



CARL VON OSSIEZKY UNIVERSITÄT OLDENBURG

INFORMATIK
BACHELORARBEIT

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

Autor:
Lorenz Mumm

Erstgutachter:
Apl. Prof. Dr. Jürgen Sauer

Zweitgutachter:
M. Sc. Julius Möller

Abteilung Systemanalyse und -optimierung
Department für Informatik

Oldenburg, 5. September 2021

Inhaltsverzeichnis

Abbildungsverzeichnis	v
Tabellenverzeichnis	vi
Abkürzungsverzeichnis	vii
1. Einleitung	1
1.1. Motivation	1
1.2. Zielsetzung	2
2. Grundlagen	3
2.1. Game of Snake	3
2.2. Reinforcement Learning	4
2.2.1. Vokabular	5
2.2.2. Funktionsweise	8
2.2.3. Arten von RL-Verfahren	8
2.3. Proximal Policy Optimization	10
2.3.1. Actor-Critic Modell	11
2.3.2. PPO Training Objective Function	11
2.3.3. PPO - Algorithmus	16
2.4. Deep Q-Network	16
2.4.1. DQN - Algorithmus	18
3. Anforderungen	19
3.1. Anforderungen an das Environment	19
3.1.1. Standardisierte Schnittstelle	19
3.1.2. Funktionalitäten	19
3.2. Anforderungen an die Agenten	20
3.2.1. Funktionalitäten	20
3.2.2. Parametrisierung	20
3.2.3. Diversität der RL Algorithmen	21
3.3. Anforderungen an die Datenerhebung	21
3.3.1. Mehrfache Datenerhebung	21
3.3.2. Datenspeicherung	21

3.4. Anforderungen an die Statistiken	22
3.5. Anforderungen an die Evaluation	22
4. Verwandte Arbeiten	24
4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning	24
4.1.1. Diskussion	26
4.2. UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene	27
4.2.1. Diskussion	28
4.3. Zusammenfassung	29
5. Konzept	30
5.1. Vorgehen	30
5.2. Environment	32
5.2.1. Spiellogik	32
5.2.2. Schnittstelle	38
5.3. Agenten	38
5.3.1. Netzstruktur	38
5.3.2. DQN	40
5.3.3. PPO	43
5.3.4. Vorstellung der zu untersuchenden Agenten	45
5.4. Optimierungen	46
5.4.1. Optimierung A - Joined Reward Function	46
5.4.2. Optimierung B - Anpassung der Lernrate	47
5.5. Datenerhebung und Verarbeitung	48
5.5.1. Datenerhebung	48
5.5.2. Datenverarbeitung und Erzeugung von Statistiken	49
6. Implementierung	50
6.1. Snake Environment	50
6.2. AV-Network	52
6.3. DQN	52
6.3.1. Q-Network	53
6.3.2. Memory	53
6.3.3. DQN-Agent	53
6.4. PPO	55
6.4.1. Actor und Critic	55
6.4.2. ActorCritic	56
6.4.3. Memory	57
6.4.4. Agent	57

6.5. Train Methoden	59
6.5.1. Test Methoden	61
6.6. Speicherung	61
6.7. Statistik	62
7. Evaluation	64
7.1. Ergebnisevaluation der Vergleiche	64
7.1.1. Evaluation der Baseline Vergleiche	64
7.1.2. Evaluation der Optimized Vergleiche	70
7.2. Anforderungsevaluation	73
7.2.1. Anforderungsevaluation der Environment	73
7.2.2. Anforderungsevaluation der Agenten	74
7.2.3. Anforderungsevaluation an die Datenerhebung	74
7.2.4. Anforderungen an die Statistiken	75
7.2.5. Anforderungen an die Evaluation	75
8. Fazit	76
8.1. Bewertung der Forschungsfrage	76
8.2. Ausblick	77
Literaturverzeichnis	78
A. Anhang	80
A.1. Backpropagation und das Gradientenverfahren	80
A.2. Tensoren	81
A.3. Convolution Neural Networks	81
A.3.1. Convolutional Layer	81
A.3.2. Pooling Layer	83
A.3.3. Fully Connected Layer	83
B. Anhang zur Implementierung	84
B.1. Around-View	84
B.2. Raytracing Distanzbestimmung	85
B.3. AV-Network	86
C. Anleitung	88
C.1. PPO Train Startargumente	88
C.2. DQN Train Startargumente	89
C.3. Test Startargumente	90

Abbildungsverzeichnis

2.1. Game of Snake	3
2.2. Reinforcement Learning	8
5.1. Flussdiagramm des Vorgehens	31
5.2. Spiellogik	32
5.3. Spielablauf	33
5.4. Observation	36
5.5. Schnittstelle	38
5.6. AV-Network	39
5.7. Critic- und Q-Net-Tail	40
5.8. DQN-Agent	40
5.9. DQN-Aktionsbestimmung	41
5.10. PPO-Agent	43
5.11. Agenten	45
6.1. Sequenzdiagramm	51
6.2. Ablaufdiagramm der action Methode	52
7.1. Baseline Vergleich Performance	65
7.2. Baseline Vergleich Siegrate	67
7.3. Baseline Vergleich Effizienz	68
7.4. Optimized Vergleich Performance	70
7.5. Optimized Vergleich Siegrate	72
A.1. Darstellung Convolutional Computation	82
A.2. Darstellung MaxPooling	83

Tabellenverzeichnis

2.1. Formelemente	12
3.1. Zu erhebende Daten	21
3.2. Evaluationskriterien	22
5.1. Kodierung der Aktionen	34
5.2. Channel-Erklärung der Around_View (AV)	35
7.1. Testdatenauswertung der Performance	66
7.2. Testdatenauswertung der Baseline Siegrate	66
7.3. Testdatenauswertung der Baseline Effizienz	67
7.4. Testdatenauswertung der Robustheit für Baseline Vergleich 1	69
7.5. Testdatenauswertung der Robustheit für Baseline Vergleich 2	69
7.6. Testdatenauswertung der Performance	71
7.7. Testdatenauswertung der Optimized Siegraten	71
7.8. Testdatenauswertung der Robustheit für Optimized Vergleich 1	73
7.9. Testdatenauswertung der Robustheit für Optimized Vergleich 2	73

Abkürzungsverzeichnis

KI	Künstliche Intelligenz
RL	Reinforcement Learning
DQN	Deep Q-Network
DQL	Deep Q-Learning
DDQN	Double Deep Q-Network
PPO	Proximal Policy Optimization
SAC	Soft Actor Critic
A2C	Advantage Actor Critic
Env	Environment
Obs	Observation
AV	around_view
SO	scalar_obs

Kapitel 1

Einleitung

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

1.1. Motivation

In den letzten Jahren erregte das Reinforcement Learning eine immer größere Aufmerksamkeit. Siege über die amtierenden Weltmeister in den Spielen GO oder Schach führten zu einer zunehmenden Beliebtheit des RL. Neue Verfahren, wie z.B. der Deep Q-Network Algorithmus auf dem Jahr 2015 Mnih u. a. (2015), der Proximal Policy Optimization (PPO) aus dem Jahr 2017 Schulman u. a. (2017) oder der Soft Actor Critic (SAC) aus dem Jahr 2018 Haarnoja u. a. (2018), haben ihr Übriges getan, um das RL auch in anderen Bereichen weiter anzusiedeln, wie z.B. der Finanzwelt, im autonomen Fahren, in der Steuerung von Robotern, in der Navigation im Web oder als Chatbot Lapan (2020).

Durch die jedoch große Menge an RL-Verfahren gerät man zunehmend in die problematische Situation, sich für einen diskreten RL Ansatz zu entscheiden. Weiter erschwert wird dieser Auswahlprozess noch durch die Tatsache, dass die einzelnen Agenten jeweils untereinander große Unterschiede aufweisen. Auch existieren häufig mehrere Ausprägungen eines RL-Verfahrens, wie z.B. der Deep Q-Network Algorith-

mus (DQN) und der Double Deep Q-Network Algorithmus (DDQN). Die Wahl des passenden Agenten kann großen Einfluss auf die Performance und andere Bewertungskriterien haben, (Bowi 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 Katastrophen-situationen, wie z.B. auf explodierten Rohölbohrplattformen oder Raffinerien, finden und retten sollen. Auch kann das Liefen von wichtigen Gütern, beispielsweise medizinischer Natur, in solche Gebiete kann durch die Forschung am Spiel Snake möglich gemacht werden.(Chunxue u. a. (2019))

1.2. Zielsetzung

Basierend auf der Motivation ergibt sich folgende Fragestellung für diese Ausarbei-tung:

Wie kann an einem Beispiel des Spiels Snake für eine nicht-triviale gering-dimensionale Umgebung ein möglichst optimaler RL Agent ermittelt werden?

Diese Fragestellung zielt darauf ab für die Umgebung Snake einen möglichst opti-malen Agenten zu bestimmen, welcher spezifische Anforderungen erfüllen soll.

Basierend auf der Forschungsfrage ergibt sich ein Mehrwert für die Wissenschaft. Durch Abnahme des Entscheidungsfindungsprozesses müssen Forscherinnen und For-scher wie auch Anwenderinnen und Anwender von RL-Verfahren nicht mehr unre-flektiert irgendeinen RL Agenten auswählen, sondern können auf Grundlage der Methodik und der daraus hervorgehenden Daten den passenden Agenten bestim-men.

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

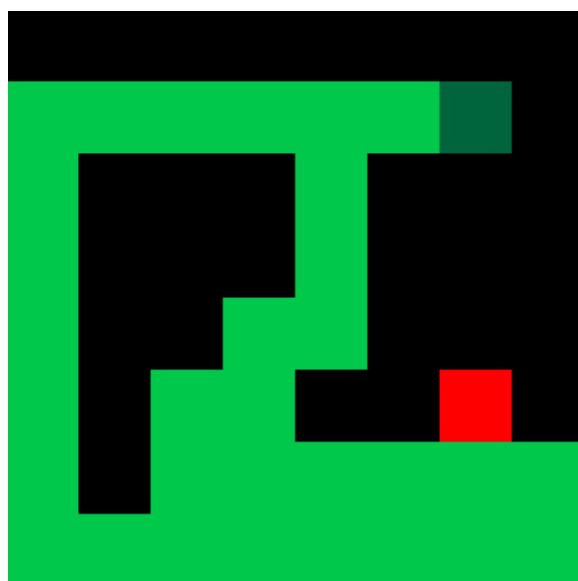


Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

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 beschreibt 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 Erkennung von Datenrepräsentationen, werden auch im RL verwendet.

Auf Menschen wirkt das RL, im Vergleich zu den anderen Disziplinen des Machine Learnings, am nachvollziehbarsten. Dies liegt an der Lernstrategie die sich dieses Verfahren zu Nutze macht. Beim RL wird anhand eines „trial-and-error“, Verfahrens gelernt. Ein gutes Beispiel für eine solche Art des Lernens ist die Erziehung eines Kindes. Wenn eben dieses Kind etwas gutes tut, dann wird es belohnt. Angetrieben von der Belohnung, versucht das Kind dieses Verhalten fortzusetzen. Entsprechend wird das Kind bestraft, wenn es etwas schlechtes tut. Schlechtes Verhalten kommt weniger häufig zum Vorschein, um Bestrafungen zu vermeiden. (Sutton und Barto, 2018, S.1 ff.)

Beim RL funktioniert es genau so. Das ist auch der Grund dafür, dass viele der Aufgaben des RL dem menschlichen Arbeitsspektrum sehr nahe sind. So wird das RL beispielsweise im Finanzhandel eingesetzt. Auch im Bereich der Robotik ist das RL auf dem Vormarsch. Wo früher noch komplexe Bewegungsabfolgen eines Roboterarms mühevoll programmiert werden mussten, ist es heute bereits möglich Roboter

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

2.2.1. Vokabular

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

Agent

Im Zusammenhang mit dem RL ist häufig die Rede von Agenten. Sie sind die zentralen Instanzen, welche die eigentlichen Algorithmen, wie z.B. den Algorithmus des Q-Learning oder eines Proximal Policy Optimization, in eine festes Objekt einbinden. Dabei werden zentrale Methoden, Hyperparameter der Algorithmen, Erfahrungen von Trainingsläufen, wie auch das NN in die Agenten eingebunden. (Lapan, 2020, S. 31)

Bei den Agenten handelt es sich gewöhnlich um die einzigen Instanzen, welche mit dem Environment (der Umgebung) interagieren. Zu dieser Interaktionen zählen das Entgegennehmen von Observations und Rewards, wie auch das Tägeln 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 liefern, 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 Datenübermittlungswäge. Bei ihnen handelt es sich um Handlungen welche im Env. ausgeführt werden. Actions können z.B. erlaubte Züge in Spielen, das Abbiegen im autonomen Fahren oder das

Ausfüllen eines Antragen sein. Es wird ersichtlich, dass die Actions, welche ein RL Agent ausführen kann, prinzipiell nicht in der Komplexität beschränkt sind. Dennoch ist es hier gängige Praxis geworden, dass arbeitsteilig vorgegangen werden soll, da der Agent ansonsten zu viele Ressourcen in Anspruch nehmen müssten.

Im Environment wird zwischen diskreten und stetigen Aktionsraum unterschieden. Der diskrete Aktionsraum umfassen eine endliche Menge an sich gegenseitig ausschließenden Actions. Beispielhaft dafür wäre das Gehen an einer T-Kreuzung. Der Agent kann entweder Links, Rechts oder zurück gehen. Es ist ihm aber nicht möglich ein bisschen Rechts und viel Links zu gehen. Anders verhält es sich beim stetigen Aktionsraum. Dieser zeichnet sich durch stetige Werte aus. Hier ist das Steuern eines Autos beispielhaft. Um wie viel Grad muss der Agent das Steuer drehen, damit das Fahrzeug auf der Spur bleibt? (Lapan, 2020, S. 31 f.)

Observation

Die Observation (Obs.) bzw. die Beobachtung ist ein weiterer der drei Datenübermittlungswege, welche den Agenten mit dem Environment verbindet. Mathematisch, handelt es sich bei der Obs. um einen oder mehrere Vektoren bzw. Matrizen.

Die Obs beschreibt dabei den momentanen Zustand es Envs. Die Obs kann daher als eine numerische Repräsentation anzusehen. (Sutton und Barto, 2018, S. 381)

Die Obs. hat einen immensen Einfluss auf den Erfolg des Agenten und sollte daher klug gewählt werden. Je nach Anwendungsbereich fällt die Obs. sehr unterschiedlich aus. In der Finanzwelt könnte diese z.B. die neusten Börsenkurse einer oder mehrerer Aktien beinhalten oder in der Welt der Spiele könnten diese die aktuelle erreichte Punktezahl wiedergeben. (Lapan, 2020, S. 32) Es hat sich als Faustregel herausgestellt, dass man sich bei dem Designing der Obs. auf das wesentliche konzentrieren sollte. Unnötige Informationen können den die Effizienz des Lernen mindern und den Ressourcenverbrauch zudem steigen lassen.

Reward

Der Reward bzw. die Belohnung ist der letzte Datenübertragungsweg. Er ist neben der Action und der Obs. eines der wichtigsten Elemente des RL und von besonderer Bedeutung für den Lernerfolg. Bei dem Reward handelt es sich um eine einfache skalare Zahl, welche vom Env. übermittelt wird. Sie gibt an, wie gut oder schlecht eine ausgeführte Action im Env. war. (Sutton und Barto, 2018, S. 42)

Um eine solche Einschätzung zu tätigen, ist es nötig eine Bewertungsfunktion zu implementieren, welche den Reward bestimmt.

Bei der Modellierung des Rewards kommt es vor alledem darauf an, in welchen Zeitabständen dieser an den Agenten gesendet wird (sekündlich, minütlich, nur ein

Mal). Aus Bequemlichkeitsgründen ist es jedoch gängige Praxis geworden, dass der Reward in fest definierten Zeitabständen erhoben und übermittelt wird. (Lapan, 2020, S. 29 f.)

Je nach Implementierung hat dies große Auswirkungen auf das zu erwartende Lernverhalten.

State

Der State bzw. der Zustand ist eine Widerspiegung 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 anderer Wert Q , welcher durch die Maximierung der Q-Value-Function $Q(s, a)$ bestimmt wird, gibt Aufschluss darüber, welche Action a im State s den größten Return (diskontierte gesamt Belohnung) über der gesamten Spieldauer 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

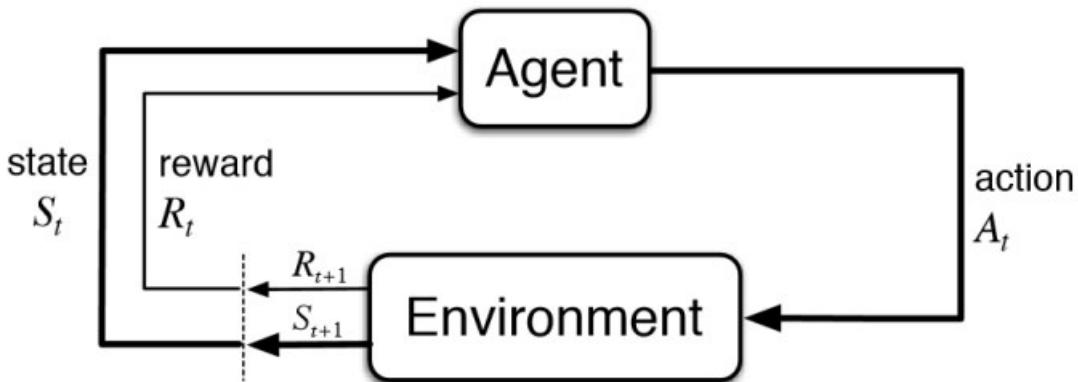


Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State S_t und falls $t \neq 0$ einen Reward R_t . Daraufhin wird vom Agenten eine Action A_t ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State S_{t+1} und Reward 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 Aufschlüsse 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 an findbar. 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 universell einsetzbar, im Gegensatz zu model-based Verfahren, welche ein Modell des Environment für das Planen benötigen (Lapan, 2020, S. 100 ff.).

Policy-Based und Value-Based Verfahren

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

Basierend auf dieser Wahrscheinlichkeitsverteilung π wird die nächste Action a bestimmt. Dabei ist es offensichtlich, dass nicht immer die optimale Aktion 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 zu dem State s führt, der über den größten Value verfügt Q , basierend auf einer Q-Value Funktion $Q_\pi(s, a)$. (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-policy 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 dem Open-AI-Team entwickelt. Im Jahr 2017 erschien das gleichnamige Paper, welches von John Schulman et al. veröffentlicht wurde. In diesem werden die Funktionsweisen genauer erläutert wird Schulman u. a. (2017).

Der PPO bietet sich besonders durch seine bereits erbrachten Erfolge Schulman u. a. (2017) und seine gute Performance an, was ihn zu einem idealen Kandidaten für einen möglichst optimalen Algorithmus, im Sinne der Forschungsfrage, auszeichnet.

2.3.1. Actor-Critic Modell

Der PPO Algorithmus ist ein policy-based RL-Verfahren, welches, im Vergleich mit anderen Verfahren, einige Verbesserungen aufweist. Er ist eine Weiterentwicklung des Actor-Critic-Verfahrens und basiert intern auf zwei NN, dem sogenanntes Actor-Network bzw. Policy-Network und das Critic-Network bzw. Value-Network. (Sutton und Barto, 2018, S. 273 f.)

Beide NN können aus mehreren Schichten bestehen, jedoch sind Actor und Critic streng voneinander 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):

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

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

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

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. Des Weiteren werden die einzelnen Summanden mit einem Discount Factor γ multipliziert, um die Zeitpräferenz des Agenten besser zu steuern. Gamma γ liegt dabei gewöhnlich zwischen einem Wert von 0.9 bis 0.99. Kleine Werte für Gamma sorgen dafür, dass der Agent langfristig eher dazu tendiert Aktionen zu wählen, welche unmittelbar zu positiven Reward führen. Entsprechend verhält es sich mit großen Werten für Gamma. (Sutton und Barto, 2018, S. 42 ff.)

Baseline Estimate

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

Advantage

Der erste Funktionsbestandteil des $L_t^{\text{CLIP}}(\theta)$?? behandelt den Advantage $\hat{A}_t(s, a)$. Dieser wird durch die Subtraktion der Discounted Sum of Rewards bzw. des Return R_t und dem Baseline Estimate $b(s_t)$ bzw. der Value-Function berechnet. Die folgende Formel ist eine zusammengefasste Version der original Formel aus Schulman u. a. (2017):

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

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

Probability Ratio

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

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

Bei genauerer 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 $\pi_{\theta_{\text{old}}}$.

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

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 $r_t(\theta)$ zwischen alter und neuer

Policy.

Intern beruht der PPO auf zwei sehr ähnlichen Objective Functions, wobei die erste $surr_1$ dieser beiden

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

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.

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

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

Zusammenfassung der PPO Training Objective Function

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

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

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

Der letzte Teil, welcher für die Bestimmung des PPO Training Loss benötigt wird, ist der Entropy Bonus. Dabei handelt es sich um die Entropie der Policy.

Es ergibt sich damit der bereits oben erwähnte PPO Training Loss ??:

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

2.3.3. PPO - Algorithmus

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

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

2.4. Deep Q-Network

Der DQN (Deep Q-Network-Algorithmus) ist ein weiterer Reinforcement Learning Algorithmus, welcher auf einer ihm zugrundeliegenden Formel basiert. Er hat bereits große Erfolge besonders in der Gaming Branche erzielen können. Daher erscheint es auch nicht weiter verwunderlich, dass sich dieser Algorithmus großer Beliebtheit erfreut. Ebenfalls wurden bereits viele Erweiterungen, wie der DDQN (Double Deep Q-Network) oder der DQN Implementierungen mit Verrauschen Netzen usw.

Aufgrund seiner großen Beliebtheit ergibt sich die Frage, ob der DQN dieser auch gerecht wird und dieser wirklich optimale Agenten zur Lösung des Spiels Snake hervorbringen kann? Eine weitere naheliegende Frage ist, ob dieser Algorithmus im Vergleich zu anderen Arten konkurrenzfähig ist? Diese beiden Fragen lassen sich mit einem Vergleich beantworten, was diesen Algorithmus zu einem weiteren guten Kandidaten macht.

Angestammtes Ziel aller Q-Learning-Algorithmen ist es, jedem State-Action-Pair (s, a) (Zustands-Aktions-Paar) einem Aktionswert (Q-Value) Q zuzuweisen (Lapan, 2020, S. 126). Dies ist beispielsweise über eine Tabelle möglich, was jedoch bei großen State-Action-Spaces (Zustands-Aktions-Räumen) schnell ineffizient wird. Ein weiterer Ansatz sind NN, welches durch seine zumeist zufällige Initialisierung der Gewichte bereits für jeden (State-Action-Pair) ein Q-Value liefert. Es sei erwähnt, dass es bei NN häufig so ist, dass nur der State als Input für das Value-NN dient.

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

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

In der Theorie konvergiert ein solches Iterationsverfahren $Q_i \rightarrow Q^*$ jedoch nur, wenn i gegen unendlich läuft $i \rightarrow \infty$, wobei Q^* die optimale Aktionswert-Funktion darstellt. Da dies jedoch nicht möglich ist, muss Q^* angenähert werden $Q(s, a; \theta) \approx Q^*$. Dies geschieht mittels eines NN.

