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, 17. August 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. Auch das Spiel Snake wird Erwähnung finden.

2.1. Game of Snake

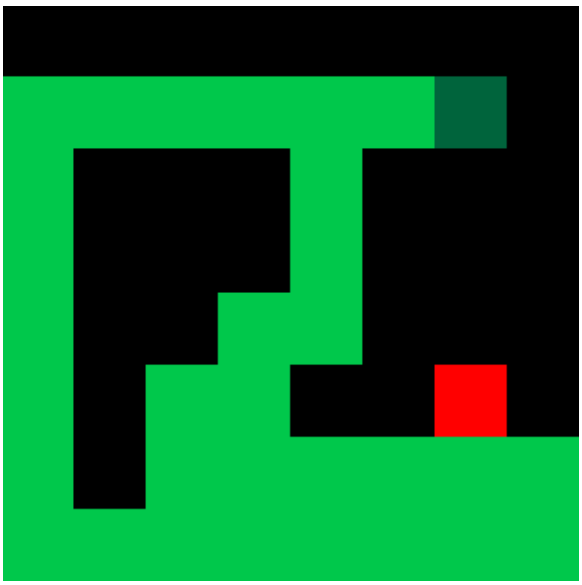

Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

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

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

2.2. Reinforcement Learning

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

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

State

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

Policy

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

Value

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

2.2.2. Funktionsweise

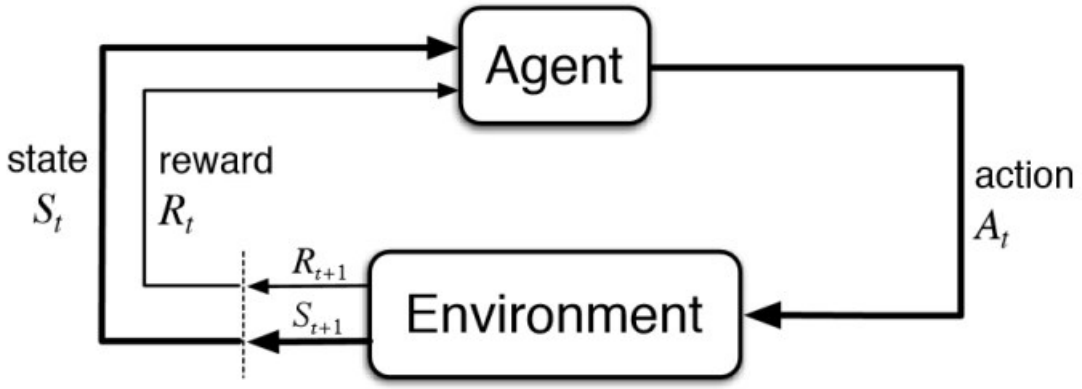

Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State S_t und falls $t \neq 0$ einen Reward R_t. Daraufhin wird vom Agenten eine Action A_t ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State S_{t+1} und Reward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

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

2.2.3. Arten von RL-Verfahren

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

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

Model-free und Model-based

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

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

Policy-Based und Value-Based Verfahren

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

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

On-Policy und Off-Policy Verfahren

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

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

2.3. Proximal Policy Optimization

Der Proximal Policy Optimization Algorithmus oder auch PPO abgekürzt wurde von den Open-AI-Team entwickelt. Im Jahr 2017 erschien das gleichnamige Paper, welches von John Schulman et al. veröffentlicht wurde. In diesem werden die besonderen Funktionsweise genauer erläutert wird Schulman u. a. (2017).

2.3.1. Actor-Critic Modell

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

2.3.2. PPO Training Objective Function

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

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

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

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

Formelelemente

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

Tabelle 2.1.: Formelelemente

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

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

Return

Der Return R_t stellt dabei die Summe der Rewards in der gesamten Spielepisode von dem Zeitpunkt t an dar. Diese kann ermittelt werden, da alle Rewards, durch das Sammeln von Daten, bereits bekannt sind. Des Weiteren werden die einzelnen Summanden mit einem Discount Factor γ multipliziert, um die Zeitpräferenz des Agenten besser zu steuern. Gamma liegt dabei gewöhnlich zwischen einem Wert von 0.9 bis 0.99. Kleine Werte für Gamma sorgen dafür, dass der Agent langfristig eher dazu tendiert 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:

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

Der letzte Teil, welcher für die Bestimmung des PPO Training Loss benötigt wird, ist der Entropy Bonus. Dabei handelt es sich um die Entropie der Policy.
Es ergibt sich damit der bereits oben erwähnte PPO Training Loss 2.1:

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

2.3.3. PPO - Algorithmus

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

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für Actor θ und Critic w zufallsbasiert. Erstelle ein Experience Buffer EB.
2. Bestimmt mit dem State s und dem Actor-NN eine Action a. Dies geschieht durch $\pi_\theta(s)$.
3. Führe Action a aus und ermittle den Reward r und den Folgezustand s'.
4. Speichern von State s, Action a, Policy $\pi_{\theta_{\text{old}}}(s, a)$, Reward r, Folge-State s'. $(s, a, \pi_{\theta_{\text{old}}}(s, a), r, s') \longrightarrow EB$
5. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis N Zeitintervalle bzw. für N Spielepisoden erreicht sind.
6. Entnehme ein Mini-Batch aus dem Buffer $(S, A, \{\pi_{\theta_{\text{old}}}\}, R, S') \longleftarrow EB$.
7. Bestimmt die Ratio $r_t(\theta)$. 2.3.2.
8. Berechne die Advantages $\hat{A}_t(s, a)$. 2.3.2.
9. Berechne die Surrogate Losses $surr_1$ und $surr_2$. 2.3.2.
10. Bestimmt den PPO Loss. 2.1
11. Update die Gewichte des Actors und Critics. $\theta_{\text{old}} \longleftarrow \theta$ und $w_{\text{old}} \longleftarrow w$
12. Wiederhole alle Schritte ab Schritt 7 K-mal erneut durch.
13. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis der Verfahren konvergiert.

2.4. Deep Q-Network

Der DQN (Deep Q-Network-Algorithmus) ist ein weiterer Reinforcement Learning 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 den DQN anschaulich erklärt. Da jedoch in diesem Algorithmus weiterhin auf Teile der Loss-Function eingegangen wird, folge eine bereinigte Version, welche sich auch für den allgemeinen Gebrauch besser anbietet (Lapan, 2020, S. 149 f.).

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für (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.

Kapitel 3

Anforderungen

In Kapitel 2, wurden die Grundlagen für den weiteren Vergleich der Reinforcement Learning Agenten gelegt. Interessant erscheint dabei der Vergleich zweier unterschiedlicher Algorithmus Arten (DQN und PPO), da diese unterschiedlichen Vorgehensweisen aufweisen basieren.
Problematisch an einem solchen Vergleich ist die Objektivität. Wie kann sicher gestellt werden, dass die Ergebnisse vergleichbar sind?
Eine Antwort basiert auf festen Anforderungen bzw. Rahmenbedingungen, welche bei der Implementierungen, Datenerhebung und bei dem Vergleich gelten müssen. Dabei gibt es vier verschiedene Anforderungsbereiche, welche Forderungen 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.
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.

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 fressen?
Robustheit	Welcher Agent ist in der Lage in einer modifizierten Umgebung das Spielziel am besten zu erreichen? In Bezug auf Snake bedeutet dies: Welcher Agent ist in der Lage auf einem größeren oder kleineren Spielfeld die meisten Äpfel zu fressen?
Trainingszeit	Welcher Agent schafft es ein festes Ziel in der geringstmöglichen Zeit zu erreichen bzw. welcher Agent ist als erstes in der Lage durchschnittlich x Äpfel zu fressen.
Spielzeit	Welcher Agent schafft es im Durchschnitt am längsten einen terminalen Zustand zu vermeiden? Auf Snake bezogen: Welcher Agent schafft es am längsten nicht zu sterben.

