyspiel Klasse definiert: alle erlaubten Methoden mit Argumenten und Rückgabewerten, objekt-eigene interne Variablen, Unterklassen wie zum Beispiel der Zustand `State` und die Behandlung von Fehlern.
Alle Methoden können bei einer Neudefinition einer Tochterklasse überladen werden und ein neues Verhalten implementiert bekommen; alle nicht überladenen Methoden werden beim Aufruf als virtuelle Methode mit übergebenen Parametern in C++ aufgerufen.

2.5.6 Algorithmen

Es ist eine große Anzahl von verschiedenen Algorithmenklassen implementiert. So sind Suchalgorithmen wie Minimax[KM75] und MCTS[Cou07], tabellarisch lernende Agenten

[4] https://github.com/deepmind/open_spiel/blob/master/docs/games.md, 21. Dezember 2022
[5] https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/pybind11/pyspiel.cc, 20. Oktober 2022

wie Q-Learning[WD92], sowie Einzel- und Multi-Agenten Algorithmen für das verstärkte Lernen umgesetzt. Aus der vollständigen Liste[6] kann ein fortschreitender Ausbau an Tests und die Schaffung weiterer Algorithmen gewährleistet werden.

Einfachere Algorithmen, die keine simulierte Umgebung zur Berechnung benötigen, können direkt mit einer Instanz des Spiels und eines dazugehörigen Zustandes agieren. Reinforcement Learning Algorithmen laufen dahingegen durch ein gegebenes Umfeld, ohne direkt auf ein Spiel zuzugreifen. In Abbildung 2 ist der Zusammenhang zwischen den einzelnen Klassen dargestellt.

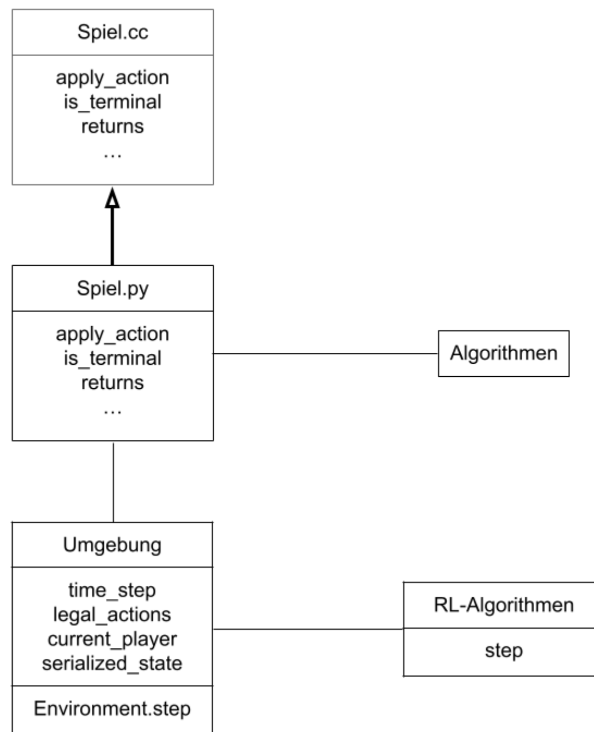


Abbildung 2: OpenSpiel Aufbau für RI

[6] https://github.com/deepmind/open_spiel/blob/master/docs/algorithms.md, 19.Dezember 2022

3 Umsetzung

Ziel des Projekts [7] ist es, ähnliche Ergebnisse der Lernalgorithmen zu erhalten, wenn nicht die Python Spiele-Definition, sondern die jeweilig gewünschten Prolog Spiele über die Brücke angesprochen werden, siehe Abbildung 3. Dazu werden auf der Prolog Seite die Logik und die Brücke voneinander getrennt. Es wird nur die Brücke von der Pythonseite angesprochen, um ein Spielobjekt zu erzeugen. Um die gleichen Umgebungen und Algorithmen anwenden zu können, benötigen die Brücke und passend die Spieleklasse einige Voraussetzungen, die zum Erstellen und Lernen gegeben sein müssen.

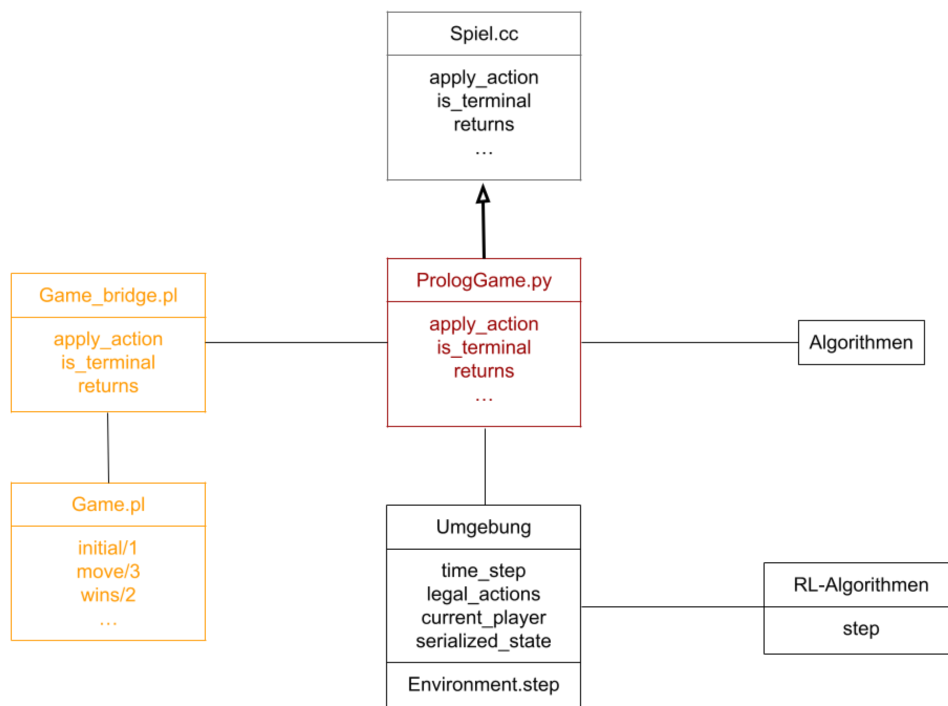


Abbildung 3: Prolog API für OpenSpiel

3.1 Spiel in Prolog

Es ist das Ziel, ein beliebiges Spiel in Prolog ansprechen zu können, welches schon in der Theorie unabhängig von der Sprache der Implementierung Algorithmen zum Lernen

[7] https://github.com/D4sherInc/Bachelor_Thesis, commit 4b5ee29

anwenden kann. Als Grundlage habe ich mir dazu Beispielumsetzungen einiger Spiele [8] sowie konkrete einzelne Spieldurchläufe (`Playthroughs / Trajectories`) [9] angeschaut. Anhand deren Umsetzung habe ich Schritt für Schritt die Anbindung an ein Prolog Äquivalent erstellt.

3.1.1 Spielelogik

Bestenfalls soll ein Spiel durch ein Anderes ersetzt werden können, um die gleichen ML-Agenten darüber laufen zu lassen, solange es sich um ein Spiel des gleichen Typs handelt. So sollte ein 2-Spieler-Nullsummenspiel ohne Zufallsknoten im Spielbaum (`Chance Nodes`) durch ein anderes 2-Spieler-Nullsummenspiel ohne Zufallsknoten ausgetauscht werden können und der Agent lernt genauso am zweiten Spiel wie auch am Ersten.
Da die Umsetzung einer Spielelogik aber alleine bei 2 Entwicklern gleich korrekt sein kann, sich aber schon die Bezeichnung von Predicates und Spielernamen unterscheiden kann, ist es an dieser Stelle nicht sinnvoll, eine einheitliche Grundlage zu erzwingen, um die Anbindung so modular halten zu können, wie es gewünscht wird. Daher habe ich mich hier dazu entschlossen, die Spielelogik unberührt zu lassen. Das Einzige, das der Datei noch angefügt wird, ist die Definition des Moduls wie in 7. Dies geschieht am Anfang der Prolog Datei. Der Name des Moduls muss dabei ein Atom sein, d.h. ein Name, beginnend mit einem kleinen Buchstaben.

