Integration von Stable Baselines3 Agenten in das GBG-Framework: Eine Java-Python-Schnittstelle zur Nutzung von Gymnasium und SB3 im GBG

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science
im Studiengang Informatik
an der Fakultät Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Leon Fynn Püschel
Matrikel-Nr.:	11126132
Adresse:	Mannsfelder Str. 88 50968 Köln leon_fynn.pueschel@smail.th-koeln.de leon.pueschel@gmx.de
eingereicht bei:	Prof. Dr. Wolfgang Konen
Zweitgutachter:	Prof. Dr. Boris Naujoks

Gummersbach, 06.06.2025

Abstract

Diese Arbeit befasst sich mit der Integration von Deep-Reinforcement-Learning-Agenten des Python-Frameworks *Stable-Baselines3* (SB3) in das Java-basierte *General-Board-Game*-Framework (GBG). Das GBG stellt sowohl Agenten zur Verfügung, die auf *Reinforcement Learning* (RL) basieren, als auch viele Brettspiele, an denen die Agenten trainiert und getestet werden können. Jedoch implementiert das GBG bisher keine Agenten, die mithilfe von *Deep Learning* – also tiefen künstlichen neuronalen Netzen – trainiert werden. Ziel ist es, den Umfang des GBGs über seine bisherigen RL-Algorithmen hinaus mit modernen Deep-RL-Algorithmen von SB3 zu erweitern.

Die zentrale Herausforderung lag im technologischen Bruch zwischen Java und Python, der eine bidirektionale Kommunikationsschnittstelle erforderlich machte. Ein praktischer Test zeigte, dass eine HTTP-basierte Schnittstelle deutlich performanter ist als eine Implementierung mit Py4J, insbesondere bei größeren Datenmengen. Daraufhin wurde die Schnittstelle über HTTP realisiert.

Die Implementierung umfasst die SB3-Python-Anwendung und die entsprechenden Erweiterung des GBGs, um die SB3-Agenten DQN, PPO und Maskable PPO zu integrieren. Dazu werden die Spiele des GBGs über eine *Gymnasium*-kompatible Schnittstelle für SB3 zugänglich gemacht.

Funktionstests mit den SB3-Agenten DQN, PPO und Maskable PPO in Spielen wie Tic-Tac-Toe, Nim und Vier gewinnt bestätigten die erfolgreiche Integration und die Fähigkeit der Agenten, intelligente Strategien zu erlernen und auch gegen die bereits vorhandenen GBG-Agenten zu bestehen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die fortschreitende Entwicklung im Bereich des Machine Learnings, insbesondere des Reinforcement Learnings (RL), eröffnet neue Möglichkeiten zur Lösung komplexer Probleme, wie sie beispielsweise in Brettspielen auftreten, wie sie das General-Board-Game-Framework implementiert. Das General-Board-Game-Framework (GBG) ist ein in Java geschriebenes Framework zur Erstellung von Brettspielen und Agenten, deren Ziel es ist, die Brettspiele möglichst erfolgreich zu spielen.

Das GBG implementiert bereits viele Arten von Agenten, bisher beschränken sich die RL-Algorithmen mit neuronalen Netzen jedoch nur auf kleine, flache neuronale Netze. Dies liegt unter anderem daran, dass das GBG in Java geschrieben wurde. Java wird selten benutzt, um tiefe neuronale Netze (Deep Learning) zu implementieren. Entweder müsste man diese selbst implementieren oder mit weniger bekannten Frameworks arbeiten, die weniger leistungsstark sind.

In der Regel wird heutzutage Python für Deep-Learning-Projekte verwendet. Für Python gibt es bereits viele mächtige Frameworks zum Erstellen und Trainieren von tiefen neuronalen Netzen, wie TensorFlow oder PyTorch. Außerdem gibt es weitere Frameworks, die besonders interessant für Reinforcement Learning und somit für den Einsatz im GBG sind, zum Beispiel Gymnasium, welches ein standardisiertes Interface für RL-Environments bereitstellt, oder Stable Baselines3 (SB3), das viele Agenten für Deep Reinforcement Learning implementiert.

Mit dem Einsatz solcher Python-Frameworks könnte man also aktuelle Deep-Learning-Frameworks sowie Deep-RL-Agenten nutzen. Jedoch muss dafür zunächst Kommunikation zwischen dem GBG, das in Java geschrieben wurde, und Python durch eine bidirektionale Schnittstelle hergestellt werden.

Dieser Zielsetzung folgend erstellt das Softwareprojekt, das in dieser Bachelorarbeit aufgearbeitet wird, eine bidirektionale Schnittstelle, die spezifisch für den Einsatz von SB3-Deep-RL-Agenten mithilfe von Gymnasium im GBG geeignet ist. Dazu wird eine neue Anwendung erstellt, die dem GBG eine Schnittstelle anbietet, um auf das SB3-Framework bzw. auf dessen RL-Algorithmen und -Agenten zuzugreifen. Diese ist in Python geschrieben und beinhaltet SB3, weshalb diese Anwendung im Folgenden SB3-Python-Anwendung genannt wird.

Des Weiteren wird das GBG erweitert, um auf die Schnittstelle der SB3-Python-Anwendung und somit auf die RL-Algorithmen von SB3 zugreifen zu können. Gleichzeitig muss auch das GBG eine Schnittstelle für die SB3-Python-Anwendung bereitstellen, damit die RL-Algorithmen auf die Spiele im GBG zugreifen können, um ihre Agenten trainieren zu können. Deshalb ist die Schnittstelle bidirektional.

In dieser Arbeit wird dieses Softwareprojekt und dessen Erstellung erläutert. Dazu werden zunächst die theoretischen Grundlagen des Reinforcement Learnings sowie in dieser

Arbeit verwendete Begriffe des Reinforcement Learnings erläutert, da für die Entwicklung einiger Teile der Software das Wissen über die Grundlagen von Reinforcement Learning Voraussetzung ist.

Als Nächstes werden die verwendeten Softwarekomponenten von Dritten erläutert, welche Rolle diese in diesem Projekt spielen und ggf. warum sie für dieses Projekt ausgewählt wurden. Zuerst wird auf die Komponenten eingegangen, die zur Nutzung von Deep-RL-Agenten notwendig sind, wie etwa SB3 und Gymnasium, und danach auf die Komponenten, die zur Erstellung der bidirektionalen Schnittstelle erforderlich sind. Hier wird durch den Vergleich zwischen HTTP und Py4J bereits eine Frage aufgegriffen, die im späteren Verlauf der Arbeit auch noch einmal praktisch betrachtet wird, und zwar: Welche Technologie – HTTP oder Py4J – eignet sich für dieses Projekt besser, um die bidirektionale Schnittstelle zu implementieren?

Im nächsten Kapitel wird die Problemstellung konkretisiert und daraus abgeleitet konkrete Anforderungen an die Schnittstelle und ihren Aufbau gestellt. Um herauszufinden, welche der beiden Technologien – HTTP oder Py4J – besser für die Anforderungen geeignet ist, wird ein praktischer Test durchgeführt. Die Ergebnisse dieses Tests zeigen klar, dass sich HTTP besser eignet, weshalb die Schnittstelle mit HTTP implementiert wurde.

Ein wesentlicher Teil der Arbeit widmet sich dem Aufbau und der Implementierung der Schnittstelle, wobei die Architektur der SB3-Python-Anwendung und die Integration in das GBG-Framework dokumentiert und Architekturentscheidungen begründet werden.

Zum Schluss wurden Funktionstests durchgeführt, die die generelle Funktionstüchtigkeit der Schnittstelle und der integrierten SB3-Agenten innerhalb des GBGs darlegen sollen. Dazu wurden die integrierten SB3-Agenten mithilfe des GBGs trainiert und wichtige Werte, die den Erfolg des Trainings sowie zeitliche Aspekte betreffen, gesammelt. Zudem wurden Testspiele zwischen den SB3-Agenten und den Agenten des GBGs in ausgewählten Spielen durchgeführt, um ihren Erfolg untereinander zu vergleichen und Referenzwerte für die neu integrierten SB3-Agenten zu schaffen.

2 Reinforcement Learning

In diesem Kapitel werden die grundlegenden Aspekte des Reinforcement Learnings (RL) erläutert. Sie bilden die Basis zum Verständnis dafür, warum die beiden Anwendungen bzw. die Schnittstelle im weiteren Verlauf der Arbeit so gestaltet wurden, wie sie sind. Für die Implementierung der Schnittstelle sind zwar keine tiefgehenden Fachkenntnisse über Reinforcement Learning oder spezielle RL-Algorithmen erforderlich, jedoch ist ein gewisses Grundlagenwissen essenziell. Wie bei jedem Softwareprojekt sollte vorab ein fundierter Überblick über die zugrunde liegende Domäne gewonnen werden, um eine fundierte Architekturentscheidung treffen zu können. Darüber hinaus unterstützt das

Wissen über RL-Algorithmen dabei, Trainingsergebnisse besser zu interpretieren und Hyperparameter sinnvoll einzustellen.

Reinforcement Learning ist ein Teilgebiet des Machine Learnings. Dabei stehen zwei Hauptinstanzen im Mittelpunkt: der Agent und das Environment (die Umgebung). Der Agent ist eine agierende Instanz, die mit dem Environment durch Aktionen interagiert (siehe Abbildung 1). Das Environment wiederum besteht aus einer Menge an Zuständen (States) und Regeln, die festlegen, wie sich diese Zustände durch die Aktionen des Agenten verändern. Übertragen auf ein Brettspiel entspricht der Spieler dem Agenten und das Spiel selbst dem Environment. Das Environment definiert außerdem, welche Zustände besonders erstrebenswert sind – zum Beispiel ein Sieg in Tic-Tac-Toe.

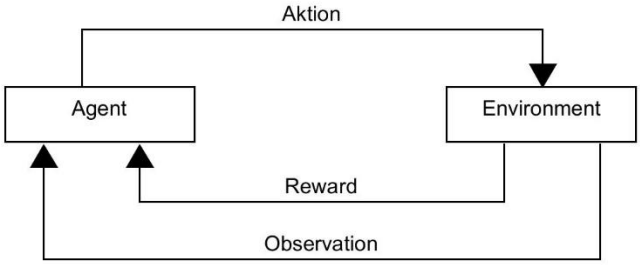


Abbildung 1: Interaktionen zwischen dem Environment und Agenten.

Ein RL-Algorithmus untersucht das Environment selbstständig und entwickelt auf Basis gesammelter Erfahrungen eine Handlungsstrategie (Policy), die dem Agenten vorgibt, wie er optimal agieren sollte, um möglichst häufig erstrebenswert Zustände zu erreichen.

Genauer gesagt wählt der Agent anhand der Policy in jedem Zeitschritt eine Aktion, die auf das Environment einwirkt. Daraufhin erhält er ein Feedback, das aus einer Belohnung (Reward) und der neuen Observation des Environments besteht. Ziel des RL-Algorithmus ist es, die Policy so zu optimieren, dass der Agent langfristig möglichst viele positive Rewards aus dem Environment erhält.

RL-Algorithmen sind daher in der Lage, komplexe Problemstellungen zu lösen, beispielsweise solche, wie sie in Brettspielen auftreten.

In der Regel werden die Environments in RL-Problemen mithilfe des mathematischen Modells des Markov Decision Process (MDP) modelliert oder mit einer abgewandelten Form davon.

Im Folgenden werden die zentralen Komponenten und Begriffe des Reinforcement Learnings erläutert. Anschließend erfolgt eine kurze Vorstellung der in dieser Arbeit verwendeten RL-Algorithmen.

2.1 Reinforcement Learning Begriffe

Aktion: Eine Handlung des Agenten im Environment. Unternimmt der Agent eine Aktion, gelangt er dadurch von einem State (Zustand) in den nächsten.

Aktionsraum: Die Menge aller Aktionen, die ein Agent in einem Environment ausführen kann, nennt man Aktionsraum. Ein Aktionsraum ist diskret, wenn sich die Aktionen in klaren Kategorien darstellen und somit in einer endlichen Menge ganzer Zahlen ausdrücken lassen. Ein kontinuierlicher Aktionsraum wird durch einen Vektor reeller Zahlen dargestellt. Alle Spiele im GBG besitzen jedoch einen diskreten Aktionsraum.

State: Die vollständige Beschreibung des Zustands des Environments zu einem bestimmten Zeitpunkt. Den ersten State des Environments nennt man Start-State (Anfangszustand), und den letzten Zustand Terminal-State (Endzustand).

Observation: Die Beschreibung des Environments, die der Agent nach jeder Aktion im Environment erhält. Die Beschreibung kann entweder vollständig oder unvollständig sein. Bei einer vollständigen Observation stehen dem Agenten alle Informationen über den aktuellen Zustand (State) des Environments zur Verfügung – die Observation entspricht dann dem State. Bei einer unvollständigen Observation kann der Agent nur einen Teil des States beobachten. Es wird nicht immer klar zwischen den Begriffen State und Observation unterschieden, und auch in dieser Arbeit kann je nach Kontext State auch für Observation stehen.

Observationsraum: Die Menge aller möglichen Observationen, die ein Agent in einem Environment beobachten kann. Dieser ist bei vielen Spielen gleich dem Zustandsraum, sofern der Agent alle Informationen des States beobachten kann.

Reward: Nach einer Aktion des Agenten im Environment erhält der Agent einen Reward (Belohnung), der beschreibt, wie positiv oder negativ sein neuer State zu bewerten ist. Beim Spiel Tic-Tac-Toe könnte dem Agenten in einem Terminal-State z. B. eine 0 für eine Unentschieden, -1 für eine Niederlage oder eine 1 für einen Sieg übergeben. So wird es auch für die meisten Spiele im GBG gehandhabt.

Step: Ein Step bezeichnet einen diskreten Zeitschritt im Environment. Ein Step besteht aus einer Interaktion (einer Aktion) des Agenten mit dem Environment. In diesem Projekt beinhaltet ein Step auch die darauffolgenden Interaktionen möglicher gegnerischer Agenten mit dem Environment, bis der zu trainierende Agent wieder an der Reihe ist.

Episode: Beschreibt den gesamten Verlauf von einem Start-State bis zu einem Terminal-State in einem Environment. Bei einem Brettspiel entspricht dies einer Spielrunde oder Partie.

Transition: Ein Tupel, das den Übergang von einem State in einen neuen beschreibt. Es besteht aus dem alten State, der Aktion, dem Reward, dem neuen State und der Information, ob das Spiel terminiert ist.

Return: Beschreibt den kumulierten Reward, der sich aus einer Sequenz von States und darauffolgenden Aktionen ergibt. Diese Sequenz nennt man auch Trajectory. Es gibt zwei Arten von Returns: Der erste beschreibt die Summe der Rewards einer endlichen Sequenz von Steps, der zweite die einer unendlichen Sequenz. Damit diese Summe konvergiert, wird jeder Reward mit einem sogenannten Discount-Faktor gewichtet. Der Return R für eine unendliche Sequenz von Steps wird durch folgende Formel, mit r: Reward, γ: Discount, t: Step, beschrieben:

$$R = \sum_{t=0}^{\infty} \gamma^t r_t$$

Policy: Die Policy beschreibt die Regeln, nach denen ein Agent entscheidet, welche Aktion er auf Basis der letzten Observation ausführen soll. Eine optimale Policy maximiert den erwarteten kumulierten Reward (Return). Ziel des Reinforcement Learning ist es also, die optimale Policy zu finden oder sich ihr zumindest anzunähern. Die verschiedenen RL-Agenten erreichen dies auf unterschiedliche Weise. Die Policy wird bei policy-based Deep-RL-Algorithmen durch ein neuronales Netz approximiert, das eine Wahrscheinlichkeitsverteilung über mögliche Aktionen erzeugt.

Value Function: Die Value Function (Zustandswertfunktion) gibt den erwarteten Return in einem State s an, wenn der Agent einer bestimmten Policy folgt. Die optimale Value Function gibt den erwarteten Return an, den man erhält, wenn man der optimalen Policy folgt.

Q-Function: Die Q-Function ist eine spezielle Form der Value Function. Sie gibt den erwarteten Return für eine bestimmte Aktion in einem bestimmten State an, vorausgesetzt, der Agent folgt anschließend einer bestimmten Policy. In value-based Deep-RL-Algorithmen wird die Q-Function mithilfe eines neuronalen Netzes approximiert.

Die Bellman-Gleichung: Die Bellman-Gleichung besagt, dass die Value Function dem erwarteten Reward für die aktuelle Aktion entspricht, zuzüglich des mit dem Discount-Faktor gewichteten erwarteten Returns im nächsten Zustand. Sie stellt somit eine rekursive Definition der Value Function dar.

2.1.1 Markov Decision Process

Ein Markov Decision Process (MDP) modelliert einen stochastischen, sequenziellen Entscheidungsprozess [1]. Dabei hängt das Ergebnis einer Aktion nur vom aktuellen

Zustand ab, nicht jedoch von vorherigen Zuständen oder Aktionen (Markov-Eigenschaft). Ein MDP wird durch ein 5-Tupel $\langle S, A, R, P, \rho_0 \rangle$ definiert, wobei:

- S die Menge der States (Zustände) ist,
- A die Menge der möglichen Aktionen ist,
- R die Reward-Funktion ist, die einem Zustand und einer Aktion einen entsprechenden Reward (Belohnung) zuweist,
- P das Übergangsmodell ist, das die Wahrscheinlichkeit beschreibt, mit der ein Agent durch Ausführung einer bestimmten Aktion von einem Zustand in einen anderen übergeht,
- ρ_0 die Wahrscheinlichkeitsverteilung über die Startzustände beschreibt, also den Zustand, in dem eine Episode beginnt.

2.2 Reinforcement-Learning-Algorithmen

In diesem Abschnitt werden die in diesem Projekt verwendeten RL-Algorithmen beschrieben und begründet, warum sie für die Integration in das GBG ausgewählt wurden. Alle in diesem Projekt eingesetzten RL-Algorithmen werden über das Framework Stable-Baselines3 (SB3) bereitgestellt. Dabei handelt es sich konkret um folgende Agenten: Deep Q-Network (DQN) [2], Proximal Policy Optimization (PPO) [3], [4] sowie Maskable PPO (MPPO) [5].

Wie alle Algorithmen aus Stable-Baselines3 handelt es sich um modellfreie RL-Algorithmen. Das bedeutet, dass sie kein explizites Modell des Environments verwenden oder Zugriff darauf haben. Zusätzlich sind es Deep-RL-Algorithmen – sie nutzen tiefe neuronale Netze, um entweder die optimale Policy (Policy-Based-Algorithmus) oder die optimale Value Function (Value-Based-Algorithmus) zu approximieren. Policy-basierte Algorithmen haben den Vorteil, dass ihre Policy stochastisch ist, also eine Wahrscheinlichkeitsverteilung über mögliche Aktionen darstellt. Diese Verteilung soll angeben, wie wahrscheinlich es ist, dass eine bestimmte Aktion zu einem hohen Return führt. Value-basierte Algorithmen hingegen wählen stets die Aktion, die den höchsten erwarteten kumulierten Reward verspricht (Greedy Policy). Die Policy ist damit deterministisch, was eine sorgfältige Balance zwischen Exploration (Ausprobieren neuer Aktionen) und Exploitation (Ausnutzen bekannter, guter Aktionen) notwendig macht.

2.2.1 DQN

Deep Q-Network (DQN) ist ein value-based Algorithmus, der darauf abzielt, die optimale Q-Function zu approximieren. Der Algorithmus wurde für dieses Projekt ausgewählt, da er weit verbreitet ist. Das liegt unter anderem daran, dass er zu den einfacheren Deep-RL-Algorithmen zählt, gleichzeitig aber robuste Ergebnisse liefert.

Ein DQN-Agent sammelt sogenannte Transitions, also Erfahrungen bestehend aus Zustand, Aktion, Reward und Folgezustand, und speichert sie in einem Replay Buffer. Bei der Interaktion mit dem Environment wählt der Agent die Aktion, die laut aktueller Q-

Function im gegebenen Zustand den höchsten erwarteten Return ergibt. Diese deterministische Strategie verhindert allerdings anfangs die ausreichende Erkundung des Environments. Um dem entgegenzuwirken, wird mit einer gewissen Wahrscheinlichkeit (Epsilon) eine zufällige Aktion gewählt – bekannt als Epsilon-Greedy Policy. Zu Beginn des Trainings liegt Epsilon meist bei 1 und wird mit zunehmender Trainingsdauer schrittweise reduziert.

Das Lernen erfolgt in regelmäßigen Abständen anhand eines zufällig ausgewählten Batches von Transitions aus dem Replay Buffer. Ziel ist es, mithilfe eines neuronalen Netzes eine möglichst präzise Approximation der optimalen Q-Function zu erreichen. Hierzu wird das Netz, bzw. die aktuelle Q-Function, mithilfe des Gradientenabstiegsverfahrens und der Backpropagation so optimiert, dass die mittlere quadratische Abweichung zwischen den sogenannten aktuell geschätzten Q-Werten und den berechneten Q-Targets minimiert wird. Für jede Transition werden der alte State, die ausgeführte Aktion, der erhaltene Reward und der neue State verwendet, um den entsprechenden Q-Wert sowie das Q-Target zu berechnen. Der Q-Wert entspricht dem erwarteten Return für die gegebene Aktion im alten State gemäß der aktuellen Q-Function. Das Q-Target besteht aus dem tatsächlich erhaltenen Reward sowie dem rabattierten, von der Q-Function geschätzten Return des nächsten States und entspricht somit der Bellman-Gleichung.

Für die Berechnung des Q-Targets wird häufig ein separates neuronales Netz verwendet, das sogenannte *Target Network*. Dessen Parameter werden in regelmäßigen Abständen mit denen der aktuellen Q-Function synchronisiert, um die Korrelation zwischen Vorhersage und Zielwert zu reduzieren und so die Stabilität des Trainings zu erhöhen.

Der Replay Buffer trägt ebenfalls zur Stabilisierung des Trainings bei. Anstatt ausschließlich die zuletzt gesammelte Transition für das Training zu verwenden, greift der Algorithmus auf ältere Transitions zurück. Dies unterbricht die zeitliche Korrelation zwischen den Trainingsdaten, die ansonsten zu instabilen Lernprozessen führen könnte. Aus diesem Grund gehört DQN zur Klasse der Off-Policy-Algorithmen. Solche Algorithmen zeichnen sich dadurch aus, dass sie auch Erfahrungen nutzen können, bei denen die Aktionen nicht gemäß der aktuell verwendeten Policy ausgewählt wurden. In der Regel sind Off-Policy-Algorithmen dadurch sample-effizienter, da sie vergangene Erfahrungen mehrfach wiederverwenden können.

2.2.2 PPO

PPO ist ein Deep-RL-Algorithmus, der zur Klasse der sogenannten Policy Gradient Methods (PGM) gehört. Er basiert auf früheren Ansätzen wie REINFORCE, verbessert jedoch insbesondere die Sample-Effizienz, was ihn trotz seiner Zugehörigkeit zur Gruppe der On-Policy-Algorithmen sehr leistungsfähig macht.