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel soll es thematisch über den momentanen Stand der bereits durchgeführten Forschung gehen. Dabei sollen die Arbeiten gezielt nach den folgenden Aspekten durchsucht und diese anschließend diskutiert werden. Die Arbeiten wurde aufgrund ihres thematischen Hintergrundes zum Spiel Snake, in Verbindung mit dem RL, ausgewählt. Zu den Aspekten gehören:

- Optimierungsstrategien
- Reward Function
- Evaluationskriterien

4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning

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

Thematisch werden in diesem Paper drei Optimierungsstrategien verwendet, welche auf einen Baseline DQN (Referenz DQN) angewendet worden sind. Bei diesen Strategien handelt es sich um den Training Gap, die Timeout Strategy und den Dual Experience Replay.
Der Dual Experience Replay (Splited Memory) besteht aus zwei Sub-Memories, in

welchen Erfahrungen belohnungsbasiert sortiert und gespeichert werden. Mem1 besteht nur aus Erfahrungen, welche einen Reward größer als ein vordefinierter Grenzwert besitzen. Die restlichen Erfahrungen wandern in Mem2. Beim Lernen wird, zu beginn, eine überproportional größere Menge an Erfahrungen aus Mem1 ausgewählt, um den Lernerfolg zu beschleunigen. Im weiteren Lernverlauf wird dann das Entnahmeverhältnis normalisiert (Mem1: 50% & Mem2: 50%).
Der Training Gap stellt die Erfahrungen dar, welche der Agent, zum Zwecke des performanteren Lernens, nicht verarbeiten soll. Zu diesen zählt die Erfahrung direkt nach dem Konsum eines Apfel, sodass der Agent die Neuplatzierung eines Apfels erlernt. Da der Agent auf diesen Prozess keinen Einfluss hat, könnte die Verarbeitung dieser Daten den Lernerfolg mindern, weshalb die Training Gap Strategie etwaige Erfahrungen löscht.
Die Timeout Strategy sorgt für eine Bestrafung, wenn der Agent über eine vordefinierte Anzahl an Schritten P keinen Apfel mehr gefressen hat. Dabei werden die Rewards mit der letzten P Erfahrungen mit einem Malus verrechnet, was den Agenten dazu anhält die schnellste Route zum Apfel zu finden. Die Höhe des Malus ist antiproportional zur Länge der Snake (geringe Länge $\rightarrow$ großer Malus; große Länge $\rightarrow$ geringer Malus).
Die optimierte Reward Function, welche das Paper verwendet, besteht damit aus dem Standard Distanz Reward ohne Training Gap Erfahrungen. Letzterer bestimmt sich aus der Distanz zwischen Kopf und Apfel und der Länge der Snake, wobei der Factor der Länge wieder antiproportional den Reward beeinflusst. Sollte die Timeout Strategy auslösen, so werden die letzten P Erfahrungen entsprechend angepasst und in Mem2 verschoben.
Als maßgebliche Kriterien zur Evaluation der Leistung des DQN wurde die Performance, daher die Anzahl an gefressenen Äpfeln und die steps survived, also die überlebten Schritte herangezogen.

4.1.1. Diskussion

Sowohl die Training Gap also auch Dual Experience Replay und Timeout Strategy stellen vielversprechende Optimierungen dar, welche, auf experimentellen Resultaten basierend, gute Ergebnisse erzielen konnten. Eine Verwendung dieser Optimierungen im Kapitel 8 bietet sich daher an. Dennoch müssen, für die Anwendung der Optimierungen einige Anpassungen durchgeführt werden.
Der Dual Experience Replay ist zwar für DQN Agenten ohne weitere Einschränkungen einsetzbar, jedoch verhindert die Logik des PPOs eine direkte Anwendung. Alternativ könnte nicht mehr nach einzelnen Erfahrungen sortieren werden, sondern nach abgeschlossenen Spielepisoden. Der Mittelwert über alle erhaltenden Rewards wäre

dann der Sortierungsfaktor.
Fraglich ist, ob die Agenten, insbesondere der PPO, damit bessere Resultate erzielen können.
Auch beim Training Gap könnte der PPO Probleme bereiten, da nicht einfach Erfahrungen aus dem Memory entfernt werden sollten. Fraglich ist, ob die Entnahme dieser nicht lernenswerten Erfahrungen die Performance erhöhen oder senken.
Die Timeout Strategy bedarf ebenfalls einer Anpassung, da der verwendete DQN nach jedem Schritt lernt. Somit ist das editieren der Rewards im Nachhinein nicht möglich. Alternativ könnte der Reward direkt und stetig während des Spiels gesenkt werden, um die Funktionalität für den DQN herzustellen.
Erwähnenswert sind ebenfalls noch die Evaluationskriterien, welche sich auf die Performance und Spielzeit (time alive) beschränken. Eine weitere Betrachtung, beispielsweise der Robustheit oder Trainingszeit, wird vernachlässigt. Dies soll in dieser Ausarbeitung jedoch geschehen, da diese Faktoren ebenfalls wichtig für die voll umfassende Bewertung des Agenten sind, besonders in Real World Applications 1.1.

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

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

Eine dieser Optimierungen wurde des olfaktorischen Sinnes konzipiert. Der Odor Effect erzeugt um den Apfel drei aneinanderliegende Kreise, in welchen ein größerer Reward zurückgegeben wird als außerhalb der Kreise. Dabei unterscheiden sich die Kreise in der Höhe des Rewards, sodass der dritte den geringsten und der erste den größten Reward von allen zurückgibt ($r_1 > r_2 > r_3$, wobei r_x der Reward des x-ten Kreises darstellt). Dieser Zonen stellen den zunehmenden Duft von Nahrung dar, wobei der Geruch immer stärker wird um so näher man sich der Quelle nährt.
Eine weitere Optimierungsstrategie basiert auf dem Loop Storm Effect, welcher das Verhalten des Laufens um den Apfel beschreibt. Die Verfasser haben festgestellt, dass dieser Effekt zu einem sehr schlechten Lernerfolg und zu langen Trainingsdau-

ern führt. Darum haben Wu et al. einen dynamischen Positionsspeicher konzipiert, welcher Loops erkennt und diese durch das Zurückgeben einer Zufallsaktion, welche nicht auf dem Loop liegt, unterbricht. Experimentelle Ergebnisse des Papers zeigen, dass der Loop Storm Effect kaum mehr präsent ist.
Auf Basis einer Standard Reward Function, welche für das Essen eines Apfels +100 und für das Sterben -100 zurückgibt, wurden ein Versuch durchgeführt. Dem Agenten war es nicht möglich gegen die optimale Lösung zu konvergieren, aufgrund von Loop Storms und einer unzureichende Reward Function. Hingegen war es dem Agenten mit dem Odor Effect und mit der Breakout Loop Storm Strategy möglich eine Konvergenz on 8 Millionen Epochs (hier Trainingsdurchläufe) zu verzeichne.
Auch in diesem Paper bleibt die Performance maßgeblicher Evaluationsfaktor, wobei der Fokus auf den erreichten Reward gelegt wurde, welcher stark mit der Performance korreliert.

4.2.1. Diskussion

Loop Storm Effect und Odor Effect stellen nach den experimentellen Ergebnissen gute Ergänzungen für das Kapitel Optimierungen 8 dar. Dennoch ist fraglich, ob der Odor Effekt bessere Resultate als der Standard Distanz Reward aus Paper eins 4.1 erzielen kann. Auch bleibt die Frage offen, ob der Loop Storm Effect wirklich ein solches Problem darstellt. Eventuell ist es möglich diesen mit einer geschickten Belegung der Kreis-Rewards zu umgehen. Auch wirft die Präventionsstrategie des Loop Storm Effects Fragen auf. Was ist beispielsweise, wenn der Loop zwischen mehreren Kreisen (Loop Kreise nicht Odor Kreise) alterniert. Die Loop Detection würde durch ein solches Verhalten erschwert werden. Alternativ wäre es möglich, Zahl an gelaufenen Schritten um den Apfel zu zählen und ab einem Grenzwert eine Zufallsaktion zu wählen. Diese Methode ist zwar nicht so präzise aber allgemeiner anwendbar. Statistische Versuche sollen zeigen ob der Odor Effect und die Breakout Loop Storm Strategy die Performance und weitere Faktoren, welche im Paper nicht behandelt wurden, verbessern können, im Vergleich zu einem Referenz Modell.

