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, 29. Juli 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

KI	**K**ünstliche **I**entelligenz
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 zumehmenden 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 Situaion, 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))
Somit kann der RL-Vergleich am Spiel Snake möglicherweise helfen, Menschen zu retten.

1.2. Zielsetzung

Ziel der Arbeit ist es für die Umgebung Sanke einen optimalen Agenten zu bestimmen, welcher spezifische Anforderungen erfüllen soll. Damit ergib sich die folgende Forschungsfrage:
Wie kann an einem Beispiel des Spiels Snake für eine nicht-triviale gering-dimensionale Umgebung ein möglichst optimaler RL Agent ermittelt werden?

Basierend auf der Forschungsfrage ergibt sich ein Mehrwert für die Wissenschaft. Durch die 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.

2.1. Game of Snake

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.

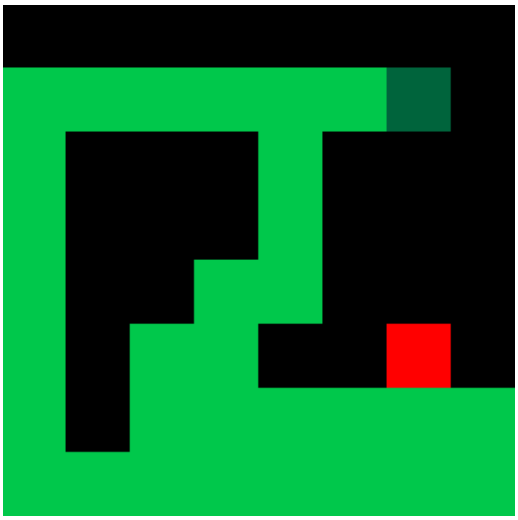

Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

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, da können wir heute bereits Roboter mit RL Agenten ausstatten, 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 RL 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 voralledem 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 üermittelt 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 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

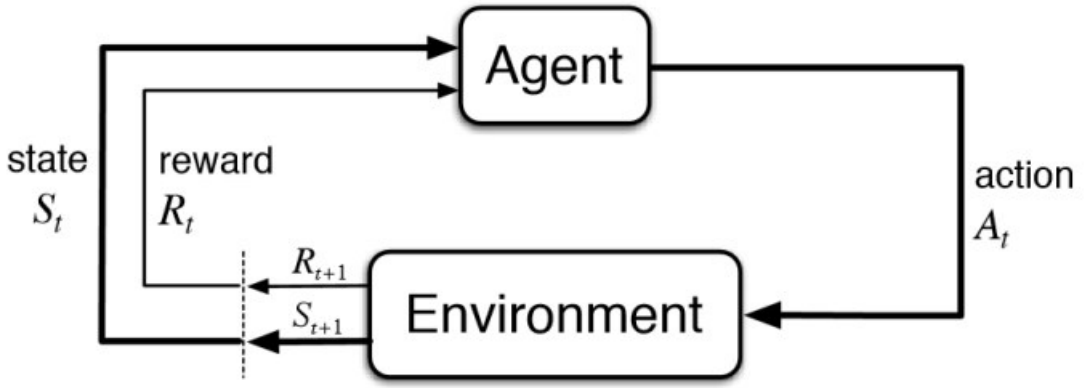

Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State S_t und falls $t \neq 0$ einen Reward R_t. Daraufhin wird vom Agenten eine Action A_t ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State S_{t+1} und Reward 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 folgenen 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 größerern Environments. wäre viel zu komplex und zu resourcenbindend. 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 bestimmt. 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).

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 $L_t^{\text{CLIP}}(\theta)$, 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).

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

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 Actions bzw. 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 $L_t^{\text{CLIP}}(\theta)$ 2.2 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 $L_t^{\text{CLIP}}(\theta)$ 2.2 zur Vervollständigung der PPO Main Objective Function. In normalen Policy Gradient Methoden bestehe ein Problem zwischen der effizienten Datennutzung und dem Updaten der Policy. Dieses Problem tritt z.B. im Zusammenhang mit dem Advantage Actor Critic (A2C) Algorithmus auf und reglementiert das effiziente Sammeln von Daten. So ist des dem A2C nur möglich von Daten zum lernen, welche on-policy (unter der momentanen Policy) erzeugt wurden. Das Verwenden von Daten, welche unter einer älteren aber dennoch ähnlichen Policy gesammelt wurden, ist daher nicht zu empfehlen. Der PPO bedient sich jedoch eines Tricks der Statistik, dem Importance-Sampling (IS, deutsch: Stichprobenentnahme nach Wichtigkeit). Wurde noch beim A2C mit folgender Formel der Loss bestimmt (Lapan, 2020, S. 591):

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

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

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

Die Daten können dabei nun mittels $\pi_{\theta_{\text{old}}}$, partiell off-policy, bestimmt werden und nicht wie beim A2C direkt on-policy. (Schulman, 2017, Zeitpunkt: 9:25)
Nun können generierte Daten mehrfach für Updates der Policy genutzt werden,

was die Menge an Daten, welche nötig ist, um ein gewisses Ergebnis zu erreichen, minimiert. Der PPO Algorithmus ist durch die Umstellung auf die Probability Ratio effizienter in der Nutzung von Daten geworden.

Surrogate Objectives

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

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

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

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

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

Zusammenfassung der PPO Training Objective Function

Insgesamt ist der Actor-Loss 2.2 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:

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

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 2.1:

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

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 $\pi_{\theta_{\mathrm{old}}}(s,a)$, Reward r, Folge-State s'. $(s, a, \pi_{\theta_{\mathrm{old}}}(s,a), r, s') \longrightarrow EB$
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 $(S, A, \{\pi_{\theta_{\mathrm{old}}}\}, R, S') \longleftarrow EB$.
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. 2.1
11. Update die Gewichte des Actors und Critics. $\theta_{\mathrm{old}} \longleftarrow \theta$ und $w_{\mathrm{old}} \longleftarrow w$
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 Agent, 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.
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 einer Loss-Function :

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}\Big[(y_i - Q(s, a; \theta_i))^2\Big] \tag{2.10}$$

wobei $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\big[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\big]$ ist Mnih u. a. (2015). Wie auf den ersten Blick erkannt werden kann ist diese Loss Formel nicht für den allgemeinen Gebrauch geeignet. Daher soll nur oberflächlich thematisiert werden.
Die Formel 2.10 besagt, dass der Loss eines zufällig ausgesuchten State-Action Tuples (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):

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

Es wird erkenntlich, dass $Q(s, a; \theta)$ dem Q-target y_i $Q(s, a; \theta) \longrightarrow y_i$ 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. Zur besseren Anwendbarkeit haben Volodymyr Mnhi et al. einen Algorithmus entworfen, welcher das Q-Learning anschaulich erklärt. Da jedoch in diesem Algorithmus weiterhin auf Teile der Loss-Function eingegangen wird, folge nun eine bereinigte Version, welche sich auch für den allgemeinen Gebrauch anbietet (Lapan, 2020, S. 149 f.).

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für (s, a) und $Q(s', a')$ zufallsbasiert. Setzt 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'. $(s, a, r, s') \longrightarrow RB$

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(s,a) durch SGD (Stochastischen Gradientenabstiegsverfahren, Englisch: Stochastic Gradient Descent)

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.

2.5. 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.

Kapitel 3

Anforderungen

In dem letzten Kapitel 2, wurden die Grundlagen für den weiteren Vergleich, der Deep Reinforcement Learning Algorithmen, gelegt. Es wurde daher entschieden, dass der Vergleich exemplarisch an zwei RL Agenten unterschiedlicher Gattungen durchgeführt werden soll. Dies erlaubt eine Abstraktion des Vorgehens auch für andere Agenten.
Problematisch an einem solchen Vergleich ist die Objektivität. Wie kann also sicher gestellt werden, dass die Ergebnisse vergleichbar sind?
Die Antwort auf diese Frage sind feste Anforderungen bzw. Rahmenbedingungen, die bei den einzelnen, am Vergleich beteiligten, Teilen gelten müssen. Dabei gibt es verschiedene Anforderungen an unterschiedliche Systembereiche. In dieser Ausarbeitung existieren es vier Anforderungsbereiche, welche Anforderungen an die folgenden Bereiche stellen:

- Spielimplementierung
- Agenten
- statistische Datenerhebung
- Evaluation

3.1. Anforderungen an das Environment

Die Anforderungen, welche an eine Reinforcement Learning Environment gestellt werden, sind vielseitiger Natur. Sie stammen unter anderem aus der Softwaretechnik, aus dem RL und aus dem gesundem Menschenverstand.

3.1.1. Kommunikation

Damit keine vollkommene Abtrennung des Environments jegliches Interagieren unmöglich macht, sollen nach dem Vorbild aus 2.2.1 drei Kommunikationskanäle implementiert

werden. Das Env. soll in der Lage sein, Actions zu empfangen und Observation und Rewards zu senden.

3.1.2. Funktionalität

Die Funktionalität des Environments ist in drei Unterfunktionalitäten aufzuteilen.