Quellcode 7: Spiele Logik als Modul erklären

```
1: % muss als erste Nicht-Kommentar Zeile eingesetzt werden
2: % der Name des Moduls kann beliebig ausgewählt werden
3: :- module(game, []).
4: %
5: % hier beginnt die Spiellogik
```

Das definierte Spiel kann in der Anbindung nur angewendet werden, wenn in der Logik einige Voraussetzungen erfüllt sind. So muss einerseits der Startzustand abrufbar sein. Ein Zug muss auf einen Zustand anwendbar sein und einen neuen Zustand liefern. Es muss anhand des `States` berechenbar sein, ob es sich um einen Endzustand handelt. Der nächste Spieler muss berechenbar sein; dabei ist es egal, ob dies aus dem Zustand oder aus dem Spielzug geschieht.

[8] https://github.com/deepmind/open_spiel/tree/master/open_spiel/python/games, 18. Dezember 2022

[9] https://github.com/deepmind/open_spiel/tree/master/open_spiel/python/examples

3.1.2 Brücke für PySwip

Das Modul dient als Brücke zwischen der Spielelogik in Prolog und der Wrapperklasse in Python für OpenSpiel. Hier werden die essentiellen Abfragen von der Seite von OpenSpiel verwertbar an die Spielelogik übertragen und auf Prologseite ausgewertet und gesetzte Variablen und Wahrheitswerte auf der Python-Seite zurückgegeben.
Am Anfang der Datei muss jetzt die passende Spielelogik wie in 8 eingefügt werden, damit genau deren Logik aufgerufen werden kann.

Quellcode 8: Spiele Logik in Brücke einbinden

```
1: % muss als erste Nicht-Kommentar Zeile eingesetzt werden
2: % der Name des Moduls kann beliebig ausgewählt werden
3: :- use_module(game, []).
4: %
5: % hier beginnen die Interface Predicates
```

Damit ein Spielobjekt erst erzeugt werden kann, müssen Informationen über das Spiel angegeben werden. Da Prolog vom Ausbau der Sprache auf Fakten beruht, lohnt es sich, diese Eigenschaften hier zu speichern.
Die Werte werden hier in den Predicates `gametype/2` und `gameinfo/2` mit den Argumenten "Attributsname" und "Attributswert" gesetzt. In der Vorlage [10] sind alle Informationen angegeben, die OpenSpiel mindestens bei der Initialisierung als Argumente benötigt. Die restlichen Argumente haben Standard-Werte, die man aber an dieser Stelle auch noch hinzufügen kann. Alle setzbaren Werte sind in der `pyspiel`- API[11] zwischen C++ und Python definiert. Bei nicht unterstützten Attributswerten wirft die Spieleklasse bei der Initialisierung einen benutzerdefinierten Fehler mit der Fehlermeldung, welche Werte nur akzeptiert werden und welcher Wert übergegeben wurde. Damit die Wrapperklasse am Ende nur noch ein Ergebnis beim Abfragen nach den Eigenschaften hat, ist für beide Arten noch ein Sammelpredicate `getGameTypes/1` und `getGameInfos/1` definiert.
Die eigentliche Anwendung der Spielelogik geschieht in den Interface-Predicates, die beim Spielablauf namentlich aufgerufen werden. Da ich mich dazu entschieden habe, den Spielzustand nicht in Prolog zu speichern, hat sich das Problem ergeben, dass der aktuelle Spieler nicht immer aus z.B. dem Zustand des Spielfeldes ermittelbar ist. Es ist bei Tic-Tac-Toe durch Abzählen der gesetzten Felder pro Spieler und dem Wissen, welches Symbol der erste Spieler besitzt, möglich, sich den derzeitigen Spieler zu errechnen, bei einem simpleren Spielfeld ohne Spielerzeichenzuweisungen wie `Nim` aber nicht. Aufgrund dessen wird in dieser Anbindung der in Prolog angewendete Spielezustand als zusammengesetzter Term wie in 9 verwendet.

Quellcode 9: Spielzustand als zusammengesetzter Term

[10] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/prolog/blank_game_bridge.pl

[11] https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/pybind11/pyspiel.cc Zeile 130-190; Zeile 226-240

```
1: % am Beispiel von init/2:
2: init(GameState, _) :-
3:     GameState = [Current_Player, State],
4:     %...Rest der Logik von init/2 ...
5:     .
```

Im Spielzustand (GameState) wird der aktuelle Spieler sowie der Zustand des Spielfeldes (State) angegeben. Der Zustand des Spielfeldes kann sich dabei von Spiel zu Spiel unterscheiden, egal ob es sich dabei um ein 2-dimensionales Spielbrett in der Form von einer Liste aus Listen der passenden Größe oder nur um eine Integer-Wert handelt. Die Verarbeitung des States erfolgt in den weiteren Methoden. Bei allen Predicates dieser Brücke haben die Spielzustände den gleichen Aufbau, um alles konsistent zu halten.
Die Namen und Stelligkeiten (Arity) der Brücken-Predicates sind für jedes unterstützte Spiel gleich und in der Vorlage [12] als leere Schablone gegeben. Mit den gegebenen Voraussetzungen aus Absatz 3.1.1 lässt sich nun die Brückenlogik implementieren. Es handelt sich dabei um die Abfragen:

- `init/2`: der Initialzustand
- `legal_actions/2`: alle zulässigen Aktionen für den aktuellen Spieler
- `is_terminal/1`: Wahrheitswert, ob der Zustand ein Endzustand ist
- `returns/3`: die Punkte für einen Spieler

Für jedes Spiel muss an dieser Stelle die Logik des Spiels aufgerufen und ausgewertet werden. Beispielhaft wäre das für die Spieledefinition von Tic-Tac-Toe in Prolog [13] und die passende Brücke [14] in Quellcode 10 zu sehen.

[12] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/prolog/blank_game_bridge.pl
[13] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/prolog/tic_tac_toe.pl
[14] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/prolog/tic_tac_toe_bridge.pl

Quellcode 10: apply_action/2 Logik in tic_tac_toe_bridge.pl

```
1: :- use_module(tic_tac_toe, []).
2: % ...
3: % apply_action(+GivenGameState, +GivenAction, -NextGameState)
4: apply_action(GameState, Action, NewGameState) :-
5:     GameState = [Current_Player, Board],
6:     dif(Current_Player, none),
7:     player_ID_(Current_Player, Pl_Symbol),
8:     tic_tac_toe:move(Board, Pl_Symbol, Action, NewBoard),
9:     tic_tac_toe:other_player(Pl_Symbol, Pl2_Symbol),
10:    player_ID_(Next_Player, Pl2_Symbol),
11:    NewGameState = [Next_Player, NewBoard].
12: % ...
13: % Helper Predicate
14: % player_ID_(?In_OpenSpiel, ?In_Prolog_Game_Logic)
15: player_ID_(0, x).
16: player_ID_(1, o).
```

In OpenSpiel werden Spieler und Aktionen intern als Integer, beginnend bei 0, nummeriert und müssen gegebenenfalls für die vorgegebene Spielelogik einheitlich übersetzt werden. Im Beispiel wird versucht, auf einen gegebenen Zustand "GameState" eine Aktion anzuwenden und den nächsten Spielzustand zu vereinen (`unify`). Als einzige Zeilen werden `move/3` und `other_player/2` aus dem Game-Modul verwendet; das Zerlegen und Zusammenbauen der Spielzustände in o.g. Form geschieht hier.

