



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, 1. Oktober 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	4
2.2.2. Funktionsweise	7
2.2.3. Arten von RL Verfahren	7
2.3. Proximal Policy Optimization	8
2.3.1. Actor Critic Modell	9
2.3.2. PPO Formelelemente	9
2.3.3. PPO Fehlerfunktion	10
2.3.4. PPO - Algorithmus	14
2.4. Deep Q-Network	15
2.4.1. DQN - Algorithmus	16
3. Anforderungen	18
3.1. Anforderungen an das Environment	18
3.1.1. Standardisierte Schnittstelle	18
3.1.2. Funktionalitäten	18
3.2. Anforderungen an die Agenten	19
3.2.1. Funktionalitäten	19
3.2.2. Parametrisierung	19
3.2.3. Diversität der RL Algorithmen	20
3.3. Anforderungen an die Datenerhebung	20
3.3.1. Mehrfache Datenerhebung	20

3.3.2. Datenspeicherung	20
3.4. Anforderungen an die Statistiken	21
3.5. Anforderungen an die Evaluation	21
4. Verwandte Arbeiten	22
4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning	22
4.1.1. Diskussion	23
4.2. UAV Autonomous Target Search Based on Deep Reinforcement Lear- ning in Complex Disaster Scene	25
4.2.1. Diskussion	26
4.3. Zusammenfassung	27
5. Konzept	28
5.1. Vorgehen	28
5.2. Environment	30
5.2.1. Spiellogik	30
5.2.2. Schnittstelle	35
5.3. Agenten	35
5.3.1. Netzstruktur	35
5.3.2. DQN	37
5.3.3. PPO	39
5.3.4. Vorstellung der zu untersuchenden Agenten	41
5.4. Optimierungen	43
5.4.1. Optimierung A - Joined Reward Function	43
5.4.2. Optimierung B - Anpassung der Lernrate	44
5.5. Datenerhebung und Verarbeitung	44
5.5.1. Datenerhebung	44
5.5.2. Datenverarbeitung und Erzeugung von Statistiken	45
6. Implementierung	47
6.1. Snake Environment	47
6.2. AV-Network	49
6.3. DQN	50
6.3.1. Q-Network	50
6.3.2. Memory	50
6.3.3. Agent	51
6.4. PPO	52
6.4.1. Actor und Critic	53
6.4.2. ActorCritic	53
6.4.3. Memory	55

6.4.4. Agent	55
6.5. Train Methoden	57
6.5.1. Test Methoden	58
6.6. Statistik	59
7. Evaluation	60
7.1. Ergebnisevaluation der Vergleiche	60
7.1.1. Evaluation der Baseline Vergleiche	60
7.1.2. Evaluation der Optimized Vergleiche	65
7.1.3. Bestimmung des optimalen Agenten	69
7.2. Anforderungsevaluation	70
7.2.1. Anforderungsevaluation der Environment	70
7.2.2. Anforderungsevaluation der Agenten	70
7.2.3. Anforderungsevaluation an die Datenerhebung	71
7.2.4. Anforderungen an die Statistiken	71
7.2.5. Anforderungen an die Evaluation	72
8. Fazit	73
8.1. Beantwortung der Forschungsfrage	73
8.2. Ausblick	74
Literaturverzeichnis	76
A. Anhang	78
A.1. Backpropagation und das Gradientenverfahren	78
A.2. Tensoren	79
A.3. Convolution Neural Networks	79
A.3.1. Convolutional Layer	79
A.3.2. Pooling Layer	81
A.3.3. Fully Connected Layer	81
B. Anhang zur Implementierung	82
B.1. Around-View	82
B.2. Raytracing Distanzbestimmung	83
B.3. AV-Network	84
C. Anleitung	86
C.1. PPO Train Startargumente	86
C.2. DQN Train Startargumente	87
C.3. Test Startargumente	88

Abbildungsverzeichnis

2.1. Game of Snake	3
2.2. Reinforcement Learning	7
5.1. Aktivitätsdiagramm des Vorgehens	29
5.2. Spielablauf	31
5.3. Observation	34
5.4. AV-Network	36
5.5. Actor-, Critic- und Q-Net-Tail	37
5.6. DQN Aktionsbestimmung	38
5.7. Agenten	41
6.1. Sequenzdiagramm	48
6.2. Ablaufdiagramm der action Methode	49
6.3. Ablaufdiagramm der Train Methode	57
7.1. Baseline Vergleich Performance	61
7.2. Baseline Vergleich Siegrate	62
7.3. Baseline Vergleich Robustheit	63
7.4. Baseline Vergleich der Effizienz für die Trainingsdaten	64
7.5. Baseline Vergleich der Effizienz für die Testdaten	65
7.6. Optimized Vergleich Performance	65
7.7. Optimized Vergleich Siegrate	67
7.8. Optimized Vergleich Siegrate	68
7.9. Optimized Vergleich Effizienz	69
7.10. Optimized Vergleich Effizienz	69
A.1. Darstellung Convolutional Computation	80
A.2. Darstellung MaxPooling	81

Tabellenverzeichnis

2.1. Formelemente des PPO Algorithmus	10
3.1. Zu erhebende Daten	20
3.2. Evaluationskriterien	21
5.1. Kodierung der Aktionen	31
5.2. Channel-Erklärung der Around_View (AV)	33
7.1. Testdatenauswertung der Performance	61
7.2. Testdatenauswertung der Baseline Siegrate	63
7.3. Testdatenauswertung der Performance	66
7.4. Testdatenauswertung der Optimized Sieg-Raten	67

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

In dieser Ausarbeitung wird das Generische Maskulinum verwendet. Dieses soll niemanden beleidigen und oder diskriminieren.

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 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, welche 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 größer werdende Aufmerksamkeit. Siege über die amtierenden Weltmeister in den Spielen GO oder Schach führten zu einer zunehmenden Beliebtheit des RL. Weiterhin wurde dieser Effekt durch 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), verstärkt. Heutzutage findet das RL auch in anderen Bereichen Anwendung, wie z.B. in der Finanzwelt, im autonomen Fahren usw.

Durch die jedoch große Menge an RL-Verfahren gerät man zunehmend in die problematische Situation, sich für einen diskreten RL Ansatz zu entscheiden. Weiter wird dieser Auswahlprozess erschwert durch die Tatsache, dass die einzelnen RL Agen-

ten jeweils untereinander große Unterschiede aufweisen. Auch existieren gelegentlich mehrere Ausprägungen eines RL Algorithmus, so wird z.B. der Deep Q-Network Algorithmus (DQN) durch den Double Deep Q-Network Algorithmus (DDQN) erweitert. Die Wahl des Agenten kann großen Einfluss auf die Performance und andere Bewertungskriterien haben (BOWEI MA, 2016), deshalb soll in dieser Ausarbeitung ein Vorgehen entwickelt werden, welches, basierend auf einem Vergleich, den optimalen Agenten für ein entsprechendes Problem bestimmt.

Eine potenzielle Umgebung, in welcher Agenten getestet und verglichen werden können, ist das Spiel Snake. Mit der Wahl dieses Spiels ist zusätzlich noch ein weiterer Mehrwert in Erscheinung getreten.

So interpretieren neue Forschungsansätze das Spiel Snake als ein dynamisches Pathfinding Problem. Mit dieser Art von Problemen sind auch unbemannte Drohne (UAV Drohnen) konfrontiert, welche beispielsweise Menschen in komplexen Katastrophensituationen, wie z.B. auf zerstörten Rohölbohrplattformen, finden und unterstützen sollen. Auch das Liefen von wichtigen Gütern in solche Gebiete zählt zu den Aufgaben dieser Drohnen. Mit der Forschung am Spiel Snake ist es möglich, zumindest einen kleinen Beitrag zu diesem Forschungsfeld zu leisten. (CHUNXUE U. A., 2019)

1.2. Zielsetzung

Basierend auf der Motivation ergibt sich folgende Fragestellung für diese Ausarbeitung:

Wie kann am Beispiel des Spiels Snake, für eine nichttriviale gering dimensionale Umgebung, ein möglichst optimaler RL Agent ermittelt werden?

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

Durch Abnahme des Entscheidungsfindungsprozesses müssen Forscher und Anwender von RL-Verfahren nicht mehr unreflektiert irgendeinen RL Agenten auswählen, sondern können auf Grundlage der Methodik und der daraus hervorgehenden Daten den passenden Agenten bestimmen.

Kapitel 2

Grundlagen

Im folgenden Kapitel soll das, zu benötigende, Wissen vermittelt werden. Dazu sollen verschiedene RL Algorithmen, wie auch grundlegende Informationen des RL selbst, thematisiert werden. Zuerst wird sich jedoch dem Spiel Snake beschäftigt.

2.1. Game of Snake

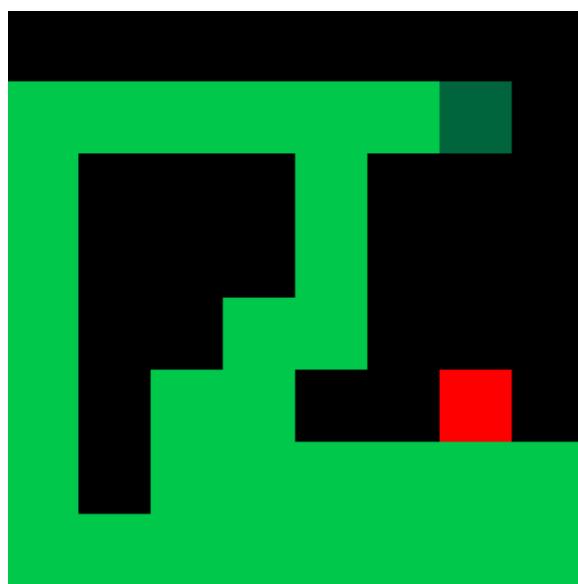


Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels, in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

Snake zählt zu den bekanntesten Computerspielen unserer Zeit. Es zeichnet sich durch sein simples und einfach zu verstehendes Spielprinzip aus. Das komplette Spielfeld wird durch eine zweidimensionale Ebene beschrieben, in welcher sich die Snake bewegt. Häufig wird diese als einfacher grüner Punkt (Quadrat) dargestellt. Dieser stellt den Kopf der Snake dar. Neben diesem befindet sich auf dem Spielfeld

auch noch der Apfel. Dieser wird häufig als roter Punkt (Quadrat) dargestellt (siehe Abbildung 2.1). Ziel der Snake ist es Äpfel zu fressen, indem der Kopf der Snake auf das Feld des Apfels läuft. Danach verschwindet der Apfel und ein neuer erscheint in einem zufälligen freien 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. Dieser ist es nicht erlaubt die Wände oder sich selbst zu berühren. Geschieht dies jedoch im Laufe des Spiels, so wird dieses sofort beendet. Diese Einschränkung führt zu einem Anstieg der Komplexität gegen Ende des Spiels. Ein Spiel gilt als gewonnen, wenn es der Snake gelungen ist das komplette Spielfeld auszufüllen.

Nach der Beleuchtung des Spiels Snake, folgen im Weiteren die Grundlagen des Reinforcement Learnings.