Tabelle 3.1.: Funktionalität des Environment

Funktionalität	Erklärung
Step	Step steht im Sachzusammenhang für das Gehen eines Schrittes im Env. Es beschreibt daher die Umsetzung der von Agenten bestimmten Actions.
Reset	Um den Fortschritt einer Spielepisode wieder zu löschen, ist das Vorhandensein einer Reset-Funktionalität erforderlich.
Render	Render beschreibt die Darstellung des Spielgeschehens auf visueller Basis. Näheres dazu unter 3.1.3

3.1.3. Visualisierung

Um den Spielfortschritt besser evaluieren zu können und aus Gründen der Ästhetik soll eine graphische Oberfläche implementiert werden. Diese soll das Spielgeschehen graphisch wiedergeben.

3.1.4. Test

Um zu garantieren, dass das Env. voll funktionsfähig ist sollen dieses mit Tests überprüft werden. Dies dient der Sicherheit, da Fehler im Env. zu Fehlern im Entscheidungsfindungsprozess führen können, wie das Hide and Seek Environment von OpenAI zeigt Baker u. a. (2019).

3.2. Anforderungen an die Agenten

Neben dem Environment gelten für die Agenten ebenfalls spezielle Anforderungen, welche zum einem objektiven Vergleich führen sollen. Auch sollen Anforderungen an die Implementierung gestellt werden, um Standards einzuführen.

3.2.1. Funktionalität

Reinforcement Learning Agenten können ein großes Repertoire an Funktionen besitzen. Zu den wichtigsten Funktionalitäten gehören das Spielen, daher das Bestimmen

von Actions und das Lernen, daher die Anpassung des NN, um bessere Actions zu wählen, was dann zu optimalen Resultaten führt.

3.2.2. Parametrisierung

Um den Vergleich der verschiedenen Systeme mit dem größtmöglichen Maß an Objektivität zu vollziehen, ist es wichtig, dass die Einzigartigkeit eines Agenten nicht alleine durch die Art (PPO oder DQN) definiert wird, sondern auch durch die dem Agenten zugehörigen Parameter. Siehe 5

3.2.3. Test

Zur Prüfung der Funktionalität der einzelnen Agenten, sollen Tests implementiert werden, welche im besonderen das Lernen und die Aktionsbestimmung abdecken sollen. Dies dient zum Ausschluss von Fehlern.

3.3. Anforderungen an die Datenerhebung

Auch die statistische Datenerhebung soll durch Anforderungen konkretisiert und definiert werden. Um die Objektivität sicherzustellen, werden dafür Anforderungen bezüglich der Durchführung, der Datenspeicherung und den Spezifikationen der Datenerhebung erhoben.

3.3.1. Mehrfache Datenerhebung

Um die Validität der zu erzeugenden Trainingsdaten zu garantieren, soll für jeden Agenten die Datenerhebung mehrmalig durchgeführt werden. Dies trägt nicht nur zur Objektivität bei, sondern sichert das Ergebnis ebenfalls auch vor Schwankungen ab, welche beim Reinforcement Learning natürlicherweise auftreten, bedingt durch die zufälligen Spielverläufe des Environment.

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 die schnell riesige Dimensionen annimmt, sollen stellvertretend nur die Daten ganzer Spiele gespeichert werden. Dies ist ein Kompromiss zwischen Vollständigkeit und effizientem Speicherplatzmanagement.
Folgenden Daten sollen für die Auswertung erhoben und gespeichert werden:

Tabelle 3.2.: 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.
scores	Die gesammelten und aufsummierten Rewards, welche der Agent in einem Spiel gesammelt hat. Auch dieser Datenwert gibt Aufschluss über den Lernerfolg. Gleichzeitig lässt sich den Scores auch noch die Effizienz der Lösung und damit des Lernens entnehmen.
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.
Learning Rate (lr)	einmalig zu speichernder skalarer Wert, welcher ein Maß für die Anpassung der Gewichte des bzw. der NN wiedergibt.
board_size	einmalige zu speichernder vektorieller Wert. Dieser gibt die Feldgröße des Environments an. Dieser ist wichtig für die Evaluation der Robustheit der Agenten.
andere Hyperparameter der Agenten	Die Agenten werden maßgeblich durch die ihnen zugrundeliegenden Paramente gesteuert. Um einen Vergleich verschiedenen Agenten durchführen zu können, sind daher die Hyperparameter von größter Wichtigkeit.

3.3.3. Variation der Datenerhebungsparameter

Um die Möglichkeit auszuschließen, dass die momentan ausgewählten Parameter, wie z.B. Feldgröße, Lernrate, Reward-Funktion usw., für einzelne Agenten vorteilhafte Bedingungen hervorbringen können, sollen die einzelnen Agenten unter verschiedenen Modifikationen getestet werden. Dazu soll die Datenerhebung unter der Anwendung einer Modifikation durchgeführt werden. Zu den Modifikationen zählen:

Variation der Reward-Funktion

Durch das Verändern der Reward-Funktion soll überprüft werden, ob die Agenten unterschiedlich gut auf eine andere Reward-Funktion reagieren. Sollte die Reward-Funktion eins $R_1()$ für den Agenten eins A_1 überdurchschnittlich bessere Resultate liefern, so wird dies erkannt und in der Evaluation berücksichtigt.

Variation der Netzstruktur

Jeder Agent ist abhängig von einer ganzen Reihe an Faktoren, welche über Erfolg und Misserfolg entscheiden. Die Netzstruktur ist dabei einer der besonders wichtigen Faktoren. Je nach Variation der Netzstruktur, kann dies die Qualität eines Agenten entscheidend beeinflussen. Es ist daher angebracht, die einzelnen Agenten unter verschiedenen Netzstrukturen laufen zu lassen, um sicherzustellen, dass die vorherrschende Netzstruktur für alle Agenten jeweils vorteilhafte ist.

Variation der Observation

Eine Überprüfung der Agenten, neben den bereits genannten Variationen, ist auch noch mit der Variation der Observation durchführbar. Diese Veränderung soll Aufschluss über die Qualität der Agenten bei unterschiedlichen Eingabeinformationen liefern. Auch wir so wieder die Wahrscheinlichkeit gesenkt, dass gerade diese Observation für einen spezifischen Agenten vorteilhaft ist.

3.4. Anforderungen an die Evaluation

Bei der Evaluation soll der optimale Agent bestimmt werden. Dabei stellt sich jedoch die Frage, was optimal im Sachzusammenhang bedeutet. Daher sollen die Agenten unter verschiedenen Gesichtspunkten evaluiert werden, um ein möglichst großes Spektrum an Kriterien abzudecken. Einige dieser Kriterien stammen aus der verwendeten Literatur 4 wie z.B. die Performance und Spielzeit die anderen ergeben sich aus den zu speichernden Daten. Die einzelnen Kriterien lauten:

Tabelle 3.3.: 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 erreichen und zu fressen? Hierbei werden beispielsweise Probleme, wie das im Kreis laufen sichtbar.
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?
Trainingszeit	Welcher Agent schafft es ein festes Ziel in der geringstmöglichen Zeit zu erreichen oder welcher Agent ist als erstes in der Lage durchschnittlich 10 Äpfel zu fressen.
Spielzeit	Welcher Agent schafft es am längsten ein durchzuhalten bevor ein terminaler Zustand erreicht wird? Auf Snake bezogen: Welcher Agent schafft es am längsten am Leben zu bleiben.
Multiprocessing fähig (boolscher Wert)	Welche Agenten sind multiprocessing fähig und können damit schneller und ausgiebiger trainiert werden.

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel soll es thematisch über den momentanen Stand der bereits durchgeführten Forschung gehen. Dabei sollen drei ausgewählte Arbeiten vorgestellt und diskutiert werden. Diese wurden aufgrund ihres thematischen Hintergrundes zu dem Spiel Snake, in Verbindung mit dem RL, ausgewählt.
Besonders folgende Aspekte soll bei der Vorstellung und Diskussion thematisiert werden.

- Auswahlprozess der Agenten Art
- Wahl der Hyperparameter
- Netzsturktur
- Observation
- Reward Function
- Leistungsmessung
- andere Besonderheiten

4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning

In der folgenden Auseinandersetzung wird sich auf die Quelle Wei u. a. (2018) bezogen. In der Arbeit Autonomous Agents in Snake Game via Deep Reinforcement Learning wurden mehrere Optimierungen an einem DQN Agenten durchgeführt, um mit diesen eine größere Performance beim Sammeln von Äpfeln im Spiel Snake zu erzielen. Die Arbeitet wurde von Zhepei Wei et al., welche an verschiedenen Universitäten und Forschungsinstituten, wie z.B. College of Computer Science and Technology Jilin University (Changchun, China) und die Science and Engineering

Nanyang Technological University (Singapore) arbeiten, verfasst und im Jahr 2018 veröffentlicht.

