Carl von Ossietzky Universität Oldenburg

Informatik
Bachelorarbeit

Vergleich von verschiedenen Deep Reinforcement Learning Agenten am Beispiel des Videospiels Snake

---

Autor:
Lorenz Mumm

Erstgutachter:
Apl. Prof. Dr. Jürgen Sauer

Zweitgutachter:
M. Sc. Julius Möller

Abteilung Systemanalyse und -optimierung
Department für Informatik

Oldenburg, 29. August 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

KI	**K**ünstliche **I**ntelligenz
RL	**R**einforcement **L**earning
DQN	**D**eep **Q**-**N**etwork
DQL	**D**eep **Q**-**L**earning
DDQN	**D**ouble **D**eep **Q**-**N**etwork
PPO	**P**roximal **P**olicy **O**ptimization
SAC	**S**oft **A**ctor **C**ritic
A2C	Advantage **A**ctor **C**ritic
Env	**Env**ironment
Obs	**Obs**ervation
AV	**a**round_**v**iew
SO	**s**calar_**o**bs

Kapitel 1

Einleitung

Das Maschine Learning ist weltweit auf dem Vormarsch und große Unternehmen investieren riesige Beträge, um mit KI basierten Lösungen größere Effizienz zu erreichen. Auch der Bereich des Reinforcement Learning gerät dabei immer mehr in das Blickfeld von der Weltöffentlichkeit. Besonders im Gaming Bereich hat das Reinforcement Learning schon beeindruckende Resultate erbringen können, wie z.B. die KI AlphaGO, welche den amtierenden Weltmeister Lee Sedol im Spiel GO besiegt hat Chunxue u. a. (2019). In Anlehnung an die vielen neuen Reinforcement Learning Möglichkeiten, die in der letzten Zeit entwickelt wurden und vor dem Hintergrund der immer größer werdenden Beliebtheit von KI basierten Lösungsstrategien, soll es in dieser Bachelorarbeit darum gehen, einzelnen Reinforcement Learning Agenten, mittels statistischer Methoden, genauer zu untersuchen und den optimalen Agenten für ein entsprechendes Problem zu bestimmen.

1.1. Motivation

In den letzten Jahren erregte das Reinforcement Learning eine immer größere Aufmerksamkeit. Siege über die amtierenden Weltmeister in den Spielen GO oder Schach führten zu einer zunehmenden Beliebtheit des RL. Neue Verfahren, wie z.B. der Deep Q-Network Algorithmus auf dem Jahr 2015 Mnih u. a. (2015), der Proximal Policy Optimization (PPO) aus dem Jahr 2017 Schulman u. a. (2017) oder der Soft Actor Critic (SAC) aus dem Jahr 2018 Haarnoja u. a. (2018), haben ihr Übriges getan, um das RL auch in anderen Bereichen weiter anzusiedeln, wie z.B. der Finanzwelt, im autonomen Fahren, in der Steuerung von Robotern, in der Navigation im Web oder als Chatbot Lapan (2020).
Durch die jedoch große Menge an RL-Verfahren gerät man zunehmend in die problematische Situation, sich für einen diskreten RL Ansatz zu entscheiden. Weiter erschwert wird dieser Auswahlprozess noch durch die Tatsache, dass die einzelnen Agenten jeweils untereinander große Unterschiede aufweisen. Auch existieren häufig mehrere Ausprägungen eines RL-Verfahrens, wie z.B. der Deep Q-Network Algorith-

mus (DQN) und der Double Deep Q-Network Algorithmus (DDQN). Die Wahl des passenden Agenten kann großen Einfluss auf die Performance und andere Bewertungskriterien haben, (Bowei Ma (2016)), deshalb soll in dieser Ausarbeitung eine Vorgehen, welches auf einem Vergleich beruht, entwickelt werden, dass den optimalen Agenten für eine entsprechendes Problem bestimmt.

Eine potenzielle Umgebung, in welcher Agenten getestet und verglichen werden können, ist das Spiel Snake. Mit der Wahl dieses Spieles ist zusätzlich zu dem oben erwähnten Mehrwert noch ein weiterer in Erscheinung getreten.
So interpretieren neue Forschungsansätze das Spiel Snake als ein dynamisches Pathfinding Problem. Mit dieser Art von Problemen sind auch unbemannte Drohne (UAV Drohnen) konfrontiert, welche beispielsweise Menschen in komplexen Katastrophensituationen, wie z.B. auf explodierten Rohölbohrplattformen oder Raffinerien, finden und retten sollen. Auch kann das Liefen von wichtigen Gütern, beispielsweise medizinischer Natur, in solche Gebiete kann durch die Forschung am Spiel Snake möglich gemacht werden.(Chunxue u. a. (2019))

1.2. Zielsetzung

Basierend auf der Motivation ergibt sich folgende Fragestellung für diese Ausarbeitung:
Wie kann an einem Beispiel des Spiels Snake für eine nicht-triviale gering-dimensionale Umgebung ein möglichst optimaler RL Agent ermittelt werden?
Diese Fragestellung zielt darauf ab für die Umgebung Sanke einen möglichst optimalen Agenten zu bestimmen, welcher spezifische Anforderungen erfüllen soll.
Basierend auf der Forschungsfrage ergibt sich ein Mehrwert für die Wissenschaft. Durch Abnahme des Entscheidungsfindungsprozesses müssen Forscherinnen und Forscher wie auch Anwenderinnen und Anwender von RL-Verfahren nicht mehr unreflektiert irgendeinen RL Agenten auswählen, sondern können auf Grundlage der Methodik und der daraus hervorgehenden Daten den passenden Agenten bestimmen.

Kapitel 2

Grundlagen

Im folgenden Kapitel soll das benötigte Wissen vermittelt werden, welcher zum Verständnis dieser Arbeit benötigt wird. Dabei sollen verschiedene Reinforcement Learning Algorithmen, wie auch grundlegende Informationen des Reinforcement Learnings selbst thematisiert werden. Auch das Spiel Snake wird Erwähnung finden.

2.1. Game of Snake

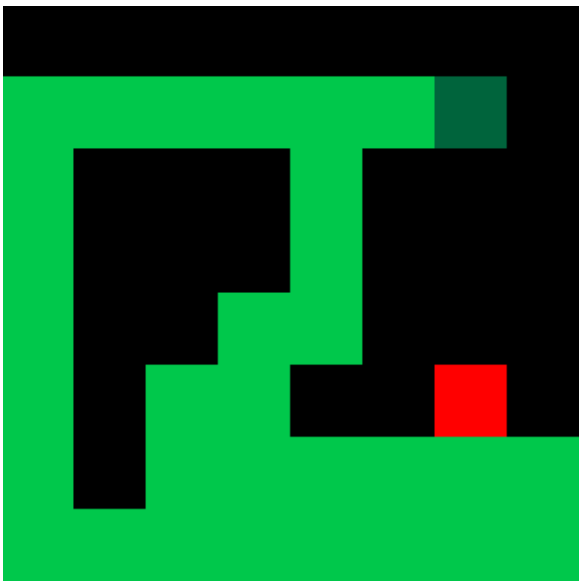

Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

Snake (englisch für Schlange) zählt zu den meist bekannten Computerspielen unserer Zeit. Es zeichnet sich durch sein simples und einfach zu verstehendes Spielprinzip aus. In seiner ursprünglichen Form ist Snake als ein zweidimensionales rechteckiges Feld. Dieses Beschreibe das komplette Spielfeld, in welchem man sich als Snake

bewegt. Häufig wird diese als einfacher grüner Punkt (Quadrat) dargestellt. Dieser stellt den Kopf der Snake dar. Neben dem Kopf der Snake befindet sich auf dem Spielfeld auch noch der sogenannte Apfel. Dieser wird häufig als roter Punkt (Quadrat) dargestellt. Ziel der Snake ist es nun Äpfel zu fressen. Dies geschieht, wenn der Kopf der Snake auf das Feld des Apfels läuft. Danach verschwindet der Apfel und ein neuer erscheint in einem noch freies Feld). Außerdem wächst, durch das Essen des Apfels, die Snake um ein Schwanzglied. Diese Glieder folgen dabei in ihren Bewegungen den vorangegangenen Schwanzglied bis hin zum Kopf. Dem Spieler ist es nur möglich den Kopf der Snake zu steuern. Der Snake ist es nicht erlaubt die Wände oder sich selbst zu berühren, geschieht dies, im Laufe des Spiels, trotzdem endet dieses sofort. Diese Einschränkung führt zu einem Ansteigen der Komplexität gegen Ende des Spiels. Ein Spiel gilt als gewonnen, wenn es der Snake gelungen ist, das komplette Spielfeld auszufüllen.

2.2. Reinforcement Learning

Das Reinforcement Learning (Bestärkendes Lernen) ist einer der drei großen Teilbereiche, die das Machine Learning zu bieten hat. Neben dem Reinforcement Learning zählen das supervised Learning (Überwachtes Lernen) und das unsupervised Learning (unüberwachtes Lernen) ebenfalls noch zum Machine Learning.
Einordnen lässt sich das Reinforcement Learning (RL) irgendwo zwischen vollständig überwachtem Lernen und dem vollständigen Fehlen vordefinierter Labels (Lapan, 2020, S. 26). Viele Aspekte des (SL), wie z.B. neuronale Netze zum Approximation einer Lösungsfunktion, Gradientenabstiegsverfahren und Backpropagation zur Erlernung von Datenrepräsentationen, werden auch im RL verwendet.
Auf Menschen wirkt das RL, im Vergleich zu den anderen Disziplinen des Machine Learnings, am nachvollziehbarsten. Dies liegt an der Lernstrategie die sich dieses Verfahren zu Nutze macht. Beim RL wird anhand eines “trial-and-error„ Verfahrens gelernt. Ein gutes Beispiel für eine solche Art des Lernens ist die Erziehung eines Kindes. Wenn eben dieses Kind etwas gutes tut, dann wird es belohnt. Angetrieben von der Belohnung, versucht das Kind dieses Verhalten fortzusetzen. Entsprechend wird das Kind bestraft, wenn es etwas schlechtes tut. Schlechtes Verhalten kommt weniger häufig zum Vorschein, um Bestrafungen zu vermeiden. (Sutton und Barto, 2018, S.1 ff.)
Beim RL funktioniert es genau so. Das ist auch der Grund dafür, dass viele der Aufgaben des RL dem menschlichen Arbeitsspektrum sehr nahe sind. So wird das RL beispielsweise im Finanzhandel eingesetzt. Auch im Bereich der Robotik ist das RL auf dem Vormarsch. Wo früher noch komplexe Bewegungsabfolgen eines Roboterarms mühevoll programmiert werden mussten, ist es heute bereits möglich Roboter

durch RL Agenten steuern zu lassen, welche selbstständig die Bewegungsabfolgen meisten. (Lapan, 2020, Kapitel 18)

2.2.1. Vokabular

Um ein tiefer gehendes Verständnis für das RL zu erhalten, ist es erforderlich die gängigen Begrifflichkeiten zu erlernen und deren Bedeutung zu verstehen.

Agent

Im Zusammenhang mit dem RL ist häufig die Rede von Agenten. Sie sind die zentralen Instanzen, welche die eigentlichen Algorithmen, wie z.B. den Algorithmus des Q-Learning oder eines Proximal Policy Optimization, in eine festes Objekt einbinden. Dabei werden zentrale Methoden, Hyperparameter der Algorithmen, Erfahrungen von Trainingsläufen, wie auch das NN in die Agenten eingebunden. (Lapan, 2020, S. 31)
Bei den Agenten handelt es sich gewöhnlich um die einzigen Instanzen, welche mit dem Environment (der Umgebung) interagieren. Zu dieser Interaktionen zählen das Entgegennehmen von Observations und Rewards, wie auch das Tätigen von Actions. (Sutton und Barto, 2018, S. 2ff.)

Environment

Das Environment (Env.) bzw. die Umgebung ist vollständig außerhalb des Agenten angesiedelt. Es spannt das zu manipulierende Umfeld auf, in welchem der Agent Interaktionen tätigen kann. An ein Environment werden unter anderem verschiedene Ansprüche gestellt, damit ein RL Agent mit ihm in Interaktion treten kann. Zu diesen Ansprüchen gehört unter anderem die Fähigkeit Observations und Rewards zu liefen, Actions zu verarbeiten. (Lapan, 2020; Sutton und Barto, 2018, S. 31 & S.2 ff.)
Actions, welche im momentanen State des Env. nicht gestattet sind, müssen entsprechend behandelt werden. Dies wirft die Frage auf, ob es in dem Env. einen terminalen Zustand (häufig done oder terminal genannt) geben soll. Existiert ein solcher terminaler Zustand, so muss eine Routine für den Reset (Neustart) des Env. implementiert sein.

Action

Die Actions bzw. die Aktionen sind einer der drei Datenübermittlungswege. Bei ihnen handelt es sich um Handlungen welche im Env. ausgeführt werden. Actions können z.B. erlaubte Züge in Spielen, das Abbiegen im autonomen Fahren oder das

Ausfüllen eines Antragen sein. Es wird ersichtlich, dass die Actions, welche ein RL Agent ausführen kann, prinzipiell nicht in der Komplexität beschränkt sind. Dennoch ist es hier gängige Praxis geworden, dass arbeitsteilig vorgegangen werden soll, da der Agent ansonsten zu viele Ressourcen in Anspruch nehmen müssten.
Im Environment wird zwischen diskreten und stetigen Aktionsraum unterschieden. Der diskrete Aktionsraum umfassen eine endliche Menge an sich gegenseitig ausschließenden Actions. Beispielhaft dafür wäre das Gehen an einer T-Kreuzung. Der Agent kann entweder Links, Rechts oder zurück gehen. Es ist ihm aber nicht möglich ein bisschen Rechts und viel Links zu gehen. Anders verhält es sich beim stetigen Aktionsraum. Dieser zeichnet sich durch stetige Werte aus. Hier ist das Steuern eines Autos beispielhaft. Um wie viel Grad muss der Agent das Steuer drehen, damit das Fahrzeug auf der Spur bleibt? (Lapan, 2020, S. 31 f.)

Observation

Die Observation (Obs.) bzw. die Beobachtung ist ein weiterer der drei Datenübermittlungswege, welche den Agenten mit dem Environment verbindet. Mathematisch, handelt es sich bei der Obs. um einen oder mehrere Vektoren bzw. Matrizen.
Die Obs beschreibt dabei den momentanen Zustand es Envs. Die Obs kann daher als eine numerische Repräsentation anzusehen. (Sutton und Barto, 2018, S. 381)
Die Obs. hat einen immensen Einfluss auf den Erfolg des Agenten und sollte daher klug gewählt werden. Je nach Anwendungsbereich fällt die Obs. sehr unterschiedlich aus. In der Finanzwelt könnte diese z.B. die neusten Börsenkurse einer oder mehrerer Aktien beinhalten oder in der Welt der Spiele könnten diese die aktuelle erreichte Punktezahl wiedergeben. (Lapan, 2020, S. 32) Es hat sich als Faustregel herausgestellt, dass man sich bei dem Designing der Obs. auf das wesentliche konzentrieren sollte. Unnötige Informationen können den die Effizienz des Lernen mindern und den Ressourcenverbrauch zudem steigen lassen.

Reward

Der Reward bzw. die Belohnung ist der letzte Datenübertragungsweg. Er ist neben der Action und der Obs. eines der wichtigsten Elemente des RL und von besonderer Bedeutung für den Lernerfolg. Bei dem Reward handelt es sich um eine einfache skalare Zahl, welche vom Env. übermittelt wird. Sie gibt an, wie gut oder schlecht eine ausgeführte Action im Env. war. (Sutton und Barto, 2018, S. 42)
Um eine solche Einschätzung zu tätigen, ist es nötig eine Bewertungsfunktion zu implementieren, welche den Reward bestimmt.
Bei der Modellierung des Rewards kommt es vor alledem darauf an, in welchen Zeitabständen dieser an den Agenten gesendet wird (sekündlich, minütlich, nur ein

Mal). Aus Bequemlichkeitsgründen ist es jedoch gängige Praxis geworden, dass der Reward in fest definierten Zeitabständen erhoben und übermittelt wird. (Lapan, 2020, S. 29 f.)
Je nach Implementierung hat dies große Auswirkungen auf das zu erwartende Lernverhalten.

State

Der State bzw. der Zustand ist eine Widerspieglung der zum Zeitpunkt t vorherrschenden Situation im Environments. Der State wird von der Obs. (Observation) repräsentiert. Häufig findet der Begriff des States in diversen Implementierungen, wie auch in vielen Ausarbeitung zum Themengebiet des RL Anwendung. (Sutton und Barto, 2018, s. 381 ff.)

Policy

Informell lässt sich die Policy als eine Menge an Regeln beschreiben, welche das Verhalten eines Agenten steuern. Formal ist die Policy π als eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen a im State s des Env. definiert. (Lapan, 2020, S. 44)
Sollte daher ein Agent der Policy π_t zum Zeitpunkt t folgen, so ist $\pi_t(a_t|s_t)$ die Wahrscheinlichkeit, dass die Aktion a_t im State s_t unter den stochastischen Aktionswahlwahrscheinlichkeiten (Policy) π_t zum Zeitpunkt t gewählt wird. (Sutton und Barto, 2018, S. 45 ff.)

Value

Values geben eine Einschätzung ab, wie gut oder schlecht eine State oder State-Action-Pair ist. Sie werden gewöhnlich mit einer Funktion ermittelt. So bestimmt die Value-Function $V(s)$ beispielsweise den Wert des States s. Dieser ist ein Maß dafür, wie gut es für den Agenten ist, in diesen State zu wechseln. Ein andere Wert Q, welcher durch die Maximierung der Q-Value-Function $Q(s, a)$ bestimmt wird, gibt Aufschluss darüber, welche Action a im State s den größten Return (diskontierte gesamt Belohnung) über der gesamten Spielepisode erzielen wird. Diese Values werden ebenfalls unter einer Policy (Regelwerk des Agenten) bestimmt, daher folgt: für die Value-Functions $V(s) = V_\pi(s)$ und $Q(s, a) = Q_\pi(s, a)$. (Sutton und Barto, 2018, S. 46)
Bei einem Verfahren wie z.B. dem Q-Learning lässt sich die Policy formal angeben: $\pi(s) = \arg\max_a Q_\pi(s, a)$. Dies ist die Auswahlregel der Actions a im State s. (Lapan, 2020, S.291)

2.2.2. Funktionsweise

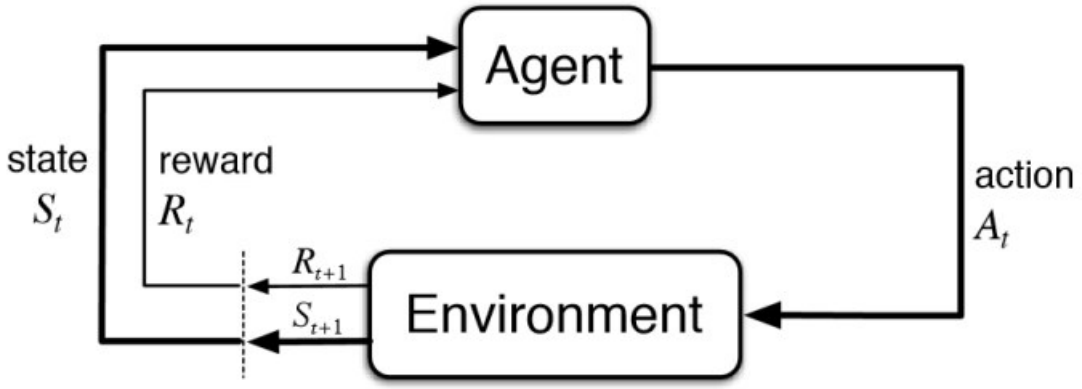

Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State S_t und falls $t \neq 0$ einen Reward R_t. Daraufhin wird vom Agenten eine Action A_t ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State S_{t+1} und Reward R_{t+1} an den Agenten. Diese Prozedur wird wiederholt. Bildquelle: (Sutton und Barto, 2018, S. 38)

Zu Beginn wird dem Agenten vom Environment der initialer State übermittelt. Auf Grundlage dieses Stat S_t wobei $t = 0$ ist, welcher inhaltlich aus der zuvor besprochenen Obs. 2.2.1 besteht, wird im Agenten ein Entscheidungsfindungsprozess angestoßen. Es wird eine Action A_t ermittelt, welcher der Agent an das Environment weiterleitet. Die vom Agenten ausgewählte Action A_t wird nun im Env. ausgeführt. Dabei kann der Agent selbstständig das Env. manipuliert oder er kann die Action an das Env. weiterleitet. Das manipulierte Environment befindet sich nun im neuen State S_{t+1}, welcher an den Agenten weitergeleitet wird. Des Weiteren wird noch einen Reward R_{t+1}, welcher vom Env. bestimmt wurde, an den Agenten übermittelt.
Mit dem neuen State S_{t+1}, kann der Agent wieder eine Action A_{t+1} bestimmen, die ausgeführt wird. Daraufhin werden wieder der neue State S_{t+2} und Reward R_{t+2} ermittelt und übertragen usw. Der Zyklus beginnt von neuem (Sutton und Barto, 2018, S. 37 ff.).

2.2.3. Arten von RL-Verfahren

Nach dem nun das Basisvokabular weitestgehend erklärt wurde, soll nun noch ein tieferer Blick in die verschiedenen Arten der RL geworfen werden.
Alle RL Verfahren lassen sich, unter gewissen Gesichtspunkten, in Klassen einordnen, welche Aufschluss über die Implementierung, den Entscheidungsfindungsprozess und die Datennutzung geben. Natürlich existieren noch viele weitere Möglichkeiten

RL-Verfahren zu klassifizieren aber vorerst soll sich auf diese die folgenden drei beschränkt werden.

Model-free und Model-based

Die Unterscheidung in model-free (modellfrei) und in model-based (modellbasiert) gibt Aufschluss darüber, ob der Agent fähig ist, ein Modell des Zustandekommens der Belohnungen (Reward) zu entwickeln.
Model-free RL Verfahren sind nicht in der Lage das Zustandekommen der Belohnung vorherzusagen, vielmehr ordnen sie einfach die Beobachtung einer Aktion oder einem Zustand zu. Es werden keine zukünftigen Beobachtungen und/oder Belohnungen extrapoliert. (Lapan, 2020; Sutton und Barto, 2018, S. 303 ff. / S. 100)
Ganz anderes sieht es da bei den model-based RL-Verfahren aus. Diese versuchen eine oder mehrere zukünftige Beobachtungen und/oder Belohnungen zu ermitteln, um die beste Aktionsabfolge zu bestimmen. Dem Model-based RL-Verfahren liegt also ein Planungsprozess der nächsten Züge zugrunde. (Sutton und Barto, 2018, S. 303 ff.)

Beide Verfahrensklassen haben Vor- und Nachteile, so sind model-based Verfahren häufig in deterministischen Environments mit simplen Aufbau und strengen Regeln anfindbar. Die Bestimmung von Observations und/oder Rewards bei größeren Environments. wäre viel zu komplex und ressourcenbindend. Model-Free Algorithmen haben dagegen den Vorteil, dass das Trainieren leichter ist, aufgrund des wegfallenden Aufwandes, welcher mit der Bestimmung zukünftiger Observations und/oder Rewards einhergeht. Sie performen zudem in großen Environments besser als model-based RL-Verfahren. Des Weiteren sind model-free RL-Verfahren universiell einsetzbar, im Gegensatz zu model-based Verfahren, welche ein Modell des Environment für das Planen benötigen (Lapan, 2020, S. 100 ff.).

Policy-Based und Value-Based Verfahren

Die Einordnung in Policy-based und Value-Based Verfahren gibt Aufschluss über den Entscheidungsfindungsprozess des Verfahrens. Agenten, welche policy-based arbeiten, versuchen unmittelbar die Policy zu berechnen, umzusetzen und sie zu optimieren. Policy-Based RL-Verfahren besitzen dafür meist ein eigenes NN (Policy-Network), welches die Policy π für einen State s bestimmt. Gewöhnlich wird diese als eine Wahrscheinlichkeitsverteilung über alle Actions repräsentiert. Jede Action erhält damit einen Wert zwischen Null und Eins, welcher Aufschluss über die Qualität der Action im momentanen Zustand des Env. liefert. (Lapan, 2020, S. 100)
Basierend auf dieser Wahrscheinlichkeitsverteilung π wird die nächste Action a be-

stimmt. Dabei ist es offensichtlich, dass nicht immer die optimalste Action gewählt wird.
Anders als bei den policy-based wird bei den value-based Verfahren nicht mit Wahrscheinlichkeiten gearbeitet. Die Policy und damit die Entscheidungsfindung, wird indirekt mittels des Bestimmens aller Values über alle Actions ermittelt. Es wird daher immer die Action a gewählt, welche zum dem State s führt, der über den größten Value verfügt Q, basierend auf einer Q-Value-Function $Q_\pi(s, a)$. (Lapan, 2020, S. 100)

On-Policy und Off-Policy Verfahren

Eine Klassifikation in On-Policy und Off-Policy Verfahren hingegen, gibt Aufschluss über den Zustand der Daten, von welchen der Agent lernen soll. Einfach formuliert sind off-polic RL-Verfahren in der Lage von Daten zu lernen, welcher nicht unter der momentanen Policy generiert wurden. Diese können vom Menschen oder von älteren Agenten erzeugt worden sein. Es spielt keine Rolle mit welcher Entscheidungsfindungsqualität die Daten erhoben worden sind. Sie können zu Beginn, in der Mitte oder zum Ende des Lernprozesses ermittelt worden sein. Die Aktualität der Daten spielt daher keine Rolle für Off-Policy-Verfahren. (Lapan, 2020, S. 210 f.)
On-Policy-Verfahren sind dagegen sehr wohl abhängig von aktuellen Daten, da sie versuchen die Policy indirekt oder direkt zu optimieren.

Auch hier besitzen beide Klassen ihre Vor- und Nachteile. So können beispielsweise Off-Policy Verfahren mit älteren Daten immer noch trainiert werden. Dies macht Off-Policy RL Verfahren Daten effizienter als On-Policy Verfahren. Meist ist es jedoch so, dass diese Art von Verfahren langsamer konvergieren.
On-Policy Verfahren konvergieren dagegen meist schneller. Sie benötigen aber dafür auch mehr Daten aus dem Environment, dessen Beschaffung aufwendig und teuer sein könnte. Die Dateneffizienz nimmt ab.(Lapan, 2020, S. 210 f.)

2.3. Proximal Policy Optimization

Der Proximal Policy Optimization Algorithmus oder auch PPO abgekürzt wurde von den Open-AI-Team entwickelt. Im Jahr 2017 erschien das gleichnamige Paper, welches von John Schulman et al. veröffentlicht wurde. In diesem werden die besonderen Funktionsweise genauer erläutert wird Schulman u. a. (2017).
Der PPO bietet sich besonders durch seine bereits erbrachten Erfolge Schulman u. a. (2017) und seine gute Performance an, was ihn zu einem idealen Kandidaten für einem möglichst optimalen Algorithmus, im Sinne der Forschungsfrage, auszeichnet.

2.3.1. Actor-Critic Modell

Der PPO Algorithmus ist ein policy-based RL-Verfahren, welches, im Vergleich mit anderen Verfahren, einige Verbesserungen aufweist. Er ist eine Weiterentwicklung des Actor-Critic-Verfahrens und basiert intern auf zwei NN, dem sogenanntes Actor-Network bzw. Policy-Network und das Critic-Network bzw. Value-Network. (Sutton und Barto, 2018, S. 273 f.)
Beide NN können aus mehreren Schichten bestehen, jedoch sind Actor und Critic streng von einander getrennt und teilen keine gemeinsamen Parameter. Gelegentlich werden den beiden Netzen (Actor bzw. Critic) noch ein weiteres Netz vorgeschoben. In diesem Fall können Actor und Critic gemeinsame Parameter besitzen. Das Actor- bzw. Policy-Network ist für die Bestimmung der Policy zuständig. Anders als bei Value-based RL-Verfahren wird diese direkt bestimmt und kann auch direkt angepasst werden. Die Policy wird als eine Wahrscheinlichkeitsverteilung über alle möglichen Actions vom Actor-NN zurückgegeben. 2.2.1
Das Critic- bzw. Value-Network evaluiert die Actions, welche vom Actor-Network bestimmt worden sind. Genauer gesagt, schätzt das Value-Network die sogenannte "Discounted Sum of Rewards" zu einem Zeitpunkt t, basierend auf dem momentanen State s, welcher dem Value-Network als Input dient. "Discounted Sum of Rewards" wird im späteren Verlauf noch weiter vorgestellt und erklärt.

