Carl von Ossietzky Universität Oldenburg

Wirtschaftsinformatik
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, 25. 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

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

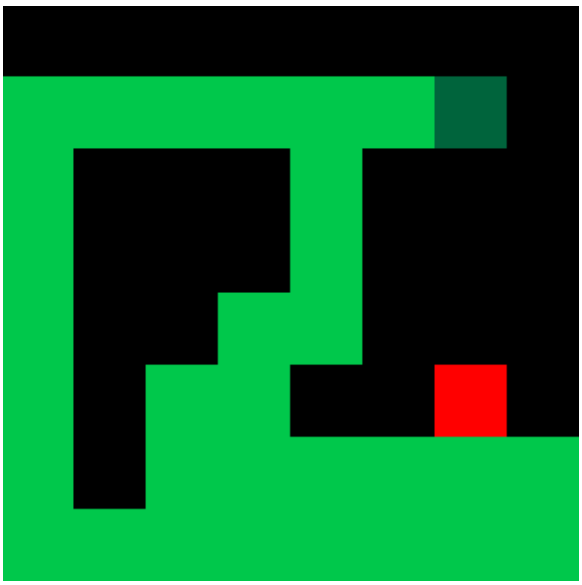

Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

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

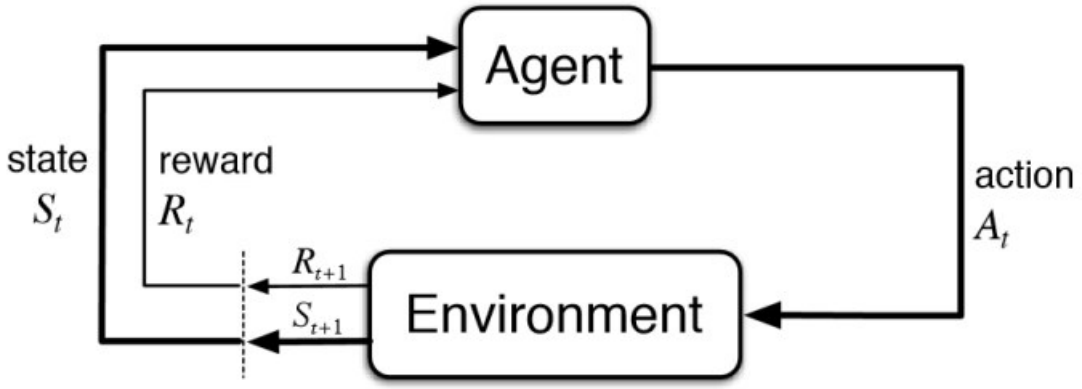

Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State S_t und falls $t \neq 0$ einen Reward R_t. Daraufhin wird vom Agenten eine Action A_t ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State S_{t+1} und Reward 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 **??** 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 Folgestate 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 die Grundlagen für den weiteren Vergleich der Reinforcement Learning Agenten gelegt, welche auf zwei unterschiedlichen Algorithmus-Arten (DQN und PPO) basieren.
Um diesem Vergleich durchzuführen soll ein System erstellt werden, welches diese geplanten Vergleich durchführen, festhalten und auswerten kann. Dieses soll aus einem Environment, aus mehreren Agenten beider Algorithmus-Arten, 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 Env das Spiel Snake implementieren soll, ergeben sich weitere zusätzliche Anforderungen.

3.1.1. Standardisierte Schnittstelle

Das Env soll eine standardisierte Schnittstelle besitzen, sodass nach 2.2.1 drei Kommunikationskanäle implementiert werden. Das Env soll in der Lage sein, Actions zu empfangen. Des Weiteren soll es Observation und Rewards an den Agenten übergeben können.

3.1.2. Funktionalitäten

Das Env soll die folgenden Funktionalitäten implementieren.

Aktionsausführung

Das Env muss eine Funktionalität beinhalten, welche die vom Agenten bestimmt Aktion ausführen kann. Diese Aktionsausführung muss sich nach den Regeln des Spiels Snake richten 2.1.

Reset

Das Env muss eine Reset Methode implementieren, um einen erbrachten Spielfortschritt zu löschen. 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.

3.2. Anforderungen an die Agenten

In diesem Abschnitt werden die Anforderungen an die Agenten, welche auf den beide Algorithmus-Arten (PPO, DQN) basieren, dargestellt.

3.2.1. Funktionalitäten

Die Agenten müssen folgende Funktionalitäten implementieren.

Aktionsbestimmung

Die Agenten müssen in der Lage sein aus der Obs Aktionen zu bestimmen, welche dem Env übergeben werden.

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 der gleichen Algorithmus-Art zu erstellen, welche sich jedoch durch verwendeten Hyperparameter und Algorithmus-Art unterscheiden. Diese Definition von Agenten ist in der Evaluation zu berücksichtigen.

3.2.3. Diversität der RL Algorithmen

Um nicht nur Agenten einer Algorithmus-Art untereinander zu vergleichen, sondern auch den Vergleich zu anderen Algorithmus-Arten zu erbringen, sollen ein DQN- und PPO-Algorithmus miteinander verglichen werden. Diese bieten sich wie in 2.3 und 2.4 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 der muss für jeden einzelnen Agenten mehrfach durchgeführt werden, um die Validität der Messung zu gewährleisten.

3.3.2. Datenspeicherung

Damit aus den erzeugten Daten statistische Schlüsse gezogen werden können ist es wichtig, dass die erzeugten Spieldaten gespeichert werden. Da jedoch die Menge an Daten schnell riesige Dimensionen annimmt, sollen stellvertretend nur die Daten ganzer Spiele gespeichert werden. Dies diese Strategie stellt einen Kompromiss zwischen Vollständigkeit und effizientem Speicherplatzmanagement dar.
Das System muss daher folgende Daten speichern:

Tabelle 3.1.: Zu erhebende Daten

Daten	Erklärung
time	Die Uhrzeit. Dies dient später dem Geschwindigkeitsvergleich.
steps	Die in einem Spiel durchgeführten Züge. Diese geben in der Evaluation später Aufschluss über den Lernerfolge und weisen auf Lernfehler der Agenten, wie beispielsweise das Laufen im Kreis, hin.
apples	Die Anzahl der gefressenen Äpfel in einem Spiel. Maßgeblicher Evaluationsfaktor zur Einschätzung des Lernerfolges.
wins	Hat der Agent das Spiel gewonnen. Dieser Wert stellt die Endkontrolle des Agenten dar.

epsilon (nur beim Deep Q-Learning Algorithmus)	Gibt die Wahrscheinlichkeit für das Ziehen einer zufälligen Aktion wieder.
board_size	einmalige zu speichernder vektorieller Wert. Dieser gibt die Feldgröße des Environments an. Dieser ist wichtig für die Evaluation der Robustheit der Agenten.

3.4. Anforderungen an die Statistiken

Das System muss in der Lage sein, Statistiken zu generieren und zu speichern, welche für die Evaluation verwendet werden. Diese Statistiken sollen auf den Werten der Evaluationskriterien 3.2 basieren.

3.5. Anforderungen an die Evaluation

Bei der Evaluation soll der möglichst optimale Agent entsprechend verschiedener Gesichtspunkten ermittelt werden. Einige dieser Kriterien stammen aus der verwendeten Literatur Wei u. a. (2018) und Chunxue u. a. (2019) wie z.B. die Performance und Spielzeit. Die anderen ergeben sich aus den zu speichernden Daten. Die einzelnen Kriterien lauten:

Tabelle 3.2.: Evaluationskriterien

Kriterium	Erläuterung
Performance	Welcher Agent erreicht, nach einer festen Anzahl an absolvierten Spielen, das beste Ergebnis? Im Sachzusammenhang mit dem Spiel Snake bedeutet dies: Welcher Agent frisst die meisten Äpfel, nach dem er über eine feste Anzahl an Spielen trainiert wurde?
Effizienz	Welcher Agent löst das Spiel mit der größten Effizienz. Bezogen auf das Spiel Snake bedeutet dies, welches Agent ist in der Lage die Äpfel mit möglichst wenig Schritten zu fressen?
Robustheit	Welcher Agent ist in der Lage in einer modifizierten Umgebung das Spielziel am besten zu erreichen? In Bezug auf Snake bedeutet dies: Welcher Agent ist in der Lage auf einem größeren oder kleineren Spielfeld die meisten Äpfel zu fressen?

Siegrate	Welcher Agent schafft die Spiele mit einem Sieg zu beenden.
Spielzeit	Welcher Agent schafft es am längsten einen terminalen Zustand zu vermeiden? Auf Snake bezogen: Welcher Agent schafft es am längsten nicht zu sterben.