4.1.1. Vorstellung

Thematisch werden zwei Optimierungen in diesem Paper vorgestellt, welche auf einen Baseline DQN (Referenz DQN) angewendet worden sind. Die Verfasser setzen dabei als Obs. 2.2.1 Screenshot des Spiels ein, um den Agenten mit Informationen über das Spiel zu versorgen. Diese werden mittels eines Convolutional Neural Network (CNN) verarbeitet, dass auf 3 Conv. und einem Linear Layer besteht.
Die vom Paper verwendete Reward-Funktion basiert auf drei Faktoren, wobei der erste die Distanz ist. Um so weiter sich die Snake von dem Apfel entfernt, um so größer (negativ) wird der Reward. Nährt sich hingegen die Snake dem Apfel an, so wird der Reward immer größer (positiv).
Der zweite Faktor ist der sogenannte Training Gap. Dieser stellt Erfahrungen dar, welche im Verlauf des Spiels nicht erlernt werden soll, wie z.B. die Erfahrungen direkt nach dem Fressen eines Apfels. Der Agent würde durch diese Erfahrungen lernen, wie das Spiel die Äpfel zufallsbasiert neu verteilt, was nicht dem Trainingsziel entspricht. Das Entfernen dieser Erfahrungen führt zu einem besseren Pathfinding.
Der dritte Faktor ist die Timeout Strategy. Diese sorgt für eine Bestrafung, wenn der Agent über eine gewissen Anzahl an Schritten P keinen Apfel gefressen hat. Der Agent wird dadurch angehalten die schnellste Route zu finden. Dabei werden die Rewards 2.2.1 der letzten P Schritte mit einem Malus verrechnet, der sich nach der Länge der Snake richtet. Insgesamt setzt sich die Reward-Funktion nun folgendermaßen zusammen:
Der Distance Reward bildet den Basis-Reward. Sollte sich das Spiel im Training Gap befinden, so wird der Reward des Training Gap gewählt. Überschreitet der Agent des weiteren die Granze von P Schritten ohne einen Apfel gefressen zu haben, wird der Timeout Strategy Reward Malus auf die letzten P Schritte aufaddiert.

Des Weiteren stellt das Paper eine neues Design von Experience Replay Buffer vor. Hierbei wird der Experience Replay Buffer in zwei Hälften geteilt, in MP_1 und MP_2. Der Unterschied zwischen beiden besteht darin, dass in MP_1 nur Erfahrungen mit einem Reward $r > 0.5$ gespeichert werden. Alle anderen Erfahrungen werden in MP_2 gespeichert.
Ähnlich der ϵ-greedy Strategie des DQN, wird das Verhältnis zwischen den Erfahrungen von MP_1 und MP_2 mit einer Konstante η gesteuert. Zu Beginn liegt diese bei $\eta = 0.8$, es werden daher durchschnittlich 80% Erfahrungen aus MP_1 und 20% aus MP_2 gewählt. Durch eine stetige Abnahme fällt η gegen ende der Abnahme auf

minimal $\eta = 0.5$. Durch diese Aufteilung soll ein schnellerer Lernerfolg gerade zu Beginn erzielt werden.

4.1.2. Diskussion

Wie bereits oben erwähnt, befasst sich die Arbeit mit zwei Optimierungsstrategien, welche nach experimentellen Experimenten bereits gute Resultate erzielt haben.
Jedoch bietet das Paper auch einiges Diskussionspotential. So verzichtet man auf einen differenzierten Auswahlprozess der Art der RL-Agenten (DQN, PPO oder A2C). Es wird einfach auf einen DQN vertraut (Wei u. a., 2018, Kapitel 1).
Auch wirft die verwendete Obs. Fragen auf. So werden lediglich Screenshot des Spiels als Obs. verwendet. Eine weitere Methode, um dem Agenten Informationen über das Spiel zukommen zu lassen wird nicht verwendet. Des Weiteren könnte das Verwenden von Screenshot des ganzen Spiels zu schlechteren Lernerfolg führen, da auch viele nicht benötigte Informationen dem Agenten übergeben werden.
Die Netzstruktur könnte diesen Effekt noch weiter verstärken, da sie aus drei Conv. Layern mit großen Filtergrößen (7x7, 5x5, 3x3) und Strides (4,2,2) besteht. Ein möglicher Informationsverlust wie auch eine Verlängerung der Lernzeit könnte durch die gewählte Konfiguration begünstigt werden.

4.2. Exploration of Reinforcement Learning to SNAKE

In der folgenden Auseinandersetzung wird sich auf die Quelle Bowei Ma (2016) bezogen. In der Arbeit Exploration of Reinforcement Learning to Snake stellen die Verfasser Bowei Ma, Meng Tang, Jun Zhang einen Vergleich zweier RL-Agenten vor. Die Arbeit war ein Ergebnis einer Projektgruppe der Universität Stanford und wurde im Jahres 2016 veröffentlicht.

4.2.1. Vorstellung

Wie bereits erwähnt, wird in dem Paper der Vergleich zweier RL-Agenten vorgestellt. Dabei handelt es sich um einen DQN und um einen SARSA (State-Action-Reward-State-Action). Beide Agenten arbeiten auf Basis einer Value Function, sie sind daher value-based 2.2.3. Nach einer kurzen Vorstellung der beiden Agenten und eines weiteren, auf einer Heuristik basierenden, optimalen Agenten, welcher als Referenz dient, präsentiert das Paper direkt seine Ergebnisse. So waren weder der DQN als auch der SARSA in der Lage das optimale Ergebnis für die entsprechende

Spielfeldgröße. Die Agenten lernten dabei zumeist unter festen Hyperparametern, welche durch Ausprobieren ermittelt wurden. Bei dem Vergleich der beiden Agenten wurde sich zudem nur auf die Performance konzentriert.
Des Weiteren wurden einige Auffälligkeiten erwähnt, wie z.B. das Lernverhalten der beiden RL-Agenten. So lernte der DQN bei kleineren Durchläufen schneller als der SARSA. Dieses ändert sich jedoch mit Anstieg der Lernzeit. Weiterhin konnte eine starke Instabilität beim Training des DQN festgestellt werden, die unter gewissen Trainingssituationen auftrat. In dem

4.2.2. Diskussion

Auch dieses Paper wirft einige Fragen auf, so ist beispielsweise die Wahl der beiden zu vergleichenden Agenten nicht begründet worden. Auch eine differenzierte Herangehensweise an die Wahl der Hyperparameter wird nicht erwähnt. Stattdessen wurden sie mittels einer "trial and error" Taktik bestimmt. Weiterhin ist besonders das Auslassen von Informationen über die Netzstruktur hervorzuheben. Ähnlich verhält es sich mit den Informationen über die Obs.
Auch wurde hauptsächlich bei dem Vergleich der beiden RL-Agenten nur die Performance verglichen. Andere Aspekte, wie beispielsweise die Robustheit, Trainingszeit oder die Spielzeit 3.4, werden nicht weiter für den Vergleich in Betracht gezogen. Die Verfasser benutzen des weiteren für das Lernen ihres Agenten eine einfach Reward Function, die +500 returned, wenn die Snake einen Apfel isst. Sollte die Snake sterben, so wird -100 ausgegeben. Ansonsten wird für jeden Step -10 veranschlagt. zwar ist die Reward Function effektiv aber es bleibt auch viel Potential ungenutzt. Eine effektive formulierte Function kann den Lernerfolg und die Lernzeit drastisch steigern.

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

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

4.3.1. Vorstellung

Das hier vorliegende Paper behandelt viele Aspekt. So haben sich die Autoren das Ziel gesetzt, einen Agenten zu erstellen, der so viele Snake Spiele wie möglich spielen kann. Die Verfasser setzen dabei auf eine ausgedehnte Netzstruktur, welche aus mehreren Convolutional, Pooling und Linear Layers besteht.
Wie auch bei den 4.1, verwendet die Autoren Screenshot des gesamten Spiels, um die Obs. aufzuspannen.
Auch bei der Reward Function bestehen gewisse Ähnlichkeiten zu 4.1. So wird eine Reward Function benutzt, welche auf einer Duftspur um den Apfel basiert. Implementiert wird diese durch einen festen Bereich um den Apfel, der einen größeren Reward returned als die allgemeine Umgebung. Dieser Bereich ist wiederum in drei weitere Ringe eingeteilt, sodass der Reward des ersten Rings größer als der des zweiten ist und der Reward des zweiten größer als der des dritten. Vergleichbar ist dies durch einen Berg, der in drei Höhenbereiche eingeteilt ist. Auf der Spitze befindet sich der Apfel. Der Reward liegt dabei höher, wenn man in einen höheren Höhenbereiche befindet. Entsprechendes gilt für tiefere Bereiche.
Ebenfalls erwähnt das Paper die in Teilen die Wahl der Hyperparameter des DQN. So wird z.B. ϵ nicht mit $\epsilon = 1.00$ initialisiert sondern mit $\epsilon = 0.1$. Dieser Wert wird in den nächsten 10.000.000 Zügen auf $\epsilon = 0.0001$ gesenkt.
Auch begründet das Paper die Wahl eines DQN, so ist dieser deutlich besser geeignet als ein Q-Table, da dieser bei großen State-Action-Räumen eine große Menge an Ressourcen bindet.
Eine weitere Besonderheit die die sogenannte Loop Strom Strategy. So kann es bei der verwendeten Reward Function passieren, dass der Agent einen größeren Reward erhält, indem er immer um den Apfel sich in Kreisen (Loops) umherbewegt. Offensichtlich leidet die Performance darunter, da kaum Äpfel gefressen werden. Daher haben Chunxue Wu et al. eine Erkennungsmethode von Loops, welche auf einem dynamischen Array basiert, entwickelt. Sollte diese erkennen, dass sich der Agent in einem Loop befindet, so sieht der Agent von einer Aktionsbestimmung via. NN ab und nutze eine Zufallsentscheidung, um den Loop zu verlassen.
Gegen Ende des Papers erfolge die Evaluation des DQN. Diese bezieht nur den erreichten Score ein, welcher sich über alle erhaltenden Rewards einer Spielepisode zusammensetzt. Verglichen werden diese Scores mit denen anderer Algorithmen, welche im Paper nicht weiter thematisiert werden und mit den Leistungen, welche von Menschen erbracht worden sind.