4.3. Zusammenfassung

Sowohl "Autonomous Agents in Snake Game via Deep Reinforcement Learning" 4.1 als auch "UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene" 4.2 bieten einige Optimierungsstrategien, welche im weiteren Verlauf dieser Ausarbeitung angewendet werden sollen. Damit die Optimierungen auf beide Agenten Arten (DQN und PPO) angewendet werden können, wurden Anpassungen Diskutiert, welche sich im statistischen Vergleichen bewähren

müssen. Diese Vergleiche werden im Kapitel Optimierungen 8, durch das Erklären der derselbigen, vorbereiten und die Resultate im Kapitel 9 dargestellt.
Auch wurde ein näherer Blick auf die Reward Function gelegt, um Anregungen für die eigene Reward Function zu erhalten. Ähnlich verhält es sich auch mit den Evaluationskriterien, die jedoch aus Sicht dieser Ausarbeitung für unzureichende erklärt werden müssen.

Kapitel 5

Agenten

Ein zentraler Aspekt 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. 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 erweiterte Definition des Agentenbegriffs, welche um den entscheidenden Faktor der Hyperparameter erweitert wird. Ein Agent ist 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 Agenten, welche untereinander verglichen werden sollen, vorgestellt. Dargestellt werden die RL-Klassen, Algorithmus-Klassen und Hyperparameter.

Der DQN-1 ist ein Deep Q-Network Algorithmus, welches auf dem Q-Learning basiert und die folgenden Hyperparameter besitzt: gamma (γ) = 0.99, batch-size = 64, epsilon-decrement = 5e−6, epsilon-end = 0.05, max-mem-size = 2^{11} und lr= $2,5\text{e}{-4}$.

Der DQN-2 ist analog zu dem DQN-1 mit folgenden Veränderungen: gamma (γ) = 0.95 epsilon-end = 0.001 lr = $1,0\text{e}{-4}$

Der PPO-3 ist ein Proximal Policy Optimization Algorithmus, welcher auf der RL-Klasse des Policy Gradient basiert. Er besitzt folgende Hyperparameter: gamma (γ) = 0.99, K-epochs = 10, epsilon-clip = 0.2, batch-size = 64, entropy_coefficient = 0.001, critic-loss-coefficient = 0.5 lr-actor = 1.0e−3, lr-critic = 1.5e−3.

Der PPO-4 ist analog zum PPO-3 mit den folgenden Unterschieden in seinen Hyperparametern: gamma (γ) = 0.95, entropy-coefficient = 0.01.

Kapitel 6

Vorgehen

In diesem Kapitel soll das weitere Vorgehen thematisiert werden. Dazu erfolgt eine genaue Erklärung der weiteren Schritte.
Angestammtes Ziel dieser Ausarbeitung ist es eine Methodik zu finden, mit welcher man einen möglichst optimalen Agenten bestimmen kann. Dabei lässt das Wort optimal viel Spielraum für Evaluationskriterien. Im Kapitel 3 wurden daher Anforderungen definiert, nach welchen evaluiert wird. Mit Hilfe von Optimierungen, welcher in Kapitel 4 herausgearbeitet und konkretisiert wurden, soll die Leistungsfähigkeit der Agenten in den verschiedenen Evaluationskriterien erhöht werden. Weitere folgen im Kapitel 8. Um jedoch einen einen Vergleich durchführen zu können wird ein Basis (Baseline) benötigt, auf welcher verglichen wird.

6.1. Bemerkung zur Netzstruktur

Das Editieren der Netzstruktur wird in dieser Ausarbeitung nicht als Optimierung in Erwägung gezogen. Veränderungen der Netzstruktur haben weitreichende Folgen auf die Obs. und einige Hyperparameter, wie z.B. die Lernrate, welche bei Veränderung der Netzstruktur ebenfalls angepasst werden sollte. In dieser Ausarbeitung wird daher nur mit der Netzstruktur, welche unter 7.2.1 vorgestellt wurde. Ein Vergleich unterschiedlicher Netzstrukturen kann daher im Vorhinein unter den nicht optimierten Agenten geschehen.

6.2. Baseline Datenerhebung, Vergleich und Evaluation

Zu Erstellung grundlegender Vergleichsdaten sollen die in Kapitel 5 vorgestellten Agenten in zwei Trainingsdurchläufe untersucht werden. Die erhaltenden Daten bzw. trainierten Agenten werden dann unter den in 3.4 definierten Anforderungen evaluiert und vergleichen.

Um die Menge an Agenten zu minimieren und damit Konvergenz herbeizuführen, werden die zwei schlechtesten Agenten für ein Anforderung, wie z.B. Performance, Effizienz, Robustheit usw., disqualifiziert. Dies geschieht auf Basis der Baseline Daten.

6.3. Anwendung der Optimierungen

Nach dem Baseline Vergleich und Aussortierungen der schlechtesten Agenten sollen diese nun optimiert werden. Dazu werden vier Optimierungen auf die einzelnen Agenten angewendet, welche im Kapitel 8 näher thematisiert werden. Anschließend erfolgt eine Datenerhebung nach den Anforderungen 3.3 der optimierten Agenten, zuzüglich deines Vergleiches und einer Evaluation.
Nach dieser ist der optimale Agent mit der für jede Anforderung gefunden sein.

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) 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 (`https://gym.openai.com/`). Dieses bietet viele Methoden und Vorgaben in der Projektstruktur, welche das Implementieren erleichtern. So besteht das Snake Environment Package (snake_env), aus den wesentlichen Files:

- gui
- observation
- snake_env
- snake_game

7.1.1. Schnittstelle

Die grundlegende Schnittstelle des Snake Env. wird in dem File snake_env bereitgestellt. In diesem wird die Klasse SnakeEnv definiert, welche durch die Vererbung der Oberklasse gym.Env zentrale Methoden übernimmt. Zu diesen gehören die step, reset, render und close Methode.
Weiterhin bietet die selbst definierte Methode post_init die Möglichkeit zentrale Einstellungen, wie z.B. die Spielfeldgröße und GUI-Aktivierung, zu editieren. Die

step-Methode startet den Ausführungsprozess der vom Agenten ausgewählten Action, durch Aufrufen weiterer Methoden 7.1.2. Nach der Abarbeitung wird die neue Obs, der Reward, das done-flag und das has_won-flag übermittelt, wobei letzteres zur Bestimmung des Sieges dient. Die reset-Methode startet das Spiel von neuem. Zum Schluss wird dann noch die Obs. zurückgegeben. Die render-Methode ruft die Methoden für die visuelle Darstellung auf. Die close-Methode terminiert das Programm.
Auf Basis der oben genannten Methoden wird ersichtlich, dass es sich bei der Klasse SnakeEnv um eine Wrapper-Klasse handelt, die als Schnittstelle dient.

7.1.2. Spiellogik

Die Spiellogik, welche durch die step-Methode 7.1.1 angestoßen wird, befindet sich im snake_game File, welches eine Klasse namens SnakeGame definiert. Neben einigen Methoden werden im SnakeGame-Objekt auch viele spielbezogene Daten, wie z.B. das Spielfeld (ground), ein Player-Objekt (p), ein GUI-Objekt (gui) und die Spielfeldgröße (shape), den Schrittzähler (step_counter) und eine Hilfsvariable (has_grown), die für die Bestimmung des Reward benötigt wird.
Der Weiteren werden die folgenden Methoden implementiert:

Tabelle 7.1.: Methoden der SnakeGame Klasse

Methode	Erklärung
action	action ist für die Ausführung der Aktionen verantwortlich.
evaluate	evaluate bestimmt den, durch die Ausführung der Action, zu erhaltenden Reward.
observe	observe stößt den Generierungsprozess der Obs an, welcher ausgelagert im observation File liegt.
make_apple	make_apple erzeugt einen neuen Apfel auf dem Spielfeld.
reset_snake_game	reset_snake_game setzt den Spielfortschritt zurück und startet es von neuem.
max_snake_length (getter)	Gibt die maximale Länge der Snake zurück.
is_done (getter)	Liefert den Lebensstatus (player.done).