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 und diese anschließend diskutiert werden. Die Arbeiten wurde aufgrund ihres thematischen Hintergrundes zum Spiel Snake, in Verbindung mit dem RL, ausgewählt. Zu den Aspekten gehören:

- Optimierungsstrategien
- Reward Function
- 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 Äutonomous 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 werden in diesem Paper drei Optimierungsstrategien verwendet, 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 sortiert und gespeichert werden. Mem1 besteht nur aus Erfahrungen, welche einen Reward größer als ein vordefinierter Grenzwert besitzen. Die restlichen Erfahrungen wandern in Mem2. Beim Lernen wird, zu beginn, eine überproportional größere Menge an Erfahrungen aus Mem1 ausgewählt,

um den Lernerfolg zu beschleunigen. Im weiteren Lernverlauf wird dann das Entnahmeverhältnis normalisiert (Mem1: 50% & Mem2: 50%).
Der Training Gap stellt die Erfahrungen dar, welche der Agent, zum Zwecke des performanteren Lernens, nicht verarbeiten soll. Zu diesen zählt die Erfahrung direkt nach dem Konsum eines Apfel, sodass der Agent die Neuplatzierung eines Apfels erlernt. Da der Agent auf diesen Prozess keinen Einfluss hat, könnte die Verarbeitung dieser Daten den Lernerfolg mindern, weshalb die Training Gap Strategie etwaige Erfahrungen löscht.
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 mit 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 Function, welche das Paper verwendet, besteht damit aus dem Standard Distanz Reward ohne Training Gap Erfahrungen. Letzterer bestimmt sich aus der Distanz zwischen Kopf und Apfel und der Snake-Länge, wobei der Faktor der Länge wieder antiproportional den Reward beeinflusst. Sollte die Timeout Strategy auslösen, so werden die letzten P Erfahrungen entsprechend angepasst und in Mem2 verschoben.
Als maßgebliche Kriterien zur Evaluation der Leistung des DQN wurde die Performance, daher die Anzahl an gefressenen Äpfeln 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 Resultaten basierend, gute Ergebnisse erzielen konnten. Jedoch ist auch Kritik am Paper anzubringen. Das Env wird im Paper nur kurz und nicht detailliert genug vorgestellt. Aus dem Abschnitt Game Environment lässt sich jedoch schließen, dass alle Anforderungen des Env 3.1 vorhanden sind.

Negativ anzumerken ist jedoch, dass die Verfasser keinerlei Vergleich mit anderen Algorithmus-Arten vorgesehen haben. Auch die Optimierungen darunter Dual Experience Replay, Training Gap und Timeout Strategy sind nicht direkt auf den PPO-Algorithmus anwendbar. Zwar sind die Funktionalitäten des im Paper verwendeten DQN gegeben 3.1.2, jedoch wurde nicht auf eine Parametrisierung 3.2.2 geachtet. Es wird nur ein unoptimierter DQN Agent und ein optimierter DQN Agent betrach-

tet. Diese unterscheiden sich nur durch die Optimierungen, andere Kriterien wurden nicht berücksichtigt.

Auch bei der statistischen Datenerhebung existieren Abweichungen zu den Anforderungen in 3.3. Zhepei Wei et al. verzichteten auf eine mehrfache Datenerhebung, stattdessen markierten sie ihre Ergebnisse als experimentell. Dem Leser werden des Weiteren keine Informationen über die erhobenen Daten direkt mitgeteilt. Aus Statistiken lässt sich jedoch schließen, dass Scores und Spielzeiten gespeichert wurden.

Weiterhin ist erwähnenswert, dass die Evaluationskriterien sich einzig auf die Performance und Spielzeit beschränken. Diese wird dabei am Score gemessen.
Eine weitere Betrachtung, beispielsweise der Robustheit oder Trainingszeit 3.2, wird vernachlässigt. Dies soll in dieser Ausarbeitung jedoch geschehen, da diese Faktoren ebenfalls wichtig für die voll umfassende Bewertung der Agenten sind, besonders in Real World Applications 1.1.

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

In der folgenden Auseinandersetzung wird sich auf 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 Gebiete, in welchen ein größerer Reward zurückgegeben wird als außerhalb der Gebiete. Dabei unterscheiden sich diese in der Höhe des Rewards, sodass der dritte den geringsten und der erste 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 und zu langen Trainingsdauern 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 zeigen, dass der Loop Storm Effect kaum mehr präsent ist.
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 dem Odor Effect und der Breakout Loop Storm Strategy möglich, eine Konvergenz nach 8 Millionen Epochs (hier Trainingsdurchläufe) zu verzeichne.
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. Dennoch sind auch bei diesem viele der in Kapitel drei gestellten Anforderungen 3 nicht beachtet oder erfüllt worden.

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 3.5 erfüllt worden sind, da ansonsten kein Training eines Agenten möglich wäre. Dennoch wurde der Leser nur unzureichend mit Details über das Env informiert.

Das Paper legt seinen Schwerpunkt deutlich mehr auf die Grundlagen des RL, wie z.B. auf den Markov decision process und die RL Kernbegriffe 2.2.1.

Auch erfüllt die Ausarbeitung alle Anforderungen an die Funktionalitäten von Agenten 3.2.1. Dennoch wurde auch hier wenig Aufmerksamkeit in die Parametrisierung der Agenten investiert. Zuzüglich missachtet dieses Paper ebenfalls die Algorithmus Diversität. Zwar werden gegen Ende einige Vergleich zu einer Hand voll anderer Agenten getätigt, jedoch wird dies nur sehr rudimentär durchgeführt. Dem Leser bleiben sowohl die genauen Anforderungen als auch die Rahmenbedingungen verwehrt.

Wie auch im ersten Paper setzen die Verfasser nicht auf eine mehrfache Datenerhebung für die statistische Auswertung. Auch wird nicht erwähnt was für Daten

im Verlauf des Trainings und es Spielens gespeichert werden. Einig die gezeigten Statistiken weisen jedoch darauf hin, dass der Score sowie der mittlere Aktionswert (Q-Value) gespeichert werden. Zuzüglich werden, für die Optimierungen, die steps und das Auftreten von Loop Storms gespeichert.

Chunxue Wu et al. verwendeten zudem nur den Score als hauptsächlichen Evaluationskriterium. Um die Effizient der im Paper verwendeten Strategien zu zeigen wurden ebenfalls die gemittelten Steps pro Spiel und das Auftreten von Loop Stoms als weitere untergeordnete Evaluationskriterien verwendet. Diese wirken jedoch aufgesetzt und ideal gewählt, um die Leistungsfähigkeit der Optimierungen zu zeigen.

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 angewendet werden sollen. Damit die Optimierungen auf beide Agenten Arten (DQN und PPO) angewendet werden können, müssen sie jedoch an die Algorithmus-Arten angepasst werden.
Des Weiteren ist zu bemerken, dass beide Paper nur sehr eingeschränkt die in Kapitel drei gewählten Anforderungen erfüllen. Besonders auffällig ist dabei die geringe Algorithmus Diversität. Es wurden hauptsächlich DQN verwendet und andere Algorithmen wurden entweder nur unzureichend in einen Vergleich eingebunden oder vollkommen ausgelassen.

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 Implementierung 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 sind.
Zu Beginn sei erwähnt, dass davon ausgegangen wird, dass alle für den Verglich benötigten Komponenten, wie z.B. Environment, Agenten und statistische Analysekomponenten entsprechende der Anforderungen 3 implementiert sind.
Als erstes werden die Agenten erstellt 5.1. Für diese Initialisierung werden die Hyperparameter aus 5.3 verwendet. Mit diesem Baseline Agenten Bestand werden nun die weiteren Vergleich durchgeführt.
Für jedes Evaluationskriterien werden aus dem gegebenen Agenten-Bestand die zwei optimalen Baseline Agenten ausgewählt. Die Algorithmus-Art besitzt dabei der Auswahl der Agenten keinen Einfluss. Genauere Details zur Durchführung der Baseline Vergleiche finden sich in 5.5.1.
Basierend auf den beiden Sieger Agenten (Agent-01 und Agent-02) 5.1, werden nun die Optimierungen auf das Env. und die Agenten angewendet. Mit diesen optimierten Agenten (Agent-01 Optimierung A bis Agent-02 Optimierung C) werden nun die Vergleiche bezüglich jedes einzelnen Evaluationskriteriums 3.2 wiederholt.
Im letzten Schritt soll in der gesamt Evaluation der optimale Agent für jedes Evaluationskriterium ermittelt werden. Dabei können dies auch Baseline Agenten sein.