4.3.2. Diskussion

Von allen bereits diskutierten Papers stellt dieses das umfangreichste dar. Die Autoren haben viele Aspekte mit in die Bearbeitung einfließen lassen. Dies stellt sich jedoch auch als ein möglicher Nachteil dar. Zwar zeigen erste Auswertungen, dass Teile der Arbeit, wie z.B. die Reward Function und die Loop Storm Strategy, die Performance verbessern, jedoch wurden dafür auch an anderen Stellen Abstriche gemacht. So fußt die Auswahl des Agenten, auf einem Vergleich mit einem Q-Table. Dieser Vergleich ist jedoch nicht sonderlich aussagekräftig, da Q-Table nur sehr eingeschränkt benutzt werden können, da sie bei großen State-Action-Räumen keine guten Ergebnisse erzielen. Ein Vergleich zu anderen konkurrierenden RL-Agenten Arten wurde nicht durchgeführt.
Auch die Wahl der Netzstruktur und der damit einhergehenden Obs. eröffnet Kritikpunkte. So wird zwar ein großes ConvNet verwendet, welches jedoch wieder das gesamte Spielfeld überblicken muss. Wie auch schon in 4.1 angesprochen, kann dies zu Informationsverlust führen. Auch ist die Strategie alle 0.03 Sekunden einen Screenshot zu erstellen und diesen dann durch das NN zu propagieren fragliche, da es zwar in real-world Anwendungen natürlich Anwendung findet wird in einem Snake Env. mit diskreten Gitterstrukturen eher wenig angepasst wirkt.
Zwar haben sich die Verfasser mehrere Gedanken zur Wahl der Hyperparameter gemacht, diese jedoch nur selten begründet. So wird der Parameter ϵ zwar genaustens beleuchtet, jedoch die ungewöhnlich niedrige Lernrate von $1e{-}6$ kaum erwähnt.
Wie auch schon bei den anderen Papers wird sich bei der Evaluation hauptsächlich nur auf einen bis zwei Parameter beschränkt. Sowohl der Score als auch die Q-Values werden genauer beleuchtet. Andere Evaluationsparameter, wie die Spielzeit, Robustheit usw. werden nicht mit eingebracht.
Dennoch kann das Paper mit seiner Loop Storm Strategy und Reward Function überzeugen.

4.4. Zusammenfassung

Zusammenfassend lässt sich sagen, dass die vorgestellten Arbeiten alle ihr Qualitäten besitzen. Das Paper "Autonomous Agents in Snake Game via Deep Reinforcement Learning" 4.1 wie auch das Paper "UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene" 4.3 verwendeten beispielsweise effizientere und kreativere Reward Functions im Vergleich zum Standard Belohnung nur für das Essen eines Apfels; Bestrafung für das Sterben; weder Belohnung noch Bestrafung für das Umherlaufen). Auch konnten die beiden Papers mit neuen Ansätzen wie der Loop Strom Strategy und der Training Gap Strategy überzeugen.

Das Paper "Exploration of Reinforcement Learning to SNAKE" geht diesbezüglich noch einen Schritt weiter und beschäftigt sich vollkommen mit einem Vergleich von RL-Agenten.

Dennoch lässt sich an den Papers auch Kritik üben. So konnte keines mit einer adäquaten Begründung aller der oben genannten Faktoren, wie z.B. Netzstruktur, Leistungsmessung usw., überzeugen. Besonders die Leistungsmessung, Wahl der Agenten Art und die Wahl der Netzstruktur fielen bei den Arbeiten besonders unzureichend aus. Häufig wurde mehr Fokus auf spezifische Besonderheiten und auf die Reward Function gelegt.

Kapitel 5

Agenten

Ein zentraler Aspekt der eines Vergleiches von verschiedenen RL-Agenten ist die genaue Definition der einzelnen Agenten. Basierende auf den Grundlagen 2.2.1 soll in diesem Kapitel der Begriff vervollständigt und die zu vergleichenden Agenten sollen vorgestellt werden.
Erste statistische Erhebungen haben gezeigt, dass die ausgewählten Hyperparameter einen immensen Einfluss auf das Verhalten der Agenten haben. Bestätigt wird diese Aussage durch die Quelle Sutton und Barto (2018). Ein Vergleich zwischen DQN und PPO mit wahllos gewählten Hyperparametern ist folglich wenig aussagekräftig. Daher ist auch die Definition des Begriffs Agent, welcher nur zwischen DQN und PPO differenziert, unzureichend.

Angebracht wäre eine neuer erweiterte Definition des Begriffs Agent für diese Ausarbeitung. Diese soll um den entscheidenden Faktor der Hyperparameter erweitert werden. Ein Agent wird daher nicht mehr alleinig durch seine RL-Klasse (Q-Learning oder Policy Gradient) und Algorithmus-Klasse(DQN oder PPO) definiert, sondern ebenfalls durch die ausgewählten Hyperparameter.

5.1. Agenten

Im Folgenden werden die einzelnen Agenten, welche untereinander verglichen werden sollen, tabellarisch vorgestellt. Daher wird Aufschluss über Details, wie z.B. die RL-Klasse, Algorithmus-Klasse, Hyperparameter und die Netzstruktur gegeben.

Tabelle 5.1.: Agenten

Agentenname	RL-Klasse	Algorithmusklasse	Hyperparameter

DQN_0.99_64_5e-6_2**12_5e-4	Q-Learning	DQN	• gamma (γ) = 0.99 • batch_size = 64 = 2^6 • epsilon_decrement = 5e−6 • max_mem_size = 2^{12} • lr = 5e−4
DQN_0.95_128_1e-5_2**13_1e-4	Q-Learning	DQN	• gamma (γ) = 0.95 • batch_size = 128 = 2^7 • epsilon_decrement = 1e−5 • max_mem_size = 2^{13} • lr = 1e−4
PPO_0.99_128_10_1e-3_1.5e-3_0.5_1e-3_128_2**11	Policy Gradient	PPO	• gamma (γ) = 0.99 • K_epochs = 10 • epsilion_clip = 0.2 • lr_actor = 1e−3 • lr_critic = 1.5e−3 • critic_loss_coefficient = 0.5 • entropy_coefficient = 0.001 • batch_size = 128 • max_mem_size = 2^{11}

PPO_0.95_128_8_0.5e-3_1e-3_0.5_1e-4_64_2**9	Policy Gradient	PPO	• gamma (γ) = 0.99 • K_epochs = 10 • epsilion_clip = 0.2 • lr_actor = 1e−3 • lr_critic = 1.5e−3 • critic_loss_coefficient = 0.5 • entropy_coefficient = 0.001 • batch_size = 128 • max_mem_size = 2^{11}

Kapitel 6

Vorgehen

In diesem Kapitel soll das weitere Vorgehen thematisiert werden. Dazu erfolgt eine genaue Erklärung der weiteren Schritte.

6.1. Bestimmung der Netzstruktur

Kapitel 7

Implementierung

Für eine Umsetzung eines solchen Vergleichs, wie er in dem Kapitel 6 beschrieben worden ist, ist es nötig eine Implementierung des Spiels Snake und der beiden Agenten, inklusive der Ablaufroutine, durchzuführen. Als Programmiersprache wurde Python (3.7.9) gewählt.
Python bietet im Bereich des DRL eine Vielzahl an Frameworks, welche nicht nur bei der Implementierung des Envs. helfen, sondern auch welche, die Funktionalität der Neuronalen Netzwerke bereitstellen.

7.1. Snake Environment

Zur Implementierung des Spiels Snake wurde das Framework gym von OpenAI genutzt. Dieses bietet viele Methoden, welche das Implementieren erleichtern. Jedoch werden nicht nur Methoden, sondern auch die File-Struktur vorgegeben. So besteht das Snake Environment Package, dass den Namen snake_env trägt, aus den wesentlichen Files:

- gui.py
- observation.py
- snake_env.py
- snake_game_2d.py

7.1.1. Schnittstelle

Die grundlegende Schnittstelle des Snake Env. wird in dem File snake_env.py bereitgestellt. In diesem File wird eine Klasse SnakeEnv definiert, welche durch das Erben von der Oberklasse gym.Env zentrale Methoden, welche teils als Schnittstellen zu den Agenten dienen, vorgegeben bekommt.
So wird dem Programmierer angeraten, die folgenden Methoden zu implementieren:

- step
- reset
- render
- close

Zuzüglich wurde noch eine Methode post_init definiert, welche für das weitere zentrale Einstellungen, wie z.B. die Spielfeldgröße und ob ein Spielfeld überhaupt visuell dargestellt werden soll, verantwortlich ist. Die step-Methode dient zur Ausführung der vom Agenten ausgewählten Action. Nach der Abarbeitung dieser Action wird die neue Observation, der erhaltende Reward, das done-flag und das has_won-flag übermittelt. Letzteres dient zur Bestimmung des Sieges.
Die reset-Methode setzt das Spiel zurück, in dem sie das Env. Objekt zerstört und durch ein neues ersetzt. Zum Schluss wird dann noch die Obs. des neues Envs. zurückgegeben.
Die render-Methode ruft die entsprechenden Methoden im gui.py File auf, welche für das visuelle darstellen des Snake Envs. verantwortlich sind.
Die close-Methode terminiert das Programm.