2.3.2. PPO Training Objective Function

Nun da einige Grundlagen näher beleuchtet worden sind, ist das nächste Ziel die dem PPO zugrunde liegende mathematische Funktion zu verstehen, um im späteren eine eigene Implementierung des PPO durchführen zu können und um einen objektiveren Vergleich der zwei RL-Verfahren durchführen zu können.
Der PPO basiert auf den folgenden mathematischen Formel, welche den Loss eines Updates bestimmt (Schulman u. a., 2017, S. 5):

$$L_t^{\mathrm{PPO}}(\theta) = L_t^{\mathrm{CLIP\ +\ VF\ +\ S}}(\theta) = \hat{\mathbb{E}}_t[L_t^{\mathrm{CLIP}}(\theta) - c_1 L_t^{\mathrm{VF}} + c_2 S[\pi_\theta](s_t)] \tag{2.1}$$

Dabei besteht die Loss-Function aus drei unterschiedlichen Teilen. Zum einen aus dem Actor-Loss bzw. Policy-Loss bzw. Main Objective Function $L_t^{\mathrm{CLIP}}(\theta)$, zum anderen aus dem Critic-Loss bzw. Value-Loss L_t^{VF} und aus dem Entropy Bonus $S[\pi_\theta](s_t)$. Die Main Objective Function sei dabei durch folgenden Term gegeben (Schulman u. a., 2017, S. 3).

$$L_t^{\mathrm{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t(s,a), \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s,a))] \tag{2.2}$$

Formelelemente

Um die dem PPO zugrundeliegende Update Methode besser zu verstehen folge eine Erklärung ihrer einzelnen mathematischen Elemente. Die einzelnen Erklärungen basieren auf den PPO Paper Schulman u. a. (2017).

Tabelle 2.1.: Formelelemente

Symbol	Erklärung
θ	Theta beschreibt die Parameter aus denen sich die Policy des PPO ergibt. Sie sind die Gewichte, welche das Policy-NN definiert.
π_θ	Die Policy bzw. Entscheidungsfindungsregeln sind eine Wahrscheinlichkeitsverteilung über alle möglichen Actions. Eine Action a wird auf Basis der Wahrscheinlichkeitsverteilung gewählt. Siehe 2.2.1. (Sutton und Barto, 2018, Summary of Notation S. xvi)
$L^{\text{CLIP}}(\theta)$	$L^{\text{CLIP}}(\theta)$ bezeichnet den sogenannten Policy Loss, welche in Abhängigkeit zu der Policy π_θ steht. Dabei handelt es sich um einen Zahlenwert, welcher den Fehler über alle Parameter approximiert. Dieser wird für das Lernen des Netzes benötigt.
t	Zeitpunkt
$\hat{\mathbb{E}}[X]$	$\hat{\mathbb{E}}[X]$ ist der Erwartungswert einer zufälligen Variable X, z.B. $\hat{\mathbb{E}}[X] = \sum_x p(x)x$. (Sutton und Barto, 2018, Summary of Notation S. xv)
$r_t(\theta)$	Quotient zwischen alter Policy (nicht als Abhängigkeit angegeben, da sie nicht verändert werden kann) und aktueller Policy zum Zeitpunkt t. Daher auch Probability Ratio genannt.
$\hat{A}_t(s, a)$	erwarteter Vorteil bzw. Nachteil einer Action a, welche im State s ausgeführt wurde. welcher sich in Abhängigkeit von dem State s und der Action a befindet.
clip	Mathematische Funktion zum Beschneidung eines Eingabewertes. Clip setzt eine Ober- und Untergrenze fest. Sollte ein Wert der dieser Funktion übergeben wird sich nicht mehr in diesen Grenzen befinden, so wird der jeweilige Grenzwert zurückgegeben.

ϵ	Epsilon ist ein Hyperparameter, welcher die Ober- und Untergrenze der Clip Funktion festlegt. Gewöhnlich wird für ϵ ein Wert zwischen 0.1 und 0.2 gewählt.
γ	Gamma bzw. Abzinsungsfaktor ist ein Hyperparameter, der die Zeitpräferenz des Agenten kontrolliert. Gewöhnlich liegt Gamma γ zwischen 0.9 bis 0.99. Große Werte sorgen für ein weitsichtiges Lernen des Agenten wohingegen ein kleine Werte zu einem kurzfristigen Lernen führen (Sutton und Barto, 2018, S. 43 bzw. Summary of Notation S. xv).

Return

Der Return R_t stellt dabei die Summe der Rewards in der gesamten Spielepisode von dem Zeitpunkt t an dar. Diese kann ermittelt werden, da alle Rewards, durch das Sammeln von Daten, bereits bekannt sind. Des Weiteren werden die einzelnen Summanden mit einem Discount Factor γ multipliziert, um die Zeitpräferenz des Agenten besser zu steuern. Gamma liegt dabei gewöhnlich zwischen einem Wert von 0.9 bis 0.99. Kleine Werte für Gamma sorgen dafür, dass der Agent langfristig eher dazu tendiert Aktionen zu wählen, welche unmittelbar zu positiven Reward führen. Entsprechend verhält es sich mit großen Werten für Gamma. (Sutton und Barto, 2018, S. 42 ff.)

Baseline Estimate

Der Baseline Estimate $b(s_t)$ oder auch die Value function ist eine Funktion, welche durch ein NN realisiert wird. Es handelt sich dabei um den Critic des Actor-Critic-Verfahrens. Die Value function versucht eine Schätzung des zu erwartenden Discounted Rewards R_t bzw. des Returns, vom aktuellen State s_t, zu bestimmen. Da es sich hierbei um die Ausgabe eines NN handelt, wird in der Ausgaben immer eine Varianz bzw. Rauschen vorhanden sein. (Mnih u. a., 2016, Kapitel 3)

Advantage

Der erste Funktionsbestandteil des $L_t^{\text{CLIP}}(\theta)$ **??** behandelt den Advantage $\hat{A}_t(s, a)$. Dieser wird durch die Subtraktion der Discounted Sum of Rewards bzw. des Return R_t und dem Baseline Estimate $b(s_t)$ bzw. der Value-Function berechnet. Die folgende Formel ist eine zusammengefasste Version der original Formel aus Schulman u. a.

(2017):

$$\hat{A}_t(s, a) = R_t - b(s_t) \tag{2.3}$$

Der Advantage gibt ein Maß an, um wie viel besser oder schlechter eine Action war, basierend auf der Erwartung der Value-Function bzw. des Critics. Es wird also die Frage beantwortet, ob eine gewählte Action a im State s_t zum Zeitpunkt t besser oder schlechter als erwartet war. (Mnih u. a., 2016, Kapitel 3)

Probability Ratio

Die Probability Ratio $r_t(\theta)$ ist der nächste Baustein des $L_t^{\text{CLIP}}(\theta)$ **??** zur Vervollständigung der PPO Main Objective Function. In normalen Policy Gradient Methoden bestehe ein Problem zwischen der effizienten Datennutzung und dem Updaten der Policy. Dieses Problem tritt z.B. im Zusammenhang mit dem Advantage Actor Critic (A2C) Algorithmus auf und reglementiert das effiziente Sammeln von Daten. So ist des dem A2C nur möglich von Daten zum lernen, welche on-policy (unter der momentanen Policy) erzeugt wurden. Das Verwenden von Daten, welche unter einer älteren aber dennoch ähnlichen Policy gesammelt wurden, ist daher nicht zu empfehlen. Der PPO bedient sich jedoch eines Tricks der Statistik, dem Importance-Sampling (IS, deutsch: Stichprobenentnahme nach Wichtigkeit). Wurde noch beim A2C mit folgender Formel der Loss bestimmt (Lapan, 2020, S. 591):

$$\hat{\mathbb{E}}_t[\log_{\pi_\theta}(a_t|s_t)A_t] \tag{2.4}$$

Bei genauer Betrachtung wird offensichtlich, dass die Daten für die Bestimmung des Loss nur unter der aktuellen Policy π_θ generiert wurden, daher on-policy erzeugt wurden. Schulman et al. ist es jedoch gelungen diesen Ausdruck durch einen mathematisch äquivalenten zu ersetzen. Dieser basiert auf zwei Policies π_θ und $\pi_{\theta_{\text{old}}}$.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{2.5}$$

Die Daten können dabei nun mittels $\pi_{\theta_{\text{old}}}$, partiell off-policy, bestimmt werden und nicht wie beim A2C direkt on-policy. (Schulman, 2017, Zeitpunkt: 9:25)
Nun können generierte Daten mehrfach für Updates der Policy genutzt werden, was die Menge an Daten, welche nötig ist, um ein gewisses Ergebnis zu erreichen, minimiert. Der PPO Algorithmus ist durch die Umstellung auf die Probability Ratio effizienter in der Nutzung von Daten geworden.

Surrogate Objectives

Der Name Surrogate (Ersatz) Objective Function ergibt sich aus der Tatsache, dass die Policy Gradient Objective Function des PPO nicht mit der logarithmierten Policy $\hat{\mathbb{E}}_t[\log_{\pi_\theta}(a_t|s_t)A_t]$ arbeitet, wie es die normale Policy Gradient Methode vorsieht, sondern mit dem Surrogate der Probability Ratio $r_t(\theta)$ 2.5 zwischen alter und neuer Policy.
Intern beruht der PPO auf zwei sehr ähnlichen Objective Functions, wobei die erste $surr_1$ dieser beiden

$$r_t(\theta)\hat{A}_t(s,a) \tag{2.6}$$

der normalem TRPO Objective-Function entspricht, ohne die, durch den TRPO vorgesehene, KL-Penalty. (Schulman u. a., 2017, S. 3 f.) Die alleinige Nutzung dieser Objective Function hätte jedoch destruktiv große Policy Updates zufolge. Aus diesem Grund haben John Schulman et al. eine zweite Surrogate Objective Function $surr_2$, dem PPO Verfahren hinzugefügt.

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s,a) \tag{2.7}$$

Die einzige Veränderung im Vergleich zur ersten Objective Function 2.6 ist, dass eine Abgrenzung durch die Clip-Funktion eintritt. Sollte sich die Probability Ratio zu weit von 1 entfernen, so wird $r_t(\theta)$ entsprechende auf $1-\epsilon$ bzw. $1+\epsilon$ begrenzt. Das hat zufolge, dass der Loss $L_t^{\text{CLIP}}(\theta)$ ebenfalls begrenzt wird, sodass es zu keinen großen Policy Updates kommen kann. Es wird sich daher in einer Trust Region bewegt, da man sich nie allzu weit von der ursprünglichen Policy entfernt Schulman u. a. (2015, 2017).

Zusammenfassung der PPO Training Objective Function

Insgesamt ist der Actor-Loss **??** ein Erwartungswert, welcher sich empirisch über eine Menge an gesammelten Daten ergibt. Dies wird durch $\hat{\mathbb{E}}_t$ impliziert. Dieser setzt sich aus vielen einzelnen Losses zusammen. Diese sind das Minimum der beiden Surrogate Objective Functions zusammen. Dies sorgt dafür, dass keine zu großen Losses die Policy zu weit von dem Entstehungspunkt der Daten wegführen (Trust Region). (Schulman u. a., 2017, S. 3f.)

Des Weiteren wird für den PPO Training Loss auch noch der Value-Loss benötigt. Dieser setzt sich folgendermaßen zusammen:

$$L_t^{\text{VF}} = (V_\theta(s_t) - V_t^{targ})^2 \text{ wobei } V_t^{targ} = r_t(\theta) \tag{2.8}$$

Der letzte Teil, welcher für die Bestimmung des PPO Training Loss benötigt wird, ist der Entropy Bonus. Dabei handelt es sich um die Entropie der Policy.
Es ergibt sich damit der bereits oben erwähnte PPO Training Loss **??**:

$$L_t^{\text{PPO}}(\theta) = L_t^{\text{CLIP + VF + S}}(\theta) = \hat{\mathbb{E}}_t[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}} + c_2 S[\pi_\theta](s_t)]$$

2.3.3. PPO - Algorithmus

Um die Theorie näher an die eigentliche Implementierung zu bewegen soll nun eine Ablauffolge diesen Abschnitt vervollständigen. Diese basiert dabei auf der Quelle Schulman u. a. (2017) und einigen weiteren Anpassungen.

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für Actor θ und Critic w zufallsbasiert. Erstelle ein Experience Buffer EB.
2. Bestimmt mit dem State s und dem Actor-NN eine Action a. Dies geschieht durch $\pi_\theta(s)$.
3. Führe Action a aus und ermittle den Reward r und den Folgezustand s'.
4. Speichern von State s, Action a, Policy $\pi_{\theta_{\text{old}}}(s, a)$, Reward r, Folge-State s'. $(s, a, \pi_{\theta_{\text{old}}}(s, a), r, s') \longrightarrow EB$
5. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis N Zeitintervalle bzw. für N Spielepisoden erreicht sind.
6. Entnehme ein Mini-Batch aus dem Buffer $(S, A, \{\pi_{\theta_{\text{old}}}\}, R, S') \longleftarrow EB$.
7. Bestimmt die Ratio $r_t(\theta)$. 2.3.2.
8. Berechne die Advantages $\hat{A}_t(s, a)$. 2.3.2.
9. Berechne die Surrogate Losses $surr_1$ und $surr_2$. 2.3.2.
10. Bestimmt den PPO Loss. **??**
11. Update die Gewichte des Actors und Critics. $\theta_{\text{old}} \longleftarrow \theta$ und $w_{\text{old}} \longleftarrow w$
12. Wiederhole alle Schritte ab Schritt 7 K-mal erneut durch.
13. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis der Verfahren konvergiert.

2.4. Deep Q-Network

Der DQN (Deep Q-Network-Algorithmus) ist ein weiterer Reinforcement Learning Algorithmus, welcher auf einer ihm zugrundeliegenden Formel basiert. Er hat bereits große Erfolge besonders in der Gaming Branche erzielen können. Daher erscheint es auch nicht weiter verwunderlich, dass sich dieser Algorithmus großer Beliebtheit erfreut. Ebenfalls wurden bereits viele Erweiterungen, wie der DDQN (Double Deep Q-Network) oder der DQN Implementierungen mit Verrauschen Netzen usw.
Aufgrund seiner großen Beliebtheit ergibt sich die Frage, ob der DQN dieser auch gerecht wird und dieser wirklich optimale Agenten zur Lösung des Spiels Snake hervorbringen kann? Eine weitere naheliegende Frage ist, ob dieser Algorithmus im Vergleich zu anderen Arten konkurrenzfähig ist? Diese beiden Fragen lassen sich mit einem Vergleich beantworten, was diesen Algorithmus zu einem weiteren guten Kandidaten macht.
Angestammtes Ziel aller Q-Learning-Algorithmen ist es, jedem State-Action-Pair (s, a) (Zustands-Aktions-Paar) einem Aktionswert (Q-Value) Q zuzuweisen (Lapan, 2020, S. 126). Dies ist beispielsweise über eine Tabelle möglich, was jedoch bei großen State-Action-Spaces (Zustands-Aktions-Räumen) schnell ineffizient wird. Ein weiterer Ansatz sind NN, welches durch seine zumeist zufällige Initialisierung der Gewichte bereits für jeden (State-Action-Pair) ein Q-Value liefert. Es sei erwähnt, dass es bei NN häufig so ist, dass nur der State als Input für das Value-NN dient.

Ein kleiner Blick in die dem Algorithmus zugrunde liegende Logik, eröffnet des weiteren einen besseren Überblick. So ist die Idee von vielen RL-Verfahren, die Aktionswert Funktion mit Hilfe der Bellman-Gleichung iterativ zu bestimmen. Daraus ergibt sich:

$$Q_{i+1}(s, a) = \mathbb{E}\big[r + \gamma \max_{a'} Q(s', a') | s, a\big] \tag{2.9}$$

In der Theorie konvergiert ein solches Iterationsverfahren $Q_i \longrightarrow Q^*$ jedoch nur, wenn i gegen unendlich läuft $i \longrightarrow \infty$, wobei Q^* die optimale Aktionswert-Funktion darstellt. Da dies jedoch nicht möglich ist, muss Q^* angenäht werden $Q(s, a; \theta) \approx Q^*$. Dies geschieht mittels eines NN.
Damit dieses nicht ausschließlich Zufallsgetrieben Q-Values ermittelt, ist eine Anpassung der Gewicht des NN nötig. Dafür muss jedoch zuerst der Loss des DQN bestimmt werden. Dies geschieht mit Hilfe mehrerer Loss-Function. Die i-te Loss-Function mit den i-ten NN-Parametern ist wie folgt definiert Mnih u. a. (2015):

$$L_i(\theta_i) = \mathbb{E}\Big[\big(r(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\big)^2\Big] \tag{2.10}$$

Die Formel 2.10 besagt, dass der Loss eines zufällig ausgesuchten State-Action Tupels (s,a) sich wie folgt zusammensetzt. Der Fehler ist die Differenz aus dem Aktionswerts $Q(s, a; \theta_i)$, welcher Aufschluss über den, in dieser Episode, zu erwartenden Reward liefert und y_i. Dabei ist y_i nichts anderen als der Reward, welche durch die Ausführung der Action a im State s im Env. erzielt wurde, addiert mit dem Q-Value der Folgeaktion a' und dem Folgezustand s'.
Da $Q(s, a)$ rekursiv definiert werden kann, ergibt sich in vereinfachter Form (Lapan, 2020, S.126):

$$Q(s, a) = r + \gamma \max_{a \in A} Q(s', a') = \mathbb{E}_{s' \sim \mathcal{S}}\big[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a\big] = y_i \qquad (2.11)$$

Es wird erkenntlich, dass $Q(s, a; \theta)$ dem Q-target $Q(s, a; \theta) \longrightarrow y_i$ entsprechen soll, darum wird die Differenz zwischen beiden bestimmt und als Loss deklariert. Dieser wird noch quadriert, damit der Loss positiv ist und damit er für den MSE (Mean Squared Error) anwendbar ist.

2.4.1. DQN - Algorithmus

Zur besseren Anwendbarkeit haben Volodymyr Mnhi et al. einen Algorithmus entworfen, welcher den DQN anschaulich erklärt. Da jedoch in diesem Algorithmus weiterhin auf Teile der Loss-Function eingegangen wird, folge eine bereinigte Version, welche sich auch für den allgemeinen Gebrauch besser anbietet (Lapan, 2020, S. 149 f.).

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für das Q-NN zufallsbasiert. Setzte Epsilon $\epsilon = 1.00$ und Erzeuge leeren Replay-Buffer RB.

2. Wähle mit der Wahrscheinlichkeit ϵ eine zufällige Action a oder nutze $a = \arg\max_a Q(s, a)$

3. Führt Action a aus und ermittle den Reward r und den Folgezustand s'.

4. Speichern von State s, Action a, Reward r und Folgezustand s'. $(s, a, r, s') \longrightarrow$ Replay-Buffer

5. Senkt ϵ, sodass die Wahrscheinlichkeit eine zufällige Action zu wählen minimiert wird. Gewöhnlich existiert eine untere Grenze für ϵ, sodass immer noch einige wenige Actions zufällig gewählt werden, je nach Grenze.

6. Entnehme auf zufallsbasiert Mini-Batches aus dem Replay Buffer.

7. Berechne für alle sich im Mini-Batch befindliche Übergänge den Zielwert $y = r$ wenn die Episode in diesem Übergang endet. Ansonsten soll $y = r + \gamma \max_{a \in A} \hat{Q}(s', a')$.

8. Berechne den Verlust $\mathbb{L} = (Q(s,a) - y)^2$
9. Update $Q_\theta(s,a)$ durch SGD (Stochastischen Gradientenabstiegsverfahren, Englisch: Stochastic Gradient Descent). Daher $Q_{\theta_i}(s,a) \longrightarrow Q_{\theta_{i+1}}(s,a)$
10. Kopierte alle N Schritte die Gewichte von $Q(s,a)$ nach $\hat{Q}(s,a)$ $\theta_{Q(s,a)} \longrightarrow \hat{Q}(s,a)$
11. Wiederhole alle Schritte von Schritt 2 an bis sich eine Konvergenz einstellt.

Kapitel 3

Anforderungen

In Kapitel 2, wurden Grundlagen für die weiteren Vergleiche der Reinforcement Learning Agenten gelegt, welche auf zwei unterschiedlichen Algorithmen (DQN und PPO) basieren.
Um diese Vergleiche zu realisieren, soll ein System entwickelt werden, dass diese durchführen, festhalten und auswerten kann. Dieses soll aus einem Environment, mehreren Agenten beider Algorithmen, sowie aus statistischen Analysekomponenten zur Leistungsbestimmung, bestehen. Zuzüglich sollen weitere Anforderungen an die Evaluation gestellt werden, um die Vergleichbarkeit sicherstellen.

3.1. Anforderungen an das Environment

In diesem Abschnitt werden die Anforderungen an das Env dargestellt. Neben der Hauptanforderung, dass das Spiel Snake implementieren werden soll, ergeben sich weitere zusätzliche Anforderungen.

3.1.1. Standardisierte Schnittstelle

Das Env soll eine standardisierte Schnittstelle besitzen, sodass drei Kommunikationskanäle implementiert werden (siehe 2.2.1). Es soll in der Lage sein, Aktionen zu empfangen. Des Weiteren soll es eine Observation und einen Rewards an den Agenten übergeben können. Diese Standardisierung erleichtert die Verwendbarkeit, auch bei anderen Algorithmen.

3.1.2. Funktionalitäten

Das Env soll die folgenden Funktionalitäten implementieren.

Aktionsausführung

Das Env muss eine Funktionalität beinhalten, die eine Aktion ausführen kann. Diese Aktionsausführung muss sich nach den Regeln des Spiels Snake richten (siehe 2.1).

Reset

Das Env muss eine Reset Methode implementieren, um einen erbrachten Spielfortschritt zurückzusetzen. Dies ist für ein stetiges Lernen unentbehrlich.

Render

Das Env muss eine Render Funktionalität implementieren, um eine Visualisierung des Spiels Snake zu ermöglichen. Diese dient der besseren Evaluation und Demonstration.

3.2. Anforderungen an die Agenten

In diesem Abschnitt werden die Anforderungen an die Agenten, welche auf dem PPO bzw. DQN Algorithmus basieren, dargestellt.

3.2.1. Funktionalitäten

Die Agenten müssen folgende Funktionalitäten implementieren.

Aktionsbestimmung

Die Agenten müssen in der Lage sein, aus einer Observation eine Aktion zu bestimmen, welche wiederum dem Env übergeben werden muss, um einen Spielfortschritt erzielen zu können.

Lernen

Die Agenten müssen fähig sein, auf Grundlage vergangener Spielepisoden zu lernen und damit ihre Spielergebnisse zu verbessern.

3.2.2. Parametrisierung

Das System muss die Möglichkeit besitzen, mehrere Agenten des gleichen Algorithmus zu erstellen, welche sich jedoch durch die verwendeten Hyperparameter unterscheiden. Diese Definition von Agenten ist in der Evaluation zu berücksichtigen und dient damit einer besseren Vergleichbarkeit.

3.2.3. Diversität der RL Algorithmen

Um nicht nur Agenten eines Algorithmus untereinander zu vergleichen, sondern auch den Vergleich zu anderen Algorithmen zu erbringen, sollen ein DQN- und PPO-Algorithmus miteinander verglichen werden. Diese bieten sich, wie in den Abschnitten 2.3 und 2.4 beschrieben, für den Vergleich an.

3.3. Anforderungen an die Datenerhebung

In diesem Teil sollen Anforderungen an die statistische Datenerhebung und an die damit verbundenen Analysekomponenten gestellt werden.

3.3.1. Mehrfache Datenerhebung

Die Datenermittlung muss für jeden einzelnen Agenten mehrfach durchgeführt werden, um die Validität der Messung zu gewährleisten.

3.3.2. Datenspeicherung

Damit aus den erzeugten Test- und Trainingsdaten statistische Schlüsse gezogen werden können ist es wichtig, dass diese gespeichert werden. Da jedoch die Menge an Daten schnell riesige Dimensionen annehmen würde, sollen stellvertretend nur die Daten ganzer Spiele gespeichert werden. Diese Strategie stellt einen Kompromiss zwischen Vollständigkeit und effizientem Speicherplatzmanagement dar.
Das System soll folgende Daten speichern:

Tabelle 3.1.: Zu erhebende Daten

Daten	Erklärung
steps	Die in einem Spiel durchgeführten Züge. Diese geben in der Evaluation später Aufschluss über die Effizienz und weisen auf Lernfehler der Agenten, wie beispielsweise das Laufen im Kreis, hin.
apples	Die Anzahl der gefressenen Äpfel in einem Spiel ist ein maßgeblicher Evaluationsfaktor zur Einschätzung des Lernerfolges.
wins	Hat der Agent das Spiel gewonnen. Dieser Wert stellt die Endkontrolle des Agenten dar. Er gibt Aufschluss über das Konvergenzverhalten.

3.4. Anforderungen an die Statistiken

Das System muss in der Lage sein, Statistiken generieren und speichern zu können. Diese sollen für die Evaluation verwendet werden. Zu jedem Evaluationskriterium (siehe 3.2) soll eine Statistik erstellt werden.

3.5. Anforderungen an die Evaluation

Bei der Evaluation soll der optimalste Agent für jedes Evaluationskriterium ermittelt werden. Einige dieser Kriterien stammen aus der verwendeten Literatur Wei u. a. (2018) und Chunxue u. a. (2019) wie z.B. die Performance, anderen ergeben sich aus den zu speichernden Daten. Die einzelnen Kriterien lauten:

Tabelle 3.2.: Evaluationskriterien

Kriterium	Erläuterung
Performance	Welcher Agent erreicht das beste Ergebnis? Im Sachzusammenhang mit dem Spiel Snake bedeutet dies: Welcher Agent frisst die meisten Äpfel, nach dem er trainiert wurde?
Effizienz	Welcher Agent löst das Spiel mit der größten Effizienz? Bezogen auf das Spiel Snake bedeutet dies: Welcher Agent ist in der Lage, die Äpfel mit möglichst wenig Schritten zu fressen? Dieser Wert ist besonders in Real-World-Applikationen von Interesse, da beispielsweise selbstfahrende Autos ihrer Ziele in einer möglichst geringen Strecke erreichen sollen, um Energie und zeit zu sparen.
Robustheit	Welcher Agent kann in einer modifizierten Umgebung das größte Performance erreichen? In Bezug auf Snake bedeutet dies: Welcher Agent ist in der Lage, auf einem größeren bzw. kleineren Spielfeld die meisten Äpfel zu fressen? Auch die Robustheit ist bei Real-World-Applikationen ein wichtiger Faktor, da sich unbemannte Drohne auch in unbekannten Umgebungen zurechtfinden müssen.

Siegrate	Welcher Agent schafft die Spiele mit einem Sieg zu beenden? Die Siegrate gibt Aufschluss darüber, ob die angelernte Aufgabe vollständig gelöst werden kann. Sie ist weiterhin ein wichtiger Faktor für das Beenden des Trainings.