Damit dieses nicht ausschließlich Zufallsgetriebene Q-Values ermittelt, ist eine Anpassung der Gewicht des NN nötig. Dafür muss jedoch zuerst der Loss des DQN bestimmt werden. Dies geschieht mit Hilfe mehrerer Loss-Function. Die i -te Loss-Function mit den i -ten NN-Parametern ist wie folgt definiert Mnih u. a. (2015):

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

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

Da $Q(s, a)$ rekursiv definiert werden kann, ergibt sich in vereinfachter Form (Lapan,

2020, S.126):

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

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

2.4.1. DQN - Algorithmus

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

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für das Q-NN zufallsbasiert. Setzte Epsilon $\epsilon = 1.00$ und Erzeuge leeren Replay-Buffer RB .
2. Wähle mit der Wahrscheinlichkeit ϵ eine zufällige Action a oder nutze $a = \arg \max_a Q(s, a)$
3. Führt Action a aus und ermittle den Reward r und den Folgezustand s' .
4. Speichern von State s , Action a , Reward r und Folgezustand s' . $(s, a, r, s') \rightarrow$ Replay-Buffer
5. Senkt ϵ , sodass die Wahrscheinlichkeit eine zufällige Action zu wählen minimiert wird. Gewöhnlich existiert eine untere Grenze für ϵ , sodass immer noch einige wenige Actions zufällig gewählt werden, je nach Grenze.
6. Entnehme auf zufallsbasiert Mini-Batches aus dem Replay Buffer.
7. Berechne für alle sich im Mini-Batch befindliche Übergänge den Zielwert $y = r$ wenn die Episode in diesem Übergang endet. Ansonsten soll $y = r + \gamma \max_{a' \in A} \hat{Q}(s', a')$.
8. Berechne den Verlust $\mathbb{L} = (Q(s, a) - y)^2$
9. Update $Q_\theta(s, a)$ durch SGD (Stochastischen Gradientenabstiegsverfahren, Englisch: Stochastic Gradient Descent). Daher $Q_{\theta_i}(s, a) \rightarrow Q_{\theta_{i+1}}(s, a)$
10. Kopierte alle N Schritte die Gewichte von $Q(s, a)$ nach $\hat{Q}(s, a)$ $\theta_{Q(s,a)} \rightarrow \hat{Q}(s, a)$
11. Wiederhole alle Schritte von Schritt 2 an bis sich eine Konvergenz einstellt.

Kapitel 3

Anforderungen

In Kapitel 2, wurden Grundlagen für die weiteren Vergleiche der Reinforcement Learning Agenten gelegt, welche auf zwei unterschiedlichen Algorithmen (DQN und PPO) basieren.

Um diese Vergleiche zu realisieren, soll ein System entwickelt werden, dass diese durchführen, festhalten und auswerten kann. Dieses soll aus einem Environment, mehreren Agenten beider Algorithmen, sowie aus statistischen Analysekomponenten zur Leistungsbestimmung, bestehen. Zuzüglich sollen weitere Anforderungen an die Evaluation gestellt werden, um die Vergleichbarkeit sicherstellen.

3.1. Anforderungen an das Environment

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

3.1.1. Standardisierte Schnittstelle

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

3.1.2. Funktionalitäten

Das Env soll die folgenden Funktionalitäten implementieren.

Aktionsausführung

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

Reset

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

Render

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

3.2. Anforderungen an die Agenten

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

3.2.1. Funktionalitäten

Die Agenten müssen folgende Funktionalitäten implementieren.

Aktionsbestimmung

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

Lernen

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

3.2.2. Parametrisierung

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

3.2.3. Diversität der RL Algorithmen

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

3.3. Anforderungen an die Datenerhebung

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

3.3.1. Mehrfache Datenerhebung

Die Datenermittlung muss für jeden einzelnen Agenten mehrfach durchgeführt werden, um die Validität der Messung zu gewährleisten. ((Goodfellow u. a., 2018, S. 135))

3.3.2. Datenspeicherung

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

Das System soll folgende Daten speichern:

Tabelle 3.1.: Zu erhebende Daten

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

3.4. Anforderungen an die Statistiken

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

3.5. Anforderungen an die Evaluation

Bei der Evaluation soll der optimale Agent für jedes Evaluationskriterium ermittelt werden. Das Kriterien der Performance wurde von verwendeten Literatur Wei u. a. (2018) und Chunxue u. a. (2019) übernommen, anderen ergeben sich aus den zu speichernden Daten. Die einzelnen Kriterien lauten:

Tabelle 3.2.: Evaluationskriterien

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

Siegrate	Welcher Agent schafft die Spiele mit einem Sieg zu beenden? Die Siegrate gibt Aufschluss darüber, ob die angelernte Aufgabe vollständig gelöst werden kann (siehe 2.1). Im Gegensatz zur Performance gibt die Siegrate die Fähigkeit zur Konvergenz an. Sie ist weiterhin ein wichtiger Faktor für das Beenden des Trainings.
----------	---

Kapitel 4

Verwandte Arbeiten

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

- Optimierungsstrategien
- Reward Funktion
- Evaluationskriterien

4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning

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

Thematisch wurden in diesem Paper drei Optimierungsstrategien vorgestellt, welche auf einen Baseline DQN (Referenz DQN) angewendet worden sind. Bei diesen Strategien handelt es sich um den Training Gap, die Timeout Strategy und den Dual Experience Replay.

Der Dual Experience Replay (Splited Memory) besteht aus zwei Sub-Memories, in welchen Erfahrungen belohnungsbasiert eingesortiert und gespeichert werden. Mem1 besteht dabei nur aus Erfahrungen, welche einen Reward aufweisen, der größer als

ein vordefinierter Grenzwert ist. Die restlichen Erfahrungen werden in Mem2 eingepflegt. Zu Beginn des Lernens werden 80% Erfahrungen aus Mem1 ausgewählt und 20% Erfahrungen aus Mem2 entnommen, um den Lernerfolg zu beschleunigen. Im weiteren Lernverlauf wird dieses Verhältnis normalisiert (Mem1: 50% and Mem2: 50%).

Der Training Gap beschreibt die Erfahrungen, welche der Agent, zum Zweck des performanteren Lernens, nicht verarbeiten soll. Zu diesen zählen die Erfahrungen direkt nach dem Konsum eines Apfels, sodass der Agent die Neuplatzierung dieses erlernen würde. Da der Agent auf diesen Prozess jedoch keinen Einfluss hat, könnte die Verarbeitung dieser Daten den Lernerfolg mindern, weshalb die Training Gap Strategie etwaige Erfahrungen nicht speichert und das Lernen während dieser Periode verhindert.

Die Timeout Strategy sorgt für eine Bestrafung, wenn der Agent über eine vordefinierte Anzahl an Schritten P keinen Apfel mehr gefressen hat. Dabei werden die Rewards der letzten P Erfahrungen mit einem Malus verrechnet, was den Agenten dazu anhält die schnellste Route zum Apfel zu finden. Die Höhe des Malus ist proportional zur Länge der Snake (geringe Länge → großer Malus; große Länge → geringer Malus).

Die optimierte Reward Funktion, welche das Paper verwendet, besteht damit aus dem Distanz Reward, welcher sich aus der Addition des vorherigen Rewards mit Δr ergibt.

$$r_{res} = r_t + \Delta r \quad (4.1)$$

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

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

Wobei t dem vorherigen, $t + 1$ dem aktuellen Zeitpunkt darstellt. L_t ist die Länge der Snake zum vorherigen Zeitpunkt und D_t und D_{t+1} stellen die Distanzen zwischen Snake und Apfel zum vorherigen und aktuellen Zeitpunkt dar.

Sollte die Timeout Strategy auslösen, so werden die letzten P Erfahrungen entsprechend angepasst und in Mem2 verschoben.

Als maßgebliches Kriterium zur Evaluation der Leistung des DQN wurde die Performance, gemessen im Score und die steps survived, also die überlebten Schritte herangezogen.

4.1.1. Diskussion

Sowohl die Training Gap also auch Dual Experience Replay und Timeout Strategy stellen vielversprechende Optimierungen dar, welche, auf experimentellen Resultaten basierend, gute Ergebnisse erzielen konnten. Jedoch existieren auch Diskussionspunkte an der Ausarbeitung. Das Env wird im Paper nur kurz vorgestellt, jedoch lässt sich aus dem Abschnitt Game Environment schließen, dass alle Anforderungen des Env 3.1 erfüllt sind.

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

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

Weiterhin wurde sich bei den Evaluationskriterien einzig auf die Performance, gemessen am Score, und Spielzeit, gemessen in steps, konzentriert. Weitere Kriterien wie, beispielsweise die Robustheit oder Siegrate 3.2, werden nicht betrachtet.

Dies soll in dieser Ausarbeitung jedoch geschehen, da diese Faktoren ebenfalls wichtig für eine voll umfassende Bewertung der Agenten sind, besonders in Realwelt-Applikationen (siehe 1.1).

Des Weiteren ist erwähnenswert, dass die Verfasser auf das Evaluationskriterium der steps survived setzten, welches im kompletten Widerspruch zur Effizienz steht. Zhepei Wei et al. sehen ein langes Überleben der Snake als positiv an, wohingegen dies in dieser Ausarbeitung kritisch betrachtet wird, da es kein zielgerichtetes Verhalten bestärkt. Ziel der Agenten in dieser Ausarbeitung ist es, das Spiel Snake möglichst effizient zu lösen und es nicht lange zu überleben.

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

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

Auch in diesem Paper verwenden die Autoren Optimierungen, um den Lernerfolg zu steigern.

Einer dieser Optimierungen wurde auf Basis des Geruchssinnes konzipiert. Der Odor Effect erzeugt um den Apfel drei aneinanderliegende Geruchszonen, in welchen ein größerer Reward zurückgegeben wird als außerhalb der Geruchszonen. Dabei unterscheiden sich diese in der Höhe des zurückgegebenen Rewards, sodass die dritte Zone den geringsten und die erste Zone den größten Reward von allen zurückgibt ($r_1 > r_2 > r_3$, wobei r_x der Reward des x-ten Kreises darstellt). Diese Zonen stellen den zunehmenden Duft von Nahrung dar, wobei dieser immer stärker wird, um so näher man sich der Quelle nährt.

Eine weitere Optimierungsstrategie basiert auf dem Loop Storm Effect, welcher das Verhalten beschreibt, um den Apfel zu laufen. Die Verfasser haben festgestellt, dass dieser Effekt zu einem schlechten Lernerfolg führt. Darum haben Wu et al. einen dynamischen Positionsspeicher konzipiert, welcher Loops erkennt und diese, durch das Zurückgeben einer Zufallsaktion, welche nicht auf dem Loop liegt, unterbricht. Experimentelle Ergebnisse des Papers haben gezeigt, dass der Loop Storm Effect kaum mehr in Erscheinung trat.

Auf Basis einer Standard Reward Funktion, welche für das Essen eines Apfels +100 und für das Sterben -100 zurückgibt, wurden ein Versuch durchgeführt. Dem Agenten war es nicht möglich gegen die optimale Lösung zu konvergieren, aufgrund von Loop Storms und einer unzureichende Reward Funktion. Hingegen war es dem Agenten mit Odor Effect und dynamischen Positionsspeicher möglich, eine Konvergenz nach 8 Millionen Epochs (hier Trainingsdurchläufe eines DQN) zu erreichen.

Die Reward Funktion entspricht daher dem Odor Effect. Auch in diesem Paper bleibt die Performance maßgeblicher Evaluationsfaktor, wobei der Fokus auf den erreichten Reward gelegt wurde, welcher stark mit der Performance korreliert.

4.2.1. Diskussion

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

Aus Graphiken geht hervor, dass die Verfasser ein Env mit einer Visualisierung besitzen. Weiterhin ist davon auszugehen, dass auch alle weiteren Anforderungen bezüglich des Env (siehe 3.5) erfüllt worden sind, da ansonsten kein Training eines Agenten möglich wäre. Dennoch wurde das Env nur auf grundlegende Weise behandelt.

Das Paper legt seinen Schwerpunkt deutlich mehr auf die Grundlagen des RL, wie z.B. auf den Markov Decision Process und die RL Kernbegriffe (siehe 2.2.1).

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

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

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

4.3. Zusammenfassung

Sowohl „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ 4.1 als auch „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ 4.2 bieten einige Optimierungsstrategien, welche im weiteren Verlauf dieser Ausarbeitung als Vorbild dienen sollen.

Besonders zu betonen ist, dass die vorgestellten Arbeiten alle verschiedene Schwerpunkte gesetzt haben und daher nicht immer die Anforderungen dieser Ausarbeitung erfüllt haben.

So wurden zwar die Anforderungen zum Environment (siehe 3.1) und zu den Agenten (siehe 3.2), mit Ausnahme der Parametrisierung (siehe 3.2.2) und der Diversität der Algorithmen (siehe 3.2.3), alle erfüllt. Dennoch wurden auch Anforderungen bei der statistischen Datenerhebung und Evaluation nicht erfüllt. Zu diesen Anforderungen zählen z.B. die mehrfache Datenerhebung (siehe 3.3.1) und die Evaluation-Anforderungen (siehe 3.5). Letztere weichen, verständlicher Weise, von den Anforderungen ab, da sich die Verfasser der Arbeiten auf andere Aspekte konzentriert haben und daher nicht die selben Evaluationskriterien gewählt haben.

Kapitel 5

Konzept

In diesem Kapitel soll das Konzept dieser Ausarbeitung vorgestellt werden. Dieses besteht aus vier Teilen. Zuerst soll das Vorgehen erklärt werden, gefolgt von der Darstellung des Environments und der Agenten. Danach soll im weiteren auf die Datenerhebung eingegangen werden.

Ziel dieses Abschnittes ist es, das Vorgehen und alle weiteren dazu benötigten Elemente unabhängig von der Implementierung darzustellen, sodass die Ergebnisse reproduzierbar sind.

5.1. Vorgehen

Das Vorgehen lässt sich am besten mithilfe eines Flussdiagramms darstellen, in welchem die einzelnen Schritte des Vergleichs visuell dargestellt werden.

Zu Beginn sei erwähnt, dass die Annahme getroffen wird, dass alle für die Vergleiche benötigten Komponenten, wie z.B. Environment, Agenten und statistische Analysekomponenten, entsprechende der Anforderungen implementiert sind (siehe Abschnitt 3).

Als erstes werden die Agenten erstellt (siehe Abbildung 5.1). Für diese diesen Zweck werden mehrere Klassen mit verschiedenen Hyperparametern generiert bzw. ausgewählt. Mit dieser Baseline-Agenten-Menge werden nun die weiteren Vergleiche durchgeführt.

Zu Beginn werden mithilfe der Evaluationskriterien die zwei optimalen Baseline Agenten bestimmt. Auf diese sollen die Optimierungen angewendet werden. Aufgrund der Tatsache, dass das Anwenden der Optimierungen viel Zeit und Ressourcen bindet, sollen diese nur auf die vielversprechendsten Agenten angewendet werden.

Der verwendete Algorithmus besitzt dabei keinen Einfluss auf die Auswahl der Agenten, sodass auch zwei DQN oder PPO Agenten optimiert werden können. Genauere Details zur Durchführung der Baseline Vergleiche finden sich im Abschnitt 5.5.1.

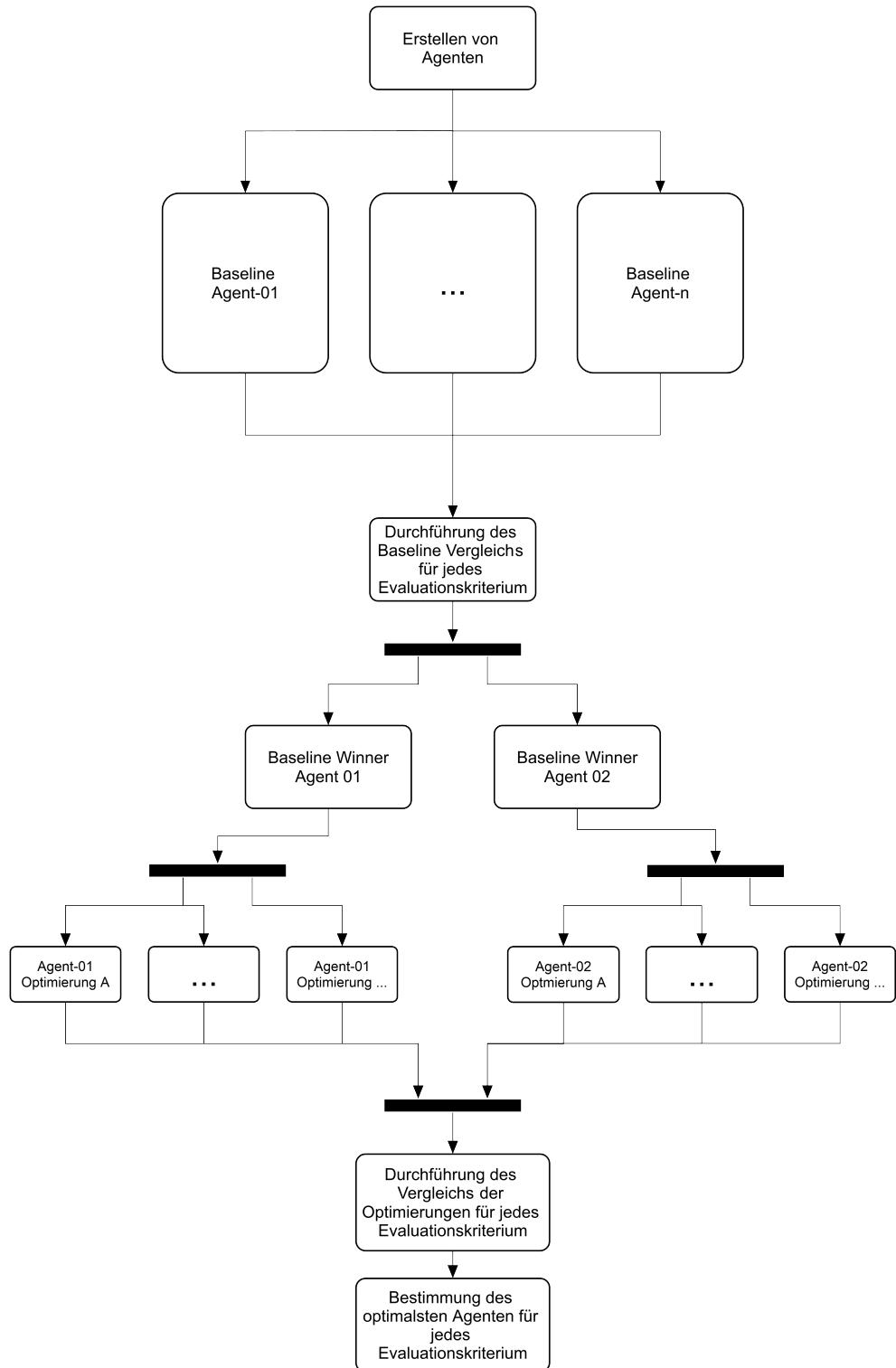


Abbildung 5.1.: Darstellung des Vorgehens.

Basierend auf den beiden Sieger Agenten (Baseline Winner Agent-01 und Baseline Winner Agent-02) der Baseline Vergleiche (siehe Abbildung 5.1), werden nun die Optimierungen angewendet. Mit diesen optimierten Agenten (Agent-01 Optimierung A bis Agent-02 Optimierung B) werden nun die Vergleiche bezüglich jedes einzelnen Evaluationskriteriums (siehe Tabelle 3.2) erneut wiederholt.

Im letzten Schritt soll in der gesamt Evaluation der optimale Agent für jedes Evaluationskriterium ermittelt werden. Dabei können dies auch Baseline Agenten sein. Es ist nicht zwingend zugesichert das optimierte Agenten die optimalsten dies.

5.2. Environment

Das Env besteht im wesentlichen aus einer Hauptkomponenten, der Spiellogik, welche von einer Schnittstellen-Komponente umschlossen wird. Diese soll mit einer standardisierten Schnittstelle (siehe 3.1.1) implementiert werden.

Die Spiellogik kapselt die Game-, Player-, Reward-, Observation- und GUI-Komponenten, welche im Folgenden näher erklärt werden.

5.2.1. Spiellogik

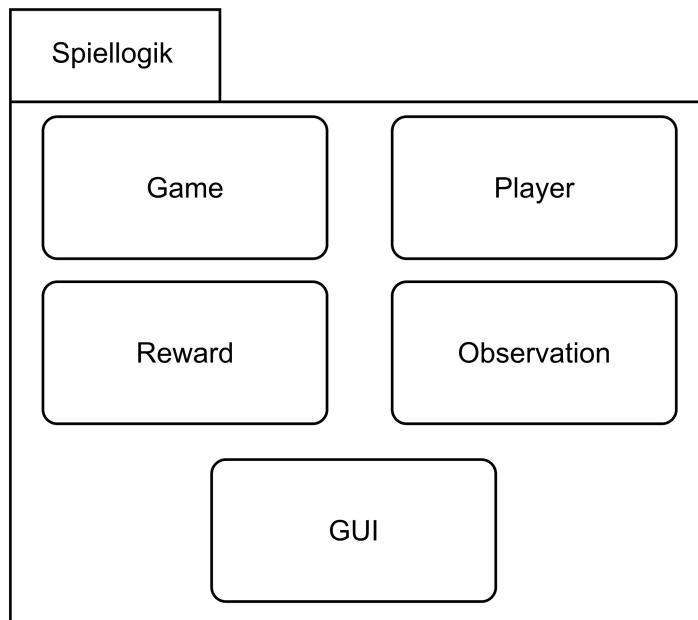


Abbildung 5.2.: Darstellung der Spiellogik mit ihren Unterkomponenten.

Die Spiellogik besteht aus den fünf Unterkomponenten, welche im obigen Abschnitt Environment bereits benannt wurden.

Die Game-Komponente stellt die Hauptkomponente dar, da sie die eigentliche Aktionsdurchführung implementiert. Sie beinhaltet jeweils Instanzen der Reward-, Observation-, GUI- und Player-Komponenten. Letztere ist eine Datenhaltungskomponente, welche die Daten der Snake, wie z.B. Position oder Ausrichtung (direction) beinhaltet.

Die Reward-Komponente bestimmt den auszugebenden Reward nach jeder Aktionsabfertigung. Dieser berechnet sich wie in Abschnitt 5.2.1 angegeben. Zuzüglich

wird im Rahmen der Optimierungen A, (siehe Abschnitt 5.4.1), eine weitere Reward Funktion implementiert.

In der Game-Komponente werden wichtige Spielbezogene Daten verwaltet. Zu diesen gehören das Spielfeld (ground), sowie die Form des Spielfeldes (shape) und die Position des Apfels auf dem Spielfeld. Sie beinhaltet viele Methoden, wie z.B. die Aktionsausführung, Observation- und Reward-Erstellung und weitere Routinen.