Zum Erhalt eines tieferen Verständnisses über die Spiellogik, soll diese exemplarisch erläutert werden. Bevor dies jedoch geschehen kann muss noch die Datenhaltungsklasse Player erwähnt werden, die ebenfalls im Snake_game File definiert ist. Diese speichert spielerbezogen Daten, wie z.B. die Position aller Schwanzglieder inkl. des Kopfes (pos, tail), Blickrichtung (direction), die Anzahl der Schritte seit dem

letzten Fressen (inter_apple_steps), den Lebensstatus (done) und einige Konstanten, welche für die Visualisierung benötigt werden (id, c_s und c_h). Zuzüglich besitzt die Player Klasse noch eine player_reset Methode und einige getter Methoden.

Da Snake ein zweidimensionales Spiels ist, wird eine gleich dimensionale Matrix (ground) zur Spieldarstellung verwendet. So kann mit dn Zeilen- und Spaltenindexen der Matrix die Position der Snake deutlich gemacht werden. So wird der Schwanz mit der Konstante c_s, der Kopf mit c_h, der Apfel mit -2 und das Ende der Snake mit -1, in der Matrix (ground) deutlich gemacht. Der Einfachheit halber werden die Zeilen- und Spaltenindexe, daher die Position, ebenfalls noch in einer List (tail) festgehalten. Dieser erleichtert später die Feststellung des Lebensstatus (done).

Mit dem folgenden Aktivitätsdiagramm soll der Fokus weiter auf die action-Methode gelegt werden. Diese wird wie folgt abgearbeitet. Als erstes wird der inter_apple_steps erhöht. Sollte dieser Zähler größer als, die vorher definierte, Obergrenze sein, so wird der Lebensstatus auf tot gesetzt (p.done = True) und die Methode wird terminiert. Im weiteren Verlauf wird diese Unterprozedur Abbruchprozedur genannt.
Anderenfalls wird als nächstes überprüft, um welche action es sich handelt, wobei die actions mit den Zahlen von null - zwei kodiert sind.

Tabelle 7.2.: Kodierung der Actions

Action	Erklärung
0	Die Snake ändert ihre Richtung um 90° nach links. Z.B. Von N $\longrightarrow$ W
1	Snake ändert ihre Richtung um 90° nach rechts. Z.B. Von N $\longrightarrow$ O
2	Die Richtung der Snake wird nicht verändert.

Entsprechende der Action wird die direction des Players angepasst. Die vier Himmelsrichtungen werden dabei mit den Zahlen von 0 - 3 dargestellt, wobei null Norden entspricht eins Osten usw.
Nach der Manipulation der direction des Players, wird pos, also die vorläufige neue Position, im Player-Objekt angepasst. Diese Änderung wird jedoch noch nicht sofort in die Matrix übertrage, da das Eintragen on Positionen außerhalb des Spielfeldes zu Fehlern führen würde. Es muss daher erst überprüft werden, ob die neue Position des Players im Spielfeld liegt. Sollte dies nicht der Fall sein, so wird die Abbruchprozedur aufgerufen.
Anderenfalls wird die neue Position in tail 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 ergibt, verglichen. Entspricht die Länge Snake der maximal mögliche, so wird die Abbruchprozedur aufgerufen. Ansonsten muss als nächster Schritt die Matrix aktualisiert werden. Jedoch muss vorher festgestellt werden, ob die Snake einen Apfel gefressen hat.
Sollte sie dies getan haben, so wird die neue Position des Kopfes in die Matrix eingepflegt, ein neuer Apfel wird auf einer zufälligen freien Stelle generiert, inter_apple_steps wird auf null und die Hilfsvariable has_grown wird auf True gesetzt. Letztere wird von der evaluate Methode verwendet, um di Höhe des rewards zu bestimmen, siehe 7.1.3.
Ist die Snake jedoch nicht gewachsen, so wird das letzte Schwanzglied aus Matrix und Liste gelöscht, um den Anschein von Bewegung zu erwecken. Zuzüglich wird die Hilfsvariable has_grown auf False gesetzt.
Zum jetzigen Zeitpunkt besteht immer noch die Möglichkeit, dass die Snake in sich selber gelaufen ist. Um dies festzustellen, wird tail auf Duplikate überprüft. Sollten sich Duplikate in tail befinden, wird die Abbruchprozedur aufgerufen.
Ansonsten wird zum Schluss noch über tail iteriert und die korrespondierenden Einträge der Matrix (ground) werden mit den Positionen von tail aktualisiert, wobei der Kopf und das letzte Schwanzglied mittels eines anderen Zahlenwert dargestellt werden.

7.1.3. Reward Function

Die evaluate Methode, welche den Reward bestimmt, befindet sich in der Snake-Game Klasse. Basierend auf dem letzten Zug wird in dieser Methode der Reward bestimmt. Dies geschieht nach folgenden Vorbild. Der Reward ist abhängig von drei Faktoren. Dem Fressen eines Apfels, dem Sieg und dem Verlust. Sollte keiner dieser genannten Faktoren eintreten, wird ein Reward von -0.01 zurückgegeben. Dies hält den Agenten dazu an den kürzesten Pfad zum Apfel zu finden, da jeder Schritt geringfügig bestraft wird.
War es der Snake möglich einen Apfel zu fressen so wird ein Reward von +1.0 zurückgegeben, da ein Sub-Goal erfüllt worden ist. Sollte die Snake gestorben sein, durch das Verlassen des Spielfeldes oder das Laufen in sich selbst oder das zu lange Umherlaufen, so wird ein Reward von -10 zurückgegeben, um dieses Verhalten in seiner Häufigkeit zu minimieren. Hat die Snake alle Äpfel gefressen, sodass das gesamte Spielfeld mit der Snake ausgefüllt ist, so wird ein Reward von +10 zurückgegeben, um ein solches Verhalten in seiner Häufigkeit zu maximieren.

7.1.4. Observation

Die Observation, welche das Snake Env. zurückgibt besteht aus zwei Teilen, der around_view (AV) und der scalar_obs (SO). Zur Erstellung der Obs wird die observe Methode in der SnakeGame Klasse aufgerufen. Diese ruft ihrerseits die make_obs Funktion auf, welches im observation File definiert ist. Mit Hilfe verschiedener Unterfunktionen wird dann die Obs generiert.
Die AV lässt sich dabei als ein Ausschnitt der Matrix (ground) beschreiben, welche einen festen Bereich um den Kopf der Snake abdeckt. Strukturen wie Wände und Teile des eigenen Schwanzes, welche vielleicht eine Sackgasse aufspannen, werden deutlich. Numerische ist die AV eine one-hot-encoded Matrix der Form (6x13x13).

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 AV Matrix sechs Channel (zweidimensionale Schichten) besitzt. Diese geben Aufschluss über folgende Informationen:

Tabelle 7.3.: Channel-Erklärung der Around_View (AV)

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

Vorteilhaft an der AV ist, dass, im Gegensatz zu den verwandten Arbeiten 4.1 und 4.2, nicht das gesamte Feld übertragen wurde sondern nur der wichtigste Ausschnitt,

was die Menge an zu verarbeiten Daten drastisch reduzieren kann. Des Weiteren ergeben sich keine Probleme mit der Input-Size der Convolutional Layer.
Ein Nachteil dieser Obs ist jedoch die Vollständigkeit. Sollte der blaue Punkt in 7.1 außerhalb des grauen Kasten und daher außerhalb der AV liegen, so besitzt der Agent keine Informationen über den Aufenthaltsort des Apfels. Auch Informationen wie z.B. der Hunger, also die verbleibenden Schritte bis das Spiel endet, die Distanzen zu den Wänden und zu exkludierten Schwanzteilen und die Blickrichtung (direction) der Snake.