Alles in allem stellt die Klasse SnakeEnv eine Wrapper-Klasse dar, die als Schnittstelle dient.

7.1.2. Spiellogik

Die eigentliche Spiellogik, welche durch die bereits zuvor erwähnte step-Methode angestoßen wird, liegt jedoch ausgelagert im snake_game_2d.py File. Dort wird eine Klasse definiert, welche SnakeGame heißt. Diese enthält unter anderen die Methode action(self), welche für die eigentliche Ausführung der Aktionen verantwortlich ist. Neben dieser befinden sich jedoch auch Methoden, wie evaluate(self), die für das Bestimmen des Rewards verwendet wird, observe(self), welche den Generierungsprozess für die Observation anstößt, make_apple(self), die nach dem ein Apfel gefressen worden ist einen neuen Apfel generiert und einige weitere selbsterklärende, die die Spiellogik aufspannen. Zu den wichtigsten Attributen der Klasse gehören "p", welcher das Player-Objekt darstellt, "ground", welche die dem Spiel zugrunde liegende Matrix repräsentiert, shape_tuple, welches die Spielfelddimensionen enthält, gui, welches das GUI-Objekt darstellt, dass für die visualisiert zuständig ist und apple, welche die Position des Apfels speichert.

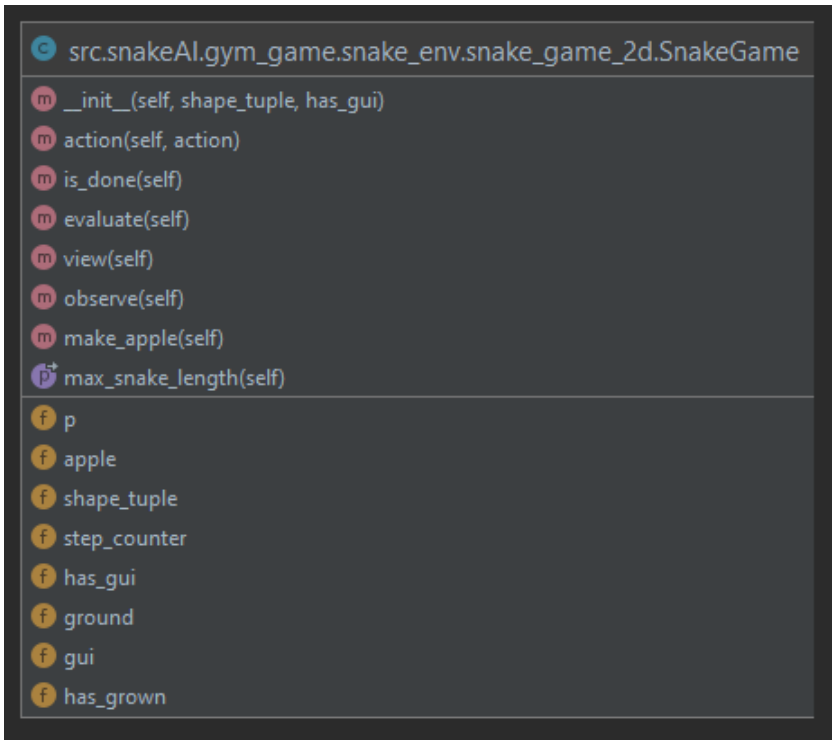

Abbildung 7.1.: Darstellung des Klassendiagramms der SnakeGame-Klasse.

Neben dieser wird noch eine Datenhaltungsklasse definiert, welche alle Daten des Spielers enthält. Zu diesen gehören z.B. die Position des Kopfes und aller Schwanzglieder, die Blickrichtung, wie viele Schritte seit dem letzten Konsum eines Apfel vergangen sind, die Information, ob der Spieler gestorben ist und eine id zuzüglich weiteres sich aus dieser ergebender Konstanten, welche zur Darstellung des Spiels benötigt werden.

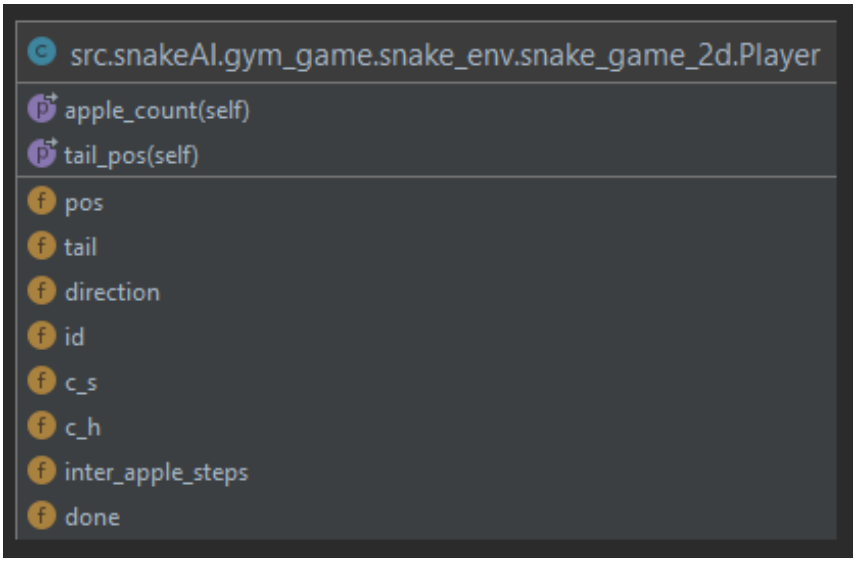

Abbildung 7.2.: dar.

Die SnakeGame Klasse definiert die grundlegenden Spiellogik. Aufgrund der Tat-

sache, das Snake ein gering dimensionales Spiels ist, welches nur eine zweidimensionales Spielfeld benötigt, bietet sich die Verwendung einer Matrix an. So kann mit entsprechenden Zahlenwerten innerhalb der Matrix der Position der Snake deutlich gemacht werden. Um auf diesem Konzept weiter aufzubauen, wird mit den Zahlenwerten nicht nur die Anwesenheit der Snake auf dem Feld angezeigt, sondern es wird durch unterschiedliche Werte auch die Position des Kopfes und es letzten Schwanzgliedes symbolisiert. Der Einfachheit halber werden die Positionen der Snake des weiteren auch noch in einer Pos-List (Positionsliste) festgehalten. Diese Positionen werden als Tupel in der List gespeichert, wobei die Tupel dabei die Zeilen- und Spalten-Indexe der Matrix angeben. Dieser Mehraufwand wird getätigt, um später schneller bestimmen zu können, ob die Snake in sich selbst oder in eine Wand gelaufen ist.

Mit dem folgenden Aktivitätsdiagramm soll der Fokus weiter auf die action-Methode gelegt werden. Diese wird wie folgt abgearbeitet. Zu aller erst wird der sich im Player-Objekt liegende inter_apple_steps erhöht. Dieser zählt die Schritt seit dem die Snake das letzte Mal einen Apfel gefressen hat. Sollte dieser counter jedoch größer als, die vorher definierte, Obergrenze an Schritten ohne Apfel sein, so wird die Methode terminiert, der Agent hat verloren und das Spiel startet von neuem.
Befindet sind der Agent Spieler unterhalb der Obergrenze, so überprüft, um welche action es sich handelt. Dabei sind die actions mit den Zahlen von 0 - 2 kodiert. Null für 90° nach links, eins für 90° rechts und zwei für nichts ändert sich, daher weiter geradeaus. Entsprechende wird die direction des Players angepasst. Da es vier Himmelsrichtungen gibt werden diese intern mit den Zahlen von 0 - 3 dargestellt. Wobei null Norden entspricht und eins Osten uns.
Nach der Manipulation der direction des Players, wird das interne Player Positions-Tupel (pos) im Player-Objekt angepasst. Diese Änderung nicht jedoch noch nicht sofort in die Matrix übertrage, da es beim verlassen des Spielfeldes zu Exceptions kommen würde. Es muss daher erst überprüft werden, ob die neue Pos. des Players im Spielfeld liegt. Sollte dies nicht der Fall sein, so wird die Methode abermals terminiert, der Spieler hat verloren und das Spiel startet von neuem.
Sollte dies jedoch nicht der Fall sein, so wird die neue Pos. in die bereits erwähnt List namens tail (im Player-Objekt) eingefügt.
Zu diesem Stand der Abarbeitung ist es möglich, dass das Spiel bereits gewonnen ist. Um dies zu überprüfen, wird die Länge der Snake mit der maximal möglichen Länge, welche sich durch die Spielfeldgröße definiert, verglichen. Wenn die Länge Snake der maximal mögliche entspricht, so wird die Methode terminiert usw.