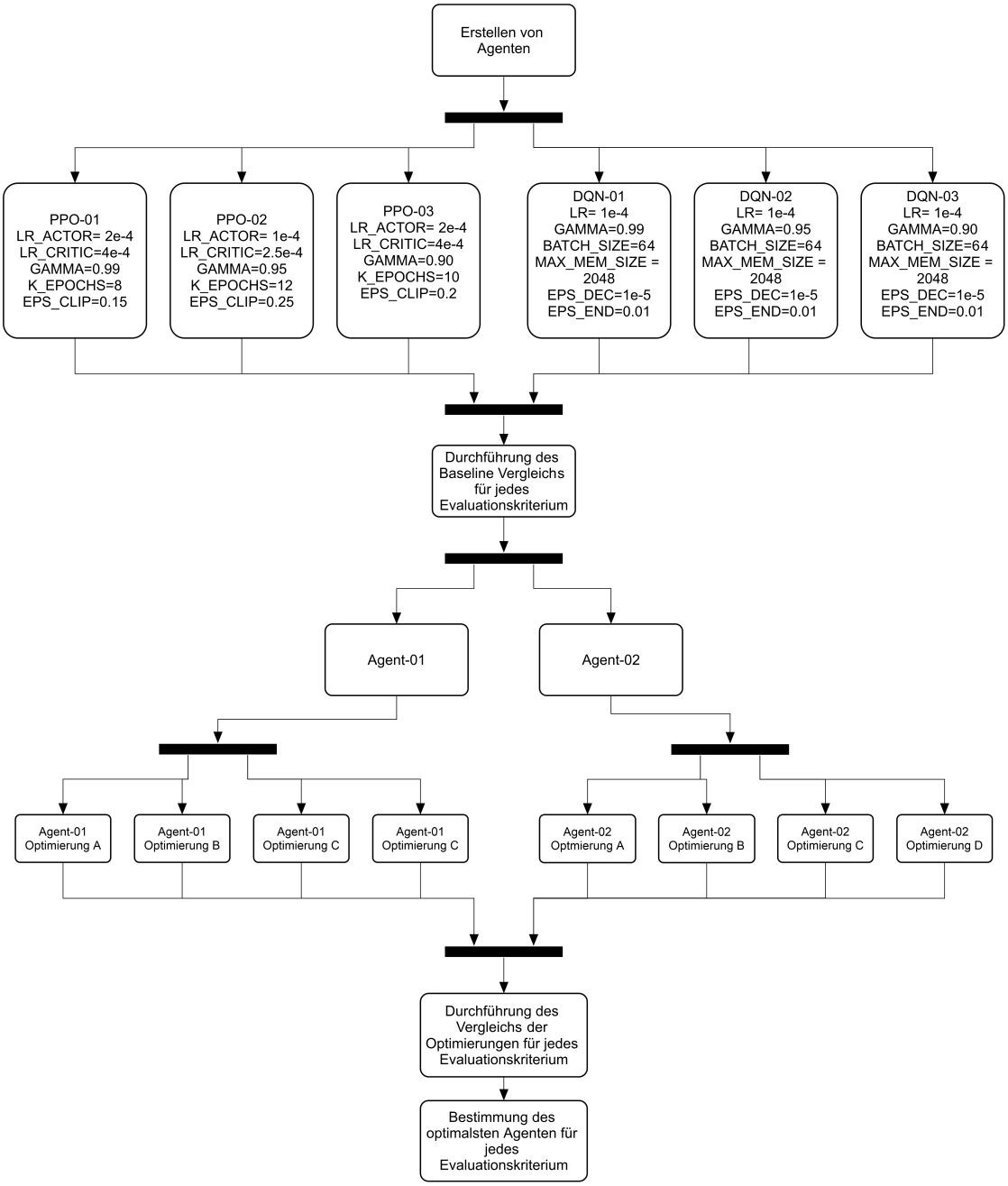

Abbildung 5.1.: Darstellung des Vorgehens.

5.2. Environment

Das Env besteht im wesentlichen aus einer Hauptkomponenten (Wrapper), welche benötigt wird, um die in 3.1.1 aufgestellten Anforderung zu erfüllen. Diese Hauptkomponente beinhaltet die Spiellogik-Komponente, welche aus fünf weiteren Unterkomponenten besteht. Zu diesen gehören die Game-, Player-, Reward-, Observation- und GUI-Komponenten.

5.2.1. Wrapper

Der Wrapper oder auch die Verbindungskomponente verbindet alle bereits erwähnten Komponenten miteinander. Zu diesem Zweck muss er einige Funktionalitäten implementieren. Diese beziehen sich auf die im Wrapper vorhandenen Komponenten.

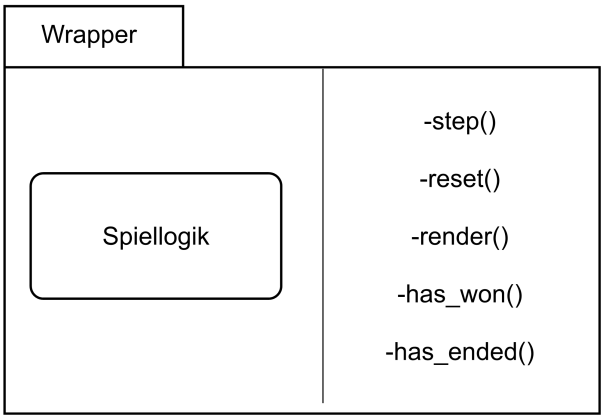

Abbildung 5.2.: Darstellung des Wrappers.

Die step Funktionalität 5.2 ist die Hauptmethode im Wrapper. Sie bekommt eine Aktion übergeben, welche zuvor vom Agent bestimmt wurde. Diese wird mit Hilfe der Spiellogik umgesetzt. Entsprechend der Anforderung 3.1.1 gibt diese Methode nur einen Reward und eine Obs zurück.
Reset setzt den bereits vorhandenen Spielfortschritt zurück, wobei auf die Spiellogikkomponente zugegriffen wird. Die Anforderung 3.1.2 ist damit erfüllt.
Die Render Methode ist für die Visualisierung verantwortlich und ruft Methoden in der GUI-Unterkomponente auf, welche ein genaues Bild von der Spiellogik-Komponente erhält. Die Anforderung 3.1.2 ist damit erfüllt.
Die has_won und has_ended Methodiken geben Statusinformationen über den Momentanen Spielstand zurück, welche für den Spiel- bzw. Trainingsablauf benötigt werden.

5.2.2. Spiellogik

Die Spiellogik besteht aus den fünf Unterkomponenten, welche in 5.2 bereits benannt wurden.
Die Game-Komponente ist die wichtigste Komponente von allen, 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 5.2.2 angegeben. Zuzüglich wird im Rahmen eine der Optimierungen, siehe 5.1, eine weitere Reward Funktion implementiert. Diese berechnet sich wie in 5.4.2 angegeben.

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, view, make_apple.
In der Player-Komponente, welche zur Datenverwaltung dient, 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 (done), 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 5.2.2 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.

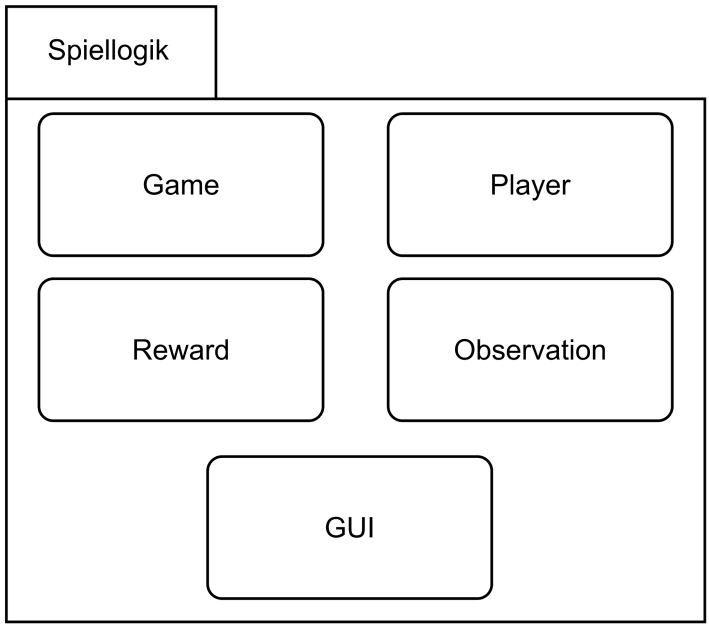

Abbildung 5.3.: Darstellung der Spiellogik mit ihren Unterkomponenten.

Spielablauf

Die eigentliche Aktionsabarbeitung wird durch das Aufrufen der step Funktionalität im Wrapper 5.2.1 bewirkt. Diese ruft die evaluate und observe Methoden auf, welche in der Game-Komponente implementiert sind 5.3. Um jedoch zuerst die Abarbeitung einer Aktion durchzuführen, muss die action Methode als aller erste aufgerufen werden, welche die von Agenten bestimmte Aktion (action) übergeben bekommt.