On-Policy-Algorithmen zeichnen sich dadurch aus, dass die Erfahrungen, die sie zum Lernen verwenden, von der aktuellen Policy gesammelt wurden. Dadurch sind sie in der

Regel stabiler im Training als Off-Policy-Algorithmen. Beim DQN-Algorithmus, einem Off-Policy-Algorithmus, zum Beispiel, stammen die Erfahrungen im Replay Buffer von einer alten, vergangenen Policy. Zwar sind Off-Policy-Algorithmen in der Regel sample-effizienter, da sie vergangene Daten mehrfach verwenden können, sie erfordern jedoch komplexere Mechanismen zur Stabilisierung des Trainings.

Für dieses Projekt wurde PPO als einer der in SB3 verfügbaren Algorithmen ausgewählt, da er in der Praxis weit verbreitet ist und sich durch Robustheit, einfache Implementierung und stabile Lernverläufe auszeichnet. Diese Eigenschaften erleichtern insbesondere das Hyperparameter-Tuning, ohne dabei nennenswerte Leistungseinbußen gegenüber komplexeren Algorithmen hinzunehmen.

Der PPO-Algorithmus wechselt während des Trainings zwischen zwei Phasen:

1. **Erfahrungsphase**: Der Agent sammelt neue Erfahrungen aus seiner Interaktion mit der Umgebung. Diese werden in Trajectories gespeichert, die aus Zuständen, Aktionen, Rewards und nächsten Zuständen bestehen.
2. **Updatephase**: Basierend auf diesen gesammelten Erfahrungen werden die Gewichte der neuronalen Netzwerke aktualisiert. Dabei werden nur die gesammelten Daten der letzten Erfahrungsphase benutzt.

Im Gegensatz zu Value-Based-Algorithmen wie DQN verwendet PGMs ein neuronales Netzwerk, das direkt eine Wahrscheinlichkeitsverteilung über mögliche Aktionen in einem gegebenen Zustand schätzt – das sogenannte Policy Network. Diese Verteilung gibt an, mit welcher Wahrscheinlichkeit eine bestimmte Aktion zum höchsten erwarteten Return führt, wenn der Agent anschließend weiterhin der aktuellen Policy folgt. Während des Trainings werden die Aktionen stochastisch ausgewählt, was bedeutet, dass nicht immer dieselbe Aktion in einem bestimmten Zustand ausgeführt wird. Diese stochastizität ermöglicht eine natürliche Balance zwischen Exploration und Exploitation, ohne dass zusätzliche Strategien wie beispielsweise Epsilon-Greedy erforderlich sind.

Zusätzlich zum Policy Network nutzt PPO ein zweites neuronales Netz: das Value Network. Dieses approximiert den erwarteten Return eines gegebenen Zustands unter der aktuellen Policy, also die sogenannte State-Value Function. Das Value Network wird in der Updatephase zur Berechnung der Advantage Function verwendet, die aussagt, wie gut eine gewählte Aktion im Vergleich zu anderen möglichen Aktionen in demselben Zustand ist. Die Advantage Function ergibt sich durch Subtraktion des vom Value Network geschätzten Returns vom tatsächlichen, rabattierten Return der Trajectory.

Das Policy-Update bei PGMs, also die Anpassung der Gewichte im Policy-Netzwerk, erfolgt über eine sogenannte Objective Function, die mithilfe des Gradientenaufstiegsverfahrens maximiert wird. Diese Funktion multipliziert die vom Policy Network geschätzte Wahrscheinlichkeit für den aktuellen State und Aktion aus dem Tupel der Trajectories mit der Advantage Function. Auf diese Weise wird ein

Gradientenvektor berechnet, der die Wahrscheinlichkeit der gewählten Aktion im gegebenen State erhöht. Da dieser Gradient anschließend mit dem Wert der Advantage Function multipliziert wird, führt ein positiver Advantage-Wert dazu, dass die Wahrscheinlichkeit für die bestimmte Aktion gesteigert wird, während ein negativer Wert sie dementsprechend reduziert. Zudem bestimmt der Betrag des Advantage-Werts die Stärke des Gewichtsupdates.

Die Besonderheit des PPO-Algorithmus besteht darin, dass er dazu entwickelt wurde, über mehrere Epochen hinweg mit denselben, zuvor gesammelten Erfahrungen zu trainieren, was die Sample-Effizienz erhöht. Um dabei zu große Gewichtsanpassungen zu vermeiden – die durch das wiederholte Verwenden derselben Daten entstehen und das Training destabilisieren könnten – verwendet PPO die sogenannte Clipped Surrogate Objective Function als Objective Function. Diese verhindert durch die Verwendung der Clip- und Min-Funktionen ein Policy-Update, wenn dieses im Verlauf einer Updatephase zu übermäßig großen Gewichtsanpassungen führen könnte. Dazu wird die Wahrscheinlichkeit einer Aktion unter der aktuellen Policy ins Verhältnis zur Wahrscheinlichkeit derselben Aktion unter der alten Policy gesetzt. Wenn diese Abweichung zu groß wird und ein weiteres Gewichtsupdate die Abweichung noch vergrößern würde, sorgt die Clip-Funktion dafür, dass der Gradient auf null gesetzt wird – ein Gewichtsupdate findet in diesem Fall nicht statt.

2.2.3 Ungültige Aktionen

Sowohl den PPO- als auch den DQN-Algorithmus betrifft das Problem, wie man damit umgeht, wenn der Agent eine ungültige Aktion auswählt – zum Beispiel, wenn der Agent bei Tic-Tac-Toe ein Kreuz setzen möchte, obwohl an dieser Stelle bereits ein Kreis vom Gegner gesetzt wurde.

In diesem Projekt wurde sich für folgende Herangehensweisen entschieden, falls der Agent eine ungültige Aktion ausführen will:

- Falls der Agent während des Trainings eine ungültige Aktion ausführen möchte, wird das Spiel vom Environment beendet und der Agent erhält den maximal negativen Reward. Dadurch soll der Agent lernen, diese ungültigen Aktionen in Zukunft zu vermeiden.
- Falls der Agent im tatsächlichen Einsatz, also nach dem Training, eine ungültige Aktion vorschlägt, wird stattdessen die nächstbeste Aktion – also die Aktion mit dem nächsthöchsten Q-Wert bzw. der höchsten Wahrscheinlichkeit – gewählt. Sollte auch diese ungültig sein, wird wiederum die nächstbeste Aktion gewählt, und so weiter. In diesem Sinne wird die Policy nach dem Training so angepasst, dass keine ungültigen Aktionen mehr ausgegeben werden.

Es gibt auch andere Ansätze, um dieses Problem zu lösen. Die hier gewählte Methode ist jedoch relativ einfach umzusetzen und stellt in vielen Fällen eine valide

Herangehensweise dar, die gute Ergebnisse erzielt. Im nächsten Abschnitt über Maskable PPOs wird ein Algorithmus vorgestellt, der dieses Problem als eine seiner Kernaufgaben betrachtet.

2.2.4 Maskable PPO

Eine Frage, die sich beim Reinforcement Learning stellt, ist, wie man mit Aktionen aus dem Aktionsraum umgeht, die in einem bestimmten State nicht zur Verfügung stehen und dementsprechend ungültig sind. Ein Ansatz ist es, beim Auftreten ungültiger Aktionen den Agenten mit einem hohen negativen Reward zu bestrafen. Jedoch hat sich gezeigt, dass dieser Ansatz nicht für Environments mit großen Aktionsräumen und vielen ungültigen Aktionen skaliert [5].

Für dieses Projekt wurde MPPO ausgewählt, da einige der im GBG implementierten Spiele viele Zustände aufweisen, in denen der Großteil der Aktionen ungültig ist. Maskable PPO bzw. Action Masking für Policy-Gradient-Methoden verfolgt einen anderen Ansatz zur Lösung dieses Problems.

Hierbei wird eine sogenannte Mask-Funktion in die Policy integriert, die die Ausgabewerte des Policy Networks für ungültige Aktionen auf einen sehr niedrigen negativen Wert setzt. Danach wird, wie üblich, die Softmax-Funktion verwendet, um die rohen Outputs in eine Wahrscheinlichkeitsverteilung über alle Aktionen abzubilden.

Zudem werden durch die Softmax-Funktion die sehr niedrigen Wert ungültiger Aktionen nahezu vollständig unterdrück – die resultierenden Wahrscheinlichkeiten liegen bei digitalen Werten bei Null. Da die Maskierung Bestandteil der Policy und somit auch der Objective Function ist, werden ungültige Aktionen während des Trainings nicht in die Gewichtsanpassungen des Policy Networks einbezogen, da der Gradient in Bezug auf ungültige Aktionen dadurch ebenfalls null wird.

2.3 Self-Play

Beim Trainieren eines Agenten in Mehrspieler-Spielen stellt sich im Reinforcement Learning oft die Frage, gegen welche Gegner der Agent trainiert werden soll. Eine Möglichkeit wäre es, andere Agenten zu verwenden, die das Environment bereits „gelöst“ haben, also darin schon gut performen. Dies hat jedoch Nachteile:

1. Das Training ist von anderen Agenten abhängig. Für Environments, für die noch keine Agenten entwickelt wurden, funktioniert diese Herangehensweise natürlich nicht.
2. Die gegnerischen Agenten könnten so stark sein, dass der zu trainierende Agent nicht effektiv lernen kann, da er in jeder Episode verliert und so keine neuen, vielversprechenden Strategien entwickeln kann. Diese Strategien würden möglicherweise schon früh unterbrochen oder verworfen, bevor sie ausreichend erforscht wurden – wodurch der Agent keinen Anreiz hätte, sie weiterzuverfolgen.

3. Der Gegner könnte zu schwach sein, was dazu führen würde, dass der Agent suboptimale Strategien lernt, die zwar gegen diesen Gegner erfolgreich sind, aber nicht allgemein funktionieren.

Diese Probleme treten beim Self-Play nicht auf, da während des Trainings gegen eine Policy des zu trainierenden Agenten selbst gespielt wird, entweder gegen eine vergangene oder gegen die aktuelle Version. Dadurch, dass der Agent gegen eine eigene Policy spielt, kann sichergestellt werden, dass der Gegner nicht zu stark oder zu schwach ist. Der Gegner befindet sich also stets auf einem ähnlichen Leistungsniveau wie der zu trainierende Agent.

In diesem Projekt wird jede Episode mit einer bestimmten Wahrscheinlichkeit gegen die aktuellste gespeicherte Policy gespielt und ansonsten gegen eine ältere Policy. Wie oft gegen die aktuelle Policy gespielt wird und aus welchem Zeitfenster vergangene Policies ausgewählt werden, sind einstellbare Hyperparameter.

Zusammengefasst soll das Speichern mehrerer vergangener Policies und deren zufällige Auswahl einen robusteren Agenten hervorbringen, der besser gegen verschiedene Spielweisen bestehen kann.

3 Eingesetzte Softwarekomponenten

In diesem Kapitel werden die Softwarekomponenten vorgestellt, die in diesem Projekt verwendet werden, um die beiden Frameworks – das General Board Game Framework (GBG) und Stable Baselines3 (SB3) – miteinander zu vernetzen, um somit die RL-Agenten aus SB3 ins GBG zu integrieren. Zunächst werden die beiden Frameworks betrachtet. Anschließend folgen die Komponenten, mit denen die bidirektionale Schnittstelle zwischen ihnen implementiert wird. Ziel ist es, einen Überblick über die eingesetzten Softwareelemente zu geben und deren Relevanz für dieses Projekt zu erläutern.

3.1 Das General Board Game Framework (GBG)

Das GBG [6], [7] ist ein Framework zur Erstellung von Machine-Learning-Agenten und Brettspielen. Es ist in Java geschrieben. Das Framework bietet standardisierte Schnittstellen für verschiedene Agententypen – sowohl für lernende Agenten wie TD-Agenten als auch für nicht lernende Agenten wie etwa Max-N- oder MCTS-Agenten. Es soll den Benutzer beim Erstellen von Agenten und Spielen unterstützen, indem es eine Reihe von Interfaces und abstrakten Klassen bereitstellt. Diese geben bereits Standards vor und nehmen dem Benutzer repetitive Arbeiten beim Programmieren ab.

Ein weiterer Kerngedanke dieses Frameworks ist die möglichst hohe Kompatibilität von Agenten und Spielen. Dadurch wird eine Plattform geschaffen, auf der eine große Anzahl an Agenten in verschiedenen Spielen trainiert, getestet und verglichen werden können.

Das Framework wird den Nutzern durch eine entsprechende grafische Benutzeroberfläche (GUI) zugänglich gemacht. Über die GUI kann man einfach neue Trainingsdurchläufe für die Agenten starten, diese gegeneinander antreten lassen oder auch direkt selbst über ein visualisiertes Spielfeld gegen die Agenten spielen. Des Weiteren kann der Nutzer über die GUI beispielsweise Parameter für Agenten einstellen oder sich die Qualität von Agenten anhand visualisierter Daten anzeigen lassen.

Da diese Projekt Deep Reinforcement Learning für das GBG ermöglichen und den Funktionsumfang des GBG erweitern soll, bildet das GBG eine der zentralen Technologien in diesem Projekt.

3.1.1 Überblick über die wichtigsten Komponenten des GBGs

Um einen besseren Überblick über das GBG zu bekommen, sollen im Folgenden die wichtigsten Klassen bzw. Interfaces vorgestellt und auf ihre Relevanz für dieses Projekt eingegangen werden.

Arena

Die abstrakte Klasse `Arena` bildet den zentralen Knotenpunkt des GBGs (siehe Abbildung 2). Sie enthält die grundlegenden Methoden zum Starten von Training und Tests, die wiederum an andere Klassen delegiert werden. In der Klasse läuft eine Schleife, die auf Änderungen eines Enums namens `taskState` wartet – beispielsweise von `IDLE` zu `TRAIN`. Nach dieser Änderung wird das Training eingeleitet. Die Änderung des `taskState` erfolgt durch den Benutzer über die GUI-Klassen. Die Schleife selbst läuft in einem Thread, der durch die Klasse `GBGLaunch` bei einer Benutzerinteraktion gestartet wird. Jedes Spiel implementiert seine eigene konkrete Arena-Unterklasse und stellt dort Factory-Methoden für Objekte bereit, die spielspezifisch sind. Beispiele für diese Klassen sind `GameBoard`, `Feature` und `XNTupleFuncs`.

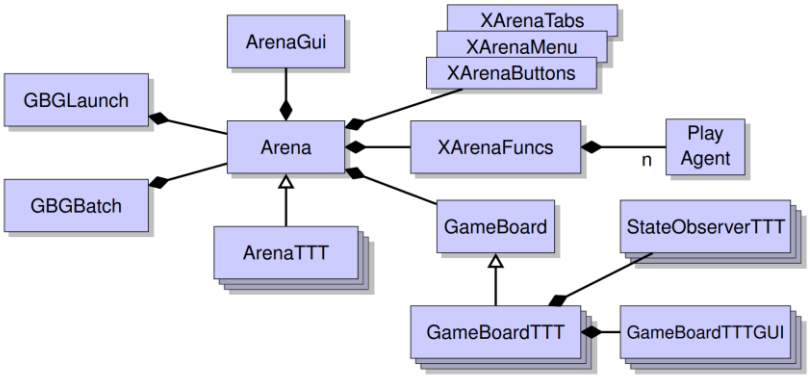


Abbildung 2: Klassen Diagramm GBG: Klassen um die Arena [6, S. 8].

XArenaFuncs

`XArenaFuncs` ist eine Hilfsklasse, die die eigentliche Trainings- und Testlogik enthält. Die Methode `train` startet eine Schleife, die über die Spielrunden (Episoden) läuft. In jeder Episode wird die Methode `trainAgent` des `PlayAgent`-Interfaces aufgerufen, um den Agenten zu trainieren. Anschließend werden Statistiken erstellt und optional eine

Evaluation durchgeführt. Außerdem enthält die Klasse Funktionen zur Erstellung neuer Agenten.

Für dieses Projekt ist vor allem die Schleife in der `train`-Methode von Bedeutung, da SB3 eine eigene Schleife zur Steuerung des Trainings besitzt und somit ein Konflikt entsteht. Dieser Konflikt wird im späteren Verlauf der Arbeit nochmal in Abschnitt 4.2 aufgegriffen.

XNTupleFuncs und BoardVector

Die Methoden in `XNTupleFuncs` sind primär für das Training von TD-Agenten mit einem n-Tupel-System vorgesehen. Daher sind die meisten Methoden für dieses Projekt nicht relevant. Verwendet werden jedoch:

- `getBoardVector`: Gibt ein `BoardVector`-Objekt zurück, das ein eindimensionales Integer-Array mit dem State des Spielfelds enthält. Bei Tic-Tac-Toe z. B. wird der Zustand jeder Kachel durch eine ganze Zahl repräsentiert: eine 0 für Kreis, eine 1 für ein leeres Feld und eine 2 für ein Kreuz.
- `getNumPositions`: Gibt die Anzahl möglicher Zustände pro Feld zurück (z. B. 3 bei Tic-Tac-Toe: leer, Kreis, Kreuz).
- `getNumPlayers`: Gibt die Anzahl der Spieler des Spiels zurück.

Die Information aus diesen Methoden werden gebraucht, um ein neues Environment mithilfe von Gymnasium zu erstellen und um dem SB3-Agnten einen Vektor, der den State des Spieles repräsentiert, zum Trainieren zur Verfügung zu stellen.

PlayAgent

Das Interface `PlayAgent` muss von allen Agenten des GBGs implementiert werden. Es dient dem GBG dazu, alle Agenten einheitlich integrieren und ansprechen zu können.

Auch wenn sich der eigentliche Agent in der SB3-Python-Anwendung befindet, muss im Rahmen dieses Projekts eine konkrete Klasse erstellt werden, die dieses Interface erfüllt, um eine möglichst nahtlose Integration des SB3-Agenten in das GBG zu gewährleisten.

Die wichtigste Methode, die für dieses Interface zu implementieren ist, ist `getNextAction2`. Sie übergibt dem Agenten eine `StateObservation` und gibt die aus Agentensicht bestmögliche Aktion zurück.

StateObservation

Das Interface `StateObservation` muss von jedem Spiel im GBG implementiert werden. Es bietet dem Agenten eine Schnittstelle, um die aktuelle Observation des States zu erhalten und diesen durch Aufruf der Methode `advance` – unter Angabe der auszuführenden Aktion – in den nächsten State zu überführen (Step).

Im Kontext dieses Projekts ist diese Schnittstelle der Zugangspunkt, über den der Agent tatsächlich mit der Spiellogik interagiert – sie stellt in diesem Sinne das Environment dar.

3.2 Stable-Baselines3

Ziel dieses Projekts ist es, die Agenten bzw. Algorithmen von Stable-Baselines3 (SB3) im GBG-Framework nutzbar zu machen. SB3 ist ein Open-Source-Framework, das modellfreie RL-Algorithmen implementiert. Das erklärte Ziel von SB3 ist es – wie der Name bereits andeutet – zuverlässige RL-Algorithmen bereitzustellen, die als Referenz (Baseline) für Experimente dienen können [8, Kap. 1]. Denn bereits kleine Unterschiede in der Implementierung können sich stärker auf die Performance auswirken als der gewählte Algorithmus selbst [9].

Die hohe Zuverlässigkeit von SB3 wird durch verschiedene Maßnahmen sichergestellt: Die Agenten werden systematisch in bekannten Environments getestet und mit früheren Versionen verglichen. Der Quellcode ist zu 95 % durch Unit Tests abgedeckt – ein höherer Wert als bei vergleichbaren RL-Frameworks [8, Kap. 1]. Zudem besteht eine aktive Community, die regelmäßig Beiträge liefert und Code-Reviews durchführt [8, Kap. 1]. Eine umfangreiche Dokumentation sowie vollständig typisierte und kommentierte Funktionen erhöhen zusätzlich die Nutzbarkeit [8, Kap. 2].

SB3 zeichnet sich durch seine einfache API aus, die bereits mit wenigen Zeilen Code ein vollständiges Training ermöglicht. Die Integration von TensorBoard und Callbacks erlaubt eine einfache Protokollierung und Visualisierung wichtiger Parameter wie dem durchschnittlichen Reward über die Trainingsdauer hinweg.

SB3 implementiert insgesamt sieben bekannte modellfreie RL-Algorithmen: A2C, PPO, DDPG, SAC, TD3, HER und DQN. Darüber hinaus werden im SB3-Contrib-Repository weitere Algorithmen angeboten, die neuere experimentelle Features enthalten. In diesem Projekt werden die SB3-Agenten DQN und PPO sowie das Maskable PPO aus SB3 Contrib verwendet.

3.2.1 Minimal Working Exempel

Um einen kurzen Einblick in die einfache API von SB3 zu geben und die wichtigsten Funktionen sowie deren Handhabung zu veranschaulichen, soll im Folgenden ein Beispiel eines Trainingsdurchlaufs gezeigt werden.

Agenten erstellen

Zuerst muss der Agent über den Konstruktor erstellt werden:

```
agent = DQN("MlpPolicy", env)
```

Dem Konstruktor müssen zwei zentrale Parameter übergeben werden:

- Der Name des Policy-Modells, z. B. `MlpPolicy` für ein vollständig verbundenes Feedforward-Netz oder `CnnPolicy` für ein Convolutional-Netz. In diesem Projekt wird jedoch ausschließlich die `MlpPolicy` verwendet.
- Ein Gymnasium RL-Environment, das für das Training verwendet wird.

Zusätzlich können weitere Parameter gesetzt werden, etwa die Lernrate (`learning_rate`), der Anteil explorativer Aktionen (`exploration_fraction`) oder `policy_kwargs`, mit denen sich die Architektur des neuronalen Netzes anpassen lässt.

Training starten

Das Training wird mit folgendem Aufruf gestartet:

```
agent.learn(total_timesteps=10_000)
```

Der Aufruf der `learn`-Methode startet die Trainingsschleife, die so lange läuft, bis die gewünschte Anzahl an `total_timesteps` erreicht ist. In dieser Schleife wird mithilfe der Methode `collect_rollouts` neue Erfahrung im Environment gesammelt, indem die `step`- und `reset`-Methoden des Gymnasium Environments aufgerufen werden. Nach einer gewissen Anzahl an Episoden oder Steps wird regelmäßig die `train`-Methode des Agenten aufgerufen, die den Agenten entsprechend den eingestellten Parametern trainiert.

Agenten benutzen

Nach Abschluss des Trainings kann der Agent über die Methode `predict` verwendet werden, um eine Aktion vorherzusagen:

```
action, _ = agent.predict(obs)
```

Hierzu wird dem Agenten die aktuelle Observation übergeben. In diesem Projekt liegt diese Observation in Form eines 1-dimensionalen `numpy`-Arrays (Observationsvektor) vor, das den aktuellen Zustand des Environments beschreibt.