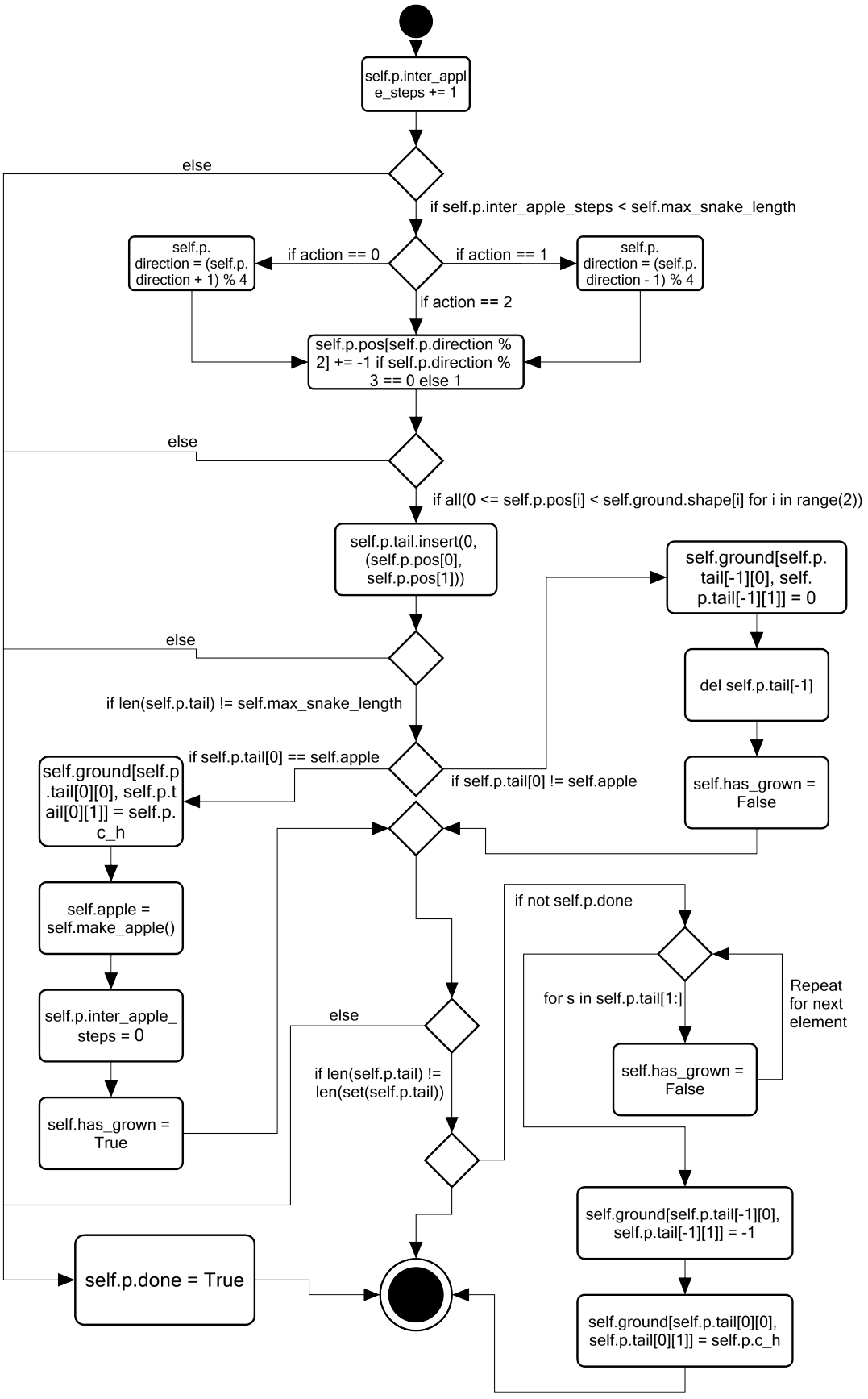

Abbildung 7.3.: Aktivitätsdiagramm der action(self, action) Methode

Sollte dies nicht der Fall sein, so muss als nächster Schritt die Matrix aktualisiert werden. Um dies jedoch zu tun, muss vorher festgestellt werden, ob die Snake einen Apfel gefressen hat. Sollte sie dies getan haben, so wird die neue Pos. des Kopfes in die Matrix eingepflegt. Des Weiteren wird noch ein neuer Apfel auf einer zufälligen freien Stelle generiert, der inter_apple_steps counter wird auf null und ein has_grown boolean-flag wird auf true gesetzt. Dieses ist für die evaluate Methode, um die Information zu erhalten, ob die Snake einen höheren reward erhält, da sie ein sub-goal erreicht hat.
Ist die Snake jedoch nicht gewachsen, so wird das letzte Schwanzglied aus der Matrix und Liste gelöscht, damit der Anschein von Bewegung entsteht. Des Weiteren wird noch das has_grown- boolean-flag auf false gesetzt.
Nach dieser Ausführung besteht nun die Möglichkeit, dass die Snake in sich selber gelaufen ist. Um dies festzustellen wird überprüft, ob die Liste mit allen Positionen irgendwelche Duplikate enthält. Sollte dies der Fall sein, so wird die Methode terminiert usw.
Sollte auch dies nicht der Fall gewesen sein, so wird zum Schluss noch durch die gesamte Pos-List durch iteriert und die Felder der Matrix werden mit der Pos-Liste aktualisiert. Dabei wird jedes Schwanzglied in der Matrix eingepflegt. Kopf und das letzte Schwanzglied werden dabei durch einen anderen Zahlenwert wieder besonders hervorgehoben.

7.1.3. Reward Function

Die evaluate Methode, welche als Reward Function zu beschreiben ist, befindet sich in der SnakeGame Klasse 7.1. Basierend auf dem letzten Zug wird in dieser Methode der Reward bestimmt. Dies geschieht nach folgenden Vorbild.

Tabelle 7.1.: Reward Function

Action	Reward
Letzter Apfel gefressen und das Spiel ist vorbei.	Sollte es der Agent geschafft haben den letzten Apfel des Spiels gefressen zu haben, also wenn die Länge der Snake die maximal mögliche Länge erreicht hat und wenn p.done wahr ist (p.done == True) 7.3, sodann wird ein Reward von +100 zurückgegeben. Dies soll ein Anreiz darstellen, damit die Snake das Spiel möglichst häufig gewinnt.

Snake ist gestorben.	Wenn die Snake gestorben, also in sich selbst, in eine Wand oder zulange umher gelaufen ist, dann wird ein Reward von -20 zurückgegeben. Diese Verlustbedingung tritt ein, sobald die Länge der Snake nicht die maximal mögliche Länge erreicht hat und p.done gleich wahr ist (p.done == True). Dieser Reward soll den Agenten dazu bewegen, nicht in einen terminalen Zustand zu geraten.
Snake hat gefressen.	Sollte die Snake einen Apfel gegessen haben und sie dabei nicht gestorben sein, sodann wird ein Reward von +2 zurückgegeben. Dies tritt ein, wenn p.done gleich falsch ist (p.done == Flase) und das has_grown wahr ist (has_grown == True) **??**. Der Reward kleine aber positive Reward soll die Snake ermutigen, mehr Äpfel zu sammeln.
Einen Schritt gehen	Solle keine der oberen Fälle eintreten sein und das p.done falsch als Wert beinhalten (p.done == Flase), so wird ein Reward von -0.01 zurückgegeben. Dies soll die im Weiteren zu einer stärkeren Optimierung des Pathfinding führen, da jeder unnötige Schritt leicht bestraft wird.

7.1.4. Observation

Die Observation, welche das Snake Env. zurückgibt besteht aus zwei Teilen. Aus der sogenannten around_view und der static_Obs, welche beide unterschiedliche Informationen in sich tragen. Zur Erstellung der Obs. wird die observe Methode in der SnakeGame Klasse aufgerufen. Dies ruft ihrerseits die make_obs Funktion auf, welches ausgelagert im observation.py File liegt. Mit Hilfe verschiedener Unterfunktionen wird dann die Obs. generiert.
Die around_view lässt sich dabei als ein Ausschnitt der Matrix beschreiben, die nicht das gesamte Spielfeld einnimmt 7.4. Dieser Ausschnitt vermittelt ein Bild der der Snake umgebenden Umgebung. Strukturen wie Wände und Teile des eigenen Schwanzes, die vielleicht eine Sackgasse aufspannen, werden deutlich. Numerische wird die around_view als eine one-hot-encoded Matrix der Form (6,13,13) zurückgegeben.
Das One-Hot-Encoding benutzt zum codieren nur null und eins. Sollte ein Merkmal vorhanden sein, so wird dieses mit eins codiert anderenfalls mit null.
Dies ist auch der grund, warum die around_view Matrix sechs zweidimensionale Schichten besitzt. Diese geben Aufschluss über folgende Informationen:

Tabelle 7.2.: Around_View

Erste dim. der Matrix (Ax13x13)	Erklärung
A = 0	Die erste zweidimensionale Schicht signalisiert den Raum außerhalb des Spielfelds. Sollte sich die Snake dem Rand nähren, so würde die around_view aus dem Spielfeld herausragen und den Eindruck verschaffen, dass dieses größer ist als es in Realität wirklich ist. Um dies zu vermeiden, werden Felder der around_view, die sich außerhalb des Spielfeldes befinden, angezeigt.
A = 1	Diese Schicht stellt alle Schwanzglieder mit Ausnahme des Kopfes und es letzten Schwanzgliedes dar.
A = 2	In dieser Schicht wird der Kopf der Snake dargestellt.
A = 3	Damit gegen Ende des Spiels der Agent noch freie Felder erkennen kann wird in dieser Schicht jedes freie und sich im Spielfeld befindliche Feld mit eins codiert.
A = 4	Die vorletzte Schicht codiert das Schwanzende der Snake.
A = 5	In der letzte Schicht wird der Apfel abgebildet.