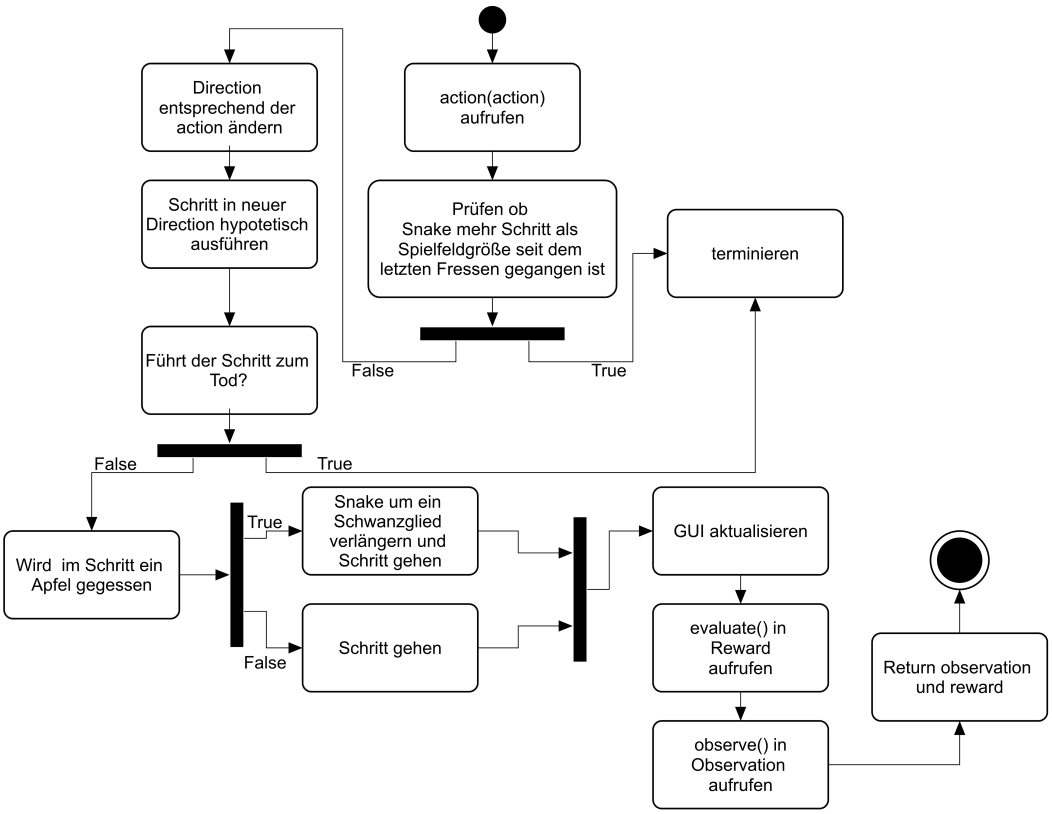

Abbildung 5.4.: Darstellung eines Schrittes in der Spielepisode.

Zu Beginn wird überprüft, ob die Snake seit dem letzten Fressen mehr Schritte als die eigentliche Spielfeldgröße gegangen ist. Die Spielfeldgröße berechnet sich dabei wie folgt (8, 8) = 64, daher ist die Spielfeldgröße 64. Sollte die Snake mehr Schritte gelaufen sein als diese Größe, so wird das Spiel terminiert, da die Snake eventuell in einer Schleife steckt und daher diese nie wieder verlassen würde.
Andernfalls wird die Aktion verarbeitet, in dem sie die direction verändert der Snake manipuliert. Das Spiel Snake besitzt in diesem Konzept drei Actions turn left, turn right oder do nothing.

Tabelle 5.1.: Kodierung der Actions

Action	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 5.1 wird klar, dass die Direktion entweder nur Norden, Osten, Süden oder Westen sein kann. Als nächstes wird ein Schritt der Snake mit der aktualisierten Aktion 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 2.1.
Anderenfalls wird der Schritt durchgeführt. Dabei wird zwischen zwei Fällen unterschieden. Sollte die Snake einen Apfel gefressen haben, also der Kopf der Snake und der 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 Platz 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, durch die direktion definierte, neue Position einnimmt.
Nach der Ausführung einer dieser beiden Fälle, wird die GUI aktualisiert mit der update_gui Methode. Nach diesem Schritt ist die Abarbeitung der action Methode abgeschlossen. Damit jedoch die 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 Method in der Reward-Komponente auf. Sie bestimmt, basierend auf dem letzten Zug, den Reward, was nach folgenden Vorbild geschieht. Der Reward ist abhängig von drei Faktoren. Dem Fressen eines Apfels, dem Sieg und 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 +1.0 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.

Observation

Die Observation, welche von der step Method des Wrappers 5.2.1 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 5.2.2 aufgerufen. Diese ruft 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.
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 dieser größer wäre als er in Realität wirklich ist. Darum werden Felder der AV, die sich außerhalb des Spielfeldes 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 verwandten Arbeiten 4.1 und 4.2, 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 A.3.1.
Ein Nachteil dieser Obs ist jedoch die Unvollständigkeit. Sollte der blaue Punkt in 5.5 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, die Distanzen zu den Wänden und zu, außerhalb der AV liegenden, Schwanzteilen und die Ausrichtung (direction) der Snake, werden durch die AV nur eingeschränkt oder gar nicht geliefert.

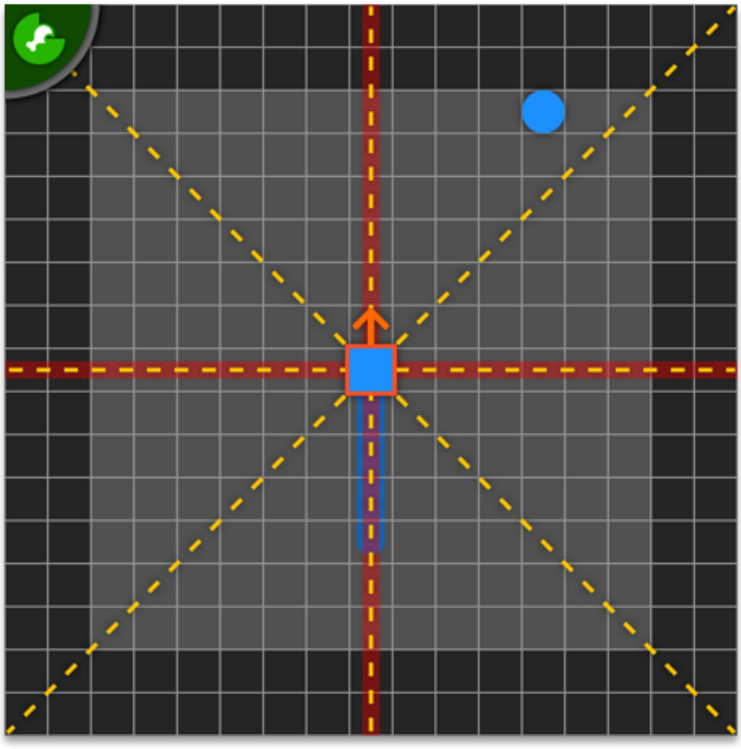

Abbildung 5.5.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar, wobei das rot umrandete Rechteck den Kopf darstellt. Die schwarzen Felder werden nicht von der AV abgedeckt, graue liegen innerhalb der AV. Die gelben gestrichelten Linien stellen ein X-Ray Distanzbestimmung dar. Der blaue Kreis stellt den Apfel dar und der grüne viertel Kreis oben links symbolisiert Hunger.

Aus diesem Grund wurde die AV durch die scalar_obs (SO) ergänzt. Diese beinhaltet skalare Informationen und ist eine Konkatenation aus X-Ray Distanzbestimmung, Hunger- und Blickrichtungsanzeige. 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 (Matrixsicht) befindet.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 ein eindimensionaler Vektor mit einem einzigen Eintrag, welcher sich aus der Differenz zwischen der Anzahl der gegangenen Schritten seit dem letzten Fressen (inter_apple_steps) und der maximalen Schrittanzahl ohne einen Apfel gefressen zu haben (max_steps), berechnet. 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 diese durch eins geteilt. Nähren sich die beiden Werte an, so rückt der 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 X-Rays Distanzbestimmungen verfahren. Bei ihnen handelt es sich um acht Distanzmesserlinien, die in 45° Abständen ausgesandt werden, siehe 5.5. 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 nach Wänden, dem eigenem Schwanz und dem Apfel gesucht. Daher wird die X-Ray Distanzbestimmung in einem Vektor der Größe 24 (3 * 8 = 24) gespeichert.