3.3 Gymnasium

Gymnasium ist eine Open-Source-Bibliothek und ein Fork des Vorgängers der Gym-Bibliothek von OpenAI. Sie wird von der Farama Foundation aktiv weiterentwickelt. Die Bibliothek stellt standardisierte Schnittstellen für Reinforcement-Learning-Environments bereit [10, Kap. 6], um RL-Algorithmen einen einheitlichen Zugriff auf diese zu ermöglichen.

Für dieses Projekt ist die Implementierung eines Gymnasium-kompatiblen Environments von zentraler Bedeutung, da Stable-Baselines3 auf genau diesen Environment-Typ setzt, um seine Agenten zu trainieren. Das GBG, welches die eigentliche Spiellogik umfasst, muss daher indirekt als Gymnasium-Environment verfügbar gemacht werden. Dies erfolgt über eine speziell dafür entwickelte Schnittstelle, damit die SB3-Python-Anwendung korrekt auf das Environment zugreifen kann.

3.3.1 Environments

Gymnasium unterscheidet zwischen mehreren Environment-Typen: `Env`, `VectorEnv` und `FuncEnv`.

- `VectorEnv` erlaubt die gleichzeitige Ausführung mehrerer identischer Umgebungen, was ein effizienteres Training durch Parallelisierung ermöglicht [10, Kap. 3].
- `FuncEnv` stellt eine alternative Schnittstelle bereit, die näher am klassischen MDP orientiert ist [10, Kap. 4.1].
- `Env` ist das Standardenvironment. In diesem Projekt kommt ausschließlich dieses Environment zum Einsatz. Die zentralen Komponenten dieses Environment-Typs werden im Folgenden beschrieben.

3.3.2 Environment Methoden

Ein Gymnasium-Environment bietet zwei primäre Methoden, um die Interaktion des Agenten bzw. Algorithmus mit dem Environment zu ermöglichen:

1. `step`-Methode: Diese Methode erhält eine Aktion als Eingabe, führt sie im Environment aus und gibt die daraus resultierende neue Observation des States sowie den dazugehörigen Reward zurück. Diese Methode führt einen Step im Environment aus. Innerhalb der Trainingsschleife von SB3 wird sie wiederholt aufgerufen, um Erfahrungen zu sammeln.
2. `reste`-Mehtode: Nach dem Abschluss einer Episode wird das Environment mithilfe dieser Methode zurückgesetzt. Dabei wird die erste Observation des initialen Sates zurückgegeben.

3.3.3 Spaces

Ein Gymnasium-Environment muss angeben, welchen Observationsraum (`observation_space`) und Aktionsraum (`action_space`) es hat – also in welcher Form es Aktionen entgegennimmt und in welcher Form die Observationen zurückgegeben werden. Hierfür stellt Gymnasium sogenannte `Space`-Klassen zur Verfügung. In diesem Projekt werden ausschließlich die `Discrete`-Klasse für Aktionsräume und die `MultiDiscrete`-Klasse für Observationsräume verwendet. Diese Klassen sind für kategorische Eingaben und Ausgaben gedacht, bei denen sich die Werte eines Zustands in klar abgrenzbare Kategorien unterteilen lassen.

Ein Beispiel für einen diskreten Aktionsraum: Wenn ein Agent nur zwei Aktionen ausführen kann – etwa „gehe nach rechts“ oder „gehe nach links“ – lassen sich diese als Elemente einer endlichen Menge {Links, Rechts} darstellen. Sie sind also klar kategorisierbar. In der Praxis werden diese Aktionen dann durch ganze Zahlen repräsentiert, z. B. 0 für „Links“ und 1 für „Rechts“. Der Wert, der hier entweder 0 oder 1 sein kann, wird auch Feature genannt.

Die `Discrete`-Klasse beschreibt einen Raum, in dem genau ein Feature aus einer endlichen Menge erlaubt ist. Sie eignet sich für die meisten Aktionsräume, wie sie in vielen Brettspielen vorkommen, in denen pro Zug nur eine eindeutige Entscheidung getroffen werden kann.

Die `MultiDiscrete`-Klasse hingegen wird verwendet, um Vektoren von diskreten, kategorisierbaren Features zu modellieren. Ein gutes Beispiel ist eine Spielfeld-Kachel, die entweder leer, vom Spieler oder vom Gegner besetzt ist – drei klar unterscheidbare Kategorien. Eine Kachel stellt ein Feature dar, und der Vektor würde dann einen Eintrag für jedes Feature des Spielfelds beinhalten. Anders als bei der `Discrete`-Klasse können so mehrere Features übermittelt werden, deren Einträge im Vektor unterschiedlichen Wertebereichen (Kardinalitäten) angehören. So kann ein Eintrag im Vektor ein Feature mit fünf verschiedenen Kategorien darstellen und der nächste Eintrag z. B. ein Feature mit nur drei Kategorien.

In SB3 werden diese Spaces unter anderem dazu verwendet, zu bestimmen, wie viele Eingabewerte und Ausgabewerte das neuronale Netz benötigt. Wichtig ist hierbei zu wissen, dass SB3 bei diskreten Spaces die Features vor der Eingabe ins neuronale Netz one-hot-encoded. Beim One-Hot-Encoding wird jede Kategorie eines Features in einem Vektor durch eine 1 oder 0 dargestellt – eine 1, wenn das Feature zu dieser Kategorie gehört, und eine 0, wenn nicht. Das bedeutet beispielsweise, dass ein dieser Eingabe-Vektor (0, 1, 2), mit drei Features mit jeweils drei Kategorien, am Ende so dargestellt wird: (1, 0, 0, 0, 1, 0, 0, 0, 1).

3.4 TensorBoard

TensorBoard ist ein Visualisierungstool, das ursprünglich von Google im Rahmen des TensorFlow-Projekts entwickelt wurde. Es ermöglicht die grafische Auswertung von Metriken während des Trainings neuronaler Netze. In diesem Projekt wird TensorBoard in Kombination mit SB3 integriert, um den Trainingsverlauf von RL-Agenten zu dokumentieren und zu analysieren. Zu den wichtigsten Metriken gehören unter anderem der durchschnittliche Reward und die Länge einer Episode. Außerdem können auch eigene Evaluationsergebnisse und Hyperparameter geloggt werden.

Die Integration in SB3 ist sehr benutzerfreundlich gestaltet, da SB3 bereits Callbacks und Logger für TensorBoard mitliefert. Die Trainingsdaten werden dabei standardmäßig im Verzeichnis `/logs` gespeichert und können über die TensorBoard-Weboberfläche in Echtzeit betrachtet werden.

3.4.1 Minimal Working Example

TensorBoard kann über den Befehl installiert werden:

```
pip install tensorboard
```

Die Nutzung von TensorBoard kann in SB3 einfach aktiviert werden, indem beim Erstellen eines neuen SB3-Agenten im Konstruktor das Verzeichnis angegeben wird, in dem der Log-Ordner gespeichert werden soll. Danach kann TensorBoard in diesem Projekt standardmäßig mit folgendem Befehl aus dem Root-Verzeichnis der Python-Anwendung gestartet werden:

```
tensorboard --logdir ./logs
```

Anschließend ist unter **`http://localhost:6006/`** eine interaktive Benutzeroberfläche erreichbar.

Des Weiteren können eigene Werte geloggt werden, indem ein neuer Callback erstellt wird. In der `_on_step`-Methode lassen sich dann benutzerdefinierte Metriken erfassen, wie es die Beispiele [11] auf der SB3-Website zeigen.

3.5 Http und Py4J

Im Rahmen erster Recherchen zur Umsetzung einer bidirektionalen Schnittstelle in diesem Projekt wurden zwei geeignete Kommunikationstechnologien identifiziert: HTTP und Py4J. Beide ermöglichen eine Datenübertragung zwischen Java- und Python-Anwendungen. In diesem Abschnitt werden sie vorgestellt und miteinander verglichen. Während hier die theoretischen Grundlagen sowie Vor- und Nachteile untersucht werden, wird in Kapitel 5 ein praktischer Test durchgeführt, um zu bestimmen, welche der beiden Alternativen sich für dieses Projekt als performanter erweist.

Eine weitere Alternative, die kurzzeitig in Betracht gezogen wurde, ist Pyjnius. Diese Technologie eignet sich jedoch nur für den unidirektionalen Zugriff von Python auf Java-Klassen und ist damit für dieses Projekt nicht geeignet.

3.5.1 Hypertext Transfer Protocol (http)

HTTP [12] ist ein weit verbreitetes Protokoll, das bereits seit 1990 eingesetzt wird. Es wird vor allem verwendet, um Hypertext- und Hypermedia-Informationen über das World Wide Web abzurufen. Genauer beschreibt HTTP ein Protokoll zur zustandslosen Datenübertragung auf der Anwendungsebene des OSI-Referenzmodells [13].

HTTP wird jedoch auch häufig verwendet, um die Kommunikation zwischen lokalen Anwendungen zu ermöglichen. Beim Austausch von Daten zwischen Anwendungen über das HTTP-Protokoll kommen oft die textbasierten Datenformate XML, YAML und JSON zum Einsatz. In diesem Projekt wird JSON verwendet, da es im Allgemeinen die beste Performance bietet. Allerdings besitzt JSON die geringste Funktionalität – diese ist jedoch für dieses Projekt vollkommen ausreichend.

3.5.2 Py4J

Py4J [14] ist eine Bibliothek, die von Barthélémy Dagenais entwickelt wurde. Sie ermöglicht die Interaktion zwischen Python und Java durch Zugriff auf Java-Objekte innerhalb der Java Virtual Machine. Zusätzlich erlaubt sie Java-Anwendungen über Callbacks, auf Python-Objekte zuzugreifen. Damit erlaubt Py4J eine bidirektionale Kommunikation zwischen Java und Python, was für dieses Projekt erforderlich ist.

3.5.3 Vergleich

HTTP ist ein allgemeines Protokoll zur Datenübertragung, das von vielen verschiedenen Anwendungen genutzt werden kann. Im Gegensatz dazu ist Py4J speziell für die Kommunikation zwischen Python und Java konzipiert. In diesem Projekt wird eine spezifische Schnittstelle zwischen den zwei bekannten Anwendungen implementiert. Es ist unwahrscheinlich, dass weitere externe Anwendungen diese Schnittstelle ebenfalls nutzen werden – dennoch wäre das mit HTTP grundsätzlich möglich.

Ein weiterer Vorteil von HTTP besteht darin, dass sich damit eine Verlagerung des Trainings auf externe Server realisieren lässt, etwa solche mit spezieller Hardware für neuronale Netze. HTTP eignet sich ideal für den Datenaustausch über das Internet, was diesen Ansatz unterstützt.

Py4J bietet hingegen eine dynamischere Möglichkeit der Interaktion zwischen den Programmen. So kann direkt aus Python auf Java-Objekte zugegriffen werden, deren Methoden ausgeführt oder Felder ausgelesen werden. Der Zustand der Objekte bleibt dabei stets aktuell. Bei HTTP hingegen müssen neue Anfragen erstellt und gesendet werden, um den aktuellen Stand eines Objekts zu erhalten. Py4J übernimmt das automatisch und agiert dadurch auf einer höheren Abstraktionsebene.

Die Implementierung mit Py4J ist in vielen Fällen einfacher, da sie den Datenaustausch vollständig übernimmt. Bei HTTP hingegen müssen beide Anwendungen HTTP-Endpunkte bereitstellen, und zusätzlich ist die Serialisierung und Deserialisierung der Daten notwendig. HTTP gewährt dafür mehr Kontrolle über den Ablauf der Kommunikation und die Komplexität der zu übertragenden Daten und Kommunikation fällt in diesem Projekt jedoch gering aus, weshalb der mehr Aufwand durch die Implantierung der Schnittstelle mit HTTP überschaubar bleiben würde.

Py4J hat zudem einen höheren technischen Overhead als eine Schnittstelle, die lediglich HTTP voraussetzt. Außerdem kann unter einer Vielzahl Softwarekomponenten ausgewählt werden, um eine HTTP-Schnittstelle implementieren. So kann nur das Nötigste implementiert werden, was zu einer besseren Performance führen kann. Da sich auch im praktischen Test (siehe Kapitel 5) HTTP als performanter herausgestellt hat, wird für dieses Projekt die Schnittstelle über HTTP implementiert.

3.6 Softwarekomponenten für eine HTTP-Schnittstelle

Für die Umsetzung einer bidirektionalen Schnittstelle zwischen dem GBG und der SB3-Python-Seite über das HTTP-Protokoll werden in beiden Anwendungen Frameworks eingesetzt, die das Hosting eines HTTP-Servers sowie die Definition von HTTP-Endpunkten ermöglichen. Zusätzlich ist es notwendig, dass jede Anwendung die Schnittstellen der jeweils anderen abfragen kann. Im Folgenden wird erläutert, welche Frameworks bzw. Bibliotheken dafür verwendet werden und aus welchen Gründen sie für dieses Projekt ausgewählt wurden.

3.6.1 FastApi

FastAPI [15] ist ein Web-Framework, um moderne, schnelle HTTP-Schnittstellen mit Python 3.7 und aufwärts zu erstellen. Als Fundament nutzt FastAPI Uvicorn [16], Starlette [17] und Pydantic [18]. Uvicorn ist ein schneller ASGI-Server, der als Host für FastAPI empfohlen wird. Starlette übernimmt das Web-Handling, also die Verarbeitung der HTTP-Anfragen und das Antworten, zum Beispiel das Routen von HTTP-Anfragen anhand ihres Pfades zur korrekten Funktion. Pydantic wird zur Validierung der Daten sowie zur Serialisierung und Deserialisierung verwendet, indem es sogenannte BaseModels mithilfe von Python-Type-Hints mit den Request-/Response-Bodys abgleicht.

Für dieses Projekt wurde FastAPI ausgewählt, da es eines der schnellsten Web-Frameworks für Python ist [19]. Außerdem spricht die einfache Handhabung der Request-/Response-Modelle für FastAPI.

In diesem Projekt wird FastAPI verwendet, um einfache HTTP-Endpunkte in der SB-Python-Anwendung einzurichten. Diese Endpunkte führen gezielte Methoden im RL-Agenten aus und erhalten die dafür nötigen Daten über die HTTP-Anfrage. Dazu werden die HTTP-Endpunkte mittels Annotationen auf Funktionen gemappt. So erkennt FastAPI, welche Funktion aufgerufen werden soll, wenn von außen ein HTTP-Anfrage an einen bestimmten Endpunkt gesendet wird.

Des Weiteren wird über die Funktionsparameter der annotierten Funktionen definiert, welche Daten der Endpunkt entgegennimmt. Dazu wird ein Funktionsparameter mit einer Klasse annotiert, die von der Pydantic-Klasse `BaseModel` erbt und die Struktur der über die HTTP-Anfrage gesendeten Daten beschreibt. FastAPI bzw. Pydantic übernimmt bei eingehenden HTTP-Anfragen automatisch die Deserialisierung sowie die Validierung der Daten. Das bedeutet, der Request-Body – in diesem Projekt im JSON-Format – wird automatisch in Objekte der annotierten Python-Klassen überführt. Sollten die Daten nicht mit den definierten Attributen der Klasse übereinstimmen, schlägt die Validierung fehl und es wird ein Fehler zurückgegeben. Auch für HTTP-Antworten wird ein Pydantic-BaseModel zurückgegeben, und FastAPI übernimmt die Serialisierung automatisch.

3.6.2 Java HTTP-Server aus dem `com.sun.net.httpserver`-Paket

Der leichtgewichtige HTTP-Server aus dem `com.sun.net.httpserver`-Paket ist seit Java 6 Teil des JDK, gehört jedoch nicht offiziell zum Java-SE-Standard. Mit seiner Hilfe lassen sich einfache REST-Endpunkte erstellen, indem sogenannte HTTP-Handler definiert werden. Diese bestimmen, welche Methoden bei einem eingehenden Request ausgeführt und welche HTTP-Antworten zurückgegeben werden.

Im Gegensatz zu höheren Frameworks wie FastAPI arbeitet dieses Server-Framework auf einer niedrigeren Abstraktionsebene und stellt keine integrierten Funktionen für Validierung, Serialisierung oder Deserialisierung bereit.

Für dieses Projekt ist die Einfachheit des Frameworks ein großer Vorteil, da dadurch der technische Overhead gering bleibt und keine zusätzlichen Abhängigkeiten notwendig sind. Zudem lässt sich das Framework gut in das bestehende GBG integrieren. Da nur wenige, einfache REST-Endpunkte benötigt werden, ist der Funktionsumfang des Frameworks ausreichend.

Da der einzige Client der Schnittstelle die Python-Anwendung auf demselben lokalen System ist, sind erweiterte Features wie HTTPS, Authentifizierung oder Multithreading für dieses Projekt nicht relevant.

3.6.3 Requests-Python-Bibliothek

Um die HTTP-Schnittstelle in Java abzufragen, wird auf der Python-Seite die requests-Bibliothek benutzt. Diese Bibliothek ermöglicht es, HTTP-Anfragen auf einfache Weise zu senden und abstrahiert dabei viele komplexe Details des HTTP-Protokolls. So sendet man in der Regel eine HTTP-Anfrage mit nur einer Zeile Code.

Aufgrund ihrer Benutzerfreundlichkeit zählt die requests-Bibliothek mit rund 30 Millionen Downloads pro Woche zu den am häufigsten verwendeten Python-Packages [20]. Auch für dieses Projekt fiel die Wahl auf die requests-Bibliothek, da sie alle nötigen Funktionalitäten mitbringt und durch ihre Einfachheit überzeugt.

Alternativen wie httpx oder urllib wären ebenfalls geeignet gewesen und bieten teilweise sogar mehr Funktionalitäten. Diese zusätzlichen Features sind für dieses Projekt jedoch nicht erforderlich.

3.6.4 Java HTTP-Client aus dem `java.net.http.HttpClient`-Paket

Der `java.net.http.HttpClient` wird im GBG in der Klasse `SB3AgentProxy` dazu benutzt, um die HTTP-Schnittstelle der Python-Seite abzufragen. Der `HttpClient` stellt eine moderne API zur Verfügung, um HTTP-Anfragen in Java zu formulieren, zu senden und Antworten zu verarbeiten. Für dieses Projekt wird `java.net.http.HttpClient` verwendet, anstatt externer Bibliotheken wie dem Apache HttpClient oder OkHttp, da er seit Java 11 integraler Bestandteil der Java-Standardbibliothek ist und somit keine zusätzlichen Abhängigkeiten erfordert, die die die Komplexität und den Pflegeaufwand erhöhen würden.

4 Problemstellung und Zielsetzung

SB3 stellt standardisierte, leistungsfähige Deep-Reinforcement-Learning-Algorithmen zur Verfügung, die im GBG bislang nicht implementiert sind. Das GBG-Framework bietet eine Vielzahl bereits entwickelter Spiele sowie eine modulare Umgebung, um Agenten zu trainieren, zu testen und im Wettbewerb gegeneinander antreten zu lassen. Ziel dieses Projekts ist es, SB3-Agenten in das GBG zu integrieren. Damit könnten diese auch auf die im GBG vorhandenen Spiele angewendet und ihre Leistungsfähigkeit mit den bisherigen

Agenten verglichen werden. Dies würde die Funktionalität des GBGs erheblich erweitern – insbesondere, da dort bislang keine vergleichbaren RL-Algorithmen auf Basis tiefer neuronaler Netze integriert sind.

Aus dieser Zielsetzung ergeben sich mehrere zentrale Problemstellungen:

1. **Technologischer Bruch:** Das GBG-Framework ist in Java, SB3 hingegen in Python implementiert. Da keine direkte Kompatibilität besteht, muss eine bidirektionale Kommunikationsschnittstelle zwischen beiden Frameworks geschaffen werden.
2. **Entscheidung über die Kommunikationstechnologie (HTTP vs. Py4J):** Bevor eine Kommunikationsschnittstelle implementiert werden kann, muss untersucht werden, welche der beiden Technologien – HTTP oder Py4J – sich besser für dieses Projekt eignet.
3. **Unterschiedliche Trainingsarchitekturen:** Sowohl das GBG als auch SB3 verfügen über eigene Trainingsschleifen zur Steuerung des Lernprozesses. Es muss festgelegt werden, welches der beiden Frameworks diese Steuerung übernimmt.
4. **Kompatibilität mit Gymnasium:** SB3 benötigt ein Environment, das mit Gymnasium kompatibel ist. Da die eigentliche Spiellogik – und somit auch das Environment – innerhalb des GBGs liegt und dort eigene Schnittstellen implementiert werden, muss dieses Environment nach außen bzw. zu SB3 hin so abgebildet werden, dass es als Gymnasium-Environment fungiert.

In den folgenden Abschnitten werden diese Herausforderungen detailliert erläutert und die Lösungsansätze beschrieben.

4.1 SB3-Python-Anwendung und das GBG

Die SB3-Python-Anwendung und das GBG sollten als zwei voneinander getrennte Anwendungen betrachtet werden, die eng miteinander interagieren, sobald ein SB3-Agent trainiert wird. Das GBG kann unabhängig betrieben werden und benötigt die SB3-Python-Anwendung nur dann, wenn ein SB3-Agent eingesetzt werden soll.

Obwohl die SB3-Anwendung stark auf das GBG zugeschnitten ist und primär dazu dient, das SB3-Framework in das GBG zu integrieren, wäre es mit geringem Anpassungsaufwand auch möglich, die bereitgestellte HTTP-API in anderen Anwendungen zu verwenden.

4.2 Kontrolle über die Trainingsschleife

Eine zentrale Frage beim Aufbau dieses Projekts war, welche Seite – das GBG oder die SB3-Python-Anwendung – die Kontrolle über die Trainingsschleife übernehmen soll. Unter der Trainingsschleife versteht man jenen Teil des Prozesses, der den Agenten fortlaufend dazu veranlasst, Steps im Environment auszuführen, und ihn in regelmäßigen Abständen auf Basis der gesammelten Erfahrungen trainiert.

Beide Seiten haben hierfür jeweils eigene Trainingsschleifen implementiert: Im GBG befindet sich die Trainingslogik in der Methode `train` der Klasse `xArenaFuncs`, während SB3 für jeden Algorithmus eine `learn`-Methode bereitstellt.