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

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

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

Spielablauf

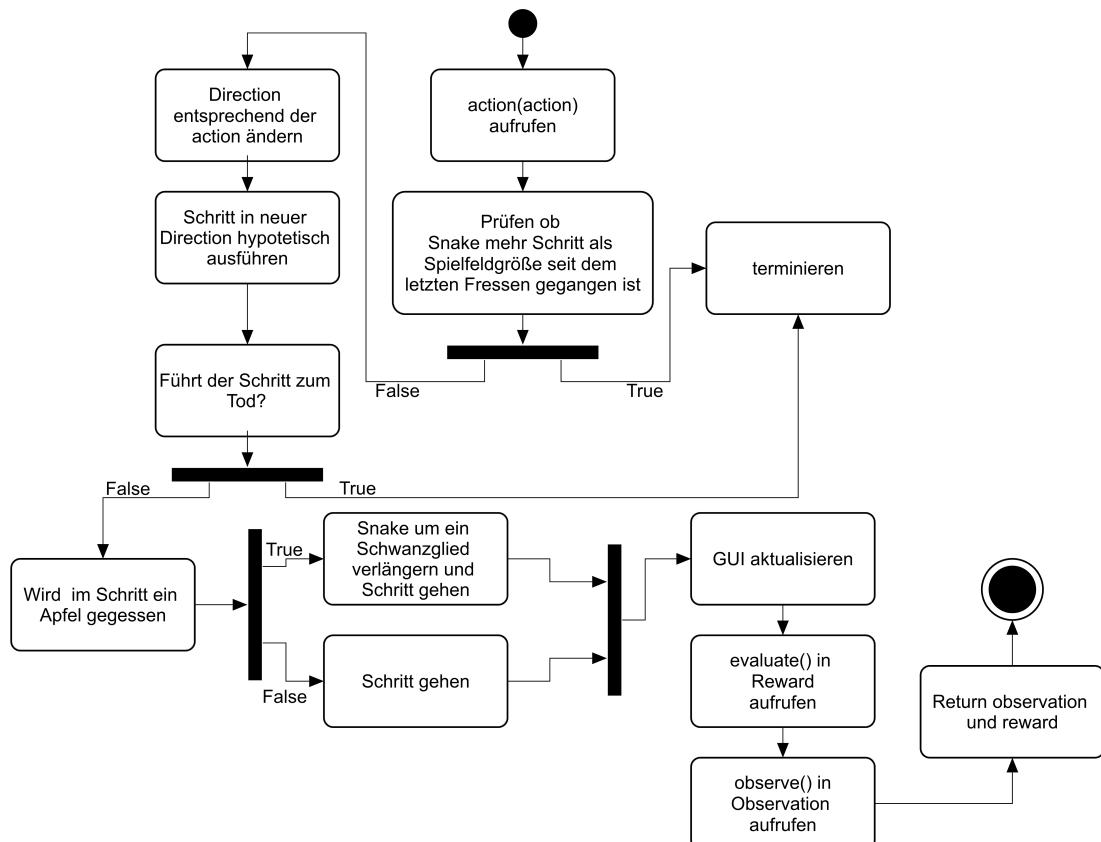


Abbildung 5.3.: Darstellung eines Schrittes in der Spieleserie.

Die eigentliche Aktionsabarbeitung wird durch das Aufrufen der step Funktionalität in der Schnittstellen-Komponente (siehe Abschnitt 5.2.2) bewirkt. Diese ruft die Routinen zur Erstellung des Rewards (evaluate) und der Observation (observe) auf, welche in der Game-Komponente implementiert sind (siehe Abbildung 5.2). Um die Abarbeitung einer Aktion durchzuführen, wird die action Funktionalität aufgerufen. Der Ablauf einer Aktionsabarbeitung ist in der Abbildung 5.3 dargestellt. Zu Beginn wird überprüft, ob die Snake seit dem letzten Fressen mehr Schritte als die eigentliche Spielfeldgröße gegangen ist. Im Rahmen der Bestimmung der Robustheit, wird sich diese im Testverlauf jedoch ändern (siehe Abschnitt 5.5.1).

Sollte die Snake mehr Schritte gelaufen sein als die Größe des Spielfelds, so wird das Spiel terminiert, da die Snake eventuell in einer Schleife steckt. Andernfalls wird die Aktion verarbeitet, indem sie die Ausrichtung (direction) der Snake manipuliert wird. Das Spiel Snake besitzt drei Aktionen: turn left, turn right oder do nothing.

Tabelle 5.1.: Kodierung der Aktionen

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

Entsprechend der Tabelle 5.1 wird deutlich, dass die Ausrichtung (direction) entweder nur Norden, Osten, Süden oder Westen sein kann. Als nächstes wird ein Schritt, mit aktualisierten Ausrichtung, hypothetisch durchgeführt. Dabei wird festgestellt, ob die Ausführung des Schrittes zum Tod der Snake führt. Sollte dies der Fall sein, so wird der Spielablauf terminiert. Schritte in sich selbst und das Verlassen des Spielfeldes führen zum Tod (siehe Abschnitt 2.1).

Andernfalls wird der Schritt durchgeführt. Dabei wird zwischen zwei Fällen unterschieden.

Sollte die Snake einen Apfel gefressen haben, also Kopf und Apfel die selbe Position einnehmen, so wächst die Snake um ein Schwanzglied. Der alte Apfel wird entfernt und ein neuer erscheint, zufallsbasiert, auf einem freien Feld des Spielfelds.

Sollte die Snake keinen Apfel gefressen haben, so geht sie einfach den Schritt, es bewegen sich daher alle Schwanzglieder auf die Vorgängerposition, mit Ausnahme des Kopfes, welcher die neue Position einnimmt.

Nach der Ausführung einer dieser beiden Fälle, wird die Spieloberfläche und GUI aktualisiert. Nach diesem Schritt ist die Aktionsabarbeitung abgeschlossen.

Damit der Agent die momentane Observation und Reward erhält, werden diese in

der Reward-Komponente bzw. Observation-Komponente bestimmt und zurückgegeben.

Reward

Der Reward wird in der Reward-Komponente, basierend auf dem letzten Zug, gebildet. Dies geschieht nach dem folgenden Vorbild. Der Standard Reward ist abhängig von drei Faktoren. Dem Fressen eines Apfels, dem Sieg oder Verlust. Sollte keiner dieser genannten Faktoren eintreten, wird ein Reward von -0.01 zurückgegeben. Dies hält den Agenten dazu an den kürzesten Pfad zum Apfel zu finden, da jeder Schritt geringfügig bestraft wird. War es der Snake möglich einen Apfel zu fressen, so wird ein Reward von +2.5 zurückgegeben, da ein Zwischenziel erreicht wurde. Sollte die Snake gestorben sein, 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 ihr ausgefüllt ist, so wird ein Reward von +10 zurückgegeben, um ein solches Verhalten in seiner Häufigkeit zu maximieren. Dies Snake hat zu diesem Zeitpunkt dann das Spiel gewonnen.

Die zweite Reward Funktion wird in Abschnitt 5.4.1 erklärt.

Observation

Die Observation, wird in der Observation-Komponente erzeugt und besteht aus der around_view (AV) un der scalar_obs (SO). Mithilfe verschiedener Unterfunktionen wird diese generiert.

Die AV lässt sich als ein Ausschnitt des Spielfeldes (ground) beschreiben, welcher einen festen Bereich um den Kopf der Snake abdeckt. Strukturen wie Wände und Teile des eigenen Schwanzes, werden deutlich. Mathematisch ist die AV eine one-hot-encoded Matrix der Form (6x13x13).

Das One-Hot-Encoding ist ein binäres encoding System. Sollte ein Merkmal vorhanden sein, so wird dieses mit eins codiert anderenfalls mit null. (Lapan, 2020, S. 359 f.)

Dies ist auch der Grund, warum die AV Matrix sechs Channel (zweidimensionale Schichten) besitzt. Diese geben Aufschluss über folgende Informationen:

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

Channel der Matrix bzw. Erste Dimension (Ax13x13)	Erklärung

A = 0	Die erste Feature Map signalisiert den Raum außerhalb des Spielfelds.
A = 1	Diese Feature Map stellt alle Schwanzglieder mit Ausnahme des Kopfes und es letzten Schwanzgliedes dar.
A = 2	In dieser Feature Map wird der Kopf der Snake dargestellt.
A = 3	Damit gegen Ende des Spiels der Agent noch freie Felder erkennen kann, wird in dieser Feature Map jedes freie und sich im Spielfeld befindliche Feld mit eins codiert.
A = 4	Die vorletzte Feature Map codiert das Schwanzende der Snake.
A = 5	In der letzte Feature Map wird der Apfel abgebildet.

Vorteilhaft an der AV ist, dass, im Gegensatz zu den Obs in den verwandten Arbeiten Wei u. a. (2018) und Chunxue u. a. (2019), nicht das gesamte Feld übertragen wird, sondern nur der wichtigste Ausschnitt, was die Menge an zu verarbeiten Daten reduziert.

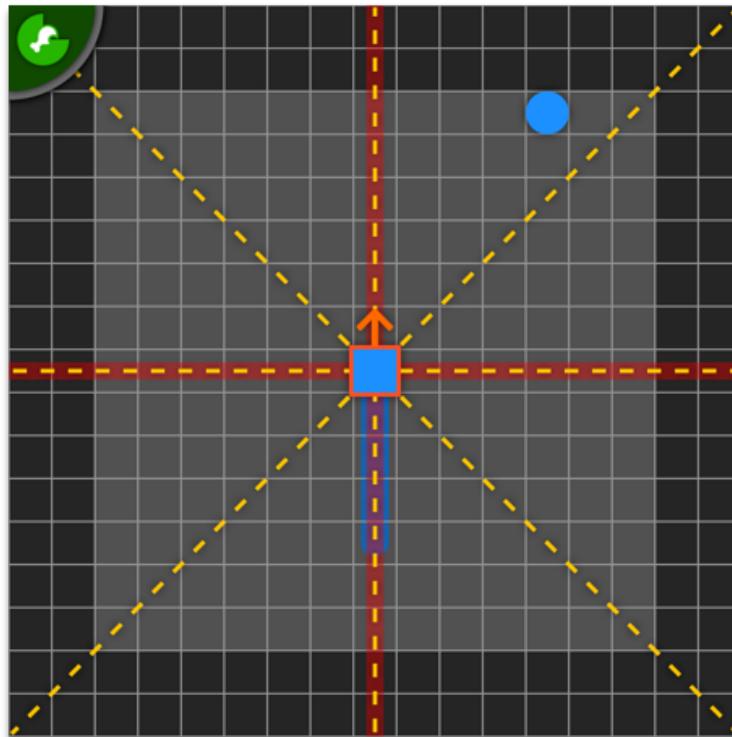


Abbildung 5.4.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar. Schwarzen wird nicht von der AV abgedeckt, graue liegen innerhalb der AV. Die gelben gestrichelten Linien stellen die Raytracing Distanzbestimmung dar. Der blaue Kreis stellt den Apfel dar und der grüne viertel Kreis oben links symbolisiert den Hunger.

Ein Nachteil dieser Obs ist jedoch die Unvollständigkeit. Sollte beispielsweise der blaue Punkt (der Apfel) in Abbildung 5.4 außerhalb des grauen Kasten und daher außerhalb der AV liegen, so bleibt der Agent im Unklaren über den Aufenthaltsort des Apfels.

Aus diesem Grund wurde die AV durch die scalar_obs (SO) ergänzt. Die SO beinhaltet skalare Informationen und ist eine Konkatenation aus Raytracing Distanzbestimmung (siehe Abschnitt B.2), Hunger- und Blickrichtungsanzeige (direction). In Quelle Glassner (1989) wird das grundlegende Verfahren, auf welchem die Raytracing Distanzbestimmung basiert, vorgestellt. Zuzüglich werden der SO noch zwei Kompassen für relative Positionsinformation zwischen Kopf und Apfel bzw. letztem Schwanzglied hinzugefügt.

Letztere sind eindimensionale Vektoren, welche mithilfe des One-Hot-Encoding anzeigen, ob sich das gesuchte Objekt relativ zum Kopf oberhalb, unterhalb oder in der selben Zeile befindet (Matrixsicht). Analog verhält es sich mit der vertikalen Sicht. Die Blickfeldanzeige ist ebenfalls one-hot-encoded und stellt mit ihrem Vektor die vier Ausrichtungen Norden, Osten, Süden und Westen dar.

Der Hunger ist die Differenz zwischen der Anzahl der gegangenen Schritte seit dem letzten Fressen (inter_apple_steps) und der maximalen Schrittanzahl (max_steps) (siehe Abschnitt 5.2.1). Zur besseren Verarbeitung für die NNs, wird dieser zudem mit -1 quadriert.

$$\min(2, \frac{1}{\text{inter_apple_steps} - \text{max_steps}}) \quad (5.1)$$

Um, mit der Unendlichkeit auftretende, Probleme zu umgehen wird zwei zurückgegeben, wenn die Differenz null wird.

In ähnlicher Weise wird mit den Raytracing Distanzbestimmungen verfahren. Bei diesem handelt es sich um acht Distanzmesserlinien, die in 45° Abständen ausgesandt werden, siehe Abbildung 5.4. Befindet sich das gesuchte Objekt in dieser Linie, so wird die Distanz ermittelt und analog zum Hunger angepasst. Gesuchte Objekte sind Wände, der eigene Schwanz und der Apfel. Daher wird die Raytracing Distanzbestimmung in einem Vektor der Größe 24 ($3 * 8 = 24$) gespeichert.

GUI

Die graphische Oberfläche oder auch GUI genannt kann optional ein- oder ausgeschaltet werden. Beim Lernen der Agenten bietet es sich beispielsweise an diese auszuschalten, da diese die Lerngeschwindigkeit senkt. Beim Start der GUI wird ein Fenster geöffnet, welches den momentanen Stand der Spielgeschehens anzeigt. Nach jeder Aktionsdurchführung wird die GUI aktualisiert.

5.2.2. Schnittstelle

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

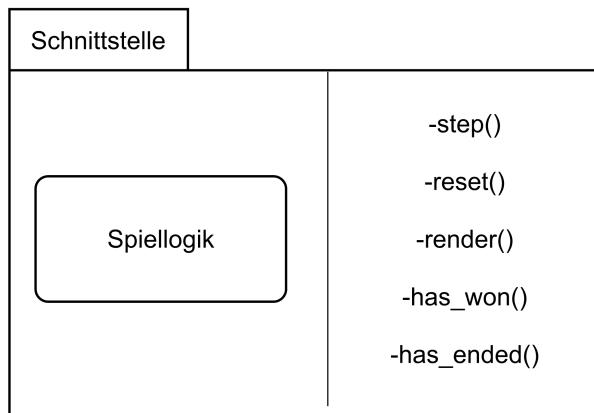


Abbildung 5.5.: Darstellung der Schnittstelle.

Die step Funktionalität (siehe Abbildung 5.5) ist für das Aufrufen der Aktionsausführung zuständig. Entsprechend der Anforderung der Standardisierte Schnittstelle (siehe Abschnitt 3.1.1) gibt sie nur Reward und Observation zurück.

Reset setzt den bereits vorhandenen Spielfortschritt zurück (siehe 3.1.2).

Render ist für die Visualisierung der Spieloberfläche verantwortlich (siehe 3.1.2).

5.3. Agenten

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

5.3.1. Netzstruktur

Zu Beginn soll die Netzstruktur erklärt werden, wobei dies unabhängig von den Algorithmen geschehen kann, da sowohl DQN als auch PPO Agenten das annähernd gleiche Netz nutzen. Im Rahmen dieser Ausarbeitung soll sich nur auf eine Netzstruktur konzentriert werden, um die Vergleichbarkeit der einzelnen Algorithmen zu erhöhen. Dennoch müssen, aufgrund der Algorithmen, kleinere Anpassungen an den Netzen vorgenommen werden.

In den Papers von Wei u. a. (2018) und Chunxue u. a. (2019) wurden einzig große CNNs (Convolutional Neural Network) genutzt, die viele unnötige Informationen verarbeiten. Zusätzlich können noch Probleme mit variablen Spielfeldgrößen auftreten, sodass in dieser Ausarbeitung auf ein zweiteilige Netzstruktur gesetzt wird.

Zuerst wird die AV durch das AV-Network geleitet. Diese besteht aus dem AV-Network und Actor-, Critic- oder Q-Tail. Dabei wird die AV durch zwei Convolutional Layer mit einer ReLU Aktivierungsfunktion propagiert. Dabei erhöht sich die Channel-Anzahl auf acht, wobei eine weitere Erhöhung, aufgrund der bereits sehr stark optimierten AV, nicht nötig ist.

Danach werden allen Feature Maps eine Null-Zeile und Null-Spalte hinzugefügt (Padding), damit beim Max-Pooling, auch die letzten Zeile und Spalte der originalen AV verarbeitet werden. Nach dem max-pooling besitzen die Feature Maps eine Größe 7x7 (Tensor: 8x7x7).

Dann folgt die Einebnung (Flatten) zu einem eindimensionalen Tensor, welcher daraufhin durch zwei weitere Fully Connected Layer (FC) propagiert wird. Der resultierende Tensor besitzt die Größe 1x128 und ist ein Zwischenergebnis, da dieser nun mit der SO verbunden wird (Join). Der Vorgang ist in der Abbildung 5.6 dargestellt. Da das NN in beide Algorithmen verwendet wird, müssen Netzwerk-Tails für den Actor, Critic und für das Q-Network (Q-Net) definiert werden. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nachdem der Joined Tensor (1x169), welcher aus dem AV-Networks Output und der SO besteht, durch zwei weitere FC Layer propagiert wurde, benötigt der Actor des PPO Algorithmus eine Wahrscheinlichkeitsverteilung über alle Aktionen. Daher auch die Ausgabe eines Tensors der Größe drei. Um diese Wahrscheinlichkeitsverteilung zu erhalten, wird die SoftMax Funktion angewendet, siehe Abbildung 5.7 links.

Der Critic des PPO Algorithmus verwendet hingegen den Critic-Tail, siehe Abbildung 5.7 Mitte. Dieser leitet den Joined Tensor durch zwei weiteres FC Layers, wobei das erste eine Aktivierungsfunktion besitzt. Da der Critic für jeden State die Discounted Sum of Rewards bestimmt (siehe Abschnitt 2.3.2), gibt dieser einen Tensor mit einem einzigen Wert zurück (Skalar).

Der Q-Net-Tail ist in seinem Aufbau sehr ähnlich zum Critic-Tail. Da dieser jedoch die Q-Values für jede Aktion im Zustand bestimmten soll, muss ein Tensor der Größe drei zurückgegeben werden. Von der Struktur der Netze sind Critic- und Q-Net-Tail,

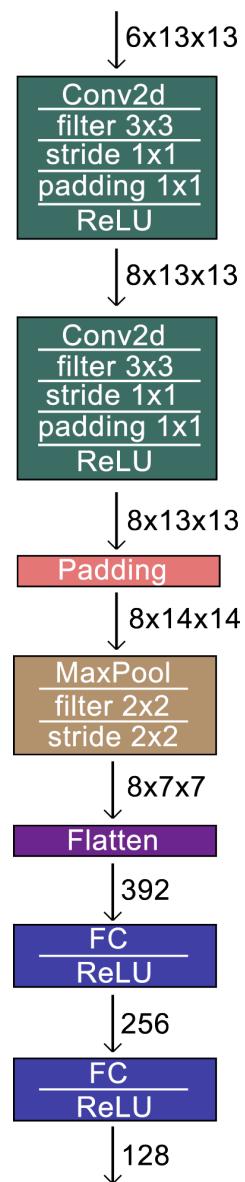


Abbildung 5.6.:
AV-Network

mit Ausnahme der Ausgabeschicht, gleich (siehe Abbildung 5.7 rechts).

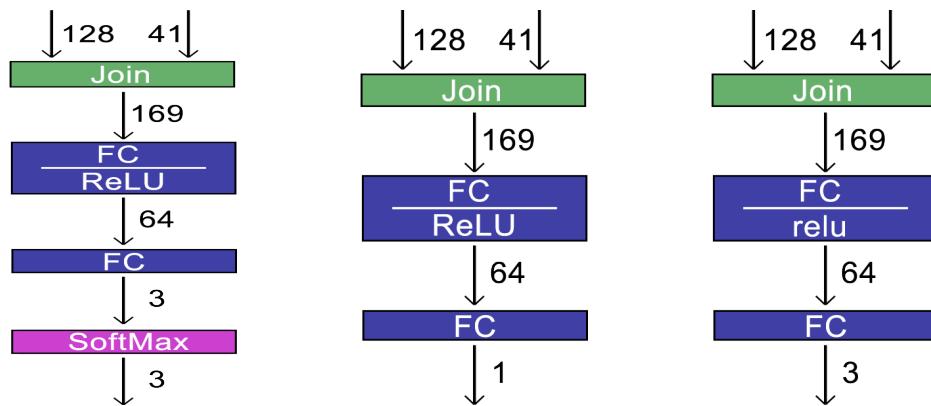


Abbildung 5.7.: Darstellung des Critic-Tail (links) und des Q-Net-Tail (rechts).

5.3.2. DQN

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

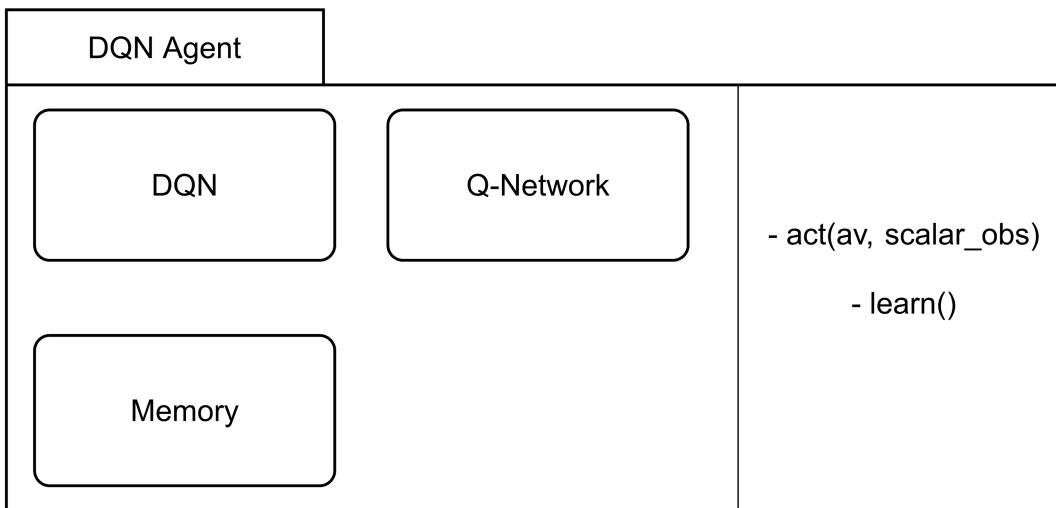


Abbildung 5.8.: Darstellung des DQN-Agent mit seinen Komponenten.