2.2. Reinforcement Learning

Das Reinforcement Learning (RL) ist einer der drei großen Teilbereiche, des Machine Learnings. Auf Menschen wirkt das RL, im Vergleich zu den anderen Disziplinen des Machine Learnings, am nachvollziehbarsten. Dies liegt an der Lernstrategie, welche Anwendung findet. Beim RL wird mithilfe eines „trial-and-error“ Verfahrens gelernt. Ein gutes Beispiel für eine solche Art des Lernens ist die Erziehung eines Kindes. Wenn das Kind etwas Gutes tut, dann wird es belohnt. Angetrieben von der Belohnung, versucht dieses das Verhalten fortzusetzen. Entsprechend wird das Kind bestraft, wenn es etwas Schlechtes tut, sodass schlechtere Verhaltensweisen weniger häufig zum Vorschein kommen, um Bestrafungen zu entgehen. (Sutton und Barto, 2018, S.1 ff.)

Beim RL funktioniert es ganz ähnlich, was auch der Grund dafür ist, dass viele der Aufgaben des RL dem menschlichen Arbeitsspektrum sehr nahe sind. Deshalb findet das RL auch immer mehr Anwendung, im Finanzhandel, in der Robotik, im autonomen Fahren usw. (Lapan, 2020, Kapitel 18)

2.2.1. Vokabular

Um ein tiefer gehendes Verständnis für das RL zu erhalten, müssen zuerst die Begrifflichkeiten erlernt und deren Bedeutung verstanden werden.

Agent

Im Zusammenhang mit dem RL, ist häufig die Rede von Agenten. Sie sind zentrale Instanzen, welche die RL Algorithmen in ein festes Objekt einbinden. Dabei werden

Methoden, Hyperparameter und 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 diesen Interaktionen zählen das Entgegennehmen von Observations und Rewards, wie auch das Übergeben von Aktionen. (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 gehören die Fähigkeiten Observations und Rewards zu liefern, sowie Aktionen zu verarbeiten. (Lapan, 2020; Sutton und Barto, 2018, S. 31 & S.2 ff.)

Action

Die Actions bzw. Aktionen sind einer der drei Datenübermittlungswege. Bei ihnen handelt es sich um Handlungen, welche im Env ausgeführt werden. Aktionen können z.B. Züge in Spielen, das Abbiegen im autonomen Fahren oder das Ausfüllen eines Antrages sein. Es wird ersichtlich, dass die Aktionen, welche ein RL Agent ausführen kann, prinzipiell nicht in der Komplexität beschränkt sind. Dennoch ist es gängige Praxis geworden, dass diese möglichst einfach seien sollen.

Im Environment wird zwischen diskreten und stetigen Aktionsraum unterschieden. Der diskrete Aktionsraum umfasst eine endliche Menge an sich gegenseitig ausschließenden Aktionen. Ein Beispiel dafür wäre das Gehen an einer T-Kreuzung. Der Agent kann entweder Links, Rechts oder zurück gehen. Anders verhält es sich bei einem stetigen Aktionsraum. Dieser zeichnet sich durch stetige Werte aus. Das Steuern eines Autos wäre hierfür beispielhaft. Es stellen sich Fragen, wie z.B. um wie viel Grad muss der Agent das Steuer drehen und um wie viel Prozent muss er bremsen, damit das Fahrzeug in der Spur bleibt? (Lapan, 2020, S. 31 f.)

Observation

Die Observation (Obs) bzw. die Beobachtung ist ein weiterer Datenübermittlungs weg, welche den Agenten mit dem Env verbindet. Bei dieser handelt es sich um einen oder mehrere Vektoren bzw. Matrizen, welche dabei den momentanen Zustand des Environments repräsentieren. (Sutton und Barto, 2018, S. 381)

Je nach Anwendungsbereich, fällt die Obs sehr unterschiedlich aus. In der Finanzwelt könnte diese z.B. die neusten Börsenkurse beinhalten oder in der Welt der Spiele könnten die aktuelle Punktezahl wiedergeben werden. (Lapan, 2020, S. 32)

Reward

Der Reward bzw. die Belohnung ist der letzte Datenübertragungsweg. Bei diesem handelt es sich um einen einfachen skalaren Wert, welche vom Env übermittelt wird. Dieser gibt an, wie gut bzw. schlecht eine ausgeführte Aktion 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.

State

Der State bzw. Zustand ist eine Widerspiegung der zum Zeitpunkt t vorherrschenden Situation im Env. Dieser wird von der Obs (Observation) repräsentiert. Häufig findet sich der Begriff des State in der Fachliteratur und in anderen Ausarbeitungen zu diesem Themengebiet. (Sutton und Barto, 2018, s. 381 ff.)

Policy

Informell lässt sich die Policy als eine Menge von 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 entspricht $\pi_t(a_t|s_t)$ der Wahrscheinlichkeit, dass die Aktion a_t im State s_t gewählt wird. (Sutton und Barto, 2018, S. 45 ff.)

Value

Die Values geben eine Einschätzung ab, wie gut oder schlecht ein State bzw. State-Action-Pair ist. Sie werden gewöhnlich mit einer Value Funktion V bestimmt, sodass beispielsweise $V(s)$ dem Wert des States s gleichkommt. Der Value entspricht dabei der diskontierten Summe aller noch zu erwartenden Rewards in der verbleibenden Spielepisode. Damit ist dieser ein Maß dafür, wie gut es für den Agenten ist, in diesen State zu wechseln.

Der sogenannte Q-Value $Q(s, a)$ gibt Aufschluss darüber, welche Aktion a im State s den größten Return (Summe aller diskontierten Rewards in der noch verbleibenden Spielepisode) erzielen wird.

Die Values werden ebenfalls unter einer Policy (Regelwerk des Agenten) bestimmt, daher folgt, dass für die Value Funktionen $V_\pi(s)$ und $Q_\pi(s, a)$, wobei π die Policy ist. (Sutton und Barto, 2018, S. 46)

2.2.2. Funktionsweise

Die folgende Beschreibung der Funktionsweise orientiert sich an der Abbildung 2.2.

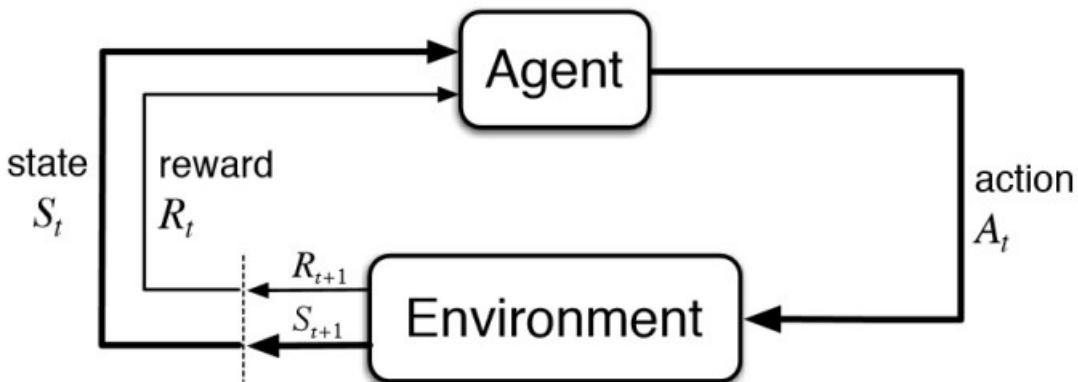


Abbildung 2.2.: Schematische Funktionsweise des Reinforcement Learning - Der Agent erhält einen state S_t und falls $t \neq 0$ einen reward R_t . Daraufhin wird vom Agenten eine action A_t ermittelt, welche im Env ausgeführt wird. Das Env übermittelt den neuen state S_{t+1} und reward R_{t+1} an den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

Zu Beginn wird dem Agenten vom Environment der initialer State $S_{t=0}$ übermittelt, welcher einen Entscheidungsfindungsprozess anstößt. Es wird eine Aktion A_t ermittelt, welche an das Env weiterleitet und dort ausgeführt wird. Das manipulierte Env befindet sich nun im neuen State S_{t+1} , welcher an den Agenten weitergeleitet wird. Des Weiteren bestimmt und sendet das Env noch einen Reward R_{t+1} . Mit dem neuen State S_{t+1} , kann der Agent wieder eine Aktion A_{t+1} bestimmen, welche ausgeführt wird. Daraufhin werden wieder der neue State S_{t+2} und Reward R_{t+2} ermittelt und übertragen usw. (Sutton und Barto, 2018, S. 37 ff.)

2.2.3. Arten von RL Verfahren

Nachdem das Basisvokabular erklärt wurde, soll nun ein tieferer Blick in die verschiedenen Arten der RL geworfen werden.

Alle RL Verfahren lassen sich, in Klassen einordnen, welche Aufschluss über verschiedene Details des Algorithmus geben. Es existieren natürlich viele Möglichkeiten, RL Verfahren zu klassifizieren, aber vorerst soll sich auf die folgenden zwei beschränkt werden.

Policy-Based und Value-Based Verfahren

Die Einordnung in policy-based und value-based Verfahren gibt Aufschluss über den Entscheidungsfindungsprozess des RL Verfahrens. Policy-Based Agenten versuchen

unmittelbar die Policy zu berechnen, umzusetzen und sie zu optimieren. Sie besitzen dafür meist ein eigenes NN, welches die Policy π für einen State s bestimmt. Gewöhnlich wird diese als eine Wahrscheinlichkeitsverteilung über alle Aktionen angegeben (siehe Unterunterabschnitt 2.2.1). Jede Aktion erhält damit einen Wert zwischen null und eins, welcher Aufschluss über die Qualität der Aktion im momentanen Zustand des Env liefert. (Lapan, 2020, S. 100)

Basierend auf dieser Wahrscheinlichkeitsverteilung π wird die nächste Aktion a bestimmt. Dabei wird nicht immer die optimale Aktion gewählt, um die Exploration voranzutreiben.

Die value-based Verfahren bestimmen ihre Aktionen nicht mit einer Wahrscheinlichkeitsverteilung. Die Policy wird indirekt durch das Bestimmen aller Values, (siehe Unterunterabschnitt 2.2.1) für alle Aktionen, ermittelt. Es wird daher immer die Aktion a gewählt, welche zu dem State s führt, der über den größten Value verfügt. (Lapan, 2020, S. 100)

On-Policy und Off-Policy Verfahren

Eine Klassifikation in on-policy und off-policy Verfahren gibt Aufschluss über den Zustand der Daten.

Off-Policy RL Verfahren sind in der Lage, von Daten zu lernen, welcher nicht unter der momentanen Policy (siehe Unterunterabschnitt 2.2.1) generiert wurden. Die Aktualität der Daten spielt daher keine Rolle für dieses RL Verfahren.

On-Policy RL Verfahren sind dagegen sehr wohl abhängig von der Aktualität der Daten, da sie versuchen die Policy indirekt oder direkt zu optimieren. Sollte ein on-policy Agent mit veralteten Daten trainiert werden, so würde dies die Policy des Agenten zerstören. (Lapan, 2020, S. 210 f.)

Nach diesen Einordnungen sollen nun die ausgewählten Algorithmen näher erklärt werden. Begonnen wird mit dem Proximal Policy Optimization Algorithmus (PPO).

2.3. Proximal Policy Optimization

Der Proximal Policy Optimization Algorithmus (PPO) wurde von dem Open-AI-Team entwickelt. Im Jahr 2017 erschien das gleichnamige Paper, welches den Aufbau und die Umsetzung des Algorithmus genauer erläutert. (Schulman u. a., 2017)

Agenten, welche auf dem PPO Algorithmus basieren (siehe Unterunterabschnitt 2.2.1), konnten, in der Vergangenheit, bereits durch gute Leistungen überzeugen.

Um einen groben Überblick des Algorithmus zu erhalten, werden zuerst, im nächsten Abschnitt, dessen Architektur und Vorläufer näher thematisiert.