Um die Namensgebung ebenfalls konsistent zu halten und damit die Brücke auch passend zum Spiel geladen wird, muss der Dateiname dem der Spielelogik entsprechen und noch ein `_bridge.pl` am Ende angehangen bekommen. Zum Beispiel:

- Spielelogik: **game.pl**
- Brücke: **game_bridge.pl**

3.2 Wrapper Klasse

Jedes Objekt einer Spieleklasse in Python, welche die Algorithmen und Umgebungen aus OpenSpiel verwenden möchte, muss entweder Objekt der pyspiel.Game Klasse in C++ oder ein API-kompatibles Objekt sein. Das wird in der Wrapperklasse (`Prolog_Game.py`) erfüllt, indem die Spieleklasse PrologGame von pyspiel.Game erbt.

3.2.1 Spiel Objekt

Diese Klasse bildet die Grundlage für die Anbindung auf der Python Seite. Ein erzeugtes Objekt dieser Klasse soll, wie auch in Python geschriebene Spieleklassen dieser Art in OpenSpiel, ein verwendbares Objekt zum Erstellen einer RL-Umgebung, zum Lernen für Agenten und zum Anwenden auf Algorithmen genutzt werden können.
Beim Erzeugen eines Objektes wird der übergebene Name als Parameter gesetzt und geladen. Unterstützte Spiele werden dabei als Klassenattribut in einer Liste gespeichert und müssen bei der Erweiterung auf weitere Spiele hier eingefügt werden. Bei Übergabe eines nicht erlaubten Namens wird ein Fehler geworfen. Hierbei wird die erste Abfrage über PySwip gestellt: Es soll die passende Prolog Datei über den Consult Befehl geladen werden, siehe Quellcode 11.

Quellcode 11: Prolog Spiele-Brücken-Datei laden

```
1: Prolog.consult("path/to/game_bridge.pl")
```

Jetzt können weitere Abfragen gestellt werden, die in der geladenen Prolog Datei verwendet werden.
Bevor aber der Konstruktor der Superklasse aufgerufen werden kann, benötigt dieser noch die GameType- und GameInfo-Attribute. Diese werden, wie in Absatz 3.1.2 beschrieben, in der Brückenklasse gespeichert und können über ein Sammelpredicate ausgemacht werden. Über eine interne Methode werden sie in einer Python Dictionary gesammelt und als Argumente für die pyspiel Unterklassen-Konstruktoren `pyspiel.GameType` und `pyspiel.GameInfo` verwendet. Wenn alle benötigten GameTypes und GameInfos über die Query erfolgreich gesammelt wurden, werden sie gesetzt. Falls ein Wert nicht erlaubt ist, wird ein benutzerdefinierter Fehler geworfen, siehe Absatz 3.2.3.
Als letzten Schritt werden das pyspiel.GameType Objekt, sowie das pyspiel.GameInfo Objekt als Argumente für den Konstruktor der Superklasse angewendet.
Dieses Objekt der Wrapperklasse kann nun zum Erstellen von Spielzuständen (GameStates) und Observern aufgerufen werden.

3.2.2 Spielzustand Objekt

Ein Spielzustand (`GameState`) wird aus Spieleklasse initialisiert. Dabei kann anfangs immer nur der Initialzustand des geladenen Spiels erzeugt werden.
Jedes Interface Predicate, das in der Brückenklasse definiert ist, erhält eine gespiegelte Methode der Klasse, die den gleichen Namen trägt, siehe Tabelle 3.

Eine Ausnahme ist die `current_player` Methode, die den aktuellen Spieler abrufen würde. Da die Spielzustände nicht auf der Prolog Seite gespeichert werden, kann auch nicht der aktuelle Spieler zuverlässig für die unterstützten Spiele abgefragt werden.

Tabelle 3: `PrologGameState`-Methoden

Methode	Rückgabewert
`__init__`	lädt den Initialzustand des Spiels
`__str__`	aktuellen Spielzustand in lesbarer Form
`current_player`	aktueller Spieler für den Zustand
`legal_actions(player)`	erlaubte Züge für den übergebenen Spieler
`is_terminal`	Wahrheitswert, ob das Spiel im Zustand beendet ist
`apply_action(move)`	wendet die übergebene Aktion auf den derzeitigen Zustand, und aktualisiert den Zustand
`get_next_player`	Spieler des nächsten Zuges
`returns`	Punktzahl für jeden Spieler, sortiert nach Spielernummer
`is_chance_node`	Wahrheitswert, ob der Zustand ein Zufallsknoten im Spielbaum ist
`action_to_string(player, move)`	Aktion des Spielers als lesbarer Zug

3.2.3 Hilfsmethoden

Bei einer PySwip Query können je nach Anfrage PySwip interne Datentypen zurückgegeben werden, die in dieser Form nicht ausgewertet werden können, siehe Tabelle 4. Diese Werte müssen erst wieder in Python Werte übersetzt werden. Die Werte lassen sich über die PySwip Klassen beziehungsweise über Python interne Datentypen übersetzen. Die interne Helfermethode **translate_from_prolog(values)** nimmt ein einzelnes Objekt oder ein Iterable Objekt und übersetzt rekursiv jedes Objekt, falls es nicht schon als ein akzeptierter Datentyp vorliegt. Solche Datentypen liegen bei den Anfragen an Prolog öfters vor, wenn nicht einzeilige Fakten, sondern weiterläufige Predicates aufgerufen werden, die auf der Prologseite noch mehr berechnen müssen. Das geschieht zum Beispiel bei den Interface-Predicates aus 3.2.2. Diesbezüglich kamen bisher aber nur die `pyswip.Atom` und Python `bytes` Datentypen vor.

Prolog Wert	Beispiel	Objekt nach Query	Beispiel
atom	atomname	pyswip.Atom	'Atom(123456)'
Variable	X	pyswip.Variable	'Variable(1234567)'
Funktor	pos(3,4)	pyswip.Functor	'Functor1234567'
Zeichenkette	'string'	bytes	b'string'

Tabelle 4: übersetzbare Werte aus Prolog

Bei der Initialisierung eines Spielobjektes müssen alle `pyspiel.GameType` und `pyspiel.GameInfo` Attribute gesetzt werden, die bei dem Aufruf des Konstruktors keinen

Standardwert haben. Sollten aber die übergebenen Werte aus den übergebenen Attributen der PySWIP Brücke unzureichend oder fehlerhaft gesetzt worden sein, wirft die `PrologGame`-Klasse beim Versuch einen **GameSettingsError**.
Dieser Error wirft den Fehler mit der Nachricht, von welchem Attribut welcher Wert erwartet wird und welcher falsche Wert übergeben wurde. So wird z.B. beim Attribut `dynamics` nur entweder der Wert `sequential`, `mean_field` oder `simultaneous` erwartet.

Nicht alle dieser Attribute können aber direkt aus der übersetzten Prolog-Representation der Wertebezeichnungen als Übergabewert eingesetzt werden. Pyspiel verlangt an mehreren Stellen dafür intern festgelegte konstante Zahlenwerte, die für die Prologseite zu unübersichtlich werden, wenn sie nur als solche Konstanten erwartet werden würden. Demzufolge werden alle diese gesammelten Attribute in der privaten Methode **_assign_game_attributes(game_types, game_infos)** systematisch nach pyspiel Vorgaben gesetzt und, falls nötig, ein `GameSettingsError` geworfen.