GUI

Die graphische Oberfläche oder auch GUI genannt kann optional ein oder aus geschaltet werden. Beim lernen der Agenten bietet es sich beispielsweise an diese auszuschalten, da diese den Lernprozess 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.

5.3. Agenten

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

5.3.1. Netzstruktur

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

Im Rahmen dieser Ausarbeitung soll sich 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. Wurden in den Papers von Wei u. a. (2018) und Chunxue u. a. (2019) nur ein einziges großes CNN A.3 genutzt, welche Gefahr läuft viele, für den Spielerfolg, unnötige Informationen zu verarbeiten und eventuelle Probleme mit variablen Spielfeldgrößen zu bekommen, so wird in dieser Ausarbeitung auf ein zweiteilige Netzstruktur gesetzt. Diese besteht aus dem ConvNet und dem Actor-, Critic- oder Q-Head. Zuerst wird die AV durch zwei Convolutional Layer mit einer ReLU Aktivierungsfunktion geleitet. 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 Conv2d Layers. Dies soll den Informationsverlust an den Rändern minimieren. Danach werden allen Feature Maps eine Null-Zeile und Spalte hinzugefügt (Padding), damit beim Max-Pooling unter der Filtergröße und dem Stride von 2x2, auch die letzten Zeile und Spalte verarbeitet wird. Der Tensor besitzt nun die Form

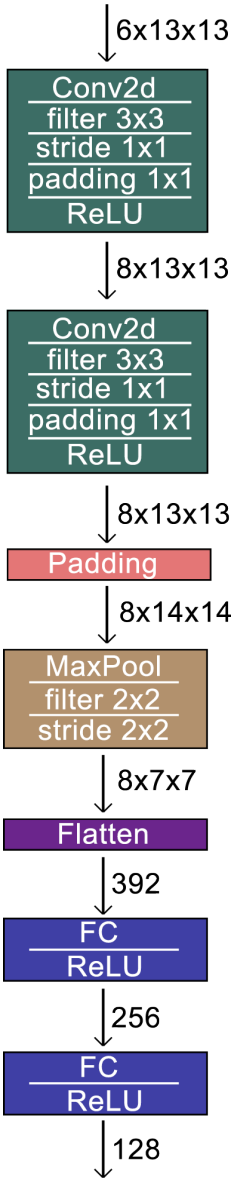

Abbildung 5.6.: ConvNet

(8x14x14). Nach dem max-pooling besitzt der Tensor, Feature Maps der Größe 7x7. Nach einer Einebnung (Flatten) zu einem eindimensionalen Tensor wird, wird dieser durch zwei weitere Fully Connected Layer (FC) mit einer ReLU Aktivierungsfunktion propagiert. Der resultierende Tensor besitzt die Größe 1x128 und ist ein zwischen Ergebnis, da dieser nun mit der SO verbunden wird (Join).

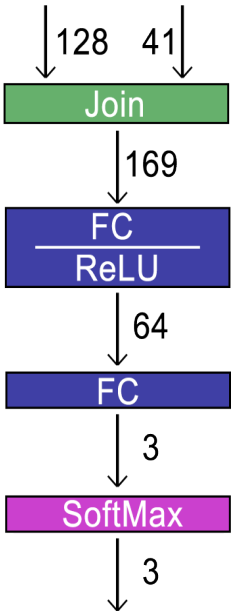

Abbildung 5.7.: Actor-Head

Da das NN in beide Algorithmus-Arten verwendet wird, müssen Netzwerkköpfe für den Actor, Critic und für das Q-Net definiert werden. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nachdem der Joined Tensor (1x169) durch zwei weitere FC Layer mit ReLU Aktivierungsfunktion propagiert wurde, benötigt der Actor des PPO-Agenten eine Wahrscheinlichkeitsverteilung über alle Actions. Daher auch die Ausgabe von einem Tensor der Größe drei. Um diese Wahrscheinlichkeitsverteilung zu erhalten, wird die SoftMax Funktion angewendet, siehe 5.7.
Der Critic des PPO verwendet hingegen den Critic-Head, siehe 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 zu bestimmt 2.3.2, gibt dieser einen Tensor mit einem einzigen Wert zurück. Der Q-Net-Kopf ist in seinem Aufbau sehr ähnlich zu dem Critic-Kopf. 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-Kopf gleich, siehe 5.8.

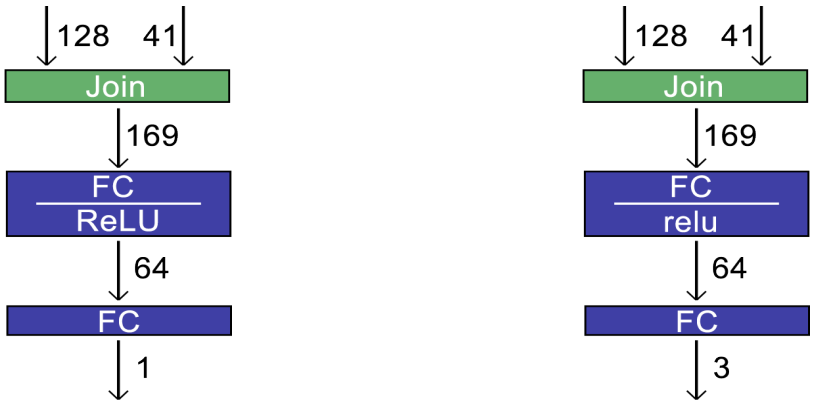

Abbildung 5.8.: Darstellung des Critic-Kopfes (links) und des Q-Net-Kopfes (rechts).

5.3.2. DQN

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

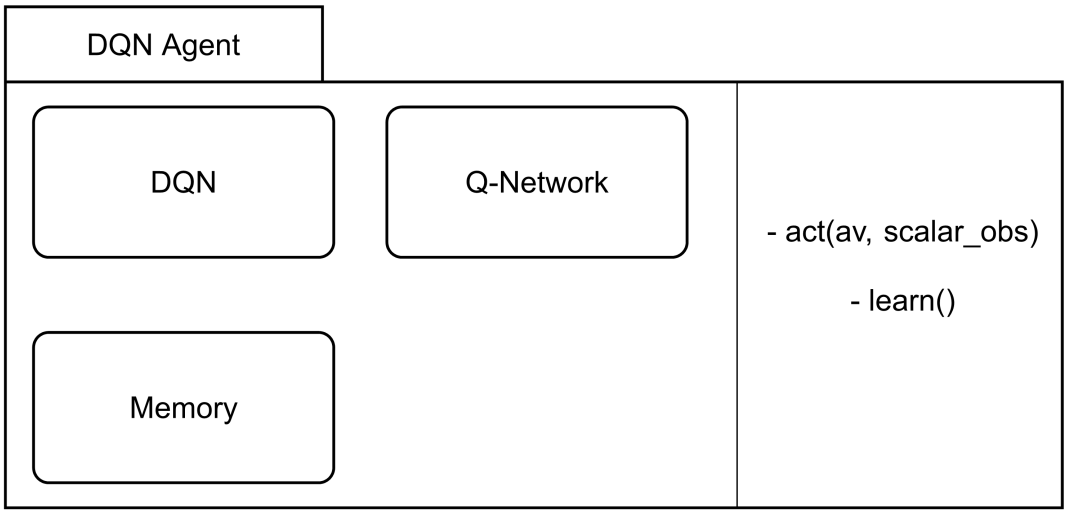

Abbildung 5.9.: Darstellung des DQN-Agent mit seinen Komponenten.

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 (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 mit den neuen überschrieben. Die gespeicherten 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 der act Methode 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 Action wird der act Methode die momentane Obs 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 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 an Stelle 2 steht.
Zum Schluss wir die Aktion zurückgegeben und die Methode terminiert.