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel soll es thematisch über den momentanen Stand der bereits durchgeführten Forschung gehen. Dabei sollen die Arbeiten gezielt nach den folgenden Aspekten durchsucht werden. Anschließend folgt eine Diskussion. Ausgewählt wurde die Arbeiten aufgrund ihres thematischen Hintergrundes zum Spiel Snake, in Verbindung mit dem RL. Zu den oben erwähnten Aspekten gehören:

- Optimierungsstrategien
- Reward Funktion
- Evaluationskriterien

4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning

In der folgenden Auseinandersetzung wird sich auf die Quelle Wei u. a. (2018) bezogen. In der Arbeit „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ wurden mehrere Optimierungen an einem DQN Agenten durchgeführt, um eine größere Performance im Spiel Snake zu erzielen. Sie wurde von Zhepei Wei et al. verfasst und im Jahr 2018 veröffentlicht.

Thematisch wurden in diesem Paper drei Optimierungsstrategien vorgestellt, welche auf einen Baseline DQN (Referenz DQN) angewendet worden sind. Bei diesen Strategien handelt es sich um den Training Gap, die Timeout Strategy und den Dual Experience Replay.
Der Dual Experience Replay (Splited Memory) besteht aus zwei Sub-Memories, in welchen Erfahrungen belohnungsbasiert einsortiert und gespeichert werden. Mem1 besteht dabei nur aus Erfahrungen, welche einen Reward aufweisen, der größer als ein vordefinierter Grenzwert ist. Die restlichen Erfahrungen werden in Mem2 eingepflegt. Zu Beginn des Lernens werden 80% Erfahrungen aus Mem1 ausgewählt und

20% Erfahrungen aus Mem2 entnommen, um den Lernerfolg zu beschleunigen. Im weiteren Lernverlauf wird dieses Verhältnis normalisiert (Mem1: 50% and Mem2: 50%).
Der Training Gap beschreibt die Erfahrungen, welche der Agent, zum Zweck des performanteren Lernens, nicht verarbeiten soll. Zu diesen zählen die Erfahrungen direkt nach dem Konsum eines Apfels, sodass der Agent die Neuplatzierung dieses erlernen würde. Da der Agent auf diesen Prozess jedoch keinen Einfluss hat, könnte die Verarbeitung dieser Daten den Lernerfolg mindern, weshalb die Training Gap Strategie etwaige Erfahrungen nicht speichert und das Lernen während dieser Periode verhindert.
Die Timeout Strategy sorgt für eine Bestrafung, wenn der Agent über eine vordefinierte Anzahl an Schritten P keinen Apfel mehr gefressen hat. Dabei werden die Rewards der letzten P Erfahrungen mit einem Malus verrechnet, was den Agenten dazu anhält die schnellste Route zum Apfel zu finden. Die Höhe des Malus ist antiproportional zur Länge der Snake (geringe Länge $\rightarrow$ großer Malus; große Länge $\rightarrow$ geringer Malus).
Die optimierte Reward Funktion, welche das Paper verwendet, besteht damit aus dem Distanz Reward, welcher sich aus der Addition des vorherigen Rewards mit Δr ergibt.

$$r_{res} = r_t + \Delta r \tag{4.1}$$

Zusätzlich wird der resultierende Reward zwischen 1 und -1 geclipt $r_{res} = clip(r_{res}, -1, 1)$. Δr ist dabei wie folgt definiert:

$$\Delta r(L_t, D_t, D_{t+1}) = \log_{L_t} \frac{L_t + D_t}{L_t + D_{t+1}} \tag{4.2}$$

Wobei t den vorherigen, $t+1$ den aktuellen Zeitpunkt darstellt. L_t ist die Läge der Snake zum vorherigen Zeitpunkt und D_t und D_{t+1} stellen die Distanzen zwischen Snake und Apfel zum vorherigen und aktuellen Zeitpunkt dar.
Sollte die Timeout Strategy auslösen, so werden die letzten P Erfahrungen entsprechend angepasst und in Mem2 verschoben.
Als maßgebliches Kriterien zur Evaluation der Leistung des DQN wurde die Performance, gemessen im Score und die steps survived, also die überlebten Schritte herangezogen.

4.1.1. Diskussion

Sowohl die Training Gap also auch Dual Experience Replay und Timeout Strategy stellen vielversprechende Optimierungen dar, welche, auf experimentellen Resulta-

ten basierend, gute Ergebnisse erzielen konnten. Jedoch existieren auch Diskussionspunkte an der Ausarbeitung. Das Env wird im Paper nur kurz vorgestellt, jedoch lässt sich aus dem Abschnitt Game Environment schließen, dass alle Anforderungen des Env 3.1 erfüllt sind.

Die Verfasser führen keinen Vergleich mit anderen Algorithmen durch, lediglich ein DQN Agent wird betrachtet. Daher sind auch die Optimierungen, darunter Dual Experience Replay, Training Gap und Timeout Strategy, nicht für den PPO-Algorithmus anwendbar. Zwar sind alle Funktionalitäten des, im Paper verwendeten, DQN gegeben (siehe 3.1.2), jedoch wurde der Fokus kaum auf eine Parametrisierung 3.2.2 gelegt, da dieses Paper hauptsächlich nur eine Agenten betrachtet hat und nicht mehrere Varianten, mit Ausnahme der optimierten Agenten. Diese unterscheiden sich nur durch die Optimierungen.

Auch bei der statistischen Datenerhebung existieren Abweichungen zu den Anforderungen in 3.3. Zhepei Wei et al. verzichteten auf eine mehrfache Datenerhebung und markierten daher ihre Ergebnisse als experimentell. Dem Leser werden nur indirekt Informationen über die erhobenen Daten mitgeteilt. Aus Statistiken lässt sich schließen, dass der Scores und die steps (Anzahl der Schritt) gespeichert wurden.

Weiterhin wurde sich bei den Evaluationskriterien einzig auf die Performance, gemessen am Score, und Spielzeit, gemessen in steps, konzentriert. Weitere Kriterien wie, beispielsweise die Robustheit oder Siegrate 3.2, werden nicht betrachtet.
Dies soll in dieser Ausarbeitung jedoch geschehen, da diese Faktoren ebenfalls wichtig für eine voll umfassende Bewertung der Agenten sind, besonders in Realwelt-Applikationen (siehe 1.1).
Des Weiteren ist erwähnenswert, dass die Verfasser auf das Evaluationskriterium der steps survived setzten, welches im kompletten Widerspruch zur Effizienz steht. Zhepei Wei et al. sehen ein langes Überleben der Snake als positiv an, wohingegen dies in dieser Ausarbeitung kritisch betrachtet wird, da es kein zielgerichtetes Verhalten bestärkt. Ziel der Agenten in dieser Ausarbeitung ist es, das Spiel Snake möglichst effizient zu lösen und es nicht lange zu überleben.

4.2. UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene

In der folgenden Auseinandersetzung wird sich auf die Quelle Chunxue u. a. (2019) bezogen. Die Arbeit „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ wurde von Chunxue Wu et al. verfasst und am 5. August 2019 veröffentlicht. Dabei wird das Spiel Snake als ein dynamisches Pathfinding Problem interpretiert, auf dessen Basis unbemannte Drohnen in Katastrophensituationen zum Einsatz kommen sollen.

Auch in diesem Paper verwenden die Autoren Optimierungen, um den Lernerfolg zu steigern.
Einer dieser Optimierungen wurde auf Basis des Geruchssinnes konzipiert. Der Odor Effect erzeugt um den Apfel drei aneinanderliegende Geruchszonen, in welchen ein größerer Reward zurückgegeben wird als außerhalb der Geruchszonen. Dabei unterscheiden sich diese in der Höhe des zurückgegebenen Rewards, sodass die dritte Zone den geringsten und die erste Zone den größten Reward von allen zurückgibt ($r_1 > r_2 > r_3$, wobei r_x der Reward des x-ten Kreises darstellt). Diese Zonen stellen den zunehmenden Duft von Nahrung dar, wobei dieser immer stärker wird, um so näher man sich der Quelle nährt.
Eine weitere Optimierungsstrategie basiert auf dem Loop Storm Effect, welcher das Verhalten beschreibt, um den Apfel zu laufen. Die Verfasser haben festgestellt, dass dieser Effekt zu einem schlechten Lernerfolg führt. Darum haben Wu et al. einen dynamischen Positionsspeicher konzipiert, welcher Loops erkennt und diese, durch das Zurückgeben einer Zufallsaktion, welche nicht auf dem Loop liegt, unterbricht. Experimentelle Ergebnisse des Papers haben gezeigt, dass der Loop Storm Effect kaum mehr in Erscheinung trat.
Auf Basis einer Standard Reward Funktion, welche für das Essen eines Apfels +100 und für das Sterben -100 zurückgibt, wurden ein Versuch durchgeführt. Dem Agenten war es nicht möglich gegen die optimale Lösung zu konvergieren, aufgrund von Loop Storms und einer unzureichende Reward Funktion. Hingegen war es dem Agenten mit Odor Effect und dynamischen Positionsspeicher möglich, eine Konvergenz nach 8 Millionen Epochs (hier Trainingsdurchläufe eines DQN) zu erreichen.
Die Reward Funktion entspricht daher dem Odor Effect. Auch in diesem Paper bleibt die Performance maßgeblicher Evaluationsfaktor, wobei der Fokus auf den erreichten Reward gelegt wurde, welcher stark mit der Performance korreliert.

4.2.1. Diskussion

Auch dieses Paper besitzt interessante Verbesserungen, welche, nach den ersten Ergebnissen, gute Resultate vorweist. Jedoch legt auch dieses Paper andere Schwerpunkte als diese Ausarbeitung mit seinen Anforderungen (siehe 3).

Aus Graphiken geht hervor, dass die Verfasser ein Env mit einer Visualisierung besitzen. Weiterhin ist davon auszugehen, dass auch alle weiteren Anforderungen bezüglich des Env (siehe 3.5) erfüllt worden sind, da ansonsten kein Training eines Agenten möglich wäre. Dennoch wurde das Env nur auf grundlegende Weise behandelt.
Das Paper legt seinen Schwerpunkt deutlich mehr auf die Grundlagen des RL, wie z.B. auf den Markov Decision Process und die RL Kernbegriffe (siehe 2.2.1).

Auch erfüllt die Ausarbeitung alle Anforderungen an die Funktionalitäten von Agenten (siehe 3.2.1). Dennoch wurde auch hier weniger der Fokus auf eine Parametrisierung der Agenten gelegt, da wieder nur ein DQN Agent näher untersucht wurde. Zwar werden gegen Ende einige Vergleich zu einer Hand voll anderer Agenten getätigt, jedoch sind diese Vergleiche nur sehr oberflächlich, da auf diesen Punkt nicht das Hauptaugenmerk der Arbeit liegt. Im Gegensatz dazu, soll in dieser Ausarbeitung der Vergleich von Agenten im Mittelpunkt liegen.

Wie auch im ersten Paper (siehe 4.1.1), setzen die Verfasser nicht auf eine mehrfache Datenerhebung für die statistische Auswertung, sondern kennzeichnen ihrer Ergebnisse als experimentell. Eine direkte Erwähnung der Daten, welche im Verlauf des Trainings und es Testens gespeichert werden, wird nicht durchgeführt. Auch dies soll im Rahmen dieser Ausarbeitung geschehen um die Parametrisierung (siehe 3.2.2) herauszustellen. Die gezeigten Statistiken weisen jedoch darauf hin, dass der Score sowie der mittlere Aktionswert (Q-Value) gespeichert wurden. Zuzüglich werden, für die Optimierungen, die steps und das Auftreten von Loop Storms erfasst und gespeichert, anders als in dieser Ausarbeitung, wo immer die gleichen Daten erhoben werden, unbeachtet von optimierten oder unoptimierten Agenten.

Chunxue Wu et al. verwendeten zudem hauptsächlichen den Score und die Q-Values als Evaluationskriterium. Um die Effizienz der im Paper verwendeten Strategien (Optimierungen) zu zeigen wurden ebenfalls die gemittelten Steps pro Spiel und das Auftreten von Loop Stoms als weitere untergeordnete Evaluationskriterien verwendet.

4.3. Zusammenfassung

Sowohl „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ 4.1 als auch „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ 4.2 bieten einige Optimierungsstrategien, welche im weiteren Verlauf dieser Ausarbeitung als Vorbild dienen sollen.
Besonders zu betonen ist, dass die vorgestellten Arbeiten alle verschiedene Schwerpunkte gesetzt haben und daher nicht immer die Anforderungen dieser Ausarbeitung erfüllt haben.

Kapitel 5

Konzept

In diesem Kapitel soll das Konzept dieser Ausarbeitung vorgestellt werden. Dieses besteht aus vier Teilen. Zuerst soll das Vorgehen erklärt werden, gefolgt von der Darstellung des Environments und der Agenten. Danach soll im weiteren auf die Datenerhebung eingegangen werden.
Ziel dieses Abschnittes ist es, das Vorgehen und alle weiteren dazu benötigten Elemente unabhängig von der Implementierung darzustellen, sodass die Ergebnisse mit jeder universell reproduzierbar sind.

5.1. Vorgehen

Das Vorgehen lässt sich am besten mit Hilfe eines Flussdiagramms darstellen, in welchem die einzelnen Schritte des Vergleichs visuell dargestellt werden.
Zu Beginn sei erwähnt, dass die Annahme getroffen wird, dass alle für die Vergleiche benötigten Komponenten, wie z.B. Environment, Agenten und statistische Analysekomponenten, entsprechende der Anforderungen implementiert sind (siehe 3).
Als erstes werden die Agenten erstellt (siehe 5.1). Für diese diesen Zweck werden mehrere Klassen mit verschiedenen Hyperparametern generiert bzw. ausgewählt. Mit dieser Baseline-Agenten-Menge werden nun die weiteren Vergleiche durchgeführt. Zu Beginn werden für jedes Evaluationskriterien die zwei optimalsten Baseline Agenten bestimmt. Auf diese sollen die Optimierungen angewendet werden. Aufgrund der Tatsache, dass das Anwenden der Optimierungen viel Zeit und Ressourcen bindet, sollen diese nur auf die vielversprechendsten Agenten angewendet werden. Dies führt zu einer Konvergenz des Verfahrens.
Der verwendete Algorithmus besitzt dabei keinen Einfluss auf die Auswahl der Agenten, sodass auch zwei DQN oder PPO Agenten optimiert werden können. Genauere Details zur Durchführung der Baseline Vergleiche finden sich im Abschnitt 5.5.1.
Basierend auf den beiden Sieger Agenten (Baseline Winner Agent-01 und Baseline Winner Agent-02) der Baseline Vergleiche (siehe 5.1), werden nun die Optimierungen angewendet. Mit diesen optimierten Agenten (Agent-01 Optimierung A bis Agent-02

Optimierung B) werden nun die Vergleiche bezüglich jedes einzelnen Evaluationskriteriums 3.2 erneut wiederholt.
Im letzten Schritt soll in der gesamt Evaluation der optimale Agent für jedes Evaluationskriterium ermittelt werden. Dabei können dies auch Baseline Agenten sein. Es ist nicht zwingend zugesichert das optimierte Agenten die optimalsten dies.

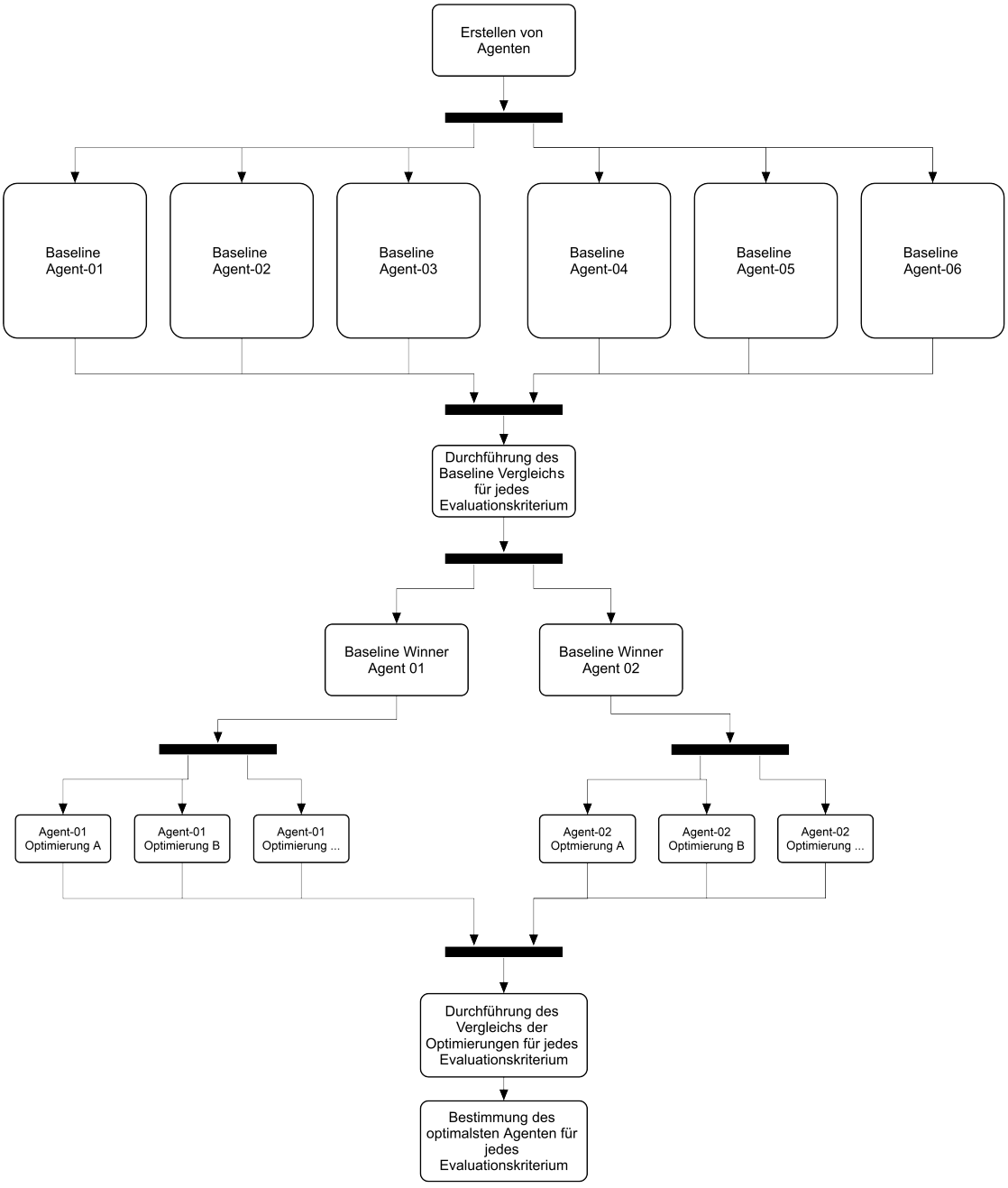

Abbildung 5.1.: Darstellung des Vorgehens.

5.2. Environment

Das Env besteht im wesentlichen aus einer Hauptkomponenten, der Spiellogik, welche von einer Schnittstellen-Komponente umschlossen wird. Diese soll mit einer standardisierten Schnittstelle (siehe 3.1.1) implementiert werden.
Die Spiellogik kapselt die Game-, Player-, Reward-, Observation- und GUI-Komponenten, welche im Folgenden näher erklärt werden.

5.2.1. Spiellogik

Die Spiellogik besteht aus den fünf Unterkomponenten, welche in 5.2 bereits benannt wurden.
Die Game-Komponente stellt die Hauptkomponente dar, da sie die eigentliche Aktionsdurchführung implementiert. Sie beinhaltet jeweils Instanzen der Reward-, Observation-, GUI- und Player-Komponenten. Letztere ist eine Datenhaltungskomponente, welche die Daten der Snake, wie z.B. Position oder Ausrichtung (direction) beinhaltet.

Die Reward-Komponente bestimmt den auszugebenden Reward nach jeder Aktionsabfertigung. Dieser berechnet sich wie in 5.2.1 angegeben. Zuzüglich wird im Rahmen der Optimierungen A, (siehe 5.4.1), eine weitere Reward Funktion implementiert.

In der Game-Komponente werden wichtige Spielbezogene Daten verwaltet. Zu diesen gehören das Spielfeld (ground), sowie die Form des Spielfeldes (shape) und die Position des Apfels auf dem Spielfeld. Sie beinhaltet viele Methoden, wie z.B. die action, observe, evaluate, reset und view Methode.

In der Player-Komponente werden Spielerbezogene Daten verwaltet. Zu diesen zählen die Position des Kopfes der Snake, sowie ihrer Schwanzglieder, ihre Ausrichtung (direction), ihre gelaufenen Schritte seit dem letzten Fressen eines Apfels (inter_apple_steps), ihr Lebensstatus (is_terminal), daher ob sie tot oder lebendig ist und weitere Farbkonstanten für die GUI.

Die Observation-Komponente beinhaltet viele einzelne Funktionen zur schrittweisen Erstellung der Observation, wie sie in Abschnitt 5.2.1 erklärt wird.

Zur Erzeugung der grafischen Oberfläche implementiert die GUI-Komponente die Funktionalität ein Fenster zu öffnen, welches das Spielgeschehen, daher das Spielfeld (ground), anzeigt und stetig an den neusten Stand anpasst.

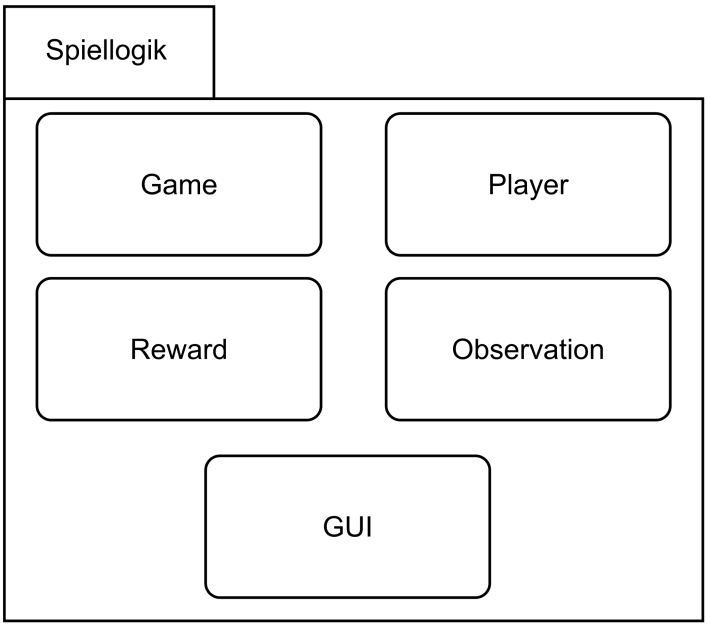

Abbildung 5.2.: Darstellung der Spiellogik mit ihren Unterkomponenten.

Spielablauf

Die eigentliche Aktionsabarbeitung wird durch das Aufrufen der step Funktionalität in der Schnittstellen-Komponente (siehe 5.2.2) bewirkt. Diese ruft die evaluate und observe Methoden auf, welche in der Game-Komponente implementiert sind (siehe 5.2). Um jedoch die Abarbeitung einer Aktion durchzuführen, muss zuerst die action Methode aufgerufen werden, welche die von Agenten bestimmte Aktion übergeben bekommt.

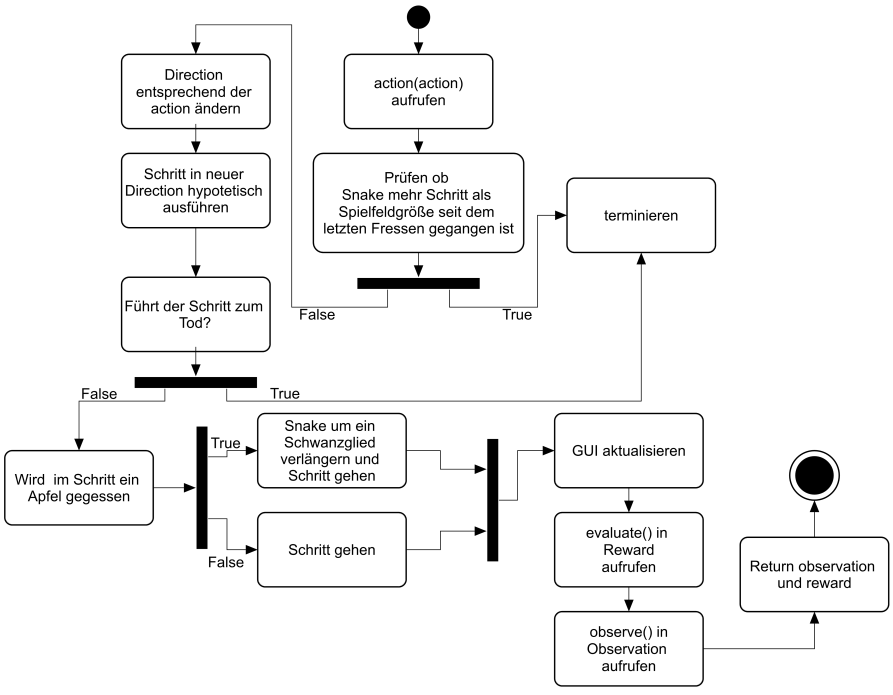

Abbildung 5.3.: Darstellung eines Schrittes in der Spielepisode.

Zu Beginn wird überprüft, ob die Snake seit dem letzten Fressen mehr Schritte als die eigentliche Spielfeldgröße gegangen ist. Diese berechnet sich dabei aus der Spielfeldform (shape). Normal liegt diese bei 64. Im Rahmen der Bestimmung der Robustheit, wird sich diese im Testverlauf jedoch noch ändern (siehe 5.5.1).
Sollte die Snake mehr Schritte gelaufen sein als die Größe des Spielfelds, so wird das Spiel terminiert, da die Snake eventuell in einer Schleife steckt.
Andernfalls wird die Aktion verarbeitet, indem sie die Ausrichtung (direction) der Snake manipuliert. Das Spiel Snake besitzt in diesem Konzept drei Aktionen: turn left, turn right oder do nothing.

Tabelle 5.1.: Kodierung der Aktionen

Aktion	Erklärung
turn left	Die Snake ändert ihre Richtung um 90° nach links. Z.B. Von N $\longrightarrow$ W
turn right	Snake ändert ihre Richtung um 90° nach rechts. Z.B. Von N $\longrightarrow$ O
do nothing	Die Richtung der Snake wird nicht verändert.

Entsprechend der Beispiele in Tabelle 5.1 wird klar, dass die Ausrichtung (direction) entweder nur Norden, Osten, Süden oder Westen sein kann. Als nächstes wird ein Schritt der Snake mit der aktualisierten Ausrichtung hypothetisch durchgeführt. Dabei lässt sich feststellen, ob die Ausführung des Schrittes zum Tod der Snake führt. Sollte dies der Fall sein, so wird der Spielablauf terminiert. Dabei führt das Laufen in sich selbst und das Verlassen des Spielfeldes zum Tod (siehe 2.1).
Anderenfalls wird der Schritt durchgeführt. Dabei wird zwischen zwei Fällen unterschieden.
Sollte die Snake einen Apfel gefressen haben, also Kopf und Apfel die selbe Position einnehmen, so wächst die Snake um ein Schwanzglied. Der alte Apfel wird entfernt und ein neuer erscheint zufallsbasiert irgendwo auf einem freien Feld des Spielfelds.
Sollte die Snake hingegen keinen Apfel gefressen haben, so geht sie einfach den Schritt, es bewegen sich daher alle Schwanzglieder auf die Vorgängerposition, mit Ausnahme des Kopfes, welcher die neue Position einnimmt.
Nach der Ausführung einer dieser beiden Fälle, wird die GUI mit der update_gui Methode aktualisiert. Nach diesem Schritt ist die Abarbeitung der action Methode abgeschlossen.
Damit jedoch der Agent den Nachfolgezustand und Reward erhält, wird die evaluate Methode in der Reward-Komponente und die observe Methode in der Observation-Komponente aufgerufen. Zum Schluss werden Obs und Reward zurückgegeben.