3.3 Observerklasse

Damit ein lernender Agent sich schrittweise in einer gegebenen Umgebung verbessern kann, benötigt er die Informationen zum Spielzustand zu jedem Schritt, den der Agent tätigt. Die benötigten Informationen werden aus dem Spielobjekt innerhalb der Umgebung gespeichert. Eine Information davon ist der `InfoState`, siehe Abbildung 1b. Dieser wird von einem Observer als Tensor erstellt. Ein Observer wird aus einer Spieleklasse erzeugt. Die Größe des Tensors sowie die Belegung des Spielfeldes ist für jedes Spiel jedoch spezifisch, abhängig von dem Spielaufbau. Eine gemeinsame Klassendefinition für einen genutzten Observer ist daher nicht sinnvoll.
Aus diesem Grund wird in der Spieleklasse für jedes unterstützte Spiel ein eigener Observer erstellt.

In der Umgebungsklasse aus Absatz 2.5.4 wird ein Spielobjekt bei der Initialisierung gespeichert. Aus diesem werden Spielzustände, beginnend beim Startzustand, geladen und aus diesen Zuständen deren Informationen abgerufen. Diese Informationen setzen sich nach einem Reset allerdings wieder zurück, sobald eine neue Lernepisode begonnen wird. Damit die Information nicht verloren geht, erstellt die Umgebung eine `Observation`, die es dem Agenten leichter macht, gespielte Züge auszuwerten.
Ein Observerobjekt kann aus einem Spielobjekt erstellt werden. Dabei wird der Name des Spiels bei der Initialisierung des Spielobjektes selbst abgespeichert. Dadurch kann immer nur ein passender Observer erstellt werden.
Bei der Initialisierung des Observers bekommt das Objekt eine interne Variable, die den Tensor zur Überwachung des Spiels speichert. Dieser hat solch eine Größe, dass jeder mögliche Zustand, aus "neutraler" und aus Sicht jedes Spielers eindeutig erzeugt werden kann. In Python existiert dafür die Python Bibliothek `Numpy` [HMW+20], die effizient einen solchen Tensor erzeugen und editieren kann.

Zuerst wird ein Tensor der gewünschten Größe erzeugt und mit Nullen gefüllt. Die Größe des Tensors ist bei den bisher unterstützten Spielen 3-dimensional: Anzahl der Spieler + 1, Anzahl der Reihen und Anzahl der Spalten des Spielfeldes. Bei einem einfacheren 1-dimensionalen Spiel wie `Nim` ist die Reihenanzahl 1; die übrig gebliebenen Stäbe sind alle "in einer Spalte gestapelt". Die Einträge werden im Tensor dann an den Stellen gesetzt, wo jeweils noch ein Feld frei oder ein Feld eines Spielers belegt ist.

4 Fallstudien

Um die Performance dieser Anbindung zu testen, habe ich mich dazu entschlossen, ein erfolgreich geladenes Spiel an verschiedenen Algorithmen schrittweise auszuprobieren, bevor ich es in eine Umgebung lade und verschiedene Agenten darüber lernen lasse. Die folgenden Tests sind zuerst mit der Prolog Definition und Brücke von Tic-Tac-Toe durchgeführt worden.

4.1 Funktionstest

4.1.1 Minimax mit Alpha-Beta-Pruning

Der Minimax-Algorithmus [KM75] [RN09] 1 ist einer der effektivsten Algorithmen für 2 Spieler-Spiele. Beiden Spielern wird eine Rolle zugewiesen: ein minimierender (`Min`) und ein maximierender Spieler (`Max`). Beide Spieler spielen abwechselnd und gegeneinander; das bedeutet für den Spielbaum, dass sich die Knotenebenen spielerweise abwechseln. Jede Entscheidung gibt ein skalares Ergebnis anhand einer Bewertung von Punkten. Ziel für jeden Spieler ist es, mit der Entscheidung eines Zuges seine Punktzahl zu minimieren beziehungsweise zu maximieren. Bei der Auswahl des Zuges bedeutet das, dass der maximierende Spieler bei einer Auswahlmöglichkeit den Zug mit den höchsten Punkten auswählt und der minimierende Spieler den Zug mit den niedrigsten Punkten.

Das Alpha-Beta-Pruning ist das "Abschneiden" und "Nicht-erkunden" von bisher unerforschten Abzweigen, wenn bereits bekannt ist, dass der erforschte Wert nicht besser für den aktuellen Spieler sein kann, als ein anderer Zug, den Minimax bisher schon kennt. Je nach Entscheidung handelt es sich dabei um entweder einen α-Cut oder β-Cut.
In Abbildung 4 hat der Spieler Max beispielsweise den linken Abzweig mit der Punktzahl $+1$ ausgewertet, was für ihn in diesem Knoten jetzt der Mindestwert wäre. Da aber der Spieler Min im Elternknoten bereits einen Wert hat, der geringer sein wird, als der minimale Wert von Max, wird es für den Elternknoten keinen Unterschied machen, welches Ergebnis beim rechten Abzweig von Max entsteht, da sich Min nicht für diesen Abzweig entscheiden wird. Damit kann Minimax sich die Berechnung sparen und den gesamten Abzweig ignorieren. Dieser Vorgang beschreibt einen α-Cut.

Algorithmus 1 Minimax mit Alpha-Beta Pruning

```
function ALPHABETA(Node, depth, alpha, beta, isMaximizingPlayer)

    if depth is 0 or state is terminal then
        return heuristic value of state
    end if
    if maximizing player then
        value = −∞
        for all child of node do
            value :=max(value, alphabeta(child, depth -1, α, β, FALSE))
            if value > β then
                break                                          ▷ β-Cut
            end if
            α := max(α, value)
        end for
        return value
    else
        value = +∞
        for all child of node do
            value :=min(value, alphabeta(child, depth -1, α, β, TRUE))
            if value < α then
                break                                          ▷ α-Cut
            end if
            β := min(β, value)
        end for
        return value
    end if
```

In Openspiel ist Minimax[15] als Python-Algorithmus bereits implementiert. Dieser erwartet das Spiel. Optional sind ein genauer Spielzustand, eine Bewertungsfunktion zur Berechnung der Belohnungswerte, falls die Suche zu weit geht, die maximale Suchtiefe und welcher Spieler Max ist.
Wenn beide Spieler perfekt nach dem Minimax-Theorem spielen, ist der Wert für Min und Max im Startzustand 0. Keiner der Spieler will dem jeweils anderen einen höheren Wert zusprechen als es nötig ist. Das heißt, dass bei perfektem Spiel keiner der beiden Spieler jemals gewinnt oder verliert. Wenn jetzt aber einer der beiden Spieler durch einen Spieler ersetzt wird, der eine andere Strategie verfolgt, kann dieser maximal nicht besser sein als der ursprüngliche Spieler. Der neue Spieler kann aber mehr Fehler begehen und damit die Berechnung des Minimax in späteren Spielzuständen beeinflussen.