Vorteilhaft an dieser ersten Art von Obs. ist, dass im Gegensatz zu den verwandten Arbeiten 4.1 und 4.3 nicht das gesamte Feld übertragen wurde sondern nur der wichtigste Ausschnitt, was die Menge an zu verarbeiten Daten drastisch reduzieren kann, je nach Spielfeldgröße. Des weiteren ergeben sich keine Probleme mit der Input-Size der Convolutional Layer.
Ein Nachteil dieser Obs., sollte man diese als einzige Obs. definieren, ist die Vollständigkeit. Sollte der blaue Punkt in 7.4 außerhalb des grauen Kasten und daher außerhalb der around_view liegen, so besitzt der Agent keinerlei Informationen über den Aufenthaltsort des Apfels. Auch ist der Agent nicht, ohne größere Suchen, in der Lage sich ein Bild der weiterer entfernten Umgebung zu machen. Auch weitere Informationen wie z.B. der Hunger, also die noch verbleibenden Schritte bis das Spiel endet, die Distanzen zu den Wänden und zu, außerhalb der around_view liegenden, Schwanzteilen und die Blickrichtung der Snake, also die direction.

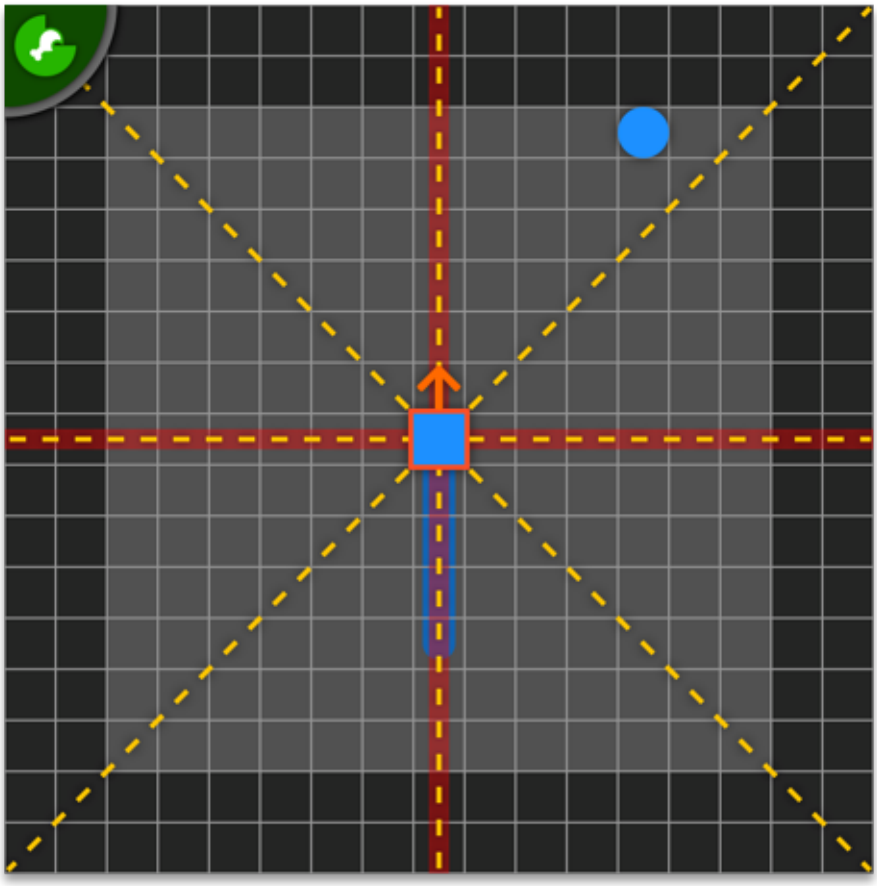

Abbildung 7.4.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar, wobei der rot umrandete Rechteck in der Mitte der Grafik den Kopf symbolisiert. Die schwarzen Felder stellen all die Felder da, welche nicht von der around_view abgedeckt werden. Graue Felder liegen innerhalb der around_view. Die gelben gestrichelten Linien stellen ein X-Ray Detektion dar und die roten dickeren Linien liefern die relative Pos. der Snake. Der blaue Kreis stellt den Apfel dar und der grüne viertel Kreis oben links symbolisiert die noch zu gehende Schritt bis das Game endet.

Aus diesem Grund wurde die around_view mit einer weiteren Obs. verfeinert, der static Obs. Diese beinhaltet Informationen, wie z.B. ein X-Ray, siehe 7.4 (gelbe gestrichelte Linien). Diese X-Rays geben die Distanz zu der nächsten Wand oder Schwanzglied an. Ebenfalls wird die direction, der Hunger, und der Apfelkompass und ein Kompass für das Schwanzende übergeben, wobei diese nicht in 7.4 dargestellt sind. Diese zeigen an in welcher Himmelsrichtung sich das gesuchte Objekt befindet. Numerisch werden die X-Ray und Hunger partizipial Obs. mit Hilfe der Funktion 1/Distanz bzw. 1/max_steps - p.inter_apple_steps definiert. Andere partizipial Obs. wie, z.B. die Kompasse und die direktion, werden mit Vektoren one-hot-encoded basier generiert. Alle partizipial Obs. werden zum Schluss zusammengefügt und mit der around_view zurückgegeben.

7.1.5. Graphische Oberfläche

Im gui.py file wird eine Klasse GUI definiert, welche die Aufgabe besitzt das Spiel graphisch darzustellen. Dies erfolgt mit Hilfe des Frameworks pygame, dass die Möglichkeit bietet graphische Oberflächen zu generieren. Dazu wird in der GUI-Klasse ein Oberfläche erzeugt, welche der dem Spiel zugrundeliegenden Matrix entspricht, siehe 2.1. Diese kann über das screen Attribut angesprochen und angepasst werden. Dies geschieht mit Hilfe der update_GUI Methode. Diese wird von der SnakeGame Methode view aufgerufen. Die update_GUI Methode überschreibt jedes einzelne Feld des Spielfeld mit dem korrespondierenden Wert der der Matrix (ground). Spielfeld und Matrix sind daher nicht direkt gekoppelt sondern müssen über die update_GUI Methode angeglichen werden.
Die Größe der Spieloberfläche wird dynamisch berechnet und kann über das Attribut Particle verändert werden. Dieses beschreibt die einzeln Feldgröße eines entsprechenden Matrixeintrages. Die anderen Methoden und Attribute der GUI Klasse sind selbsterklärend und erfordern daher keiner genaueren Erklärung.

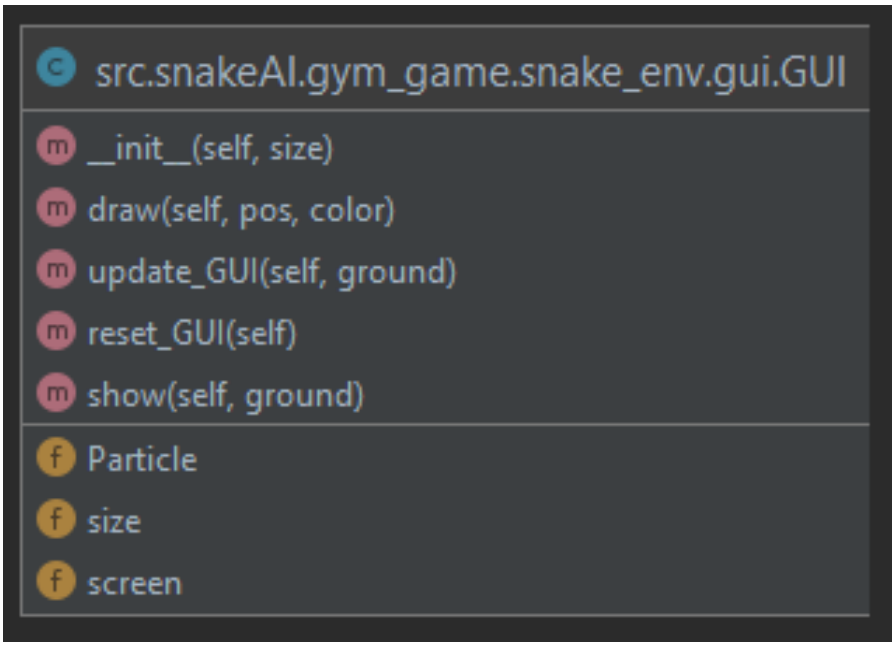

Abbildung 7.5.: Darstellung des Klassendiagramms der GUI-Klasse.

7.2. Agenten

Dieser teile der Implementierung soll sich mit den Agenten befassen. Dabei wird näher auf die Netzstruktur, den Aktionsauswahlprozess die Lern-Methode und den Speicher (Replay Buffer), eingegangen.

7.2.1. Netzstruktur

Zu Beginn soll die Netzstruktur erklärt werden. Dies kann unabhängig von den Agenten geschehen, da sowohl die DQN als auch die PPO Agenten das gleiche Netz nutzen, mit geringfügigen Änderungen.