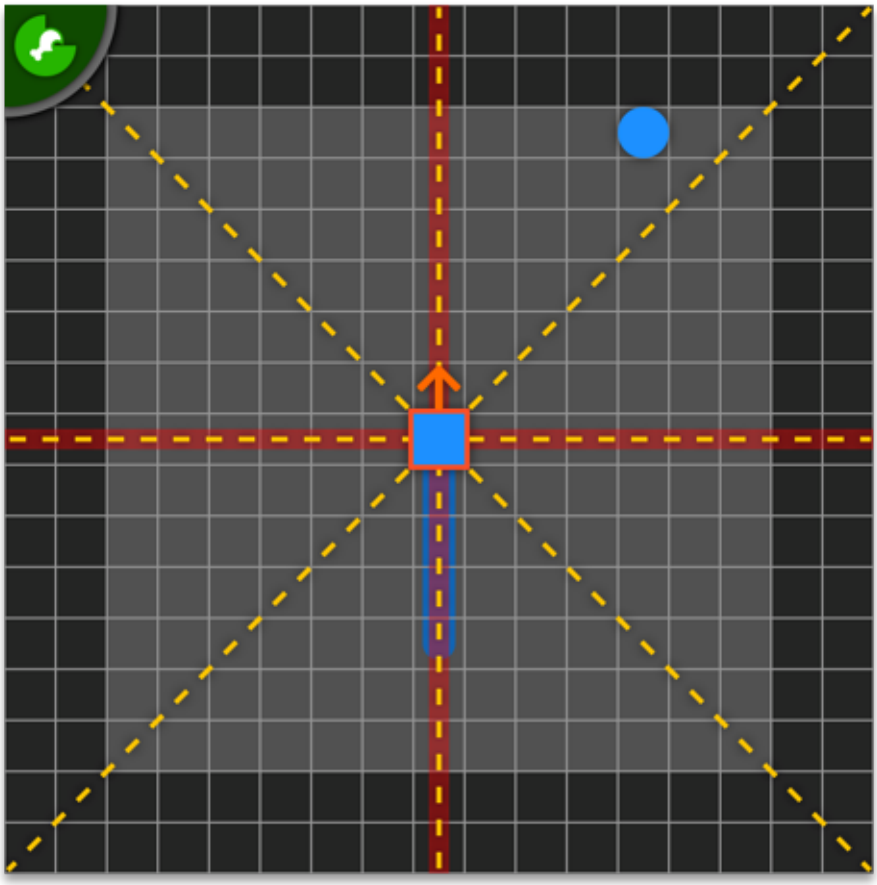

Abbildung 7.1.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar, wobei das rot umrandete Rechteck den Kopf darstellt. Die schwarzen Felder werden nicht von der AV abgedeckt, graue liegen innerhalb der AV. Die gelben gestrichelten Linien stellen ein X-Ray Distanzbestimmung dar. Der blaue Kreis stellt den Apfel dar und der grüne viertel Kreis oben links symbolisiert Hunger.

Aus diesem Grund wurde die AV mit der scalar_obs (SO) ergänzt. Diese beinhaltet skalare Informationen und ist eine Konkatenation aus X-Ray Distanzbestimmung, Hunger- Blickrichtungsanzeige und zwei Kompasse für die relative Positionsinformation zwischen Kopf und Apfel bzw. letztem Schwanzglied. Letztere sind eindimensionale Vektoren, welche über das One-Hot-Encoding anzeigen, ob sich das gesucht

Objekt relativ zum Kopf oberhalb, unterhalb oder in der selben Zeile (Matrixsicht) befindet. Analog verhält es sich mit der vertikalen Sicht.
Die Blickfeldanzeige ist ebenfalls one-hot-encoded und stellt mit seinem Vektor die vier Ausrichtungen Norden, Osten, Süden und Westen dar. Da der Hunger bei einer großen Differenz zwischen inter_apple_steps und max_steps einen kleinen und bei einer geringen Differenz einen großen Einfluss besitzen soll, wurde die Differenz durch eins geteilt. Nähren sich die beiden Werte, so rückt der resultierende näher an unendlich, da den Nenner immer kleiner wird. Um mit der Unendlichkeit auftretende Probleme zu umgehen wir zwei zurückgegeben, wenn die Differenz null ist.
In ähnlicher Weise wird mit den X-Rays Distanzbestimmungen verfahren. Bei ihnen handelt es sich um acht Distanzmesserlinien, die in 45° Abständen ausgesandt werden, siehe 7.1. Befindet sich das gesucht Objekt in dieser Linie, so wird die durch eins dividierte Differenz zwischen Kopf und Objekt zurückgegeben. Es wird nach Wänden, dem eigenem Schwanz und dem Apfel gesucht. Daher wird die X-Ray Distanzbestimmung in einem Vektor der Größe 24 (3 * 8 = 24) gespeichert.

7.1.5. Graphische Oberfläche

Im gui File wird eine Klasse GUI definiert, welche das Spiel mit Hilfe des Frameworks pygame (`https://www.pygame.org/`) darstellt. Dazu wird in der GUI-Klasse ein Oberfläche (screen) erzeugt, welche die Matrix ground darstellt, siehe 2.1. Die Methode update_GUI, welche von der SnakeGame Methode view aufgerufen wird, überschreibt dazu jeden einzelnen Eintrag des pygame-Spielfelds mit dem korrespondierenden Wert von ground. Spielfeld (GUI) und Matrix sind daher nicht direkt gekoppelt sondern müssen über update_GUI angeglichen werden.
Die Fenstergröße der Spieloberfläche wird dynamisch berechnet und kann über das Attribut Particle verändert werden, welches die Feldgröße eines einzelnen Matrixeintrags darstellt. Die draw Methode ist für das Generieren der einzelnen Spielfeld-Rechtecke zuständig und reset_GUI versetzt das Spielfeld zurück in den Ursprungszustand.

7.1.6. Testung des Environment

7.2. Agenten

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

7.2.1. Netzstruktur

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

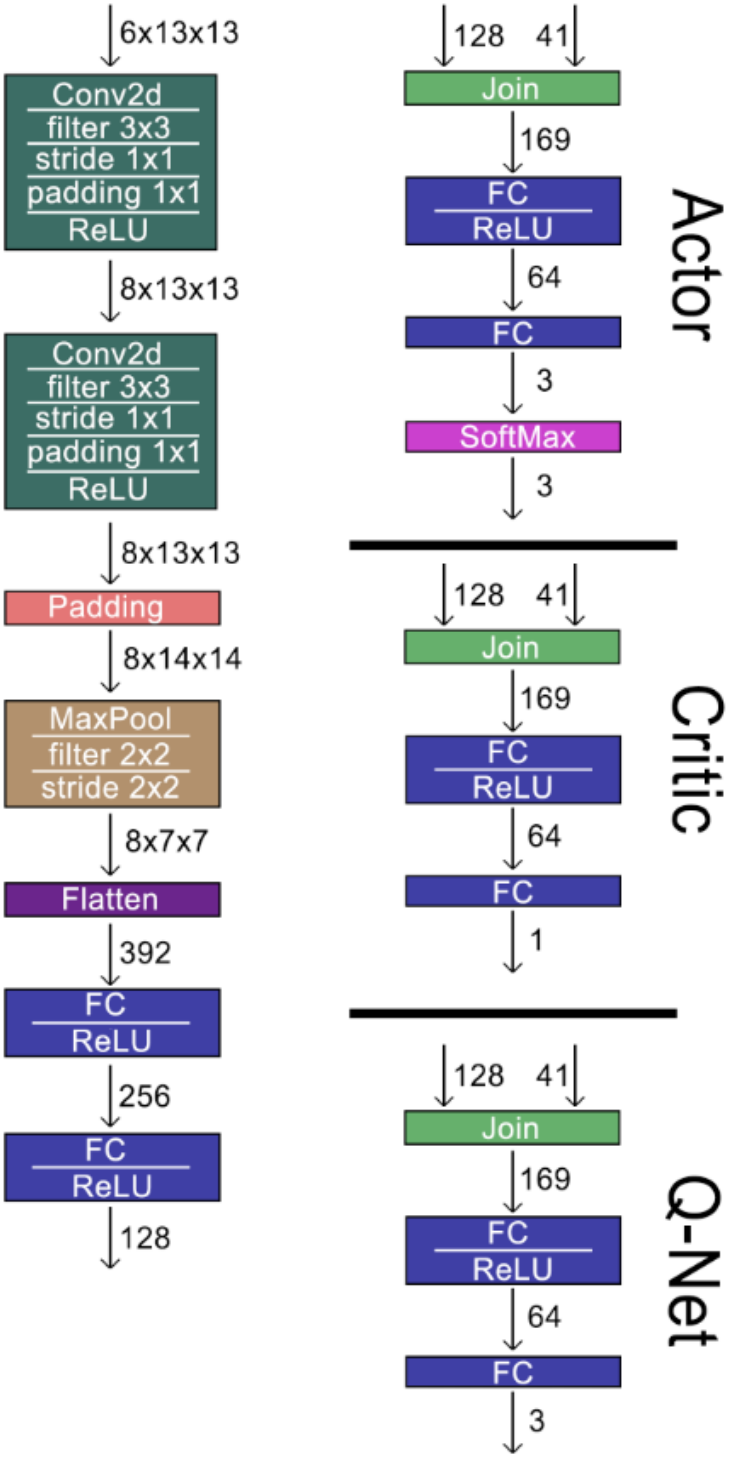

Abbildung 7.2.: Darstellung des NN, welches als Standard für den weiteren Vergleich dient.