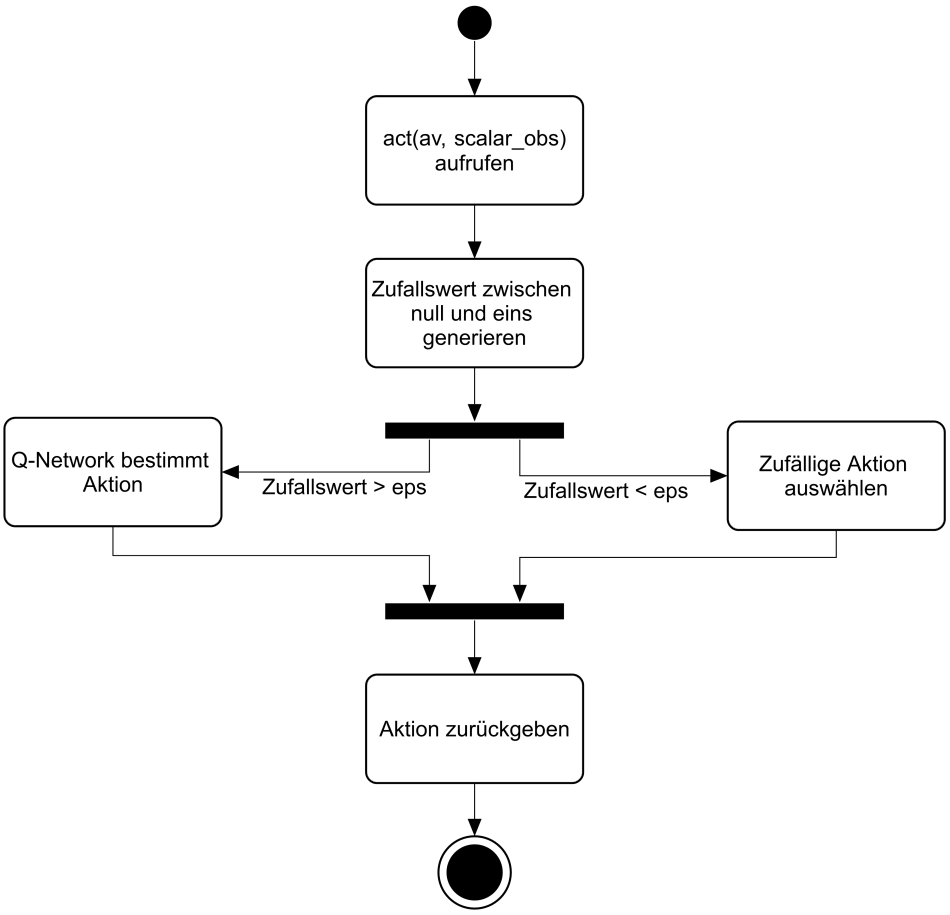

Abbildung 5.10.: Darstellung der Aktionsbestimmung des DQN-Agent.

Lernprozess

Der Lernprozess wird über die learn Methode ausgeführt und stellt sich dabei wie folge 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 2.10, wobei s_i die Observation (AV und SO), a_i die gewählte Aktion im State s_i und θ die Netzwerkparameter des Q-Network, darstellten. Dieser wird als Q-Eval definiert. Als nächstes 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 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 2.10. Dieser Wert wird als Q-Target definiert und entspricht Q-Eval. Am Ende wird der Mean Squared Error zwischen den Q-Targets und Q-Evals aus dem Mini-Batch gebildet. Auf Basis dieses Fehlers soll das Q-Network mittels Backpropagation und Gradientenverfahren A.1 angepasst werden.

5.3.3. PPO

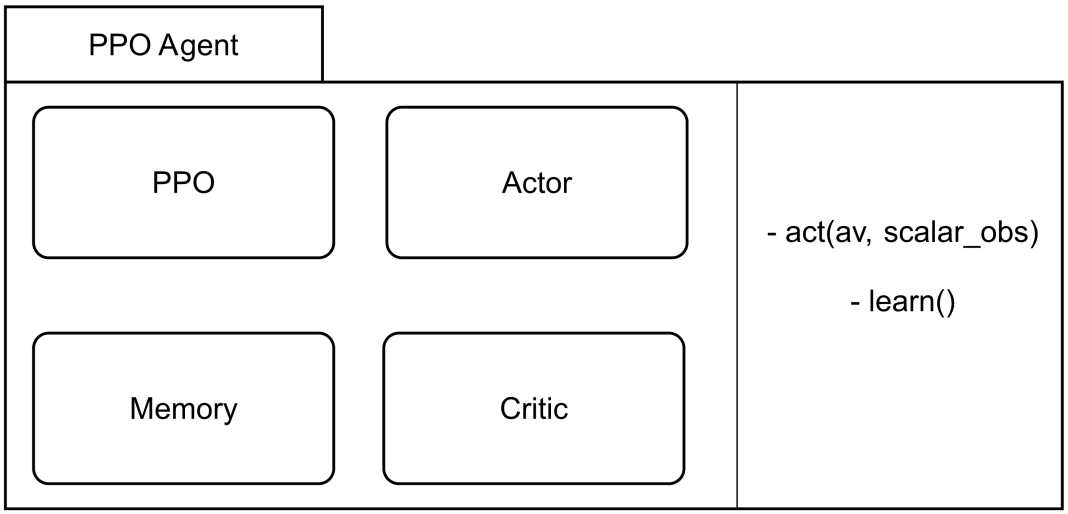

Abbildung 5.11.: Darstellung des PPO-Agent mit seinen Komponenten.

Der PPO Algorithmus und damit auch die Agenten, welche auf diesem Algorithmus basieren, 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) 2.3.2, 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 des 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 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 erreicht 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 2.8.

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird durch die act Methode in der PPO-Komponente angestoßen, welche die around_view und die scalar_obs 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, 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.
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 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 entstanden sein, so wird dieser auf null gesetzt, da der Return der diskontierten Summe aller Rewards bis zum Ende der Spielepisode entspricht. Nach einem terminalen Zustand werden keine weiteren zustände besucht, sodass keine neuen Rewards gesammelt werden können und daher der Return gleich null ist.
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.
Als nächstes werden die logarithmierte Wahrscheinlichkeiten für die gespeicherten Aktionen a_i $\pi_\theta(a_t|s_t)$ bestimmt 2.3.2. Dazu werden die aus dem Memory entnommenen AVs (around_view) und SOs (scalar_obs) durch das Actor- und Critic-NN propagiert. Anschließend werden die logarithmierte Wahrscheinlichkeiten für die Aktionen bestimmt und zusammen mit den Baseline Estimates 2.3.2 und den Entropien der Wahrscheinlichkeitsverteilungen 2.3.2 zurückgegeben.
Daraufhin werden die Probability Ratios aus der eben bestimmten logarithmierte Wahrscheinlichkeiten und den alten logarithmierte Wahrscheinlichkeiten des Memories bestimmt 2.3.2. Nachfolgend werden die Advantages durch Subtraktion der Returns mit den Baseline Estimates berechnet $\hat{A}_t(s,a) = R_t - b(s_t)$ 2.3.2.
Des Weiteren werden als nächstes 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 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. Um zum Schluss den gesamt Loss des PPO zu bestimmen zu können, 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 Entropy-Loss bestimmt 2.3.2. Diese wenden dann alle zusammengerechnet, entsprechend der Formel: $L_t^{\text{PPO}}(\theta) = L_t^{\text{CLIP + VF + S}}(\theta) = \hat{\mathbb{E}}_t[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}} + c_2 S[\pi_\theta](s_t)]$ 2.3.2.
Das Actor- und Critic-NN werden mit dem Loss unter Zuhilfenahme von Backpropagation und Gradientenverfahren aktualisiert.

5.3.4. Vorstellung der zu untersuchenden Agenten

Ein zentraler Aspekt eines Vergleiches von verschiedenen RL-Agenten ist die genaue Definition dieser einzelnen. Basierende auf den Grundlagen 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.

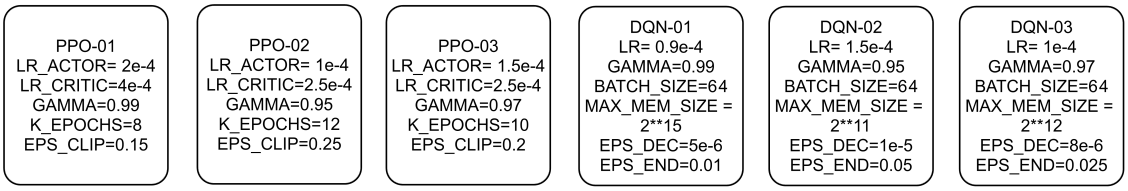

Abbildung 5.12.: Darstellung der zu untersuchenden Agenten.

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. Mit einem K_EPOCHS-Wert von acht wird das Lernen weiter verlangsamt und verstetigt. 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 des Actors von 1e-4 und des Critics von 2.5e-4 gewählt, jedoch sorgt relativ großer 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 des Actors und Critics von 1.5e-4 bzw. 2.5e-4 sollte ein schneller und zugleich stetiger Lernfortschritt erzielt werden. Der GAMMA-Wert von 0.97 soll als Kompromiss zwischen Kurz- und Weitsichtigkeit dienen. 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 mittleren LR on (LR = 0.9e-4) und einem großen GAMMA-Wert von 0.99, wird ein stetiges und zukunftsorientiert Lernen bestärkt. Eine BATCH_SIZE und Memory-Size (MAX_MEM_SIZE) 64 und 2**15, soll zudem das Lernen verlangsamen. Geringe Werte für Eps_Dec und

EPS_MIN von 5e-6 und 0.01 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 1.5e-4 in Verbindung mit einem mittleren bis kleinen Wert für Gamma von 0.95 soll einen schnellen Lernfortschritt generieren. Dies wird durch die niedrige Memory-Size (MAX_MEM_SIZE) von 2**11 und die große Epsilon-Dekrementierung (EPS_DEC) von 1e-5 verstärkt. Auch sorgt der verhältnismäßig große Wert für EPS_MIN von 0.05 für eine schnellerer Exploration und damit für ein schnelles Lernen.