In diesem Projekt wurde entschieden, die Kontrolle über die Trainingsschleife bei SB3 zu belassen. Dies hat folgende Gründe:

- Alle in diesem Projekt verwendeten RL-Algorithmen von SB3 implementieren die `learn`-Methode auf unterschiedliche Weise. Würde stattdessen die GBG-Trainingsschleife verwendet, müssten für jeden Algorithmus eigene Methoden implementiert werden, um einen Step oder eine Episode zu trainieren. Dies würde den Aufwand deutlich erhöhen.
- Neue SB3-Algorithmen, die in Zukunft genutzt werden sollen und einen eigenen Aufbau der Trainingsschleife mitbringen, können dann ebenfalls einfach über die `learn`-Methode eingebunden werden, ohne neue Methoden schreiben zu müssen.
- Das GBG verwendet eine Schleife über Episoden, während SB3 eine Schleife über Steps nutzt. Um ein episodisches Training bei On-Policy-Algorithmen von SB3 zu ermöglichen, müsste die `collect_rollouts`-Methode angepasst werden.
- Die Idee hinter SB3 ist es, einheitliche und vergleichbare RL-Algorithmen bereitzustellen (siehe Abschnitt 3.2). Diese Grundidee würde verloren gehen, wenn Trainingsabläufe auch nur leicht unterschiedlich implementiert würden.

Aus diesen Gründen wurde entschieden, die Kontrolle bei SB3 zu belassen, auch wenn dadurch die Struktur des Trainingsablaufs im GBG gebrochen wird. Das `PlayAgent`-Interface wird in diesem Fall nicht mehr erfüllt, und das Training von SB3-Agenten stellt nun einen Sonderfall dar. Es mussten daher Änderungen in der Klasse `xArenaFuncs` vorgenommen werden, wie im Abschnitt 6.3.5 gezeigt wird. Diese Änderungen sorgen dafür, dass das GBG während des Trainings weiterhin auf dem neuesten Stand gehalten wird, insbesondere durch die Aktualisierung von Variablen, die den Trainingsprozess dokumentieren.

4.3 Anforderungen an die Schnittstelle

Damit das GBG und die SB3-Python-Anwendung miteinander kommunizieren können, muss eine bidirektionale Schnittstelle entwickelt werden. Das bedeutet, dass beide Seiten in der Lage sein müssen, Funktionen oder Methoden der jeweils anderen Anwendung aufzurufen und Daten auszutauschen.

Der Großteil dieser Daten wird numerisch sein und hauptsächlich aus Ganzzahlen bestehen. Dies lässt sich damit begründen, dass die meisten Daten voraussichtlich beim Übertragen von Observationsvektoren anfallen, welche die aktuelle Observation des Agenten abbilden. Der SB3-Agent benötigt diese Daten sowohl beim Training als auch später bei der Auswahl der jeweils besten Aktion. Da im GBG derzeit nur Spiele mit

diskreten Zustandsräumen implementiert sind, bestehen die Observationsvektoren ausschließlich aus Ganzzahlen.

Im Folgenden werden die notwendigen Zugangspunkte beschrieben, die sowohl in der SB3-Python-Anwendung als auch im GBG implementiert werden müssen, um die Kommunikation zu ermöglichen.

4.3.1 Zugangspunkte der SB3-Python-Anwendung

1.1 **Agent und Environment erstellen:** Dieser Zugangspunkt ermöglicht es dem GBG-Client, neue SB3-Agenten sowie passende Gymnasium-Environments für einen Trainingsdurchlauf in der SB3-Python-Anwendung zu erzeugen. Die Schnittstelle nimmt dabei diverse Parameter entgegen – beispielsweise zur Beschreibung des Environments (z. B. Größe des Observations- und Aktionsraums) sowie des Agenten (z. B. Agententyp oder die Struktur des neuronalen Netzes, das im Agenten verwendet wird).

1.2 **Training eines Agenten starten:** Da das Training in der GBG-Anwendung durch den Benutzer gestartet, aber anschließend von SB3 gesteuert wird, muss die SB3-Python-Anwendung darüber informiert werden, das Training zu starten. Zudem werden über diesen Zugangspunkt Parameter übergeben, die den Trainingsprozess beschreiben, z. B. aus wie vielen Steps das Training besteht.

1.3 **Agenten speichern:** Nach dem Training eines Agenten kann der Benutzer über die GBG-Anwendung Agenten speichern. Daher muss auch die SB3-Python-Anwendung einen Zugangspunkt zur Verfügung stellen, der es ermöglicht, einen Agenten auf der Festplatte zu persistieren. Dabei werden Parameter übergeben, z. B. in welches Verzeichnis der Agent gespeichert werden soll.

1.4 **Agenten laden:** Damit der Benutzer über das GBG-GUI auch SB3-Agenten laden kann, muss die SB3-Python-Anwendung ebenfalls einen Zugangspunkt bereitstellen, der persistierte Agenten von der Festplatte lädt. Die übergebenen Parameter beschreiben unter anderem, welcher gespeicherte Agent geladen werden soll.

1.5 **Vorhersage der besten Aktion:** Dies ist wohl einer der wichtigsten Zugangspunkte der SB3 Python Anwendung, da hierüber der SB3 Agent die Aktion für einen State (Spielzustand) zurückgeben kann die er als am besten betrachtet. Dieser State muss in Form des Observationsvektors über diesen Zugangspunkt mitgeteilt werden, damit anschließend eine Aktion zurückgegeben werden kann.

1.6 **Self-Play:** Ähnlich wie beim Zugangspunkt Nr. 1.5 wird hier ein Observationsvektor übergeben und eine als optimal angesehene Aktion zurückgegeben. Dieser Zugangspunkt soll jedoch ausschließlich während des Trainings verwendet werden, um gemäß der Self-Play-Methode (siehe Abschnitt 2.3) Aktionen für Gegner des zu trainierenden Agenten vorherzusagen.

4.3.2 Zugangspunkte des GBG

2.1 Zurücksetzen des Environments: Wie in der Problemstellung geschildert, muss das GBG Methoden des Gymnasium-Environments über seine Schnittstelle verfügbar machen. Dieser Zugangspunkt erfüllt die Methode `reset` eines Gymnasium-Environments. Sie muss aufgerufen werden, nachdem eine Episode zu Ende ist, und soll die Spielumgebung zurücksetzen sowie die erste Observation des States in einem Observationsvektor zurückgeben.

2.2 Schritt im Environment (Step): Dieser Zugangspunkt entspricht der Methode `step` eines Gymnasium-Environments. Es ist der am häufigsten aufgerufene Zugangspunkt während des Trainings, da er vor jedem Spielzug genutzt wird. Eine Aktion wird übergeben, diese wird in der Spielumgebung ausgeführt, eventuelle Züge der Gegner werden vorgenommen und der daraus resultierende neue Observationsvektor, der Reward und die Information, ob das Spiel beendet ist, werden zurückgegeben. Mithilfe dieser Informationen trainiert SB3 den Agenten.

2.3 Training beendet: Da das Training über das GBG gestartet wird und anschließend die Kontrolle an die SB3-Python-Anwendung übergeht, muss das GBG darüber informiert werden, wenn das Training abgeschlossen ist. Danach läuft die GBG-Anwendung normal weiter, als ob ein GBG-eigener Agent trainiert worden wäre.

2.4 Verfügbare Aktionen: Dieser Zugangspunkt wird ausschließlich während des Trainings eines Maskable-PPO-Agenten verwendet. Er entspricht der `action_mask` Methode des Gymnasium-Environments. Der Maskable-PPO-Agent muss vor jedem Step wissen, welche Aktionen legal sind. Über diesen Zugangspunkt werden die legalen Aktionen zurückgegeben.

2.5 Evaluieren: Dieser Zugangspunkt ermöglicht es, eine Evaluation des SB3-Agenten zu starten. Dabei werden der durchschnittliche Reward sowie die Siege, Niederlagen und Unentschieden über alle Spiele hinweg zurückgegeben. Es müssen Parameter übergeben werden, wie z. B. gegen welchen Agenten evaluiert werden soll und wie viele Episoden gespielt werden sollen. Dieser Zugangspunkt wird während des Trainingsprozesses regelmäßig verwendet; die Ergebnisse werden in einer TensorBoard-Datei zur Dokumentation gespeichert. Zusätzlich wird der aktuelle Policy des Agenten gespeichert, sofern diese besser ist als frühere Versionen.

4.4 Kommunikationsablauf eines Trainings

Im Folgenden soll die Nutzung der Zugangspunkte im Ablauf eines typischen Trainingsprozesses beschrieben werden. Zur Vereinfachung sind für diesen Ablauf die Evaluation bzw. TensorBoard deaktiviert, und es wird nicht mit einem Maskable-PPO Agent trainiert.

1. **Initialisierung**: Das GBG verwendet Zugangspunkt 1.1 der SB3-Python-Anwendung, um Spiel- und Agentenparameter zu übermitteln. Anschließend

werden das passende Gymnasium-Environment sowie der Agent erzeugt. Der erste Kommunikationsschritt geht vom GBG aus, da der Nutzer hier die Trainingseinstellungen vornimmt und das Training initiiert.

2. **Trainingsstart**: Danach nutzt das GBG Zugangspunkt 1.2, um das Training zu starten. In der SB3-Python-Anwendung wird das Training dann über die `learn`-Methode des SB3-Agenten gestartet.
3. **Episodenbeginn:** In der Trainingsschleife des SB3-Agenten wird nun zu Beginn jeder Episode die `reset`-Methode der Gymnasium-Environment-Klasse aufgerufen. Dort wird Zugangspunkt 2.1 verwendet, um das Environment im GBG zurückzusetzen und die erste Observation zu erhalten.
4. **Training:** Nun wird aus der Trainingsschleife heraus so lange Zugangspunkt 2.2 über die `step`-Methode der Gymnasium-Environment-Klasse aufgerufen, bis die Episode beendet ist. Sobald eine Episode endet, wird – wie in Punkt 3 beschrieben – erneut Zugangspunkt 2.1 aufgerufen. Dieser Zyklus wiederholt sich, bis die konfigurierte Anzahl an Steps erreicht ist. Der SB3-Agent trainiert währenddessen mit den hierbei gewonnenen Informationen.
5. **Trainingsende:** Nach Abschluss des Trainings teilt die SB3-Anwendung dem GBG über Zugangspunkt 2.3 mit, dass das Training beendet ist.

Bei Verwendung eines Maskable PPO-Agenten wird vor jedem `step`-Aufruf zusätzlich Zugangspunkt 2.4 genutzt, um die gültigen Aktionen abzufragen.

Ist TensorBoard aktiviert, wird nach definierten Zeitintervallen Zugangspunkt 2.5 zur Evaluation des Agenten aufgerufen.

5 HTTP und Py4J Praktischer Test

In der frühen Phase des Projekts musste eine geeignete Kommunikationstechnologie für die Schnittstelle zwischen dem GBG und der Python-Seite ausgewählt werden. Nach einer ersten theoretischen Analyse (siehe Abschnitt 3.5) der verfügbaren Optionen konzentrierte sich die Auswahl auf zwei Technologien: HTTP und Py4J.

Im nächsten Schritt sollte durch einen praktischen Vergleich untersucht werden, welche der beiden Varianten sich besser zur Umsetzung einer Schnittstelle eignet. Dazu wurden vier Dummy-Anwendungen entwickelt – jeweils eine HTTP- und eine Py4J-Version für die GBG- [21] und Python-Seite [22]. Diese Anwendungen simulieren das Verhalten der späteren realen Implementierung.

Ziel dieser Tests war es zum einen, die praktische Handhabung und die Eignung beider Technologien im Kontext dieses Projekts zu evaluieren. Zum anderen sollte analysiert werden, welchen Einfluss die jeweilige Kommunikationslösung auf die Performance hat – insbesondere im Hinblick auf die Trainingszeit eines SB3-Agenten. Zusätzlich konnten

bereits in diesem frühen Teststadium wertvolle Erkenntnisse über Aufbau und Integration gewonnen werden, die die spätere Entwicklung der Schnittstelle vereinfachen.

5.1 Testaufbau

In diesem Test wurde versucht, möglichst reale Bedingungen zu erzeugen. Der Testaufbau ist für beide Seiten – abgesehen von der verwendeten Kommunikationstechnologie, also HTTP oder Py4J – identisch. Der Test simuliert einen typischen Ablauf des Trainings eines SB3-Agenten (siehe Abschnitt 4.4). Die Python-Anwendung trainiert während des Tests einen SB3-Agenten mit zufälligen Daten für den Observationsvektor, die über die GBG-Schnittstelle mit der gewählten Kommunikationstechnologie übermittelt werden.

Es wurde bewusst entschieden, während der Testdurchläufe SB3 zu integrieren und einen tatsächlichen Agenten zu trainieren – anstatt lediglich Daten zwischen zwei Dummy-Anwendungen auszutauschen. Dies soll möglichst praxisnahe Bedingungen schaffen. So kann ermittelt werden, inwieweit die Kommunikationstechnologie tatsächlich die Trainingszeit beeinflusst. Außerdem werden die Schnittstellen nur dann aufgerufen, wenn sie im Trainingsprozess tatsächlich benötigt werden, was die normale Auslastung realistischer widerspiegelt.

Die übermittelten Observationen besteht aus einem Vektor mit Ganzzahlen der Größe 10, 50, 100, 200 und 1000 für verschiedene Testdurchläufe. Somit kann die Performance der Kommunikationstechnologien bei unterschiedlich großen Datenmengen verglichen werden. Die beiden Kommunikationstechnologien werden immer abwechselnd in einer Testrunde mit 1000 Spielzügen getestet.

In jeder Testrunde wird die Zeit gemessen, die vom Start des Trainingsprozesses bis zum Abschluss benötigt wird. Zeiten wie das Hochfahren der Server werden dabei vernachlässigt. Diese Zeit dient als zentrales Kriterium zur Bewertung der Performance der jeweiligen Technologie in diesem Projekt.

Jede Technologie wird in einem Testdurchlauf jeweils 15-mal getestet, also gibt es pro Testdurchlauf insgesamt 30 Testrunden. Die Steuerung der Testdurchläufe sowie das Starten der Anwendungen übernimmt ein PowerShell-Skript. Nach jeder Runde folgt eine 5-sekündige Pause.

Die Testdurchläufe wurden auf einem Windows-11-Betriebssystem durchgeführt, wobei darauf geachtet wurde, dass keine anderen rechenintensiven Anwendungen derweil liefen. Durch das abwechselnde Testen der Technologien sollen systembedingte Auslastungsschwankungen ausgeglichen werden, sodass beide Technologien möglichst faire Bedingungen erhalten. Die Hardwarekomponenten des verwendeten Systems sind in Anhang 1, Tabelle 9 aufgelistet.

5.2 Ergebnisse und Auswertung

Anhang 1 zeigt die Zeitmessungen der Testdurchläufe. Die gemessenen Zeiten fallen über alle Testdurchläufe hinweg recht konstant aus. Die Standardabweichung ist für alle Testdurchläufe sehr gering. In keinem Testdurchlauf beträgt die Standardabweichung mehr als 1,3 % des Mittelwerts.

Die Ergebnisse zeigen, dass eine HTTP-Schnittstelle für dieses Projekt performanter ist als eine mit Py4J implementierte. Die Zeitdifferenz zwischen den beiden Kommunikationstechnologien, die ein simuliertes Training benötigt, nimmt zu, je größer der Observationsvektor wird. Bei kleinen Observationsvektoren fällt diese Differenz noch moderat aus: Bei einem Observationsvektor der Größe 10 benötigt das Training mit Py4J etwa 37 % mehr Zeit als mit HTTP. Die Zeitdifferenz steigt jedoch beträchtlich mit der Größe des Observationsvektors an: Bei einer Vektorgröße von 50 benötigt die Py4J-Schnittstelle bereits ca. 87 % länger und bei einer Größe von 200 ca. 175 % länger im Vergleich zur HTTP-Schnittstelle. Die Ergebnisse sind in den folgenden Abbildungen 3 und 4 zusammengefasst.

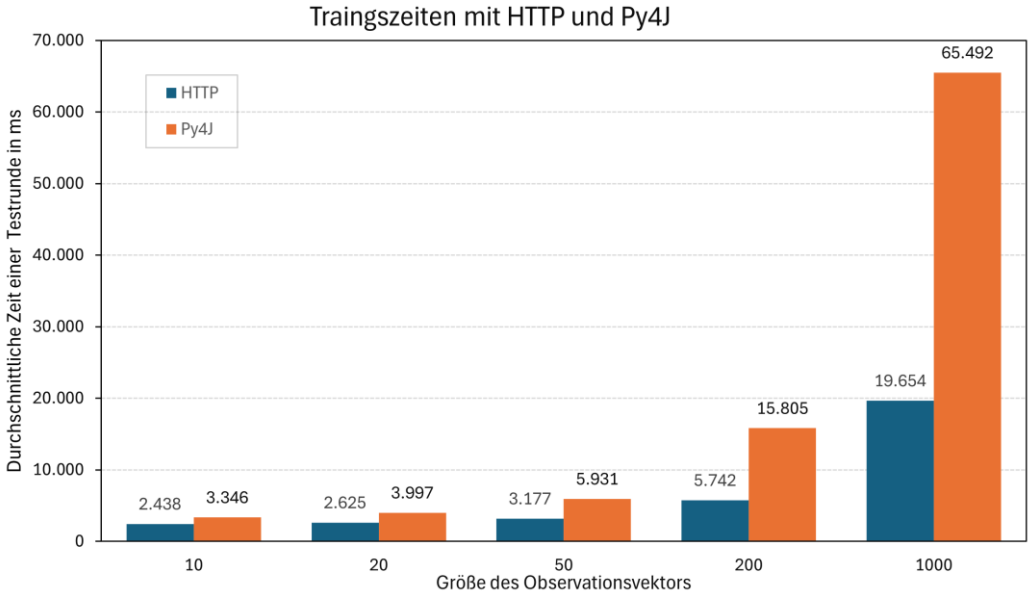


Abbildung 3: Durchschnittliche Trainingszeiten der Testdurchläufe mit der HTTP- und Py4J-Schnittstelle bei unterschiedlichen Größen des Observationsvektors.

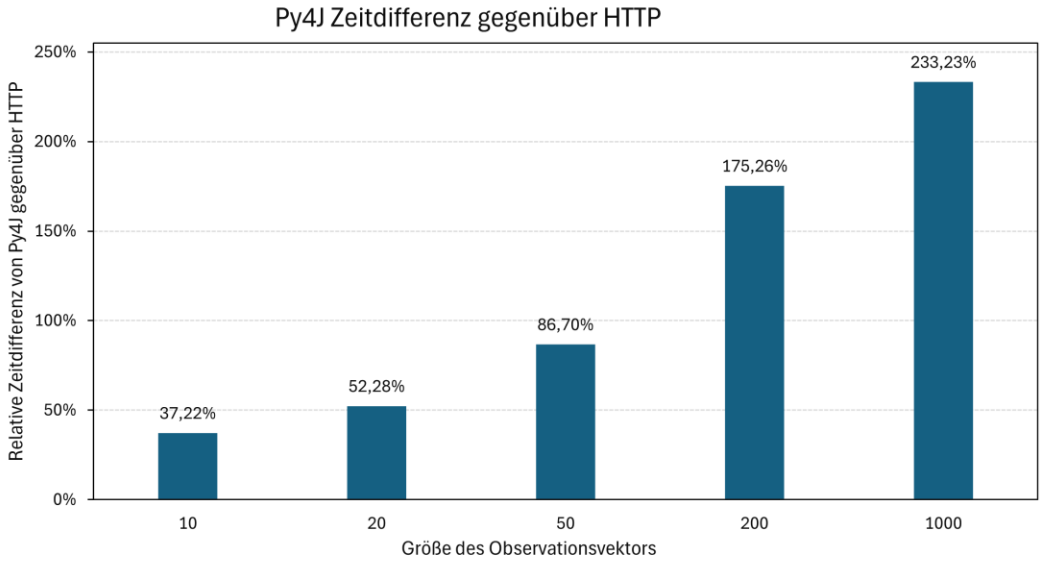


Abbildung 4: Durchschnittliche Trainingszeitverlängerung bei Verwendung der Py4J-Schnittstelle im relativen Vergleich zur HTTP-Schnittstelle.

5.3 Auswirkung der Ergebnisse auf das Projekt

Wie in der Ergebnisauswertung gezeigt, ist eine HTTP-basierte Schnittstelle für dieses Projekt performanter als eine Implementierung mit Py4J. Während der Unterschied bei kleinen Observationsvektoren noch moderat bleibt, wird er bei größeren Observationsvektoren deutlich relevanter.

Für das Spiel Tic-Tac-Toe zum Beispiel, dessen Observationsvektor zehn Elemente umfasst (neun für das Spielfeld und eines für die Spielerrolle), lässt sich davon ausgehen, dass der Zeitunterschied für das Training eines SB3-Agenten mit Py4J nur höchstens 37 % länger wäre als mit HTTP. In komplexeren Spielen wie Vier gewinnt, wo der Observationsvektor mindestens 43 Komponenten hat, könnte es laut Test bereits etwa 70% länger dauern einen Agenten mit der Py4J-Schnittstelle zu trainieren.

Solche Abweichungen sind bei der Wahl der finalen Kommunikationsschnittstelle nicht mehr vernachlässigbar. Auch die in Abschnitt 3.5.3 beschriebenen Vorteile von Py4J, etwa die einfachere Handhabung, rechtfertigen diesen Performanceverlust nicht. Aus diesem Grund fiel die Entscheidung zugunsten einer Integration der SB3-Agenten über eine HTTP-Schnittstelle.

6 Integration der SB3-Agenten in das GBG

In diesem Kapitel wird die Implementierung der SB3-Python-Anwendung sowie der Teil des GBG beschrieben, der für das Training von SB3-Agenten zuständig ist. Dabei werden die Architektur und Klassenstruktur erläutert, die jeweiligen Rollen und Verantwortlichkeiten der Klassen diskutiert sowie die Kommunikationsabläufe innerhalb und zwischen beiden Anwendungen dargestellt. Der Code dieses Softwareprojekts – der

SB3-Python-Anwendung sowie der Erweiterung des GBG – ist auf der GitHub-Seite des GBG zu finden [7].

6.1 Trainingsablauf

Der Ablauf des Trainings orientiert sich an den in Abschnitt 4.4 beschriebenen Punkten. Eine detaillierte Übersicht darüber, welche Funktionen, Methoden bzw. HTTP-Endpunkte zu welchem Zeitpunkt im Trainingsprozess aufgerufen werden, liefert das folgende UML-Sequenzdiagramm (siehe Abbildung 5).