Diese Hauptfunktionalitäten sind in der DQN-Komponente eingebettet, welche die zentrale Instanz des DQN darstellt. In ihr werden wichtige Konstanten für den DQN Algorithmus, wie z.B. Gamma, Epsilon (eps), Epsilon-Dekrementierung (eps_dec), der minimal Wert für Epsilon (eps_min), die Batch-Size (batch_size), die maximale Größe des Memory (max_mem_size) und die Lernrate (lr), gespeichert.

Die Memory-Komponente speichert Erfahrungen des DQN Agenten in einer Ring-Buffer Struktur. Sollte dieser Buffer voll sein, so werden die ältesten Erfahrungen mit den neuen überschrieben. Dies Verfahren wird in Quelle (Mnih u. a., 2013, S. 5)

dargestellt. Abzuspeichernde Werte, für jeden Schritt, sind die around_view (AV), die scalar_obs (SO) die Aktion (action), der Reward (reward), die Information, ob man sich in einem terminalen Zustand befindet (terminal) und die around_view (AV_) und scalar_obs (SO_) des Nachfolgezustandes (siehe Abschnitt 2.4.1).

Die Q-Network-Komponente verwaltet das NN (Q-Network). Dieses wird zur Q-Value Bestimmung und damit zur Aktionsbestimmung genutzt. Es wird durch Lernprozesse aktualisiert.

Aktionsauswahlprozess

Eine genaue Darstellung der Aktionsbestimmung befindet sich in Abbildung 5.9. Um eine Aktion zu bestimmen, muss zuerst ein Zufallswert zwischen null und eins, was den Wahrscheinlichkeiten von 0% bis 100% entspricht generiert werden.

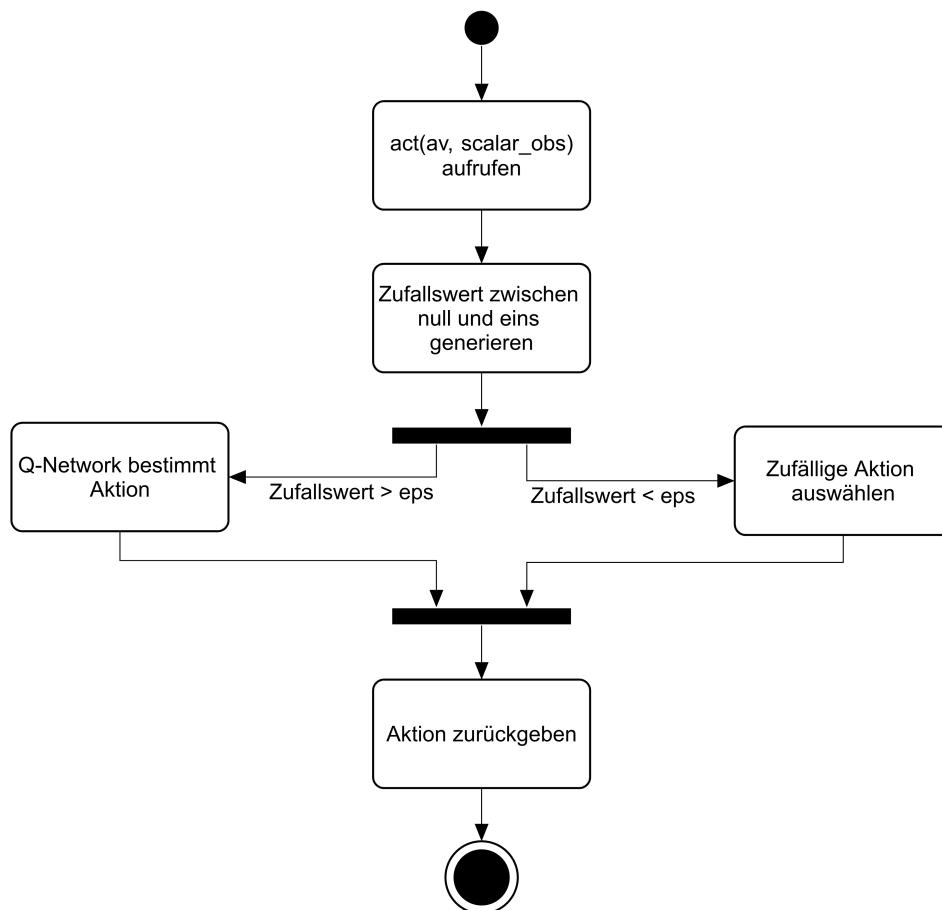


Abbildung 5.9.: Darstellung der Aktionsbestimmung des DQN-Agent.

Ist der Zufallswert größer als den momentane Epsilon-Wert, so wird die Aktion durch das Q-Network bestimmt. Andernfalls wird eine zufällige Aktion ausgewählt. Die Bestimmung der Aktion durch das Q-Network geschieht dabei wie folgt:

Die around_view (AV) und die scalar_obs (SO) werden durch das Q-Network, entsprechende der Ausführungen in Abschnitt 5.3.1, geleitet. Dieses gibt einen Tensor

der Größe drei wieder, welcher die Q-Values der Aktionen turn left (0), turn right (1) und do nothing (2) beinhaltet. Es wird daraufhin die Aktion gewählt, welche dem Index des größten Q-Values entspricht.

Sei $(0.32, -0.11, 0.45)$ ein Tensor, welcher vom Q-Network zurückgegeben wurde, dann würde no nothing (2) gewählt werden, da 0.45 der größte Q-Value ist und dieser an Stelle 2 steht.

Die oben beschriebene Prozedur stellt den Aktionsauswahlprozess während des Trainings dar. Während Testläufen, wird die Aktion immer durch das Q-Network bestimmt (siehe Abschnitt 6.3.3).

Lernprozess

Der Lernprozess stellt sich wie folgt dar:

Zuerst wird überprüft, ob im Memory genügend Experiences (Exp) gespeichert sind, um einen Mini-Batch mit der zuvor definierten Batch-Size, zu entnehmen. Sollte dies nicht der Fall sein, wird die Methode terminiert. Andernfalls wird ein Mini-Batch aus zufälligen Exp gebildet. Ab diesem Punkt wird das Verfahren, der Übersichtlichkeit wegen, nur noch für eine einzige Erfahrung beschrieben.

Danach wird der Q-Value der gespeicherten Aktion $Q(s_i, a_i; \theta_i)$ bestimmt (siehe Gleichung 2.10), wobei s_i den State (siehe Abschnitt 2.2.1), a_i die gespeicherte Aktion im Zustand s_i und θ die Netzwerkparameter des Q-Networks, darstellten. Dieser wird als Q-Eval definiert.

Danach werden die Q-Values des Nachfolgezustandes (AV__ und SO__) $Q(s', a')$ bestimmt. Sollte der Nachfolgezustand ein terminaler Zustand sein, so werden die Q-Values auf null gesetzt, da diese die zu erwartende Discounted Sum of Rewards angeben (siehe Abschnitt 2.3.2). In einem terminalen Zustand ist diese Summe gleich null, da keine Zustände mehr besucht werden (siehe Abschnitt 2.4).

Daraufhin wird der maximale Q-Value bestimmt, mit Gamma multipliziert und mit dem erhaltenen Reward addiert $r(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$ (siehe Gleichung 2.10). Dieser Wert wird als Q-Target definiert und soll Q-Eval entsprechen.

Am Ende wird der Mean Squared Error zwischen Q-Targets und Q-Evals aus dem Mini-Batch gebildet. Auf Basis dieses Fehlers wird das Q-Network, mittels Backpropagation und Gradientenverfahren, angepasst.

5.3.3. PPO

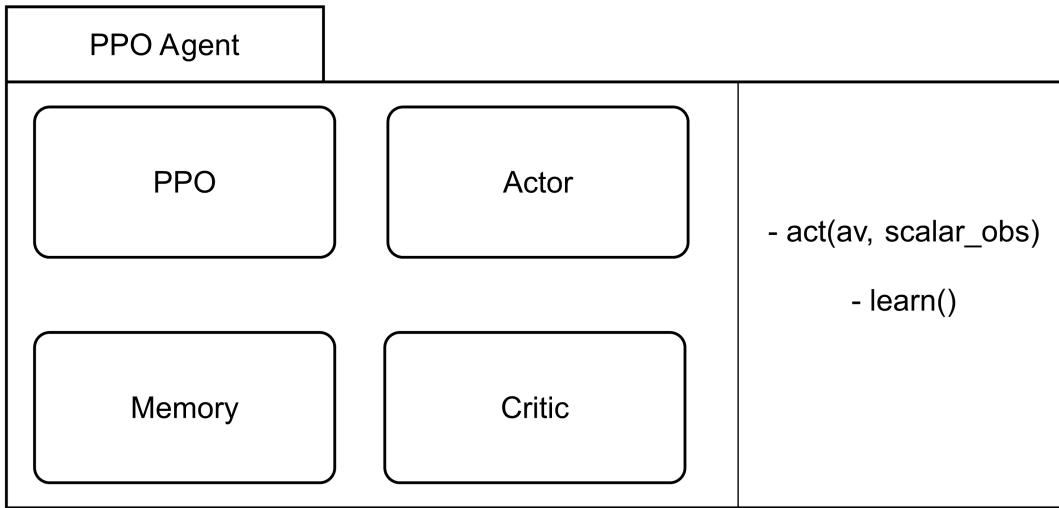


Abbildung 5.10.: Darstellung des PPO-Agenten mit seinen Komponenten.

Der PPO Algorithmus und seine Agenten, bestehen aus vier Komponenten. Diese ermöglichen die Implementierung der Aktionsbestimmung und Lernprozedur.

In der PPO-Komponente werden wichtige Konstanten für den PPO Algorithmus, wie z.B. Gamma (γ), der Epsilon-Clip-Wert (eps_clip) (siehe 2.3.2), die Anzahl der Trainingsläufe pro Datensatz (K_{Epochs}), die Lernrate (lr) und weitere statische Konstanten, gespeichert. Auch werden weitere Unterkomponenten, die im weiteren Verlauf erklärt werden, in dieser Komponente gespeichert.

Die Memory-Komponente speichert Erfahrungen eines PPO-Agenten. Abzuspeichernde Werte, für jeden Schritt, sind dabei die `around_view` (AV), die `scalar_obs` (SO) die Aktion (action), der Reward (reward), die Information, ob man sich in einem terminalen Zustand befindet (terminal) und die logarithmierte Wahrscheinlichkeit der ausgewählten Aktion (`log_prob`).

Die Actor-Komponente verwaltet das Actor-NN. Dieses wird zur Aktionsauswahl genutzt. Die Critic-Komponente verwaltet das Critic-NN. Dieses wird einzig von der Lernprozedur verwendet, um die erwartete Discounted Sum of Rewards zu bestimmen. Mit dieser wird im Trainingsverlauf der Value Loss bestimmt (siehe Gleichung 2.8).

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird in der PPO-Komponente angestoßen. Die AV und SO werden daraufhin durch das Actor-NN propagiert. Der, vom Actor-NN ausgegebene, Tensor der Größe drei (drei mögliche Aktionen), beinhaltet eine Wahrscheinlichkeitsverteilung, auf dessen Basis die nächste Aktion bestimmt wird.

Sei $(0.05, 0.05, 0.9)$ die Wahrscheinlichkeitsverteilung über alle Aktionen. Bestimme man 100 Aktionen unter dieser Verteilung, so würde durchschnittlich 90-mal die Aktion zwei gewählt werden. Aktion null und eins nur rund fünfmal.

Wie beim DQN, wird für die Testläufe immer die Aktion ausgewählt, welche die größte Wahrscheinlichkeit vorweist (siehe Abschnitt 5.3.2).

Lernprozess

Beim Lernprozess des PPO wird wie folgt verfahren:

Zu Beginn werden die Erfahrungen, aus den gespielten Spielen, aus dem Memory (Replay-Buffer) entnommen. Um den Return zu erhalten, werden die einzelnen Rewards aus dem Memory diskontiert (siehe Abschnitt 2.3.2).

Danach wird die folgende Prozedur mehrmals ausgeführt, um das NN zu trainieren. Danach terminiert die Lernprozedur. Für ein besseren Verständnis, wird der Ablauf exemplarisch an einer Erfahrung erklärt.

Zunächst wird die logarithmierte Wahrscheinlichkeit $\pi_\theta(a|s)$ für die gespeicherte Aktion a bestimmt (siehe Abschnitt 2.3.2). Dazu wird die aus dem Memory entnommene AV (around_view) und SO (scalar_obs) durch das Actor- und Critic-NN propagiert. Anschließend wird die logarithmierte Wahrscheinlichkeit der Aktion bestimmt und zusammen mit dem Baseline Estimate (siehe Abschnitt 2.3.2) und der Entropie der Wahrscheinlichkeitsverteilungen (siehe Abschnitt 2.3.2) zurückgegeben. Daraufhin wird die Probability Ratio, aus der soeben bestimmten logarithmierten Wahrscheinlichkeit und der alten logarithmierten Wahrscheinlichkeit des Memory, bestimmt (siehe Abschnitt 2.3.2).

Nachfolgend wird der Advantage, durch Subtraktion des Return mit dem Baseline Estimate, berechnet $\hat{A}(s, a) = R - b(s)$ (siehe Abschnitt 2.3.2).

Als nächstes werden die Surrogate Objective Losses Surr1: $r(\theta) \times \hat{A}(s, a)$ und Surr2: $\text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \times \hat{A}(s, a)$ bestimmt (siehe Abschnitt 2.3.2), mit welchen der Actor-Loss $L^{\text{CLIP}}(\theta) = \min(r_t(\theta) \times \hat{A}(s, a), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \times \hat{A}(s, a))$ berechnet wird (siehe Gleichung 2.2). Um den gesamt Loss des PPO zu bestimmen, wird zusätzlich noch der Value-Loss $L^{\text{VF}} = (V_\theta(s) - V^{\text{targ}})^2$ wobei $V^{\text{targ}} = r(\theta)$ und der Entropy-Loss bestimmt (siehe Abschnitt 2.3.2). Diese werden dann alle zusammengerechnet, entsprechend der Formel: $L^{\text{PPO}}(\theta) = L^{\text{CLIP} + \text{VF} + \text{S}}(\theta) = [L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}} + c_2 S[\pi_\theta](s)]$ (siehe Abschnitt 2.3.2).

Actor- und Critic-NN werden dann mit dem Loss, unter Zuhilfenahme von Backpropagation und Gradientenverfahren, aktualisiert.

5.3.4. Vorstellung der zu untersuchenden Agenten

Ein zentraler Aspekt eines Vergleichs von verschiedenen RL-Agenten ist die genaue Definition dieser einzelnen. Basierende auf den Grundlagen (siehe Abschnitt 2.2.1), sollen hier die zu vergleichenden Agenten vorgestellt werden.

Da die ausgewählten Hyperparameter einen immensen Einfluss auf das Verhalten der Agenten besitzen, ist ein Vergleich zwischen DQN und PPO Agenten mit wahllos gewählten Hyperparametern folglich wenig aussagekräftig. Darum sollen im Weiteren die Wahl der Hyperparameter hier begründet werden.

Wie in der Abbildung 5.11 zu erkennen ist, können beliebig viele Agenten miteinander vergleichen werden. In dieser Ausarbeitung hingegen sollen 6 Agenten definiert und miteinander verglichen werden.

PPO-01 LR_ACTOR= 2e-4 LR_CRITIC=4e-4 GAMMA=0.99 K_EPOCHS=10 EPS_CLIP=0.15	PPO-02 LR_ACTOR= 1.0e-4 LR_CRITIC=2.0e-4 GAMMA=0.93 K_EPOCHS=12 EPS_CLIP=0.2	PPO-03 LR_ACTOR= 1.5e-4 LR_CRITIC=3.0e-4 GAMMA=0.95 K_EPOCHS=10 EPS_CLIP=0.2	DQN-01 LR= 8.0e-4 GAMMA=0.99 BATCH_SIZE=128 MAX_MEM_SIZE = 2**11 EPS_DEC=3e-5 EPS_END=0.001	DQN-02 LR= 2.0e-4 GAMMA=0.90 BATCH_SIZE=128 MAX_MEM_SIZE = 2**10 EPS_DEC=6e-5 EPS_END=0.005	DQN-03 LR= 2.5e-4 GAMMA=0.95 BATCH_SIZE=128 MAX_MEM_SIZE = 2**11 EPS_DEC=4.0e-5 EPS_END=0.002
--	---	---	---	---	---

Abbildung 5.11.: Darstellung der zu untersuchenden Agenten.

Der erste Agent PPO-01 soll ein langsamer aber stetiger Lerner sein. Mit einer ACTOR Lernrate (ACTOR_LR) von 2e-4 und einer CRITC-Lernrate von 4e-4 (CRITIC_LR) wurden Lernraten gewählt, welche, spezifisch für diese Netzstruktur, im Mittelfeld liegen. Ein hoher Wert für GAMMA von 0.99 sorgt für ein zukunftsorientiertes Lernen. Damit der PPO-01 keine zu großen Aktualisierungen der Netze unternimmt, wurde EPS_CLIP auf 0.15 gesetzt, was, verglichen mit der Literatur (Schulman u. a., 2017, S. 6), recht niedrig ist.

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

PPO-03 soll ein Kompromiss zwischen schnellen Lernen und stetigem Fortschritt sein. Mit mittleren Lernraten von 1.5e-4 (Actor) bzw. 2.5e-4 (Critic) sollte ein schneller und zugleich stetiger Lernfortschritt erzielt werden. Der GAMMA-Wert von 0.95 soll das schnelle Lernen unterstützen. Auch die Werte von K_EPOCHS mit zehn

und EPS_CLIP von 0.2 werden in der Literatur (Schulman u. a., 2017, Anhang A) empfohlen und stellt ein gutes Mittelmaß dar.

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

DQN-02 ist wieder als Schnelllerner konzipiert worden. Eine vergleichsweise hohe Lernrate (LR) von 2.0e-4 in Verbindung mit einem kleinen Wert für Gamma von 0.90 soll einen schnellen Lernfortschritt generieren. Dies wird durch eine normale Memory-Size (MAX_MEM_SIZE) von $2^{**}11$ und die große Epsilon-Dekrementierung (EPS_DEC) von 5e-5 soll das Lernen weiter beschleunigt und verstärkt werden. Auch sorgt der verhältnismäßig große Wert für EPS_MIN von 0.0075 für eine schnellerer Exploration und damit für ein schnelles Lernen.

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

5.4. Optimierungen

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

5.4.1. Optimierung A - Joined Reward Function

Die Joined Reward Function wurde im Paper „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ Wei u. a. (2018) vorgestellt und im Abschnitt 4.1 erklärt. Sie setzt sich aus drei Teilen zusammen. Die Basis bildet ein Distanz Reward, welcher abhängig von der Distanz und Schwanzlänge ist. Um unerwünschte

Lerneffekte, von beispielsweise der Neuerzeugung eines Apfels, zu verhindern werden diese Erfahrungen nicht im Memory gespeichert. Zur Verstärkung des Pathfindings wird die Timeout Strategy angewendet, welche den Agenten für nicht zielgerichtetes Verhalten, wie z.B. das unnötige Umherlaufen, bestraft.

Eine genaue Implementierung dieser vorgestellten Reward Funktion erscheint nicht Sinnvoll, da bereits in der Diskussion (siehe 4.1.1) zur Quelle Wei u. a. (2018) festgestellt worden ist, dass die Agenten nicht für das effiziente Lösen des Snake-Spiels konzipiert worden sind, sondern für das lange Überleben. Daher muss die Reward Funktion entsprechendes Verhalten begünstigen.

Dennoch erscheint die Adaptierung einiger Reward Funktion Elemente als sinnvoll, um eine eigene, für die Performance und Effizienz optimierte, Reward Funktion zu designen.

Die Implementierung dieser neuen Reward Funktion findet in der Reward-Komponente statt. Dabei wird auf den Distanz Reward der Quelle Wei u. a. (2018) gesetzt, da dieser viele Faktoren des Spiels berücksichtigt, im Gegensatz zur Standard Reward Funktion (siehe Abschnitt 5.2.1). Dabei wird der Reward wie folgt berechnet:

$$\Delta r(L_t, D_t, D_{t+1}) = \log_{L_t} \frac{L_t + D_t}{L_t + D_{t+1}} \quad (5.2)$$

Wobei t den vorherigen, $t + 1$ den aktuellen Zeitpunkt darstellen. L_t ist die Länge der Snake zum vorherigen Zeitpunkt und D_t und D_{t+1} stellen die Distanzen zwischen Snake und Apfel zum vorherigen und aktuellen Zeitpunkt dar.

Dieser Distanz Reward wird, wie in Quelle Wei u. a. (2018) dargestellt, auf einen Initial-Wert aufaddiert. Nur setzt sich dieser nicht aus den vergangenen Rewards zusammen, sondern ist in dieser Ausarbeitung fest auf -0.01 gesetzt. Zum Schluss wird dann der Reward noch zwischen -0.02 und -0.005 geclipt, damit der Agent stetig bemüht ist, den optimalen Weg zu finden. $r_{res} = clip((-0.01 + \Delta r), -0.02, -0.005)$

5.4.2. Optimierung B - Anpassung der Lernrate

Die zweite Optimierung wurde mithilfe von Anregungen aus der Literatur (Lapan, 2020, S. 331 f.) erzeugt. In dieser wurde die Steigerung der Lernrate diskutiert, um einen schnelleren Lernerfolg zu erzielen. Gegenteilig könnte jedoch die Senkung der Lernrate während des Trainings die Performance und Siegrate verstärken, da die Aktualisierung des NN nicht mehr so stark ausfällt und bestehender Fortschritt damit erhalten bleibt. Darum soll die Lernrate immer dann mit 0.95 multipliziert werden, sobald keine Performance Steigerung in den letzten 100 Epochs bzw. Trainingsspiele erzielt wurde.

5.5. Datenerhebung und Verarbeitung

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

5.5.1. Datenerhebung

Die Hauptablaufroutine, implementiert den eigentlichen Test und Trainingsablauf. Um diese auszuführen werden wichtige Hyperparameter des Ablaufs, wie z.B. die zu absolvierenden Trainingsspiele (N_ITERATIONS), die Spielfeldgröße (BOARD_SIZE) und weitere Agenten spezifische Hyperparameter, übergeben. Um einen angemessenen Zeitraum für das Lernen zu schaffen, sollen 30.000 Spiele bzw. Epochs für einen Trainingslauf absolviert werden. Die Spielfeldgröße soll dabei standardmäßig, für das Training, bei (8x8) liegen.

Bei der Datenerhebung ist streng zwischen Test- und Trainingsdaten zu unterscheiden, wobei die Testdaten aus einfachen Spielabläufen bzw. Testabläufen und die Trainingsdaten aus den Trainingsabläufen stammen. Letztere werden wie folgt erhoben.

Zu Beginn werden die Datenhaltungsobjekt apples, wins, steps initialisiert, welche für jedes absolvierte Trainingsspiel die, dem Namen des Objektes entsprechenden, Werte speichert. Nach der Erstellung des Agenten und Environments startet der Spielverlauf.