Der DQN-03 besitzt die folgenden Hyperparameter, LR = 1e-4, GAMMA = 0.97, BATCH_SIZE = 64, MAX_MEM_SIZE = 2**12, EPS_DEC = 8e-6 und EPS_END = 0.025 und ist mit dieser Kombination an Parametern ein mittelfristiger Lerner.

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 vier Optimierungen auf die Baseline Agenten (Agenten ohne Optimierungen) angewendet werden, welche aus einer verwandten Arbeit 4.1 und eigenen Ideen stammten.

Die Optimierungen welche aus dem Paper „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ Chunxue u. a. (2019) konnten bezüglich ihrer Qualität nicht überzeugen. Das aper hat als Optimierung eine neue Reward Funktion vorgestellt die sehr einfach gehalten ist und wenige Spielfaktoren mit in die Berechnung des Rewards mit einfließen lasst sodass zwei Optimierungen aus dem Paper „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ Wei u. a. (2018) für den folgenden Vergleich ausgewählt wurden.

5.4.1. Optimierung A - Dual Experience Replay

Die Idee für Dual Experience Replay oder auch hier Splited Memory genannt, stammt aus der Arbeit „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ Wei u. a. (2018) und wurde in 4.1 bereits erwähnt.
Diese Optimierung zielt darauf ab, den Replay Buffer (Memory) zu zweiteilen, sodass ein Mem1 und Mem2 entstehen. In Mem1 werden ausschließlich Erfahrungen

gespeichert, welche einen Reward vorweisen können, der größer als ein vordefinierter Grenzwert ist. In Mem2 werden alle übrigen Erfahrungen gespeichert. Diese Aufteilung zielt darauf ab, dass zu beginn ein größerer Anteil an guten Erfahrungen für das Lernen verwendet wird, um das Lerntempo und die Stabilität zu erhöhen. Den Verfassern schwebt ein Verhältnis von (80% guten und 20 % schlechteren Erfahrungen vor). Dieses Verhältnis normalisiert sich über die Trainingszeit, daher zum Verhältnis von (50% zu 50%).
Problematisch ist jedoch die erfahrungsorientierte Aufteilung, da diese für den PPO nur schlecht umsetzbar ist.
Alternativ kann die Sortierung nicht auf Erfahrungsbasis, sondern Episoden basiert geschehen. Es werden daher die besten Spielepisoden, gemessen an ihren Scores, in die Memories einsortiert.
Zu diesem Zweck wird eine weitere Memory-Art in den Memory-Komponenten des DQN und PPO definiert, welchen diese beschriebene Funktionalität bereitstellt. Diese trägt den Namen Splited-Memory
Beim DQN Memory wird zusätzlich darauf geachtet, dass beim Einfügen einer ganzen Episode an Erfahrungen die Ring-Buffer Eigenschaften des Memory erhalten bleiben. Beim Memory des PPO ist dies nicht nötig.

5.4.2. Optimierung B - 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 in 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 aus dem Memory gelöscht, was die Training Gap Strategy und den zweiten Teil der Reward Funktion darstellt. Zur Verstärkung des Pathfindings wird die Timeout Strategy angewendet, welche den Agenten für nicht zielgerichtetes Verhalten, wie z.B. das unnötige Umherlaufen, bestraft.
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: $r_{res}(dis, len, size, steps) = r_{distanz}(dis, len, size) - r_{timeout}(steps, len)$.

5.4.3. Optimierung C - 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.

5.4.4. Optimierung D - Parameteraddition zu den Netzwerken

Die vierte Optimierung versucht durch das Hinzufügen von Parametern zu den Netzwerken des Actors, Critics und des Q-Networks die Performance zu steigern. Konkret sollen die Parameter der Netzwerke ansatzweise verdoppelt werden. Natürlich bietet sich dies nicht an jeder Stelle der Netzstruktur an, daher werden nur bestimmte Layer des NN von der Optimierung betroffen sein. An der Struktur des NN wird jedoch nichts verändert. Die optimierte Netzstruktur ist im Anhang A.3 dargestellt.

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 Methde 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 (N_ITERATIONS) für einen Trainingslauf gespielt werden. DieSpielfeldgröße soll dabei standardmäßig bei (8x8) liegen.
Bei der Datenerhebung ist streng zwischen Spiel- und Trainingsdaten zu unterscheiden, wobei die Spieldaten 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 5.3.2 bzw. PPO-Lernprozess 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 Spieldaten findet annähernd auf die gleiche Weise statt. Jedoch werden die Daten in nicht nach jedem Trainingslauf entnommen, sondern nach jeder Spielepisode. Bei den Daten handelt es sich um die gleichen wie in der oben erwähnten Datenerhebung für Trainingsdaten. Diese werden dann ebenfalls der Statistik-Komponente übergeben 5.5.2.

5.5.2. Datenverarbeitung und Erzeugung von Statistiken

Nach der Erstellung der Trainings- und Spieldaten für jeden Agenten, wenden diese Daten entsprechend der Evaluationskriterien verarbeitet 3.2. Diese Verarbeitung sieht wie folgt aus.
Die Performance wird anhand der gefressenen Äpfel gemessen. Das Datenobjekt apples besitzt diese Daten für jeden Trainingsdurchlauf.
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.
Die Robustheit wird mit Hilfe von Spieldaten bestimmt. Dabei wird ein trainierter Agent auf einem Spielfeld mit variabler Größe spielen. Die Robustheit wird danach gleich der Performance gemessen. Es werden daher die gefressenen Äpfel (apples) in

der neuen Umgebung gemessen. Die Spielfeldgröße soll dabei zwischen (6x6) bis zu (12x12) liegen. Für die Erhebung werden wie beim Trainingsprozess 30.000 Spiele gespielt.
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.
Die Spielzeit berechnet sich aus der vergangenen Zeit für ein Trainingsspiel pro Trainingsspiel.

Da bei 30.000 gespielten Spielen für jedes Evaluationskriterien gleich viele Werte entstehen, jedoch die Statistik nur eine gewisse Bildgröße besitzen kann, sollen jeweils 10 dieser errechneten Werte zu einem, mit Hilfe der Durchschnitts-, Max- und Min-Funktion, zusammen gefasst werden. Dies soll die Lesbarkeit der Grafiken verbessern und Schwankungen minimieren. Die Statistiken besitzen daher 3.000 (30.000 / 10 = 3.000) Datenpunkte.
Diese Datenpunkte werden für alle Agenten eines Vergleiches pro Funktion (Durchschnitts-, Max- und Min-Funktion) aufgetragen. Es entstehen daher pro Datensatz aller Agenten drei Grafiken welche die Minima, Maxima und den Durchschnitt der letzten 10 Datenpunkte der Evaluationskriterien wiedergibt.
Eine nähere Darstellung wird im Kapitel Implementierung geliefert 6.

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 sowohl von den Agenten als auch vom Snake-Environment 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.

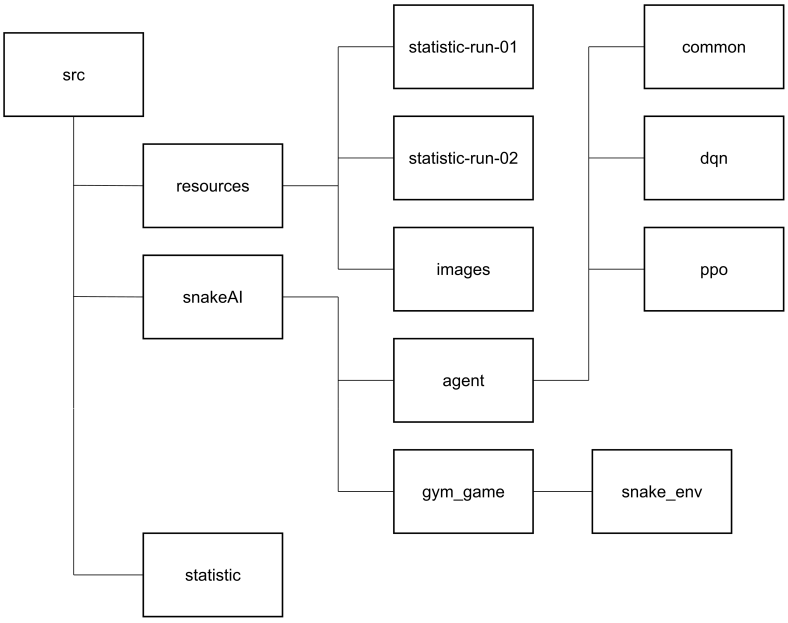

Abbildung 6.1.: Darstellung der Package Struktur.