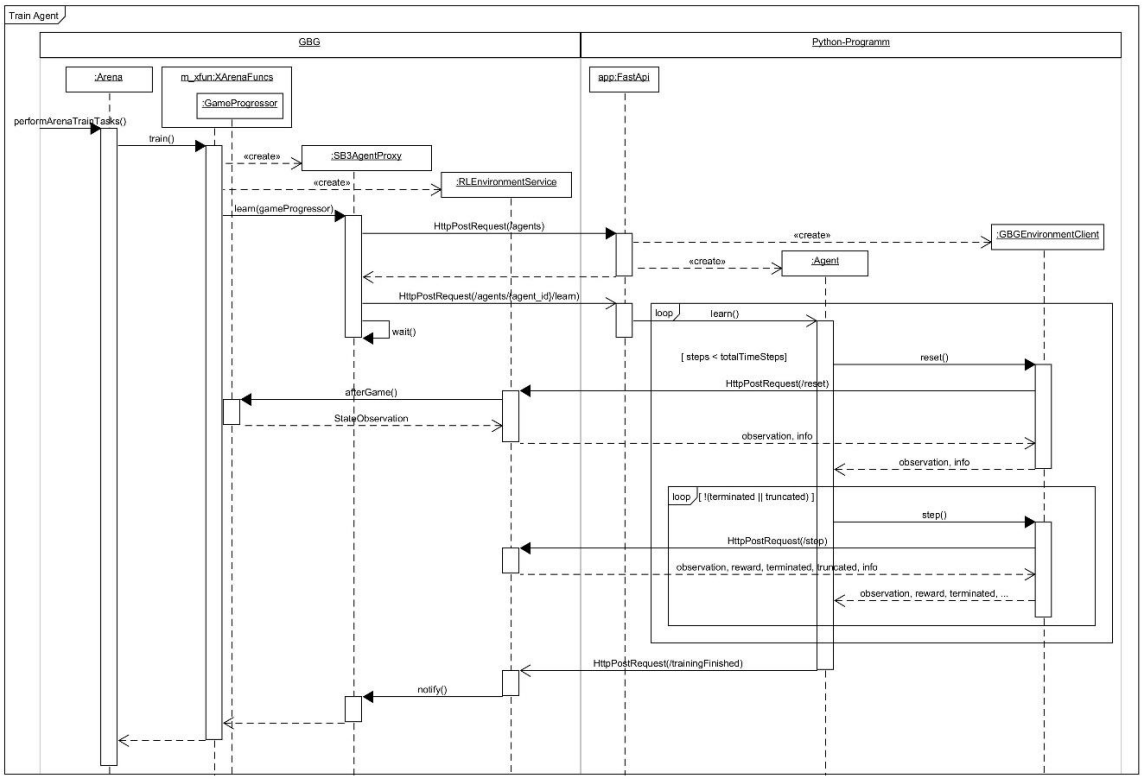


Abbildung 5: UML-Sequenzdiagramm zum Trainingsablauf.

6.2 SB3-Python-Anwendung

Im Folgenden soll der in Python geschriebene Teil des Projekts vorgestellt werden. Der Code der SB3-Python-Anwendung verteilt sich auf diese Dateien: `sb3_agent_service.py`, `environment_connector.py`, `agent_trainer.py`. Jede der drei Dateien kann jeweils ein klarer Verantwortlichkeitsbereich innerhalb der Anwendung zugeordnet werden. Diese Verantwortlichkeiten werden im Folgenden erläutert. Darüber hinaus wird dargestellt, wie diese Komponenten miteinander in Beziehung stehen, und es wird auf die konkrete Implementierung eingegangen.

6.2.1 SB3-Agent-Service

Diese Datei enthält den Code für die HTTP-Schnittstelle, über die das GBG mit der SB3-Python-Anwendung kommuniziert. Sie enthält außerdem den Einstiegspunkt der

Anwendung: die `main`-Funktion, welche eine FastAPI-Serverinstanz startet und somit die Anwendung initialisiert.

Alle für die Kommunikation mit dem GBG notwendigen HTTP-Endpunkte werden hier definiert. Diese Endpunkte erfüllen die im Abschnitt 4.3.1 beschriebenen Anforderungen.

Die Datei verwaltet außerdem die aktuell verfügbaren bzw. geladenen Agenten. Damit eine Anfrage korrekt verarbeitet werden kann, muss bei fast allen Endpunkten die UUID des Agenten im URL-Pfad angegeben werden – mit Ausnahme des Endpunkts zur Erstellung eines neuen Agenten. In diesem Fall wird die UUID im Request-Body übergeben.

Zu den wichtigsten HTTP-Endpunkten zählen:

- `/agents`: Zur Erstellung eines neuen Agenten,
- `/agents/{agent_id}/learn`: Startet das Training,
- `/agents/{agent_id}/predict`: Rückgabe der vorhergesagten Aktion zu einer gegebenen Observation.

Die HTTP-Endpunkte werden mithilfe von FastAPI durch annotierte Funktionen definiert (siehe Abschnitt 3.6.1). Zusätzlich befinden sich in dieser Datei die Pydantic-Base-Models, die zur Validierung und (De-)Serialisierung der über HTTP übermittelten Daten dienen.

Für eine detaillierte API-Dokumentation der HTTP-Schnittstelle kann die OpenAPI-Dokumentation aufgerufen werden, die automatisch von FastAPI generiert wird, nachdem eine Serverinstanz erstellt wurde. Die Dokumentation ist dann unter folgender URL zu finden: **`http://127.0.0.1:8095/docs/`**.

6.2.2 Agent-Trainer

Die Datei `agent_trainer.py` enthält hauptsächlich eine Wrapper-Klasse namens `Agent`, die SB3-Agenten kapselt. Zusätzlich sind dort mehrere Callback-Klassen definiert, die während des Trainingsprozesses zum Einsatz kommen. Der Wrapper ist dazu gedacht, die verschiedenen unterstützten SB3-Agenten (DQN, PPO, Maskable PPO) auf einheitliche Weise nutzen zu können und zusätzliche Funktionalitäten wie Self-Play und Evaluation hinzuzufügen. Im Folgenden sollen diese Funktionalitäten genauer erläutert werden.

Erstellung neuer Agenten mit Parametern

Die Erstellung neuer Agenten erfolgt über die `form_parameters` Methode der `Agent` Klasse. Sie nimmt Wesentlichen ein Gymnasium Environment und drei Arten von Parametern entgegen, mit denen ein SB3-Agent erstellt werden kann. Die Parameter werden als `dict` gespeichert. Sie teilen sich auf in:

- `baseParameters`: Parameter, die jeder SB3-Agent besitzt,

- `agentParameters`: Parameter, die spezifisch für den jeweiligen Agenten sind,
- `networkParameters`, Parameter, die beschreiben, wie das neuronale Netz innerhalb des Agenten aufgebaut werden soll.

Diese Parameter entsprechen direkt den Übergabeparametern, die beim Erstellen neuer SB3-Agenten an den Konstruktor übergeben werden können. Sie können in der GUI des GBG im Parameter-Tab für SB3 eingestellt werden und werden dem SB3-Agent-Service bei der Erstellung neuer Agenten übermittelt.

Um diese Parameter möglichst flexibel zu halten, werden sie innerhalb der SB3-Python-Anwendung lediglich in einem `dict` gespeichert und daher nicht durch Pydantic validiert (siehe Abschnitt 3.6.1). Das bedeutet, dass die vom GBG übergebenen Parameter fehlerhaft sein oder gar nicht verfügbar sein könnten. Der Vorteil liegt jedoch in der einfachen Erweiterbarkeit: Es können leichter neue Parameter hinzugefügt oder weggelassen werden, und der Implementierungsaufwand ist deutlich geringer.

Daher ist es wichtig, bei der Implementierung neuer Parameter für SB3-Agenten in der GBG-Anwendung sicherzustellen, dass diese für den vorgesehenen Agenten existieren, korrekt benannt sind und Werte, die diese annehmen können, zulässig sind.

Self-Play

Self-Play ist eine weitere Funktionalität, die die Wrapper-Klasse `Agent` bereitstellt. Wie im Abschnitt 2.3 über Self-Play beschrieben, benötigt ein Agent Gegner auf vergleichbarem Niveau, um effektiv trainieren zu können. Zur Umsetzung speichert die `Agent`-Klasse nach einer festgelegten Anzahl an Trainingsschritten Kopien der jeweils aktuellen Policy. Dabei wird nicht nur die zuletzt verwendete Policy gespeichert, sondern eine konfigurierbare Anzahl früherer Versionen. So lernt der Agent, mit verschiedenen Gegnern und Taktiken umzugehen.

Benötigt das GBG nun eine Aktion für einen Gegenspieler, kann – sofern in der Konfiguration Self-Play als Gegner ausgewählt wurde – eine HTTP-Anfrage an den Endpunkt `/agents/{agent_id}/selfPlay` gesendet werden. Dieser wird vom SB3-Agent-Service an den `Agent`-Wrapper weitergeleitet, der daraufhin eine Aktion zurückgibt, die von einer früheren Policy ausgewählt wird.

Der zeitliche Abstand, in dem neue Policies gespeichert werden, die Anzahl gespeicherter Versionen und die Häufigkeit, mit der die aktuellste Policy verwendet wird, sind Hyperparameter, die das Training maßgeblich beeinflussen. Diese lassen sich im Parameter-Tab für SB3 im GBG konfigurieren.

Die Predict-Methode

Die Standard-`predict`-Methode eines SB3-Agenten liefert für eine gegebene Observation die Aktion zurück, die der Agent für optimal hält. Das GBG benötigt jedoch häufig zusätzlich eine Bewertung aller möglichen Aktionen – etwa für die visuelle Darstellung bei einem Spiel gegen einen menschlichen Spieler.

Aus diesem Grund erweitert die Wrapper-Klasse `Agent` die `predict`–Methode so, dass neben der besten Aktion auch die Bewertungen aller legalen Aktionen beim Aufruf des HTTP-Endpunkts `/agents/{agent_id}/predict` an das GBG zurückgegeben werden. Die Bedeutung dieser Werte unterscheidet sich je nach Agententyp: Bei PPO und Maskable PPO geben sie die Wahrscheinlichkeit an, mit der eine Aktion als optimal eingeschätzt wird. Beim DQN hingegen entsprechen sie dem erwarteten maximalen kumulierten Reward.

Dabei werden ausschließlich gültige Züge berücksichtigt. Da DQN- und PPO-Agenten standardmäßig keine Filterung ungültiger Aktionen bei `predict` vornehmen, wurde diese Logik ebenfalls durch den Wrapper erweitert. Sollte die ursprünglich berechnete Aktion ungültig sein, gibt die Methode die nächstbeste gültige Aktion zurück.

Wichtig ist, dass diese `predict`-Methode nicht vom trainierenden Agenten selbst verwendet wird. Während des Trainings wird mit ungültigen Aktionen auf andere Weise umgegangen (siehe Abschnitt 2.2.3). Die `predict`-Methode bzw. der entsprechende HTTP-Endpunkt wird vom GBG nach dem Training zum Spielen mit dem Agenten aufgerufen.

Callbacks

Abgesehen von der `Agent`-Klasse sind die übrigen drei Klassen in der Datei `agent_trainer.py` Callbacks, die von SB3s `BaseCallback` erben und ausschließlich innerhalb der `Agent`-Klasse verwendet werden. Die Callbacks werden beim Starten des Trainings über die `learn`-Methode dem SB3-Agenten hinzugefügt. Während des Trainingsprozesses ruft SB3 bei jedem Step die `_on_step`-Methode des jeweiligen Callbacks auf, in der z. B. Trainingsdaten gespeichert werden können. In diesem Fall sollen die meisten Callbacks jedoch nur nach einer bestimmten Anzahl an Steps aufgerufen werden, weshalb sie mit einem Decorator-Callback namens `EveryNTimesteps` versehen werden.

Die Callbacks haben folgende Funktionen:

- Der **AddToSelfPlayCallback** speichert alle *n* Steps die aktuelle Policy für Self-Play.
- Der **EvaluationCallback** sendet alle *n* Steps eine HTTP-Anfrage an den GBG-HTTP-Endpunkt `/eval` und stößt damit eine Evaluation des Agenten an. Das Ergebnis, in Form des durchschnittlichen Rewards, der Siege, Gleichstände und Niederlagen, während der Evaluationsepisoden, wird mit der HTTP-Antwort vom GBG zurückgeschickt und in einer TensorBoard-Datei gespeichert. Diese Metrik hilft dabei, den Trainingsverlauf besser zu verstehen und den Erfolg des Trainings zu bewerten.
- Der **HParamCallback** speichert beim Start des Trainings statische Hyperparameter in der TensorBoard-Datei, um im Nachhinein besser analysieren

zu können, zu welchen Ergebnissen bestimmte Parameterkombinationen geführt haben.

6.2.3 Environment-Connector

In dieser Datei ist vor allem die Klasse `GBGEnvironmentClient` zu finden. Sie implementiert ein Gymnasium-Environment (`Env`) und stellt die wichtigen Methoden `step` und `reset` bereit (siehe Abschnitt 3.3.2), die SB3 nutzt, um Agenten in Gymnasium-Environments zu trainieren. Außerdem implementiert die Klasse die Methode `action_masks`, die allerdings nur beim Training eines Maskable PPOs durch SB3 verwendet wird.

Obwohl die Klasse `GBGEnvironmentClient` die zentralen Methoden eines Gymnasium-Environments bereitstellt, enthält sie keinerlei Logik oder Regeln, die das Environment bzw. das Spiel selbst definieren. Die tatsächliche Spiellogik wird innerhalb der Spiele des GBGs festgelegt. Daher werden sämtliche Methodenaufrufe an das GBG weitergeleitet. Dies geschieht über die HTTP-Schnittstelle des GBGs, wobei die jeweiligen HTTP-Endpunkte gleichnamig zu den Methoden sind. Die Methoden geben somit lediglich die Antworten des GBGs auf die HTTP-Anfragen zurück.

Beim Erstellen eines neuen `GBGEnvironmentClient` müssen dem Konstruktor sogenannte `EnvironmentParameters` übergeben werden. Diese werden benötigt, um den Observations- und Aktionsraum des Environments zu definieren (siehe Abschnitt 3.3.3). In diesem Projekt ist der Observationsraum stets vom Typ `MultiDiscrete` und der Aktionsraum vom Typ `Discrete`, da alle Observationen und Aktionen im GBG ausschließlich diskrete Werte annehmen.

6.2.4 Interaktion der drei Komponenten

Sobald das GBG den HTTP-Endpunkt `/agents` aufruft, um einen neuen SB3-Agenten zu erstellen, wird in der Funktion `create_agent` des SB3-Agent-Service sowohl eine Instanz des `GBGEnvironmentClient` als auch des Agent-Wrappers initialisiert. Da zur Instanziierung eines SB3-Agenten ein Gymnasium-Environment übergeben werden muss, wird die zuvor erstellte Instanz des `GBGEnvironmentClient` an den Agent-Wrapper übergeben, der damit wiederum seinen SB3-Agenten erstellt.

Nach dem Funktionsaufruf `create_agent` verfügt der SB3-Agent-Service über eine Instanz des Agent-Wrappers, und der SB3-Agent innerhalb des Wrappers besitzt die einzige Instanz des `GBGEnvironmentClient`. Diese wird vom SB3-Agenten ausschließlich während des Trainings verwendet.

Die meisten beim Agent-Service eingehenden HTTP-Anfragen werden an den Agent-Wrapper weitergeleitet. Der Agent-Service übernimmt dabei hauptsächlich die Aufgabe, die eingehenden Daten in das jeweils benötigte Format zu transformieren.

6.3 Integration in das GBG-Framework

Im Folgenden soll der Teil bzw. der Code dieses Projekts vorgestellt werden, der neu zum GBG-Framework hinzugefügt wurde, um die Nutzung von SB3-Agenten im GBG zu ermöglichen.

Der größte Teil des neu implementierten Codes befindet sich im `SB3`-Package innerhalb des `controllers`-Packages. Dort finden sich die beiden für dieses Projekt relevanten Klassen `RLEnvironmentService` und `SB3AgentProxy`. In einem eigenen Unter-Package names `HttpServer` liegen zusätzlich die Klassen `RLEnvironmentServer`, die den Server implementiert, sowie `EnvironmentHttpHandler`, einen abstrakten HTTP-Handler bereitstellt, von dem alle konkreten HTTP-Handler erben.

Außerhalb des für dieses Projekt vorgesehenen `SB3`-Packages wurden zwei weitere Klassen für diese Projekt hinzugefügt: `StateObservationVectorFuncs` im Package `games` und der `GameProgressor`, der als innere Klasse der Klasse `XArenaFuncs` definiert ist.

Die Funktionen all dieser Klassen werden im Folgenden erläutert – ebenso ihre Struktur untereinander und, falls vorhanden, ihr Zusammenhang mit der SB3-Python-Anwendung.

6.3.1 Agent Proxy

Die Aufgabe der Klasse `SB3AgentProxy` ist es, als Stellvertreter für einen echten Agenten zu fungieren. Sie implementiert das Interface `PlayAgent`, das im GBG für alle Agenten benötigt wird, um diese einheitlich in das System zu integrieren (siehe *PlayAgent* in Abschnitt 3.1.1). Die Klasse bildet dabei lediglich das Verhalten eines Agenten ab, während die eigentliche Logik auf der Seite der SB3-Python-Anwendung liegt. Wie beim Environment-Connector werden daher Methodenaufrufe, die die Agentenlogik betreffen, direkt an die SB3-Anwendung weitergeleitet.

Ein zentrales Beispiel ist die Methode `getNextAction2`, die im `PlayAgent`-Interface definiert ist. Diese ruft die vom Agenten geschätzte beste Aktion über den HTTP-Endpunkt `/agents/{agent_id}/predict` ab.

Alle HTTP-Requests dieser Klasse erfolgen über den bereits beschriebenen HTTP-Client aus dem `java.net.http.HttpClient`-Paket (siehe Abschnitt 3.6.4).

Da die Kontrolle über die Trainingsschleife auf der Seite der SB3-Anwendung liegt, implementiert die `SB3AgentProxy`-Klasse nicht die Methode `trainAgent`, die im Normalfall vom Interface `PlayAgent` verlangt wird, sondern wirft stattdessen eine `RuntimeException`. Um den Trainingsprozess zu starten, muss stattdessen die Methode `learn` aufgerufen werden. Diese erstellt zuerst in der SB3-Anwendung einen neuen Agenten über den `/agents`-HTTP-Endpunkt und startet danach den Trainingsprozess über den Endpunkt `/agents/{agent_id}/learn`.

6.3.2 Java Server

Die Klasse `RLEnvironmentServer` stellt den Server bereit, den die SB3-Anwendung – insbesondere die Klasse `GBGEnvironmentClient` – benötigt, um HTTP-Anfragen an das GBG zu stellen zu können.

Sie ist als Singleton implementiert, um sicherzustellen, dass während der Laufzeit nur eine Instanz existiert.

Zum Einsatz kommt der Java HTTP-Server aus dem `com.sun.net.httpserver`-Paket (siehe Abschnitt 3.6.2). Eingehende HTTP-Anfragen werden zunächst an spezialisierte HTTP-Handler weitergeleitet. Diese übergeben die Anfragen anschließend an den `RLEnvironmentService`, der im folgenden Abschnitt näher beschrieben wird.

6.3.3 Environment-Service

Der Environment-Service, der durch die Klasse `RLEnvironmentService` implementiert wird, ist im Kern die Verbindungstelle zwischen der Logik der Environments, die in den Spielen des GBGs liegt, und SB3, das über die Methoden eines Gymnasium Environments auf diese zugreifen muss.

Die drei Methoden `reset`, `step` und `action_masks`, die der `GBGEnvironmentClient` implementiert, um dem Typ eines Gymnasium Environments zu entsprechen, werden letztendlich alle über HTTP-Anfragen an den Environment-Service weitergeleitet. Dementsprechend greift der Environment-Service auf die Domain-Logik des GBGs zu und macht diese für SB3 zugänglich. Dafür nutzt der Environment-Service die Klassen und Methoden des GBGs und stellt diese nach außen in der Form eines Gymnasium Environments zusammengefasst in den drei erwähnten Methoden dar.

Aus diesem Grund hat der `RLEnvironmentService` eine Instanz der Klasse `StateObservation` des aktuellen Spiels, die die Domain-Logik des Spiels beinhaltete. Über die Klasse `StateObservation` lässt sich ein Spiel steuern, außerdem beinhaltet sie den aktuellen State des Spiels.

Die wichtigsten Methoden des `RLEnvironmentService` sind:

- **`reset`**: Wenn SB3 eine neue Episode über den `GBGEnvironmentClient` starten lässt, wird als Konsequenz diese Methode ausgeführt. In dieser Methode wird zuerst eine neue `StateObservation` über die Methode `afterGame` des `SB3AgentProxy` eingeholt. Daraufhin werden die Aktionen aller gegnerischen Agenten ausgeführt, sollten diese ihren Zug vor dem zu trainierenden Agenten haben, und dann die erste Observation des Spiels zurückgegeben.
- **`step`**: Diese Methode wird ausgeführt, wenn SB3 die `step`-Methode des `GBGEnvironmentClient` aufruft, also jedes Mal, wenn ein Step im Environment erfolgen soll. Es wird eine Aktion übergeben, die im Spiel, über die `advance`-Methode der Klasse `StateObservation`, ausgeführt wird. Anschließend werden

die gegnerischen Züge ausgeführt und die daraus resultierende `Transition` zurückgegeben. Eine `Transition` ist ein Data-Transfer-Object (DTO) und enthält die Observation, den Reward und die Information, ob das Spiel terminiert ist.

- **`switchPlayerPostions`**: Diese Methode wird innerhalb der `step`-Methode jedes Mal aufgerufen, falls eine Episode zu Ende ist. Dann werden die Positionen aller Spieler rotiert. So lernt der Agent, auf jeder Position spielen zu können. Wenn z. B. im Spiel Tic-Tac-Toe in der letzten Episode der trainierende Agent das Symbol „Kreis“ hatte, wird er nun mit „Kreuz“ spielen.

  Zusätzlich können verschiedene Arten von gegnerischen Agenten als Gegner beim Training verwendet werden. Dies soll helfen, den zu trainierenden Agenten einer größeren Vielfalt an gegnerischen Taktiken und Strategien auszusetzen, damit er universeller gegen diese antreten kann. Dazu implementiert diese Methode eine Karussell-System für die gegnerischen Agenten, welches bewirkt, dass in jeder neuen Episode die gegnerischen Agenten rotieren, sodass alle gleichverteilt auf unterschiedlichen Positionen als Gegner im Training eingesetzt werden.

- **`getAvailableActions`**: Diese Methode wird nur vom Maskable PPO benötigt und gibt die aktuell gültigen Aktionen zurück.

6.3.4 Parameter

Um die Parameter für die drei integrierten Agenten einstellen zu können, wurde für die GUI des GBG ein neues Parameter-Tab für SB3-Agenten erstellt. Dieses wurde über die gängige Praxis im GBG realisiert. Dazu wurden die zwei neuen Klassen `ParSB3` und `SB3Params` erstellt.

Im Parameter-Tab können die wesentlichen Parameter eingestellt werden, die das Training der Agenten beeinflussen. Dazu ist die Benutzeroberfläche des Parametertabs in zwei Hälften aufgeteilt. Auf der linken Seite des Tabs befinden sich die allgemeinen Parameter, die jeder SB3-Agent zum Trainieren benötigt, sowie weitere Einstellungen zum Agenten. Über ein Dropdown-Menü auf der linken Seite kann einer der drei integrierten SB3-Agenten ausgewählt werden, der trainiert werden soll. Je nachdem, welcher Agent dort ausgewählt ist, werden auf der rechten Seite des Parametertabs die spezifischen Parameter für diesen Agenten angezeigt.