2.3.1. Actor Critic Modell

Der PPO Algorithmus ist ein policy-based (siehe Unterunterabschnitt 2.2.3) RL Verfahren, welches, im Vergleich zu anderen Verfahren, einige Verbesserungen aufweist. So ist der PPO eine Weiterentwicklung normaler Policy Gradient Verfahrens und basiert intern auf dem sogenanntes Actor bzw. Policy Network und auf dem Critic bzw. Value Network. (Sutton und Barto, 2018, S. 273 f.)

Beide neuronalen Netzwerke können aus mehreren Schichten bestehen, jedoch sind Actor und Critic streng voneinander getrennt und besitzen mit einer Ausnahme keine gemeinsamen Parameter. Gelegentlich werden jedoch beiden Netzen (Actor bzw. Critic) noch ein weiteres Netz vorgeschoben. In diesem Ausnahmefall können Actor und Critic gemeinsame Parameter besitzen.

Das Actor Network ist für die Bestimmung der Policy zuständig. Anders als bei value-based (siehe Unterunterabschnitt 2.2.3) RL Verfahren, wird diese direkt bestimmt und kann auch direkt angepasst werden. Die Policy wird als eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen vom Actor Network zurückgegeben. (siehe Unterunterabschnitt 2.2.1)

Das Critic bzw. Value Network evaluiert die Aktionen, welche vom Actor Network bestimmt worden sind. Genauer gesagt, schätzt das Value Network den sogenannten Return bzw. die Discounted Sum of Rewards (siehe Unterunterabschnitt 2.3.3) zu einem Zeitpunkt t , basierend auf dem momentanen State s . Auf dieser Architektur wird ein PPO Agent aufgebaut.

Neben der Architektur, ist die Aktualisierungsprozedur ein weiterer wichtiger Bestandteil des Algorithmus. Diese wird daher im Folgenden näher betrachtet.

2.3.2. PPO Formelelemente

Das Herzstück eines jeden RL Verfahrens wird durch seine Aktualisierungsprozedur dargestellt, in welcher der Fehler (Loss) des Algorithmus bestimmt wird. Um die dem PPO zugrunde liegende Aktualisierungsprozedur besser zu verstehen, folge eine Erklärung ihrer einzelnen mathematischen Bestandteile. Diese basieren auf den PPO Paper (Schulman u. a., 2017).

Symbol	- 2.5cmErklärung
t	- 2.5cmZeitpunkt
$\hat{\mathbb{E}}[X]$	- 2.5cm $\hat{\mathbb{E}}[X]$ 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)
θ	- 2.5cmTheta beschreibt die Parameter, aus denen sich die Policy des PPO ergibt. Sie sind die Gewichte, welche das Policy Network definiert.

π_θ	- Die Policy bzw. Entscheidungsfindungsregeln sind eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen. Eine Aktion a wird auf Basis der Wahrscheinlichkeitsverteilung gewählt (siehe (Sutton und Barto, 2018, Summary of Notation S. xvi) und Unterabschnitt 2.2.1).
$L^{\text{CLIP}}(\theta)$	- $L^{\text{CLIP}}(\theta)$ bezeichnet den sogenannten Policy bzw. Actor Fehler, 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 Actor Networks benötigt.
$r_t(\theta)$	- $r_t(\theta)$ stellt den Quotienten zwischen alter Policy (nicht als Abhängigkeit angegeben, da sie nicht verändert wird) und aktueller Policy zum Zeitpunkt t dar. Daher auch Probability Ratio genannt.
$\hat{A}_t(s, a)$	- Erwartete Vorteil bzw. Nachteil einer Aktion a , welche im State s ausgeführt wurde.
clip	- Mathematische Funktion zur Beschränkung eines Eingabewertes. Clip setzt eine Ober- und Untergrenze fest. Sollte ein Wert, der dieser Funktion übergeben wird, sich nicht mehr innerhalb der 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	- Der Gamma-Wert bzw. Abzinsungsfaktor ist ein Hyperparameter, welcher 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 kleine Werte zu einem kurzfristigen Lernen führen (Sutton und Barto, 2018, S. 43 bzw. Summary of Notation S. xv).

Tabelle 2.1.: Formelelemente des PPO Algorithmus

2.3.3. PPO Fehlerfunktion

Nun, da die Grundlagen näher beleuchtet worden sind, ist das nächste Ziel, die Fehlerfunktion des PPO zu verstehen, damit diese im weiteren Verlauf angewendet werden kann. Der PPO Fehler wird durch folgende Funktion definiert (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 Fehlerfunktion aus dem Actor Fehler $L_t^{\text{CLIP}}(\theta)$, aus dem Critic Fehler L_t^{VF} und aus dem Entropy Fehler bzw. Bonus $S[\pi_\theta](s_t)$. Der Actor Fehler ist

dabei durch folgenden Term gegeben (Schulman u. a., 2017, S. 3):

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

Dieser wird im Weiteren kleinschrittig erklärt. Begonnen wird mit dem Advantage $\hat{A}_t(s, a)$, welcher sich aus dem Return und dem Baseline Estimate berechnet.

Return

Der Return R_t stellt die diskontierte Summe der Rewards in der vom Zeitpunkt t an verbleibenden Spieldiode dar. Die einzelnen Rewards werden dabei mit dem Discount Factor γ multipliziert (siehe Tabelle 2.1), 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 eher dazu tendiert Aktionen zu wählen, welche unmittelbar zu positiven Rewards führen. Entsprechend verhält es sich mit großen Werten für Gamma γ . (Sutton und Barto, 2018, S. 42 ff.) Für die Bestimmung des Advantage benötigt man neben dem Return jedoch auch noch den Baseline Estimate Wert.

Baseline Estimate

Der Baseline Estimate Wert $b(s_t)$ ist das Ergebnis der Value Funktion (siehe Unterabschnitt 2.2.1), welche durch das Critic Network (siehe Unterabschnitt 2.3.1) realisiert wird. Die Value Funktion schätzt den noch zu erwartenden Return R_t , vom aktuellen State s_t an. Da es sich hierbei um die Ausgabe eines NN handelt, wird in dieser immer eine Varianz bzw. ein Rauschen vorhanden sein. (Mnih u. a., 2016, Kapitel 3)

Mit diesem letzten Wert lässt sich nun der Advantage bestimmen.

Advantage

Der Advantage $\hat{A}_t(s, a)$ stellt einen Funktionsbestandteil des Actor Fehlers dar. Dieser wird durch die Subtraktion des Returns R_t mit dem Baseline Estimate Wert $b(s_t)$ bestimmt. Die folgende Formel (siehe Gleichung 2.3) ist eine zusammengefasste Version der Originalformel aus (Schulman u. a., 2017):

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

Der Advantage gibt an, um wie viel besser oder schlechter eine Aktion gewesen ist, basierend auf der Erwartung der Value Funktion des Critics. Es wird also die Frage beantwortet, ob eine gewählte Aktion a im Zustand s_t zum Zeitpunkt t besser oder schlechter als erwartet war. (Mnih u. a., 2016, Kapitel 3)

Um nun den ersten Surrogate Fehler, welcher im Weiteren noch thematisiert wird (siehe Unterunterabschnitt 2.3.3), zu bestimmen, wird noch die Probability Ratio benötigt.

Probability Ratio

Die Probability Ratio $r_t(\theta)$ ist der nächste Funktionsbestandteil des $L_t^{\text{CLIP}}(\theta)$ (siehe Gleichung 2.2). In normalen Policy Gradient Methoden bestehe ein Problem zwischen der effizienten Datennutzung und der Aktualisierung der Policy. Dieses Problem tritt z.B. im Zusammenhang mit dem Advantage Actor Critic (A2C) Algorithmus auf und reglementiert die effiziente Nutzung von Daten. So ist es dem A2C nur möglich von Daten zu lernen, welche on-policy (siehe Unterunterabschnitt 2.2.3) erzeugt wurden. Der PPO umgeht diese Problematik jedoch partiell durch das Importance-Sampling (IS, deutsch: Stichprobenentnahme nach Wichtigkeit). Bei der Fehlerfunktion des A2C Algorithmus (siehe Gleichung 2.4) wird deutlich,

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

dass die Daten für die Fehlerbestimmung nur unter der aktuellen Policy π_θ generiert wurden (on-policy) (Lapan, 2020, S. 591).

Schulman et al. ist es jedoch gelungen, diesen Ausdruck durch einen mathematisch äquivalenten zu ersetzen, welcher auf zwei Policies basieren kann. Eine davon stellt die aktuelle π_θ und die andere eine ältere $\pi_{\theta_{\text{old}}}$ dar.

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

Die Daten können nun mittels $\pi_{\theta_{\text{old}}}$, effizienter genutzt werden, da sie nun nicht mehr aus der aktuellen Policy stammen müssen. Dies erlaubt ein häufigeres Nutzen der Daten, ohne eine Zerstörung der Policy zu riskieren. (Schulman, 2017, Zeitpunkt: 9:25)

Surrogate Fehlerfunktion

Die Surrogate Fehlerfunktionen ergeben sich aus der Tatsache, dass die Fehlerfunktion des PPO nicht mit der gewöhnlich verwendeten logarithmierten Policy $\hat{\mathbb{E}}_t[\log_{\pi_\theta}(a_t|s_t)A_t]$ arbeitet, sondern mit dem Surrogate (Ersatz) der Probability Ratio $r_t(\theta)$ (siehe Unterunterabschnitt 2.3.3) zwischen alter und neuer Policy. (Schulman u. a., 2015)

Der Actor Fehler wird mithilfe zweier Surrogate Fehlerfunktionen bestimmt. Die

erste Fehlerfunktion $surr_1$ (siehe Gleichung 2.6)

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

stellt die normale TRPO Fehlerfunktion dar, ohne die durch den TRPO Algorithmus vorgesehene KL-Penalty. (Schulman u. a., 2017, S. 3 f.) Die alleinige Nutzung dieser Funktion hätte jedoch destruktiv große Policy Aktualisierungen zufolge. Aus diesem Grund haben John Schulman et al. eine zweite Surrogate Fehlerfunktion $surr_2$ (siehe Gleichung 2.7), dem PPO Algorithmus hinzugefügt.

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

Die einzige Veränderung dieser Funktion im Vergleich zur Ersten ist, dass der berechnete Fehler geclipped (abgegrenzt) wird. Sollte sich die Probability Ratio zu weit von eins entfernen, so wird $r_t(\theta)$ entsprechend auf $1 - \epsilon$ bzw. $1 + \epsilon$ begrenzt. Es wird sich daher in einer Trust Region bewegen, da man sich nie allzu weit von der ursprünglichen Policy entfernt. (Schulman u. a., 2015, 2017)

Zusammenfassung der PPO Fehlerfunktion

Der Actor Fehler lässt sich nun für jede Erfahrung, welche gesammelt wurde, bestimmen. Dabei wird das Minimum der beiden Surrogate Fehlerfunktionen ausgewählt und als Fehler zurückgegeben (siehe Gleichung 2.8).

$$\min(r_t(\theta) \hat{A}_t(s, a), \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t(s, a)) \quad (2.8)$$

Da für das Lernen immer mehrere Erfahrungen genutzt werden, wird der empirische Durchschnittsfehler bestimmt. Dies wird durch $\hat{\mathbb{E}}_t$ impliziert. Somit ergibt sich die Actor Fehlerfunktion (siehe Gleichung 2.2). (Schulman u. a., 2017, S. 3f.)