Dabei wird wie in Abschnitt 2.2.2 vorgegangen. Der Agent erhält eine Obs, bestimmt seine Aktion und diese wird sogleich im Env ausgeführt. Danach werden (neue) Obs und Reward sowie weitere Statusinformationen ausgegeben. Diese Daten werden im Memory des jeweiligen Agenten für das sich anschließende Training gespeichert. Dieses wird, wie im DQN-Lernprozess (siehe Abschnitt 5.3.2) bzw. PPO-Lernprozess (siehe Abschnitt 5.3.3) dargestellt, durchgeführt. Danach werden die oben genannten Datenhaltungslisten aktualisiert und die Prozedur beginnt von neuem. Wenn alle Epochs (N_ITERATION) absolviert wurden, werden die Erhobenen Daten weiter in der Statistik-Komponente verarbeitet (siehe Abschnitt 5.5.2).

Die Datenerhebung für Testdaten findet annähernd auf die gleiche Weise statt. Jedoch werden die Daten in nicht aus Trainingsläufen entnommen, sondern aus Testläufen. Bei den Daten handelt es sich um die gleichen, wie in der oben erwähnten Datenerhebung für Trainingsdaten. Der Unterschied zwischen Test- und Trainingsläufen besteht darin, dass die Spielfeldgröße im Rahmen des Testlaufes für

die Robustheit variiert wird. Sie kann dabei Größen zwischen (6, 6) bis (10, 10) annehmen. Bei der Messung der Siegrate, Effizienz und der Performance bleibt das Spielfeld jedoch konstant bei einer Größe von (8x8). Anders als bei Trainingsläufen werden zudem bei keine 30.000 Epochs durchgeführt sondern nur 5.000. Dies entspricht ungefähr der Größe einer Datenmenge zur Validierung (Goodfellow u. a., 2018, S. 134).

Diese Testdaten werden dann ebenfalls der Statistik-Komponente übergeben (siehe Abschnitt 5.5.2). Um die Validität der Statistiken zu garantieren, sollen alle Daten für die Statistiken zwei mal erhoben werden. Damit soll der Anforderung der mehrfachen Datenerhebung entsprochen werden (siehe Abschnitt 3.3.1).

5.5.2. Datenverarbeitung und Erzeugung von Statistiken

Nach der Erstellung der Trainings- und Testdaten für jeden Agenten, werden diese, entsprechend der Evaluationskriterien (siehe Tabelle 3.2), verarbeitet. Die Trainingsdaten werden dabei mit Statistiken abgebildet. Die Testdaten werden in Tabellen dargestellt, da diese keinen Verlauf mehr darstellen, sondern nur den momentanen Stand wiedergeben.

Diese Verarbeitung der Daten geschieht wie folgt:

Die Performance wird anhand der gefressenen Äpfel gemessen. Die Effizienz errechnet sich aus der Anzahl der gefressenen Äpfel (apples) dividiert durch die gegangenen Schritte (steps). Es entsteht daher die Apfel pro Schritt Rate. Die Robustheit wird gleich der Performance gemessen, nur werden dafür Testdaten, welche auf Spielfeldern mit variabler Größe erhoben worden sind, genutzt. Die Spielfeldgröße soll dabei zwischen (6x6) bis zu (10x10) liegen. Die Siegrate wird mithilfe der wins bestimmt. Dabei werden die Siege der jeweils 100 letzten Spiele bzw. Epochs gemittelt.

Diese Daten werden dann mithilfe von Statistiken ausgewertet. Dabei soll jeweils eine Statistik bzw. Tabelle pro Evaluationskriterium angefertigt werden. Die Daten, welche aus den mehrfachen Erhebungen stammen, sollen, bei der Auswertung, gemittelt werden, um jeweils eine Statistik bzw. Tabelle für jeden Vergleich zu erhalten. Auf diese Verfahrensweise wird jedoch nochmal in der Evaluation (siehe Abschnitt 7.2.3) hingewiesen.

Kapitel 6

Implementierung

In diesem Kapitel soll die Implementierung des Spiels Snake, der beiden Algorithmen, der Ablaufroutinen sowie der Statistik-Erzeugung thematisiert werden. Als Programmiersprache wurde Python (3.7) gewählt, da es über viele Frameworks im Bereich Machine Learning verfügt. In dieser Implementierung wird das Machine Learning Framework PyTorch (<https://pytorch.org/>) verwendet.

6.1. Snake Environment

Zur Implementierung des Spiels Snake wurde das Framework gym von OpenAI genutzt (siehe <https://gym.openai.com/>). Das Snake Environment, implementiert, entsprechend der Anforderung 3.1.1, drei zentrale Methoden, welche für Informationsaustausch mit dem Agenten sorgen. Diese sind im Konzept (siehe Abschnitt 5.2.2) aufgeführt. Die step Methode wird in der Ablaufprozedur (Trainings- bzw. Testmethode) aufgerufen. Um die ihr übergebene Aktion auszuführen, ruft diese die action Methode in SnakeGame auf (siehe Abbildung 6.1). Diese führt die Aktion aus und manipuliert damit das SnakeGame. Da die action Methode die eigentliche Aktionsausführung (siehe Abschnitt 5.2.1) implementiert, wird diese noch näher erläutert.

Nach dem Aufrufen von action ist die Aktion ausgeführt und es wird die nächste Observation, mithilfe von make_obs Methode, bestimmt, welche dem Agenten übergeben wird, damit dieser die nächste Aktion bestimmen kann. Im Anschluss wird ebenfalls noch der Reward gebildet sowie die Statusinformation, ob sich das Spiel in einen terminalen Zustand befindet und damit zurückgesetzt werden muss.

Die reset Methode (siehe Abschnitt 5.2.2) wird zu Beginn des Spielablaufs aufgerufen, um eine initiale Obs zu erhalten. Sie setzt den momentanen Spielfortschritt zurück, (siehe Abschnitt 5.2.1). Dazu wird die reset_snake_game Methode aufgerufen, welche das SnakeGame und den Player zurücksetzt. Anschließend wird eine neue Obs mithilfe der make_obs Methode (siehe Abschnitt 5.2.1) erstellt und zurückgegeben.

Die render Methode aktualisiert die GUI, in dem diese die updateGUI aufruft, siehe 6.1.

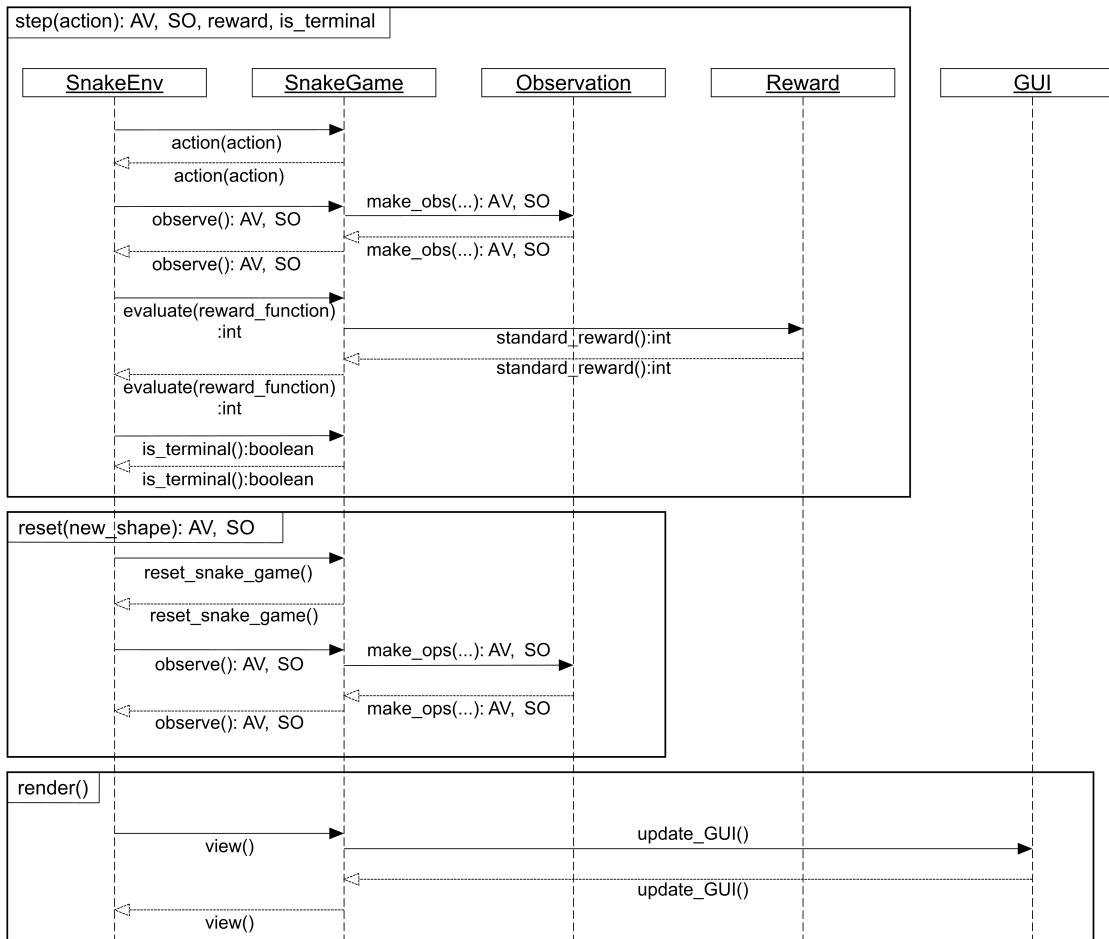


Abbildung 6.1.: Sequenzdiagramm der Schnittstellenmethoden step, reset und render

Die Spiellogik ist hauptsächlich in der action Methode implementiert. Sie basiert auf den Beschreibungen des Konzepts (siehe Abschnitt 5.2.1). Ein schematischer Ablauf der action Methode ist in Abbildung 6.2 dargestellt. Zu Beginn wird geprüft, ob die maximale Anzahl an Schritten ohne einen Apfel gefressen zu haben überschritten ist. Sollte dies der Fall sein, so wird is_terminal gesetzt und die Methode terminiert. Andernfalls, wird die Aktion umgesetzt, indem die Player.direction angepasst wird. Diese gibt die Laufrichtung an und bestimmt im nächsten Schritt, in welcher neuen Position sich der Kopf der Snake befindet. Daraufhin wird überprüft, ob der neue Kopf außerhalb des Spielfeldes liegt. Wenn dies zutrifft, wird das Spiel terminiert. Andernfalls wird der neue Kopf in die Liste alle Snake-Glieder (Player.tail), an erster Stelle, eingefügt. Sollte die Snake zu diesem Zeitpunkt das gesamte Spielfeld ausfüllen, so hat sie gewonnen und action terminiert. Sonst wird geprüft, ob die Snake im momentanen Schritt einen Apfel gefressen hat oder nicht. Sollte dies eintreten, so wird der Apfel entfernt und ein neuer generiert. Zudem wird der inter_apple_steps

Zähler zurückgesetzt. Ansonsten wird das letzte Schwanzstück entfernt, um die Illusion von Bewegung zu erzeugen und der inter_apple_steps Zäher wird des Weiteren inkrementiert.

Zu diesem Zeitpunkt ist es noch möglich, dass sich Duplikate in der Player.tail List befinden. Sollte dies der Fall sein, so ist die Snake in sich selbst gelaufen und action terminiert. Andernfalls wird Playground mit den neuen Elementen der Snake aktualisiert.

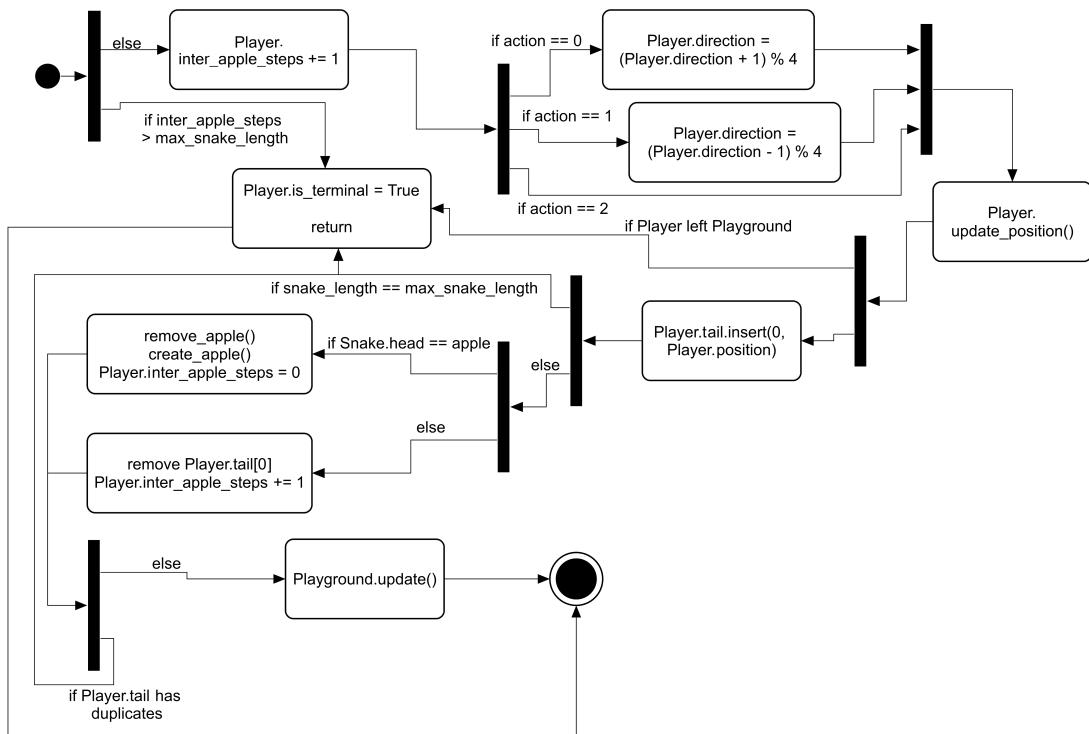


Abbildung 6.2.: Ablaufdiagramm der action Methode

6.2. AV-Network

Das AV-Network stellt die Netzstruktur dar, welche die AV (around_view) verarbeitet. Um dies bewerkstelligen zu können, wird das Netz mithilfe des PyTorch Frameworks erstellt. Dabei wird so verfahren, wie es im Konzept (siehe 5.3.1) gefordert ist. Eine genauere Darstellung der Implementierung des AV-Networks findet sich im Anhang (siehe B.3). Das AV-network wird von allen Agenten genutzt.

6.3. DQN

Der DQN Algorithmus, aus welchen die DQN Agenten hervorgehen, stellt einen der beiden zu implementierenden Algorithmen dar und beinhaltet die folgenden Klassen: Die Agent Klasse implementiert die act und learn Methoden. Sie ist dabei als DQN-

Komponente zu interpretieren (siehe Abschnitt 5.3.2). Die Memory Klasse speichert die Erfahrungen der Agenten für das spätere Lernen (siehe Abschnitt 5.3.2). Die QNetwork Klasse beinhaltet das NN zur Aktionsbestimmung (siehe Abschnitt 2.4). Diese sollen vor der näheren Thematisierung der act und lern Methode, erklärt werden.

6.3.1. Q-Network

Das Q-Network stellt das Instrument zur Bestimmung der Q-Values dar und besteht aus dem AV-Network (AV_NET) (siehe Abschnitt 6.2) und dem Q-Network-Tail (Q-net)(siehe Abschnitt 5.3.1). Es wird durch das PyTorch Framework realisiert und besteht aus den in Abschnitt 5.3.1 erwähnten Elementen. Das Q-net besteht dabei aus den, im Konzept erwähnten, Elementen (siehe 5.3.1). Die forward Methode leitet die AV (av) durch das AV_NET (siehe B.3). Danach wird der Output des AV_Net mit der SO (scalar_obs) verbunden und durch das Q_net geleitet. Das Ergebnis wird zurückgeliefert.

6.3.2. Memory

Die Klasse Memory besteht aus einer Reihe von Tensoren, welche die generierten Daten, mittels einer zusätzlichen Dimension, speichern. Sollte die AV die Form (6x13x13) besitzen, so besitzt der Speicher-Tensor (AV) die Form (MEM_SIZEx6x13x13). Das Memory verfügt über eine get_data Methode, welche einen zufallsbasierten Batch an Erfahrungen zurückliefert. Die im Konzept erwähnte Ring Buffer Funktionalität wird durch einen Zähler realisiert, welcher die position der neuen Erfahrung angibt. Sollte dieser die maximale Memory Größe überschreiten, so wird dieser wieder auf null gesetzt. (siehe Abschnitt 5.3.2)

6.3.3. DQN-Agent

Der DQN Agent besteht aus den act Methoden, welche die Aktionen bestimmen und aus der learn Methode, die für das Training zuständig ist. Zudem verwaltet er Instanzen der Memory und QNetwork Klassen.

Aktionsbestimmung

In der Agent Klasse werden die zwei Methoden act und act_test definiert. Die act Methode wird dabei analog zur Beschreibung im Konzept (siehe Abschnitt 5.3.2)) implementiert. Die act_test Methode ist für die Aktionsbestimmung während der Testläufe zuständig und ermittelt seine Aktionen ausschließlich mittels des Q-Networks (siehe Code Listing 6.1).

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

Code Listing 6.1: Aktionsbestimmung

Trainingsroutine

Die learn Methode stellt das Herzstück eines Agenten dar. Sie wird entsprechend der Grundlagen (siehe Abschnitt 2.3) und der Ausführungen in Abschnitt 5.3.2 implementiert. Die learn Methode überprüft als erstes, ob sich aus dem Memory genügend Daten für einen Mini-Batch entnehmen kann. Sollte dies nicht der Fall sein, so terminiert die Methode. Danach wird die get_data Methode des Memory aufgerufen, welche einen Mini-Batch liefert. Danach werden die Schritt, entsprechend der Beschreibung im Konzept (siehe 5.3.2), durchgeführt.

Zu Beginn werden daher die Q-Values der gespeicherten Aktion für die gegenwärtigen Zustände bestimmt. Danach folgen Q-Values aller Aktionen der Nachfolgezustände (siehe Code Listing 6.2).

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

Code Listing 6.2: Bestimmung der Q-Values

Daraufhin wird, wie in den Abschnitten 2.10 und 5.3.2 dargestellt, Q-Target ($r(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$) bestimmt, siehe auch Code Listing 6.3.

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

Code Listing 6.3: Bestimmung von Q-Target

Zum Schluss wird der Loss des DQN bestimmt und das Q-Net aktualisiert.

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

Code Listing 6.4: Bestimmung des DQN Loss & Update des Q-Networks

Zusätzlich wird noch Epsilon verringert, um die Anzahl an Zufallsaktionen während des nächsten Trainingslauf zu senken.

6.4. PPO

Der PPO Algorithmus, aus welchen die PPO Agenten hervorgehen, beinhaltet die folgenden Klassen:

Die Agent Klasse definiert, die learn Methode. Sie ist dabei als PPO-Komponente zu interpretieren (siehe Abschnitt 5.3.3). Die Memory Klasse speichert die gesammelten Erfahrungen. Die ActorNetwork bzw. CriticNetwork Klassen beinhaltet die NN des Actors bzw. Critics. In der ActorCritic Klasse befindet sich zentrale Methoden zur Durchführung des Lernens, wie z.B. evaluate. Des Weiteren verbindet die ActorCritic Klasse Actor-NN und Critic-NN miteinander. Eine weitere Betrachtung der einzelnen Klassen folgt im weiteren Verlauf.

6.4.1. Actor und Critic

Der Actor bzw. Critic wird durch die Klasse ActorNetwork bzw. CriticNetwork aufgespannt, welche von der Module Klasse des PyTorch Frameworks erbt. Damit lassen sich ActorNetwork und CriticNetwork als NN-Baustein benutzen. Neben dem AV-Network (siehe Abschnitt 6.2 und 5.3.1) wird noch der Actor-Tail bzw. Critic-Tail definiert. Dabei handelt es sich um das NN, welches die Ausgabe vom AV-Network mit der SO verbindet und aus diesem Ergebnis eine Wahrscheinlichkeitsverteilung über alle Aktionen bzw. ein Value bestimmt. Exemplarisch wird die Klasse Actor-Networks im Code Listing 6.5 dargestellt. Analog verhält es sich beim Critic, mit der Ausnahme, dass dieser nicht Tensoren der Länge drei sondern der Länge eins ausgibt. Diese stellen die Baseline Estimates bzw. Value Werte (siehe 2.3.2) dar.

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

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

Code Listing 6.5: ActorNetwork

Die forward Methoden von Actor und Critic sind dabei analog zu der, welche im Abschnitt 6.3.1 vorgestellt wurde. Sowohl ActorNetwork als auch CriticNetwork findet man jedoch nur gekapselt in der ActorCritic Klasse vor.

6.4.2. ActorCritic

Die ActorCritic Klasse verbindet Actor und Critic miteinander. Sie dient daher als eine NN Schnittstelle, welche die drei Methoden act, act_test und evaluate definiert. Wie die ActorNetwork und CriticNetwork Klassen, erbt die ActorCritic Klasse von Module. Sie kann daher aus den ActorNetwork und dem CriticNetwork ein gesamt Network aufspannen, welches alle Parameter der beiden NNs besitzt. in diesem ist auch die Aktionsdurchführung ()

Aktionsbestimmung

Die PPO Agenten verfügen, wie die DQN Agenten, über zwei act Methoden. Die erste leitet die AV und SO durch das ActorNetwork und erhält eine Wahrscheinlichkeitsverteilung über alle Aktionen. Diese wird auch als Policy bezeichnet. Daraufhin wird eine Aktion entsprechend der Wahrscheinlichkeitsverteilung bestimmt und zusammen mit ihrer logarithmierten Wahrscheinlichkeit und den Tensor AV und SO zurückgegeben. Die act Methode wird dabei analog zum Konzept (siehe Abschnitt 5.3.3) implementiert.

Die act_test Methode verzichtet, wie beim DQN (siehe Abschnitt 6.3.3), wieder auf Zufallselemente. Am Ende der act_test Methode (siehe 6.6) wird jedoch lediglich die Aktion ausgewählt, welche die größte Wahrscheinlichkeit vorweist.

```
@T.no_grad()
def act_test(self, av, scalar_obs):
    av = T.from_numpy(av).to(self.DEVICE)
    scalar_obs = T.from_numpy(scalar_obs).to(self.DEVICE)
    policy = self.ACTOR(av, scalar_obs)
    return av, scalar_obs, T.argmax(policy).item()
```

Code Listing 6.6: Darstellung der act_test Methode

Neben der Aktionsbestimmung implementiert die ActorCritic Klasse auch noch die evaluate Methode, welche für den Lernablauf unerlässlich ist.

Evaluate Methode

Die evaluate Methode, bestimmt die logarithmierte Wahrscheinlichkeit einer Aktion unter einer anderen bzw. älteren Policy (Wahrscheinlichkeitsverteilung) (siehe 5.3.3).

Neben dieser, bestimmt sie auch noch den Value-Wert des Critics und ermittelt die Entropy der Policy.

```
def evaluate(self, av, scalar_obs, action):
    policy, value = self.forward(av, scalar_obs)
    dist = Categorical(policy)
    action_probs = dist.log_prob(action)
    dist_entropy = dist.entropy()
    return action_probs, T.squeeze(value), dist_entropy
```

Code Listing 6.7: Darstellung der evaluate Methode

Diese Werte werden im Weiteren für die Bestimmung des PPO Loss benötigt. Damit dieser bestimmt werden kann, bedarf es Erfahrungen, welche sich im Memory befinden.