Auf der linken Seite sind zudem vier Schaltflächen platziert, die jeweils ein neues Fenster öffnen, in dem thematisch gruppierte weitere Parameter und Einstellungen angezeigt werden.

Zu den jeweiligen Parametern und Einstellungen erhält man zusätzliche Informationen, wenn man mit dem Mauszeiger darüber verweilt. Dies soll das Einstellen der Parameter für die Benutzer erleichtern.

6.3.5 Integration des SB3-Agent-Proxys in das restliche GBG

In den meisten Fällen integriert sich der `SB3AgentProxy` wie andere Agenten über die vorgesehenen Mechanismen in das GBG. Allerdings sind an einigen Stellen spezielle Anpassungen erforderlich. In diesem Abschnitt wird beschrieben, welche Änderungen außerhalb des SB3-Packages vorgenommen wurden, um die Besonderheiten dieses Projekts zu berücksichtigen.

Änderungen innerhalb der `XArenaFuncs`

Die wesentlichste Abweichung des SB3AgentProxy von einem regulären Agenten, der das `PlayAgent`-Interface implementiert, besteht darin, dass er die Methode `trainAgent` nicht unterstützt. Dies liegt daran, dass die Trainingslogik und die Trainingsschleife vollständig in der SB3-Anwendung implementiert ist (siehe Abschnitt 4.2). Daher wird in der Methode `train` der Klasse `XArenaFuncs` vor Beginn der Trainingsschleife geprüft, ob es sich beim zu trainierenden Agenten um einen `SB3AgentProxy` handelt. Ist dies der Fall, wird dessen `learn`-Methode aufgerufen und die Trainingskontrolle an die SB3-Anwendung übergeben.

`GameProgressor`

Damit die restlichen Funktionen, die standardmäßig in der Trainingsschleife des GBG ausgeführt werden – wie das Tracken des Lernfortschritts – trotz der ausgelagerten Trainingslogik erhalten bleiben, werden diese nun in einer inneren Klasse der `XArenaFuncs`, dem `GameProgressor`, ausgeführt.

Dazu ruft der `EnvironmentService` vor jeder neuen Episode über den `Agent-Proxy` die Callback-Methode `afterGame` des `GameProgressor` auf. In dieser Methode wird dann – mit Ausnahme des `trainAgent`-Aufrufs auf das `PlayAgent`-Interface – derselbe Code ausgeführt, wie er auch innerhalb der GBG-Trainingsschleife verwendet wird. Dafür müssen bei der Erstellung eines neuen `GameProgressor` alle relevanten Variablen, die in der `train`-Methode definiert sind, dem `GameProgressor` im Konstruktor übergeben werden.

6.3.6 State-Observation-Vektor

Die Klasse `StateObservationVectorFuncs` ist hauptsächlich dafür verantwortlich, eine `StateObservation` (siehe *StateObservation* in Abschnitt 3.1.1) in ein Integer-Array (State-Observation-Vektor) zu transformieren, welches von SB3-Agnten als Input verwendet werden kann, um Aktionen zu schätzen.

Diese Aufgabe wird durch die Methode `getStateObservationVector` umgesetzt, die eine `StateObservation` entgegennimmt und ein Integer-Array zurückgibt. Des Weiteren beinhaltet die Klasse mehrere Methoden wie `getActionSpaceSize` und `getStateObservationVectorStarts`, über die sich der Observationsraum beschreiben lässt und deren Werte zum Erstellen eines neuen SB3-Agenten benötigt werden.

Die Klasse `StateObservationVectorFuncs` ist als allgemeine Klasse gedacht, die für alle Spiele im GBG einen sinnvollen State-Observation-Vektor erstellt. Dazu benutzt sie die `XNTupleFuncs` des jeweiligen Spiels, die im Konstruktor übergeben werden (siehe *XNTupleFuncs und BoardVector* in Abschnitt 3.1.1). Die `XNTupleFuncs` implementieren bereits für jedes Spiel die Methode `getBoardVector`, die ein `BoardVector`-Objekt zurückgibt. Die Instanzvariable `bvec` der Klasse `BoardVector` ist dem State-Observation-Vektor bereits sehr ähnlich; es wird lediglich noch das Feature ergänzt, das beschreibt, auf welcher Position der Agent spielt – zum Beispiel Kreuz oder Kreis bei Tic-Tac-Toe.
Im Rahmen dieses Projekts wurde jedoch nur die Kompatibilität mit den untersuchten Spielen getestet.

Es ist möglich, dass für bestimmte Spiele ein anderer Observationsvektor sinnvoll ist, z. B. mit zusätzlichen Features. Zu diesem Zweck kann eine neue Klasse erstellt werden, die von `StateObservationVectorFuncs` erbt. Ähnlich wie bei den `XNTupleFuncs` kann dann in der jeweiligen `Arena`-Klasse des Spiels die Methode `makeStateObservationVectorFuncs` angepasst werden, sodass sie die spezialisierte `StateObservationVectorFuncs`-Klasse für dieses Spiel zurückgibt.

6.3.7 Struktur der neu Intergierten Klassen

Zu Beginn eines neuen Trainingsprozesses werden innerhalb der Methode `constructAgent` in der Klasse `XArenaFuncs` die drei zentralen Klassen `SB3AgentProxy`, `RLEnvironmentService` und `RLEnvironmentServer` instanziiert.

Dabei werden auch der aggregierenden Klasse `RLEnvironmentServer` und dem `RLEnvironmentService` die entsprechenden Instanzen der Komponenten übergeben. `RLEnvironmentServer` erhält beim Trainieren eines neuen SB3-Agenten stets den aktuellen `RLEnvironmentService`, welcher für jeden Trainingsdurchlauf neu erzeugt wird, da der Service an die neuen Parameter und das jeweilige Spiel angepasst werden muss.

Ebenso erhält der `RLEnvironmentService` jeweils die aktuellen Instanzen des `SB3AgentProxy`, der `StateObservationVectorFuncs` sowie der gegnerischen Agenten, sofern nicht nur Self-Play aktiviert ist.

Die Architektur wurde bewusst so gestaltet, dass zirkuläre Abhängigkeiten vermieden werden. Dadurch bleiben die Verantwortlichkeiten der Klassen klar getrennt, was eine klare Hierarchie und bessere Wartbarkeit der Anwendung gewährleistet.
Eine detaillierte Darstellung der Architektur ist dem folgenden UML-Klassendiagramm zu entnehmen (siehe Abbildung 6).

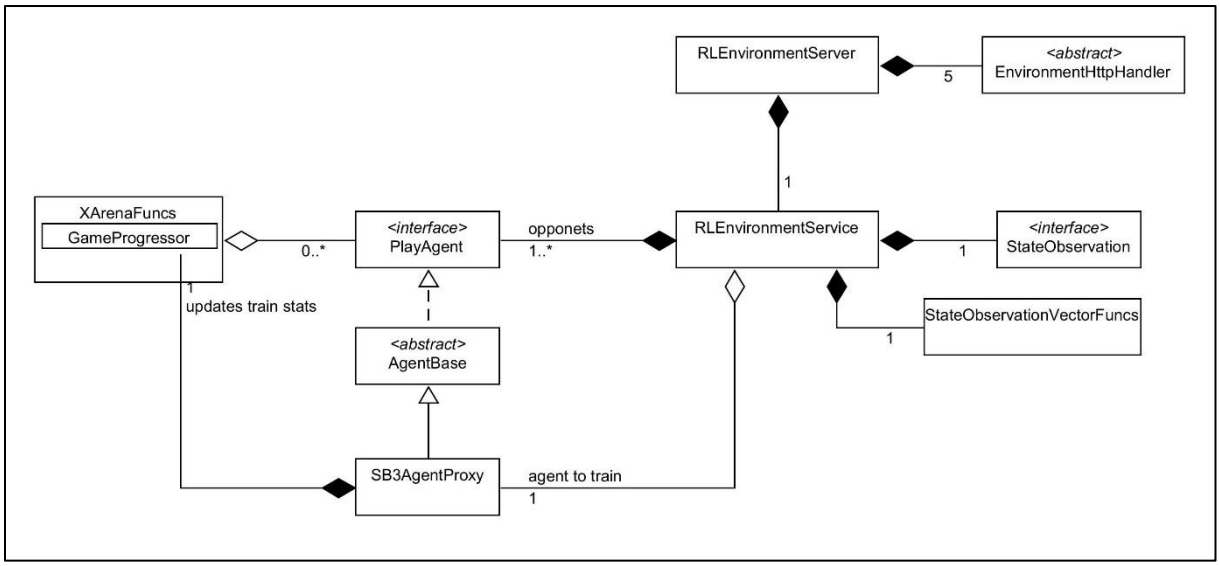


Abbildung 6: UML-Klassendiagramm des für die Integration von SB3-Agenten relevanten Teils des GBGs.

7 Funktionstests

In diesem Kapitel soll die allgemeine Funktionstüchtigkeit der für dieses Projekt implementierten Schnittstelle zwischen der SB3-Python-Anwendung und dem GBG getestet werden. Das heißt, es soll gezeigt werden, dass die SB3-Agenten in der Lage sind, über diese Schnittstelle trainiert zu werden und die Spiele im GBG anschließend mit einem gewissen Erfolg durch intelligente Strategien zu bewältigen.

Dazu werden die integrierten SB3-Agenten DQN, PPO und Maskable PPO an den Spielen Tic-Tac-Toe, Vier gewinnt und Nim getestet. Es sollen mehrere Tests durchgeführt werden. Der erste Test ist allgemein gehalten: Hier werden alle drei Agenten in den drei Spielen trainiert und getestet. Im nächsten Test werden die besten Agenten im jeweiligen Spiel mit verbesserten Hyperparametern und längeren Trainingszeiten erneut trainiert, um bessere Resultate zu erzielen, daraufhin treten diese Agenten gegen Agenten aus dem GBG an, um die Qualität der SB3-Agenten im Vergleich zu beurteilen.

Während des Trainings der Agenten wird vor allem untersucht, wie viel Zeit ein Agent zum Lernen benötigt und wie gut er sich gegen den Zufallsagenten des GBGs schlägt, der nur zufällige Aktionen ausführt. Die Trainingszeit wird in „Steps pro Sekunde“ gemessen und soll einen groben Überblick geben, in welchem Zeitraum ein Agent über diese Schnittstelle trainiert werden kann – auch wenn diese Zeit natürlich stark vom verwendeten Computersystem abhängt.

Der Erfolg gegen den Zufallsagenten wird über regelmäßige Evaluationsaufrufe im Trainingsverlauf gemessen. Er wird durch den durchschnittlichen Reward angegeben, den der Agent nach 600 Spielen gegen den Zufallsagenten erzielt hat. In den drei untersuchten Spielen erhält der Agent für einen Sieg einen Reward von 1, für eine Niederlage -1 und für ein Unentschieden – nur möglich in Tic-Tac-Toe und Vier gewinnt – einen Reward von 0.

Alle folgenden Tests werden auf dem gleichen Computersystem ausgeführt, das auch bereits für den Test von HTTP und Py4J als potenzielle Kommunikationsschnittstelle verwendet wurde (siehe Anhang 2.1, Tabelle 9).

7.1 Untersuchte Spiele

Im Rahmen dieses Projekts werden nur die Spiele Tic-Tac-Toe, Nim und Vier gewinnt des GBGs untersucht bzw. auf ihre Kompatibilität mit der Schnittstelle und den SB3-Agenten getestet. Diese drei Spiele eignen sich gut, da sie eine vergleichsweise geringe Komplexität aufweisen. Daher ist zu erwarten, dass ein gewisser Trainingserfolg auch ohne optimierte Hyperparameter erzielt werden kann.

Jedoch zeigen sich bereits in diesen drei Spielen erhebliche Unterschiede in der Komplexität bzw. der Größe der Zustandsräume. Bei Tic-Tac-Toe fällt diese im Vergleich zu Vier gewinnt zum Beispiel relativ gering aus. So kann die Performance der Schnittstelle in verschiedenen Schwierigkeitsstufen demonstriert werden – auch in Bezug auf die reine Trainingsdauer bei unterschiedlich großen Observationen.

7.2 Allgemeiner Test

In diesem Test werden alle integrierten Agenten (DQN, PPO, Maskable PPO) in den drei ausgewählten Spielen untersucht.

Die Hyperparameter sind so gewählt, dass sie für ein Spiel unter den verschiedenen Agenten möglichst vergleichbar sind, um herauszufinden, welcher Agent am geeignetsten für ein bestimmtes Spiel sein könnte – auch wenn der DQN-Algorithmus und die beiden PPO-Algorithmen teilweise unterschiedliche Hyperparameter besitzen. Zum Beispiel werden die Agenten in allen drei Spielen für 150.000 Steps trainiert, und die neuronalen Netze der drei Agenten sind jeweils für jedes Spiel identisch aufgebaut. Alle Hyperparameter sind im Anhang 2.1 dokumentiert.

Wie in der Abbildung 7 zu erkennen ist, ist in allen drei Spielen ist der Maskable-PPO-Algorithmus der langsamste und benötigt die längste Trainingszeit bzw. erzielt die wenigsten Steps pro Sekunde. Dies liegt unter anderem daran, dass für jeden Step sowohl der HTTP-Endpunkt `/step` als auch `/availableActions` der GBG-Schnittstelle aufgerufen werden muss. Der PPO-Algorithmus ist in allen drei Spielen der schnellste der getesteten Algorithmen. Des Weiteren ist ein Trend erkennbar, dass die Steps pro Sekunde mit einem größeren Observationsvektor der Spiele sinken, da mehr Daten zwischen den beiden Anwendungen übertragen werden müssen.

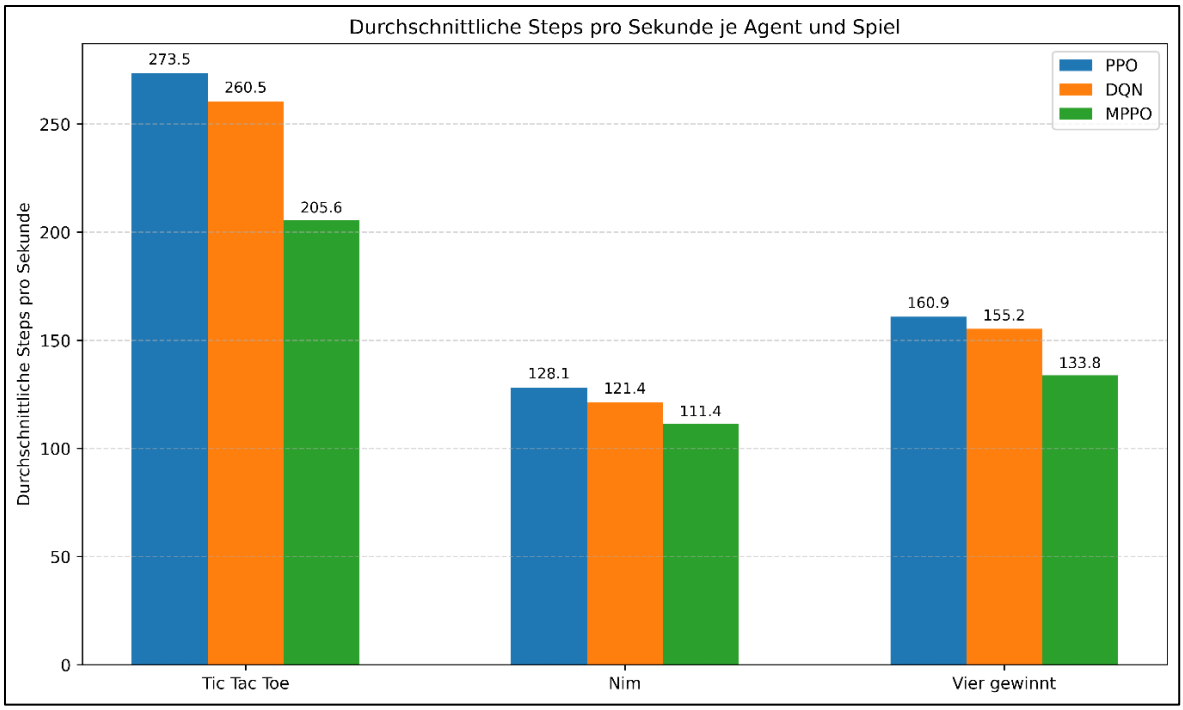


Abbildung 7: Durchschnittliche Steps pro Sekunde je Agent und Spiel im Allgemeinen Test.

Im Folgenden werden die Trainingsergebnisse der drei Agenten in den jeweiligen Spielen analysiert.

7.2.1 Testergebnisse im Spiel Tic-Tac-Toe

Tic-Tac-Toe ist das simpelste der drei untersuchten Spiele, und alle Agenten zeigen keine Schwierigkeiten, es innerhalb kurzer Trainingszeiten mit geringen Steps zu erlernen. Der durchschnittliche Reward gegen den Zufallsagenten steigt bei allen Agenten nahezu auf 0,9 (siehe Abbildung 8). Allerdings scheint es, als wäre der durchschnittliche Reward noch nicht vollständig konvergiert. Mit einer höheren Anzahl an Steps könnten wahrscheinlich ein noch besseres Ergebnis erreicht werden.

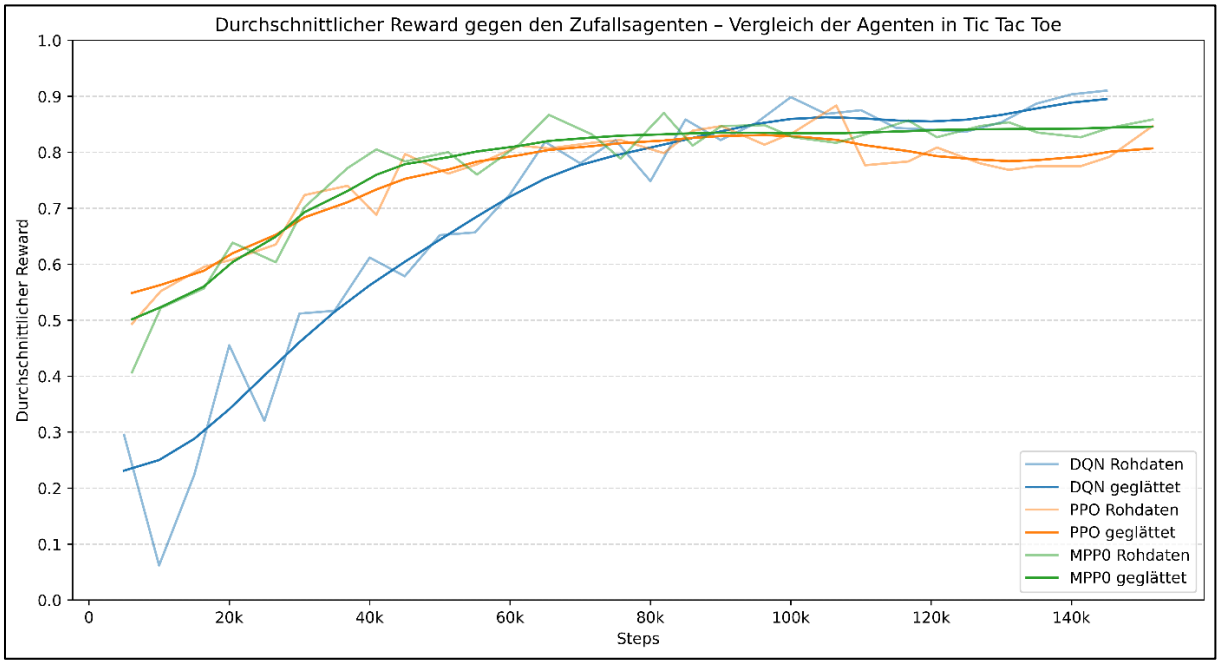


Abbildung 8: Durchschnittlicher Reward der RL-Agenten nach 600 Spielen in Tic-Tac-Toe gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in Tausenderschritten formatiert.

7.2.2 Testergebnisse im Spiel Nim

Für das Training der drei Agenten wurde das Spiel Nim so konfiguriert, dass drei Haufen mit jeweils zehn Elementen vorhanden sind und pro Zug eine beliebige Anzahl von Elementen aus einem Haufen gezogen werden kann.

Wie in Abbildung 9 zu sehen ist, erzielt der Maskable-PPO-Algorithmus in diesem Spiel die besten Ergebnisse. Dies ist vermutlich darauf zurückzuführen, dass in Nim während einer Spielrunde häufig States auftreten, in denen nur wenige gültige Aktionen zur Verfügung stehen. Der Maskable-PPO-Algorithmus ist speziell für solche Environments konzipiert, was seine überlegene Leistung erklärt (siehe Abschnitt 2.2.4).

Trotzdem liegt der Spitzenwert des durchschnittliche Reward lediglich bei etwas unter 0,7 – dem niedrigsten Spitzenwert unter den drei getesteten Spielen. Der DQN-Algorithmus erreicht in Nim die zweitbesten Resultate und könnte mit längerer Trainingsdauer sowie optimierten Hyperparametern eventuell vergleichbare Ergebnisse erzielen. Der klassische PPO-Algorithmus hingegen zeigt in diesem Spiel die schwächste Leistung und überschreitet in den Evaluationsspielen nicht einen durchschnittlichen Reward von 0,25.

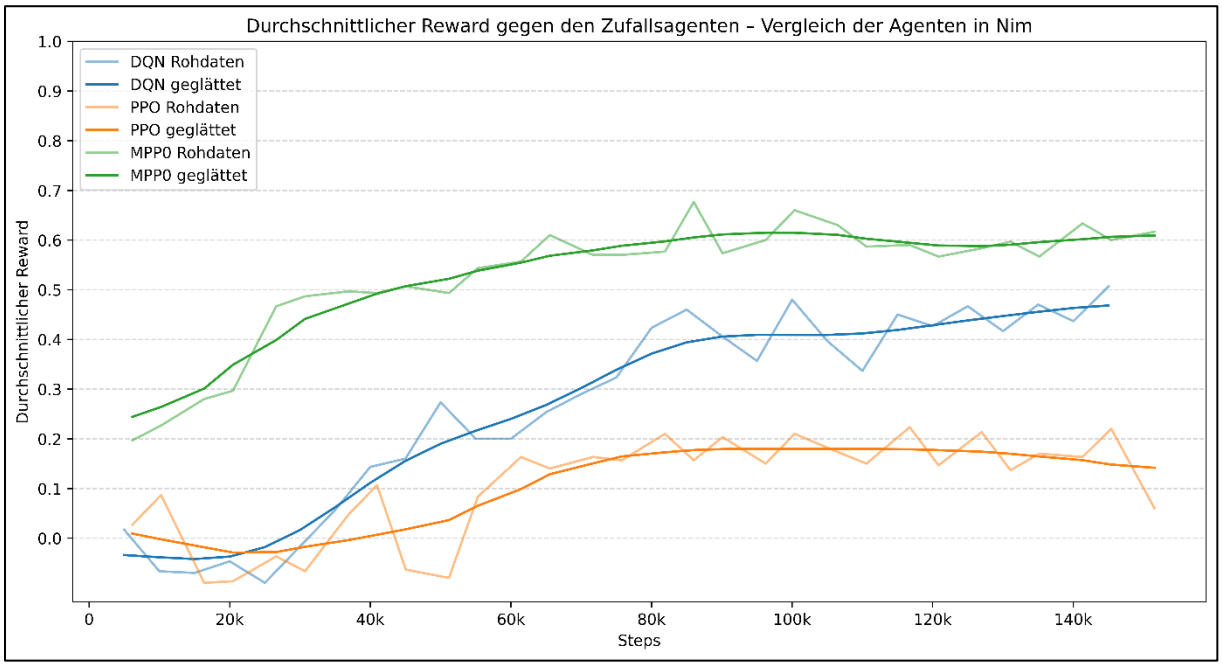


Abbildung 9: Durchschnittlicher Reward der RL-Agenten nach 600 Spielen in Nim gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in Tausenderschritten formatiert.