[15] https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/algorithms/minimax.py, 18. Dezember 2022

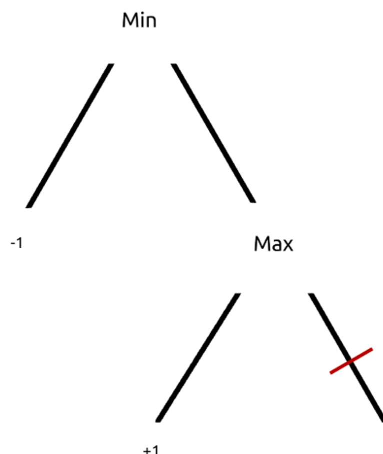


Abbildung 4: Minimax: ein α-Cut

Um Minimax an der Brücke auszutesten[16], habe ich zuerst beide Spieler nach Minimax ihren Zug entscheiden lassen; 1000 mal Max und 1000 mal Min als Spieler mit dem ersten Zug. Nach 2000 Episoden hat jedes der Spieldurchläufe in einem Unentschieden geendet.
Als zweiten Test habe ich den minimierenden Spieler durch einen zufällig auswählenden Spieler ausgetauscht. Hier sollte es für den maximierenden Spieler ebenfalls nicht zu einer Niederlage kommen, aber durch die fehlerbehaftete zufällige Auswahl aus den erlaubten Zügen sollte es dem maximierenden Spieler schon ab dem ersten Fehltritt möglich sein, seinen Sieg zu versichern. Diese Konstellation habe ich ebenfalls über 2000 Episoden laufen lassen und es hat sich gezeigt, dass der Spieler Max bei mehreren Durchläufen zwischen 1790 und 1820 mal gewinnt aber weiterhin 0 mal verliert.
Für Minimax zeigt sich, dass die Anbindung funktioniert und der Algorithmus alle benötigten Werte aus dem Spielzustands- und Spielobjekt abrufen kann.

4.1.2 MCTS: Monte Carlo Tree Search

Monte Carlo Tree Search [BPW+12] ist ein Suchalgorithmus, der versucht, den optimalen Zug für einen Knoten zu ermitteln, in dem es eine einstufige Breitensuche mit Random-Sampling des weiteren Spielverlaufs kombiniert. Ausgewertet wird dabei immer nur ein zuende gespielter Durchlauf, nicht jeder einzelne Zug.
Die grundlegende Idee hinter MCTS besteht darin, dass von einem Zustand viele Spieldurchläufe bis zum Spielende simuliert werden, um deren Ergebnisse bei der Auswahl für den optimalen Zug zu verwenden. Dabei wird in jedem Zug diejenige Aktion ausgewählt,

[16] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/python/Prolog_Minimax_test.py

die aus den Simulationen die höchsten erwarteten Belohnungen geschätzt erreicht.
Der Algorithmus geht dabei wie in 5 in vier Schritten vor:

- `Select`: Von einem gegebenen Spielbaum wird ein Knoten nach einer vorgegebenen Strategie ausgewählt. Dies kann zufällig sein, ein Knoten, der bisher weniger erforscht ist, als alle anderen dieser Ebene oder eine Kombination der beiden. Der Wurzelknoten dieses Baums stellt den aktuellen Zustand dar; die Tochterknoten alle möglichen Züge aus diesem Zustand.
- `Expand`: Von dem ausgewählten Knoten werden alle möglichen Tochterknoten ermittelt und an allen diesen Knoten Spieldurchläufe berechnet.
- `Simulate`: Aus den Tochterknoten werden jetzt vollständige Spieldurchläufe bis zum Ende des Spiels simuliert. Dafür werden eine Reihe an Regeln verwendet, um das Ergebnis zu bestimmen. Anhand der gesammelten Ergebnisse wird ein Wert für den aktuellen Knoten geschätzt, der im Schnitt in diesem Knoten erreicht werden kann.
- `Back-Up`: Sobald alle Züge bewertet wurden, werden die Informationen an Elternknoten zurückgegeben und es wird der optimale Zug ausgewählt.

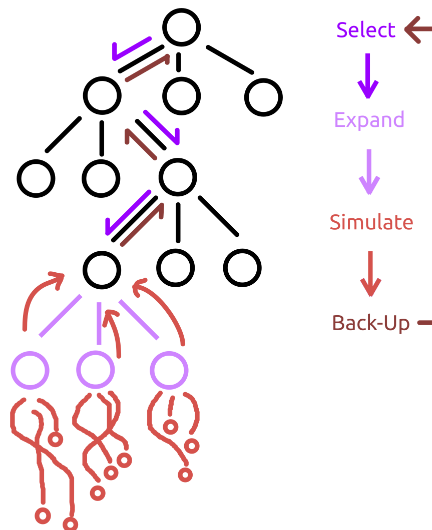


Abbildung 5: Monte Carlo Tree Search

In OpenSpiel [LLL+19] ist der MCTS-Algorithmus[17] ebenfalls implementiert. Ein großer Unterschied ist, dass die finale Auswahl des nächsten Zuges abhängig von der Häufigkeit

[17] `https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/algorithms/mcts.py`, 30. November 2022

der Auswahl dieses Zuges bei den Simulationen bestimmt wird: der Zug, der bei den Simulationen am häufigsten verwendet wurde, wird als nächster Knoten ausgewählt.
Anhand des vorgegebenen Beispiels [18] habe ich die Anbindung ausgetestet. Die Spieledefinition von `pyspiel.load_game(game_name)` habe ich durch ein Objekt der `Prolog_Game` meiner Anbindung ersetzt. Durch Flags können der Spielname, die Art eines Spielers (z.B. menschlich, MCTS oder zufällig auswählend), die Anzahl an gespielten Durchläufen, Konstanten zur Initialisierung von MCTS Bots und noch weitere Einstellungen zum Debuggen gesetzt werden.
Für den Test werden 1000 Züge simuliert, bevor ein Zug ausgewählt wird. Ein Ergebnis wird 5 mal von 200 Episoden ausgewertet.
Auch hier hat sich ergeben, dass zwei gegeneinander spielende MCTS-Agenten ein Unentschieden für fast immer garantieren können. Ein MCTS Agent kann gegen einen zufälligen Spieler jedoch ab dem ersten Fehltritt durch eine zufällige Auswahl eines schlechten Zuges den Sieg garantieren. Nur durch die Auswahl aller guten Züge für den zweiten Spieler kann dieser eine Niederlage verhindern, jedoch auch nicht gewinnen.

4.2 Lerntest

Um den Lerneffekt eines Agenten auf einem Prolog Spiel zu testen, habe ich mich dazu entschieden, zwei Varianten des Q-Learners[WD92] in OpenSpiel zu laden. Zu erwarten ist ein annähernd identischer Lerneffekt über die gleichen Lernperioden, wenn alles außer die Spieledefinition identisch ist.

4.2.1 Umgebung

Damit der Q-Learner eine Umgebung kennen lernen kann, braucht er dazu eine in Abschnitt 2.5.4 beschriebene Instanz der Umgebungsklasse. Diese nimmt ein Spielobjekt als Argument entgegen.

4.2.2 Q-Learner

Ein Q-Learner [WD92] ist ein Lernalgorithmus aus dem Bereich des verstärkten Lernens, der Anhand einer gegebenen Umgebung den bestmöglichen Zug für einen gegeben Zustand in der Umgebung ermitteln möchte. Die sogenannten **Q-Values** werden für jedes Zustand-Aktions-Paar berechnet, welche die erwarteten zukünftigen Belohnungen beschreiben, die der Spieler durch die Auswahl des Zuges in diesem Zustand am Ende erhalten wird. Ziel des Q-Learners ist es, eine Strategie für die beste Auswahl an Zügen zu finden.
Der Algorithmus funktioniert, indem er schrittweise die Umgebung erkundet und sich

[18] `https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/examples/mcts.py`, 30. November 2022

die gegangenen Pfade und deren resultierenden Belohnungen merkt. Folgende Formel 1 beschreibt das aktualisieren eines Q-Werts:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