Das Netz findet sich in den Files actor und critic der PPO Package und im q_net

des DQN Package. Diese Klassen besitzen eine forward Methode, welche die Obs (AV, SO), nach dem umwandelt in Tensoren, durch das NN propagiert und einen Output bestimmt. Dieser Prozess geschieht dabei folgendermaßen:
Zuerst wird die AV durch zwei Convolutional Layer mit einer ReLU Aktivierungsfunktion durch propagiert. Dabei erhöht sich die Channel-Anzahl auf acht, da die AV bereits sehr stark optimiert wurde und ein Anstieg auf 16 oder 32 Channeln nicht nötig ist. Die Feature Map wird während dieses Prozesses nicht minimiert, aufgrund eines Paddings. Dies soll den Informationsverlust an den Rändern minimieren. Danach werden allen Feature Maps eine Null-Zeile und Spalte hinzugefügt (Padding), damit beim Max-Pooling unter der Filtergröße und dem Stride von 2x2, auch die letzten Zeile und Spalte verarbeitet wird. Der Tensor besitzt nun die Form (8x14x14) Nach dem max-pooling besitzt der Tensor Feature Maps der Größe 7x7. Nach der Einebnung (Flatten) des Tensoren zu einem eindimensionalen Tensor-Vektor wird dieser durch zwei weitere Fully Connected Layer (FC) mit einer ReLU Aktivierungsfunktion durch propagiert. Der resultierende Tensor besitzt die Größe 1x128 und ist ein zwischen Ergebnis, da dieser nun mit der SO verbunden wird (Join).
Da das NN in beide Algorithmus-Arten verwendet wird, müssen drei unterschiedliche Netzwerkköpfe definiert werden, siehe 7.2 rechts. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nachdem der Joined Tensor (1x128) durch zwei weitere FC Layer mit ReLU Aktivierungsfunktion propagiert wurde, benötigt der Actor des PPO-Agenten eine Wahrscheinlichkeitsverteilung über alle Actions. Daher auch die Ausgabe von einem Tensor der Größe drei, auf welchen zur Erstellung der Wahrscheinlichkeitsverteilung SoftMax angewendet wird, siehe 7.2 rechts oben.
Die Critics der PPOs und die DQNs verwendet den rechts unten dargestellten Netzwerkkopf (ValueNet-Head) 7.2. Beim den Critics werden Tensoren mit einem einzigen Zahlenwert zurückgegeben, wohingegen bei den DQNs Tensoren mit drei Q-Values entsprechende der Aktionsanzahl zurückgegeben werden.

7.2.2. Testung der Agenten

Aufgrund der Tatsache, dass die Agenten zu einem großen Teil auf dem Zufall basieren, stellt sich eine Testung mittels Testmethoden als nicht durchführbar heraus. Dennoch wurden beide Agenten mit Debug-Methoden manuell getestet. Dabei stellten sie sich als funktionsfähig heraus. Als beleg für diese Behauptung dienen die Evaluationsresultate, welche ohne funktionsfähige Agenten nicht zu erbringen seien.

7.2.3. DQN

Der DQN Algorithmus ist einer der beiden Algorithmus-Arten, welche im Rahmen dieser Ausarbeitung, implementiert wurden. Diese Implementierung findet hauptsächlich

in vier Files statt, welche im dqn Package liegen, dass wiederum zum agents Package gehört. Im dqn File wird die Agentenklasse definiert, welche die Aktionsbestimmungsmethode act und die lern Methode enthält. Im memoryDQN File wird die Memory-Klasse definiert, welche den Replay-Buffer darstellt 2.4. Die Files dqn_train und dqn_play beinhalten die eigentlichen main-Methoden, welche Agenten und Env. erstellen und den Trainings- bzw. Spielprozess umsetzt.

Aktionsauswahlprozess

Zur Bestimmung der nächsten Action wird der act Methode die momentane Obs übergeben. Die Bestimmung der Actions durch den DQN Agenten ist maßgeblich vom ϵ Wertes abhängig 2.4, welcher das Verhältnis zwischen der Wahl einer zufälligen und einer NN-basierten Action bestimmt. Dabei wird folgendermaßen vorgegangen 2.4. Zuerst wird ein Zufallswert *rand* zwischen null und eins generiert, welcher mit ϵ verglichen wird. Wenn $rand < \epsilon$ so wird die Action mittels des NN bestimmt. Anderenfalls wird eine zufällige Action gewählt. Zur NN-basierten Bestimmung der Action werden aus der Obs zwei Tensoren generiert, welche zum vordefinierten Device (CPU oder GPU) geschickt werden. Danach folge die Prozedur, welche in 7.2.1 dargestellt ist.

Trainingsprozess

Der Trainingsprozess wird über die learn Methode ausgeführt und stellt sich dabei wie folge dar:
Zuerst wird überprüft, ob im Memory genügend Experiences (Exp) gespeichert sind, um einen Mini-Batch mit der zuvor definierten batch_size, zu extrahieren. Sollte dies nicht der Fall sein, wird die Methode terminiert. Anderenfalls wird ein Mini-Batch aus zufälligen Exp. ohne Duplikate gebildet.

Das Memory (Replay Buffer) wird durch Tensoren dargestellt, welche die Erfahrungen in einer zusätzlichen Dimension speichern. So besitzt der Tensor der AV die Form (max_mem_size, 6, 13, 13). Die Tensoren werden dabei wie eine Überlaufliste verwaltet, sodass wenn die Anzahl an Exp die max_mem_size überschreitet, die ältesten Einträge durch die neuen ersetzt werden.

Danach folgt die Bestimmung aller Q-Values der States (s) des Mini-Batch, wobei der Q-Value der zum State (s) gespeicherten Action entnommen und als q_eval, gespeichert wird. Anschließend werden alle Q-Values der Nachfolge-States (s_next) bestimmt und als q_next gespeichert. 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 q-target zu bestimmen, werden die maximalen Q-Values der s_next bestimmt, diskontiert und mit dem, im Mini-Batch gespeicherten, Reward addiert, siehe 2.10. Nachfolgend wird der MSE (Mean Squared Error) zwischen q_eval und q_target bestimmt. Zuletzt wird der Fehler Mit Hilfe des PyTorch Frameworks mit Hilfe von Backpropagation und Gradient Descent A.1 zurück propagiert und entsprechend die Parameter des NN angepasst.

Main-Methode

Die Hauptmethode oder auch main Methode genannt implementiert den eigentlichen Spiel und Trainingsablauf. Die main Methde des DQN ist im dqn Package im File dqn_train und dqn_play zu finden. Letzteres wird für das reine visuelle Spielen genutzt und wird daher nicht weiter betrachtet.
Um den Spiel- und Trainingsablauf durchzuführen, wird in dem File dqn_train eine Methode train_dqn definiert, welche wichtige Hyperparameter des Ablaufen, wie z.B. die Anzahl der zum Training zu absolvierenden Spiele (N_ITERATIONS), die Lernrate (LR) und die Spielfeldgröße (BOARD_SIZE), die Batch Size (BATCH_SIZE), die Maximale Größe des Speichers (MAX_MEM_SIZE), Epsilon Decrement (EPS_DEC) und Epsilon End (EPS_END) übergeben bekommt.
Zu Beginn werden Datenhaltungslisten apples, wins, dtime, eps, steps_list initialisiert, welche für jedes absolvierte Spiel die, dem Namen der List entsprechenden, Werte speichert. Nach der Erstellung des Agenten und Environments startet der Spielverlauf.
Dabei wird wie in 2.2.2 vorgegangen. Der Agent erhält eine Obs, bestimmt seine Action und diese wird sogleich im Env ausgeführt. Danach werden die neue Obs sowie Reward und weitere Statusinformationen, wei z.B. das done-Flag usw., ausgegeben. Diese Daten werden im Memory für das sich anschließende Training gespeichert. Dieses wird wie in 7.2.3 durchgeführt. Danach werden die oben genannten Datenhaltungslisten aktualisiert und die Prozedur beginnt von neuem. Wenn diese N_ITERATIONS Spiele alle absolviert wurden werden die Erhobenen Daten aus den Listen in einem CSV-File gespeichert und die Methode wird terminiert.