Reward

Die evaluate Methode befindet sich in der Game-Komponente und ruft ihrerseits die standard_reward Methode in der Reward-Komponente auf. Sie bestimmt, basierend auf dem letzten Zug, den Reward. Dies geschieht nach dem folgenden Vorbild. Der Standard Reward ist abhängig von drei Faktoren. Dem Fressen eines Apfels, dem Sieg oder dem Verlust. Sollte keiner dieser genannten Faktoren eintreten, wird ein Reward von -0.01 zurückgegeben. Dies hält den Agenten dazu an den kürzesten Pfad zum Apfel zu finden, da jeder Schritt geringfügig bestraft wird.
War es der Snake möglich einen Apfel zu fressen, so wird ein Reward von +2.5 zurückgegeben, da ein Sub-Goal erfüllt worden ist. Sollte die Snake gestorben sein, durch das Verlassen des Spielfeldes oder das Laufen in sich selbst oder das zu lange Umherlaufen, so wird ein Reward von -10 zurückgegeben, um dieses Verhalten in seiner Häufigkeit zu minimieren. Hat die Snake alle Äpfel gefressen, sodass das gesamte Spielfeld mit der Snake ausgefüllt ist, so wird ein Reward von +10 zurückgegeben, um ein solches Verhalten in seiner Häufigkeit zu maximieren. Dies Snake hat zu diesem Zeitpunkt dann das Spiel gewonnen.
Die zweite Reward Funktion wird im Rahmen der Optimierung A (siehe 5.4.1) näher erklärt.

Observation

Die Observation, welche von der step Methode der Schnittstellen-Komponente (siehe 5.2.2) zurückgegeben wird, besteht aus der around_view (AV) und der scalar_obs (SO). Zur Erstellung der Obs wird die observe Methode in der Game-Komponente (siehe 5.2.1) aufgerufen. Diese ruft wiederum ihrerseits die make_obs Funktion in der Observation-Komponente auf. Mit Hilfe verschiedener Unterfunktionen wird dann die Obs generiert.
Die AV lässt sich dabei als ein Ausschnitt des Spielfeldes (ground) beschreiben, welcher einen festen Bereich um den Kopf der Snake abdeckt. Strukturen wie Wände und Teile des eigenen Schwanzes, welche vielleicht eine Sackgasse aufspannen könnte, werden dadurch deutlich. Mathematisch ist die AV eine one-hot-encoded Matrix der Form (6x13x13).

Das One-Hot-Encoding ist ein binäres encoding System. Sollte ein Merkmal vorhanden sein, so wird dieses mit eins codiert anderenfalls mit null. (Lapan, 2020, S. 359 f.)
Dies ist auch der Grund, warum die AV Matrix sechs Channel (zweidimensionale Schichten) besitzt. Diese geben Aufschluss über folgende Informationen:

Tabelle 5.2.: Channel-Erklärung der Around_View (AV)

Channel der Matrix bzw. Erste Dimension (Ax13x13)	Erklärung
A = 0	Die erste Feature Map signalisiert den Raum außerhalb des Spielfelds. Nährt sich die Snake dem Rand, so würde der Ausschnitt der AV aus dem Spielfeld herausragen und den Eindruck erwecken, dass dieses größer wäre als es in Realität wirklich ist. Darum werden Felder der AV, die sich außerhalb des Spielfelds befinden, angezeigt.
A = 1	Diese Feature Map stellt alle Schwanzglieder mit Ausnahme des Kopfes und es letzten Schwanzgliedes dar.
A = 2	In dieser Feature Map wird der Kopf der Snake dargestellt.
A = 3	Damit gegen Ende des Spiels der Agent noch freie Felder erkennen kann, wird in dieser Feature Map jedes freie und sich im Spielfeld befindliche Feld mit eins codiert.
A = 4	Die vorletzte Feature Map codiert das Schwanzende der Snake.
A = 5	In der letzte Feature Map wird der Apfel abgebildet.

Vorteilhaft an der AV ist, dass, im Gegensatz zu den Obs in den verwandten Arbeiten Wei u. a. (2018) und Chunxue u. a. (2019), nicht das gesamte Feld übertragen wird, sondern nur der wichtigste Ausschnitt, was die Menge an zu verarbeiten Daten reduziert. Des Weiteren ergeben sich keine Probleme zwischen variablen Spielfeldgrößen und der Input-Size von Convolutional Layers (siehe A.3.1).
Ein Nachteil dieser Obs ist jedoch die Unvollständigkeit. Sollte der blaue Punkt (der Apfel) in Abbildung 5.4 außerhalb des grauen Kasten und daher außerhalb der AV liegen, so bleibt der Agent im Unklaren über den Aufenthaltsort des Apfels. Auch Informationen wie z.B. der Hunger, also die verbleibenden Schritte bis das Spiel endet, Distanzen zu den Wänden und Schwanzteilen und die Ausrichtung (direction) der Snake, werden durch die AV nur eingeschränkt oder gar nicht geliefert.

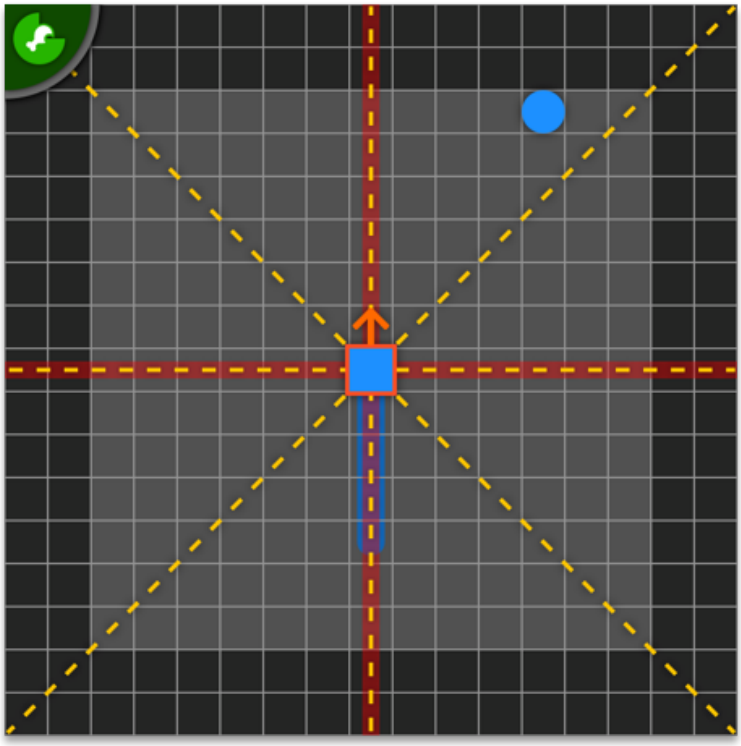

Abbildung 5.4.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar, wobei das rot umrandete Rechteck den Kopf darstellt. Die schwarzen Felder werden nicht von der AV abgedeckt, graue liegen innerhalb der AV. Die gelben gestrichelten Linien stellen eine Raytracing Distanzbestimmung dar. Der blaue Kreis stellt den Apfel dar und der grüne viertel Kreis oben links symbolisiert den Hunger.

Aus diesem Grund wurde die AV durch die scalar_obs (SO) ergänzt. Die SO beinhaltet skalare Informationen und ist eine Konkatenation aus Raytracing Distanzbestimmung, Hunger- und Blickrichtungsanzeige (direction). Zuzüglich werden der SO noch zwei Kompasse für relative Positionsinformation zwischen Kopf und Apfel bzw. letztem Schwanzglied hinzugefügt.
Letztere sind eindimensionale Vektoren, welche über das One-Hot-Encoding anzeigen, ob sich das gesucht Objekt relativ zum Kopf oberhalb, unterhalb oder in der selben Zeile befindet (Matrixsicht). Analog verhält es sich mit der vertikalen Sicht. Die Blickfeldanzeige ist ebenfalls one-hot-encoded und stellt mit ihrem Vektor die vier Ausrichtungen Norden, Osten, Süden und Westen dar.
Der Hunger ist die Differenz zwischen der Anzahl der gegangenen Schritten seit dem letzten Fressen (inter_apple_steps) und der maximalen Schrittanzahl (max_steps). Diese maximale Schrittanzahl ist als Spielfeldgröße definiert. Da der Hunger bei einer großen Differenz einen kleinen Einfluss und bei einer geringen Differenz einen großen Einfluss besitzen soll, wurde dieser durch eins geteilt. Nähren sich die beiden Werte im Nenner an, so rückt der gesamt Wert des Bruches näher an unendlich, da

den Nenner immer kleiner wird.

$$\min(2, \frac{1}{inter\_apple\_steps - max\_steps}) \tag{5.1}$$

Um mit der Unendlichkeit auftretende Probleme zu umgehen wird zwei zurückgegeben, wenn die Differenz null ist.
In ähnlicher Weise wird mit den Raytracing Distanzbestimmungen verfahren. Bei ihnen handelt es sich um acht Distanzmesserlinien, die in 45° Abständen ausgesandt werden, siehe Abbildung 5.4. Befindet sich das gesucht Objekt in dieser Linie, so wird die Distanz zwischen dem Kopf der Snake und dem Objekt durch eins dividiert und zurückgegeben. Es wird dabei nach Wänden, dem eigenem Schwanz und dem Apfel gesucht. Daher wird die Raytracing Distanzbestimmung in einem Vektor der Größe 24 (3 * 8 = 24) gespeichert.

GUI

Die graphische Oberfläche oder auch GUI genannt kann optional ein- oder ausgeschaltet werden. Beim Lernen der Agenten bietet es sich beispielsweise an diese auszuschalten, da diese die Lerngeschwindigkeit senkt. Beim Start der GUI wird ein Fenster geöffnet, welches den momentanen Stand der Spielgeschehens anzeigt. Da Snake eine zweidimensionale Spieloberfläche besitzt, wird diese im Fenster dargestellt. Nach jeder Aktionsdurchführung muss die GUI mit der update_gui Methode aktualisiert werden, um stets den neusten Stand des Spiels zu zeigen.

5.2.2. Schnittstelle

Die Schnittstelle umschließt die bereits erwähnte Spiellogik-Komponente mit ihren Unterkomponenten, um eine standardisierte Schnittstelle zu erzeugen.

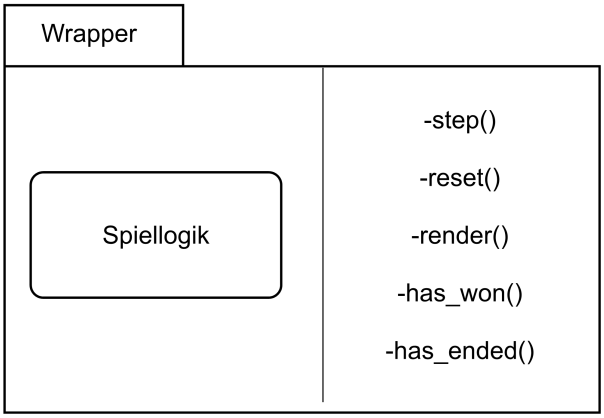

Abbildung 5.5.: Darstellung der Schnittstelle.

Die step Funktionalität (siehe 5.5) stellt die Hauptmethode der Schnittstellen-

Komponente dar. Sie bekommt eine Aktion übergeben, welche zuvor von einem Agenten bestimmt wurde. Diese wird mit Hilfe der Spiellogik umgesetzt. Entsprechend der Anforderung der Standardisierte Schnittstelle (siehe 3.1.1) gibt diese Methode nur Reward und Observation zurück.
Reset setzt den bereits vorhandenen Spielfortschritt zurück, wobei die reset Methode der Game-Komponente aufgerufen wird. Dies soll entsprechend der Reset Anforderung (siehe 3.1.2) geschehen.
Die Render Methode ist für die Visualisierung verantwortlich und ruft die update_gui Methode in der GUI-Unterkomponente auf, welche das Spielfeld, entsprechend der Render Anforderung (siehe 3.1.2) graphisch darstellt.
Die has_won und has_ended Methoden geben Statusinformationen über den momentanen Spielstand zurück, welche für die Test- und Trainingsabläufe benötigt werden.

5.3. Agenten

In diesem Abschnitt des Konzepts sollen die Agenten inklusive ihrer Netzstruktur vorgestellt werden. Zu diesem Zweck müssen die Algorithmen (DQN und PPO) näher beleuchtet werden.

5.3.1. Netzstruktur

Zu Beginn soll die Netzstruktur erklärt werden, wobei dies unabhängig von den Algorithmen geschehen kann, da sowohl DQN als auch PPO Agenten das annähernd gleiche Netz nutzen.

Im Rahmen dieser Ausarbeitung soll sich nur auf eine Netzstruktur konzentriert werden, um die Vergleichbarkeit der einzelnen Algorithmen zu erhöhen. Dennoch müssen, aufgrund des Algorithmus, kleinere Anpassungen an der Netzstruktur vorgenommen werden. Diese werden im Weiteren erklärt.
Dieses NN stellt dabei einen Kompromiss zwischen Vollständigkeit und Effizienz dar. In den Papers von Wei u. a. (2018) und Chunxue u. a. (2019) wurden nur großes CNN (siehe A.3) genutzt, die Gefahr laufen viele unnötige Informationen verarbeiten zu müssen. Zusätzlich können noch Probleme mit variablen Spielfeldgrößen auftreten, sodass in dieser Ausarbeitung auf ein zweiteilige Netzstruktur gesetzt wird. Diese besteht aus dem AV-Network und dem Actor-, Critic- oder Q-Tail. Zuerst wird die AV durch das AV-Network geleitet. Dabei wird die AV durch zwei Convolutional Layer mit einer ReLU Aktivierungsfunktion propagiert. Dabei erhöht sich die Channel-Anzahl auf acht, wobei eine weitere Erhöhung der Channel-Anzahl aufgrund der bereits sehr stark optimierten AV nicht nötig ist. Die Feature Map wird während dieses Prozesses nicht minimiert, aufgrund des Paddings der Convolutional Layers. Dies soll den Informationsverlust an den Rändern minimieren.
Danach werden allen Feature Maps eine Null-Zeile und Null-Spalte hinzugefügt (Padding), damit beim Max-Pooling, unter der Filtergröße und dem Stride von 2x2, auch die letzten Zeile und Spalte verarbeitet werden.

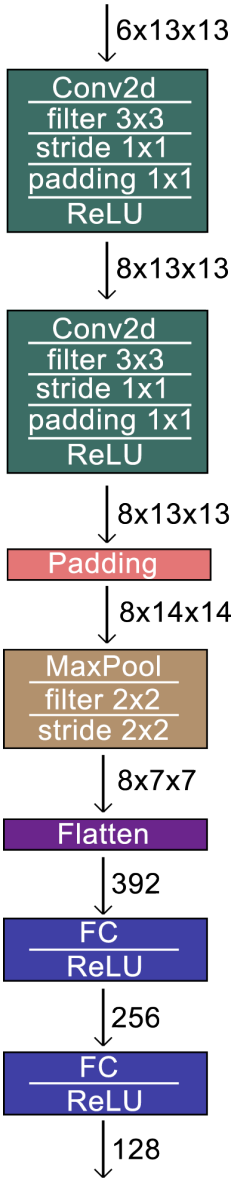

Abbildung 5.6.: AV-Network

Der Tensor besitzt nun die Form (8x14x14). Nach dem
max-pooling besitzen die Feature Maps jedoch nur noch eine Größe 7x7 (Tensor: 8x7x7).
Dann folgt die Einebnung (Flatten) zu einem eindimensionalen Tensor, welcher daraufhin durch zwei weitere Fully Connected Layer (FC) mit einer ReLU Aktivierungsfunktion propagiert wird. Der resultierende Tensor besitzt die Größe 1x128 und ist ein Zwischenergebnis, da dieser nun mit der SO verbunden wird (Join). Der Vorgang ist in der Abbildung 5.6 dargestellt.

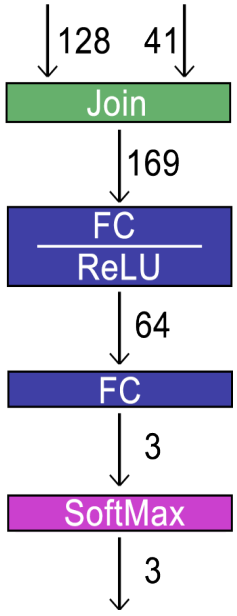

Abbildung 5.7.: Actor-Tail

Da das NN in beide Algorithmen verwendet wird, müssen Netzwerkschwänze für den Actor, Critic und für das Q-Network (Q-Net) definiert werden. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nachdem der Joined Tensor (1x169), welcher aus dem Zwischenergebnis des AV-Networks und der SO besteht, durch zwei weitere FC Layer mit ReLU Aktivierungsfunktion propagiert wurde, benötigt der Actor des PPO Algorithmus eine Wahrscheinlichkeitsverteilung über alle Aktionen. Daher auch die Ausgabe von einem Tensor der Größe drei. Um diese Wahrscheinlichkeitsverteilung zu erhalten, wird die SoftMax Funktion angewendet, siehe Abbildung 5.7.
Der Critic des PPO Algorithmus verwendet hingegen den Critic-Tail, siehe Abbildung 5.8 links. Dieser leitet den Joined Tensor durch ein weiteres FC Layer mit ReLU Aktivierung. Der Resultierende Output wird danach durch ein weiteres FC Layer ohne Aktivierung geleitet. Da der Critic für jeden State die Discounted Sum of Rewards bestimmt (siehe 2.3.2), gibt dieser einen Tensor mit einem einzigen Wert zurück (Skalar). Der Q-Net-Tail ist in seinem Aufbau sehr ähnlich zum Critic-Tail. Da dieser jedoch die Q-Values für jede Aktion im Zustand angeben soll, muss ein Tensor der Größe drei zurückgegeben werden. Von der Struktur des Netzes sind jedoch der Critic- und Q-Net-Tail, mit Ausnahme der Ausgabeschicht, gleich (siehe 5.8).

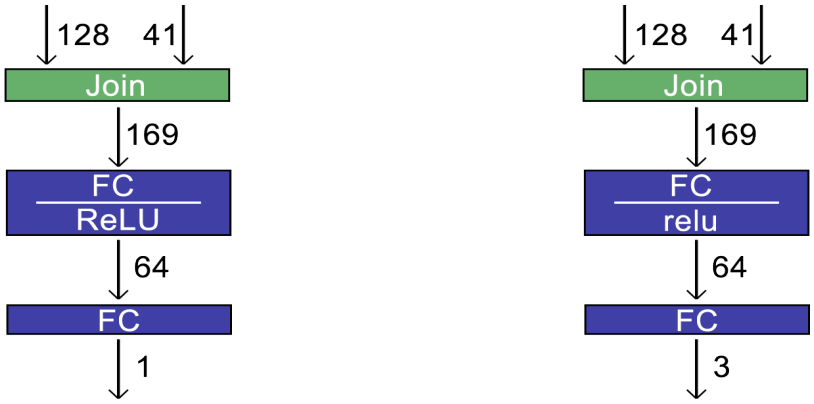

Abbildung 5.8.: Darstellung des Critic-Tail (links) und des Q-Net-Tail (rechts).

5.3.2. DQN

Der DQN Algorithmus und damit auch die Agenten, welche auf diesem basieren, bestehen aus drei Komponenten. Diese ermöglichen die Implementierung der Hauptmethoden act und learn.

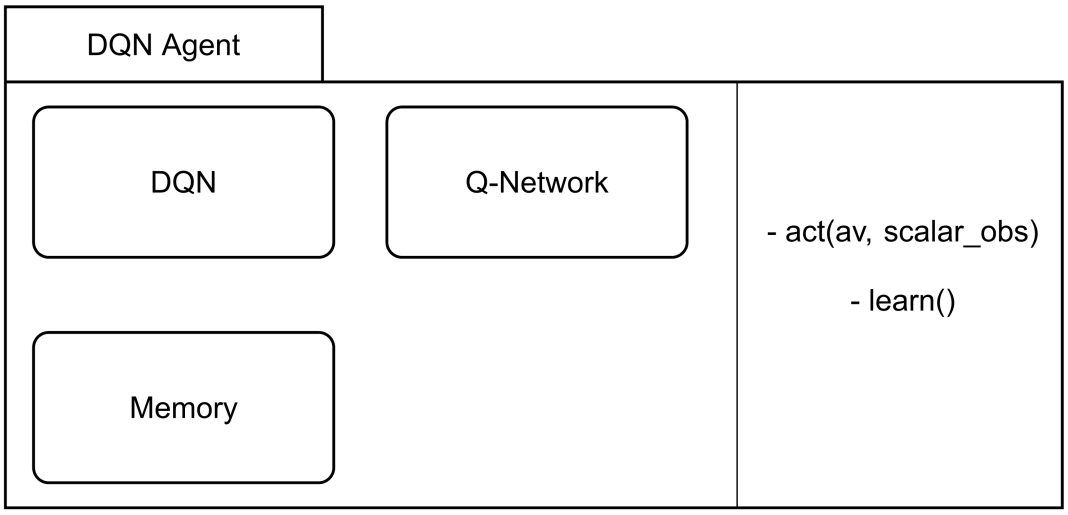

Abbildung 5.9.: Darstellung des DQN-Agent mit seinen Komponenten.

Diese Hauptmethoden sind in der DQN-Komponente eingebettet, welche die zentrale Instanz des DQN darstellt. In ihr werden zudem die Memory und Q-Network Komponente verwaltet. Des Weiteren werden wichtige Konstanten für den DQN Algorithmus, wie z.B. Gamma, Epsilon (eps), Epsilon-Dekrementierung (eps_dec), der minimal Wert für Epsilon (eps_min), die Batch-Size (batch_size), die maximale Größe des Memory (max_mem_size) und die Lernrate (lr), in der DQN-Komponente gespeichert.
Die Memory-Komponente speichert die gesammelten Erfahrungen des DQN-Agenten in einer Ring-Buffer Struktur. Sollte dieser Buffer voll sein, so werden die ältesten Erfahrungen, welche am Anfang stehen, mit den neuen überschrieben. Die gespei-

cherten Erfahrungen werden im weiteren Verlauf von der learn Methode abgerufen, um mit ihnen den Lernprozess durchzuführen. Abzuspeichernde Werte, für jeden Schritt, sind dabei die around_view (AV), die scalar_obs (SO) die Aktion (action), der Reward (reward), die Information, ob man sich in einem terminalen Zustand befindet (terminal) und die around_view (AV_) und scalar_obs (SO_) des Nachfolgezustandes.
Die Q-Network-Komponente verwaltet das NN (Q-Network). Dieses wird von den act Methoden dazu genutzt, um die Aktionen für das Env zu bestimmen. Des Weiteren wird das Q-Network durch die learn Methode aktualisiert, sodass eine höhere Performance erreicht werden kann.

Aktionsauswahlprozess

Zur Bestimmung der nächsten Aktion wird der act Methode die momentane Obs (AV, SO) übergeben. Diese generiert einen Zufallswert zwischen null und eins, was den Wahrscheinlichkeiten von 0% bis 100% entspricht. Ist der Zufallswert größer als den momentane Epsilon-Wert, so wird die Aktion durch das Q-Network bestimmt. Anderenfalls wird eine zufällige Aktion ausgewählt. Die Bestimmung der Aktion durch das Q-Network geschieht dabei wie folgt:
Die around_view (AV) und die scalar_obs (SO) werden durch das Q-Network, entsprechende der Ausführungen in Abschnitt 5.3.1, geleitet. Dieses gibt einen Tensor der Größe drei wieder, welche die Q-Values der Aktionen turn left (0), turn right (1) und do nothing (2) beinhaltet. Es wird daraufhin die Aktion gewählt, welche dem Index des größten Q-Values entspricht.
Sei (0.32, -0.11, 0.45) ein Tensor, welcher vom Q-Network zurückgegeben wurde, dann würde no nothing (2) gewählt werden, da 0.45 der größte Q-Value ist und dieser an Stelle 2 steht.
Die oben beschriebene Prozedur stellt dabei den Aktionsauswahlprozess während des Trainings dar. Sollten Testläufe, mit einem beispielsweise angelernten Netz, durchgeführt werden, so wird die Aktion immer durch das Q-Network bestimmt.
Zum Schluss wir die Aktion zurückgegeben und die Methode terminiert. Eine genaue Darstellung der Aktionsbestimmung befindet sich in Abbildung 5.10.

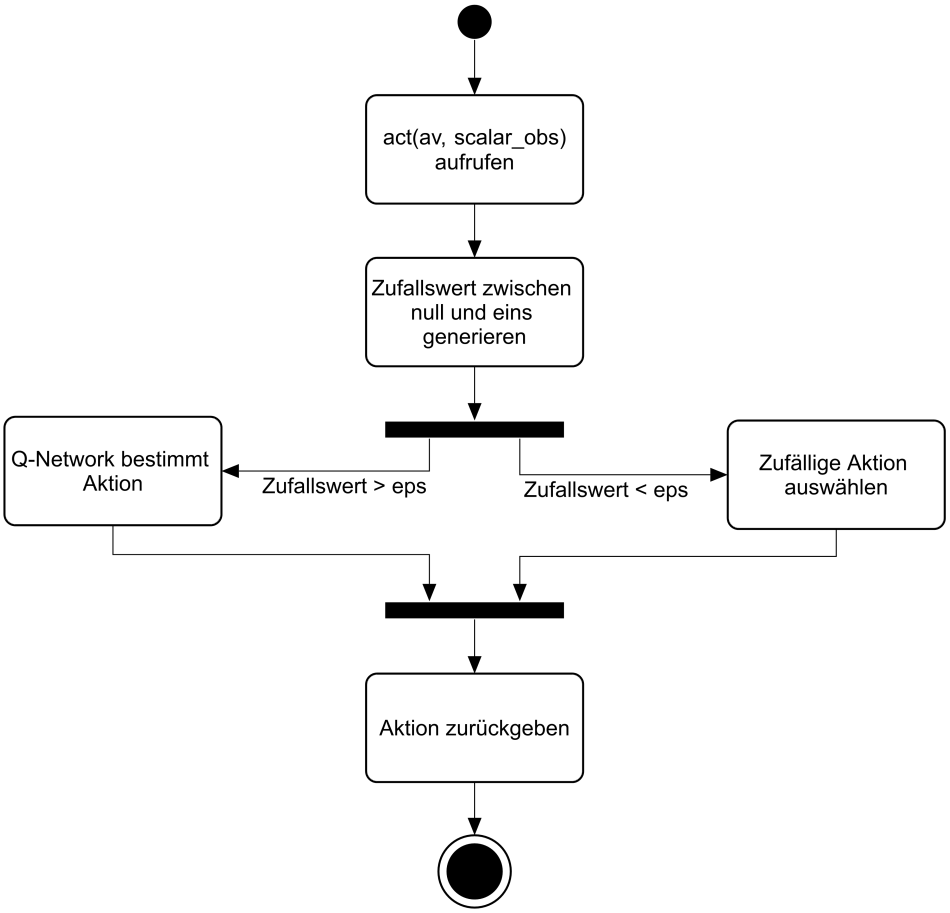

Abbildung 5.10.: Darstellung der Aktionsbestimmung des DQN-Agent.

Lernprozess

Der Lernprozess wird über die learn Methode ausgeführt und stellt sich dabei wie folgt dar:
Zuerst wird überprüft, ob im Memory genügend Experiences (Exp) gespeichert sind, um einen Mini-Batch mit der zuvor definierten Batch-Size, zu extrahieren. Sollte dies nicht der Fall sein, wird die Methode terminiert. Anderenfalls wird ein Mini-Batch aus zufälligen Exp ohne Duplikate gebildet.
Danach wird der Q-Value bestimmt, welcher zu der abgespeicherten Aktion gehört. Es wird daher $Q(s_i, a_i; \theta_i)$ bestimmt (siehe 2.10), wobei s_i die Observation (AV und SO), a_i die gewählte Aktion im Zustand s_i und θ die Netzwerkparameter des Q-Network, darstellten. Dieser wird als Q-Eval definiert.
Danach werden die Q-Values des Nachfolgezustandes (AV_ und SO_) bestimmt. Sollte der Nachfolgezustand ein terminaler Zustand sein, so wird der Q-Value auf null gesetzt, da die Q-Values die zu erwartende Discounted Sum of Rewards angeben. In einem terminalen Zustand ist diese gleich null, da keine Zustande mehr besucht werden können (siehe 2.4).
Daraufhin wird der maximale Q-Value bestimmt, mit Gamma multipliziert und

mit dem erhaltenen Reward addiert. Es wird daher $r(s,a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$ bestimmt (siehe 2.10). Dieser Wert wird als Q-Target definiert und soll Q-Eval entsprechen.
Am Ende wird der Mean Squared Error zwischen Q-Targets und Q-Evals aus dem Mini-Batch gebildet. Auf Basis dieses Fehlers soll das Q-Network mittels Backpropagation und Gradientenverfahren (siehe A.1) angepasst werden.