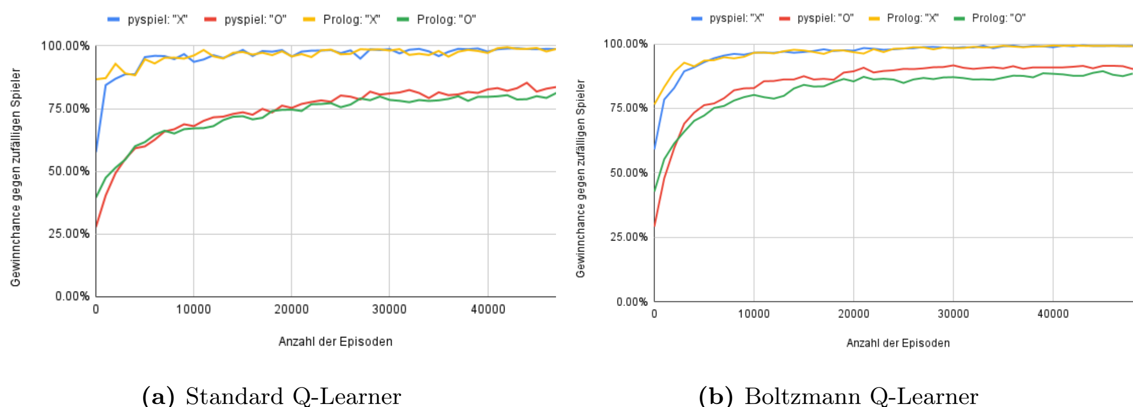
$Q(s,a)$ beschreibt den Q-Wert für den aktuellen Zustand s mit der Auswahl des Zuges a. r ist die Belohnung für das Erreichen des nächsten Zustands s'. Die Lernrate α beinflusst, wie viel Einfluss ein neuer Wert für die Q-Tabelle hat: je höher α, desto mehr lernt der Agent. Der Abschlag (`discount`) γ beschreibt, wieviel Wert ein Zug über den Verlauf verliert. Je später eine Belohnung gefunden wird, desto geringer ist sie.
Eine Abwandlung von diesem Algorithmus ist der Boltzmann Q-Learner[KG18]. Der Unterschied liegt darin, dass sich der Q-Learner hier nicht für den höchsten Q-Wert für einen der erlaubten Züge entscheidet, sondern die Auswahl über die Boltzmann-Wahrscheinlichkeit 2, mithilfe der Temperatur T des Systems, getroffen wird.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \sum_{s' \in S} T(s,a,s') \max_{a' \in A} Q(s',a') - Q(s,a) \right] \tag{2}$$

Für $T \rightarrow 0$ handelt der Boltzmann Q-Learner "greedy"wie der normale Q-Learner und wählt den höchsten Q-Wert. Für $T \rightarrow \infty$ wählt der Agent dagegen zufällig.

Ich habe beide Instanzen des Algorithmus getestet und die von OpenSpiel geschriebene Spielklasse für Tic-Tac-Toe sowie eine Prologdefinition des Spiels angewendet. Alle Durchläufe wurden auf dem gleichen Gerät durchgeführt. Aus den gesammelten Daten[19] habe ich einen Durchschnitt berechnet und ausgewertet.
Ein Agent hat 50000 Episoden in der Umgebung gelernt. Nach jeweils 1000 Episoden wurde der Agent in weiteren 2000 Episoden ausgewertet ohne weiter zu lernen. In dieser Zeit wurde ihm ein Spieler als Gegner zugewiesen, der seinen Zug zufällig aus den erlaubten Zügen auswählt; 1000 mal hat der lernende Agent als erster und 1000 mal als zweiter Spieler agiert.
Dieser Vorgang wurde für beide Q-Learner Agenten fünf mal durchgeführt und deren Mittelwerte in der Abbildung 6 graphisch dargestellt.

[19] `https://github.com/D4sherInc/Bachelor_Thesis/blob/master/game%20logs/Q%20Learner%20WinRates%20-%20Boltzmann%20Q-Learner.csv`



(a) Standard Q-Learner

(b) Boltzmann Q-Learner

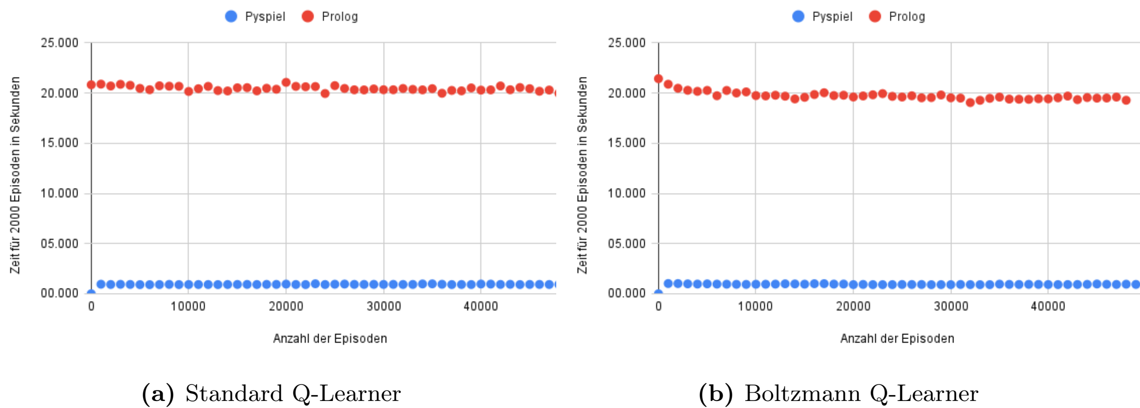
Abbildung 6: Lernraten der verschiedenen Q-Learner: OpenSpiel Implementation und die Prolog API

In der Grafik lässt sich schnell erkennen, dass die beiden Spieler bereits vor Beendigung der 50000 Lernepisoden eine Art Grenzwert erreichen. Die Spieler der ersten Position erreichen beide annähernd eine Gewinnchance von 100%, die Spieler der zweiten Position etwa eine Gewinnchance von 82%.
Auffallend ist hier, dass jeweils die Kurven für den gleichen Spieler in den beiden verschiedenen Spielen sehr nah beieinander und im Falle des ersten Spielers nahezu deckungsgleich sind.
Daraus lässt sich schlussfolgern, dass die Lernalgorithmen sowohl für die bereits getestete Spieledefinition sowie auch für die Prolog Spiele annähernd identisch gut lernen. Hiermit konnte gezeigt werden, dass die Schnittstelle für dieses Beispiel wie erwartet funktioniert.

4.3 Performancetest

Mit den selben Tests für den Lerneffekt habe ich auch die Zeit gemessen, die OpenSpiel und die Anbindung jeweils benötigen, um 3000 Durchläufe zu produzieren. Dabei habe ich die Systemzeit auf Millisekunden ausgeben lassen. Parallel zum Lerntest habe ich die Zeiten über 5 Durchläufe aufgenommen und einen Mittelwert der Versuche aufgezeichnet.
In erster Betrachtung ist zu erwarten, dass die Schnittstelle langsamer läuft, da bei jeder Anfrage Datentypen übersetzt werden müssen.
Das zeigt sich auch deutlich in der Messung in Abbildung 7: Die optimierte Anbindung von Pythonspielen an C++ benötigt pro 3000 Episoden im Schnitt weniger als eine Sekunde (∅ 0.935 Sekunden für Standard Q-Learner, ∅ 0.926 Sekunden für Boltzmann Q-Learner) Berechnungszeit. Die Prolog API hingegen benötigt pro 3000 Episoden über 20 Sekunden (∅ 20.457 Sekunden, ∅ 19,744 Sekunden) Berechnungszeit.

Mit dieser erhöhten Laufzeit wurde das gewünschte Benchmark-Ergebnis nicht erreicht, da sie doch höher ausgefallen ist als erwartet. Weitere Analysen sollten dazu beitragen, den technischen Engpass zu finden und die API zu optimieren.