7.2.4. PPO

Der PPO Algorithmus stellt die zweite Algorithmus-Art dar. Seine Implementierung findet im ppo Package, welches die Files actor, actor_critic, critic, memoryPPO, ppo, ppo_play und ppo_train beinhaltet. actor und critic implementieren die entsprechenden NN, actor_critic stellt eine Verwaltungs-Klasse dar, welche den Optimizer und die act (choose_action) und die evaluate Methode beinhaltet, wobei letztere für den

Trainingsprozess benötigt wird. MemoryPPO definiert das Memory (Replay Buffer), welches Erfahrungen speichert und im ppo File wird die Agentenklasse mit der learn Methode erstellt. ppo_train und ppo_play besitzen analoge Funktionalitäten wie in 7.2.3 eingangs erwähnt.

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird im actor_critc File implementiert. Von diesem existieren zwei Instanzen, welche die alte und neue Policies darstellen. Der act Methode der alten Policy wird die momentane Obs, welche aus AV (around_view) und SO (scalar_obs) besteht, übergeben und zu PyTorch Tensoren umgewandelt. Danach wird diese durch das Actor-NN (Policy-NN) durch propagiert, sodass eine Policy übergeben wird. Auf dieser Wahrscheinlichkeitsverteilung wird dann die nächste Action bestimmt, welche sogleich mit der logarithmierten Wahrscheinlichkeiten der ausgewählten Action und der zu Tensoren umgewandelten Obs, ausgegeben wird.

Trainingsprozess

Der Trainingsprozess ist in der Agentenklasse implementiert, welche sich im ppo Package befindet. Zu Beginn wird aus dem Memory ein oder mehrere Mini-Batch/es generiert. Um den Return 2.3.2 zu erhalten, werden die einzelnen in Rewards diskontiert. Sollte ein terminalen Zustand vorhanden sein, werden die discounted Rewards auf null gesetzt, da diese dem zu erwarteten Reward bis zum Ende der Spielepisode entsprechen. Da die Episode terminiert, können keine weiteren Rewards mehr gesammelt werden.
Um ein gleichmäßigeres Lernen zu unterstützen, werden die Rewards nach dem sie zu einem Tensoren umgewandelt worden sind, normalisiert. Danach wird die folgende Prozedur (K_epochs) mal ausgeführt, um das NN zu trainieren. Danach terminiert die learn Methode
Zuerst wird die evaluate Methde der ActorCritic-Klasse aufgerufen welche sich im actor_critic File befindet. Dieser werden die Obs aus dem Mini-Batch bzw. Mini-Batches übergeben. Zusätzlich werden noch die alten korrespondierenden Actions übergeben, welche zur Bestimmung von $\pi_\theta(a_t|s_t)$ 2.3.2 dient. Evaluate gibt neben $\pi_\theta(a_t|s_t)$ auch noch die State Values 2.3.2 und die Entropie der neuen Policy π_θ aus, siehe 2.3.2.
2.3.2 entsprechend werden als nächstes die Probability Ratios über alle Erfahrungen des Mini-Batch/es bestimmt. Daraufhin folgt die Bestimmung der Advantages, welches entsprechend zu 2.3.2 geschieht. Nach der Berechnung beider Surrogates Loss wird die Surrogate Objective Loss gebildet 2.3.2.
Als nächstes erfolgt die Bestimmung des Value Loss 2.8 und die des Entropy Loss,

wobei letzterer durch die evaluate Methode bereits bestimmt wurde.
Zum Schluss werden alle Losses entsprechend 2.3.2 zusammengefügt. Die NN's werden daraufhin durch mit Hilfe des PyTorch Frameworks durch Backpropagation und Gradient Desent A.1 angepasst.
Da der PPO mit unterschiedlichen Policy arbeitet, wird vor der Terminierung der Trainingsmethode die alte Policy $\pi_{\theta_{\text{old}}}$ durch die neu erzeugte π_θ ersetzt. Trainiert wird immer auf der neuen Policy π_θ und die Daten werden stets auf der alten $\pi_{\theta_{\text{old}}}$ erhoben.

Main-Methode

Die main Methode des PPO ist bis auf kleinere Unterschiede analog zu der des DQN Algorithmus, siehe 7.2.3. Sie befindet sich einzig im ppo_train File, in ihr werden keine DQN spezifischen Daten, wie z.B. epsilon, abgespeichert und es existieren zwei Lernraten für Actor und Critic.

Kapitel 8

Optimierungen

In diesem Kapitel werden die anzuwendenden Optimierungen vorgestellt und erklärt, welche nach dem Basis Vergleich die Leistung in den einzelnen Evaluationskategorien noch weiter verstärken soll. Zu diesem Zweck sollen vier Optimierungen auf die Baseline Agenten (Agenten ohne Optimierungen) angewendet werden, welche aus den verwandten Arbeiten 4 und eigenen Ideen stammten.

8.1. Optimierung 1 - Dual Experience Replay

Die Idee für Dual Experience Replay oder auch hier Splited Memory genannt, stammt aus der Arbeit „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ und wurde in 4.1 bereits analysiert und diskutiert.
Diese Optimierung zielt darauf ab, den Replay Baffer (Memory) zu zweiteilen, sodass Mem1 und Mem2 entstehen. In Mem1 werden ausschließlich Erfahrungen gespeichert, welche einen Reward vorweisen können, der größer als ein vordefinierter Grenzwert ist. In Mem2 werden alle übrigen Erfahrungen gespeichert. Diese Aufteilung zielt darauf ab, dass zu beginn ein größerer Anteil an guten Erfahrungen für das Lernen verwendet wird, um das Lerntempo zu erhöhen. Den Verfassern schwebt ein Verhältnis von (80% guten und 20 % schlechteren Erfahrungen vor). Dieses Verhältnis normalisiert sich über die Trainingszeit, daher zum Verhältnis von (50% zu 50%).
Wie bereits in 4.1 angesprochen, ist die erfahrungsorientierte Aufteilung schlecht bis gar nicht umsetzbar, da die Rewards des PPO im Nachhinein noch diskontiert werden müssen. Dafür muss die komplette Episode an Erfahrungen in ihrer ursprünglichen Reihenfolge vorhanden sein, was nach dem Sortieren nicht der Fall wäre. Daher scheidet ein Sortieren Rewards aus.
Alternativ wurde vorgeschlagen, dass die Sortierung nicht belohnungsbasiert, sondern Episoden basiert geschehen kann. Es werden daher die besten Episoden, gemessen an ihren Scores, in die Memories einsortiert.

Zu diesem Zweck wird eine weitere Klasse SplitMemory in den Memory Files des DQN und PPO definiert, welchen diese beschriebene Funktionalität bereitstellt. Diese enthält drei Instanzen der ursprünglichen Memory-Klasse, Mem1, Mem2 und ein temporären Memory. In letzteren werden die Erfahrungen temporär eingelagert bis die Episode terminiert ist. Dann wird anhand des gesamt Scores der Episode die Zuteilung in Mem1 oder Mem2 erfolgen.
Beim DQN Memory wird zusätzlich darauf geachtet, dass beim Einfügen einer ganzen Episode an Erfahrungen die Ring-Buffer Eigenschaften des Memory erhalten bleiben. Beim Memory des PPO ist dies nicht nötig.