Zusammenfassung der PPO Fehlerfunktion

Zum Schluss wird die PPO Fehlerfunktion definiert. Damit dies jedoch geschehen kann, muss vorher noch der Critic L_t^{VF} und Entropie Fehler $S[\pi_\theta](s_t)$ bestimmt werden. Die Critic Fehlerfunktion setzt sich folgendermaßen zusammen:

$$L_t^{\text{VF}} = (V_\theta(s_t) - V_t^{\text{targ}})^2 \text{ wobei } V_t^{\text{targ}} = r_t(\theta) \quad (2.9)$$

Der letzte Teil, welcher für die Bestimmung der PPO Fehlerfunktion benötigt wird, ist der Entropy Fehler bzw. Bonus. Dabei handelt es sich um die Entropie der Policy (Wahrscheinlichkeitsverteilung (siehe Unterunterabschnitt 2.2.1)).

Es ergibt sich damit der bereits oben erwähnte PPO Training Loss (siehe Gleichung 2.1):

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

c_1 und c_2 sind in dieser Steuerungsfaktoren zur Regulierung der Fehlerhöhe.

Um den PPO Algorithmus mit seiner Fehlerfunktion richtig anwenden zu können, wird im Weiteren noch der Algorithmus näher erklärt.

2.3.4. PPO - Algorithmus

Um die Theorie in die Praxis zu überführen, soll nun der Algorithmus komprimiert dargestellt werden, um diesen später umsetzen zu können. Diese Darstellung basiert dabei auf der Quelle (Schulman u. a., 2017) und einigen weiteren Anpassungen.

1. Initialisiere alle Hyperparameter und die neuronalen Netzwerke für Actor und Critic mit den Gewichten θ bzw. w . Erstelle einen Experience Buffer EB .
2. Bestimme mit dem Zustand s und dem Actor Network eine Aktion a . Dies geschieht durch $\pi_\theta(s)$.
3. Führe Aktion a aus und ermittle den Reward r und den Folgezustand s' .
4. Speichere Zustand, Aktion, Policy der Aktion, Reward und is_terminal in EB .
5. Ersetze den Zustand mit dem Folgezustand $s' \rightarrow s$.
6. Wiederhole alle Schritte ab Schritt 2, bis die Spielepisode endet.
7. Entnehme einen Batch aus dem Buffer.
8. Bestimmt die Ratios $r_t(\theta)$. (siehe Unterunterabschnitt 2.3.3).
9. Berechne die Advantages $\hat{A}_t(s, a)$. (siehe Unterunterabschnitt 2.3.3).
10. Berechne die Surrogate Fehler $surr_1$ und $surr_2$. (siehe Unterunterabschnitt 2.3.3).
11. Bestimmt den PPO Fehler. (siehe Unterunterabschnitt 2.3.3)
12. Aktualisiere die Gewichte des Actors und Critics. $\theta_{\text{old}} \leftarrow \theta$ und $w_{\text{old}} \leftarrow w$.
13. Wiederhole alle Schritte ab Schritt 7 K -mal. K stellt ein Hyperparameter dar.
14. Wiederhole alle Schritte ab Schritt 2 erneut, bis das Verfahren konvergiert.

Neben dem PPO Algorithmus soll auch noch eine weiter behandelt werden. Der DQN Algorithmus wird im Folgenden vorgestellt.

2.4. Deep Q-Network

Der DQN (Deep Q-Network-Algorithmus) ist ein weiterer Reinforcement Learning Algorithmus, welcher auf einer ihm zugrunde liegenden Formel basiert. Er hat bereits große Erfolge erzielen können, besonders im Gaming Bereich (Mnih u. a., 2013). Daher erscheint es auch nicht weiter verwunderlich, dass sich dieser Algorithmus einer großen Beliebtheit erfreut. Unter anderem wurden aus diesem Grund bereits viele Erweiterungen implementiert, wie der DDQN (Double Deep Q-Network) oder der DQN mit Verrauschen Netzen usw.

Basierend auf seiner großen Beliebtheit ergibt sich die Frage, ob der DQN dieser auch gerecht wird? Diese Frage lässt sich am besten mit einem Vergleich beantworten, was diesen Algorithmus zu einem guten Kandidaten für diesen 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 komplexen Environments schnell ineffizient wird. Ein weiterer Ansatz sind NN, welche die Q-Value Funktion nachbilden und erlernen können.

Ein kleiner Blick in die dem Algorithmus zugrunde liegende Logik eröffnet dahingehend einen besseren Überblick. So ist die Idee von vielen RL Verfahren, die Aktionswert Funktion mithilfe der Bellman Gleichung iterativ zu bestimmen. Daraus ergibt sich die folgende Formel (Mnih u. a., 2013)

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q(s', a') | s, a \right] \quad (2.10)$$

wobei, $Q_{i+1}(s, a)$ den $i + 1$ aktualisierten Aktionswert der Aktion a im Zustand s darstellt. Die iterativ angewendete Bellman Gleichung (siehe Gleichung 2.10) besagt daher, dass die Aktionswert Funktion Q dem empirischen Durchschnitt aller iterativen Aktualisierungen entspricht. 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^* approximiert werden $Q(s, a; \theta) \approx Q^*$. Dies geschieht mittels eines NN. Damit dieses nicht ausschließlich zufällige Q-Values ermittelt, ist eine Anpassung der Gewichte nötig. Dafür muss jedoch zuerst der Fehler des DQN Algorithmus bestimmt werden, was mithilfe der Fehlerfunktionen geschieht. Diese ist wie folgt definiert (Lapan, 2020, S.146 f.):

$$L(\theta) = \mathbb{E} \left[((r(s, a) + \gamma \max_{a'} Q(s', a'; \theta)) - Q(s, a; \theta))^2 \right] \quad (2.11)$$

Die Formel (siehe Gleichung 2.11) besagt, dass der Fehler eines zufällig ausgesuchten State-Action Tupels (s, a) sich wie folgt zusammensetzt. Der Fehler ist die Differenz aus dem Aktionswerts $Q(s, a; \theta)$, welcher Aufschluss über den, in dieser Episode, zu erwartenden Reward liefert und y_i . Dabei ist y_i der Reward, welcher durch die Ausführung der Aktion a im Zustand s erzielt wurde, addiert mit dem maximalen Q-Value der Folgeaktion a' im Folgezustand s' .

Da $Q(s, a)$ rekursiv definiert werden kann, ergibt sich in vereinfachter Form (Lapan, 2020, S.126 und S. 146 f.):

$$Q(s, a) = r + \gamma \max_{a' \in A} Q(s', a') = \mathbb{E}\left[r + \gamma \max_{a'} Q(s', a'; \theta) | s, a\right] = y_i \quad (2.12)$$

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 Fehler deklariert. Dieser wird noch quadriert, damit er stets positiv ist.

2.4.1. DQN - Algorithmus

Zur besseren Anwendbarkeit haben Volodymyr Mnhi et al. einen Algorithmus entworfen, welcher den DQN anschaulich erklärt (Mnih u. a., 2013). Da jedoch in diesem Algorithmus weiterhin auf Teile der Fehlerfunktion eingegangen wird, folge eine berinigte Version, welche sich auch für den allgemeinen Gebrauch besser anbietet. (Lapan, 2020, S. 149 f.)

1. Initialisiere alle Hyperparameter das Q-NN mit zufallsbasierten Gewichten. Setzte Epsilon $\epsilon = 1.00$ und erzeuge einen leeren Experience Buffer EB .
2. Wähle mit der Wahrscheinlichkeit ϵ eine zufällige Aktion a oder nutze $a = \arg \max_a Q(s, a)$.
3. Führt Aktion a aus und ermittle den Reward r und den Folgezustand s' .
4. Speichern von Zustand s , Aktion a , Reward r und Folgezustand s' .
5. Reduziere ϵ , sodass die Wahrscheinlichkeit, eine zufällige Aktion zu wählen, minimiert wird. Gewöhnlich existiert eine untere Grenze für ϵ , sodass immer noch einige wenige Aktionen zufällig gewählt werden.
6. Entnehme zufallsbasiert einen Batch aus dem Experience Buffer EB .
7. Berechne $y = r + \gamma \max_{a' \in A} \hat{Q}(s', a')$.
8. Berechne den Fehler $\mathbb{L} = (Q(s, a) - y)^2$

9. Update $Q_\theta(s, a)$ mithilfe von Gradientenabstiegsverfahren. Daher $Q_{\theta_i}(s, a) \rightarrow Q_{\theta_{i+1}}(s, a)$
10. Wiederhole alle Schritte von Schritt 2 an, bis sich eine Konvergenz ergibt.

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 Unterunterabschnitt 2.2.1). Es soll in der Lage sein, Aktionen zu empfangen. Des Weiteren soll eine Observation und ein Reward an den Agenten übergeben werden. 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 Abschnitt 2.1).

Reset

Das Env muss eine Reset Funktionalität implementieren, um einen erbrachten Spielfortschritt zurückzusetzen. Dies ist für den Spielablauf 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 Spieldurchgang zu erzielen zu können.

Lernen

Die Agenten müssen fähig sein, auf Grundlage vergangener Spieldurchgänge 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 ihre 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 Abschnitt 2.3 und Abschnitt 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	- 5cmErklärung
steps	- 5cmDie in einem Spiel durchgeführten Züge. Diese geben in der E später Aufschluss über die Effizienz und weisen auf Lernfehler der Agenten beispielsweise das Laufen im Kreis.
apples	- 5cmDie Anzahl der gefressenen Äpfel in einem Spiel ist ein maßgeblicher Faktor zur Einschätzung des Lernerfolges.
wins	- 5cmHat der Agent das Spiel gewonnen? Dieser Wert stellt die Endkondition 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 Tabelle 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 Kriterium der Performance wurde von verwendeten Literatur Wei u. a. (2018) und Chunxue u. a. (2019) übernommen, die anderen ergeben sich aus den zu speichernden Daten. Die einzelnen Kriterien lauten:

Tabelle 3.2.: Evaluationskriterien

Kriterium	- 5cmErläuterung
Performance	- Welcher Agent erreicht das beste Ergebnis? Im Sachzusammenhang mit dem Spiel Snake bedeutet dies: Welcher Agent frisst die meisten Äpfel, nachdem er trainiert wurde?
Siegrate	- Welcher Agent schafft die Spiele mit einem Sieg zu beenden? Im Gegensatz zur Performance, gibt die Siegrate Aufschluss über das Konvergenzverhalten.
Robustheit	- Welcher Agent kann in einer modifizierten Umgebung die 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? Dies ist in Real World Applikationen ein wichtiger Faktor, da sich unbemannte Drohnen in unbekannten Umgebungen zurechtfinden müssen.
Effizienz	- Welcher Agent löst das Spiel mit der größten Effizienz? Bezug 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.

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel soll es thematisch über den momentanen Stand der bereits durchgeföhrten Forschung gehen. Dabei sollen die Arbeiten gezielt unter den folgenden Aspekten untersucht werden.

- Optimierungsstrategien
- Reward Funktion
- Evaluationskriterien

Anschließend folgt die Diskussion über die einzelnen verwandten Arbeiten. Ausgewählt wurde diese aufgrund ihres thematischen Hintergrundes zum Spiel Snake, in Verbindung mit dem RL.

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 durchgeföhrt, 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) Agenten angewendet worden sind. Bei diesen Strategien handelt es sich um den Training Gap, die Timeout Strategy und den Dual Experience Replay.

Dieser (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 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 vorgefinierte 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 Malusses ist antiproportional 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 (siehe Gleichung 4.1).