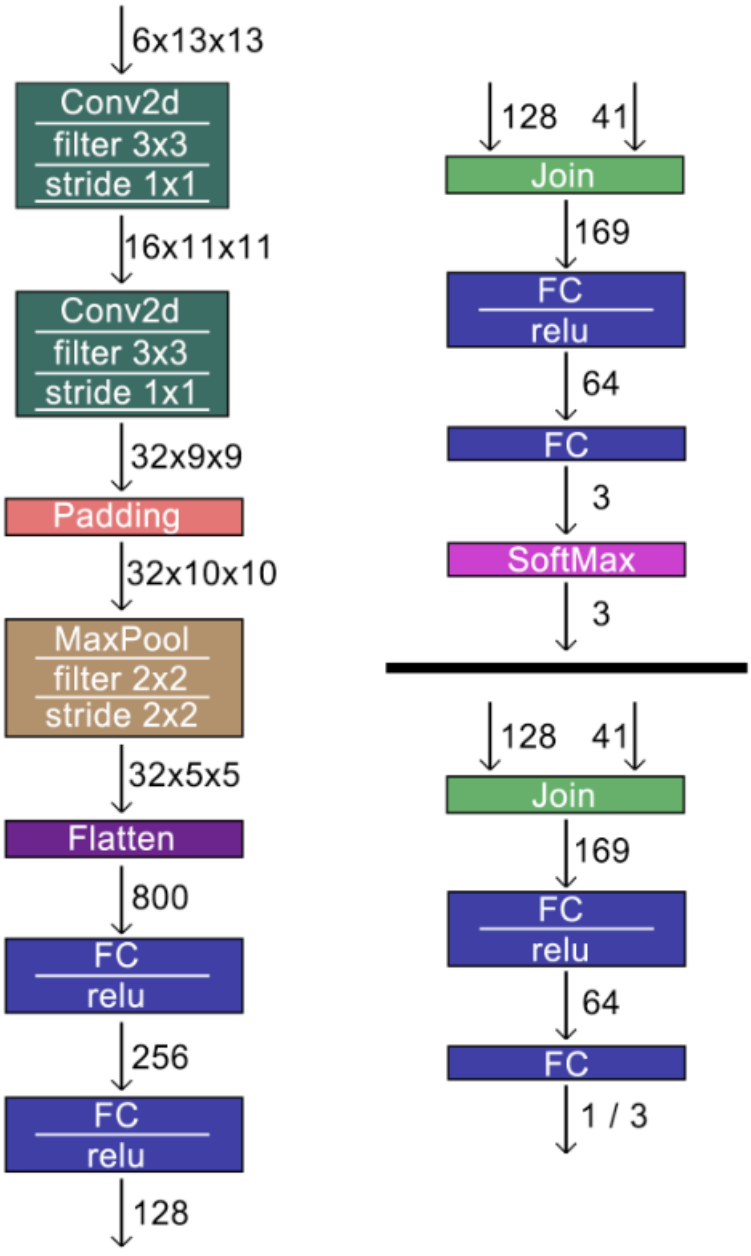

Abbildung 7.6.: Darstellung des BaseNet, welches als Standard für den weiteren Vergleich dient.

Aufgrund der Agentenunabhängigkeit, ist das NN als eine eigenständige Klasse im base.py File ausgelagert. Dieses BaseNet-Klasse besitzt eine forward Methode, welche die Input-Tensoren (av, static_obs) durch das NN propagiert und einen Output bestimmt. Dieser Prozess geschieht dabei folgendermaßen:
Zuerst wird die av durch zwei Convolutional Layer mit einer ReLU Aktivierungs-

funktion durch propagiert. Dabei erhöht sich die Channel-Anzahl auf 32, um eine bessere Merkmalextraktion zu erhalten und die Feature Map wird von 13x13 auf 9x9 minimiert, was ein Effekt der Convolutional Layern ist. Danach wird alles Feature Maps eine Zeile und Spiele mit nullen hinzugefügt, damit beim Max-Pooling unter einer Filtergröße und einem Stride von 2x2, auch die neunte Zeile und Spalte verarbeitet wird. Nach dem max-pooling besitzt die der resultierende Tensor 32 Channel mit Feature Maps der Größe 5x5. Diese werden nun zu einem einzigen eindimensionalen Tensor geebnet (Flatten). Dieser wird durch zwei weitere Fully Connected Layer durch propagiert. Der resultierende Tensor besitzt die Größe 1x128 und ist ein zwischen Ergebnis, da dieser nun mit der static_obs verbunden wird (Join).
Da das NN für beide Algorithmusarten verwendet werden soll müssen mehrere unterschiedliche Outputs definiert werden. Dazu wurden drei Netzwerkköpfe definiert, siehe 7.6. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nach den der Joined Tensor durch zwei weitere FC Layer propagiert wurde, benötigt der Actor der PPO-Agenten eine Wahrscheinlichkeitsverteilung über alle Actions. Daher auch die Ausgabe von einem Tensor der Größe 3 nach dem zweiten FC Layer, siehe 7.6 rechts oben. Zur Generierung einer Wahrscheinlichkeitsverteilung wird auf den Tensor noch SoftMax angewendet.
Der Critic der PPOs und die DQNs verwendet den in 7.6 rechts unten dargestellten Netzwerkkopf, den ValueNet-Head. In diesem wird auf den SoftMax verzichtet. Beim Critic wird Tensor mit einziger Zahlenwert und bei den DQNs ein Tensor mit drei Zahlenwerte, entsprechende der Actions zurückgegeben.

7.2.2. DQN

Der DQN Algorithmus ist einer der beiden Algorithmus-Arten, welche im Rahmen dieser Ausarbeitung, implementiert wurden. Diese Implementierung erstreckt sich im hauptsächlich über vier Files. Im dqn.py File wird die Agentenklasse definiert, welche die Aktionsausführungsmethode act und die lern Methode enthält. Im memoryDQN.py File wird die Memory-Klasse definiert, welche den Replay-Buffer 2.4. Das dqn_train.py und dqn_play.py File enthalten die eigentlichen main-Methoden, welche die Agenten und das Env. erstellen und den Trainings bzw. Spielprozess umsetzt.

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird durch die act Methode durchgeführt. Diese ist im dqn.py File definiert unter der Agent Klasse des DQN. Zur Bestimmung der nächsten Action wird der Methode die momentane Obs. übergeben. Die Q-Learning Agenten bestimmen ihre Actions mit des ϵ Wertes, siehe 2.4. Dieser bestimmt das Verhältnis zwischen der Wahl einer zufälligen Action und einer von NN bestimmten Action.

Dabei wird folgendermaßen vorgegangen.
Zuerst wird ein Zufallswert a generiert, welcher mit ϵ verglichen wird. Sollte dieser Zufallswert kleiner als ϵ sein $a < \epsilon$ so wird die Action mittels des NN bestimmt. Sollte dieser Zahlenwert größer sein $a > \epsilon$ sodann wird eine zufällige Action gewählt.
Zur Bestimmung der Action durch das NN wird einfach aus der Obs. ein bzw. zwei Tensoren generiert und dieser bzw. diese zu dem vordefinierten Device (CPU oder GPU) geschickt. Danach werden dieser bzw. diese durch das NN propagiert. Zum Schluss wird die neue Action zurückgegeben.

Trainingsprozess

Der Trainingsprozess wird über die learn Methode ausgeführt, welche in der Agent-Klasse des definiert ist. Der Ablauf der Lernprozedur stellt sich dabei wie folge dar: Zuerst wird überprüft, ob im Replay Buffer (Memory) genügend Experiances (Exp.) gespeichert sind um einen Mini-Batch zu füllen. Sollte dies nicht der Fall sein, wird die Methode terminiert. Anderenfalls wird ein Mini-Batch aus zufälligen Exp. ohne Duplikate gebildet.

Der Replay Buffer bzw. Memory wird durch einen Tensoren dargestellt welche jeweils immer eine Dimension mehr bezitzen als z.B. der Reward die av oder die static_obs. Dadurch lassen sich diese in der neuen Dimension speichern. So besitzt der Tensor der av die Form (max_mem_size, 6, 13, 13). Die Tensoren werden dabei wie eine eine Überlaufliste verwaltet. Wenn die Anzahl an Exp. die max_mem_size überschreitet, dann werden die ältesten Einträge durch die neu einzufügenden ersetzt.

Als nächstes werden alle Q-Values der States (s) des Mini-Batch berechnet. Daraufhin wird der Q-Value der im Replay-Buffer gespeicherten Action entnommen und unter dem Namen q_eval, zwischengespeichert. Danach werden die alle Q-Values der Nachfolge-States (s_next) bestimmt und ebenfalls zwischengespeichert. Die Q-Values terminaler States werden anschließend auf null gesetzt, da die Discounted Sums of Rewards bis zum Ende der Spielepisode null entspricht, siehe 2.4.
Um nun den sogenannten q-target Wert zu bestimmen, werden die maximalen Q-Values der s_next bestimmt, diskontiert und mit dem, im Mini-Batch gespeicherten, Reward addiert. Die Logik, warum diese geschieht wird näher in 2.4 erklärt. Nachfolgend wird der MSE (Mean Squared Error) zwischen q_eval und q_target bestimmt. Zuletzt wird der Fehler Mit Hilfe des Pytorch Frameworks zurück propagiert und entsprechend die Parameter des NN angepasst.

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*. Video. August 2017. – URL `https://www.youtube.com/watch?v=xvRrgxcpaHY&`

[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 ; Ming, Zhang ; Tan, Ah-Hwee ; Miao, Chunyan ; Zhou, You: Autonomous Agents in Snake Game via Deep Reinforcement Learning, 07 2018, S. 20–25

Anhang A

Anhang

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