8.2. Optimierung 2 - Joined Reward Function

Die Joined Reward Function wurde im Paper „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ vorgestellt und in 4.1 erklärt. Sie setzt sich aus drei Teilen zusammen. Die Basis bildet ein Distanz Reward, welcher dessen Höhe antiproportionale zur Distanz ist. Um unerwünschte Lerneffekte, von beispielsweise der Neuerzeugung eines Apfels, zu verhindern wird werden diese Erfahrungen aus dem Memory gelöscht, was die Training Gap Strategie und den zweiten teil der Reward Function darstellt. Zur Verstärkung des Pathfindings wird die Timeout Strategy angewendet, welche den Agenten für nicht zielgerichtetes Verhalten, wie z.B. das unnötige Umherlaufen, bestraft.

Die Implementierung dieser neuen Reward Function findet in der SnakeGame Klasse im snake_game File statt. Dort wird die evaluate Methode erweitert, sodass das Wechseln zwischen verschiedenen Reward Functions ermöglicht wird. Dabei stellt die die Distanz Reward Function wie folgt dar:

$$r_{distanz}(dis, len, size) = \frac{10}{dis} \times \frac{len}{size} \tag{8.1}$$

dis stellt die Distanz gemessen mit der Euklidischen Norm dar, len ist die Länge der Snake und $size$ ist die Größe des Spielfeldes (bei einer Spielfeldform von 8x8 ergibt sich einer Größe von 64).
Da die Training Gap Strategy zu Problemen beim PPO führen würde, wird diese nicht beachtet. Die Timeout Strategy wird in abgewandelter Form, wie in 4.1.1 diskutiert, implementiert. Dies geschieht nach folgender Formel:

$$r_{timeout}(steps, len) = \frac{1}{100} \times \frac{steps}{len} \tag{8.2}$$

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

8.3. Optimierung 3 - Odor Reward Function & Loop Storm Strategy

Die Odor Reward Function wurde im Paper „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ vorgestellt um in 4.2 erklärt und diskutiert. Die Odor Reward Function basiert auf dem olfaktorisch Sinn. Dabei wird um den Apfel drei Geruchszonen generiert, in welchen der Reward größer ist als in der geruchsfreien Zone. Die drei Geruchszonen unterscheiden sich in ihrer Intensität der Geruch in Zone eins $z1$ ist größer als in Zone zwei $z2$. Daher gilt für den Geruch und damit auch für den Reward $z1 > z2 > z3 >$ außerhalb der Zonen. Da RL Agenten nach immer größer werdenden Rewards sterben besteht eine große Gefahr, dass der Agent um den Apfel rotiert, um so einen größeren Reward in dieser Episode zu erreichen. Um diesen sogenannte „Loop Storm Effect“ zu vermeiden, setzten die Verfasser auf eine dynamischer Array welches Loops erkennt und bei Eintreten eine zufällige Action auswählt um den Loop zu unterbrechen.

Auch die Odor Reward Function wird in der SnakeGame-Klasse, welche sich im snake_game File befindet, implementiert. Aufgrund der geringen Größe des Standard Spielfeldes (8x8), werden nur zwei Zonen mit einer jeweiligen Breite von einem Kästchen verwendet. So soll ein Reward von 0,5 in der zweiten und ein Reward von 1,5 in der ersten Zone zurückgegeben werden. Die loop Strom Strategy wird wie in 4.2.1 implementiert. Es werden daher die Schritte in der Zone gezählt und bei Übersteigen der Anzahl an Kästchen in den Zone, wird eine zufällige Actions ausgeführt, welche den Loop unterbricht. Diese Erweiterung wird in der Aktionswahlprozedur, welche in 7.2.3 und 7.2.4 dargestellt sind, implementiert.

8.4. Optimierung 4 - Dynamische Lernrate

Die vierte Optimierung versucht das Lernen des Agenten durch das stetige Anpassen der Lernrate (LR) zu verbessern. Mit Hilfe eines vom PyTorch bereitgestellten Schedulers lässt sich die Lernrate während des Trainingsprozess manipulieren. Dabei wird diese Optimierung in der main Methode implementiert.
Die Lernrate wird immer dann mit 0.95 multipliziert und als neue LR gesetzt, sobald keine Performance Steigerung in den lernen 200 Schritten erfolgt ist.

Kapitel 9

Evaluation

Literaturverzeichnis

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

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

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

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

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

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

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

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

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

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

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

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

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

Anhang A

Anhang

A.1. Backpropagation und das Gradientenverfahren

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

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

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

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

A.2. Tensoren

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

A.3. Convolution Layers

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

A.3.1. Convolutional Layer

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

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

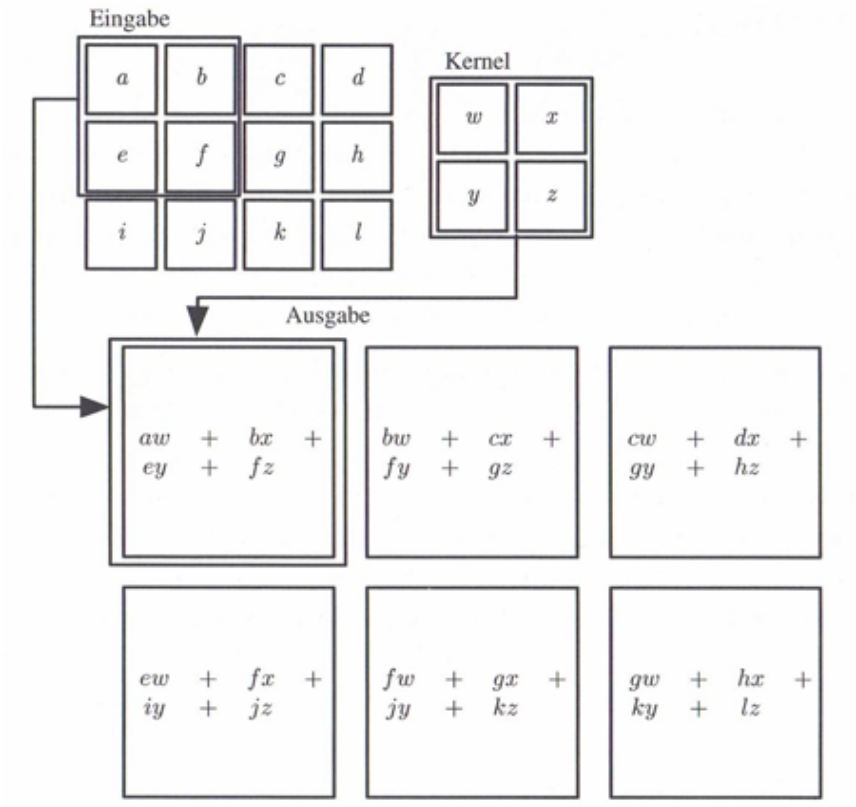

Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

A.3.2. Pooling Layer

Pooling beschreibt die Minimierung der Feature Maps, wie es bereits in A.3.1 angesprochen wurde. Zu diesem Zweck wird in A.2 ein Kernel und Stride der Form (2x2) gewählt, welcher ein Max-Pooling durchführen soll. Dieser Kernel bewegt sich über die Feature Map entsprechende des Strides und gibt den maximalen Wert innerhalb der Kernels wieder. Somit wird aus einer (4x4) eine (2x2) Feature Map. (Goodfellow u. a., 2018, S. 379 ff.)

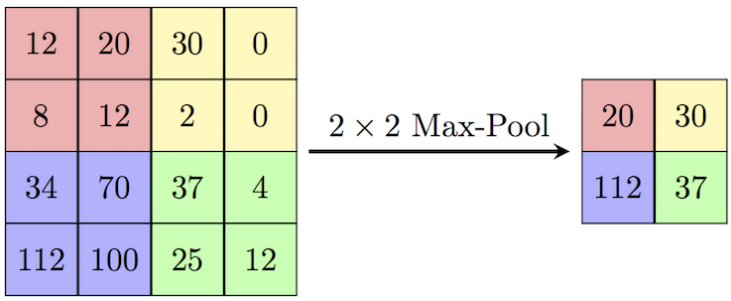

Abbildung A.2.: Darstellung des Max-Poolings. `https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png`

A.3.3. Fully Connected Layer

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

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

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

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