5.3.3. PPO

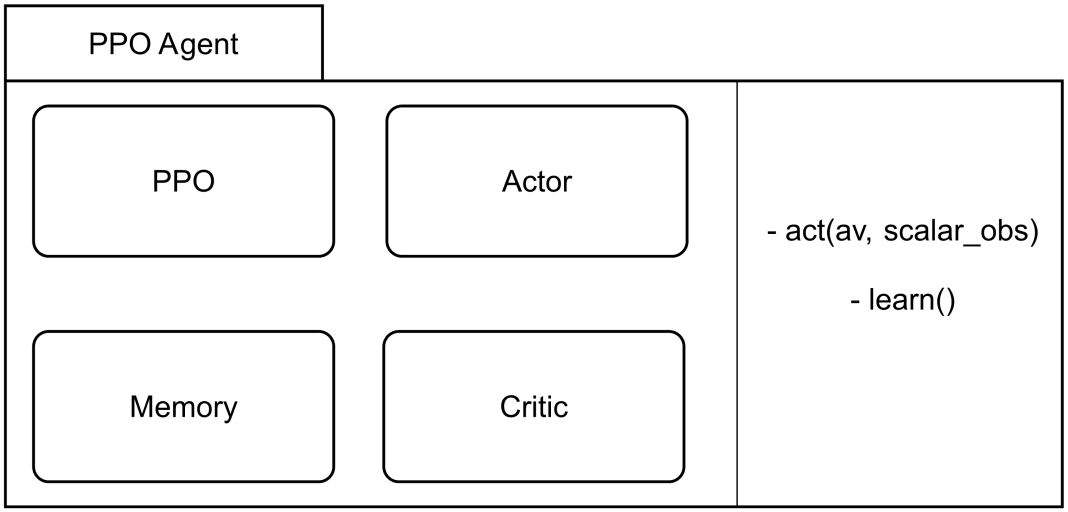

Abbildung 5.11.: Darstellung des PPO-Agent mit seinen Komponenten.

Der PPO Algorithmus und seine Agenten, bestehen aus vier Komponenten. Diese ermöglichen die Implementierung der Hauptmethoden act und learn.
Diese Hauptmethoden sind in der PPO-Komponente eingebettet, welche die zentrale Instanz des DQN darstellt. In ihr werden zudem die Memory-, Actor- und die Critic-Komponenten verwaltet. Des Weiteren werden wichtige Konstanten für den PPO Algorithmus, wie z.B. Gamma (gamma), der Epsilon-Clip-Wert (eps_clip) (siehe 2.3.2), die Anzahl der Trainingsläufe pro Datensatz (K_Epochs), die Lernrate (lr) und weitere statische Konstanten, in der PPO-Komponente gespeichert.
Die Memory-Komponente speichert die gesammelten Erfahrungen eines PPO-Agenten. Diese werden im weiteren Verlauf von der learn Methode abgerufen, um mit ihnen den Lernprozess durchzuführen. Abzuspeichernde Werte, für jeden Schritt, sind dabei die around_view (AV), die scalar_obs (SO) die Aktion (action), der Reward (reward), die Information, ob man sich in einem terminalen Zustand befindet (terminal) und die logarithmierte Wahrscheinlichkeit der ausgewählten Aktion (log_prob).
Die Actor-Komponente verwaltet das Actor-NN. Dieses wird von der act Methode dazu genutzt, um die Aktionen für das Env zu bestimmen. Des Weiteren wird das Actor-NN durch die learn Methode aktualisiert, sodass eine höhere Performance er-

reicht werden kann.
Die Critic-Komponente verwaltet das Critic-NN. Dieses wird einzig von der learn Methode verwendet, um die erwartete Discounted Sum of Rewards zu bestimmen. Mit dieser wird im Trainingsverlauf der Value-Loss bestimmt (siehe 2.8).

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird durch die act Methode in der PPO-Komponente angestoßen, welche die AV und SO übergeben bekommt. Dieser werden sogleich durch das Actor-NN, welches sich in der Actor-Komponente befindet, propagiert. Der vom Actor-NN ausgegebene Tensor der Größe drei (drei mögliche Aktionen), beinhaltet eine Wahrscheinlichkeitsverteilung. Auf Basis dieser Verteilung wird die nächste Aktion bestimmt.
Sei (0.05, 0.05, 0.9) die Wahrscheinlichkeitsverteilung über alle Aktionen. Bestimmte man 100 Aktionen unter dieser Verteilung, so würde durchschnittlich 90-mal die Aktion zwei gewählt werden. Aktion null und eins nur rund fünfmal.
Wie auch beim DQN, wird für die Testläufe immer die Aktion ausgewählt, welche die größte Wahrscheinlichkeit vorweist (siehe 5.3.2).
Zum Schluss wird die Aktion zurückgegeben und die act Methode wird terminiert.

Lernprozess

Der Lernprozess des PPO wird durch die learn Methode angestoßen. Dabei wird wie folgt verfahren:
Zu Beginn werden die Erfahrungen, aus den gespielten Spielen, aus dem Memory (Replay-Buffer) extrahiert. Der Memory bzw. Replay Buffer befindet sich in der Memory-Komponente.
Um den Return (siehe 2.3.2) zu erhalten, werden die einzelnen Rewards aus dem Memory, welche extrahiert worden sind, diskontiert. Sollte der Reward dabei aus einem terminalen Zustand stammen, so wird dieser auf null gesetzt, da der Return der Discounted Sum of Rewards entsprechen soll (siehe 5.3.3).
Um ein gleichmäßigeres Lernen zu unterstützen, werden die Rewards in Anschluss noch normalisiert. Danach wird die folgende Prozedur (K_epochs) mal ausgeführt, um das NN zu trainieren. Danach terminiert die learn Methode.
Zu nächstes werden die logarithmierte Wahrscheinlichkeiten $\pi_\theta(a_t|s_t)$ für die gespeicherten Aktionen a_i bestimmt (siehe 2.3.2). Dazu werden die aus dem Memory entnommenen AVs (around_views) und SOs (scalar_obs) durch das Actor- und Critic-NN propagiert. Anschließend werden die logarithmierten Wahrscheinlichkeiten für die Aktionen bestimmt und zusammen mit den Baseline Estimates (siehe 2.3.2) und den Entropien der Wahrscheinlichkeitsverteilungen (siehe 2.3.2) zurückgegeben.

Daraufhin werden die Probability Ratios aus der eben bestimmten logarithmierten Wahrscheinlichkeiten und den alten logarithmierten Wahrscheinlichkeiten des Memories bestimmt (siehe2.3.2). Nachfolgend werden die Advantages durch Subtraktion der Returns mit den Baseline Estimates berechnet $\hat{A}_t(s,a) = R_t - b(s_t)$ (siehe 2.3.2).
Als nächstes werden die Surrogate Objective Losses Surr1: $r_t(\theta)\hat{A}_t(s,a)$ und Surr2: $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s,a)$ bestimmt (siehe 2.3.2), mit welchen der Actor-Loss $L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t(s,a), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s,a))]$ berechnen wird (siehe 2.2). Um den gesamt Loss des PPO zu bestimmen, wird zusätzlich noch der Value-Loss $L_t^{\text{VF}} = (V_\theta(s_t) - V_t^{targ})^2$ wobei $V_t^{targ} = r_t(\theta)$ und der Entropy-Loss bestimmt (siehe 2.3.2). Diese werden dann alle zusammengerechnet, entsprechend der Formel: $L_t^{\text{PPO}}(\theta) = L_t^{\text{CLIP + VF + S}}(\theta) = \hat{\mathbb{E}}_t[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}} + c_2 S[\pi_\theta](s_t)]$ (siehe 2.3.2).
Das Actor- und Critic-NN werden mit dem Loss unter Zuhilfenahme von Backpropagation und Gradientenverfahren aktualisiert (siehe A.1).

5.3.4. Vorstellung der zu untersuchenden Agenten

Ein zentraler Aspekt eines Vergleichs von verschiedenen RL-Agenten ist die genaue Definition dieser einzelnen. Basierende auf den Grundlagen (siehe 2.2.1), soll in diesem Abschnitt die zu vergleichenden Agenten vorgestellt werden.
Da die ausgewählten Hyperparameter einen immensen Einfluss auf das Verhalten der Agenten besitzen, ist ein Vergleich zwischen DQN und PPO Agenten mit wahllos gewählten Hyperparametern folglich wenig aussagekräftig. Darum sollen im Weiteren die Wahl der Hyperparameter der Agenten hier begründet werden.

Wie bereits in der Abbildung 5.1 zu erkennen ist, sollen 6 Agenten definiert und miteinander verglichen werden.

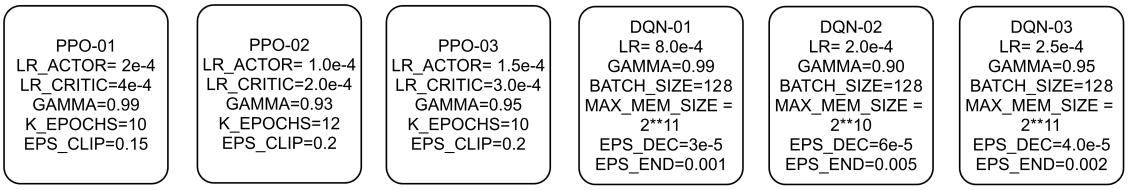

Abbildung 5.12.: Darstellung der zu untersuchenden Agenten.

Der erste Agent PPO-01 soll ein langsamer aber stetiger Lerner sein. Mit einer ACTOR-LR von 2e-4 und einer CRITC-LR von 4e-4 wurden Lernraten gewählt, welche, spezifisch für diese Netzstruktur, im Mittelfeld liegen. Ein hoher Wert für GAMMA von 0.99 sorgt für ein zukunftsorientiertes Lernen. Damit der PPO-01 keine zu großen Aktualisierungen der Netze unternimmt, wurde EPS_CLIP auf 0.15

gesetzt, was, verglichen mit der Literatur (Schulman u. a., 2017, S. 6), recht niedrig ist.

Der PPO-02 soll ein schnell lernendes Verhalten zeigen. Zu diesem Zweck wurden zwar niedrige Lernraten von 1e-4 (Actor) und 2.5e-4 (Critic) gewählt, jedoch sorgt der relativ große K_EPOCHS-Wert von zwölf für ein stärkeres Aktualisieren der Netzwerkparameter von Actor und Critic. Dies wird ebenfalls durch den EPS_CLIP-Wert von 0.25 unterstützt, welcher, nach der PPO Literatur (Schulman u. a., 2017, S. 6), höher als der Durchschnittswert ist. Der GAMMA-Wert von 0.95 bestärkt zudem den schnelleren Lernerfolg, aufgrund der Kurzzeitpräferenz des Agenten.

PPO-03 soll ein Kompromiss zwischen schnellen Lernen und stetigem Fortschritt sein. Mit mittleren Lernraten von 1.5e-4 (Actor) bzw. 2.5e-4 (Critic) sollte ein schneller und zugleich stetiger Lernfortschritt erzielt werden. Der GAMMA-Wert von 0.95 soll das schnelle Lernen unterstützen. Auch die Werte von K_EPOCHS mit zehn und EPS_CLIP von 0.2 werden in der Literatur (Schulman u. a., 2017, Anhang A) empfohlen und stellt ein gutes Mittelmaß dar.

Der DQN-01 ist wieder als langsamer Lerner gedacht. Mit einer großen LR von 8.0e-4 und einem großen GAMMA-Wert von 0.99, wird ein stetiges und zukunftsorientiert Lernen bestärkt. Eine BATCH_SIZE von 128 soll zudem das Lernen beständiger machen. Geringe Werte für Eps_Dec und EPS_MIN von 3e-5 und 0.001 sollen die Neugierde des Agenten zu Beginn stärken und die Wahl von Zufallsaktionen in späteren Trainingsphasen senken.

DQN-02 ist wieder als Schnelllerner konzipiert worden. Eine vergleichsweise hohe Lernrate (LR) von 2.0e-4 in Verbindung mit einem kleinen Wert für Gamma von 0.90 soll einen schnellen Lernfortschritt generieren. Dies wird durch die niedrige Memory-Size (MAX_MEM_SIZE) von 2**10 und die große Epsilon-Dekrementierung (EPS_DEC) von 5e-5 verstärkt. Auch sorgt der verhältnismäßig große Wert für EPS_MIN von 0.0075 für eine schnellerer Exploration und damit für ein schnelles Lernen.

Der DQN-03 ist wieder als Kompromiss gedacht und besitzt die folgenden Hyperparameter: LR = 2.5e-4, GAMMA = 0.95, BATCH_SIZE = 128, MAX_MEM_SIZE = 2**11, EPS_DEC = 4e-5 und EPS_END = 0.002.

5.4. Optimierungen

In diesem Abschnitt werden die anzuwendenden Optimierungen vorgestellt und erklärt, welche nach dem Baseline-Vergleich die Leistung in den einzelnen Evaluationskategorien noch weiter verstärken soll. Zu diesem Zweck sollen zwei Optimierungen auf die Baseline Agenten (Agenten ohne Optimierungen) angewendet werden. Die Erstellung von Optimierung A wurde durch die verwandten Arbeiten Chunxue u. a. (2019) und Wei u. a. (2018) unterstützt. Die Optimierung B wurde nach dem Lesen der Literatur (Lapan, 2020, S. 331 f.) entwickelt.

5.4.1. Optimierung A - Joined Reward Function

Die Joined Reward Function wurde im Paper „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ Wei u. a. (2018) vorgestellt und im Abschnitt 4.1 erklärt. Sie setzt sich aus drei Teilen zusammen. Die Basis bildet ein Distanz Reward, welcher abhängig von der Distanz und Schwanzlänge ist. Um unerwünschte Lerneffekte, von beispielsweise der Neuerzeugung eines Apfels, zu verhindern werden diese Erfahrungen nicht im Memory gespeichert. Zur Verstärkung des Pathfindings wird die Timeout Strategy angewendet, welche den Agenten für nicht zielgerichtetes Verhalten, wie z.B. das unnötige Umherlaufen, bestraft.

Eine genaue Implementierung dieser vorgestellten Reward Funktion erscheint nicht Sinnvoll, da bereits in der Diskussion (siehe 4.1.1) zur Quelle Wei u. a. (2018) festgestellt worden ist, dass die Agenten nicht das effiziente Lösen des Snake-Spiels konzipiert worden ist, sondern für das lange Überleben. Daher muss die Reward Funktion entsprechenden Verhalten begünstigen. Dennoch erscheint die Adaptierung einiger Reward Funktion Elemente als sinnvoll, um eine eigens für die Performance und Effizienz optimierte Reward Funktion zu designen.
Die Implementierung dieser neuen Reward Funktion findet in der Reward-Komponente statt. Dabei wird der Reward wie folgt berechnet:

$$r_{distanz}(dis, len, size) = \frac{10}{dis} \times \frac{len}{size} \tag{5.2}$$

dis stellt die Distanz gemessen mit der Euklidischen Norm dar, len ist die Länge der Snake und $size$ ist die Größe des Spielfeldes (bei einer Spielfeldform von 8x8 ergibt sich eine Größe von 64).
Da die Training Gap Strategy zu Problemen beim PPO führen würde, wird diese nicht beachtet. Die Timeout Strategy wird in abgewandelter Form implementiert.

Dies geschieht nach folgender Formel:

$$r_{timeout}(steps, len) = \frac{1}{100} \times \frac{steps}{len} \tag{5.3}$$

steps beschreibt die Anzahl an Schritten, welche seit dem letzten Konsum eines Apfels gelaufen worden sind. Die Joined Reward Function lautet daher: $r_{res}(dis, len, size, steps) = r_{distanz}(dis, len, size) - r_{timeout}(steps, len)$.

5.4.2. Optimierung B - Anpassung der Lernrate

Die dritte Optimierung versucht das Lernen des Agenten durch das stetige Anpassen der Lernrate (LR) zu verbessern. Die Lernrate wird immer dann mit 0.95 multipliziert und als neue LR gesetzt, sobald keine Performance Steigerung in den letzten 200 Schritten erfolgt ist.

5.5. Datenerhebung und Verarbeitung

In diesem Abschnitt soll die Datenerhebung näher thematisiert werden. Zu diesem Zweck wird soll zuerst die Hauptmethode näher erklärt werden, welche Agenten und Env mit einander interagieren lässt. Danach wird die Statistik-Komponente mit ihren Methoden vorgestellt.

5.5.1. Datenerhebung

Die Hauptmethode oder auch main Methode genannt, implementiert den eigentlichen Spiel und Trainingsablauf. Die main Methode der Agenten ist außerhalb der der Komponenten der Agenten zu finden, da beide Algorithmus-Arten und ihren Agenten, die gleiche Standardisierte Schnittstelle verwenden. 3.1.1. Um den Spiel- und Trainingsablauf durchzuführen, wird die train Methode der DQN bzw. PPO Agenten aufgerufen, welche wichtige Hyperparameter des Ablaufs, wie z.B. die zu absolvierenden Trainingsspiele (N_ITERATIONS), die Spielfeldgröße (BOARD_SIZE) und weitere Agenten spezifische Hyperparameter, übergeben bekommt.
Um einen angemessenen Zeitraum für das Lernen zu schaffen, soll 30.000 Spiele bzw. Epochs (N_ITERATIONS) für einen Trainingslauf absolviert werden. Die Spielfeldgröße soll dabei standardmäßig, für das Training bei (8x8) liegen. Sollte es der Agent jedoch vorher schaffen eine Siegrate von 60% zu erreichen, so stoppt der Trainingsprozess, um Overfitting zu vermeiden.
Bei der Datenerhebung ist streng zwischen Test- und Trainingsdaten zu unterscheiden, wobei die Testdaten aus einfachen Spielabläufen und die Trainingsdaten aus den Trainingsabläufen stammen. Die Trainingsdaten werden wie folgt erhoben.

Zu Beginn werden die Datenhaltungsobjekt apples, wins, time, (eps für DQN), steps initialisiert, welche für jedes absolvierte Training die, dem Namen des Objektes entsprechenden, Werte speichert. Nach der weiteren Erstellung des Agenten und Environments startet der Spielverlauf.
Dabei wird wie in 2.2.2 vorgegangen. Der Agent erhält eine Obs, bestimmt seine Action und diese wird sogleich im Env ausgeführt. Danach werden die neue Obs sowie Reward und weitere Statusinformationen, wie z.B. der Lebensstatus (done bzw. terminal) usw., ausgegeben. Diese Daten werden im Memory des jeweiligen Agenten für das sich anschließende Training gespeichert. Dieses wird wie in DQN-Lernprozess (siehe 5.3.2) bzw. PPO-Lernprozess (siehe 5.3.3) durchgeführt. Danach werden die oben genannten Datenhaltungslisten aktualisiert und die Prozedur beginnt von neuem. Wenn diese N_ITERATIONS Spiele alle absolviert wurden werden die Erhobenen Daten weiter in der Statistik-Komponente verarbeitet 5.5.2.
Die Datenerhebung für Testdaten findet annähernd auf die gleiche Weise statt. Jedoch werden die Daten in nicht aus dem Trainingslauf entnommen, sondern aus einem Testlauf. Bei den Daten handelt es sich um die gleichen, wie in der oben erwähnten Datenerhebung für Trainingsdaten. Der Unterschied zwischen Test und Trainingslauf besteht darin, dass die Spielfeldgröße im Rahmen des Testlaufes für die Robustheit variiert wird. Sie kann dabei Größen von (6, 6) bis zu (10, 10) annehmen. Diese werden dann ebenfalls der Statistik-Komponente übergeben (siehe 5.5.2).

5.5.2. Datenverarbeitung und Erzeugung von Statistiken

Nach der Erstellung der Trainings- und Testdaten für jeden Agenten, wenden diese Daten entsprechend der Evaluationskriterien (siehe 3.2) verarbeitet. Für die Erhebung der Test-Spiele wurden 5.000 Spiele bzw. Epochs und für die Erhebung der Trainingsdaten werden 30.000 Spiele bzw. Epochs absolviert. Solle während des Trainings ein Agent die Abbruchbedingung erreicht haben, entsprechend weniger als 30.000 Spiele bzw. Epochs. Die Abbruchbedingung ist im Abschnitt Datenerhebung (siehe 5.5.1) thematisiert.
Diese Verarbeitung sieht wie folgt aus.
Die Performance wird anhand der gefressenen Äpfel gemessen. Das Datenobjekt apples besitzt diese Daten für jeden Testdurchlauf.
Die Effizienz errechnet sich aus der Anzahl der gefressenen Äpfel (apples) dividiert durch die gegangenen Schritten (steps). Es entsteht daher die Apfel pro Schritt Rate. Es werden dafür Testdaten verwendet.
Die Robustheit wird gleich der Performance gemessen, nur werden dafür Testdaten, die auf Spielfeldern mit variabler Größe erhoben worden sind, genutzt. Es werden

daher die gefressenen Äpfel (apples) in der neuen Umgebung gemessen. Die Spielfeldgröße soll dabei zwischen (6x6) bis zu (10x10) liegen.
Die Siegrate wird mit Hilfe der Wins bestimmt. Dabei werden die Siege der jeweils zehn letzten Spiele zusammenaddiert und durch die Trainingsspiel dividiert, sodass die Rate Siege der letzten zehn Spiele pro Spiel entsteht.

Diese Daten werden dann mit Hilfe von Statistiken ausgewertet. Dabei soll jeweils eine Statistik pro Evaluationskriterium angefertigt werden.

Kapitel 6

Implementierung

Für eine Umsetzung eines solchen Konzepts, wie es in dem Kapitel 5erwähnt worden ist, wird es nötig eine Implementierung des Spiels Snake, der beiden Algorithmus-Arten, Ablaufroutine sowie der Statistik-Erzeugung durchzuführen. Als Programmiersprache wurde Python (3.7) gewählt.
Python bietet im Bereich des Maschine Learning eine Vielzahl an Frameworks, welche nicht nur bei der Implementierung des Envs. helfen, sondern auch welche, die Funktionalität der Neuronalen Netzwerke bereitstellen.
Das in dieser Implementierung wurde sich für das Maschine Learning Framework PyTorch (`https://pytorch.org/`) entschieden. Dieses erlaubt auf einfacher Weise das Konzipieren der in 5.3.1 vorgestellten NNs.

6.1. Package Struktur

Für eine bessere Abtrennung der einzelnen Komponenten wurde sich für die folgende Projektstruktur entschieden.
Alle Ordner werden in src aufbewahrt. Dieser besitzt die vier Unterordner resources, snakeAI, statistic common.
In dem Ordner resources werden alle erzeugten Daten der Trainingsläufe und Testläufe sowie einige weitere images führ die Projektpräsentation aufbewahrt.
Der Ordner statistic beinhaltet die Files, in welche die Klasse zur Auswertung der Trainings- und Testdaten, aus welchen Statistiken generiert werden, aufbewahrt wird.
Der common Ordner enthält Elemente, welche von den Agenten, vom Snake-Environment und von den Statistik-Funkionen benutzt werden.
Der snakeAI Ordner beinhalte dient als Sammelordner zur Lagerung der gym_games, zu welches auch das, im Rahmen dieser Ausarbeitung, implementierte Spiel Snake gehört. Auch die Agenten sind in dem Ordner agent, in Unterordnern, aufbewahrt. Jeder Algorithmus erhält dabei seinen eigenen Unterordner. Zusätzlich befindet sich in diesem auch noch ein weiterer common Ordner, welche Element hält, die von allen

Agenten benutzt werden.

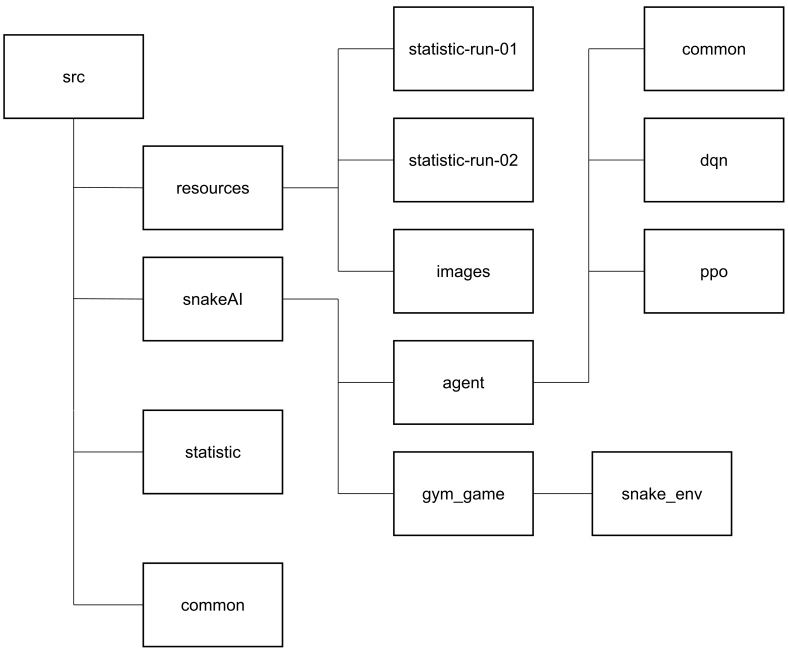

Abbildung 6.1.: Darstellung der Package Struktur.

6.2. Snake Environment

Zur Implementierung des Spiels Snake wurde das Framework gym von OpenAI genutzt (`https://gym.openai.com/`). Dieses bietet viele Methoden und Vorgaben in der Projektstruktur, welche das Implementieren erleichtern. So besteht das Snake Environment, welches im snake_env Package liegt (siehe 6.1), aus den wesentlichen Files:

- gui
- observation
- reward
- snake_env
- snake_game

6.2.1. Spiellogik

Die Spiellogik ist ist hauptsächlich im snake_game File implementiert. Anders als im Konzept, existiert keine Spiellogik-Klasse. Viel mehr wird die gesamte Verwaltung von der SnakeGame Klasse übernommen, welche der Game-Komponente gleichkommt. Die Gesamtheit aller Komponenten spannt dabei die Spiellogik auf. Diese

Abweichung des Konzepts hat sich angeboten, da so die Implementierung einer weiteren Klasse verhindert werden konnte. Dies minimiert den Implementierungsaufwand.

Die Game-Komponente, welche der SnakeGame Klasse entspricht, befindet sich im snake_game File.

```
def __init__(self, shape, has_gui):
  self.ground = np.zeros((shape[0], shape[1]), dtype=np.int8)
  pos = np.array((randint(0, shape[0] - 1), randint(0, shape[1] - 1))
      )
  self.p = Player(pos=pos, tail=[(pos[0], pos[1])], direction=randint
      (0, 3), id=1, c_s=1, c_h=2,
  inter_apple_steps=0, done=False)
  self.reward = Reward(self)
  self.shape = shape
  self.has_gui = has_gui
  self.step_counter = 0
  self.ground[pos[0], pos[1]] = self.p.c_h
  self.apple = self.make_apple()
  if has_gui:
  self.gui = GUI(self.shape)
```

Zur Erstellung der SnakeGame-Klasse werden die Spielfeldmaße (shape) und has_gui übertragen und in der Klasse gespeichert. Letzteres ist ein Boolean, welche die GUI ein oder ausschaltet. Als nächstes wird die Spieloberfläche (ground) generiert, welche durch ein Numpy Array (`https://numpy.org/`) implementiert wird. Die Position des Spieles (pos) wird daraufhin als nächstes zufallsbasiert bestimmt. Mit dieser Information kann die Datenhaltungsklasse des Players erzeugt werden (siehe 6.2.2). Der step_counter dient zur Bestimmung der gegangenen Schritte der Snake. Zum Schluss wird das Spielfeld mit der Position der Snake aktualisiert, ein Apfel wird mit der make_apple() Methode generiert, welche sogleich das Spielfeld anpasst und je nach dem has_gui Boolean wird ein GUI-Objekt (siehe 6.2.5) instanziiert.