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

Zusätzlich wird der resultierende Reward zwischen 1 und -1 geclipt $r_{res} = clip(r_{res}, -1, 1)$. Δr ist dabei wie folgt definiert (siehe Gleichung 4.2):

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

Wobei t den vorherigen und $t + 1$ den 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 herangezogen, gemessen am Score und an den steps survived, also die überlebten Schritte.

4.1.1. Diskussion

Sowohl die Training Gap Strategy 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 (siehe Abschnitt 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 geeignet. Zwar sind alle Funktionalitäten des im Paper verwendeten DQN gegeben (siehe Unterabschnitt 3.2.1), jedoch wurde der Fokus kaum auf eine Parametrisierung (siehe Unterabschnitt 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 Abschnitt 3.3. Zhepei Wei et al. verzichteten auf einen mehrfache Datenerhebung ihrer experimentellen Ergebnisse. Dem Leser werden nur indirekt Informationen über die erhobenen Daten mitgeteilt. Aus Statistiken lässt sich jedoch schließen, dass der Scores und die steps (Anzahl der Schritt) gespeichert wurden.

Weiterhin wurde sich bei den Evaluationskriterien einzig auf die Performance und Spielzeit konzentriert. Weitere Kriterien wie beispielsweise die Robustheit oder Siegrate (siehe Tabelle 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 Real World Applikationen (siehe Abschnitt 1.1).

Des Weiteren ist erwähnenswert, dass die Verfasser auf das Evaluationskriterium der steps survived setzten, welches im kompletten Widerspruch zur Effizienz (siehe Tabelle 3.2) 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 der x-ten Zone darstellt). Diese Zonen stellen den zunehmenden Duft von Nahrung dar, wobei dieser immer stärker wird, umso 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, nach Implementierung des dynamischen Positionsspeichers 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 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 natürlich 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 ihren Anforderungen (siehe Kapitel 3).

Aus Grafiken geht hervor, dass die Verfasser ein Env mit einer Visualisierung besitzen. Weiterhin ist davon auszugehen, dass auch alle weiteren Anforderungen bezüglich des Env (siehe Abschnitt 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 Unterabschnitt 2.2.1).

Auch erfüllt die Ausarbeitung alle Anforderungen an die Funktionalitäten von Agenten (siehe Abschnitt 3.2). 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 Handvoll 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 Unterabschnitt 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 Unterabschnitt 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 (siehe Tabelle 3.1).

Chunxue Wu et al. verwendeten zudem hauptsächlichen den Score und die Q-Values als Evaluationskriterien. Um die Effizienz der im Paper verwendeten Strategien (Optimierungen) zu zeigen, wurden ebenfalls die gemittelten Steps pro Spiel und das Auftreten von Loop Storms als weitere untergeordnete Evaluationskriterien verwendet. In diesem Paper werden daher im Vergleich zum ersten Paper (siehe Ab-

schnitt 4.1), deutlich mehr Evaluationskriterien genutzt, welche sich partiell mit denen aus dieser Ausarbeitung überschneiden. Dennoch legt das hier betrachtete Paper deutlich mehr Wert auf den Fehler des Agenten. In dieser Ausarbeitung liegt der Fokus deutlich mehr auf der Leistung und Effizienz.

4.3. Zusammenfassung

Sowohl „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ (siehe Abschnitt 4.1) als auch „UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene“ (siehe Abschnitt 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 Env (siehe Abschnitt 3.1) und zu den Agenten (siehe Abschnitt 3.2), mit Ausnahme der Parametrisierung (siehe Unterabschnitt 3.2.2) und der Diversität der Algorithmen (siehe Unterabschnitt 3.2.3), alle erfüllt. Dennoch wurden auch Anforderungen bei der statistischen Datenerhebung und Evaluation nicht erfüllt. Zu diesen zählen z.B. die mehrfache Datenerhebung (siehe Unterabschnitt 3.3.1) und die Evaluation-Anforderungen (siehe Abschnitt 3.5). Letztere weichen, verständlicherweise, von den Anforderungen ab, da sich die Verfasser der Arbeiten auf andere Aspekte konzentriert haben und daher nicht dieselben Evaluationskriterien gewählt haben.

Im Weiteren wird nun mit dem Konzept dieser Ausarbeitung fortgefahrene.

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. Zum Schluss 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 Aktivitätsdiagramms darstellen, in welchem die einzelnen Schritte des Vergleichs visuell dargestellt werden.

Als erstes werden die Agenten erstellt (siehe Abbildung 5.1), indem mehrere Instanzen der Agent Klasse von DQN und PPO mit verschiedenen Hyperparametern generiert werden.

Mit diesen Agenten werden nun die weiteren Baseline Vergleiche durchgeführt, in welchen für jedes Evaluationskriterium der jeweils beste Agent bestimmt wird. Grund dafür ist, dass das Anwenden der Optimierungen viele Ressourcen bindet. Daher sollen nur die vielversprechendsten Agenten zu den Optimized Vergleichen zugelassen werden. Der verwendete Algorithmus besitzt dabei keinen Einfluss auf die Auswahl der Agenten, sodass auch nur DQN oder PPO Agenten Gewinner der Baseline Vergleiche sein können. Genauere Details zur Durchführung der Baseline Vergleiche finden sich im Unterabschnitt 5.5.1. Auf die Sieger Agenten der Baseline Vergleiche werden daraufhin die Optimierungen angewendet (siehe Abbildung 5.1). Mit diesen optimierten Agenten werden nun die sogenannten Optimized Vergleiche bezüglich jedes einzelnen Evaluationskriteriums (siehe Tabelle 3.2) erneut wiederholt. Im vorletzten Schritt soll wieder der optimale Agent für jedes Evaluationskriterium ermittelt werden. Dabei können dies auch Baseline Agenten sein.

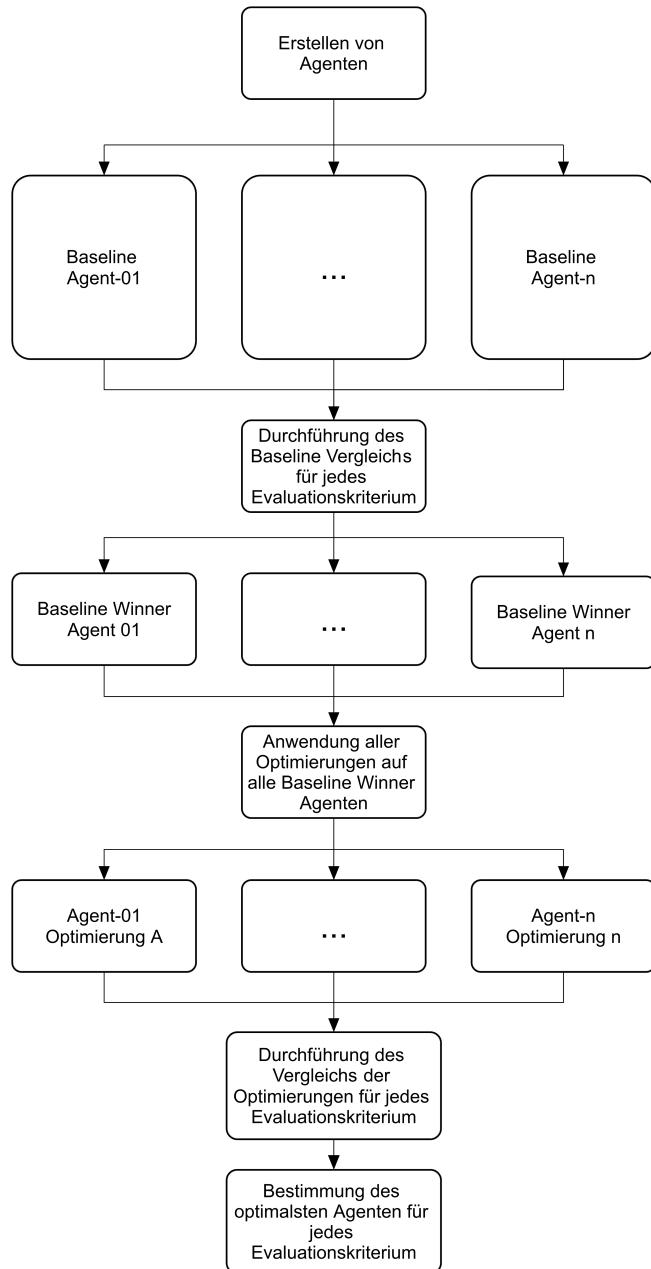


Abbildung 5.1.: Darstellung des Vorgehens.

Zum Schluss wird auf Grundlage der Sieger Agenten entsprechend der Forschungsfrage der optimale Agent ermittelt. Dafür ist eine Priorisierung der Evaluationskriterien nötig, um zu bestimmen, welche Eigenschaften des Agenten am wichtigsten für den Benutzer sind. In dieser Ausarbeitung wird die Performance priorisiert, gefolgt von der Sieg-Rate, der Robustheit und zum Schluss der Effizienz. Diese Priorisierung bedeutet jedoch nicht, dass der Agent mit der besten Performance automatisch gewonnen hat. Sie liefert lediglich ein Verhältnis zwischen den Evaluationskriterien, auf dessen Basis der Benutzer den nach seinen Maßstäben besten Agenten auswählen kann.

5.2. Environment

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

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

5.2.1. Spiellogik

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

Die Game-Komponente stellt die Hauptkomponente dar, da sie die eigentliche Aktionsdurchführung implementiert. Sie beinhaltet jeweils eine Instanz der Reward-, Observation-, GUI- und Player-Komponente. Letztere ist eine Datenhaltungskomponente, die 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 Unterunterabschnitt 5.2.1, angegeben. Zuzüglich wird im Rahmen der Optimierungen A (siehe Unterabschnitt 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 Spielfelds (shape) und die Position des Apfels. Die Game-Komponente beinhaltet zudem viele Methodiken, wie z.B. die Aktionsausführung, die Observation- und Reward-Erstellung usw.

In der Player-Komponente werden spielerbezogene Daten verwaltet. Zu diesen zählen die Position des Kopfes der Snake, sowie die Positionen aller Schwanzglieder, die Ausrichtung (direction), die geläufenen Schritte seit dem letzten Fressen eines Apfels (inter_apple_steps) und der Lebensstatus (is_terminal).

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

Zur Erzeugung der grafischen Oberfläche implementiert die GUI-Komponente die Funktionalität ein Fenster zu öffnen, welches das Spielgeschehen anzeigt.

Spielablauf

Die eigentliche Aktionsabarbeitung wird durch das Aufrufen der step Funktionalität in der Schnittstellen-Komponente (siehe Unterabschnitt 5.2.2) bewirkt. Diese ruft daraufhin die Routinen zur Erstellung des Rewards (evaluate) und der Observation (observe) auf, welche in der Game-Komponente implementiert sind. Um die Ab-

arbeitung einer Aktion durchzuführen, wird die action Funktionalität aufgerufen. Der Ablauf einer Aktionsabarbeitung ist in der Abbildung 5.2 dargestellt. Zu Beginn wird überprüft, ob die Snake, seit dem letzten Fressen, mehr Schritte als die eigentliche Spielfeldgröße gegangen ist. Diese liegt standardmäßig bei $(8 \times 8 \rightarrow 64)$.