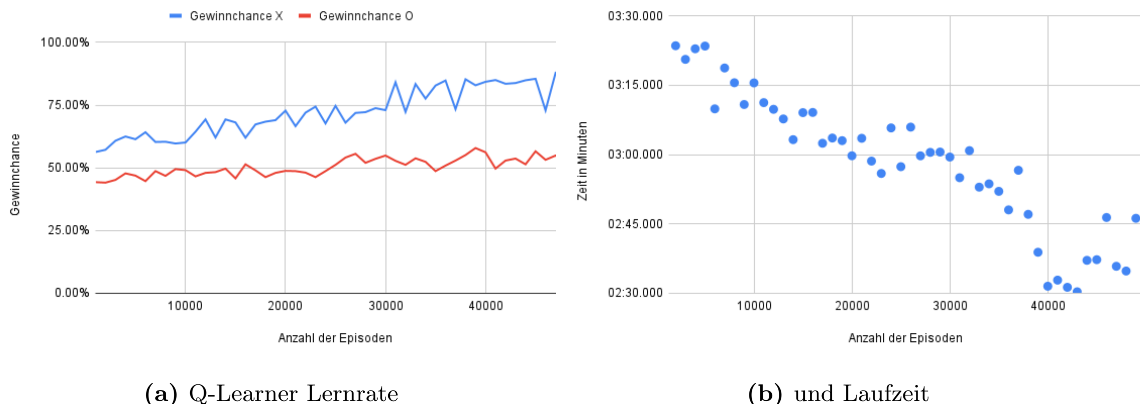
(a) Standard Q-Learner

(b) Boltzmann Q-Learner

Abbildung 7: Performance der verschiedenen Q-Learner: OpenSpiel Implementation und die Prolog API

4.4 Erweitert auf weitere unterstützte Spiele

Auf die gleiche Art und Weise habe ich ein weiteres Spiel ausgetestet: Connect4 [20]. Für Connect4 mit einem Spielfeld von 6 Reihen und 7 Spalten habe ich den Q-Learner mit den gleichen Ausgangswerten wie aus Abschnitt 4.2 und 4.3 verwendet. Bei einmaligen Durchlauf sind die Ergebnisse aus Abbildung 8a entstanden.



(a) Q-Learner Lernrate

(b) und Laufzeit

Abbildung 8: Performance des Boltzmann Q-Learners bei Connect4 mit der Prolog API

Hier ist zu sehen, dass der Q-Learner über 50000 Episoden als erster Spieler langsam einen Lerneffekt erzielt, dieser aber nicht stabil ist. Da der Spielraum von Connect4 so viele Felder besitzt, explodiert die Anzahl aller möglichen Spieldurchläufe kombinatorisch relativ schnell. Der Lerneffekt von Q-Learning basiert auf bereits erforschten Spieldurchläufen. Daraus lässt sich schließen, dass auch der Lerneffekt bei Standardwerten eher schleppend

[20] https://github.com/D4sherInc/Bachelor_Thesis/blob/master/src/prolog/connect4.pl, ursprünglich von https://github.com/rvinas/connect-4-prolog, 18. Dezember 2022

verläuft, da der Spielraum nur langsam vollständig ausgelernt wird.
Bei der Performance, siehe Abbildung 8b sieht man, dass sich mit erhöhter Lernkurve die Laufzeit pro 3000 Episoden verringert. Das lässt sich vermutlich daraus schließen, dass bereits gelernte Spielabläufe vermehrt in späteren Evaluationen vorkommen und zu einer schnelleren Auswahl eines nächsten Zuges führen. Die Laufzeit ist anfangs bei über 3 Minuten pro 3000 Episoden.

5 Zukünftige Arbeiten

In diesem Abschnitt werden Bereiche für die weitere Forschung erwähnt, die auf meinem Ergebnis aufbauen könnten.

5.1 Erweiterung

Die Anbindung ist bisher auf eine Untergruppe an Spielen begrenzt. Mit den gegebenen Grundlagen, welche die Prolog Spiele einbinden, lassen sich zukünftig aber weitere Spieledefinitionen anbinden. Die Bausteine dafür sind in der Erstellung der Brückenklasse aus Abschnitt 3.1.2 gelegt; die Spieleinformationen müssen für ein neues Spiel eingetragen werden und die Predicates zur Anbindung müssen für die Spiele definiert werden. Zusätzliche Spieleinformationen, die von OpenSpiel mit Standardwerten gesetzt werden, können dabei auch bei den bisher existierenden Brückendateien nachträglich angefügt werden.

5.2 Speichern in Prolog

Ein großer Eckpunkt der Recherche war, dass die PySwip.Prolog Instanz als Singleton-Klasse es nicht erlaubt, zwei Instanzen des Interpreters gleichzeitig laufen zu lassen, ohne dass sie miteinander interagieren. Durch den Versuch, den aktuellen Spielzustand abzuspeichern und nach jedem Schritt zu aktualisieren, wirft die Anbindung einen Fehler, wenn zwei Instanzen nacheinander auf den gleichen Zustand zugreifen wollen. Dieser ist für den ersten Aufruf vorhanden, wird bei diesem verändert und neu gespeichert. Der zweite Aufruf versucht ebenfalls auf den ersten Zustand zuzugreifen und findet ihn nicht. Das ist vor allem beim Q-Learning passiert.
In der weiteren Forschung kann man noch erkunden, wie die Schnittstelle reagiert, wenn nicht nur der aktuelle Zustand gespeichert wird, sondern auch bei einem Schritt der neue Zustand hinzugefügt wird, ohne den alten zu aktualisieren.

5.3 Error Handling

Da sich Python und Prolog im Behandeln von Laufzeitfehlern ähneln, ist es möglich, die Fehler nur auf einer Seite laufen zu lassen. So ist es möglich, in Prolog geworfene Fehlermeldungen als Pythonfehler abzufangen und weiterzuverarbeiten. Beispiele dafür wären Fehler für fehlende Predicates, die erwartet werden oder ein undefiniertes Modul.

6 Fazit

In dieser Bachelorarbeit war es das Ziel, eine Anbindung der Algorithmen aus OpenSpiel an eine bisher noch nicht angebundene Programmiersprache zu entwickeln. Durch die Verwendung einer starken API zwischen Prolog und Python ist es mir gelungen, Lernalgorithmen mit in Prolog definierten Spielen lernen zu lassen.
Es hat sich ergeben, dass trotz der Funktionaltät für einen Kernbereich an Anwendungen die Performance noch ein Hindernis bildet. Grund dafür können die Übersetzungen von Prolog- zu Python-Datentypen bei jeder Abfrage sein. Ein weiterer einwirkender Faktor könnte die Hardware sein, die zum Testen verwendet wurde, welche nicht für solche Berechnungen optimiert ist. Zusätzlich könnte das Nicht-speichern auf der Prolog Seite zu erhöhten Laufzeiten führen, wodurch trotz vorhandener und gespeicherter Ergebnisse für einen Zustand die Ergebnisse von Prolog dennoch neu berechnet werden.
Insgesamt hat diese Arbeit einen Anteil zur Forschung beigetragen, indem es die Umsetzbarkeit zeigt, Python Algorithmen mit Prolog Logik zu verbinden. Während es noch Raum zur Verbesserung an Stellen der Laufzeit gibt, zeigt diese Arbeit einen wichtigen Schritt nach vorne und ermöglicht weitere und neue Möglichkeiten zur Forschung im Bereich künstlicher Intelligenz für die Programmierung und Auswertung neuer Spiele.