6.4.3. Memory

Das Memory oder auch Replay Buffer genannt, besteht aus einer Reihe von Tensoren, welche die generierten Daten mittels einer zusätzlichen Dimension speichert (siehe Abschnitt 6.3.2). Die Rewards und Terminals (is_terminal) werden jedoch in Listen eingepflegt, da dies das spätere Diskontieren (Abzinsen) erleichtert.

Mit der get_data Methode werden die gesamten Erfahrungen der letzten Spielepisode zurückgegeben. Sobald diese zum Lernen herangezogen worden sind, werden diese gelöscht bzw. überschreiben. Dazu wird die Lernprozedur verwendet, welche sich in der Agent Klasse befindet.

6.4.4. Agent

Die Agent Klasse implementiert die Lernmethode und verwaltet das Memory und zwei ActorCritic Networks. Diese stellen die alte und neue Policy dar. Während mit der alten Policy immer die Trainingsdaten generiert werden, wird mit der neuen Policy trainieren. Nach dem Lernen wird die alte Policy mit der neuen aktualisiert. Dieser Lernprozess geschieht dabei wie im Abschnitt 6.4.4 dargestellt.

Learn Methode

Als erstes wird überprüft, ob das Memory mehr als 64 Erfahrungen besitzt. Sollte dies nicht der Fall sein, so wird die Methode terminiert. Die bereits gespeicherten Erfahrungen bleiben erhalten. Ansonsten werden die Daten aus dem Memory mit der get_data Methode entnommen. Danach werden die Rewards mit der generate_rewards Methode diskontiert (abgezinst), (siehe Code Listing 6.8 und Abschnitt

5.3.3). Diese implementiert die Funktionalität, welche im Abschnitt 2.3.2 dargestellt wird.

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

Code Listing 6.8: Diskontierung der Rewards

Diese diskontierten Rewards (Return) werden im Anschluss noch normalisiert, um ein stetigeres Lernen zu ermöglichen. Danach wird die folgende Prozedur K_EPOCHS mal wiederholt.

Die Erfahrungen werden zufallsbasiert durchmischt, um ein stabileres Lernen zu ermöglichen. Danach wird die evaluate Methode mit den Erfahrungen aufgerufen. Diese bestimmt die neuen logarithmierten Wahrscheinlichkeiten (log_probs) aller Aktionen. Mit diesen und den gespeicherten alten (old_log_probs) werden daraufhin die ratios (siehe Abschnitt 2.3.2) gebildet.

Zuzüglich wird mithilfe der Values, welche von evaluate stammten (siehe Abschnitt 6.4.2), die Advantages aller Erfahrungen erstellt (siehe Abschnitt 2.3.2). Dies geschieht dabei wie im Abschnitt 5.3.3 beschrieben.

```
probs, state_values, dist_entropy = self.POLICY.evaluate(old_av_b,
                                                       old_scalar_b, old_action_b)
ratios = T.exp(probs - probs_old_b)
advantages = rewards_b - state_values.detach()
```

Code Listing 6.9: Bestimmung der Ratios und Advantages

Mit diesen Werten ist es nun möglich, die Surrogate Losses (siehe Abschnitt 2.3.2) und anschließend den Actor-Loss bzw. Clip-PPO-Loss (siehe Gleichung 2.2) zu bestimmen.

```
surr1 = ratios * advantages
surr2 = T.clamp(ratios, 1 - self.EPS_CLIP, 1 + self.EPS_CLIP) *
        advantages
loss_actor = -(T.min(surr1, surr2) + dist_entropy * self.
               ENT_COEFFICIENT).mean()
```

Code Listing 6.10: Bestimmung der Surrogate Losses

In diesem befindet sich, zur einfacheren Handhabung, noch der Entropy-Loss mit integriert (siehe Abschnitt 2.3.2). Zur Bestimmung des gesamt Loss fehlt noch der Critic-Loss, welcher mit dem MSE (Mean Squared Error) bestimmt wird. Dabei wird der quadrierte Fehler zwischen den Returns und den Values des Critics ermittelt und im Anschluss gemittelt. Danach können Actor- und Critic-Loss zusammenaddiert und mit diesem entstandenen PPO-Loss dann das Actor_Critic NN aktualisiert werden.

```
loss_critic = self.CRITIC_COEFFICIENT * self.LOSS(rewards_b,
                                                 state_values)
loss = loss_actor + loss_critic
self.POLICY.OPTIMIZER.zero_grad()
loss.backward()
self.POLICY.OPTIMIZER.step()
```

Code Listing 6.11: Bestimmung des PPO-Losses & Update der Netze

Zu Schluss wird, nachdem die Prozedur K_EPOCHS-mal durchgeführt wurde, die alte Policy mit der neuen aktualisiert. Zuzüglich wird das Memory geleert bzw. mit den nächsten Erfahrungen überschrieben.

6.5. Train Methoden

Zu Beginn werden Agent und Env zusammen mit den Datenhaltungslisten (siehe 5.5.1) erstellt. Ein Scheduler wird, für die Optimierung B (siehe 5.4.2), ebenfalls erzeugt. Danach wird die Lernprozedur durchgeführt.

Zu Beginn dieser, wird eine Obs bestehend aus AV (around_view) und SO (scalar_obs) generiert. Diese Obs wird dem Agenten übergeben, welcher eine Aktion und die weiteren benötigten Daten für den jeweiligen Algorithmus zurückgibt, beim PPO z.B. die logarithmierte Wahrscheinlichkeit für die ausgewählte Aktion (log_probability). Stellvertretend für alle Agenten wird hier die Trainingsmethode des PPO für Beispiele herangezogen. Analog wird beim DQN verfahren. Danach wird diese bestimmte Aktion im Env ausgeführt. Dabei wird mit dem OPTIMIZATION Hyperparameter (siehe C.1) gesteuert, welche Reward Funktion verwendet wird (siehe 5.4.1).

```
for i in range(1, N_ITERATIONS + 1):
    av, scalar_obs = game.reset(get_random_game_size() if
                                 RAND_GAME_SIZE else None)

    while not game.hasEnded:
        av, scalar_obs, action, log_probability = agent.OLD_POLICY.act(
            av, scalar_obs)
```

```

    av_, scalac_obs_, reward, is_terminal, won = game.step(action,
                                                       OPTIMIZATION)
    agent.MEM.store(av, scalar_obs, action, log_probability, reward
                    , is_terminal)

    av = av_
    scalar_obs = scalac_obs_

    agent.learn()

    wins.append(won)
    apples.append(game.apple_count)
    steps_list.append(game.game.step_counter)

```

Code Listing 6.12: Spielablauf & Datenspeicherung

Danach wird das Memory mit den entstandenen Erfahrungen aktualisiert. Sollte die Spielepisode terminieren, so wird die learn Methode des Agent aufgerufen und die generierten Episodendaten werden in die Datenhaltungslisten eingefügt (siehe 5.5.1). Beim DQN wird die learn Methode nach jedem fünften Schritt aufgerufen, um eine die Trainingsdatenmenge zu erhöhen und damit das Lernen zu stabilisieren. Ansonsten ist das Prozedere analog zum PPO.

Damit der Lernprozess, wie im Konzept dargestellt, bei einer Siegrate von 60% stoppt, wird die Siegrate der letzten 100 Spiele ermittelt und sollte diese größer als 60% sein, so wird der Lernprozess terminiert und die Daten in den Datenhaltungslisten werden gespeichert.

```

if sum(wins[-100:]) / 100 > 0.6:
    save("PPO", AGENT_NUMBER, STATISTIC_RUN_NUMBER, "train", RUN_TYPE,
         RAND_GAME_SIZE, agent, dtime,
         steps_list, apples, scores, wins)
    return

```

Code Listing 6.13: Abbruchbedingung

Wird die Optimierung B (siehe 5.4.2) verwendet, so wird alle 100 Spielepisoden die Steigung der Performance der letzten 100 Trainingsspiele bzw. Epochs ermittelt. Sollte diese nicht größer als null sein, so wird die Lernrate mit 0.95 multipliziert und damit gesenkt. Dies soll jedoch erst in der Endphase des Lernen umgesetzt werden, um die Lernrate nicht zu Beginn zu stark zu senken. Daher wird diese erst bei Überschreiten von 15.000 Epochs (Trainingsspielen) angepasst.

```

if i > 15000 and i % 100 == 0 and SCHEDULED_LR:
    m, b, _, _, _ = linregress(list(range(100)), apples[-100:])
    if m <= 0:

```

```
scheduler.step()
```

Code Listing 6.14: Anwendung von Optimierung B - Scheduler

Sind alle 30.000 Epochs abgeschlossen, wird das NN und die Trainingsdaten gespeichert und die Methode terminiert.

Sollte man sich dazu entscheiden den Trainingsprozess vorzeitig abbrechen zu wollen, so wird man vom System gefragt, ob die Daten und das NN gespeichert werden sollen. Mit der Betätigung von der Taste „y“ werden NN und Daten gespeichert mit „n“ werden diese nicht gespeichert. Danach terminiert die Methode.

6.5.1. Test Methoden

Die Test Methoden sind bis auf wenige Ausnahmen mit den Train Methoden übereinstimmend. Es werden in diesem Abschnitt daher nur die Unterschiede aufgezeigt. Einer dieser besteht in dem Hyperparameter MODEL_PATH, welche den Speicherort repräsentiert, unter dem sich das NN befindet. Mit diesem wird dann der Test durchgeführt. Alle Elemente des Lernens sind aus der test Methode entfernt, zu diesen gehören der Scheduler, das Aufrufen der learn Methode und das Speichern des NNs.

Hinzukommt die Prozedur, um die Spielfeldgröße zu ändern, für die Bestimmung der Robustheit (siehe 5.5.2 und 3.5).

Des Weiteren wird für die Aktionsbestimmung nun die act_test Methode verwendet und es kommt die Funktionalität der graphischen Umsetzung hinzu, indem die render Methode des Env aufgerufen wird, sofern die GUI in den Hyperparametern nicht ausgeschaltet worden ist. Ansonsten treten keine Unterschied zwischen den Train und der Test Methoden auf.

6.6. Speicherung

Sämtliche Daten und Networks werden in den baseline-run-n Unterordnern gespeichert. Man findet daher in den Hyperparametern die STATISTIC_RUN_NUMBER, welche für die Benennung des Unterordners zuständig ist. Sollte die save Methode, welche für das Speichern der Daten und des NNs zuständig ist, mit einer STATISTIC_RUN_NUMBER aufgerufen werden, für die noch kein Ordner erstellt wurde, so wird dies automatisch geschehen. Der Parameter RUN_TYPE ist für die Benennung des Unterordners zuständig. Es gibt die Möglichkeit zwischen "baseline und optimized". Der USE_CASE gibt an, ob es sich um einen Test- oder Trainingslauf handelt. Die AGENT_NUMBER bestimmt unter welchen Namen das

NN und die Daten abgespeichert werden soll. Wird eine eins übergeben so werden die Dateien "PPO-01-train.csv" und "PPO-01-train.model" gespeichert. Sollte ein Training bzw. Test mit der zufallsverteilten Spielfeldgröße durchgeführt werden, so wird dies, in der Datei, mit dem Namenszusatz "rgs"(random game size) signalisiert. Die Trainingsdateien würden dann "PPO-01-rgs-train.csv" und "PPO-01-rgs-train.model" heißen. Mit dem Parameter OPTIMIZATION lässt sich die Optimierung auswählen, welche im Namen der Dateien deutlich gemacht wird. Eine NN, welches mit einer Optimierung trainiert wurde, würde dann unter dem Namen "PPO-01-opt-a-train.model" gespeichert werden usw.

6.7. Statistik

Die Statistiken werden mit der generate_statistic Funktion erstellt, welche sich in der statisticTool Datei befindet. Diese ist wiederum im statistic Ordner definiert (siehe ??). Ihr wird die STATISTIC_RUN_NUMBER, der RUN_TYPE die OPTIMIZATION und eine Agenten-Liste übergeben. Die STATISTIC_RUN_NUMBER gibt an, welcher Run ausgewertet werden soll. Der RUN_TYPE differenziert zwischen der Auswertung von baseline und optimized Daten. Der USE_CASE unterscheidet die Auswertung von Test- und Trainingsdaten. Mit der Agenten-Liste können gezielt Statistiken zu einzelnen Agenten angefertigt werden. Wird eine leere List übergeben, so wird jeder Agent, welcher über Daten im Run-Ordner verfügt, untersucht.

Zuerst wird der, durch die übergebenen Parameter aufgespannte, Path bestimmt. Danach werden alle CSV-Dateien, welche die Test- und Trainingsdaten enthalten, eingelesen und deren Daten gespeichert. Sollte die Agenten List nicht leer sein, so werden die Daten, der Agenten, die sich nicht in der Liste befinden, gelöscht. Die CSV-Datei bleibt erhalten. Danach werden die Daten entsprechend der Darstellungen im Abschnitt 5.5 bereitgestellt. Dazu werden Dataframes des Framework Pandas (siehe <https://pandas.pydata.org/>) genutzt. In diesen befinden sich die Anzahl der gefressenen Äpfel (apples), der gegangenen Schritte (steps) und der Siege (wins). Diese Werte werden mithilfe des Pandas Frameworks für die Statistik aufbereitet. In diesem Fall wird der Durchschnitt über die letzten 100 Werte pro Epoch (Spiel) gebildet. Dieser Vorgang wird für alle Agenten durchgeführt.

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

```
values()]\nwin_rate = [value["wins"].rolling(100).mean().fillna(0) for value\nin agent_dict.values()]
```

Code Listing 6.15: Erstellung der Daten entsprechende der Evaluationskriterien

Danach werden die Statistiken mit der make_statistics Unterfunktion erstellt. Diese generiert, mithilfe des Matplotlib Frameworks, die Graphiken. Dafür wird die Größe und Achsenbeschriftung definiert, sowie die Farben der Kurven und das Vorhandensein einer Gitters in der Grafik.

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

Kapitel 7

Evaluation

In dem Kapitel Evaluation sollen die Ergebnisse der angewendeten Methodik präsentiert werden. Des Weiteren wird eine Anforderungsanalyse durchgeführt, um zu beurteilen, welche Anforderungen umgesetzt wurden. Sollen einige nicht umsetzbar gewesen sein, so werden diese in diesem Abschnitt erläutert.

7.1. Ergebnisevaluation der Vergleiche

In der Evaluation sollen die Ergebnisse, welche aus den Vergleichen stammten präsentiert werden. Dabei wird sich am Vorgehen orientiert.

7.1.1. Evaluation der Baseline Vergleiche

Basierend auf dem Vorgehen (siehe 5.1) wird mit den Baseline Vergleichen begonnen. Bei diesen handelt es sich um die Vergleiche, welche von nicht optimierten (baseline) Agenten durchgeführt worden sind (siehe Abbildung 5.11). Diese Agenten wurden in einem Trainingsverlauf, entsprechend der Beschreibung in Abschnitt 5.5.1, trainiert. Während dieses Prozesses wurden die Trainingsdaten erhoben, die im weiteren Verlauf graphisch dargestellt werden. Darauffolgend wurden die Testdaten in Testläufen ermittelt, welche im Folgenden tabellarisch ausgewertet werden.

Performance

Einleitend in die Baseline Vergleichsauswertung soll mit der Performance gestartet werden. Dabei ist in der Abbildung 7.1 die Performance bzw. der Apfeldurchschnitt der letzten 100 Epochs pro Epoch des ersten Baseline Vergleiches abgebildet.

Die DQN Agenten waren in den Vergleichen nicht in der Lage, eine durchschnittliche Apfelsammelrate von 30 Äpfeln pro Spiel zu erreichen. Eine vermutliche Erklärung warum die Agenten des DQN Algorithmus keine guten Leistungen erzielen konnte, liegt in der Wahl von Zufallsaktionen. Die Komplexität des Spiels Snake steigt gegen Ende immer mehr an, da die Snake, mit zunehmenden Spielfortschritt, keine nicht

zielführenden Schritte mehr gehen darf. In einer beengten Spielsituation könnte jeder falsche Schritt zum Tod führen, wobei durch das zufällige Wählen von Aktionen falsche Schritte unternommen würden. Dadurch, dass das Spiel immer vorzeitig beendet würde, könnte der DQN Algorithmus auch seine Leistung nicht weiter steigern, weil ihm die Daten zum Lernen fehlten.

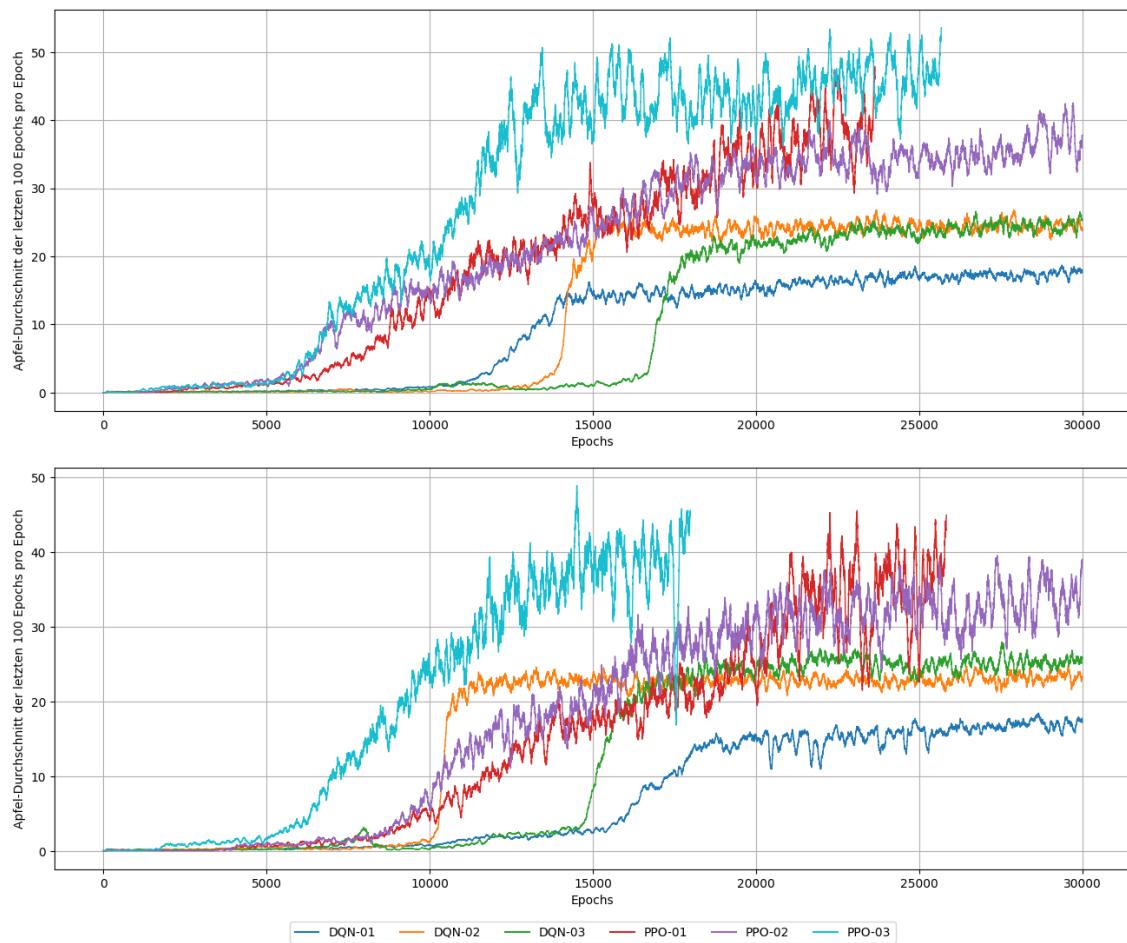


Abbildung 7.1.: Baseline Vergleich eins (oben) und zwei (unten) der Performance

Besonders die Leistungen der PPO Agenten PPO-03 und PPO-01 konnten, in den Vergleichen, überzeugen. Diese beiden Agenten waren ebenfalls die einzigen, welche es geschafft haben den Trainingsverlauf vorzeitig, mit einer Siegrate von 60%, abzubrechen. Der PPO-03 kann dabei besonders mit seinem schnellen Lernerfolg punkten, wohingegen der PPO-01 mit seiner Stetigkeit überzeugen kann.

Agent	Apfel-durchschnitt-BV-1	Apfel-durchschnitt-BV-2	Varianz-BV-1	Varianz-BV-2
DQN-01	18.493	17.069	15.817	19.526
DQN-02	27.876	24.969	44.968	30.306
DQN-03	25.410	25.490	45.609	46.872

PPO-01	49.854	42.201	614.484	794.192
PPO-02	41.002	36.463	171.257	369.649
PPO-03	53.975	49.084	302.203	565.363

Tabelle 7.1.: Testdatenauswertung der Performance

Auch die Auswertung der Testdaten (siehe Tabelle 7.1) zeigt den, sich in den Trainingsdaten abzeichnenden, Trend. So erbringt der PPO-03 die beste und der PPO-01 die zweitbeste Leistung. Die Varianz der Agenten PPO-03 und PPO-01 zeigt jedoch, dass die Leistungen noch verbesserbar sind.

Aufgrund der erzielten Performance dieser PPO Agenten ist daher festzustellen, dass diese beiden (PPO-03 und PPO-01) das Evaluationskriterium der Performance am besten erfüllt haben. Sie sind daher die Sieger im Evaluationskriterium der Performance.

Siegrate

Ähnlich, wie mit der Performance, sieht es auch mit dem Evaluationskriterium der Siegrate aus (siehe Abbildung 7.2).

Die PPO Agenten PPO-03 und PPO-01 besitzen die besten Siegraten während des Trainings (siehe Abbildung 7.2). Die DQN Agenten sind nicht in der Lage gewesen, Siege zu erreichen und fallen daher aus der Betrachtung heraus.

Der PPO-02 zeigt eine geringe Siegrate, trotz ähnlicher Performances zu den Agenten PPO-01 und PPO-03 (siehe Abbildung 7.1).

Agent	Siegraten-durchschnitt-BV-1	Siegraten-durchschnitt-BV-2
PPO-01	0.7654	0.626
PPO-02	0.099	0.086
PPO-03	0.668	0.662

Tabelle 7.2.: Testdatenauswertung der Baseline Siegrate

Auch die Testdaten in der Tabelle 7.2, zeigt, dass die PPO Agenten PPO-01 und PPO-03 die Sieger sind, wobei dieses Mal der PPO-01 bessere Resultate in den Testdaten erzielen konnte als PPO-03. Es ist daher festzuhalten, dass der PPO-01 und PPO-03 die Sieger im Evaluationskriterium der Siegrate sind .