Die SnakeGame-Klasse implementiert nach dem Konzept (siehe 5.2.1) die folgenden Methoden: action, observe, evaluate, reset, view.

Action

Die action Methode implementiert die Spiellogik und damit die Aktionsabarbeitung. Aus diesem Grund wird ihr die, vom Agenten bestimmte, Aktion in Form eines Integers übergeben. Daraufhin wird überprüft, ob die Snake bereits die maximale Schrit-

tanzahl überschnitten hat. Danach werden step_counter und p.inter_apple_steps inkrementiert und die Aktion wird umgesetzt. Um dies umzusetzen wird die direction im Player Objekt angepasst, entsprechend der Beschreibung im Konzept (siehe 5.2.1). Die directions und die actions sind dabei mit den Zahlen von eins bis vier bzw. von ein bis drei codiert.

```
def action(self, action):
  if self.p.inter_apple_steps >= self.max_snake_length:
    self.p.done = True
    return
  self.p.inter_apple_steps += 1
  self.step_counter += 1
  if action == 0:
    self.p.direction = (self.p.direction + 1) % 4
  elif action == 1:
    self.p.direction = (self.p.direction - 1) % 4
  else:
    pass
```

Nach der Manipulation der Snake direction wird ein Schritt gegangen. Dazu wird die p.pos entsprechende der neuen direction angepasst. Da zwar p.pos angepasst ist jedoch noch nicht die Liste p.tail, welche alle Snake-Glieder beinhaltet und das Spielfeld (ground) kann nun überprüft werden, ob die Aktion zum sofortigen Tod führt. Dabei werden noch nicht alle Todesmöglichkeiten berücksichtigt. Einzig die Tode, welche durch das Verlassen des Spielfeldes auftreten, da diese zu Exception führen würden.
Ein Beispiel dafür wäre, der Versuch das Spielfeld (ground) zu aktualisieren mit einer Position die außerhalb lägt.
Sollte dieser Fall eintreten, so wird p.done auf True gesetzt, was deinen terminalen Zustand ankündigt und die Methode würde beendet. Sollte diese Aktion nicht zum Tod führen, so wird der neue Snake-Kopf in die p.tail an vorderster Stelle eingefügt.

```
self.p.pos[self.p.direction % 2] += -1 if self.p.direction % 3 == 0
    else 1
if not all(0 <= self.p.pos[i] < self.ground.shape[i] for i in range
    (2)):
  self.p.done = True
  return
self.p.tail.insert(0, (self.p.pos[0], self.p.pos[1]))
```

Danach wird überprüft ob die Snake alle möglichen Äpfel gegessen hat, ob sie daher gewonnen hat. Sollte dies der Fall sein, so wird wieder p.done auf True gesetzt und die Methode terminiert.

```
if len(self.p.tail) == self.max_snake_length:
 self.p.done = True
 return
```

Hat die Snake nicht gewonnen haben, so wird eine Fallunterscheidung zwischen zwei Situationen durchgeführt. Der erste Fall tritt ein sofern die Snake einen Apfel gefressen hat, dann sind die Positionen des neuen Snake-Kopfes und des Apfels gleich. Daher wird die Matrix mit dem neuen Snake-Kopf aktualisiert und ein neuer Apfel wird mit der Methode make_apple generiert. Dieser wird gleich in das Spielfeld (ground) eingepflegt. Zum Schluss wird noch der p.inter_apple_steps, welcher die Schritte seit dem letzten Fressen aufaddiert auf null gesetzt, da ein Apfel gefressen wurde und reward.has_grown wird auf True gesetzt, da die Snake gewachsen ist. reward.has_grown dient dabei nur zur Bestimmung des Reward und st daher in der Reward-Klasse definiert.

```
if self.p.tail[0] == self.apple:
  self.ground[self.p.tail[0][0], self.p.tail[0][1]] = self.p.c_h
  self.apple = self.make_apple()
  self.p.inter_apple_steps = 0
  self.reward.has_grown = True
```

Ist die Snake jedoch nicht gewachsen, so triff der zweite Fall ein. Um die Illusion von Bewegung zu erzeugen, muss das letzte Schwanzstück der Snake entfernt werden, da ansonsten die Snake um ein Glied gewachsen wäre. Dieses wird sowohl vom Spielfeld als auch aus p.tail entfernt. Danach wird noch reward.has_grown auf False gesetzt, da die Snake nicht gewachsen ist.

```
else:
  self.ground[self.p.tail[-1][0], self.p.tail[-1][1]] = 0
  del self.p.tail[-1]
  self.reward.has_grown = False
```

Nach dieser Fallunterscheidung muss überprüft werden, ob die Snake nicht in sich selber gelaufen ist.

```
if len(self.p.tail) != len(set(self.p.tail)):
  self.p.done = True
  return
```

Zum Schluss der action Methode wir, sofern die Snake bis zu diesem Punkte nicht gewonnen oder verloren hat, über die p.tail List, welche alle die Positionen alles Schwanzglieder gespeichert hat, iteriert. Dabei wird das Spielfeld mit allen Gliedern erneut aktualisiert. Dabei wird jedem normalen Schwanzglied die Zahl eins in der

Matrix zugeordnet. Ausnahmen stellen der Kopf und das letzte Schwanzstück der Snake dar. Diese werden in der Matrix mit zwei bzw. -1 dargestellt. Die Werte für die Schwanzglieder sind in der Player-Klasse definiert (siehe 6.2.2).

```
if not self.p.done:
  for s in self.p.tail:
    self.ground[s[0], s[1]] = self.p.c_s
  self.ground[self.p.tail[-1][0], self.p.tail[-1][1]] = -1
  self.ground[self.p.tail[0][0], self.p.tail[0][1]] = self.p.c_h
```

Observe

Die Observe Methode ruft die make_obs Funktion im observation File auf. Zu diesem Zweck werden der Funktion die für die Obs benötigten Inforamtionen als Argumente übergeben. Näheres zur Obs in 6.2.3.

```
  def observe(self):
    return make_obs(self.p.id, self.p.pos, self.p.tail_pos, self.p.
        direction, self.ground, self.apple, self.p.inter_apple_steps)
```

Evaluate

Der evaluate Methode wird ein String übergeben, welche zur Auswahl der Reward-Funktion dient. Dies ist nötig, da im Rahmen der Optimierung B eine neue Reward-Funktion definiert wurde. Ansonsten ruft evaluate die entsprechende Reward-Funkion auf, welche in der Reward-Klasse definiert wurde (siehe 6.2.4).

```
def evaluate(self, reward_function="standard"):
  if reward_function == "standard":
    return self.reward.standard_reward
  elif reward_function == "optimized":
    return self.reward.optimized_reward
  else:
    raise ValueError("Wrong reward function.")
```

Reset

In der Reset Methode wird der bisherige Spielfortschritt zurückgesetzt. Dafür wird das Spielfeld mit Nullen überschrieben und die Player eigene reset Methode wird aufgerufen, welchedas Player-Objekt in seinen Ursprungszustand zurückversetzt. Danach verläuft die Methode analog zur Erstellung des SnakeGame-Objektes (siehe 6.2.1)

```
def reset_snake_game(self):
  self.ground.fill(0)
  pos = np.array((randint(0, self.shape[0] - 1), randint(0, self.
      shape[1] - 1)))
  self.p.player_reset(pos)
  self.step_counter = 0
  self.reward.has_grown = False
  self.ground[pos[0], pos[1]] = self.p.c_h
  self.apple = self.make_apple()
  if self.has_gui:
    self.gui.reset_GUI()
```

View

Die view Methode ruft ihrerseits die update_GUI Methode. Sie dient daher als Wrapper-Methode.

```
def view(self):
  if self.has_gui:
    self.gui.update_GUI(self.ground)
```

6.2.2. Player

Die Player-Klasse dient als Datenhaltungsklasse. Sie beinhaltet Informationen, wie z.B. die Position des Kopfes der Snake, die List tail, welche die Positionen der Schwanzglieder enthält, die direction der Snake, daher die Blickrichtung, die inter_apple_steps, also die Schritte aufaddierten Schritte sein dem letzten Fressen eines Apfels und der done Boolean, welcher angibt ob ein terminaler Zustand erreicht wurde. Neben diesen Informationen werden zusätzlich noch eine id und die Farbkonstanten c_s und c_h für color_snake und color_head.
Neben den Getter Methoden apple_count, last_tail_pos und snake_len, wurde noch eine Methode player_reset definiert, welche die oben aufgelisteten Informationen pos, tail, direction, inter_apple_steps und done zum Ursprungszustand zurücksetzt.

6.2.3. Observation

Die Obs wird im observation File erstellt, wobei für ihrer Erstellung keine Klasse verwendet wird. Vielmehr ruft die Hauptfunktion make_obs verschiedene Unterfunktionen auf, welche ebenfalls im observation File definiert sind.

Die Erklärung für die around_view (AV) und die Distanzbestimmung, für die scalar_obs (SO), befindet sich im Anhang der Implementierung (siehe B.1 und B.2). Neben den Distanzen existieren noch drei weitere Arten an Observations, welche für die SO benötigt werden. Zu diesen gehören die direction_obs, die compass_obs und die hunger_obs.
Die direction_obs wird mit Hilfe des One-Hot-Encodings wie folgt generiert:

```
def direction_obs(direction):
  obs = np.zeros(4, dtype=np.float64)
  obs[0 + direction] = 1
  return obs
```

Die Implementierung der compass_obs geschieht dabei wie im Konzept beschrieben (siehe 5.2.1). der compass_obs Unterfunktion wird die Position des Snake-Kopfes und eines Objektes übergeben. Für jede Zeile und Spalte wird dann überprüft, ob sich das Objekt im Vergleich zum Snake-Kopf höher, niedriger oder in der selben Zeile bzw. Spalte befindet. Die Information wird dann in ein Array eingepflegt.

```
def compass_obs(pos, obj):
  obs = np.zeros(6, dtype=np.float64)
  if obj is None:
    return obs
  obs[0] = 1 if pos[0] < obj[0] else 0
  obs[1] = 1 if pos[1] > obj[1] else 0
  obs[2] = 1 if pos[0] > obj[0] else 0
  obs[3] = 1 if pos[1] < obj[1] else 0
  obs[4] = 1 if pos[0] == obj[0] else 0
  obs[5] = 1 if pos[1] == obj[1] else 0
  return obs
```

Die hunger_obs wird ebenfalls wie im Konzept erwähnt berechnet und dann in einen array der Länge eins (Skalar) ausgegeben. Der scalare Wert wird in ein Array gebunden, damit er später besser in die scalar_obs (SO) eingebunden werden kann.

```
def hunger_obs(inter_apple_steps, size):
  obs = np.zeros(1, dtype=np.float64)
  obs[0] = 1 / (size - inter_apple_steps) if inter_apple_steps !=
      size else 2
  return obs
```

Zum Schluss werden alle Observations der SO zusammengefügt. Zu diesen gehören die distance_obs (distances) (siehe B.2), direction_obs (direction), apple_obs, tail_obs und die hunger_obs (hunger). Die apple_obs und tail_obs sind dabei compass_obs mit

den Objeten Apfel und letztes Schwanzglied der Snake. Die SO besitzt damit eine Länge von 41.

```
def make_obs(p_id, pos, tail_pos, direction, ground, food,
    iter_apple_counter):
  around_view = create_around_view(pos, p_id, ground)
  distances = create_distances(pos, ground)
  direction = direction_obs(direction)
  apple_obs = compass_obs(pos, food)
  tail_obs = compass_obs(pos, tail_pos)
  hunger = hunger_obs(iter_apple_counter, ground.size)
  scalar_obs = np.concatenate((distances, direction, apple_obs,
      hunger, tail_obs))
  return around_view, np.expand_dims(scalar_obs, axis=0)
```

6.2.4. Reward

Die Reward-Klasse, welche sich gleichnamigen File befindet, ist für die Bestimmung des Rewards zuständig. Zu diesem Zweck wird ihr die SnakeGame Instanz übergeben, damit sie über alle nötigen Spielinformationen besitzt. Sie verfügt über zwei Methoden, welche die Rewards bestimmen. Reward-Funktion eins (standard_reward) wird in Abschnitt Reward des Konzepts (siehe 5.2.1) näher erläutert.

```
@property
def standard_reward(self):
  if len(self.snakeGame.p.tail) == self.snakeGame.max_snake_length
      and self.snakeGame.p.done:
    return 100
  elif len(self.snakeGame.p.tail) != self.snakeGame.max_snake_length
      and self.snakeGame.p.done:
    return -10
  elif self.has_grown:
    return 2.5
  else:
    return -0.01
```

Reward-Funktion zwei (optimized_reward) wird im Abschnitt der Optimierungen (siehe 5.4.2) näher erläutert.

```
@property
def optimized_reward(self):
  len = self.snakeGame.max_snake_length
  r_distance = 10 / np.linalg.norm(self.snakeGame.p.pos - np.array(
      self.snakeGame.apple)) * (self.snakeGame.p.snake_len / len)
```

```
    r_timeout = (1 / len) * (self.snakeGame.p.inter_apple_steps / self.
        snakeGame.p.snake_len)
    return r_distance - r_timeout
```

6.2.5. GUI

Die GUI-Komponente befindet sich im gui File, in welchem die Klasse GUI definiert wird. Da die GUI unabhängig von der Spiellogik programmiert wurde, lässt sich diese je nach Wunsch ein oder ausschalten. Dies geschieht durch das Erzeugen oder nicht Erzeugen einer Instanz dieser Klasse.
Das GUI-Objekt bekommt bei der Initialisierung die Größe der Spieloberfläche übergeben. Die graphische Oberfläche wird dabei mit dem Framework Pygame `https://www.pygame.org/` implementiert. Damit dieses eine Spieloberfläche generiert, wird das Framework initialisiert und es wird ein display-Objekt generiert. Diesem wird die Länge und Breite der Spieloberfläche multipliziert mit der Kästchengröße, bei Initialisierung, übergeben. Dieses display-Objekt besteht aus einzelnen Vierecken in der Anzahl und Anordnung der Spielfeldgröße, welche zu Beginn schwarz eingefärbt (RGB = (0, 0, 0)) werden. Das Fenster besitzt daher die Form der Spieloberfläche (ground). Zum Schluss der Erzeugung des GUI-Objektes werden noch einige Einstellungen am Framework getätigt.

```
  def __init__(self, size):
    self.Particle = 60
    self.size = size
    pygame.init()
    self.screen = pygame.display.set_mode((self.Particle * self.size
        [0], self.Particle * self.size[1]))
    self.screen.fill((0, 0, 0))
    pygame.display.set_caption('Snake')
    pygame.PYGAME_HIDE_SUPPORT_PROMPT = 1
```

Die update_GUI methode ist die Hauptmethode dieser Klasse. Sie aktualisiert die GUI bei aufruft. Um dies zu tun, wird ihr das momentane Spielfeld übergeben. Danach wird das bisherige Fenster zurückgesetzt (schwarz eingefärbt) und überprüft, ob sich die Spielfeldgröße verändert hat. Sollte dies der Fall sein, so wird das display-Objekt mit der neuen Größe neue generiert. Ansonsten passiert nicht.

```
  def update_GUI(self, ground):
    self.reset_GUI()
    if self.size != ground.shape:
      self.size = ground.shape
```

```
    del self.screen
    self.screen = pygame.display.set_mode((self.Particle * self.size
        [1], self.Particle * self.size[0]))
```

Daraufhin wird die Funktionalität des Verschiebens und Schließens des Fensters implementiert. Sollte die GUI ordnungsgemäß geschlossen werden, so terminiert das Programm.

```
for event in pygame.event.get():
  if event.type == pygame.QUIT:
    pygame.quit()
    raise StopGameException()
```

Hiernach wird über jeden Eintrag des Spielfeldes iteriert und die entsprechenden Kästchen der GUI mit den neuen Werten angepasst. Dies geschieht mit der draw Methode. Sollten alle Kästchen der GUI aktualisiert sein, wird die update Methode von Pygame aufgerufen, welche die aktualisierte GUI nun darstellt.

Die draw Methode erzeugt die Vierecke und fügt diese an die für sie zugeordneten Positionen.

```
def draw(self, pos, color):
  Cords = [pos[0] * self.Particle, pos[1] * self.Particle]
  pygame.draw.rect(self.screen, color, (Cords[0], Cords[1], self.
      Particle, self.Particle), 0)
```

Des Weiteren existiert noch eine reset_GUI Methode, welche alle Vierecks des Fensters schwarz einfärbt.

```
def reset_GUI(self):
  self.screen.fill((0, 0, 0))
```

6.2.6. Wrapper

Die SnakeEnv Klasse erbt von der gym.Env Klasse und stellt damit einen Wrapper dar. Dadurch wird eine feste Methodenstruktur vorgegeben, welche eine standardisierte Schnittstelle erzeugt. Die SnakeEnv Klasse befindet sich im snake_env File und bekommt bei Erzeugung einer Instanz die Spielfeldgröße und den has_gui Boolean übergeben, welche die GUI entweder ein- oder ausschaltet. Diese Inforationen werden gespeichert und mit diesen die SnakeGame Instanz erzeugt (siehe 6.2.1).

```
def __init__(self, shape=(8, 8), has_gui=False):
  self.shape = shape
  self.has_gui = has_gui
```

```
    self.game = SnakeGame(self.shape, self.has_gui)
```

Die SnakeEnv Klasse implementiert die Methoden step, reset, render, close.

Die step Methode bekommt die action als Integer und die reward_function als String übergeben. Sie ruft daraufhin die action Methode der SnakeGame Klasse auf. Sobald diese terminiert ist wird die observe und evaluate Methode (siehe 6.2.1 und 6.2.1) der SnakeGame Klasse aufgerufen, sowie der is_done Getter, welcher ebenfalls in der SnakeGame Klasse definiert wurde. Die step Methode returned die von den Methoden zurückgegebenen Daten, inklusive des win Boolean, welche bei einem Sieg True und bei Verlust False zurückgibt.

```
def step(self, action, reward_function="standard"):
  self.game.action(action=action)
  around_view, scalar_obs = self.game.observe()
  reward = self.game.evaluate(reward_function=reward_function)
  done = self.game.is_done
  return around_view, scalar_obs, reward, done, self.game.
      max_snake_length == self.game.p.apple_count + 1
```

Die reset Methode ruft die reset_snake_game und observe Methoden der SnakeGame Klasse auf (siehe 6.2.1 und 6.2.1) und gibt die neue initiale Obs zurück.

```
def reset(self):
  self.game.reset_snake_game()
  around_view, scalar_obs = self.game.observe()
  return around_view, scalar_obs
```

Die render Methode ruft nur die view Methode der SnakeGame Klasse auf (siehe 6.2.1)

```
def render(self, close=False):
  self.game.view()
```

Die close Methode terminiert das Programm.

```
def close(self):
  raise StopGameException()
```

6.3. Agenten

Die Agenten befinden sich im agent Unterordner, wie in Abbildung 6.1 dargestellt.

6.3.1. AV-Network

Das AV-Network stellt die Netzstruktur dar, welche die AV (around_view) verarbeitet. Um dies bewerkstelligen zu können, wird das Netz mit Hilfe des PyTorch Frameworks durchgeführt. Dabei wird so verfahren, wie im Konzept (siehe 5.3.1) gefordert. Eine geneuere Darstellung der Implementierung des AV-Networks findet sich im Anhang (siehe B.3).

6.3.2. DQN

Der DQN Algorithmus, aus welchen die DQN Agenten hervorgehen, befindet sich im gleichnamigen Unterordner. Er beinhaltet die folgenden Files:
Das dqn File ist das Hauptfile. In ihm wird die Agenten Klasse definiert, welche die act und learn Methoden implementiert. Die Agenten Klasse ist dabei als DQN-Komponente zu interpretieren.
Das memoryDQN File implementiert den Replay Buffer bzw. das Memory, in welchem die gesammelten Erfahrungen gespeichert werden.
Das q_net File beinhaltet die Netzstruktur des Q-Networks.
Das dqn_train File implementiert die Trainingsroutine, welche den Agenten trainiert, die Trainingsdaten generiert und diese speichert.
Das dqn_test File implementiert die Testroutine, welche die Testdaten generiert und abspeichert.

Q-Network

Das Q-Network ist im q_net File definiert. Es besteht aus dem AV-Network (AV_NET) (siehe 6.3.1) und dem Q-Network-Tail (Q-net). Des Weiteren ist in der QNetwork Klasse noch der für dieses Network bestimmte Optimizer (OPTIMIZER), welche das Gradientenverfahren durchführt.

```
class QNetwork(nn.Module):
  def __init__(self, OUTPUT, LR, SCALAR_INPUT=41, DEVICE="cuda:0"):
    super(QNetwork, self).__init__()
    T.set_default_dtype(T.float64)
    T.manual_seed(10)
    self.AV_NET = AV_NET()
    self.Q_net = nn.Sequential(
      nn.Linear(128 + SCALAR_INPUT, 64),
      nn.ReLU(),
      nn.Linear(64, OUTPUT)
    )
    self.OPTIMIZER = Adam(self.parameters(), lr=LR)
```

```
        self.DEVICE = DEVICE
        self.to(self.DEVICE)

    def forward(self, av, scalar_obs):
        av_out = self.AV_NET(av)
        cat = T.cat((av_out, scalar_obs), dim=-1)
        q_values = self.Q_net(cat)
        return q_values
```

Das Q-net besteht dabei aus den im Konzept erwähnten Elementen (siehe 5.3.1). Die forward Methode leitet die AV (av) durch das AV_NET (siehe B.3). Danach wird das Output des AV_Net mit der SO (scalar_obs) verbunden und durch das Q_net geleitet. Das Ergebnis wird zurückgeliefert.

Memory

Das Memory oder auch Replay Buffer genannt, befindet sich im memoryDQN File. Dort wird die Klasse Memory definiert, welche bei ihrer Erstellung die maximale Größe des Memory, die Dimension der AV (6x13x13) und der SO (41), die Batch-Size und das Device (CPU oder GPU). Dabei besteht das Memory aus eine reiche von tensoren, welche die vom Spiel und Agenten generierten Daten mittels einer zusätzlichen Dimension speichert. Sollte die AV also die Form (6x13x13) besitzen, so besitzt der Speicher-Tensor (AV) die Form (MEM_SIZEx6x13x13).
Um eine Erfahrung bestehend aus (AV, SO, action, reward, is_terminal, AV-next, SO-next) in das Memory einzupflegen, wird die store Methode aufgerufen. Diese bestimmt einen Index, der angibt an welche Stelle im Memory die Erfahrung gespeichert werden soll. Sollte das Memory voll sein, so wird der Index am Anfang des Memory on neuen beginnen und alte Erfahrungen überschreiben.

Mit der get_data Methode wird aus dem gesamten Memory ein Mini-Batch entnommen und zurückgegeben.

```
    def get_data(self, returned_data=None):
        max_mem = min(self.counter, self.MEM_SIZE)
        size = self.BATCH_SIZE if not bool(returned_data) else
            returned_data
        batch = np.random.choice(max_mem, size, replace=False)

        batch_index = T.arange(size, dtype=T.long, device=self.DEVICE)
        return self.AV[batch], self.SCALAR_OBS[batch], self.ACTION[batch],
            self.REWARD[batch], \
```

```
        self.IS_TERMINAL[batch], self.AV_[batch], self.SCALAR_OBS_[
            batch], batch_index
```

DQN-Agent

Der DQN-Agent wird im dqn File definiert. Die Agenten Klasse erhält bei Erstellung eine Reihe an Hyperparameter, wie z.B. die Lernrate, die Anzahl der Aktionen, GAMMA, die Batch-Size, Epsilon Start (normal bei 1.0), Epsilon Decrement, und die Memory-Größe. Diese Werte werden gespeichert. Des Weiteren wird das Memory und Q-Network initialisiert.

textbfAct Methoden
In der Agenten Klasse werden zwei act Methoden definiert. Die erste namens act, implementiert die im Konzept dargestellte Aktionsbestimmung für den Trainingsbetrieb (siehe 5.3.2). Damit PyTorch die AV und SO, welche bei Übergabe Numpy-Arrays sind, verwenden kann, müssen aus diesen erst Tensoren generiert werden. Danach wird der Zufallswert bestimmt und sollte diese kleiner als Epsilon sein, so wird die Aktion vom Q-Network bestimmt. Ansonsten wird eine Zufallsaktion ausgewählt. AM ende werden die Tensoren der AV und der SO zusammen mit der Aktion zurückgegeben. Die Rückgabe der ersteren beiden dient dabei der Speichereffizienz, damit nicht unnötig viele Duplikate erstellt werden.

```
def act(self, av, scalar_obs):
  av = T.from_numpy(av).to(self.Q_NET.DEVICE)
  scalar_obs = T.from_numpy(scalar_obs).to(self.Q_NET.DEVICE)
  if np.random.random() > self.EPSILON:
    with T.no_grad():
      actions = self.Q_NET(av, scalar_obs)
      action = T.argmax(actions).item()
  else:
    action = np.random.choice(self.ACTION_SPACE)
  return av, scalar_obs, action
```

Die zweite act Methode namens act_test bestimmt die Aktion ausschließlich über das Q-Network.

```
@T.no_grad()
def act_test(self, av, scalar_obs):
  av = T.from_numpy(av).to(self.Q_NET.DEVICE)
  scalar_obs = T.from_numpy(scalar_obs).to(self.Q_NET.DEVICE)
  q_values = self.Q_NET(av, scalar_obs)
```

```
    return av, scalar_obs, T.argmax(q_values).item()
```

Es wird daher auf alle Verfahrensweisen mit Epsilon verzichtet.

Learn Methode

Die learn Methode überprüft bei Aufruf als erstes, ob sich aus dem Memory genügend Daten für einen Mini-Batch entnehmen kann. Sollte dies nicht der Fall sein, so terminiert die Methode.

```
def learn(self):
  if self.MEM.counter < self.BATCH_SIZE:
    return
```

Danach wird die get_data Methode des Memory aufgerufen. Diese liefert einen Mini-Batch, welche im Folgenden für das lernen benutzt wird. Danach werden die Schritt, entsprechend der Beschreibung im Konzept (siehe 5.3.2), durchgeführt.
Zu Beginn werden die Q-Values der gespeicherten Aktion für die gegenwärtigen Zustande bestimmt. Danach folgen Q-Values aller Aktionen der Nachfolgezustände.

```
av, scalar_obs, actions, rewards, is_terminal, av_, scalar_obs_,
    batch_index = self.MEM.get_data()
q_eval = self.Q_NET(av, scalar_obs)[batch_index, actions]
q_next = self.Q_NET(av_, scalar_obs_)
```

Daraufhin wird wie in Abschnitt 2.10 $r(s,a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$ bestimmt. Dies wurde genauer in Abschnitt 5.3.2 thematisiert.

```
q_next[is_terminal] = 0.0
q_target = rewards + self.GAMMA * T.max(q_next, dim=1)[0]
```

Zum Schluss wird der Loss des DQN bestimmt und, mit Hilfe des Optimizer, das Q-Net aktualisiert.

```
loss = self.LOSS(q_target, q_eval)
self.Q_NET.OPTIMIZER.zero_grad()
loss.backward()
self.Q_NET.OPTIMIZER.step()
```

Zusätzlich wird noch Epsilon verringert, um die Anzahl an Zufallsaktionen zu senken.

```
self.EPSILON = self.EPSILON - self.EPS_DEC if self.EPSILON > self.
    EPS_MIN else self.EPS_MIN
```

6.3.3. PPO

Der PPO Algorithmus, aus welchen die PPO Agenten hervorgehen, befindet sich im gleichnamigen Unterordner. Er beinhaltet die folgenden Files:
Das ppo File stellt das Hauptfile dar. In diesem wird die Agenten Klasse definiert, welche die learn Methoden implementiert. Die Agenten Klasse ist dabei als PPO-Komponente zu interpretieren (siehe 5.3.3).
Das memoryDQN File implementiert den Replay Buffer bzw. das Memory, in welchem die gesammelten Erfahrungen gespeichert werden.
Das actor File beinhaltet die Netzstruktur des Actors.
Das critic File beinhaltet die Netzstruktur des Critics.
Das actor_critic File definiert eine ActorCritic Klasse, welche Actor-NN und Critic-NN miteinander verbindet. Dies sorgt für eine einfachere Handhabung. Weiterhin sind die act Methoden in der ActorCritic Klasse definiert. Das ppo_train File implementiert die Trainingsroutine, welche den Agenten trainiert, die Trainingsdaten generiert und diese speichert.
Das ppo_test File implementiert die Testroutine, welche die Testdaten generiert und abspeichert.