Abbildungsverzeichnis

1 Tic Tac Toe Spielzustand im InfoState 11
2 OpenSpiel Aufbau für RI 13
3 Prolog API für OpenSpiel 14
4 Minimax: ein α-Cut 24
5 Monte Carlo Tree Search 25
6 Lernraten der verschiedenen Q-Learner: OpenSpiel Implementation und die Prolog API 28
7 Performance der verschiedenen Q-Learner: OpenSpiel Implementation und die Prolog API 29
8 Performance des Boltzmann Q-Learners bei Connect4 mit der Prolog API . . 29

Tabellenverzeichnis

1 Überblick über einige Python Datentypen 6
2 Time Step Werte 11
3 `PrologGameState`-Methoden 20
4 übersetzbare Werte aus Prolog 20

Algorithmenverzeichnis

1 Minimax mit Alpha-Beta Pruning 23

Quellcodeverzeichnis

1 Liste Beispiel 3
2 einfaches Prolog Beispiel 4
3 PySwip installieren 7
4 eine Query von Python an Prolog erstellen 8
5 OpenSpiel installieren 10
6 OpenSpiel zur Entwicklung installieren 10
7 Spiele Logik als Modul erklären 15
8 Spiele Logik in Brücke einbinden 16
9 Spielzustand als zusammengesetzter Term 16
10 apply_action/2 Logik in tic_tac_toe_bridge.pl 18
11 Prolog Spiele-Brücken-Datei laden 19

Literatur

[Alp14] ALPAYDIN, Ethem: *Introduction to Machine Learning*. The MIT Press, 2014. – ISBN 0262028182

[BPW+12] BROWNE, Cameron B. ; POWLEY, Edward ; WHITEHOUSE, Daniel ; LUCAS, Simon M. ; COWLING, Peter I. ; ROHLFSHAGEN, Philipp ; TAVENER, Stephen ; PEREZ, Diego ; SAMOTHRAKIS, Spyridon ; COLTON, Simon: A Survey of Monte Carlo Tree Search Methods. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), Nr. 1, S. 1–43. http://dx.doi.org/10.1109/TCIAIG.2012.2186810. – DOI 10.1109/TCIAIG.2012.2186810

[Bra90] BRATKO, Ivan: *PROLOG Programming for Artificial Intelligence*. 2nd. USA : Addison-Wesley Longman Publishing Co., Inc., 1990. – ISBN 0201416069

[Cou07] COULOM, Rémi: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: HERIK, H. J. d. (Hrsg.) ; CIANCARINI, Paolo (Hrsg.) ; DONKERS, H. H. L. M. (. (Hrsg.): *Computers and Games*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–75538–8, S. 72–83

[CR96] In: COLMERAUER, Alain ; ROUSSEL, Philippe: *The Birth of Prolog*. New York, NY, USA : Association for Computing Machinery, 1996. – ISBN 0201895021, 331–367

[CWA+22] CARLSSON, Mats ; WIDEN, Johan ; ANDERSSON, Johan ; ANDERSSON, Stefan ; BOORTZ, Kent ; NILSSON, Hans ; SJÖLAND, Thomas: *SICStus Prolog User's Manual*. Bd. 4.7.1. Kista, Sweden : Swedish Institute of Computer Science, 2022

[HMW+20] HARRIS, Charles R. ; MILLMAN, K. J. ; WALT, Stéfan J. ; GOMMERS, Ralf ; VIRTANEN, Pauli ; COURNAPEAU, David ; WIESER, Eric ; TAYLOR, Julian ; BERG, Sebastian ; SMITH, Nathaniel J. ; KERN, Robert ; PICUS, Matti ; HOYER, Stephan ; KERKWIJK, Marten H. ; BRETT, Matthew ; HALDANE, Allan ; RÍO, Jaime F. ; WIEBE, Mark ; PETERSON, Pearu ; GÉRARD-MARCHANT, Pierre ; SHEPPARD, Kevin ; REDDY, Tyler ; WECKESSER, Warren ; ABBASI, Hameer ; GOHLKE, Christoph ; OLIPHANT, Travis E.: Array programming with NumPy. In: *Nature* 585 (2020), September, Nr. 7825, 357–362. http://dx.doi.org/10.1038/s41586-020-2649-2. – DOI 10.1038/s41586–020–2649–2

[JRM17] JAKOB, Wenzel ; RHINELANDER, Jason ; MOLDOVAN, Dean: *pybind11 – Seamless operability between C++11 and Python*. 2017. – https://github.com/pybind/pybind11

[KG18] KIANERCY, Ardeshir ; GALSTYAN, Aram: Dynamics of Boltzmann Q-Learning in Two-Player Two-Action Games. In: *USC Information Sciences Institute* 2 (2018), October

[KM75] Knuth, Donald E. ; Moore, Ronald W.: An Analysis of Alpha-Beta Pruning. In: *Artificial Intelligence* 6 (1975), S. 293–326

[LBH15] LeCun, Yann ; Bengio, Yoshua ; Hinton, Geoffrey: Deep learning. In: *Nature* 521 (2015), Nr. 7553, S. 436–444

[LLL+19] Lanctot, Marc ; Lockhart, Edward ; Lespiau, Jean-Baptiste ; Zambaldi, Vinicius ; Upadhyay, Satyaki ; Pérolat, Julien ; Srinivasan, Sriram ; Timbers, Finbarr ; Tuyls, Karl ; Omidshafiei, Shayegan ; Hennes, Daniel ; Morrill, Dustin ; Muller, Paul ; Ewalds, Timo ; Faulkner, Ryan ; Kramár, János ; Vylder, Bart D. ; Saeta, Brennan ; Bradbury, James ; Ding, David ; Borgeaud, Sebastian ; Lai, Matthew ; Schrittwieser, Julian ; Anthony, Thomas ; Hughes, Edward ; Danihelka, Ivo ; Ryan-Davis, Jonah: OpenSpiel: A Framework for Reinforcement Learning in Games. In: *CoRR* abs/1908.09453 (2019). http://arxiv.org/abs/1908.09453

[RN09] Russell, Stuart ; Norvig, Peter: *Artificial intelligence*. 3. Upper Saddle River, NJ : Pearson, 2009

[SB18] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. MIT Press, 2018

[SOM+20] Stein, Sebastian ; Ochal, Mateusz ; Moisoiu, Ioana-Adriana ; Gerding, Enrico ; Ganti, Raghu ; He, Ting ; La Porta, Tom: Strategyproof Reinforcement Learning for Online Resource Allocation. In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC : International Foundation for Autonomous Agents and Multiagent Systems, 2020 (AAMAS '20). – ISBN 9781450375184, S. 1296–1304

[SS94] Sterling, Leon ; Shapiro, Ehud: *The Art of Prolog (2nd Ed.): Advanced Programming Techniques*. Cambridge, MA, USA : MIT Press, 1994. – ISBN 0262193388

[SSJ18] Shailaja, K. ; Seetharamulu, B. ; Jabbar, M. A.: Machine Learning in Healthcare: A Review. In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018, S. 910–914

[Tc20] Tekol, Yüce ; contributors: *PySwip v0.2.10*. https://github.com/yuce/pyswip. Version: 2020

[VR+07] Van Rossum, Guido u. a.: Python programming language. In: *USENIX annual technical conference* Bd. 41, 2007, S. 1–36

[VRD10] Van Rossum, Guido ; Drake, Fred L.: *The Python Language Reference*. Amsterdam, Netherlands : Python Software Foundation, 2010

[WD92] Watkins, Christopher J. ; Dayan, Peter: Q-learning. In: *Machine learning* 8 (1992), Nr. 3-4, S. 279–292

[WSTL12] Wielemaker, Jan ; Schrijvers, Tom ; Triska, Markus ; Lager, Torbjörn: SWI-Prolog. In: *Theory and Practice of Logic Programming* 12 (2012), Nr. 1-2, S. 67–96. – ISSN 1471–0684