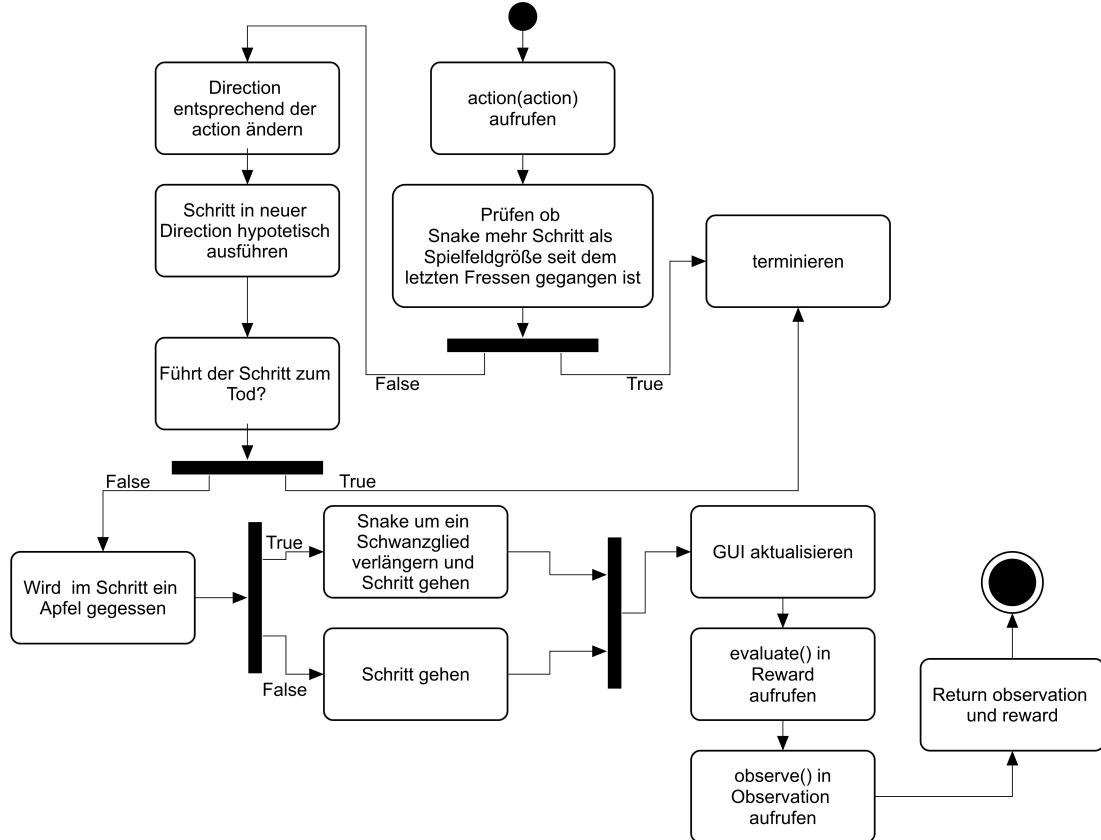


Abbildung 5.2.: Darstellung eines Schritts in der Spieldiode.

Im Rahmen der Bestimmung der Robustheit wird sich diese im Testverlauf jedoch ändern (siehe Unterabschnitt 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	- 4cmErklärung
turn left	- 4cmDie Snake verändert ihre Richtung um 90° nach links. Z.B. von N \rightarrow W
turn right	- 4cmSnake verändert ihre Richtung um 90° nach rechts. Z.B. von N \rightarrow E
do nothing	- 4cmDie 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 aktualisierter Ausrichtung, hypothetisch durchgeführt. Dabei wird festgestellt, ob die Ausführung des Schritts zum Tod der Snake führt. Sollte dies der Fall sein, so wird der Spielablauf terminiert. Andernfalls wird der Schritt durchgeführt. Dabei wird zwischen zwei Fällen unterschieden.

Sollte die Snake einen Apfel gefressen haben, also Kopf und Apfel dieselbe 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 (siehe Abschnitt 2.1).

Sollte die Snake keinen Apfel gefressen haben, so führt sie den Schritt aus und es bewegen sich alle Schwanzglieder auf die Vorgängerposition, mit Ausnahme des Kopfes, welcher die neue Position einnimmt (siehe Abschnitt 2.1).

Nach der Ausführung einer dieser beiden Fälle, wird die Spieloberfläche (ground) und GUI aktualisiert. Danach ist die Aktionsabarbeitung abgeschlossen.

Damit der Agent die neue Observation und den neuen Reward erhält, werden diese in der Reward bzw. Observation-Komponente bestimmt und zurückgegeben.

Reward

Der Reward wird in der Reward-Komponente, basierend auf dem letzten Zug, gebildet. Dies geschieht nach folgendem Vorbild. Der Standard Reward ist abhängig vom Fressen eines Apfels, vom Sieg und vom Verlust einer Spielepisode. 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 das Spiel gewonnen (siehe Abschnitt 2.1).

Die zweite Reward Funktion wird in Abschnitt (siehe Unterabschnitt 5.4.1) erklärt.

Observation

Die Observation wird in der Observation-Komponente mithilfe verschiedener Unterfunktionen erzeugt und besteht aus der around_view (AV) und der scalar_obs (SO).

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 Codierungsschema. 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 (siehe Tabelle 5.2):

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

Channel der Matrix (Ax13x13)	- 5cmErklärung
A = 0	- 5cmDie erste Feature Map signalisiert den Raum außerhalb des Spielfeldes.
A = 1	- 5cmDiese Feature Map stellt alle Schwanzglieder mit Ausnahme des Kopfes letzten Schwanzgliedes dar.
A = 2	- 5cmIn dieser Feature Map wird der Kopf der Snake dargestellt.
A = 3	- 5cmDamit gegen Ende des Spiels der Agent noch freie Felder erkennen wird in dieser Feature Map jedes freie und sich im Spielfeld befindliche Felder codiert.
A = 4	- 5cmDie vorletzte Feature Map codiert das Schwanzende der Snake.
A = 5	- 5cmIn der letzte Feature Map wird der Apfel abgebildet.

Vorteilhaft an der AV ist, dass im Gegensatz zu den verwandten Arbeiten Wei u. a. (2018) und Chunxue u. a. (2019), nicht das gesamte Spielfeld übertragen wird, sondern nur der wichtigste Ausschnitt. Dies reduziert die Menge an zu verarbeitenden Daten. Ein Nachteil dieser Obs ist jedoch die Unvollständigkeit. Sollte beispielsweise der blaue Punkt (der Apfel) in Abbildung 5.3 außerhalb des grauen Kastens 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, Hunger- und Blickrichtungsanzeige (direction). In Quelle (Glassner, 1989) wird das grundlegende Verfahren, auf welchem die Raytracing Distanzbestimmung basiert, vorgestellt.

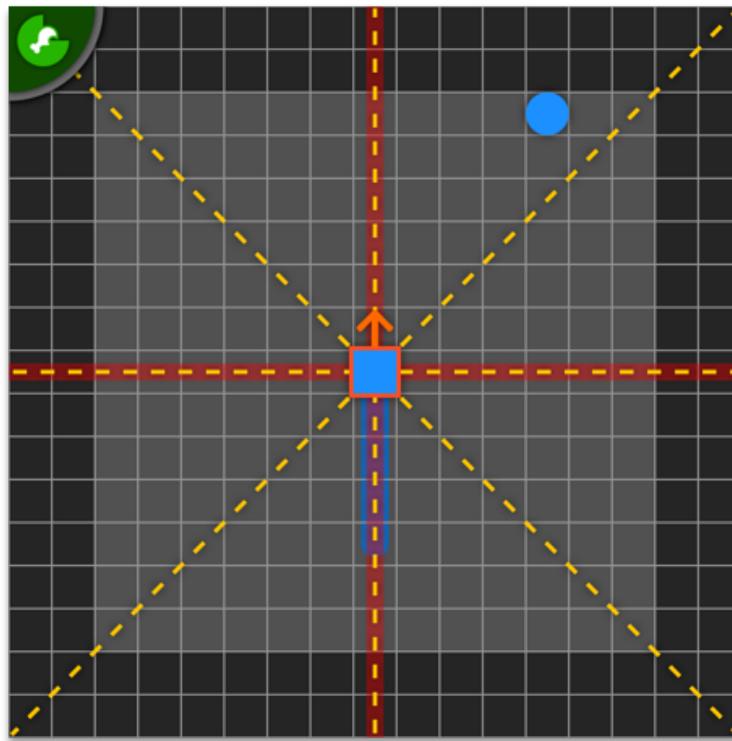


Abbildung 5.3.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar. Schwarze Rechtecke werden nicht von der AV abgedeckt und Graue liegen innerhalb dieser. Die gelben gestrichelten Linien stellen die Raytracing Distanzbestimmung dar. Der blaue Kreis stellt den Apfel dar und der grüne Viertelkreis oben links symbolisiert den Hunger.

Zuzüglich werden der SO noch zwei Kompassse für die relativen Positionsinformationen 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 oder unterhalb oder in derselben Zeile befindet (Matrixsicht). Analog verhält es sich mit den Spalten.

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 Unterunterabschnitt 5.2.1). Zur besseren Verarbeitung für die neuronalen Netzwerke, wird diese Differenz zudem mit -1 quadriert. Um mit der Unendlichkeit auftretende Probleme zu umgehen wird zwei zurückgegeben, wenn die Differenz null wird. Daraus ergibt sich die folgende Formel (siehe Gleichung 5.1).

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

In ähnlicher Weise wird mit den Raytracing Distanzbestimmungen verfahren. Bei

diesen handelt es sich um acht Distanzmesserlinien, die in 45° Abständen ausgesandt werden (siehe Abbildung 5.3). Befindet sich das gesuchte Objekt in dieser Linie, so wird die Distanz ermittelt und analog zum Hunger angepasst. Gesuchte Objekte der Raytracing Distanzbestimmungen 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 grafische 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 des 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. Die step Funktionalität ist für das Aufrufen der Aktionsausführung zuständig. Entsprechend der Anforderung der standardisierten Schnittstelle (siehe Unterabschnitt 5.2.2) gibt sie nur Reward und Observation zurück.

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

Render ist für die Visualisierung der Spieloberfläche verantwortlich (siehe Unterunterabschnitt 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 dieser kleineren Anpassungen an den Netzen

vorgenommen werden.

In den Ausarbeitungen 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 eine zweiteilige Netzstruktur verwendet wird.

Die gesamte Netzstruktur besteht aus dem AV-Network und dem Actor-, Critic-, Q-Net-Tail. Begonnen wird mit dem AV-Network, welches 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 letzte Zeile und Spalte der originalen AV verarbeitet werden. Nach dem max-pooling besitzen die Feature Maps eine Größe von 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 5.4 dargestellt. Da das NN in beide Algorithmen verwendet wird, müssen Network-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 der Ausgabe des AV-Networks 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, Abbildung 5.5 links. Der Critic des PPO Algorithmus verwendet hingegen den Critic-Tail, siehe Abbildung 5.5 Mitte. Dieser leitet den Joined Tensor durch zwei weitere FC Layers.

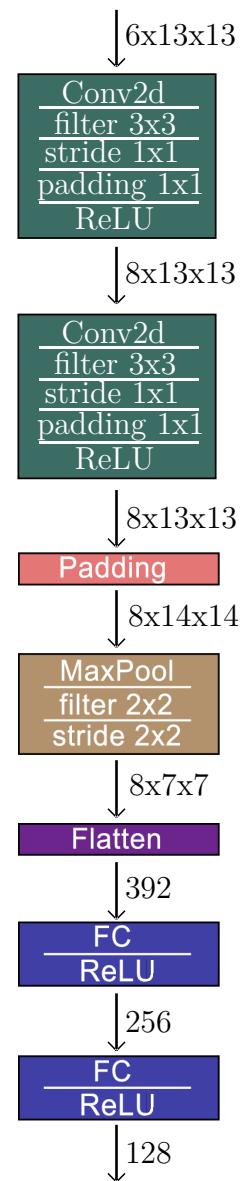


Abbildung 5.4.:
AV-Network

Da der Critic für jeden State die Discounted Sum of Rewards bestimmt (siehe Unterunterabschnitt 2.3.3), gibt dieser einen Tensor mit einem einzigen Wert zurück (Skalar). Der Q-Net-Tail ist in seinem Aufbau sehr ähnlich zum Critic-Tail, (siehe Abbildung 5.5) rechts. Da dieser jedoch die Q-Values für jede Aktion im Zustand bestimmten soll, muss ein Tensor der Größe drei zurückgegeben werden.