7.2.3 Testergebnisse im Spiel Vier gewinnt

Im Spiel Vier gewinnt kommen der PPO- sowie der Maskable-PPO-Algorithmus am Ende des Trainings auf einen ähnlichen durchschnittlichen Reward von etwa 0,85 nach einer Evaluation gegen den Zufallsagenten (siehe Abbildung 10).

Dass die Agenten beide sehr ähnliche Ergebnisse erzielen, lässt sich wohl dadurch erklären, dass es im Spiel Vier gewinnt nur selten zu Situationen kommt, in denen das Maskieren von Aktionen einen Vorteil bietet, da ungültige Aktionen erst in langen Spielrunden auftreten. Die durchschnittliche Spiellänge liegt jedoch in den Trainingsdurchläufen für beide Agenten zwischen 7 und 9 Runden. Das heißt, es kommen nur selten Stats mit ungültigen Aktionen vor, und die beiden Algorithmen verhalten sich dementsprechend nahezu identisch.

Der DQN-Algorithmus kommt in den Evaluationen nicht über einen durchschnittlichen Reward von 0,55 hinaus. Jedoch könnte dieser Wert mit längerer Trainingszeit noch steigen, da es scheint, als sei der Reward noch nicht vollständig konvergiert.

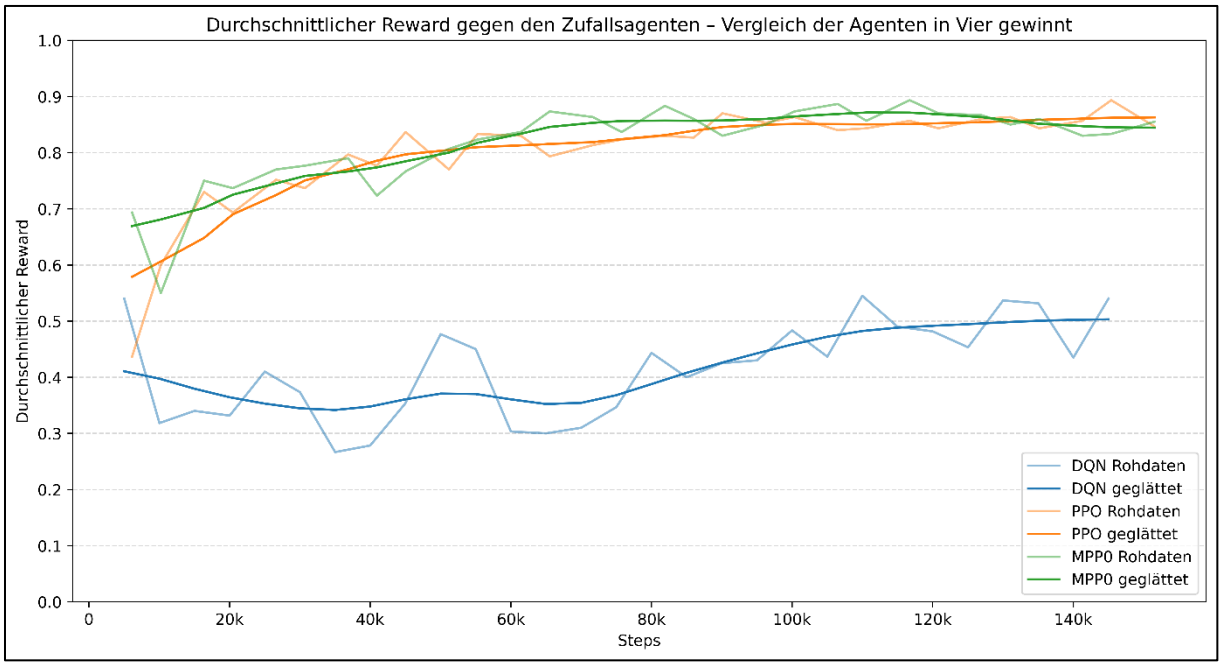


Abbildung 10: Durchschnittlicher Reward der RL-Agenten nach 600 Spielen in Vier gewinnt gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in Tausenderschritten formatiert.

7.3 Test mit Verbesserten Hyperparametern

Um das volle Potenzial der integrierten SB3-Agenten besser auszuschöpfen, wurden weitere Trainingsdurchläufe in allen drei Spielen durchgeführt. Dabei kamen jeweils die Agenten zum Einsatz, die im allgemeinen Test die besten Ergebnisse erzielt hatten. In diesen Durchläufen wurde mit einer höheren Anzahl an Steps gearbeitet; zudem wurden größere künstliche neuronale Netze eingesetzt, um die Leistung weiter zu steigern. Für jeden Agenten wurden mehrere Trainingsläufe durchgeführt, wobei die

Hyperparameter jeweils angepasst wurden. Die Konfiguration der Parameter der jeweils besten Trainingsläufe für jedes Spiel und jeden Agenten sind im Anhang 2.2 dokumentiert.

Nach dem Training der Agenten mit den verbesserten Hyperparametern wurden Testspiele gegen die Agenten des GBGs durchgeführt. Dazu wurden jeweils 1000 Spiele gegen alle für diese Spiele verfügbaren GBG-Agenten durchgeführt, die nicht vorab trainiert werden müssen. Die GBG-Agenten traten mit ihren voreingestellten Standardparametern für das jeweilige Spiel an. Um faire Bedingungen zu gewährleisten, wurden die SB3- und GBG-Agenten gleichmäßig auf die Spielerpositionen verteilt, also jeweils in gleichem Maße als Spieler mit dem ersten oder zweiten Zug eingesetzt.

Diese Testspiele sollen nicht ermitteln, welcher der verschiedenen Algorithmen für welches Spiel die besten Resultate erzielt, da auf beiden Seiten die Parameter und Trainingszeiten angepasst werden können, um bessere Ergebnisse zu erreichen. Vielmehr soll auch dieser Test als Funktionstest verstanden werden, der zeigt, dass sich die SB3-Agenten in das GBG-Framework einfügen und dort – wie die GBG-Agenten – über die „Competition“-Funktionen verwendet werden können.

Des Weiteren werden durch diesen Test Benchmarks gesetzt, bei denen die Performance der SB3-Agenten im Vergleich zu den GBG-Agenten unter bestimmten Hyperparametern betrachtet wird und es wird gezeigt wie die veränderten Hyperparameter die Leitung der Agenten beeinflusst hat.

Im Folgenden werden diese Trainingsdurchläufe, ihre spezifischen Hyperparameter und die Ergebnisse der Testspiele gegen die GBG-Agenten näher betrachtet.

7.3.1 Training und Testspiele des DQN-Algorithmus im Spiel Tic-Tac-Toe

Für das Spiel Tic-Tac-Toe fiel die Wahl auf den DQN-Agenten, da dieser im allgemeinen Test die besten Ergebnisse erzielt hatte. In diesem Trainingsdurchlauf wurde der Agent über 500.000 Steps trainiert und erreichte gegen Ende einen durchschnittlichen Reward von ca. 0,95 gegen den Zufallsagenten (siehe Abbildung 11). Damit übertraf er nochmals die Leistung aus dem allgemeinen Test (siehe Abbildung 8). Das Training dauerte insgesamt etwa 30 Minuten.

Die Testergebnisse gegen die GBG-Agenten zeigen, dass sich die Leistung des DQN-Agenten gegenüber dem allgemeinen Test in fast allen Fällen verbessert hat (siehe Tabelle 1 und 2).

Wie aus Tabelle 2 ersichtlich ist, beherrscht der optimierte DQN-Agent das Spiel Tic-Tac-Toe nahezu perfekt. Gegen die meisten GBG-Agenten endet ein Spiel nur noch unentschieden – ein Hinweis darauf, dass beide Seiten optimal spielen. Dennoch verliert der DQN-Agent gelegentlich gegen den Zufallsagenten, den Max-N- und den RHEA-SI-Agenten, was darauf schließen lässt, dass er noch nicht alle gegnerischen Strategien vollständig kontern kann.

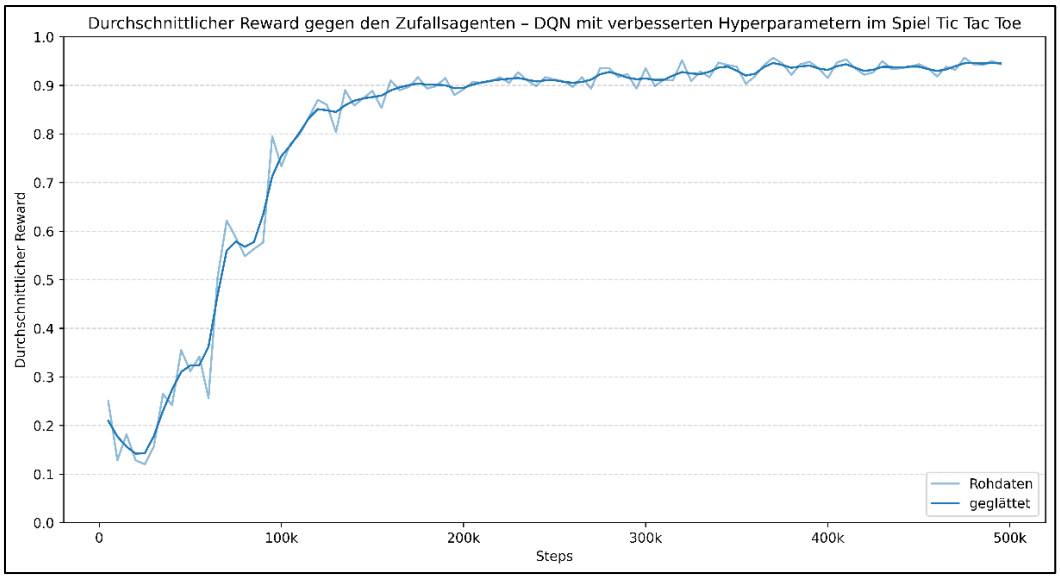


Abbildung 11: Durchschnittlicher Reward des DQN-Agenten nach 600 Spielen in Tic-Tac-Toe gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in Tausenderschritten formatiert.

Tabelle 1: Ergebnisse des DQN-Agenten aus dem allgemeinen Test im Spiel Tic-Tac-Toe: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten aus dem allgemeinen Test					
Agent	Gewonnen	Verloren	Unentschieden	Gewinnrate	Durchschnittlicher Reward
Random	910	37	53	0,937	0,873
Max-N	0	563	437	0,22	-0,563
MC-N	476	8	516	0,734	0,468
RHEA-SI	739	191	70	0,774	0,548
MCTS	15	164	821	0,425	-0,149
TD-Ntuple-4	572	407	21	0,583	0,165
TD-Ntuple-3	191	621	188	0,285	-0,43
Qlearn-4	0	500	500	0,25	-0,5
Sarsa-4	0	1000	0	0	-1

Tabelle 2: Ergebnisse des DQN-Agenten mit verbesserten Hyperparametern im Spiel Tic-Tac-Toe: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten mir den verbesserten Hyperparametern					
Agent	Gewonnen	Verloren	Unentschieden	Gewinnrate	Durchschnittlicher Reward
Random	948	2	50	0,973	0,946
Max-N	0	33	967	0,48	-0,033
MC-N	0	0	1000	0,5	0
RHEA-SI	797	20	183	0,889	0,777
MCTS	23	0	977	0,511	0,023
TD-Ntuple-4	0	0	1000	0,5	0
TD-Ntuple-3	0	0	1000	0,5	0
Qlearn-4	0	0	1000	0,5	0
Sarsa-4	0	0	1000	0,5	0

7.3.2 Training und Testspiele des Maskable-PPO-Algorithmus im Spiel Nim

Im Spiel Nim erwies sich der Maskable PPO-Agent als am besten geeignet, um in einem Environment zu lernen, das in vielen States eine Vielzahl ungültiger Aktionen enthält.

Der erfolgreichste Trainingsdurchlauf wurde mit 1,5 Millionen Steps durchgeführt. Das künstliche neuronale Netz wurde dabei deutlich vergrößert und umfasst nun sechs verborgene Schichten. Die erste und letzte Schicht bestehen aus jeweils 500 Neuronen, während die vier mittleren Schichten jeweils 6000 Neuronen enthalten. Um trotz der erhöhten Netzkomplexität eine akzeptable Trainingszeit zu gewährleisten, wurde der Trainingsprozess mit GPU-Unterstützung unter Linux ausgeführt. PyTorch unterstützt GPU-beschleunigtes Training nämlich nativ nur auf Linux-Systemen. Trotz dieser Optimierungen lag die Trainingsdauer bei etwa 14 Stunden.

Die Evaluationswerte gegen den Zufallsagenten zeigen im Vergleich zum allgemeinen Test bereits eine deutliche Verbesserung: Mit den neu konfigurierten Parametern erreicht der Agent einen durchschnittlichen Reward von knapp unter 0,9 (siehe Abbildung 12). Auch gegen die anderen Agenten des GBGs verbessert sich der MPPO-Agent erheblich im Vergleich zu seinem Vorgänger (siehe Tabelle 3 und 4). Besonders hervorzuheben ist die stark erhöhte Gewinnrate gegen den MCTS-Agenten sowie eine nahezu verdoppelte Gewinnrate gegen den Max-N-Agenten.

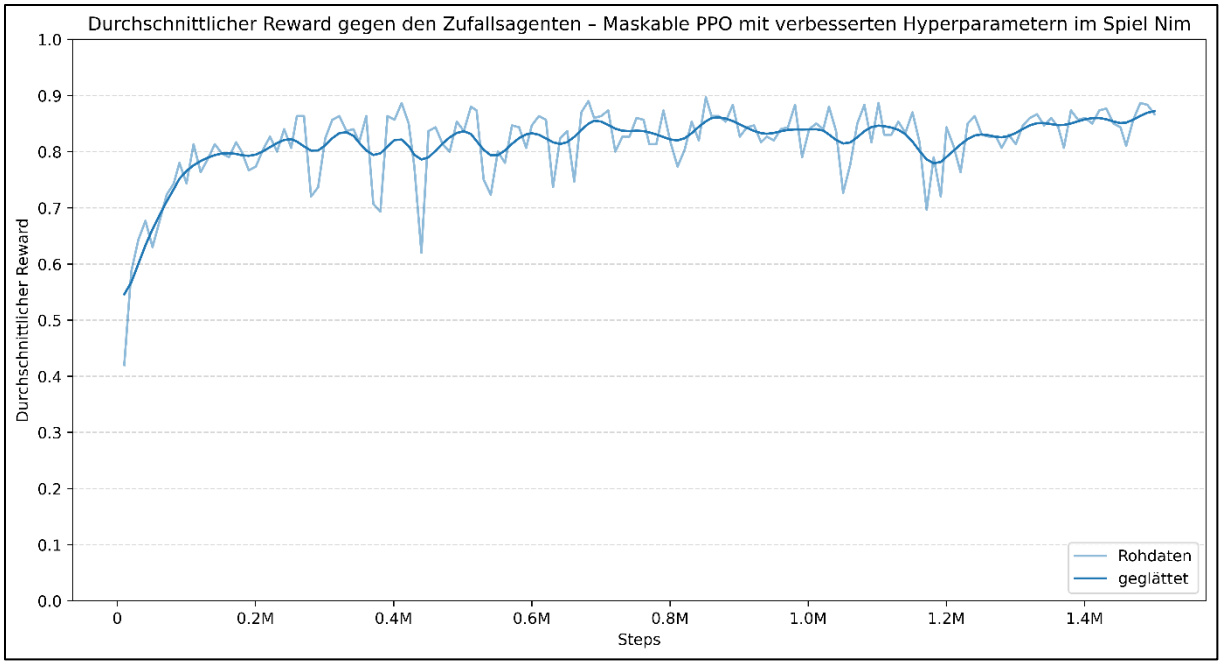


Abbildung 12: Durchschnittlicher Reward des Maskable PPO-Agenten nach 600 Spielen in Nim gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in millionstel Schritten formatiert.

Tabelle 3: Ergebnisse des Maskable PPO-Agenten aus dem allgemeinen Test im Spiel Nim: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten aus dem allgemeinen Test				
Agent	Gewonnen	Verloren	Gewinnrate	Durchschnittlicher Reward
Random	826	174	0,826	0,652
Max-N	404	596	0,40	-0,192
MC-N	411	589	0,411	-0,178
RHEA-SI	610	390	0,61	0,22
MCTS	41	959	0,041	-0,918
Bouton	0	1000	0	-1

Tabelle 4: Ergebnisse des Maskable PPO-Agenten mit verbesserten Hyperparametern im Spiel Nim: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten mir den verbesserten Hyperparametern				
Agent	Gewonnen	Verloren	Gewinnrate	Durchschnittlicher Reward
Random	929	71	0,929	0,858
Max-N	803	197	0,803	0,606
MC-N	646	354	0,646	0,292
RHEA-SI	751	249	0,751	0,502
MCTS	636	364	0,636	0,272
Bouton	0	1000	0	-1

7.3.3 Training und Testspiele des PPO-Algorithmus im Spiel Vier gewinnt

In den allgemeinen Tests schnitten im Spiel Vier gewinnt beide PPO-Agenten am besten ab. Da in diesem Spiel jedoch nur selten ungültige Aktionen maskiert werden müssen und die Trainingszeit beim normalen PPO-Agenten deutlich geringer ist als beim Maskable PPO-Agenten, wurde entschieden, die Hyperparameter des normalen PPO-Algorithmus weiter zu optimieren.

Der Agent wurde über 1,5 Millionen Steps trainiert. Das verwendete künstliche neuronale Netz besteht aus vier versteckten Schichten mit jeweils 6000 Neuronen. Wie im vorherigen Abschnitt beim Maskable PPO-Agenten wurde auch hier mit GPU-Unterstützung unter Linux trainiert. Die Trainingsdauer lag dennoch bei etwa 11 Stunden.

Die Evaluationsergebnisse gegen den Zufallsagenten (siehe Abbildung 13) zeigen keine wesentlichen Verbesserungen im Vergleich zum allgemeinen Test. Im direkten Vergleich mit den anderen Agenten des GBGs konnte dieser Agent jedoch etwas bessere Ergebnisse erzielen als sein Vorgänger aus dem allgemeinen Test (siehe Tabelle 5 und 6). Wenn jedoch der erheblich gestiegene Ressourcenaufwand und eine um ein Vielfaches höhere Trainingszeit betrachtet werden, fallen die Verbesserungen eher gering aus – das Training des PPO-Agenten im allgemeinen Test dauerte nur ca. 14 Minuten.

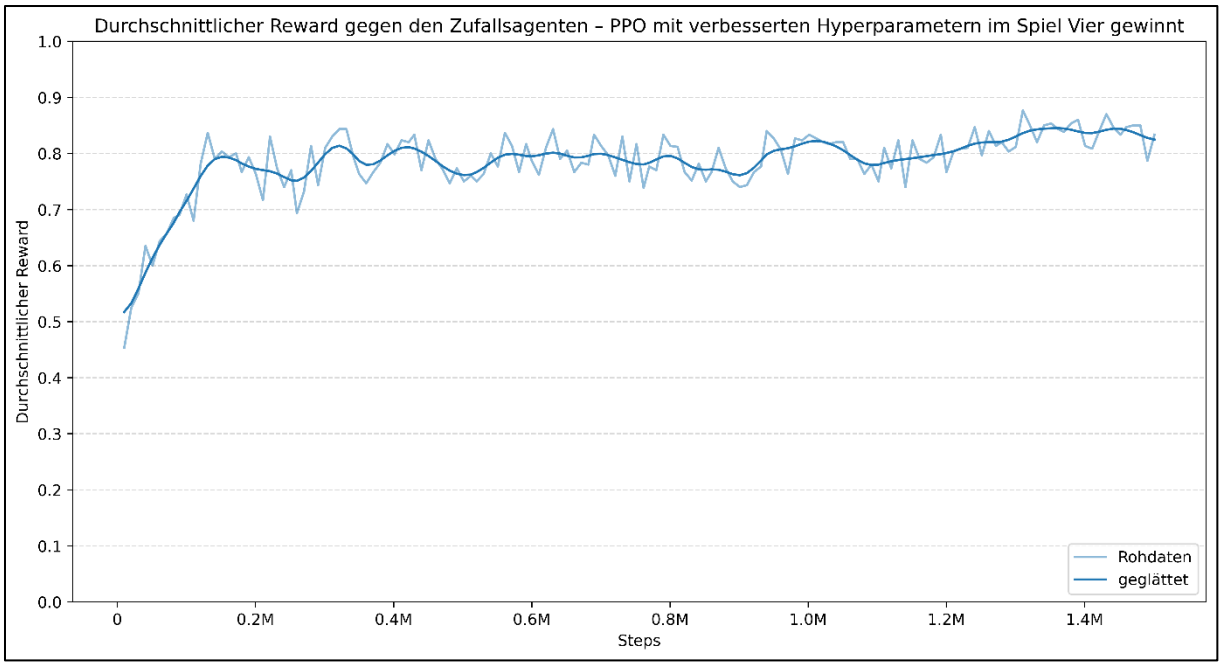


Abbildung 13: Durchschnittlicher Reward des PPO-Agenten nach 600 Spielen in Vier gewinnt gegen den Zufallsagenten über die Trainings-Steps. Die durchgezogenen Linien zeigen geglättete Werte, berechnet mit einem Gauß-Filter (Standardabweichung =2), um Trends klarer darzustellen. Halbtransparente Linien entsprechen den rohen Daten. Die x-Achse ist in millionstel Schritten formatiert.