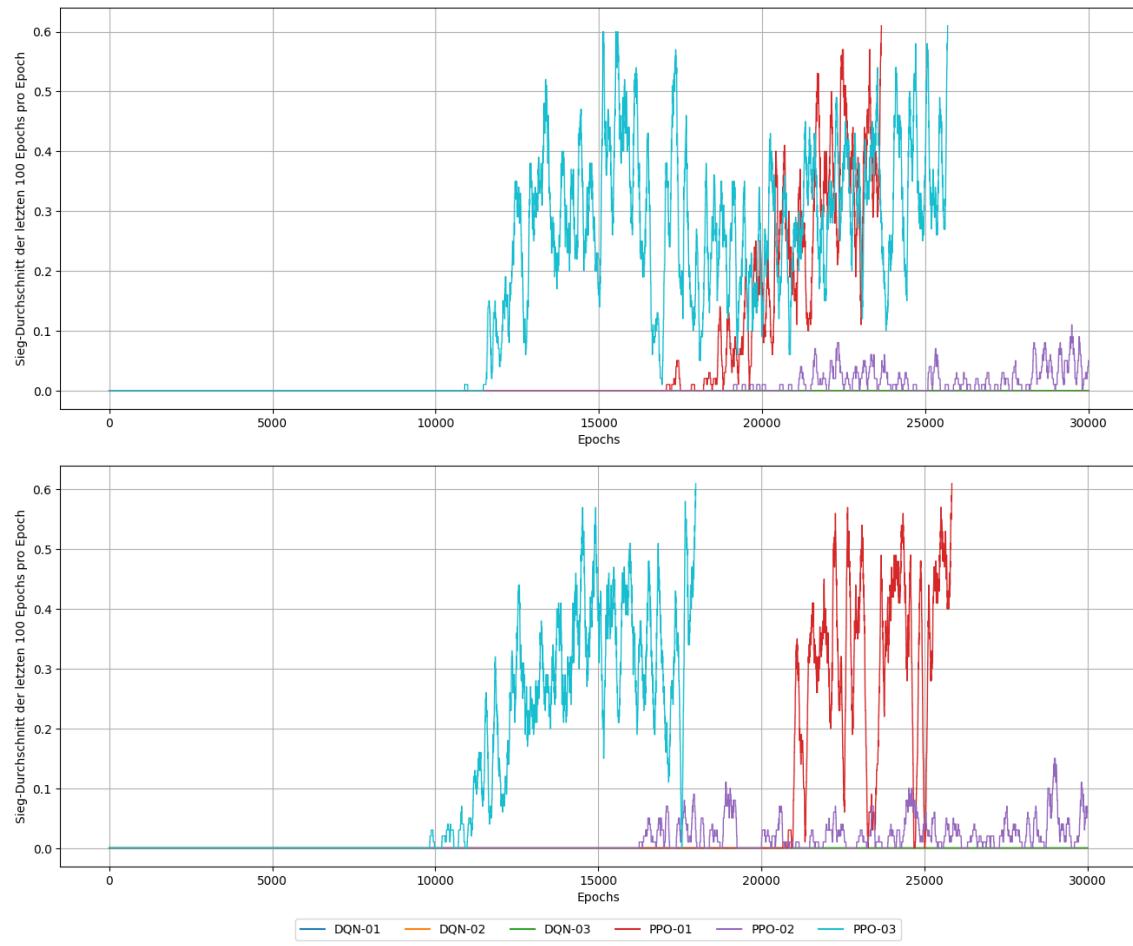


Abbildung 7.2.: Baseline Vergleich eins (oben) und zwei (unten) der Siegrate

Effizienz

Agent	Effizienz-durchschnitt-BV-1	Effizienz-durchschnitt-BV-2
DQN-01	0.144	0.154
DQN-02	0.132	0.138
DQN-03	0.129	0.133
PPO-01	0.077	0.069
PPO-02	0.114	0.090
PPO-03	0.097	0.075

Tabelle 7.3.: Testdatenauswertung der Baseline Effizienz

Bei der Effizienz treten interessante Effekte auf. Wie in der Abbildung 7.3 zu erkennen ist, sind die DQN Agenten DQN-01 und DQN-02 diejenigen, welche die beste Effizienz-Leistung erbringen. Die Effizienz ist dabei jedoch definiert als Äpfel pro

Schritt. Die Effizienzbestimmung ist jedoch nicht abhängig von der erbrachten Performance. Darum bieten die Agenten DQN-01 und DQN-02 die besten Effizienzen, bei niedrigeren Performances. Denkbar wären diese in Einsatzgebieten mit wenigen Zielen aber großen Distanzen, sodass der Effizienzcharakter des Agenten hilft Kraftstoff oder Energie, am Beispiel der unbemannter Drohnen, zu sparen.

Die Tabelle 7.3 unterstützt die Trainingsergebnisse. In den Testläufen hatte der DQN-02 Agent eine, im Mittel, geringfügig bessere Effizienz als der DQN-03. Darum sind die DQN Agenten DQN-01 und DQN-02 die Sieger dieses Effizienz-Vergleiches auf reiner Basis der Effizienz.

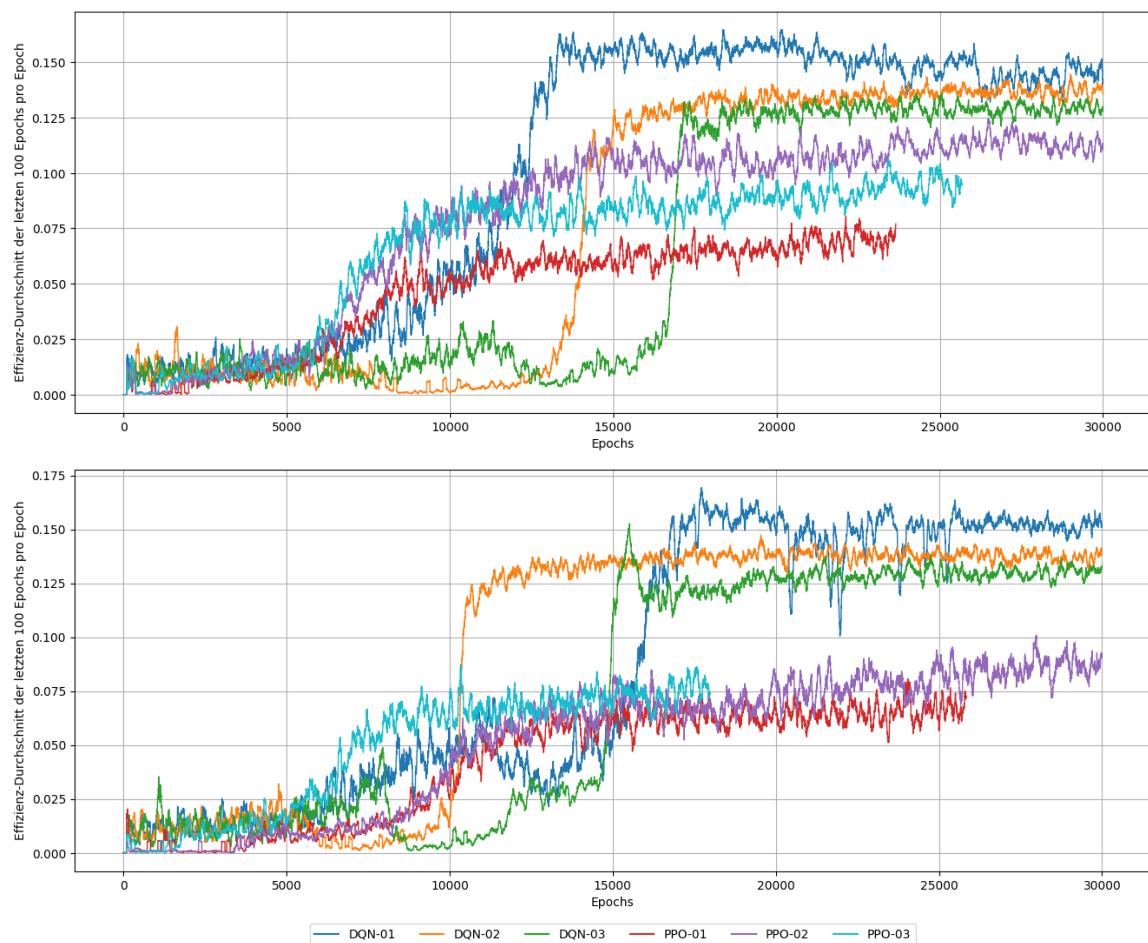


Abbildung 7.3.: Baseline Vergleich eins (oben) und zwei (unten) der Effizienz

Robustheit

Die Robustheit stellt ein besonderes Evaluationskriterium dar, denn sie wird nur aus Testdaten bestimmt. Es existieren daher keine Trainingsgrafiken. Die Robustheit ist definiert als die Performance auf unbekannten Spielfeldern, mit variierenden Größen von (6x6) zu (10x10). Dabei treten alle Spielfeldgrößen gleichverteilt häufig auf.

Agent / Apfeldurchschnitt auf Spielfeld mit der Größe	(6x6)	(7x7)	(8x8)	(9x9)	(10x10)
DQN-01	14.514	16.552	18.656	20.428	22.666
DQN-02	20.177	24.337	27.986	31.660	34.304
DQN-03	18.082	21.896	25.335	28.052	30.002
PPO-01	25.143	24.081	50.580	41.568	41.302
PPO-02	27.149	34.946	41.025	41.561	39.698
PPO-03	30.217	36.844	53.107	53.503	51.559

Tabelle 7.4.: Testdatenauswertung der Robustheit für Baseline Vergleich 1

Agent / Apfeldurchschnitt auf Spielfeld mit der Größe	(6x6)	(7x7)	(8x8)	(9x9)	(10x10)
DQN-01	13.259	15.291	17.122	18.734	20.336
DQN-02	18.426	21.958	25.007	27.854	30.445
DQN-03	18.595	22.271	25.245	27.902	30.244
PPO-01	19.793	21.653	43.447	36.957	65.815
PPO-02	22.422	29.334	37.193	41.107	41.097
PPO-03	24.471	29.740	49.447	50.309	69.691

Tabelle 7.5.: Testdatenauswertung der Robustheit für Baseline Vergleich 2

Wie bei den DQN-Agenten zu beobachten ist, steigt die Anzahl der gesammelten Äpfel ungefähr linear zur nächstgrößeren Spielfeldgröße. Dies zeugt jedoch von einer schlechten Robustheit, da die Komplexität des Spiels Snake mit zunehmender Spielfeldgröße nicht linear sondern exponentiell zunimmt, aufgrund der Ausdehnung des Möglichkeitsraumes. Die DQN-Agenten erreichen auf kleineren Spielfeldgrößen bessere Ergebnisse als auf der Standard Spielfeldgröße von 8x8. Sollte sich jedoch das Spielfeld vergrößern, so stagnieren ihre Leistungen. Bei den PPOs zeigt sich ein ähnliches Bild. Ihre Leistungen steigen ebenfalls nicht mit dem sich vergrößernden Spielfeld an. Jedoch waren der PPO-03 und der PPO-01 in der Lage die besten Ergebnisse in der unbekannten Gebieten zu erzielen. Im Baseline Run 2 war es dem PPO-03 sogar möglich, ein deutlich besseres Ergebnis zu erzielen, als mit der Standard Spielfeldgröße (8, 8).

Daher sind die Agenten PPO-03 und PPO-01 die Gewinner Agenten was das Evaluationskriterium der Robustheit betrifft.

7.1.2. Evaluation der Optimized Vergleiche

Nach der Durchführung der Baseline Vergleiche werden nun, entsprechend des Vorgehens (siehe 5.1), die Baseline Gewinner Agenten (siehe Abbildung 5.1) optimiert und dann untereinander verglichen. Dabei werden ebenfalls die Baseline Agenten Gewinner mit in den Vergleich eingebunden. Die Ergebnisse dieser Vergleiche finden sich in den Folgenden Abschnitten.

Performance

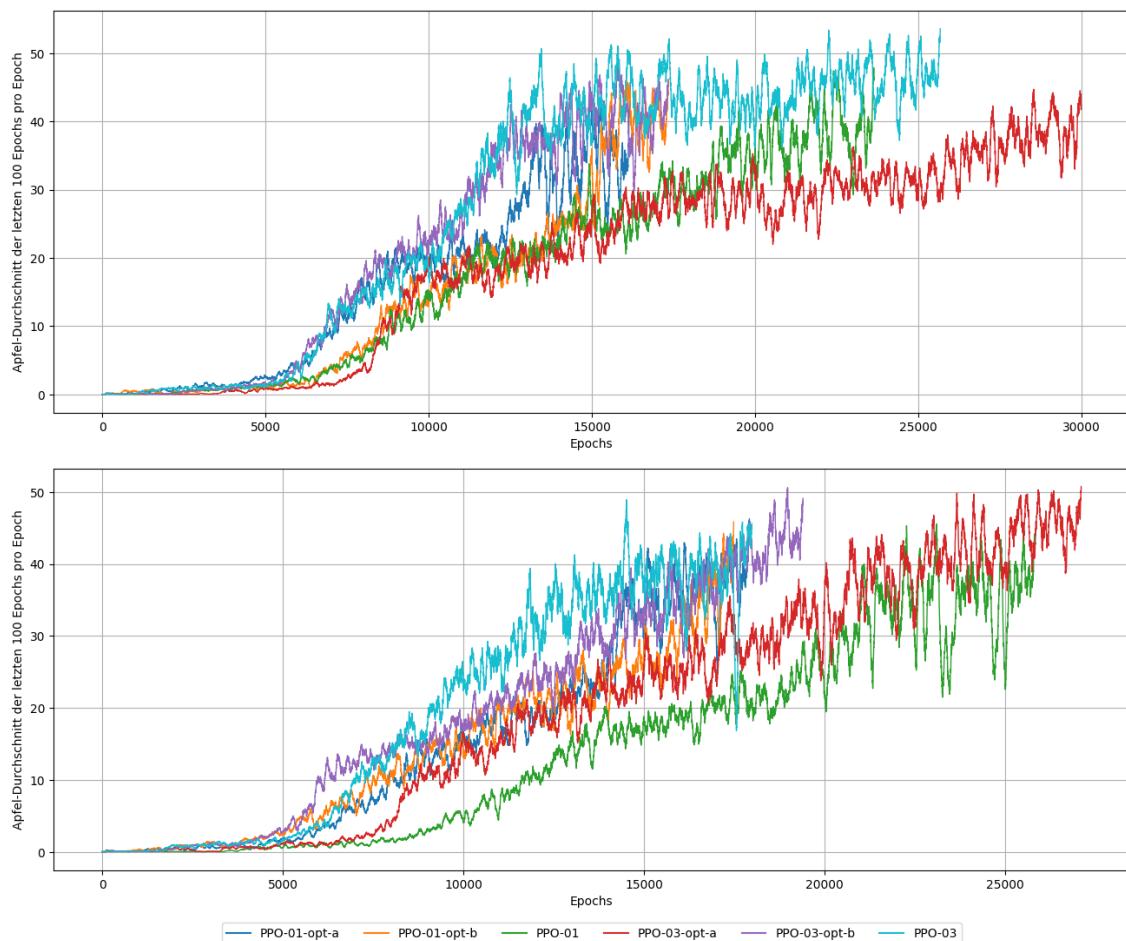


Abbildung 7.4.: Baseline Vergleich eins (oben) und zwei (unten) der Effizienz

Wie in Abbildung 7.4 zu erkennen ist, konnten alle Agenten, mit Ausnahme vom PPO-03-opt-a im OV-1 (Optimized Vergleich 1), den Lernprozess vorzeitig beenden. Auffällig ist, dass die Agenten PPO-01-opt-a (dunkelblau), PPO-01-opt-b (orange) und der PPO-03-opt-b (lila) Agent ungefähr zeitgleich eine Siegrate von 60% erreichten, gefolgt vom den Baseline Agenten und PPO-03-opt-a.

Agent	Apfel-durchschnitt-BV-1	Apfel-durchschnitt-BV-2	Varianz-BV-1	Varianz-BV-2
PPO-01	49.854	42.201	614.484	794.192
PPO-03	53.975	49.084	302.203	565.363
PPO-01-opt-a	45.071	42.227	746.026	748.823
PPO-01-opt-b	47.745	47.907	687.583	680.407
PPO-03-opt-a	46.438	54.444	551.219	303.706
PPO-03-opt-b	49.630	50.581	607.097	530.553

Tabelle 7.6.: Testdatenauswertung der Performance

Aus den Testdaten (siehe Tabelle 7.6) geht hervor, dass sowohl PPO-03 als auch PPO-03-opt-a die besten Agenten dieses Vergleiches sind. Jedoch konnte der Baseline Agent PPO-03 die besseren Ergebnisse, im Mittel, vorweisen. Mit einer durchschnittlichen Performance von 51.5295 Äpfeln in den Testläufen, stellt der PPO-03 den optimalen Agenten für das Evaluationskriterium der Performance dar. Jedoch verrät die Varianz ein instabiles Sammelverhalten, sodass bei der Anwendung dieses Agenten davon auszugehen ist, dass dieser die gezeigte Leistung nicht konstant erbringen wird.

Siegrate

Agent	Siegraten-durchschnitt-BV-1	Siegraten-durchschnitt-BV-2
PPO-01	0,765	0,626
PPO-03	0,668	0,662
PPO-01-opt-a	0,681	0,611
PPO-01-opt-b	0,732	0,733
PPO-03-opt-a	0,478	0,721
PPO-03-opt-b	0,741	0,727

Tabelle 7.7.: Testdatenauswertung der Optimized Siegraten

Bei der Siegrate konnten fast immer alle Agenten sehr guten Ergebnissen erzielen. Einzig PPO-03-opt-a lag im ersten Optimized Vergleich unter der erwarteten Siegrate von 60%, wie in Abbildung 7.5 zu erkennen ist.

Wie die Auswertung der Testdaten (siehe Tabelle 7.7) aufzeigt, liefern die Agenten PPO-01-opt-b und PPO-03-opt-b sehr gute Ergebnisse. Jedoch gewinnt der PPO-

03-opt-b diesen Vergleich mit einer durchschnittlichen Siegrate von 0,73405, dicht gefolgt vom PPO-01-opt-b mit 0,7325. Insgesamt ist dennoch zu bemerken, dass dieser kleine Abstand in der Siegrate auch durch Zufallseffekte zu erklären ist.

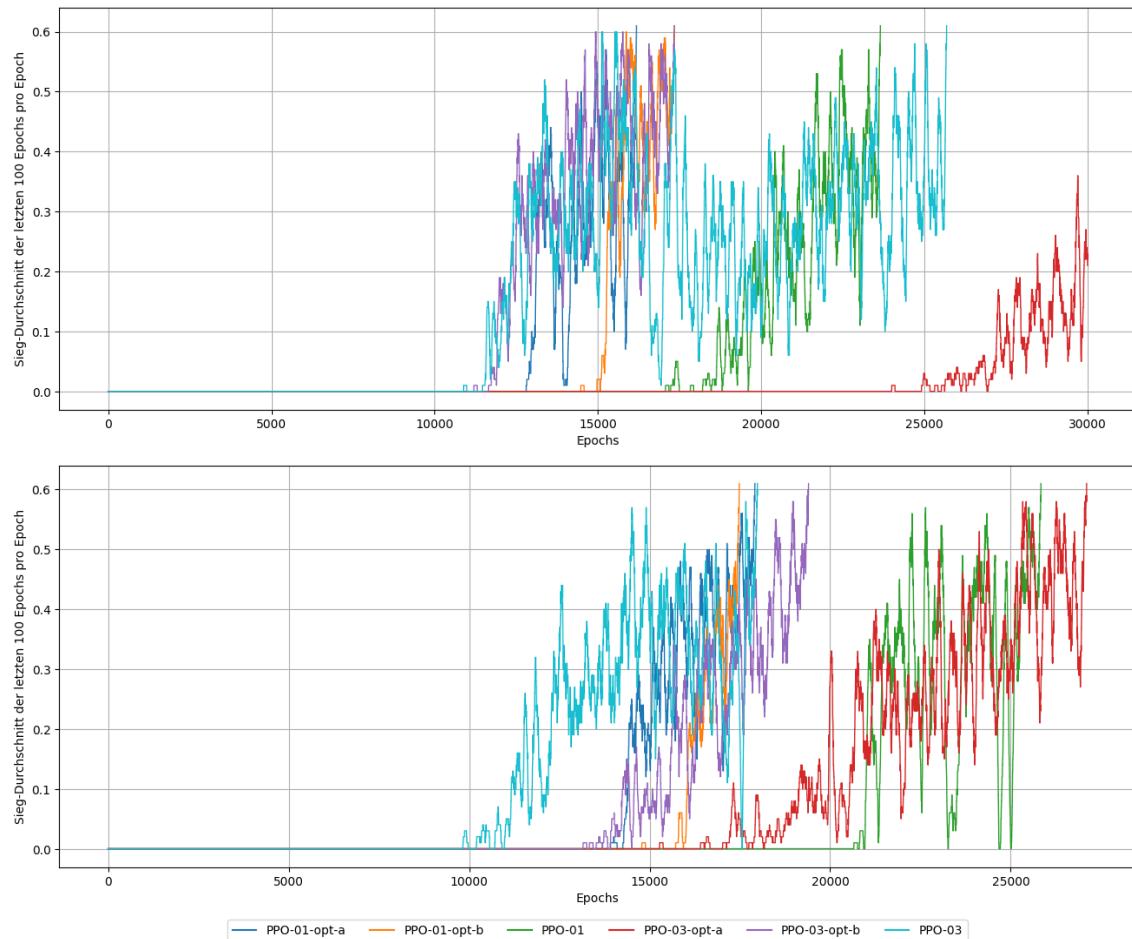


Abbildung 7.5.: Optimized Vergleich eins (oben) und zwei (unten) der Siegrate

Effizienz

folgt im selben Stil.

Robustheit

Nach den Testauswertungen (siehe Tabelle 7.8 und 7.9) hat sich der PPO-01-opt-b als der robusteste Agent dargestellt. Von allen Agenten weist er die kontinuierlichste Performance vor. Besonders auf größeren Spielfeldern zeigt er eine gute Leistung und damit eine solide Robustheit.

Es ist daher festzuhalten, dass der PPO-01-opt-b der Sieger der Vergleiche für das Evaluationskriterium der Robustheit ist.

Agent / Apfeldurchschnitt auf Spielfeld mit der Größe	(6x6)	(7x7)	(8x8)	(9x9)	(10x10)

PPO-01	25.143	24.081	50.580	41.568	41.302
PPO-03	30.217	36.844	53.107	53.503	51.559
PPO-01-rgs-opt-a	20.823	23.032	44.746	39.912	45.084
PPO-01-rgs-opt-b	26.135	31.067	52.574	56.541	80.265
PPO-03-rgs-opt-a	25.218	34.149	47.113	51.403	56.545
PPO-03-rgs-opt-b	24.966	30.395	47.946	57.580	74.758

Tabelle 7.8.: Testdatenauswertung der Robustheit für Optimized Vergleich 1

Agent / Apfeldurchschnitt auf Spielfeld mit der Größe	(6x6)	(7x7)	(8x8)	(9x9)	(10x10)
PPO-01	19.793	21.653	43.447	36.957	65.815
PPO-03	24.471	29.740	49.447	50.309	69.691
PPO-01-rgs-opt-a	22.534	24.358	42.942	42.223	39.528
PPO-01-rgs-opt-b	23.666	26.969	47.885	40.227	77.224
PPO-03-rgs-opt-a	29.808	35.390	54.276	51.236	59.522
PPO-03-rgs-opt-b	25.422	32.201	52.448	46.262	47.342

Tabelle 7.9.: Testdatenauswertung der Robustheit für Optimized Vergleich 2

7.2. Anforderungsevaluation

Zu Beginn sollen die Anforderungen an das Environment evaluiert werden. Danach folgen die Evaluationen der Anforderungen Agenten und der Datenerhebung. Zum Schluss sollen evaluiert werden, ob alle Anforderungen bezüglich der Statistiken und der Evaluation selbst erfüllt worden sind.