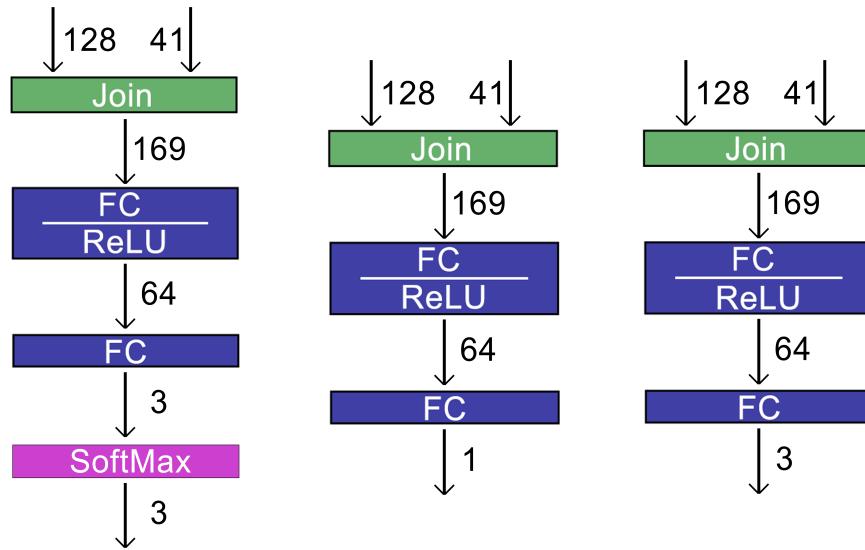


Abbildung 5.5.: Darstellung des Actor-Tail (links), Critic-Tail (Mitte) und des Q-Net-Tail (rechts).

5.3.2. DQN

Der DQN Algorithmus und damit auch die Agenten, welche auf diesem basieren, bestehen aus drei Komponenten. Diese ermöglichen die Aktionsbestimmung und Lernprozedur. Diese Hauptfunktionalitäten sind in der DQN-Komponente eingebettet, welche die zentrale Instanz des DQN darstellt. In ihr werden wichtige Konstanten für den 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 Unterabschnitt 2.4.1). Die Q-Network-Komponente verwaltet das NN (Q-Network). Dieses wird zur Q-Value Bestimmung und damit zur Aktionsbestimmung genutzt. Zudem wird es durch

den Lernprozesse aktualisiert.

Im Weiteren folgt die Erklärung dieser Prozeduren. Begonnen wird dabei mit der Aktionsbestimmung.

Aktionsauswahlprozess

Eine genaue Darstellung der Aktionsbestimmung befindet sich in Abbildung 5.6. Um eine Aktion zu bestimmen, muss zuerst ein Zufallswert zwischen null und eins, was den Wahrscheinlichkeiten von 0% bis 100% entspricht, generiert werden.

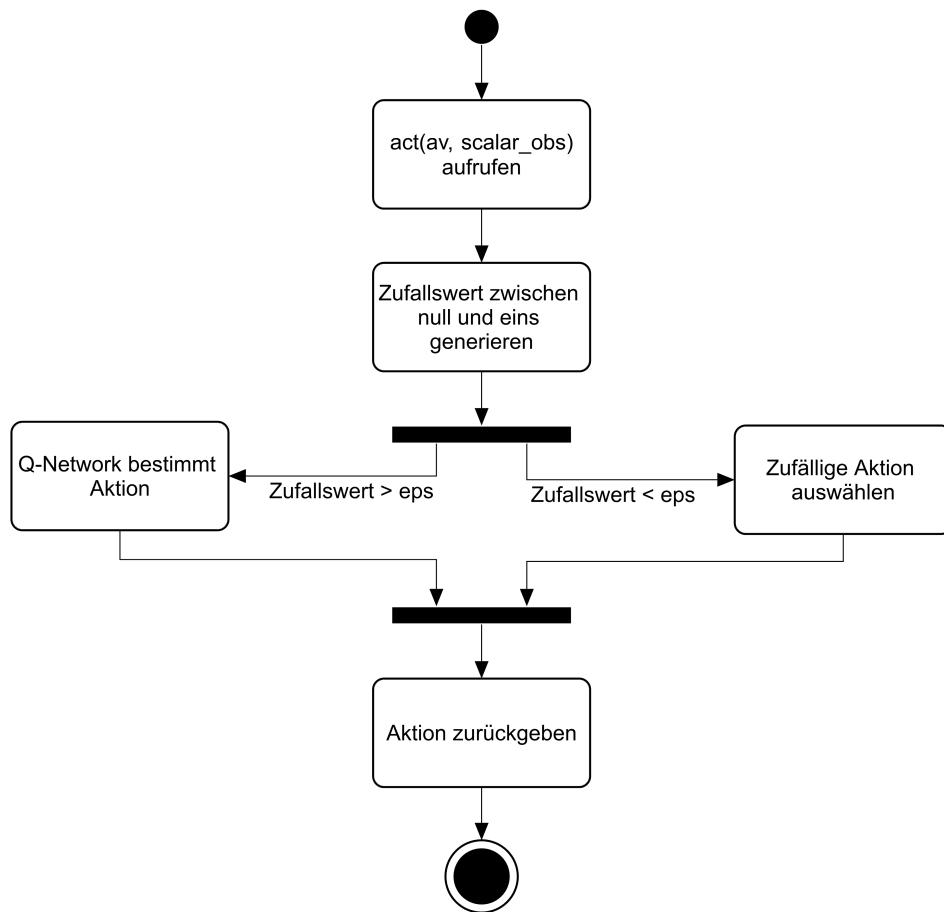


Abbildung 5.6.: Darstellung der Aktionsbestimmung des DQN Agent.

Ist der Zufallswert größer als der 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 entsprechend der Ausführungen in Unterabschnitt 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 $q = (0.32, -0.11, 0.45)$ 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 ??.

Lernprozess

Der Lernprozess stellt sich wie folgt dar:

Zuerst wird überprüft, ob im Memory genügend Erfahrungen (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 beschienen.

Danach wird der Q-Value $Q(s_i, a_i; \theta)$ der gespeicherten Aktion a_i im State s_i unter den Netzwerkparametern θ bestimmt (siehe Gleichung 2.11). Dieser wird als Q-Eval definiert.

Danach werden die Q-Values $Q(s', A)$ aller Aktionen A des Folgezustandes s' bestimmt. Sollte der Folgezustand ein terminaler Zustand sein, so werden die Q-Values auf null gesetzt, da diese die zu erwartende Discounted Sum of Rewards angeben. In einem terminalen Zustand ist diese Summe gleich null, da keine Zustand mehr besucht werden (siehe Abschnitt 2.4).

Daraufhin wird der maximale Q-Value des Folgezustands bestimmt $Q(s', a')$, mit Gamma multipliziert und mit dem erhaltenen Reward addiert $r(s, a) + \gamma \max_{a'} Q(s', a'; \theta)$ Gleichung (siehe Gleichung 2.11). 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

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 (gamma), der Epsilon-Clip-Wert (eps_clip) (siehe Unterunterabschnitt 2.3.3), die Anzahl der Trainingsläufe pro Datensatz (K_Epochs), die Lernrate (lr) und weitere statische Konstanten, gespeichert. Auch Instanzen der Unterkomponenten, die im weiteren Verlauf erklärt werden, sind in dieser Komponente

aufbewahrt.

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 (is_terminal) und die logarithmierte Wahrscheinlichkeit der ausgewählten Aktion (log_prob).

Die Actor-Komponente verwaltet das Actor-Network, welches zur Aktionsauswahl genutzt wird und die Critic-Komponente verwaltet das Critic-Network, welches einzig von der Lernprozedur verwendet wird, um die erwartete Discounted Sum of Rewards zu bestimmen (siehe Unterunterabschnitt 2.3.3). Mit dieser wird im Trainingsverlauf der Critic Fehler bestimmt (siehe Gleichung 2.9).

Aktionsauswahlprozess

Der Aktionsauswahlprozess wird in der PPO-Komponente angestoßen. Die AV und SO werden daraufhin durch das Actor-Network propagiert. Der vom Actor ausgegebene Tensor der Größe drei (drei mögliche Aktionen) beinhaltet eine Wahrscheinlichkeitsverteilung, auf dessen Basis die nächste Aktion bestimmt wird (siehe Unterabschnitt 5.3.1).

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

Für die Testläufe wird immer die Aktion ausgewählt, welche die größte Wahrscheinlichkeit vorweist (siehe Unterunterabschnitt 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 Unterunterabschnitt 2.3.3).

Danach wird die folgende Prozedur mehrmals ausgeführt, um Actor und Critic zu trainieren. Danach terminiert die Lernprozedur. Für ein besseres Verständnis, wird der Ablauf exemplarisch an einer einzelnen Erfahrung erklärt.

Zunächst wird die logarithmierte Wahrscheinlichkeit $\pi_\theta(a|s)$ für die gespeicherte Aktion a bestimmt (siehe Unterunterabschnitt 2.3.3). Dazu wird die, aus dem Memory entnommene, AV (around_view) und SO (scalar_obs) durch das Actor- und Critic-Network propagiert. Anschließend wird die logarithmierte Wahrscheinlichkeit der Aktion bestimmt und zusammen mit dem Baseline Estimate (siehe Unterunterabschnitt 2.3.3) und der Entropie der Wahrscheinlichkeitsverteilungen zurückgegeben

(siehe Unterunterabschnitt 2.3.3). Daraufhin wird die Probability Ratio aus der so eben bestimmten logarithmierten Wahrscheinlichkeit und der alten logarithmierten Wahrscheinlichkeit bestimmt (siehe Unterunterabschnitt 2.3.3).

Nachfolgend wird der Advantage, durch Subtraktion des Return mit dem Baseline Estimate, berechnet $\hat{A}(s, a) = R - b(s)$ (siehe Unterunterabschnitt 2.3.3).

Als Nächstes werden die Surrogate Fehler 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 Unterunterabschnitt 2.3.3), mit welchen der Actor Fehler $L^{\text{CLIP}}(\theta)$ berechnet wird (siehe Gleichung 2.2). Um den Gesamtfehler des PPO zu bestimmen, wird zusätzlich noch der Critic Fehler $L^{\text{VF}}(\theta)$ (siehe Gleichung 2.9) und der Entropy Fehler bzw. Bonus bestimmt (siehe Gleichung 2.1). Diese werden dann alle miteinander verrechnet, 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 Unterunterabschnitt 2.3.3). Actor- und Critic-Network werden dann mit dem Fehler 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 Agenten. Basierende auf den Grundlagen (siehe Abschnitt 2.3 und Abschnitt 2.4), sollen an dieser Stelle 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 soll im Weiteren die Auswahl der Hyperparameter begründet werden.