Tabelle 5: Ergebnisse des PPO-Agenten aus dem allgemeinen Test im Spiel Vier gewinnt: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten aus dem allgemeinen Test					
Agent	Gewonnen	Verloren	Unentschieden	Gewinnrate	Durchschnittlicher Reward
Random	904	96	0	0,904	0,808
Max-N	52	948	0	0,052	-0,896
MC-N	32	968	0	0,032	-0,936
RHEA-SI	381	619	0	0,381	-0,238
MCTS	9	991	0	0,009	-0,982
AlphaBeta	128	872	0	0,128	-0,744
AlphaBeta-DL	6	994	0	0,006	-0,998

Tabelle 6: Ergebnisse des PPO-Agenten mit verbesserten Hyperparametern im Spiel Vier gewinnt: 1000 Runden gegen verschiedene GBG-Agenten.

Ergebnisse des Agenten mir den verbesserten Hyperparametern					
Agent	Gewonnen	Verloren	Unentschieden	Gewinnrate	Durchschnittlicher Reward
Random	918	82	0	0,918	0,836
Max-N	142	854	4	0,14	-0,712
MC-N	26	974	0	0,026	-0,948
RHEA-SI	410	590	0	0,41	-0,18
MCTS	10	990	0	0,01	-0,98
AlphaBeta	210	790	0	0,21	-0,58
AlphaBeta-DL	12	988	0	0,012	-0,976

8 Fazit und Ausblick

Das Ziel dieser Arbeit war die Integration von Deep-Reinforcement-Learning-Agenten des Python-Frameworks Stable-Baselines3 (SB3) in das in Java geschriebene General-Board-Game-Framework (GBG). Diese Integration sollte den Funktionsumfang des GBGs erheblich erweitern, da das Framework bisher nur RL-Algorithmen mit kleinen, flachen neuronalen Netzen implementierte und Java generell selten für die Implementierung tiefer neuronaler Netze genutzt wird. Die Nutzung von SB3 ermöglicht den Zugriff auf Deep-RL-Agenten mit tiefen neuronalen Netzen.

Eine zentrale Herausforderung dieses Projekts lag im technologischen Bruch zwischen Java und Python, der die Schaffung einer bidirektionalen Kommunikationsschnittstelle erforderte. Im Rahmen erster Recherchen wurden HTTP und Py4J als geeignete Technologien für diese Schnittstelle identifiziert. Ein praktischer Test wurde durchgeführt, um die Eignung und insbesondere die Performance beider Optionen zu evaluieren. Die Ergebnisse dieses Tests zeigen klar, dass eine HTTP-Schnittstelle deutlich performanter ist als eine Implementierung mit Py4J (siehe Kapitel 5). Die Performanceunterschiede zwischen den beiden Optionen steigen dabei insbesondere mit Größe des Observationsvektors erheblich an. Obwohl Py4J eine dynamischere Interaktion ermöglicht und einfacher zu implementieren wäre, rechtfertigte dies den gemessenen Performanceverlust nicht, weshalb die Entscheidung zugunsten einer HTTP-Schnittstelle fiel.

Die konkrete Integration von SB3-Agneten – DQN, PPO und Maskable PPO - in das GBG erfolgte durch die Entwicklung der SB3-Python-Anwendung, sowie Anpassungen auf der Seite des GBGs (siehe Kapitel 6). Dazu wurde auf beiden Seiten eine HTTP-Schnittstelle implementiert, um eine bidirektionale Kommunikation zu ermöglichen. Die SB3-Python-Anwendung nutzt FastAPI zum Hosting des HTTP-Servers und zur Definition der HTTP-Endpunkte. Auf Java-Seite kommt ein leichtgewichtiger HTTP-Server aus dem `com.sun.net.httpserver`-Paket zum Einsatz. Die Kommunikation von Python zu Java wird mittels der requests-Bibliothek realisiert, während Java den `java.net.http.HttpClient` nutzt.

Eine weitere wichtige Entscheidung, während der Implementierung, war die Verlagerung der Kontrolle über die Trainingsschleife zur SB3-Python-Anwendung. Dies ermöglichte eine einfachere Einbindung der SB3-Algorithmen und deren spezifischer `learn`-Methoden, auch wenn dies eine Anpassung der Trainingslogik im GBG erforderte.

Um SB3 nutzen zu können, welches ein Gymnasium-kompatibles Environment benötigt, wurde eine Schnittstelle im GBG (`RLEnvironmentService`) implementiert, die das Environment nach außen als Gymnasium-Environment (mittels `GBGEnvironmentClient` auf Python-Seite) abbildet. Dies beinhaltet die

Implementierung der zentralen `step`- und `reset`-Methoden, sowie `action_masks` für den Maskable PPO Algorithmus.

Funktionstests wurden an den Spielen Tic-Tac-Toe, Nim und Vier gewinnt durchgeführt, um die Funktionstüchtigkeit der Schnittstelle und die Lernfähigkeit der integrierten SB3-Agenten (DQN, PPO, Maskable PPO) zu überprüfen (siehe Kapitel 7). Diese Spiele wurden aufgrund ihrer unterschiedlichen Komplexität ausgewählt, um die Performance der Schnittstelle und der Agenten unter verschiedenen Bedingungen zu demonstrieren. Die Tests zeigen, dass die SB3-Agenten in der Lage sind, über die Schnittstelle zu lernen und in den Spielen intelligente Strategien zu entwickeln. Dazu wurde durchschnittlichen Reward gegen den Zufallsagenten, der immer zufällige Aktionen auswählt, über die Trainings-Steps gemessen. Die Kurve dieser Rewards lässt bei alle getestet Agenten und Spiele, darauf schließen, dass die Agenten die Spiele erfolgreich erlernen. Insbesondere der Maskable-PPO-Algorithmus erwies sich in Nim als überlegen, was seine Eignung für Environments mit vielen ungültigen Aktionen unterstreicht. Mit verbesserten Hyperparametern und längeren Trainingszeiten konnten die Agenten ihre Leistung weiter steigern und sich auch erfolgreich gegen verschiedene vorhandene GBG-Agenten behaupten, was die erfolgreiche Integration und Nutzbarkeit der SB3-Agenten innerhalb des GBG-Frameworks bestätigte.

Zusammenfassend lässt sich sagen, dass dieses Projekt die erfolgreiche Integration von SB3-Deep-RL-Agenten in das Java-basierte GBG-Framework durch die Implementierung einer bidirektionalen HTTP-Schnittstelle ermöglicht hat. Dies erweitert die Möglichkeiten des GBGs erheblich und schafft eine Plattform zur Nutzung und zum Vergleich moderner Deep-RL-Algorithmen in Brettspielen. Die Arbeit liefert nicht nur die technische Grundlage für diese Integration, sondern auch wertvolle Erkenntnisse über die Performance von Kommunikationslösungen zwischen Java und Python im Kontext von Reinforcement Learning und die Anwendbarkeit spezifischer SB3-Algorithmen auf Brettspiele unterschiedlicher Komplexität.

In Zukunft könnte es interessant sein, die integrierten SB3-Agenten an noch mehr Spielen zu testen oder weitere Agenten aus SB3 zu integrieren. Des Weiteren könnten den SB3-Agenten mehr Features zur Verfügung gestellt werden, da sie momentan nur die rohen Board-Vektoren zum Trainieren verwenden. Dazu wurde die Klasse `StateObservationVectorFuncs` bewusst so konzipiert, dass sich leicht auch andere Features der Spiele in die Observationsvektoren integrieren lassen. Eine Möglichkeit wäre es z. B., zu versuchen, die SB3-Agenten mit den n-Tuple-Features trainieren zu lassen. Dadurch könnte die im Vergleich schwache Leistung der SB3-Agenten im Spiel Vier gewinnt wahrscheinlich deutlich gesteigert werden.

9 Literaturverzeichnis

[1] M. L. Puterman, „Chapter 8 Markov decision processes“, in *Handbooks in Operations Research and Management Science*, Bd. 2, Elsevier, 1990, S. 331–434. doi: 10.1016/S0927-0507(05)80172-0.

[2] V. Mnih *u. a.*, „Playing Atari with Deep Reinforcement Learning“, 19. Dezember 2013, *arXiv*: arXiv:1312.5602. doi: 10.48550/arXiv.1312.5602.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, und O. Klimov, „Proximal Policy Optimization Algorithms“, 28. August 2017, *arXiv*: arXiv:1707.06347. doi: 10.48550/arXiv.1707.06347.

[4] D. Bick, „Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization“, Master’s Research Project, University of Groningen, 2021.

[5] S. Huang und S. Ontañón, „A Closer Look at Invalid Action Masking in Policy Gradient Algorithms“, *Int. FLAIRS Conf. Proc.*, Bd. 35, Mai 2022, doi: 10.32473/flairs.v35i.130584.

[6] W. Konen, „The GBG Class Interface Tutorial V2.3: General Board Game Playing and Learning“, Cologne Institute of Computer Science, TH Köln, Deutschland, Technischer Bericht, Sep. 2022.

[7] *GBG: General Board Game Playing*. Java. Zugegriffen: 4. Juni 2025. [Online]. Verfügbar unter: https://github.com/WolfgangKonen/GBG

[8] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, und N. Dormann, „Stable-Baselines3: Reliable Reinforcement Learning Implementations“, *J. Mach. Learn. Res.*, Bd. 22, Nr. 268, S. 1–8, 2021.

[9] L. Engstrom *u. a.*, „Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO“, 25. Mai 2020, *arXiv*: arXiv:2005.12729. doi: 10.48550/arXiv.2005.12729.

[10] M. Towers *u. a.*, „Gymnasium: A Standard Interface for Reinforcement Learning Environments“, 8. November 2024, *arXiv*: arXiv:2407.17032. doi: 10.48550/arXiv.2407.17032.

[11] „Tensorboard Integration — Stable Baselines3 2.6.1a1 documentation“. Zugegriffen: 1. Juni 2025. [Online]. Verfügbar unter: https://stable-baselines3.readthedocs.io/en/master/guide/tensorboard.html

[12] H. Nielsen *u. a.*, „Hypertext Transfer Protocol – HTTP/1.1“, Internet Engineering Task Force, Request for Comments RFC 2616, Juni 1999. doi: 10.17487/RFC2616.

[13] Uttarakhand Technical University, G. Bora, S. Bora, S. Singh, und S. M. Arsalan, „OSI Reference Model: An Overview“, *Int. J. Comput. Trends Technol.*, Bd. 7, Nr. 4, S. 214–218, Jan. 2014, doi: 10.14445/22312803/IJCTT-V7P151.

[14] „Welcome to Py4J — Py4J“. Zugegriffen: 5. April 2025. [Online]. Verfügbar unter: https://www.py4j.org/

[15] „FastAPI“. Zugegriffen: 1. Juni 2025. [Online]. Verfügbar unter: https://fastapi.tiangolo.com/

[16] „Uvicorn“. Zugegriffen: 1. Juni 2025. [Online]. Verfügbar unter: https://www.uvicorn.org/

[17] „Starlette“. Zugegriffen: 1. Juni 2025. [Online]. Verfügbar unter: https://www.starlette.io/

[18] „Welcome to Pydantic - Pydantic“. Zugegriffen: 1. Juni 2025. [Online]. Verfügbar unter: https://docs.pydantic.dev/latest/

[19] „TechEmpower Web Framework Performance Comparison“, www.techempower.com. Zugegriffen: 5. April 2025. [Online]. Verfügbar unter: https://www.techempower.com/benchmarks/#section=data-r23&hw=ph&test=query&l=zijzen-7

[20] *requests: Python HTTP for Humans.* Python. Zugegriffen: 7. April 2025. [OS Independent]. Verfügbar unter: https://requests.readthedocs.io

[21] L. Püschel, *Py4J und HTTP Test - Java Anwendung*. Java. Zugegriffen: 4. Juni 2025. [Online]. Verfügbar unter: https://github.com/Jack00mie/py4JTestJava

[22] L. Püschel, *Py4J und HTTP Test - Python Anwendung*. Python. Zugegriffen: 4. Juni 2025. [Online]. Verfügbar unter: https://github.com/Jack00mie/py4jTest

Anhang 1: HTTP und Py4J Test

Tabelle 7: Testergebnisse mit der HTTP-Schnittstelle.
Die gemessenen Zeiten in Millisekunden (ms) für jede Testrunde der Testdurchläufe mit unterschiedlichen Größen des Observationsvektors sind in den Tabellen dargestellt. Unter den Tabellen sind die Mittelwerte, Standardabweichungen und Standardfehler der Mittelwerte jeweils für den entsprechenden Testdurchlauf aufgeführt.

	HTTP-Schnittstelle					
	Testrunde	Testdurchläufe mit unterschiedlich großen Observationsvektoren				
		10	20	50	200	1000
	1	2434	2639	3175	5730	19711
	2	2465	2612	3110	5692	20214
	3	2443	2630	3187	5751	20028
	4	2459	2598	3159	5749	19568
	5	2390	2647	3135	5812	19602
	6	2417	2651	3244	5803	19677
	7	2387	2643	3219	5824	19360
	8	2469	2610	3216	5775	19671
	9	2461	2611	3160	5721	19703
	10	2428	2621	3184	5711	19508
	11	2468	2606	3170	5807	19456
	12	2455	2668	3215	5692	19749
	13	2422	2638	3206	5719	19595
	14	2424	2590	3095	5679	19394
	15	2451	2609	3177	5662	19572
Mittelwert:		2438,2	2624,9	3176,8	5741,8	19653,9
Standartabweichung:		26,7	22,2	41,3	52,3	224,2
Standartabweichung realtiv zum Mittelwert (Variationskoeffizient):		0,0109	0,0085	0,0130	0,0091	0,0114
Geschätzter Standardfehler des Mittelwertes (SEM):		6,89	5,73	10,66	13,49	57,89
SEM relativ zum Mittelwert:		0,0028	0,0022	0,0034	0,0023	0,0029

Tabelle 8: Testergebnisse mit der Py4J-Schnittstelle.
Die gemessenen Zeiten in Millisekunden (ms) für jede Testrunde der Testdurchläufe mit unterschiedlichen Größen des. Observationsvektors sind in den Tabellen dargestellt. Unter den Tabellen sind die Mittelwerte, Standardabweichungen und Standardfehler der Mittelwerte jeweils für den entsprechenden Testdurchlauf aufgeführt

	Py4J-Schnittstelle				
Testrunde	Testdurchläufe mit unterschiedlich großen Observationsvektoren				
	10	20	50	200	1000
1	3326	4016	5861	15758	65681
2	3327	3990	5934	15847	65577
3	3340	3987	5968	15883	65924
4	3351	4000	5915	15887	65455
5	3336	3975	5922	15904	65223
6	3310	3941	5866	15943	65258
7	3326	3997	5958	15716	65371
8	3416	4045	5965	15721	65597
9	3333	4010	5968	15698	65827
10	3367	3976	5955	15768	65326
11	3356	4021	6016	15844	65618
12	3334	4027	5952	15718	65853
13	3359	3971	5924	15883	65157
14	3341	3979	5848	15761	65404
15	3362	4023	5915	15745	65103
Mittelwert:	3345,6	3997,2	5931,1	15805,1	65491,6
Standartabweichung:	25,1	27,0	45,9	82,0	259,3
Standartabweichung realtiv zum Mittelwert (Variationskoeffizient):	0,0075	0,0068	0,0077	0,0052	0,0040
Geschätzter Standardfehler des Mittelwertes (SEM):	6,48	6,97	11,86	21,17	66,94
SEM relativ zum Mittelwert:	0,0019	0,0017	0,0020	0,0013	0,0010

Tabelle 9: Hardwarespezifikationen.
Hardwarespezifikationen des für die Tests verwendet Computersystems.

CPU					RAM	GPU				
Name	Kerne	Threads	Basistaktung	Boost-Taktung	Kapazität	Name	Grafikspeicher	Basistaktung	Boost-Taktung	CUDA-Recheneinheiten
AMD Ryzen™ 7 5800X	8	16	3.8 GHz	4.7 GHz	32 GB DDR4	GeForce RTX 4060 Ti	16 GB GDDR6	2,31 GHz	2,54 GHz	4352

Anhang 2: Hyperparameter der Agenten

Anhang 2.1: Hyperparameter der Agenten im allgemeinen Test

Tabelle 10: Hyperparameter des DQN-Agenten im allgemeinen Test.
Hyperparameter des Agenten in den drei getesteten Spielen. Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

	Vier gewinnt	Nim	Tic Tac Toe
Agent	DQN	DQN	DQN
Batch size	32	32	32
Buffer size	1000000	1000000	1000000
Evaluation options / Evaluate every X steps	5000	5000	5000
Evaluation options / Number of games	300	300	300
Evaluation options / Opponents	Self Play, Random, MC-N	Self Play, Random, MC-N	Self Play, Random, MC-N
Evaluation options / Save best	1	1	1
Exploration final epsilon	0,05	0,05	0,05
Exploration fraction	0,5	0,5	0,5
Exploration initial epsilon	1	1	1
Gamma	0,99	0,99	0,99
Gradient steps	1	1	1
Layer 0	200	220	180
Layer 1	200	220	180
Layer 2	200	220	180
Learning rate	0,0001	0,0001	0,0001
Learning starts at	100	100	100
Max gradient norm	10	10	10
Optimize memory usage	0	0	0
Self play parameters / Add policy every steps	3000	3000	3000
Self play parameters / Policy window size	30	30	30
Self play parameters / Use latest policy ratio	0,35	0,25	0,25
Stats window size	100	100	100
Target update interval	10000	10000	10000
Tau	1	1	1
Train frequency	4	4	4

Tabelle 11: Hyperparameter des Maskable PPO-Agenten im allgemeinen Test.
Hyperparameter des Agenten in den drei getesteten Spielen. Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

	Vier gewinnt	Nim	Tic Tac Toe
Agent	MaskablePPO	MaskablePPO	MaskablePPO
Train time steps	150000	150000	150000
Batch size	64	64	64
Clip range	0,2	0,2	0,2
Entropy coefficient	0	0	0
Evaluation options / Evaluate every X steps	5000	5000	5000
Evaluation options / Number of games	300	300	300
Evaluation options / Opponents	Self Play, Random, MC-N	Self Play, Random, MC-N	Self Play, Random, MC-N
Evaluation options / Save best	1	1	1
gae_lambda	0,95	0,95	0,95
gamma	0,99	0,99	0,99
Layer 0	200	220	180
Layer 1	200	220	180
Layer 2	200	220	180
Learning rate	0,0001	0,0001	0,0001
Max gradient norm	0,5	0,5	0,5
n_epochs	10	10	10
n_steps	2048	2048	2048
Normalize advantage	1	1	1
Self play parameters / Add policy every steps	3000	3000	3000
Self play parameters / Policy window size	30	30	30
Self play parameters / Use latest policy ratio	0,35	0,25	0,25
Stats window size	100	100	100
vf_coef	0,5	0,5	0,5

Tabelle 12: Hyperparameter des PPO-Agenten im allgemeinen Test.
Hyperparameter des Agenten in den drei getesteten Spielen. Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

	Vier gewinnt	Nim	Tic Tac Toe
Agent	PPO	PPO	PPO
Train time steps	150000	150000	150000
Batch size	64	64	64
Clip range	0,2	0,2	0,2
Entropy coefficient	0	0	0
Evaluation options / Evaluate every X steps	5000	5000	5000
Evaluation options / Number of games	300	300	300
Evaluation options / Opponents	Self Play, Random, MC-N	Self Play, Random, MC-N	Self Play, Random, MC-N
Evaluation options / Save best	1	1	1
gae_lambda	0,95	0,95	0,95
Gamma	0,99	0,99	0,99
Layer 0	200	220	180
Layer 1	200	220	180
Layer 2	200	220	180
Learning rate	0,0001	0,0001	0,0001
Max gradient norm	0,5	0,5	0,5
n_epochs	10	10	10
n_steps	2048	2048	2048
Normalize advantage	1	1	1
SDE sample frequency	-1	-1	-1
Self play parameters / Add policy every steps	3000	3000	3000
Self play parameters / Policy window size	30	30	30
Self play parameters / Use latest policy ratio	0,35	0,25	0,25
Stats window size	100	100	100
vf_coef	0,5	0,5	0,5

Anhang 2.2: Verbesserte Hyperparameter der Agenten

Tabelle 13: Verbesserte Hyperparameter des DQN-Agenten im Spiel Tic-Tac-Toe.
Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

Agent	DQN
Train time steps	500.000
Batch size	32
Buffer size	1.000.000
Evaluation options / Evaluate every X steps	5000
Evaluation options / Number of games	300
Evaluation options / Opponents	Self Play, Random, MCTS
Evaluation options / Save best	1
Exploration final epsilon	0,05
Exploration fraction	1
Exploration initial epsilon	1
Gamma	1
Gradient steps	1
Layer 0	180
Layer 1	180
Layer 2	180
Learning rate	0,00007
Learning starts at	100
Max gradient norm	10
Optimize memory usage	0
Self play parameters / Add policy every steps	3000
Self play parameters / Policy window size	30
Self play parameters / Use latest policy ratio	0,25
Stats window size	100
Target update interval	10000
Tau	1
Train frequency	4

Tabelle 14: Verbesserte Hyperparameter des Maskable PPO-Agenten im Spiel Nim.
Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

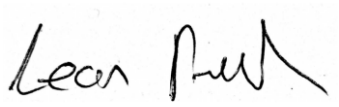
Agent	Maskable PPO
Train time steps	1.500.000
Batch size	64
Clip range	0,2
Entropy coefficient	0
Evaluation options / Evaluate every X steps	10000
Evaluation options / Number of games	300
Evaluation options / Opponents	Self Play, Random, MCTS
Evaluation options / Save best	1
gae_lambda	0,95
Gamma	0,99
Layer 0	500
Layer 1	6000
Layer 2	6000
Layer 3	6000
Layer 4	6000
Layer 5	500
Learning rate	0,0001
Max gradient norm	0,5
n_epochs	8
n_steps	2048
Normalize advantage	1
Self play parameters / Add policy every steps	15000
Self play parameters / Policy window size	12
Self play parameters / Use latest policy ratio	0,3
Stats window size	100
vf_coef	0,5

Tabelle 15: Verbesserte Hyperparameter des PPO-Agenten im Spiel Vier gewinnt.
Die Hyperparameter sind gleich oder ähnlich benannt, wie sie auch im Parameter-Tab, SB3, des GBGs eingestellt werden können.

Agent	PPO
Train time steps	1500000
Batch size	64
Clip range	0,2
Entropy coefficient	0
Evaluation options / Evaluate every X steps	10000
Evaluation options / Number of games	300
Evaluation options / Opponents	Self Play, Random, MCTS
Evaluation options / Save best	1
gae_lambda	0,95
Gamma	0,99
Layer 0	6000
Layer 1	6000
Layer 2	6000
Layer 3	6000
Learning rate	0,0001
Max gradient norm	0,5
n_epochs	13
n_steps	2048
Normalize advantage	1
SDE sample frequency	-1
Self play parameters / Add policy every steps	15000
Self play parameters / Policy window size	10
Self play parameters / Use latest policy ratio	0,3
Stats window size	100
vf_coef	0,5

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.



Köln, 06.06.2025