Actor

Das Actor File befindet sich im ppo Unterordner und definiert eine neue Klasse ActorNetwork. Dieses erbt, wie bei Q-Network (siehe 6.3.2), von der Module Klasse des PyTorch Frameworks. Damit lässt sich das ActorNetwork als NN-Baustein bedachten.
Bei der Erzeugung einer Instanz der Klasse wird dieser die Anzahl an Outputs und die Dimension der SO (scalar_obs) übergeben. Neben dem AV-Network (siehe 6.3.1) wird noch der Actor-Tail definiert. Dabei handelt es sich um das NN, welches die Ausgabe vom AV-Network mit der SO verbindet und aus diesem Ergebnis eine Wahrscheinlichkeitsverteilung über alle Aktionen generiert (Output des Actors).

```
class ActorNetwork(nn.Module):
  def __init__(self, OUTPUT=3, SCALAR_IN=41):
  super(ActorNetwork, self).__init__()
  self.AV_NET = AV_NET()

  self.ACTOR_TAIL = nn.Sequential(
    nn.Linear(128 + SCALAR_IN, 64),
    nn.ReLU(),
    nn.Linear(64, OUTPUT),
    nn.Softmax(dim=-1)
  )
```

```
    def forward(self, AV, SCALAR_OBS):
      av_out = self.AV_NET(AV)
      cat = T.cat((av_out, SCALAR_OBS), dim=-1)
      actor_out = self.ACTOR_TAIL(cat)
      return actor_out
```

Die forward Methode ist dabei gleich der in Abschnitt 6.3.2.

Critic

Das Critic File befindet sich im ppo Unterordner und definiert eine neue Klasse CriticNetwork. Dieses erbt, wie bei Q-Network (siehe 6.3.2), von der Module Klasse des PyTorch Frameworks. Damit lässt sich das CriticNetwork als NN-Baustein bedachten.
Bei der Erzeugung einer Instanz der Klasse wird dieser nur die Dimension der SO (scalar_obs) übergeben. Neben dem AV-Network (siehe 6.3.1) wird noch der Critic-Tail definiert, mit welchem später den Value-Wert bestimmen wird. Die forward Methode leitet die AV durch das AV-Network und verbindet das resultierende Ergebnis mit der SO. Danach wird dieses durch das Critic-Tail-NN geleitet. Als Ergebnis wird ein eindimensionaler Vektor mit einem Eintrag (Skalar) zurückgegeben.

```
class CriticNetwork(nn.Module):
  def __init__(self, SCALAR_IN=41):
    super(CriticNetwork, self).__init__()
    self.AV_NET = AV_NET()

    self.CRITIC_TAIL = nn.Sequential(
      nn.Linear(128 + SCALAR_IN, 64),
      nn.ReLU(),
      nn.Linear(64, 1),
    )

  def forward(self, av, scalar_obs):
    av_out = self.AV_NET(av)
    cat = T.cat((av_out, scalar_obs), dim=-1)
    critic_out = self.CRITIC_TAIL(cat)
    return critic_out
```

ActorCritic

Die ActorCritic Klasse ist im actor_critic File definiert und verbindet Actor und Critic miteinander. Bei Initialisierung wird der Instanz die Anzahl an Aktionen , die Lernraten on Actor und Critic und das Device (cpu oder GPU) übergeben.
Wie auch schon die ActorNetwork und CriticNetwork Klassen erbt die ActorCritc Klasse von der Module Klasse des PyTorch Frameworks. Sie kann daher aus ActorNetwork und CriticNetwork ein gesamt Network aufspannen, welches alle Parameter der beiden NN besitzt. Daher ist auch nicht verwunderlich, dass die ActorCritic Klasse sowohl das ActorNetwork als auch das CriticNetwork implementiert. Des Weiteren wird ein Optimizer definiert, welche die Parameter von Actor-NN und Critic-NN erhält. Dies ist möglich, da Actor und Critic mit dem gleichen Loss trainiert werden.
Die ActorCritic Klasse implementiert die act Methode, daher die Aktionsbestimmung.

Act Methoden

Die PPO Agenten verfügen wie die DQN Agenten über zwei act Methoden. Die erste namens act ist für die Aktionsauswahl während des Trainings. Sie nimmt die AV und SO entgegen und wandelt diese zu PyTorch Tensoren um. Danach wird die AV und die SO dem Actor übergeben, welcher die Wahrscheinlichkeitsverteilung über alle Aktionen zurückgibt. Diese wird auch als Policy bezeichnet.

```
@T.no_grad()
def act(self, av, scalar_obs):
  av = T.from_numpy(av).to(self.DEVICE)
  scalar_obs = T.from_numpy(scalar_obs).to(self.DEVICE)
  policy = self.ACTOR(av, scalar_obs)
  dist = Categorical(policy)
  action = dist.sample()
  return av, scalar_obs, action.item(), dist.log_prob(action)
```

Daraufhin wird eine Aktion entsprechend der Wahrscheinlichkeitsverteilung generiert und zusammen mit ihrer logarithmierten Wahrscheinlichkeit und den Tensor AV und SO zurückgegeben.

Die zweite act Methode namens act_test verzichtet wieder auf den Zufallsprozess bei der Aktionsbestimmung. Nachdem auch hier wieder die Tensoren der AV und SO gebildet worden sind, werden diese durch das Actor-NN geleitet.

```
@T.no_grad()
def act_test(self, av, scalar_obs):
```

```
av = T.from_numpy(av).to(self.DEVICE)
scalar_obs = T.from_numpy(scalar_obs).to(self.DEVICE)
policy = self.ACTOR(av, scalar_obs)
return av, scalar_obs, T.argmax(policy).item()
```

Am Ende wird die Aktion ausgewählt, welche die größte Wahrscheinlichkeit vorweist.

Evaluate Methode

Neben den act Methoden verfügt die ActorCritic Klasse noch über eine evaluate Methode, welche die logarithmierte Wahrscheinlichkeit einer Aktion unter einer anderen bzw. älteren Policy (siehe 2.2.1) bestimmten kann. Neben dieser bestimmt sie auch noch den Value-Wert des Critics und ermittelt die Entropy der Policy. All diese Werte sind essentiell für die learn Methode.

```
def evaluate(self, av, scalar_obs, action):
policy, value = self.forward(av, scalar_obs)
dist = Categorical(policy)

action_probs = dist.log_prob(action)
dist_entropy = dist.entropy()
return action_probs, T.squeeze(value), dist_entropy
```

Memory

Das Memory oder auch Replay Buffer genannt, befindet sich im memoryPPO File. Dort wird die Klasse Memory definiert, welche bei ihrer Erstellung die maximale Größe des Memory, die Dimension der AV (6x13x13) und der SO (41), die Batch-Size und das Device (CPU oder GPU). Dabei besteht das Memory aus eine reiche von Tensoren, welche die vom Spiel und Agenten generierten Daten mittels einer zusätzlichen Dimension speichert (siehe 6.3.2). Die Rewards und Terminals (is_terminal) werden jedoch in Listen gespeichert, da dies das spätere Diskontieren (Abzinsen) erleichtert.
Um eine Erfahrung bestehend aus (AV, SO, action, log_porb, reward, is_terminal,) in das Memory einzupflegen, wird die store Methode aufgerufen. Diese fügt die Erfahrung immer an vorderster Stelle ein. Dies ist möglich, da immer pro Spielepisode gelernt. Die neuen Erfahrungen überschreiben steht die alten, da diese nicht aus den Tensoren gelöscht werden. Viel mehr wird sich mit Hilfe eines Counters gemerkt, bis zu welchem Index im Tensor die jetzige Erfahrung reicht. Nur Erfahrungen bis zu diesem Index werden zurückgegeben, sobald die get_data Methode aufgerufen wird. Die Reward- und Is_terminal-Liste werden nach jedem Lernen gelöscht.
Mit der get_data Methode werden die gesamten Erfahrungen der letzten Spielepisode

zurückgegeben.

```
def get_data(self):
  return self.AV[:self.counter, ...], \
         self.SCALAR_OBS[:self.counter, ...], \
         self.ACTION[:self.counter, ...], \
         self.LOG_PROBABILITY[:self.counter, ...], \
         self.REWARD, \
         self.IS_TERMINAL
```

PPO

Die Agent Klasse wird im ppo File definiert und nimmt bei ihrer Erstellung die Lernraten von Actor und Critic, Gamma, K_Epochs und Epsilon-Clip (siehe 2.3.3) und GPU entgegen. Letzteres steuert das Device, sodass wenn GPU den Wert True annimmt, wird die Grafikkarte verwendet, sofern diese dafür geeignet ist. Im PPO wird der Memory und zwei ActorCritic Networks erstellt. Diese stellen die alte und neue Policy dar. Während mit der alten Policy immer die Trainingsdaten generiert werden, wird die neue Policy mit diesen Daten trainieren und zu einem späteren Zeitpunkt die alte Policy aktualisieren. Mit diesem Verfahren lässt sich die Datenerhebung auf mehreren Maschinen gleichzeitig durchführen, um mehr Daten zu erhalten. Dies spielt in dieser Implementierung jedoch keine Rolle.

Learn Methode

Sollte der Memory nicht mehr als 64 Erfahrungen besitzen, so wird die learn Methode terminiert. Ansonsten werden die Daten einer bzw. zu Beginn mehrerer Spielepisoden aus dem Memory mit der get_data Methode geliefert.

```
  def learn(self):
  if self.MEM.counter < 64:
    return

  old_av, old_scalar, old_action, probs_old, reward_list, dones_list
     = self.MEM.get_data()
```

Danach werden die Rewards diskontiert (abgezinst) and damit die Returns generiert. Dies geschieht mit der Untermethode generate_rewards. Diese implementiert die Funktionalität, welche im Abschnitt Return 2.3.2 dargestellt wurde. Am Ende wird eine neue Liste mit diskontierten Rewards (Returns) zurückgegeben.

```
def generate_reward(self, rewards_in, terminals_in):
  rewards = []
  discounted_reward = 0
```

```
    for reward, is_terminal in zip(reversed(rewards_in), reversed(
        terminals_in)):
      if is_terminal:
        discounted_reward = 0
      discounted_reward = reward + self.GAMMA * discounted_reward
      rewards.insert(0, discounted_reward)
    return rewards
```

Daraufhin wird diese Liste in einen PyTorch Tensor überführt und im Anschluss noch normalisiert, um ein stetigeres Lernen zu ermöglichen.

```
rewards = self.generate_reward(reward_list, dones_list)
rewards = T.tensor(rewards, dtype=T.float64, device=self.DEVICE)
rewards = (rewards - rewards.mean()) / (rewards.std() + 1e-7)
```

Im Anschluss wird die folgende Prozedur K_Epochs mal wiederholt.
Die Erfahrungen einer bzw. mehrerer Spielepisode/n werden zufallsbasiert durchmischt, um ein stabileres Lernen zu ermöglichen. Dies ist zum jetzigen Zeitpunkt möglich, dar alle Rewards diskontiert wurden. Danach wird die evaluate Methode mit den Erfahrungen aufgerufen. Wie bereits im Abschnitt der ActorCritic Klasse erwähnt (siehe 6.3.3), bestimmt diese die neuen logarithmierten Wahrscheinlichkeiten (log_probs) aller Aktionen. Mit diesen und den gespeicherten alten (old_log_probs) wird daraufhin die ratio gebildet.
Darauf folgende wird mit Hilfe der Values, welche die evaluate Methode ebenfalls zurückgibt, die Advantages aller Erfahrungen erstellt (siehe 2.3.2).

```
batch = np.random.randint(self.MEM.counter, size=(self.K_EPOCHS, self
    .MEM.counter))

for j in range(self.K_EPOCHS):
  old_av_b = old_av[batch[j, ...]]
  old_scalar_b = old_scalar[batch[j, ...]]
  old_action_b = old_action[batch[j, ...]]
  probs_old_b = probs_old[batch[j, ...]]
  rewards_b = rewards[batch[j, ...]]

  probs, state_values, dist_entropy = self.POLICY.evaluate(old_av_b,
      old_scalar_b, old_action_b)

  ratios = T.exp(probs - probs_old_b)

  advantages = rewards_b - state_values.detach()
```

Mit diesen Werten ist es nun möglich die Surrogate Losses zu bestimmten (siehe 2.3.2) und mit diesen den Actor-Loss bzw. Clip-PPO-Loss (siege 2.2) zu bestimmen.

```
surr1 = ratios * advantages
surr2 = T.clamp(ratios, 1 - self.EPS_CLIP, 1 + self.EPS_CLIP) *
    advantages

loss_actor = -(T.min(surr1, surr2) + dist_entropy * self.
    ENT_COEFFICIENT).mean()
```

In diesem ist zugleich noch der Entropy-Loss mit integriert (siehe 2.3.2). neben dem Actor-loss fehlt noch der Critic-Loss, welcher sogleich mit dem MSE (Mean Squared Error) bestimmt wird. Dabei wird der mittlere quadratische Fehler zwischen den Returns und den Values des Critics ermittelt.

```
loss_critic = self.CRITIC_COEFFICIENT * self.LOSS(rewards_b,
    state_values)
```

Die beiden Losses werden zusammenaddiert und dann wird das Actor-NN und Critic-NN mit Hilfe des Optimizers aktualisiert.

```
loss = loss_actor + loss_critic
            self.POLICY.OPTIMIZER.zero_grad()
loss.backward()
self.POLICY.OPTIMIZER.step()
```

Zu Schluss wird, nachdem die Prozedur K_Epochs-mal durchgeführt wurde, die alte Policy mit der neuen aktualisiert und das Memory wird geleert.

```
self.OLD_POLICY.load_state_dict(self.POLICY.state_dict())
T.cuda.empty_cache()
self.MEM.clear_memory()
```

6.4. Main Methoden

Die Hauptmethoden befinden sich in den jeweiligen Agenten Unterordnern. Es existieren für jeden Agenten zwei Main Methoden. Die erste implementiert jeweils die Trainingsroutine und die zweite die Testroutine. Aufgrund der Standardisierten Schnittstelen, lassen sich die Hauptmethoden beider Agenten gemeinsam erklären, da sie sich nicht besonders voneinander unterscheiden.

6.4.1. Train Methode

Die Train Methode des jeweiligen Agenten bekommt die Hyperparameter übergeben, welche in den Abschnitten C.2 und C.1 beschrieben sind. Danach wird der Agent und das Env zusammen mit den Datenhaltungslisten (siehe 5.5.1) erstellt. Neben diesen werden auch noch einige Zeiten gespeichert, um später den Fortschritt des Lernens darstellen und ein Scheduler, für die Optimierung B (siehe 5.4.2).

```
try:
  start_time = time_ns()
  scores, apples, wins, dtime, steps_list, dq = [], [], [], [], [],
     deque(maxlen=100)
  agent = Agent(LR_ACTOR=LR_ACTOR, LR_CRITIC=LR_CRITIC, GAMMA=GAMMA,
     K_EPOCHS=K_EPOCHS, EPS_CLIP=EPS_CLIP,
  GPU=GPU)
  game = SnakeEnv(BOARD_SIZE, False)
  scheduler = ExponentialLR(agent.POLICY.OPTIMIZER, 0.95, verbose=
     True)
  iter_time = time_ns()
```

Danach wird die Lernprozedur durchgeführt. Zu Beginn wird eine Obs bestehend aus AV (around_view) und SO (scalar_obs) generiert. Diese Obs wird dem Agenten übergeben, welcher eine Aktion und die weiteren benötigten Daten für den jeweiligen Algorithmus zurückgibt, beim PPO z.B. die logarithmierte Wahrscheinlichkeit für die ausgewählte Aktion (log_probability). Danach wird diese Aktion im Env ausgeführt. Dabei wird mit der RUN_TYPE Variable (siehe C.1) gesteuert, welche Reward Funktion verwendet wird, siehe Optimierung A im Abschnitt 5.4.1.

```
for i in range(1, N_ITERATIONS + 1):
  score = 0
  av, scalar_obs = game.reset(get_random_game_size() if
     RAND_GAME_SIZE else None)
  while not game.has_ended:
    av, scalar_obs, action, log_probability = agent.OLD_POLICY.act(av
       , scalar_obs)

    av_, scalac_obs_, reward, is_terminal, won = game.step(action,
       RUN_TYPE)

    agent.MEM.store(av, scalar_obs, action, log_probability, reward,
       is_terminal)
    score += reward
```

```
        av = av_
        scalar_obs = scalac_obs_

    agent.learn()

    scores.append(score)
    wins.append(won)
    apples.append(game.apple_count)
    dtime.append(datetime.now().strftime("%H:%M:%S"))
    steps_list.append(game.game.step_counter)
```

Mit Rückgabe der neuen AV und SO sowie dem Reward, dem terminal Boolean (is_terminal) und dem won Boolean, kann das Memory mit den entstandenen Erfahrungen aktualisiert werden. Daraufhin wird die score Variable mit dem neuen Reward aktualisiert und die alten Obs wird durch die neue Obs ersetzt, damit das Spiel voranschreitet.
Sollte die Spielepisode terminierten so wird die learn Methode des Agent aufgerufen und die generierten Episodendaten werden in die betreffenden Listen eingefügt (siehe 5.5.1). Beim DQN wird die learn Methode nach jedem fünften Schritt aufgerufen. Ansonsten ist das Prozedere analog zum DQN.
Um einen besseren Überblick über das Lerngeschehen zu erhalten wird mit Hilfe eine print_progress Methode der Fortschritt graphisch in der Konsole dargestellt. Diese Darstellung beinhaltet die durchschnittliche Performance und den durchschnittlichen Score der letzten fünf Spiele zuzüglich der vergangenen und der noch zu erwartenden Lernzeit.

```
t = time_ns()
dq.append(((t - iter_time) / 1_000) * (N_ITERATIONS - i))
time_step = str(timedelta(microseconds=(median(dq)))).split('.')[0]
passed_time = str(timedelta(microseconds=(t - start_time) / 1_000)).
    split('.')[0]
iter_time = time_ns()
suffix_1 = f"P.Time: {passed_time} | R.Time: {time_step}"
suffix_2 = f" | A_avg: {round(mean(apples[-5:]), 2)} | S_avg: {round(
    mean(scores[-5:]), 2)}"
print_progress(i, N_ITERATIONS, suffix=suffix_1 + suffix_2)
```

Damit der Lernprozess wie im Konzept dargestellt bei einer Siegrate von 60% stoppt, wird die Siegrate der letzten 100 Spiele ermittelt und sollte diese über oder gleich 60% wird der Lernprozess terminiert und die Trainingsdaten werden gespeichert.

```
if sum(wins[-100:]) / 100 > 0.6:
```

```
    save("PPO", AGENT_NUMBER, STATISTIC_RUN_NUMBER, "train", RUN_TYPE,
        RAND_GAME_SIZE, agent, dtime, steps_list, apples, scores, wins)
        return
```

Wird die Optimierung B (siehe 5.4.2) verwendet, so wird alle 100 Spielepisoden die Steigung der Performance der letzen 100 Episoden bzw. Epochs ermittelt. Sollte diese nicht größer oder gleich null sein, so wird die Lernrate mit 0.95 multipliziert und damit gesenkt.

```
if i % 100 == 0 and SCHEDULED_LR:
  m, b, _, _, _ = linregress(list(range(100)), apples[-100:])
  if m <= 0:
    scheduler.step()
```

Sind alle 30.000 Epochs abgeschlossen wird das NN und die Trainingsdaten gespeichert und die Methode terminiert.
Sollte man sich dazu entscheiden den Trainingsprozess vorzeitig abbrechen zu wollten so wird man vom System gefragt, ob die Daten und das NN gespeichert werden sollen. Mit der Betätigung von "y"werden NN und Daten gespeichert mit "n"werden diese nicht gespeichert. Danach terminiert die Methode.

```
except (KeyboardInterrupt, StopGameException):
  repeat = True
  MODEL_DIR_PATH = str(Path(__file__).parent.parent.parent.parent) +
      f"\\resources\\{RUN_TYPE}-run-0{STATISTIC_RUN_NUMBER}"
  while repeat:
    answer = input(f"\nDo you want to save the files in a new Folder
        at {MODEL_DIR_PATH}? y/n \n")
    if answer == 'y':
      save("PPO", AGENT_NUMBER, STATISTIC_RUN_NUMBER, "train",
          RUN_TYPE, RAND_GAME_SIZE, agent, dtime,
      steps_list, apples, scores, wins)
      repeat = False
    elif answer == 'n':
      repeat = False
    else:
      print("Wrong input!")
```

6.4.2. Test Methoden

Die Test Methoden sind bis auf wenige Ausnahmen mit den Train Methoden übereinstimmend. Es werden in diesem Abschnitt daher nur die Unterschiede aufgezeigt. Bei Start einer test Methode werden die Hyperparameter verwendet, welche in der Anleitung

(siehe C.3) dargestellt sind. Zu diesen gehört auch der MODEL_PATH, welche den Speicherort angibt, unter welchem sich das NN befindet. Mit diesem wird dann der Test durchgeführt. Alle Element des Lernens sind aus der test Methode entfernt, zu diesen zählen der Scheduler, und das Aufrufen der learn Methode. Hinzukommt die Prozedur, um die Spielfeldgröße zu ändern, für die Bestimmung der Robustheit (siehe 5.5.2 und 3.5).

```
rand_size = None
if RAND_GAME_SIZE:
  rand_size = get_random_game_size()
av, scalar_obs = game.reset(new_shape=rand_size)
while not game.has_ended:
  av, scalar_obs, action = agent.OLD_POLICY.act_test(av, scalar_obs)
  ...
```

Des Weiteren wird für die Aktionsbestimmung nun die act_test Methode verwendet und es kommt der Funktionalität der graphischen Umsetzung hinzu, indem die render Methode des Env aufgerufen wird, sofern die GUI in den Hyperparametern nicht ausgeschaltet worden ist.

```
...
av_, scalar_obs_, reward, done, won = game.step(action)
score += reward
if game.has_gui:
  game.render()
...
```

Ansonsten treten keine unterschied zwischen den Train und der Test Methoden auf, mit einer Ausnahme. Bei der Test Methode wird nicht das NN gespeichert, da dieses bereits vorhanden ist.

6.5. Speicherung

Sämtliche Daten und Networks werden in den baseline-run-n Unterordnern gespeichert. Häufig findet man daher in den Hyperparametern die STATISTIC_RUN_NUMBER, welche für die Benennung des Unterordners zuständig ist. Sollte die save Methode, welche für das speichern der Daten und Networks zuständig ist mit einer STATISTIC_RUN_NUMBER aufgerufen werde, für die noch kein Ordner erstellt wurde, wird dies automatisch geschehen. Der Parameter RUN_TYPE gibt ist für die Benennung und des Unterordners zuständig. Es gibt die Möglichkeit zwischen "baselineünd öptimized". Der RUN_TYPE stellt daher die Art des Laufes dar, ob es ein Baseline- oder Optimized-Vergleich ist. Die AGENT_NUMBER bestimmt, unter welchen Na-

men das NN und die Daten abgespeichert werden. Sollte eine eins übergeben werden so werden bei einem Trainings-Save die Files "PPO-01-train.csvünd "PPO-01-train.model"gespeichert. Sollte ein Training bzw. Test mit der zufallsverteilten Spielfeldgröße durchgeführt werden, so wird dies im File mit dem Namenszusatz "rgs"für random game size signalisiert. Die Trainings Dateien würden dann "PPO-01-rgs-train.csvünd "PPO-01-rgs-train.model"heißen. Mit dem Parameter OPTIMIZATION lässt sich die Optimierung auswählen, welche im Namen der Dateien deutlich gemacht wird. Der OPTIMIZATION Papameter existiert ausschließlich in den Main-Methoden für das Training. Eine NN File, welches mit einer Optimierung trainiert wurde, würde dann "PPO-01-optimization-a-train.model"heißen.