7.2.1. Anforderungsevaluation der Environment

Die Hauptanforderung an das Env besagt, dass das Spiel Snake implementiert werden soll. Im Rahmen dieser Ausarbeitung wurde das Spiel Snake nach der Beschreibung in 2.1 implementiert. Diese Anforderung kann daher als erfüllt angesehen werden. Eine Darstellung der Implementierung findet sich im Abschnitt des Konzepts (siehe 5.2) und in der Implementierung (siehe 6.1).

Ebenfalls wurde die Anforderung der Standardisierten Schnittstelle (siehe 3.1.1), welche zu einer Normung und damit zu einer einfacheren Benutzung des Environment führen soll, erfüllt. Wie in den Abschnitten 5.2.2 und 6.5 zu sehen ist, werden Aktionen, durch die step Methoden entgegengenommen, Rewards und Observatio-

nen zurückgegeben. Neben der step Methode wird die Observation auch noch durch die Schnittstellenmethode reset geliefert. Diese beiden Methoden spannen die standardisierte Schnittstelle auf.

Auch sind die funktionalen Anforderungen, welche den geregelten Ablauf im Environment garantieren, beachtet worden. So wird die Aktionsausführung (siehe 3.1.2) durch die action Methode in der SnakeGame Klasse durchgeführt, welche von der Schnittstellenmethode step aufgerufen wird. Die reset und render Anforderungen (siehe 3.1.2 und 3.1.2) werden durch die gleichnamigen Schnittstellenmethode abgedeckt (siehe 5.2.1 und 6.1).

7.2.2. Anforderungsevaluation der Agenten

Der nächste großer Anforderungsbereich behandelt die Agenten. Diese sollen funktionalen Anforderungen entsprechen, damit sie funktionsfähig und genormt sind. Zu diesem Zweck wurden die Anforderungen der Aktionsbestimmung und des Lernens aufgestellt. Diese stellen die grundlegenden Funktionen der Agenten dar. Wie im Konzept dargestellt ist, wurde sowohl der DQN als auch der PPO Agent mit einer Aktionsauswahlmethode (siehe ?? und 6.3.3) und einer learn Methode (siehe 6.3.3 und 6.4.4) ausgestattet. Diese implementieren das geforderte Verhalten aus den Anforderungen (siehe 3.2.1).

Neben den funktionalen, existieren noch zwei weitere Anforderungen. Zu diesen gehört die Diversität der Algorithmen (siehe 3.2.3).

Diese fordert den Vergleich verschiedener Algorithmen und dabei insbesondere des PPO und DQN Algorithmus. Mit dieser Forderung kann die entwickelte Methodik (siehe 5.1) besser bewertet werden. Diese Anforderung kann ebenfalls als erfüllt angesehen werden, da sowohl ein DQN (siehe 6.3.3) als auch ein PPO Agent (siehe 6.4.4) implementiert wurden.

Die andere der zwei erwähnten Anforderungen behandelt die Parametrisierung der Agenten. Das System soll mehrere Agenten gleichen Algorithmus erstellen können, welche sich jedoch durch die Hyperparameter unterscheiden sollen. Wie auch in der Evaluation der Ergebnisse (siehe 7.1) und in dem Konzept (siehe 5.3.4) zur erkennen ist, werden mehrere Agenten des gleichen Algorithmus miteinander verglichen.

7.2.3. Anforderungsevaluation an die Datenerhebung

Auch an die Datenerhebung wurden einige Anforderungen im Rahmen dieser Ausarbeitung gestellt. Zu diesen gehört die Forderung, dass die zu erhebenden Daten

mehrfach erhoben werden sollen (siehe 3.3.1). Dies soll die Validität der statistischen Untersuchung steigern. Wie im Abschnitt der Datenerhebung (siehe 5.5.1) und in der Evaluation der Ergebnisse zu sehen ist, werden die auszuwertenden Daten doppelt erhoben. Daher sind aus immer zwei Statistiken zu einem Evaluationskriterium zu sehen.

Daraus lässt sich ebenfalls schließen, dass die Daten für die Statistiken gespeichert werden. Damit wird die Anforderung der Datenspeicherung erfüllt (siehe 3.3.2). Dies wurde darüber hinaus in dem Abschnitt der Datenerhebung des Konzepts (siehe 5.5.1) und in der Implementierung (siehe 6.5) dargestellt.

7.2.4. Anforderungen an die Statistiken

Insgesamt sind die Anforderungen zu den Statistiken (siehe 3.4) darauf ausgelegt, dass mit dem implementierten System Statistiken entsprechend der Evaluationskriterien generiert werden können. Ein solche Funktionalität wurde implementiert, wie die Abschnitte 5.5 und 6.7 beweisen. Auch die Grafiken in der Evaluation der Ergebnisse (siehe 7.1) bestätigen dies.

7.2.5. Anforderungen an die Evaluation

In den Anforderungen der Evaluation war gefordert, dass die Evaluationskriterien Performance, Effizienz Robustheit und Siegrate untersucht werden. Wie im Abschnitt Ergebnisevaluation der Vergleiche 7.1 zu erkennen ist, wurde genau dies durchgeführt. Mithilfe dieser Evaluationskriterien konnte, auf Grundlage der erhobenen Statistiken, ein optimaler Agent für jedes Kriterium ausgewählt werden.

Kapitel 8

Fazit

Im Fazit sollen die Ergebnisse der Vergleiche (siehe 7.1) zusammengefasst und komprimiert dargestellt werden. Des Weiteren wird beurteilt, ob die Forschungsfrage (siehe 1.2) durch die Ausarbeitung erfüllt wurde. Anschließend wird ein Ausblick auf weitere Schritte in diesem Forschungsgebiet gegeben.

8.1. Bewertung der Forschungsfrage

In diesem Abschnitt soll die Beantwortung der eingangs gestellte Forschungsfrage: „Wie kann an einem Beispiel des Spiels Snake für eine nicht-triviale gering-dimensionale Umgebung ein möglichst optimaler RL Agent ermittelt werden?“ behandelt werden. Da das Wort „optimal“ einen großen Interpretationsspielraum zulässt, wurde sich auf vier Evaluationskriterien konzentriert. Zu diesen zählen die Performance, Effizienz, Siegrate und die Robustheit. Für jedes dieser Kriterien wurde der optimale Agent bestimmt (siehe 7.1). Die Forschungsfrage wurde dabei mithilfe einer, im Rahmen dieser Ausarbeitung erarbeiteten, Methodik beantwortet (siehe 5.1). In dieser wurden die vordefinierte Agenten erst mehreren Baseline Vergleichen unterzogen. Bei diesen wurden den Agenten keine weiteren Parameter wie z.B. eine neue Reward Funktion beim Lernen oder ein Lernraten Scheduling hinzugefügt.

Zu diesem Zweck wurde ein Konzept erarbeitet (siehe 5), welches als Grundlage zur Beantwortung der Forschungsfrage dienen. Dieses wurde im weiteren Verlauf, unter anderem in der Implementierung (siehe 6) umgesetzt.

Die Baseline Agenten wurden auf Basis der Evaluationskriterien verglichen und die zwei besten (Baseline Winner Agenten) wurden ausgewählt. Für die Baseline Vergleiche waren dies der PPO-03 und PPO-01 für die Performance, der DQN-01 und DQN-02 für die Effizienz, der PPO-01 und PPO-03 für die Siegrate und der PPO-03 und PPO-01 für die Robustheit.

Im Folgenden traten die Agenten gegen ihrer optimierten Varianten an, entspre-

chend des Vorgehens (siehe 5.1). Aus diesen Optimized Vergleichen ergaben sich die folgenden Sieger: Der PPO-03 ist der optimale Agent für das Kriterium der Performance, der (kommt noch) ist der optimale Agent für die Effizienz, der PPO-03-opt-b ist der optimale Agent für die Siegrate und der PPO-01-opt-b ist der optimale Agent für die Robustheit.

Aufgrund dieser konkreten Ergebnisse kann die Forschungsfrage, für die aufgestellten Evaluationskriterien, als beantwortet angesehen werden.

8.2. Ausblick

Nach der Durchführung der Vergleiche, welche zur Bestimmung des optimalen Agenten für jedes Evaluationskriterium geführt haben, ergeben sich weiterführende Fragestellungen.

Die DQN Agenten konnten, wie in der Evaluation dargestellt, keine guten Ergebnisse, mit Ausnahme in der Evaluation der Effizienz, erzielen. Die Vermutung, dass die Wahl von Zufallsaktionen dazu führt, könnte in einer anschließenden Ausarbeitung weiter untersucht werden.

Aber auch bezüglich der Methodik lassen sich noch weitere Untersuchungen durchführen. So würde zusätzliche Vergleich von weiteren Agenten, die eventuell auch auf andere Algorithmen basieren, eine gute Erweiterung darstellen.

Möglicherweise könnten auch eine Änderung der Untersuchungsparameter zu veränderten Resultaten führen. Zu diesen Veränderungen könnten die Spielregeln des Spiel Snake, die Evaluationskriterien, die Optimierungen für die Agenten usw. gehören.

Denkbar wäre auch ein praktischer Einsatz der Methodik in RL-Anwendungen. Ein-gangs wurde die Quelle Chunxue u. a. (2019) erwähnt, da sie unbemannte Drohnen mit RL-Agenten fliegen lassen will. Die hier erhobenen Ergebnisse legen nahe, dass die Wahl eines PPO, was die Performance betrifft, die bessere Wahl wäre, um Drohnen zu steuern.

Denkbar wäre auch die Anwendung des Verfahrens auf Probleme, welche nicht in thematischen Zusammenhang mit dem Spiel Snake stehen, wie z.B. Finanzapplikationen oder Steuerungsprogramme für Roboter.

Abschließend lässt sich behaupten, dass die Möglichkeiten für weiterführende Forschung auf dem Gebiet der Auswahl von RL-Agenten, viele Möglichkeiten bietet und das diese Arbeit einen kleinen Beitrag zu diesem Forschungszweig beiträgt.

Literaturverzeichnis

[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>

[GLASSNER 1989] GLASSNER, Andrew S. (Hrsg.): *An Introduction to Ray Tracing*. GBR : Academic Press Ltd., 1989. – ISBN 0122861604

[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. 2013] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOGLOU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin A.: Playing Atari with Deep Reinforcement Learning. In: *CoRR* abs/1312.5602 (2013). – URL <http://arxiv.org/abs/1312.5602>
- [Mnih u. a. 2015] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; RUSU, Andrei A. ; VENESS, Joel ; BELLEMARE, Marc G. ; GRAVES, Alex ; RIEDMILLER, Martin ; FIDJELAND, Andreas K. ; OSTROVSKI, Georg ; PETERSEN, Stig ; BEATTIE, Charles ; SADIK, Amir ; ANTONOGLOU, Ioannis ; KING, Helen ; KUMARAN, Dharshan ; WIERSTRA, Daan ; LEGG, Shane ; HASSABIS, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Februar, Nr. 7540, S. 529–533. – URL <http://dx.doi.org/10.1038/nature14236>. – ISSN 00280836
- [Schulman 2017] SCHULMAN, John: *Deep RL Bootcamp Lecture 5: Natural Policy Gradients, TRPO, PPO*. <https://www.youtube.com/watch?v=xvRrgxcpaHY>. Oktober 2017
- [Schulman u. a. 2015] SCHULMAN, John ; LEVINE, Sergey ; MORITZ, Philipp ; JORDAN, Michael I. ; ABBEEL, Pieter: Trust Region Policy Optimization. In: *CoRR* abs/1502.05477 (2015). – URL <http://arxiv.org/abs/1502.05477>
- [Schulman u. a. 2017] SCHULMAN, John ; WOLSKI, Filip ; DHARIWAL, Prafulla ; RADFORD, Alec ; KLIMOV, Oleg: Proximal Policy Optimization Algorithms. In: *CoRR* abs/1707.06347 (2017). – URL <http://arxiv.org/abs/1707.06347>
- [Sutton und Barto 2018] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. – URL <http://incompleteideas.net/book/bookdraft2018jan1.pdf>
- [Wei u. a. 2018] WEI, Zhepei ; WANG, Di ; ZHANG, Ming ; TAN, Ah-Hwee ; MIAO, Chunyan ; ZHOU, You: Autonomous Agents in Snake Game via Deep Reinforcement Learning. In: *2018 IEEE International Conference on Agents (ICA)*, 2018, S. 20–25

Anhang A

Anhang

A.1. Backpropagation und das Gradientenverfahren

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

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

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

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

A.2. Tensoren

Tensoren beschreiben die grundlegenden Einheiten eines jeden DL-Frameworks. Aus der Sicht der Informatik handelt es sich bei ihnen um mehrdimensionale Arrays, welche jedoch kaum etwas mit der mathematischen Tensorrechnung bzw. Tensoralgebra gemein haben. (Lapan, 2020, S. 67) Genauer gesagt lassen sich Tensoren als, Anordnung von Zahlen in einem regelmäßigen Raster mit variabler Anzahl von Achsen. (Goodfellow u. a., 2018, S. 35)

Ein DL-Framework kann aus z.B. einem Numpy Array <https://numpy.org/> einen Tensor konstruieren, welche dann über für die Verwendung in einem NN nutzbar ist. Dieser könnte Daten oder die gewichte eines NN beinhalten. Der entstandene Tensor dient dabei als eine Wrapper, welcher die Informationen des Arrays teilt oder kopiert hat. Neben vielen Zusatzfunktionen ist keine so wichtig wie das Aufbauen eines Backward Graphs. Ein DL-Framework ist mit diesem Graph in der Lage, jede Veränderung des Tensors einzusehen, um darauf aufbauend Backpropagation durchzuführen. (Lapan, 2020, S. 72 ff.)

A.3. Convolution Neural Networks

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

A.3.1. Convolutional Layer

Convolutional Layers (Conv Layers) sind der zentrale Baustein von CNNs, da sie besonders für die Feature Detektion geeignet sind. Ihre Gewichte sind in einem Tensor gespeichert. In diesen befinden sich die sogenannten Kernels oder auch Filter genannt, welche zweidimensionale Arrays darstellen. Bei Initialisierung der Conv Layers werden die Ein- und Ausgabe Channel der Inputs bzw. Outputs angegeben, sodass entsprechende dieser Informationen die Kernels erstellt werden können.

Des Weiteren wird noch die Größe der Kernels übergeben, wobei häufig Größen von (3x3), (5x5), (7x7) oder (1x1) genutzt werden.

Die Ausgabe eines Output Channels berechnet sich aus der Addition aller Input Channel Feature Maps, welche durch die Verrechnung mit den Kernels (Convolutionale Prozedur) entstanden sind. Für jeden Output Channel existiert ein Input Channel viele Kernel mit, daher ergibt sich eine Gewichtsmatrix der Form (Output_Channel, Input_Channel, Kernel_Size[0], Kernel_Size[1]) (Goodfellow u. a., 2018, S. 369 ff.).

Die Funktionsweise der Convolutional 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 Eingabekernquadrat 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 hinzugefügt 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.

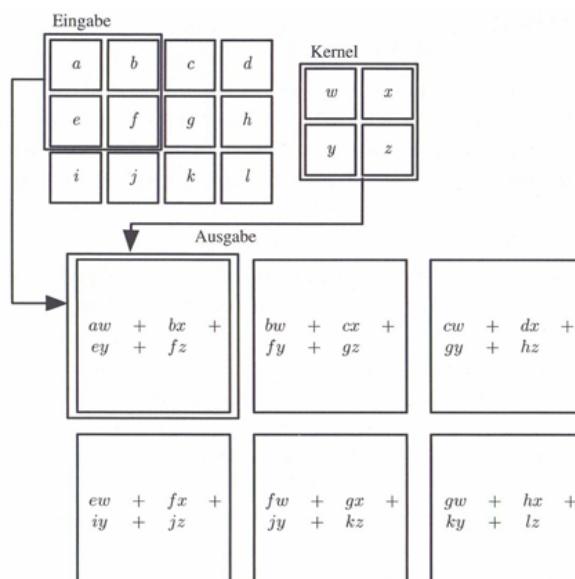


Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

A.3.2. Pooling Layer

Pooling beschreibt die Minimierung der Feature Maps, wie es bereits in ?? ange- sprochen 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.)

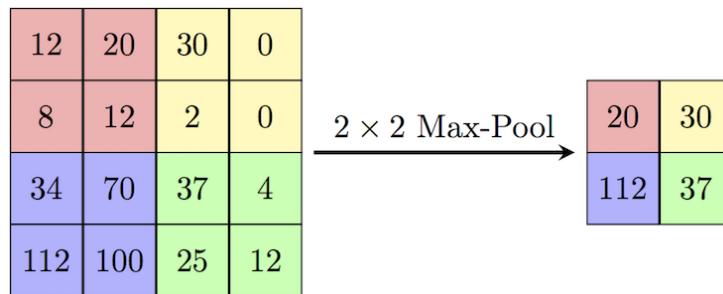


Abbildung A.2.: Darstellung des Max-Poolings. <https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png>

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:

$$y = xA^T + b \quad (\text{A.1})$$

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

Anhang B

Anhang zur Implementierung

B.1. Around-View

Zu Erstellung der around_view (AV) werden zunächst einige Konstanten gesetzt bzw. berechnet, wie z.B. die width oder c_s und c_h erstellt. Erstere entspricht dem Radius des Ausschnittes um den Kopf. Die letzteren sind Farbkonstanten, welche im Player definiert wurden (siehe ??).

Danach wird über jeden Eintrag, welcher im Ausschnitt der AV liegt, iteriert. Dieser Ausschnitt besitzt die Form eines Vierecks in dessen Mittelpunkt sich der Kopf der Snake befindet. Für jeden dieser Einträge wird überprüft, ob er einem der Merkmale aus 5.2 entspricht. Sollte ein Merkmal erkannt worden sein, wie z.B. die Position des Apfels, so wird diese in den dazugehörigen Channel eingepflegt.

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

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

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

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

```

    continue

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

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

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

return np.expand_dims(tmp_arr, axis=0)

```

B.2. Raytracing Distanzbestimmung

Ein wesentlicher Teil der scalar_obs (SO) besteht aus den Distanzen, welche von der create_distances Funktion generiert werden. Zu Beginn wird ein Null-Array der Länge 24 ($3 * 8 = 24$), in welches die Distanzen eingetragen werden und eine Liste namens grad_list erstellt. In dieser werden Faktoren gehalten, welche den Grad der Sucherlinie (siehe 5.4) angeben. Dabei entsprechen die Faktoren den Himmels- bzw. Nebenhimmelsrichtungen in der folgenden Reihenfolge ($0^\circ = N$, $45^\circ = NO$, $90^\circ = O$, $135^\circ = SO$, $180^\circ = S$, $225^\circ = SW$ und $270^\circ = W$, $315^\circ = NW$).

Danach wird über die drei zu suchenden Objekte (Wände, Snake-Glieder, Apfel) iteriert, wobei für jedes dieser Objekt die dist Unterfunktion aufgerufen wird. Die Rückgaben werden in das Array eingepflegt und dieses „sobald alle Distanzen bestimmt worden sind, zurückgegeben.“

```

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

```

Der dist Unterfunktion werden das Spielfeld (ground), die Position des Snake-Kopfes (p_pos) die zu suchenden Werte (wanted) und fac_0 und fac_1 übergeben. Diese stellen die oben genannten Faktoren dar, welcher die Ausrichtung der Suchlinien definiert. Nach der Initialisierung weiterer Hilfsvariablen und der Distanz (dist_),

wird der erste Index generiert, welcher dem ersten Feld auf dem Weg der Linie entspricht. Sollte der Eintrag an dieser Stelle dem gesuchtem Objekt entsprechen, so wird die Zahl zwei zurückgegeben. Andernfalls wird dist_ inkrementiert und der nächste höhere Index der auf dem Weg der Linie liegt generiert. Dann wird wieder so verfahren wie oben bereits erwähnt.

Sollte einer der Indexe jedoch das Spielfeld verlassen, so wird eine Distanz von null zurückgegeben.

Diese Prozedur endet entweder mit der Rückgabe einer Distanz zum Objekt, falls dieses gefunden wird, oder durch Rückgabe von null falls das Objekt nicht im Pfad der Linie liegt.

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

B.3. AV-Network

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

```
class AV_NET(nn.Module):
    def __init__(self):
        super(AV_NET, self).__init__()
        self.AV_NET = nn.Sequential(
            nn.Conv2d(in_channels=6, out_channels=8, kernel_size=(3, 3),
                     stride=(1, 1), padding=(1, 1)),
            nn.ReLU(),
            nn.Conv2d(in_channels=8, out_channels=8, kernel_size=(3, 3),
                     stride=(1, 1), padding=(1, 1)),
            nn.ReLU(),
            nn.ZeroPad2d((0, 1, 0, 1)),
```

```
nn.MaxPool2d(kernel_size=2, stride=2),  
nn.Flatten(),  
nn.Linear(392, 256),  
nn.ReLU(),  
nn.Linear(256, 128)  
)  
  
def forward(self, av):  
    return self.AV_NET(av)
```

Code Listing B.1: PyTorch Implementierung des AV-NET

Diese besteht aus zwei Convolutional (Conv2d), einem Max-Pooling und zwei Fully Connected Layers (Linear). Zwischen den Layers finden weitere Transformationen statt, welche in Abschnitt 5.3.1 erklärt wurden. Jedes NN-Element muss zwingend eine forward Methode implementieren, um die Propagierung der Input-Daten durch das Netzwerk zu steuern. Aus diesem Grund existiert eine forward Methode für diese Klasse, welche die AV (around_view) durch die dargestellten NN-Schichten und Funktionen propagierte. Das Ergebnis wird zurückgegeben.

Anhang C

Anleitung

Zur besseren Anwendbarkeit der Software, wurden die Files main_train und main_play erstellt. Mit diesen kann ein Benutzer die Trainings- und Spielroutine starten. Alternativ lässt sich dies auch durch die Verwendung von Python spezifischen Entwicklungsumgebungen durchführen.

Zum Start des Trainings muss main_train mit den folgenden Parametern gestartet werden. Dabei ist jedoch zu beachten, dass je nach Algorithmus-Art unterschiedliche Parameter übergeben werden müssen. Die Algorithmus-Art wird zu diesem Zweck als erstes Startargument übergeben.

C.1. PPO Train Startargumente

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

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

So könnte ein Start mittels Kommandozeile aussehen:

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

C.2. DQN Train Startargumente

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

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

So könnte ein Start mittels Kommandozeile aussehen:

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

C.3. Test Startargumente

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

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

So könnte ein Start mittels Kommandozeile aussehen:

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

Erklärung

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

Lorenz Mumm