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 5.2.2) 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 Schrittanzahl überschnitten hat. Danach werden step_counter und p.inter_apple_steps inkrementiert und die Aktion wird umgesetzt. Um dies umzusetzen wird die directi-

on im Player Objekt angepasst, entsprechend der Beschreibung im Konzept (siehe 5.2.2). 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 5.2.2). 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 5.2.2) 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()
```

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.

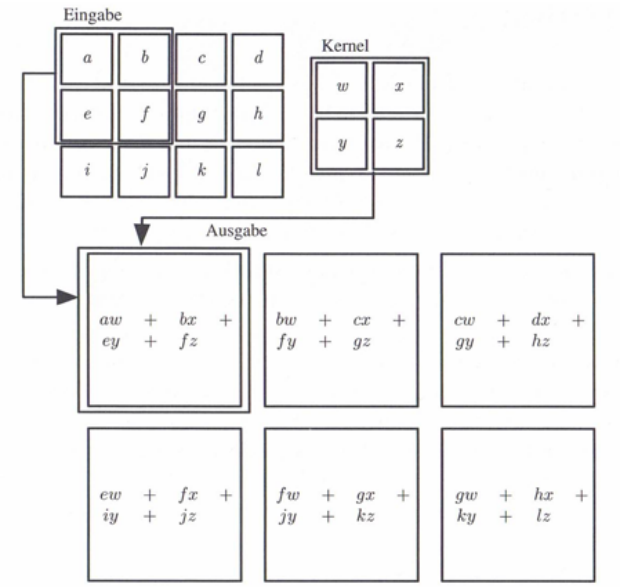

Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

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.)

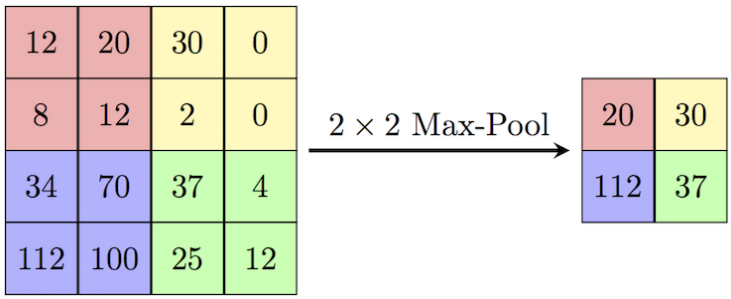

Abbildung A.2.: Darstellung des Max-Poolings. `https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png`

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`

A.4. Optimierte Netzwerke

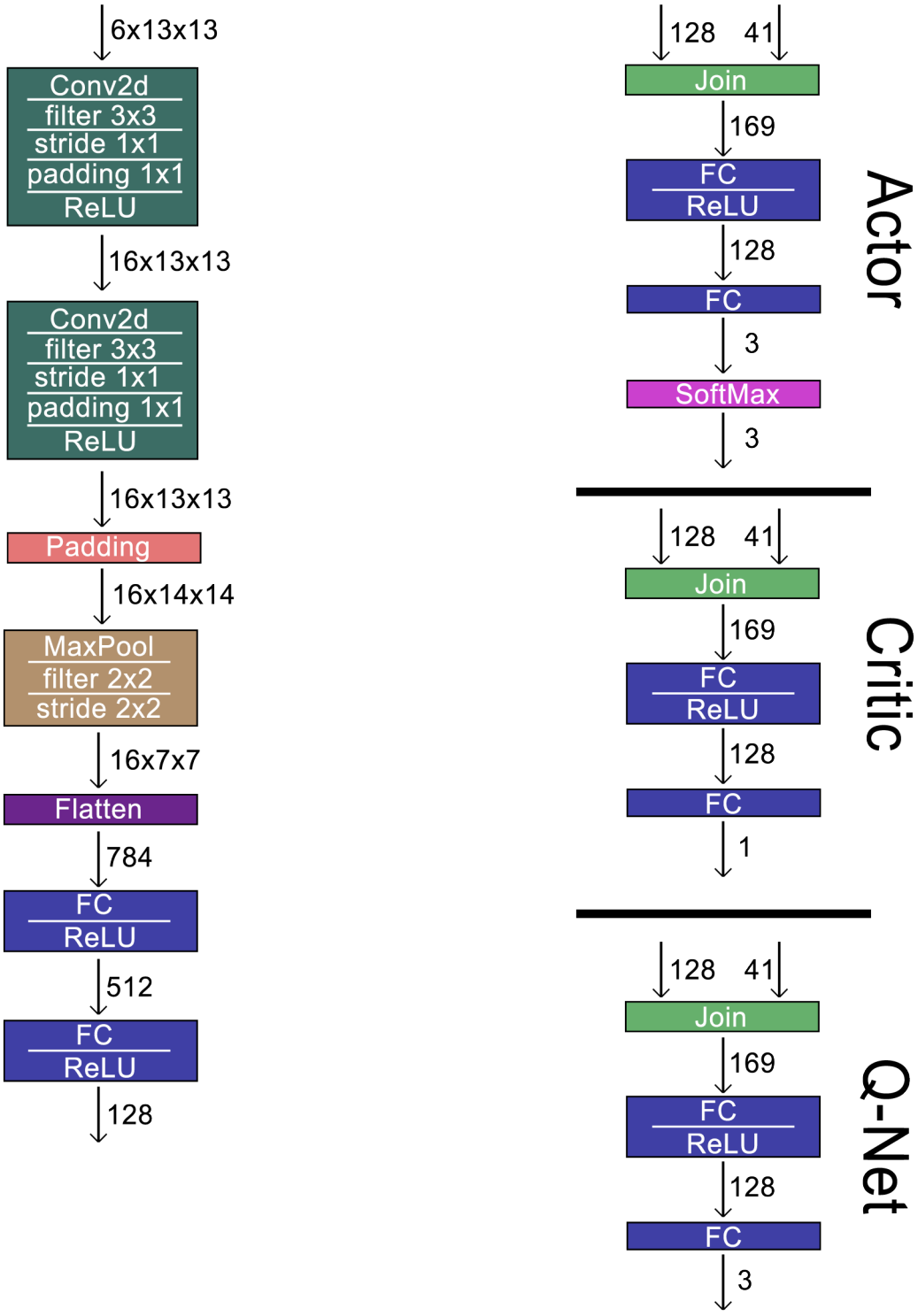

Abbildung A.3.: Optimierte Netzstruktur

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 5.5) 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
```

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) ⟶ Die Trainingsdaten und as NN-Model werden unter "./src/resources/statistic-run-0"+ STATISTIC_RUN_NUMBER gespeichert.

10. AGENT_NUMBER (int) ⟶ Nummer des zu untersuchenden Agenten. Die zu speichernden Daten werden in den Dateien "./statistic-run-01/PPO-0"+ AGENT_NUMBER + "train.csv"bzw. "./statistic-run-01/PPO-0"+ AGENT_NUMBER + "train.model"gespeichert.

11. 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 1 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) ⟶ Die Trainingsdaten und as NN-Model werden unter "./src/resources/statistic-run-0"+ STATISTIC_RUN_NUMBER gespeichert.

11. AGENT_NUMBER (int) ⟶ Nummer des zu untersuchenden Agenten. Die zu speichernden Daten werden in den Dateien "./statistic-run-01/DQN-0"+ AGENT_NUMBER + "train.csv"bzw. "./statistic-run-01/DQN-0"+ AGENT_NUMBER + "train.model"gespeichert.

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

C.3. Test Startargumente

Da die Test Methoden 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 (Int) ⟶ Anzahl l der zu spielenden Spiele.
4. BOARD_SIZE (Tuple of Ints) ⟶ Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
5. STATISTIC_RUN_NUMBER (Int) ⟶ Die Testdaten werden unter "./src/resources/statistic-run-0"+ STATISTIC_RUN_NUMBER gespeichert.
6. AGENT_NUMBER (int) ⟶ Nummer des zu untersuchenden Agenten. Die zu speichernden Daten werden in der Datei "./statistic-run-01/(DQN bzw. PPO)-0"+ AGENT_NUMBER + test.csv"gespeichert.
7. 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_File\Bachelor-Snake-AI\src\main_test.py "PPO" "Path_to_Model" 30000 "(8, 8)" 1 2 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