6.6. Statistik

Die Statistiken werden mit der generate_statistic Methode erstellt welche sich im statisticTool File befindet. Dieses ist wiederum im statistic Ordner definiert (siehe 6.1). Ihr wird die STATISTIC_RUN_NUMBER, der RUN_TYPE, der USE_CASE und eine Agenten-Liste übergeben. Die STATISTIC_RUN_NUMBER gibt an welcher Run ausgewertet werden soll. Der RUN_TYPE differenziert zwischen der Auswertung von baseline und optimized Daten. Der USE_CASE unterscheidet die Auswertung von Test- und Trainingsdaten. Mit der Agenten-Liste können gezielt Statistiken zu einzelnen Agenten angefertigt werden. Wird eine leere List übergeben, so wird jeder Agent, welcher über Daten im Run-Ordner verfügt, untersucht. Zuerst wird der durch die übergebenen Parameter aufgespannte Path bestimmt.

```
_, MODEL_DIR_PATH = save_path(statistic_run_number=
    STATISTIC_RUN_NUMBER, alg_type="PPO", agent_number=1,
            random_game_size=False, use_case=USE_CASE, run_type=
                RUN_TYPE, optimization=None)
```

Danach werden alle CSV-Datein, welche die Test- und Trainingsdaten enthalten, eingelesen und die Daten in ein Pandas Dataframe geladen. Diese werden dann in eine Python dict Objekt eingepflegt. Dieses kommt eine Map gleich, welche als Key den Agentennamen und als Value den Dataframe erhält. Für die Verarbeitung von Daten mit zufallsbasierter Spielfeldgröße wird eine weiteres dict Objekt (Map) erzeugt.

```
directory = os.path.join(MODEL_DIR_PATH)
df_dict = {}
df_dict_rgs = {}
for _, _, files in os.walk(directory):
  for file in files:
```

```
    print(file)
    if file.endswith(".csv") and USE_CASE in file and "rgs" not in
        file:
      df_dict[file[:6]] = pd.read_csv(MODEL_DIR_PATH + "\\" + file)
    elif file.endswith(".csv") and USE_CASE in file and "rgs" in file
        :
      df_dict_rgs[file[:6]] = pd.read_csv(MODEL_DIR_PATH + "\\" +
          file)
```

Sollte die Agent-List nicht leer sein, werden alle Agenten aus den dicts aussortiert, welche nicht vorhanden sind.

```
agent_dict = {k: df_dict[k] for k in AGENT_LIST if k in df_dict}
agent_dict_rgs = {k: df_dict_rgs[k] for k in AGENT_LIST if k in
    df_dict_rgs}
if not agent_dict:
  agent_dict = df_dict
  agent_dict_rgs = df_dict_rgs
```

Danach werden die Daten entsprechend der Darstellung im Abschnitt 5.5 bereitgestellt.

```
performance_lists = [value["apples"].rolling(100).mean().fillna(0)
    for value in agent_dict.values()]
efficiency_lists = [(i["apples"] / i["steps"]).rolling(100).mean().
    fillna(0) for i in agent_dict.values()]
robustness_lists = [(i["apples"]).rolling(100).mean().fillna(0) for i
     in agent_dict_rgs.values()]
win_rate = [value["wins"].rolling(100).mean().fillna(0) for value in
    agent_dict.values()]
```

Danach werden die Statistiken mit der Unterfunktion make_statistics erstellt.

```
make_statistics(list(df_dict.keys()), "Epochs", "Apfel-Durchschnitt
    der letzten 100 Epochs pro Epoch",
MODEL_DIR_PATH + "\\performance-rate.png", *performance_lists)
make_statistics(list(df_dict.keys()), "Epochs", "Effizienz-
    Durchschnitt der letzten 100 Epochs pro Epoch",
MODEL_DIR_PATH + "\\effizienz-rate.png", *efficiency_lists)

make_statistics(list(df_dict_rgs.keys()), "Epochs", "Durchschnitt der
     Robustheit der letzten 100 Epochs pro Epoch",
MODEL_DIR_PATH + "\\robustheit-rate.png", *robustness_lists)
```

```
make_statistics(list(df_dict.keys()), "Epochs", "Sieg-Durchschnitt
    der letzten 100 Epochs pro Epoch",
MODEL_DIR_PATH + "\\win-rate.png", *win_rate)
```

Diese Unterfunktion generiert mit Hilfe des Matplotlib Frameworks die Graphiken. Dafür wird die Größe und die Achsenbeschriftung definiert, sowie die Farben der Kurven.

```
def make_statistics(agent_names: list, x_label: str, y_label: str,
    FIG_PATH: str, *args):
  plt.figure(figsize=(16, 6))
  plt.grid(True)
  plt.xlabel(x_label)
  plt.ylabel(y_label)
  color_list = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd'
      , '#17becf']
```

Daraufhin wird iterativ jeder übergebenen Datensätze geplotted. Zum Schluss wird noch eine Legende hinzugefügt, damit die Agenten besser unterschieden werden können und die Statistik wird als File unter dem übergebenen Path gespeichert.

```
for i, value in enumerate(list(args)):
  length = len(value)
  plt.plot([s for s in range(length)], value, linewidth=1, color=
      color_list[i])

handles = []
for i, value in enumerate(agent_names):
  handles.append(mlines.Line2D([], [], color=color_list[i],
      markersize=30, label=value))

plt.legend(handles=handles, loc="lower center", ncol=len(agent_names)
    , bbox_to_anchor=(0.5, -0.2))
plt.savefig(FIG_PATH, bbox_inches='tight')
```

Kapitel 7

Evaluation

Literaturverzeichnis

[Baker u. a. 2019] Baker, Bowen ; Kanitscheider, Ingmar ; Markov, Todor M. ; Wu, Yi ; Powell, Glenn ; McGrew, Bob ; Mordatch, Igor: Emergent Tool Use From Multi-Agent Autocurricula. In: *CoRR* abs/1909.07528 (2019). – URL http://arxiv.org/abs/1909.07528

[Bowei Ma 2016] Bowei Ma, Jun Z.: *Exploration of Reinforcement Learning to SNAKE*. 2016. – URL http://cs229.stanford.edu/proj2016spr/report/060.pdf

[Chunxue u. a. 2019] Chunxue, Wu ; Ju, Bobo ; Wu, Yan ; Lin, Xiao ; Xiong, Naixue ; Xu, Guangquan ; Li, Hongyan ; Liang, Xuefeng: UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene. In: *IEEE Access* PP (2019), 08, S. 1–1. – URL https://ieeexplore.ieee.org/abstract/document/8787847

[Goodfellow u. a. 2018] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning. Das umfassende Handbuch*. MITP Verlags GmbH, 2018. – URL https://www.ebook.de/de/product/31366940/ian_goodfellow_yoshua_bengio_aaron_courville_deep_learning_das_umfassende_handbuch.html. – ISBN 3958457002

[Haarnoja u. a. 2018] Haarnoja, Tuomas ; Zhou, Aurick ; Abbeel, Pieter ; Levine, Sergey: Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In: *CoRR* abs/1801.01290 (2018). – URL http://arxiv.org/abs/1801.01290

[Lapan 2020] Lapan, Maxim: *Deep Reinforcement Learning - Das umfassende Praxis-Handbuch*. MITP Verlags GmbH, 2020. – URL https://www.ebook.de/de/product/37826629/maxim_lapan_deep_reinforcement_learning.html. – ISBN 3747500366

[Mnih u. a. 2016] Mnih, Volodymyr ; Badia, Adrià P. ; Mirza, Mehdi ; Graves, Alex ; Lillicrap, Timothy P. ; Harley, Tim ; Silver, David ; Kavukcuoglu, Koray: Asynchronous Methods for Deep Reinforcement Learning. In: *CoRR* abs/1602.01783 (2016). – URL http://arxiv.org/abs/1602.01783

[Mnih u. a. 2015] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg ; Petersen, Stig ; Beattie, Charles ; Sadik, Amir ; Antonoglou, Ioannis ; King, Helen ; Kumaran, Dharshan ; Wierstra, Daan ; Legg, Shane ; Hassabis, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Februar, Nr. 7540, S. 529–533. – URL `http://dx.doi.org/10.1038/nature14236`. – ISSN 00280836

[Schulman 2017] Schulman, John: *Deep RL Bootcamp Lecture 5: Natural Policy Gradients, TRPO, PPO*. `https://www.youtube.com/watch?v=xvRrgxcpaHY`. Oktober 2017

[Schulman u. a. 2015] Schulman, John ; Levine, Sergey ; Moritz, Philipp ; Jordan, Michael I. ; Abbeel, Pieter: Trust Region Policy Optimization. In: *CoRR* abs/1502.05477 (2015). – URL `http://arxiv.org/abs/1502.05477`

[Schulman u. a. 2017] Schulman, John ; Wolski, Filip ; Dhariwal, Prafulla ; Radford, Alec ; Klimov, Oleg: Proximal Policy Optimization Algorithms. In: *CoRR* abs/1707.06347 (2017). – URL `http://arxiv.org/abs/1707.06347`

[Sutton und Barto 2018] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. – URL `http://incompleteideas.net/book/bookdraft2018jan1.pdf`

[Wei u. a. 2018] Wei, Zhepei ; Wang, Di ; Zhang, Ming ; Tan, Ah-Hwee ; Miao, Chunyan ; Zhou, You: Autonomous Agents in Snake Game via Deep Reinforcement Learning. In: *2018 IEEE International Conference on Agents (ICA)*, 2018, S. 20–25

Anhang A

Anhang

A.1. Backpropagation und das Gradientenverfahren

Nachdem nun alle Agenten-Klassen vorgestellt sind, man sich vielleicht der eine oder andere Leser frage, wie denn nun das eigentliche Lernen vonstattengeht. Die dem Lernen zugrunde liegenden Verfahren sind das Backpropagation und das Gradient Descent. Dabei wird häufig fälschlicherweise angenommen, dass sich hinter dem Begriff Backpropagation der komplette Lernprozess verbirgt. Dem ist jedoch nicht so. Der Backpropagation-Algorithmus oder auch häufig einfach nur Backprop genannt, ist der Algorithmus, welcher zuständig für die Bestimmung der Gradienten in einer Funktion. Häufig wird ebenfalls angenommen, dass Backprop nur für NN anwendbar sind, den ist jedoch nicht so. Prinzipiell können mit dem Backprob-Algorithmus die Gradienten einer jeden Funktion bestimmt werden, egal ob NN oder eine Aktivierungsfunktion, wie z.B. Sigmoid oder TanH (Goodfellow u. a., 2018, S. 90ff.).

Das Gradientenverfahren oder im Englischen auch Gradient Descent genannt, wird dafür eingesetzt um die eigentliche Optimierung des NN durchzuführen. Dafür werden jedoch die Gradienten benötigt, welche im Vorhinein durch den Backprop-Algorithmus bestimmt wurden. Jedes NN definiert je nach den Gewichten des NN eine mathematische Funktion. Diese steht in Abhängigkeit von den Inputs und berechnet auf deren Basis die Outputs bzw. Ergebnisse. Basierende auf dieser Funktion lässt sich eine zweite Funktion definieren, die Loss Function oder Kostenfunktion oder Verlustfunktion usw. Diese gibt den Fehler wieder und soll im Optimierungsverlauf minimiert werden, um optimale Ergebnisse zu erhalten. Diese Fehlerfunktion zu minimieren müssen die Gewichte des NN soweit angepasst werden, der die Fehlerfunktion geringe Werte ausgibt. ist diese für alle Daten, mit welchen das NN jemals konfrontiert wird geschafft, so ist das NN perfekt angepasst (Goodfellow u. a., 2018, S. 225ff.).

Ein näheres Eingehen auf die Bestimmung der Gradienten im Rahmen des Backpropagation-

Algorithmus und auf die Anpassung der Gewicht im Rahmen des Gradientenverfahrens wird der Übersichtlichkeit entfallen. Des Weiteren machen moderne Framework wie Facebooks PyTorch, Googles Tensorflow oder Microsofts CNTK das detaillierte Wissen um diese Verfahren für anwendungsorientiert Benutzer obsolet.

A.2. Tensoren

Tensoren beschreiben die grundlegenden Einheiten eines jeden DL-Frameworks. Aus der Sicht der Informatik handelt es sich bei ihnen um mehrdimensionale Arrays, welche jedoch kaum etwas mit der mathematischen Tensorrechnung bzw. Tensoralgebra gemein haben. (Lapan, 2020, S. 67) Genauer gesagt lassen sich Tensoren als, Anordnung von Zahlen in einem regelmäßigen Raster mit variabler Anzahl von Achsen. (Goodfellow u. a., 2018, S. 35)
Ein DL-Framework kann aus z.B. einem Numpy Array `https://numpy.org/` einen Tensor konstruieren, welche dann über für die Verwendung in einem NN nutzbar ist. Dieser könnte Daten oder die gewichte eines NN beinhalten. Der entstandene Tensor dient dabei als eine Wrapper, welcher die Informationen des Arrays teilt oder kopiert hat. Neben vielen Zusatzfunktionen ist keine so wichtig wie das Aufbauen eines Backward Graphs. Ein DL-Framework ist mit diesem Graph in der Lage, jede Veränderung des Tensors einzusehen, um darauf aufbauend Backpropagation durchzuführen. (Lapan, 2020, S. 72 ff.)

A.3. Convolution Neural Networks

Convolutional Neural Networks (CNNs) sind eine Form neuronaler Netzwerke, die besonders für das Verarbeiten von rasterähnlichen Daten geeignet sind. Sie bestehen meist aus verschiedenen Layers, wie z.B. Convolutional, Pooling und FC Layers.

A.3.1. Convolutional Layer

Convolutional Layers (Conv Layers) sind der zentrale Baustein von CNNs, da sie besonders für die Feature Detektion geeignet sind. Ihre Gewichte sind in einem Tensor gespeichert. In diesen befinden sich die sogenannten Kernels oder auch Filter genannt, welche zweidimensionale Arrays darstellen. Bei Initialisierung der Conv Layers werden die Ein- und Ausgabe Channel der Inputs bzw. Outputs angegeben, sodass entsprechende dieser Informationen die Kernels erstellt werden können.
Des Weiteren wird noch die Größe der Kernels übergeben, wobei häufig Größen von (3x3), (5x5), (7x7) oder (1x1) genutzt werden.

Die Ausgabe eines Output Channels berechnet sich aus der Addition aller Input Channel Feature Maps, welche durch die Verrechnung mit den Kernels (Convolutionale Prozedur) entstanden sind. Für jeden Output Channel existiert ein Input Channel viele Kernel mit, daher ergibt sich eine Gewichtsmatrix der Form (Output_Channel, Input_Channel, Kernel_Size[0], Kernel_Size[1]) (Goodfellow u. a., 2018, S. 369 ff.)
Die Funktionsweise der Convolutionalen Prozedur stellt sich wie folgt dar:
Jeder einzelne Kernel wird mit der Eingabe entsprechend A.1 multipliziert und addiert. Ist dieser Berechnungsschritt abgeschlossen, bewegt sich das Eingabequadrat um dem sogenannten Stride weiter nach rechte (in der Grafik wird ein Stride von eins verwendet). Sollte ein Stride von zwei verwendet werden, so würde die Ausgabe $bw + cx + fy + gz$ nicht existieren und die resultierende Feature Map wäre kleiner. Daher wird der Stride gerne dazu verwendet, um die Feature Map Size und damit den Berechnungsaufwand zu senken. Des Weiteren lässt sich der Stride nicht nur in der Horizontalen sondern auch in der Vertikalen anwenden.
Nicht in A.1 abgebildet ist das sogenannte Padding. Bei diesem wird an den Feature Maps weitere Nullzeilen bzw. Nullspalten angefügt. Dabei kann an allen vier Seiten oder an speziell ausgewählten Zeilen bzw. Spalten hinzuaddiert werden. Wie in A.1 zu erkennen ist, hat die Feature Map die Form (3x4), jedoch ist der Output nur noch von der Form (2x3). Die Durchführung der Convolution Prozedur sorgt für eine Verkleinerung, welche durch Padding verhindert werden kann. (Goodfellow u. a., 2018, S. 369 ff.) Die Ausgaben in A.1 bilden eine Feature Map, welche mit allen weiteren Feature Maps zusammenaddiert werden müsste um einen Output Channel zu bilden.

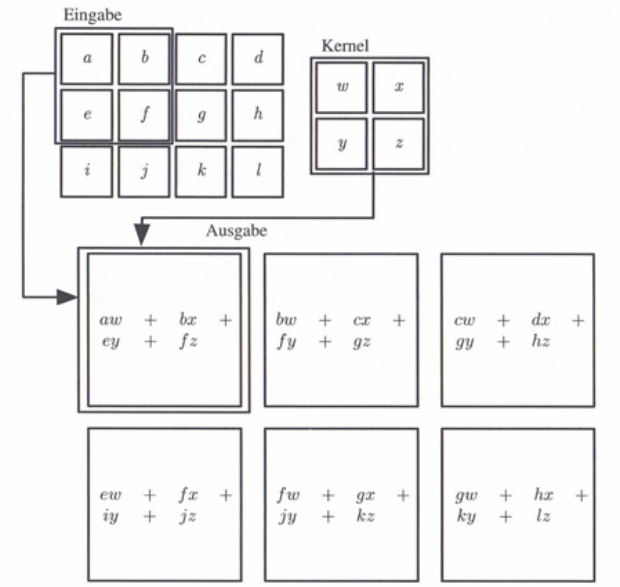

Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

A.3.2. Pooling Layer

Pooling beschreibt die Minimierung der Feature Maps, wie es bereits in **??** angesprochen wurde. Zu diesem Zweck wird in A.2 ein Kernel und Stride der Form (2x2) gewählt, welcher ein Max-Pooling durchführen soll. Dieser Kernel bewegt sich über die Feature Map entsprechende des Strides und gibt den maximalen Wert innerhalb der Kernels wieder. Somit wird aus einer (4x4) eine (2x2) Feature Map. (Goodfellow u. a., 2018, S. 379 ff.)

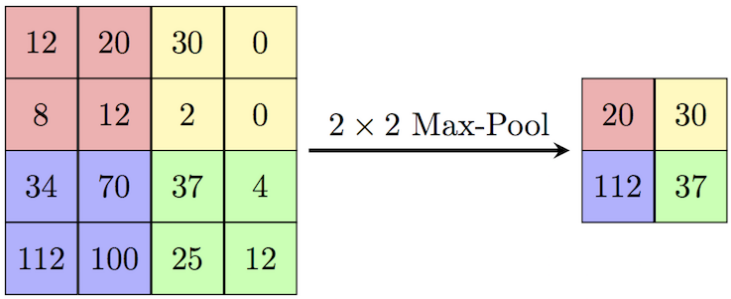

Abbildung A.2.: Darstellung des Max-Poolings. `https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png`

A.3.3. Fully Connected Layer

Die Fully Connected Layer (FC) sind die grundlegendsten Elemente eines NN. Ein solches Layer besteht aus zwei Teilen, die beide einer Initialisierung benötigen. Die Gewichte des FC sind als Tensor mit der Form (Anzahl der Output Features, Anzahl der Input Features) initialisiert. Zuzüglich kann optional noch ein Bias erstellt werden, welcher auf jedes Output Feature noch einen Rauschwert aufaddiert. Die Größe dieser Rauschwerte werden durch Backpropagation und Gradientenverfahren bestimmt, wie es auch bei den Gewichten der Fall ist. Die FC Layer implementieren daher folgende Funktion:

$$y = xA^T + b \tag{A.1}$$

wobei y das Ergebnis, x der Input Tensor, A^T die Gewichte und b das Bias, darstellt. `https://pytorch.org/docs/stable/generated/torch.nn.Linear.html`

Anhang B

Anhang zur Implementierung

B.1. Around-View

Zu Erstellung der around_view (AV) werden zunächst einige Konstanten gesetzt bzw. berechnet, wie z.B. die width oder c_s und c_h erstellt. Erstere entspricht dem Radius des Ausschnittes um den Kopf. Die letzteren sind Farbkonstanten, welche im Player definiert wurden (siehe 6.2.2).
Danach wird über jeden Eintrag, welcher im Ausschnitt der AV liegt, iteriert. Dieser Ausschnitt besitzt die Form eines Vierecks in dessen Mittelpunkt sich der Kopf der Snake befindet. Für jeden dieser Einträge wird überprüft, ob er einem der Merkmalen aus 5.2 entspricht. Sollte ein Merkmal erkannt worden sein, wie z.B. die Position des Apfels, so wird diese in den dazugehörigen Channel eingepflegt.

```
def create_around_view(pos, id, g):
  width = 6
  c_h = id * 2
  c_s = id
  tmp_arr = np.zeros((6, width * 2 + 1, width * 2 + 1), dtype=np.
     float64)

  for row in range(-width, width + 1):
    for column in range(-width, width + 1):
      if on_playground(pos[0] + row, pos[1] + column, g.shape):
        a, b = pos[0] + row, pos[1] + column
        if g[a, b] == c_s:
          tmp_arr[1, row + width, column + width] = 1
          continue

        elif g[a, b] == c_h:
          tmp_arr[2, row + width, column + width] = 1
          continue
```

```
            elif g[a, b] == 0:
              tmp_arr[3, row + width, column + width] = 1
              continue

            elif g[a, b] == -1:  # End of snake tail.
              tmp_arr[4, row + width, column + width] = 1
              continue

            elif g[a, b] == -2:
              tmp_arr[5, row + width, column + width] = 1

          else:
          tmp_arr[0, row + width, column + width] = 1

    return np.expand_dims(tmp_arr, axis=0)
```

B.2. Distanzbestimmung

Ein wesentlicher Teil der scalar_obs (SO) besteht aus den Distanzen, welche von der create_distances Funktion generiert werden. Zu Beginn wird ein Null-Array der Länge 24 (3 * 8 = 24), in welches die Distanzen eingetragen werden und eine List namens grad_list erstellt. In dieser werden Faktoren gehalten, welche den Grad der Sucherlinie (siehe 5.4) angeben. Dabei entsprechen die Faktoren den Himmels- bzw. Nebenhimmelsrichtungen in der folgenden Reihenfolge (0° = N, 45° = NO, 90° = O, 135° = SO, 180° = S, 225° = SW und 270° = W, 315° = NW).
Danach wird über die drei zu suchenden Objekte (Wände, Snake-Glieder, Apfel) iteriert, wobei für jedes dieser Objekt die dist Unterfunktion aufgerufen wird. Die Rückgaben werden in das Array eingepflegt und dieses ,sobald alle Distanzen bestimmt worden sind, zurückgegeben.

```
  def create_distances(pos, ground):
    obs = np.zeros(24, dtype=np.float64)
    a = 0
    grad_list = [(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0,
        -1), (-1, -1)]
    for wanted in [[], [-1, 1, 2], [-2]]:
      for grad in grad_list:
        obs[a] = dist(ground, pos, wanted, *grad)
        a += 1
    return obs
```

Der dist Unterfunktion werden das Spielfeld (ground), die Position des Snake-Kopfes (p_pos) die zu suchenden Werte (wanted) und fac_0 und fac_1 übergeben. Diese stellen die oben genannten Faktoren dar, welcher die Ausrichtung der Suchlinien definiert. Nach der Initialisierung weiterer Hilfsvariablen und der Distanz (dist_), wird der erste Index generiert, welcher dem ersten Feld auf dem Weg der Linie entspricht. Sollte der Eintrag an dieser Stelle dem gesuchtem Objekt entsprechen, so wird die Zahl zwei zurückgegeben. Anderenfalls wird dist_ inkrementiert und der nächste höhere Index der auf dem Weg der Linie liegt generiert. Dann wird wieder so verfahren wie oben bereits erwähnt.
Sollte einer der Indexe jedoch das Spielfeld verlassen, so wird eine Distanz von null zurückgegeben.
Diese Prozedur endet entweder mit der Rückgabe einer Distanz zum Objekt, falls dieses gefunden wird, oder durch Rückgabe von null falls das Objekt nicht im Pfad der Linie liegt.

```
def dist(ground, p_pos, wanted_hit, fac_0, fac_1):
  dist_, i_0, i_1 = 0, 1, 1
  p_0 = p_pos[0] + fac_0 * i_0
  p_1 = p_pos[1] + fac_1 * i_1
  while on_playground(p_0, p_1, ground.shape) and ground[p_0, p_1]
      not in wanted_hit:
    i_0 += 1
    i_1 += 1
    dist_ += 1
    p_0 = p_pos[0] + fac_0 * i_0
    p_1 = p_pos[1] + fac_1 * i_1
  if not on_playground(p_0, p_1, ground.shape) and bool(wanted_hit):
    return 0
  return 1 / dist_ if dist_ != 0 else 2
```

B.3. AV-Network

Mit Hilfe der PyTroch Oberklasse Module, kann eine neue NN-Element erstellt werden. Es wurde daher die Klasse AV_NET definiert, welche zugleich als NN-Schicht benutzt werden kann.

```
class AV_NET(nn.Module):
  def __init__(self):
    super(AV_NET, self).__init__()
    self.AV_NET = nn.Sequential(
```

```
        nn.Conv2d(in_channels=6, out_channels=8, kernel_size=(3, 3),
            stride=(1, 1), padding=(1, 1)),
        nn.ReLU(),
        nn.Conv2d(in_channels=8, out_channels=8, kernel_size=(3, 3),
            stride=(1, 1), padding=(1, 1)),
        nn.ReLU(),
        nn.ZeroPad2d((0, 1, 0, 1)),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(392, 256),
        nn.ReLU(),
        nn.Linear(256, 128)
        )

    def forward(self, av):
        return self.AV_NET(av)
```

Diese besitzt eine forward Methode, welche die AV (around_view) durch die dargestellten NN-Schichten und Funktionen propagiert. Das Ergebnis wird zurückgegeben.

B.4. DQN-Memory

Anhang C

Anleitung

Zur besseren Anwendbarkeit der Software, wurden die Files main_train und main_play erstellt. Mit diesen kann ein Benutzer die Trainings- und Spielroutine starten. Alternativ lässt sich dies auch durch die Verwendung von Python spezifischen Entwicklungsumgebungen durchführen.
Zum Start des Trainings muss main_train mit den folgenden Parametern gestartet werden. Dabei ist jedoch zu beachten, dass je nach Algorithmus-Art unterschiedliche Parameter übergeben werden müssen. Die Algorithmus-Art wird zu diesem Zweck als erstes Startargument übergeben.

C.1. PPO Train Startargumente

1. "PPO" ⟶ Algorithmus-Art
2. N_ITERATIONS (Int) ⟶ Anzahl l der zu spielenden Spiele
3. LR_ACTOR (Float) ⟶ Lernrate der Actor-NN
4. LR_CRITIC (Float) ⟶ Lernrate des Critic-NN
5. GAMMA (Float) ⟶ Abzinsungsfaktor 2.1
6. K_EPOCHS (Int) ⟶ Gibt die Anzahl der Lernzyklen eines Batches bzw. Spieles an. Siehe 2.3.3
7. EPS_CLIP (Float) ⟶ Clip Faktor, welcher St Standard bei 0.2. Siehe 2.3.2
8. BOARD_SIZE (Tuple of Integers) ⟶ Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
9. STATISTIC_RUN_NUMBER (Int) ⟶ Nummer des Statistik-Runs.
10. AGENT_NUMBER (int) ⟶ Nummer des zu untersuchenden Agenten.

11. RUN_TYPE (String) ⟶ "baselineöder öptimizedRun. Wichtig für die Speicherung.
12. RAND_GAME_SIZE (Boolean) ⟶ Wenn True wird auf einem sich dynamisch pro Spielepisode ändernden Spielfeld zwischen (6, 6) bis zu (10, 10) gespielt.
13. SCHEDULED_LR (Boolean) ⟶ Wenn True, dann wird die Lernrate heruntergesetzt, falls die Steigung der Performance in den letzten 100 Epochs nicht größer null war.
14. GPU (Boolean) ⟶ Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Trainingsprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

Path_to_File\Bachelor-Snake-AI\src\python main_train.py "PPO" 30000 0.0001 0.0004 0.95 10 0.2 "(8, 8)" 1 2 "baseline" False False True

C.2. DQN Train Startargumente

1. "DQN" ⟶ Algorithmus-Art
2. N_ITERATIONS (Int) ⟶ Anzahl l der zu spielenden Spiele
3. LR (Float) ⟶ Lernrate der Q-NN
4. GAMMA (Float) ⟶ Abzinsungsfaktor 2.1
5. BATCH_SIZE (Int) ⟶ Größe des zu entnehmenden Batches 2.4.1
6. MAX_MEM_SIZE (Int) ⟶ Maximale Größe des Memory.
7. EPS_DEC (Float) ⟶ Der Absenkungsfaktor von Epsilon 2.4.1
8. EPS_END (Float) ⟶ Der Endwert von Epsilon 2.4.1
9. BOARD_SIZE (Tuple of Ints) ⟶ Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
10. STATISTIC_RUN_NUMBER (Int) ⟶ Nummer des Statistik-Runs.
11. AGENT_NUMBER (int) ⟶ Nummer des zu untersuchenden Agenten.
12. RUN_TYPE (String) ⟶ "baselineöder öptimizedRun. Wichtig für die Speicherung.

13. RAND_GAME_SIZE (Boolean) ⟶ Wenn True wird auf einem sich dynamisch pro Spielepisode ändernden Spielfeld zwischen (6, 6) bis zu (10, 10) gespielt.
14. OPTIMIZATION (String) ⟶ Ä", "B", oder None. Wählt die Optimierung A (siehe 5.4.1), B (siehe 5.4.2) oder keine aus.
15. GPU (Boolean) ⟶ Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Trainingsprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

python Path_to_Directory\Bachelor-Snake-AI\src\main_train.py "DQN" 30000 0.0001 0.99 64 2048 0.00007 0.01 "(8, 8)" 1 2 "baseline" False False True

C.3. Test Startargumente

Da die Testmethoden der beiden Algorithmus-Arten nahe zu identisch sind, teilen sie alle Startargumente bis auf die Algorithmus-Art. Daher können die Startparameter beider Methoden zusammen erklärt werden.

1. "DQN/ "PPO"⟶ Algorithmus-Art
2. MODEL_PATH (String) ⟶ Path der Model Datei für das NN des Agenten.
3. N_ITERATIONS (Integer) ⟶ Anzahl l der zu spielenden Spiele.
4. BOARD_SIZE (Tuple of Integers) ⟶ Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
5. STATISTIC_RUN_NUMBER (Integer) ⟶ Nummer des Statistik-Runs.
6. AGENT_NUMBER (Integer) ⟶ Nummer des zu untersuchenden Agenten.
7. RUN_TYPE (String) ⟶ "baselineöder öptimizedRun. Wichtig für die Speicherung.
8. RAND_GAME_SIZE (Boolean) ⟶ Wenn True wird auf einem sich dynamisch pro Spielepisode ändernden Spielfeld zwischen (6, 6) bis zu (10, 10) gespielt.
9. GPU (Boolean) ⟶ Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Spielprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

python Path_to_Directory\Bachelor-Snake-AI\src\main_test.py "PPO" "Path_to_Model" 30000 "(8, 8)" 1 2 "baseline" False True

Erklärung

Hiermit versichere ich, Lorenz Mumm, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Lorenz Mumm