Wie in der Abbildung 5.1 zu erkennen ist, können beliebig viele Agenten miteinander verglichen werden. In dieser Ausarbeitung hingegen sollen sechs Agenten definiert und miteinander verglichen werden (siehe Abbildung 5.7). Dabei sollen die Hyperparameter so gewählt werden, dass stets ein langsamer, ein schneller und ein Hybrid aus beiden zuvor erwähnten Agenten definiert wird.

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 5.7.: Darstellung der zu untersuchenden Agenten.

Der erste Agent PPO-01 soll ein langsamer, jedoch 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. 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 Wert 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 durch eine große Epsilon-Dekrementierung (EPS_DEC) von 6e-5 verstärkt. Auch sorgt der verhältnismäßig große Wert für EPS_MIN von 0.0075 für eine schnellere Exploration und damit für ein schnelles Lernen.

Der DQN-03 ist wieder als Kompromiss gedacht. Eine kleine Lernrate (LR) von 2.5e-4 und eine große Epsilon-Dekrementierung (EPS_DEC) von 4e-5 sind dem schnell lernenden Agenten zuzuordnen. Eine große Batch- (BATCH_SIZE) und Memory-Size (MAX_MEM_SIZE) von 128 und $2^{**}11$ sind dagegen dem langsam lernenden Agenten zuzuschreiben.

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 (siehe 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 ziellose Umherlaufen, bestraft.

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

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

Die Implementierung dieser neuen Reward Funktion findet in der Reward-Komponente statt. Dabei wird der Distanz Reward der Quelle Wei u. a. (2018) benutzt, da dieser, im Gegensatz zur Standard Reward Funktion (siehe Unterunterabschnitt 5.2.1), viele Faktoren des Spiels berücksichtigt. 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 der Hauptablauf näher erklärt werden, welcher Agenten und Env miteinander interagieren lässt. Danach wird die Statistik-Komponente mit ihren Funktionalitäten vorgestellt.

5.5.1. Datenerhebung

Die Hauptablaufroutine implementiert die eigentlichen Test- und Trainingsabläufe. Um diese auszuführen, werden wichtige Hyperparameter des Ablaufs, wie z.B. die zu absolvierenden Trainingsspiele (N_ITERATIONS), die Spielfeldgröße (BOARD_SIZE) und weitere 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. Testläufen und die Trainingsdaten aus den Trainingsläufen stammen. Letztere werden wie folgt erhoben. Zu Beginn werden die Datenhaltungsobjekte apples, wins und steps initialisiert, welche für jedes absolvierte Trainingsspiel die entsprechenden Werte speichert (siehe Tabelle 3.1). Nach der Erstellung des Agenten und Environments startet der

Spielverlauf.

Dabei wird wie in Unterabschnitt 2.2.2 vorgegangen. Der Agent erhält eine Obs, bestimmt seine Aktion und diese wird im Env ausgeführt. Danach wird die neue Obs und der neue 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 Unterunterabschnitt 5.3.2) bzw. PPO-Lernprozess (siehe Unterunterabschnitt 5.3.3) dargestellt, durchgeführt. Danach werden die oben genannten Datenhaltungsobjekte aktualisiert und die Prozedur beginnt von Neuem. Wenn alle Epochs (N_ITERATION) absolviert wurden, werden die erhobenen Daten weiter in der Statistik-Komponente verarbeitet Abschnitt (siehe Unterabschnitt 5.5.2).

Die Datenerhebung für Testdaten findet annähernd auf die gleiche Weise statt, jedoch werden die Daten nicht aus Trainingsläufen entnommen, sondern aus Testläufen. 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 (6x6) bis (10x10) annehmen. Bei den übrigen Evaluationskriterien 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 Unterabschnitt 5.5.2). Um die Validität der Statistiken zu garantieren, sollen alle Daten für die Statistiken zweimal erhoben werden. Damit soll der Anforderung der mehrfachen Datenerhebung entsprochen werden (siehe Unterabschnitt 3.3.1).

5.5.2. Datenverarbeitung und Erzeugung von Statistiken

Nach der Erstellung der Test- und Trainingsdaten für jeden Agenten werden diese entsprechend der Evaluationskriterien (siehe Tabelle 3.2) verarbeitet. Sie werden dabei mit Statistiken und Tabellen abgebildet. Die Verarbeitung der Daten geschieht dabei wie folgt:

Die Performance wird anhand der gefressenen Äpfel gemessen. Es entsteht daher die Äpfel pro Epoch (Spiel) Rate.

Die Effizienz wird durch die durchschnittliche Schrittanzahl pro Apfelanzahl pro Epoch (Spiel) dargestellt.

Die Robustheit wird durch die durchschnittliche Apfelanzahl pro Epoch (Spiel) pro Spielfeldgröße dargestellt. Die Spielfeldgröße variiert dabei zwischen (6x6) und (10x10).

Die Siegrate wird mithilfe der wins bestimmt. Dazu wird jedem Sieg eine eins und jedem Verlust eine null zugeordnet. Damit lässt sich dann die Siegrate bestimmen. Diese Daten werden dann mithilfe von Statistiken und Tabellen ausgewertet, welche aus den mehrfachen Erhebungen stammen. Dazu sollen diese bei der Auswertung gemittelt werden, um jeweils eine Statistik bzw. Tabelle für jedes Evaluationskriterium zu erhalten. Weitere Details zur statistischen Auswertung befinden sich in der Evaluation (siehe Unterabschnitt 7.2.3).

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 diese 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 den Informationsaustausch mit dem Agenten sorgen. Diese sind im Konzept 5.2.2 aufgeführt. Die step Methode (siehe Abbildung 6.1) wird in der Ablaufprozedur (Test- bzw. Trainingsmethode) aufgerufen. Um die übergebene Aktion auszuführen, ruft diese die action Methode in der SnakeGame Instanz auf. Diese führt die Aktion aus und manipuliert damit das SnakeGame.

Nach dem Aufruf von der action Methode ist die Aktion ausgeführt und es wird die nächste Observation mithilfe der make_ obs Methode bestimmt, welche dem Agenten übergeben wird. Im Anschluss wird zudem noch der Reward gebildet sowie die Statusinformation, ob sich das Spiel in einen terminalen Zustand befindet (siehe Abbildung 6.1).

Die reset Methode (siehe Abbildung 6.1) wird zu Beginn des Spielablaufs aufgerufen, um eine initiale Obs zu erhalten. Sie setzt den momentanen Spielfortschritt zurück (siehe Unterunterabschnitt 5.2.1). Dazu wird die reset_snake_game Methode aufgerufen, welche SnakeGame und Player zurücksetzt. Anschließend wird eine neue Obs mithilfe der make_ obs Methode erstellt und zurückgegeben.

Die render Methode (siehe Abbildung 6.1) aktualisiert die GUI, indem diese die updateGUI aufruft. Sie wird nur in Testmethoden aufgerufen (siehe Unterabschnitt 6.5.1).

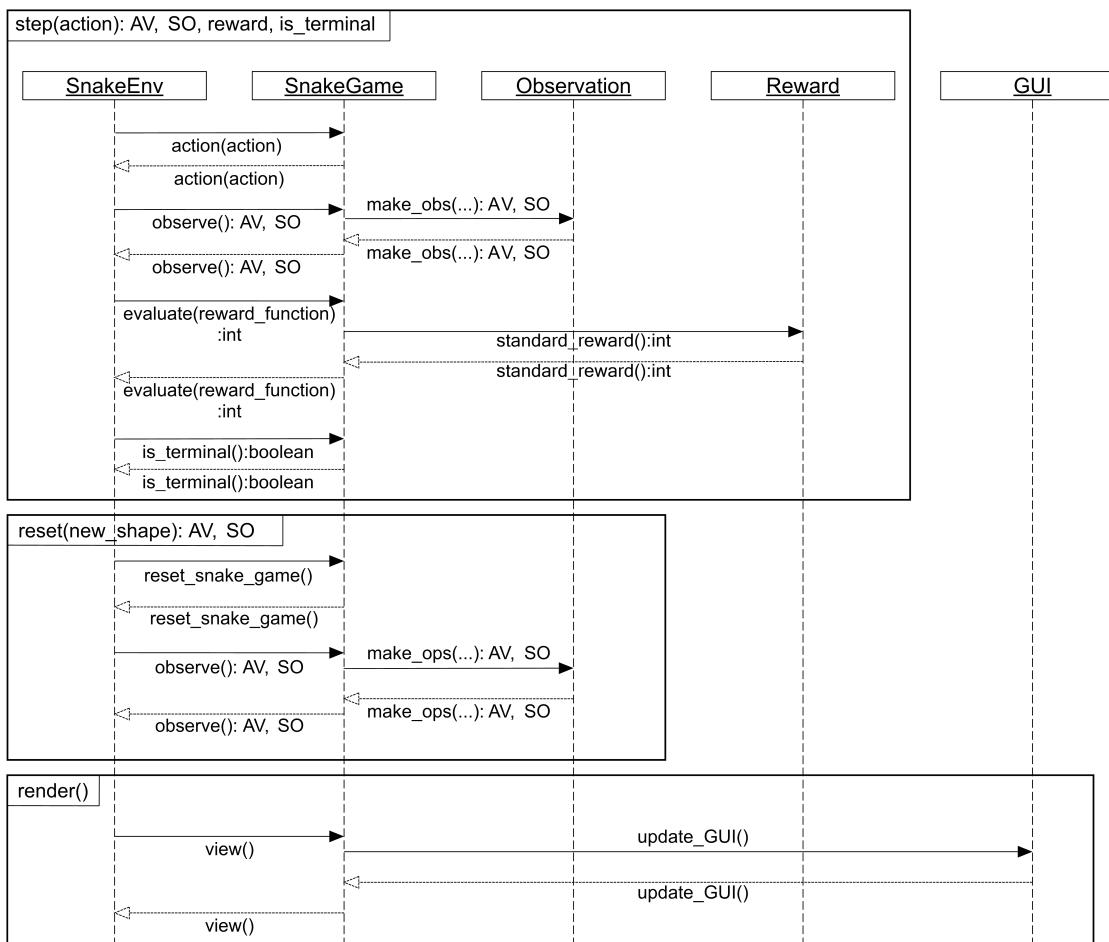


Abbildung 6.1.: Darstellung der Schnittstellenmethoden step, reset und render in einem Sequenzdiagramm.

Da die action Methode die eigentliche Aktionsausführung (siehe Unterunterabschnitt 5.2.1) implementiert, wird diese noch näher erläutert. Die Spiellogik ist hauptsächlich in der action Methode implementiert. Sie basiert auf den Beschreibungen des Konzepts (siehe Unterunterabschnitt 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 Spielfelds liegt. Wenn dies zutrifft, wird das Spiel terminiert. Ansonsten wird der neue Kopf in die Liste aller Snake-Glieder (`Player.tail`), an erster Stelle, eingefügt. Sollte die Snake zu diesem Zeitpunkt das gesamte Spielfeld ausfüllen, so hat sie gewonnen und die Methode terminiert. Ansonsten wird geprüft, ob die Snake im momentanen Schritt einen Apfel gefressen hat oder nicht. Sollte sie einen Apfel gefressen haben, so wird der Apfel

entfernt und ein Neuer generiert. Zudem wird der inter_apple_steps Zähler zurückgesetzt. Hat die Snake keinen Apfel gefressen, wird das letzte Schwanzstück entfernt, um die Illusion von Bewegung zu erzeugen und der inter_apple_steps Zähler wird 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 die Methode terminiert. Andernfalls wird das Spielfeld (Playground) mit den Elementen der Snake aktualisiert.

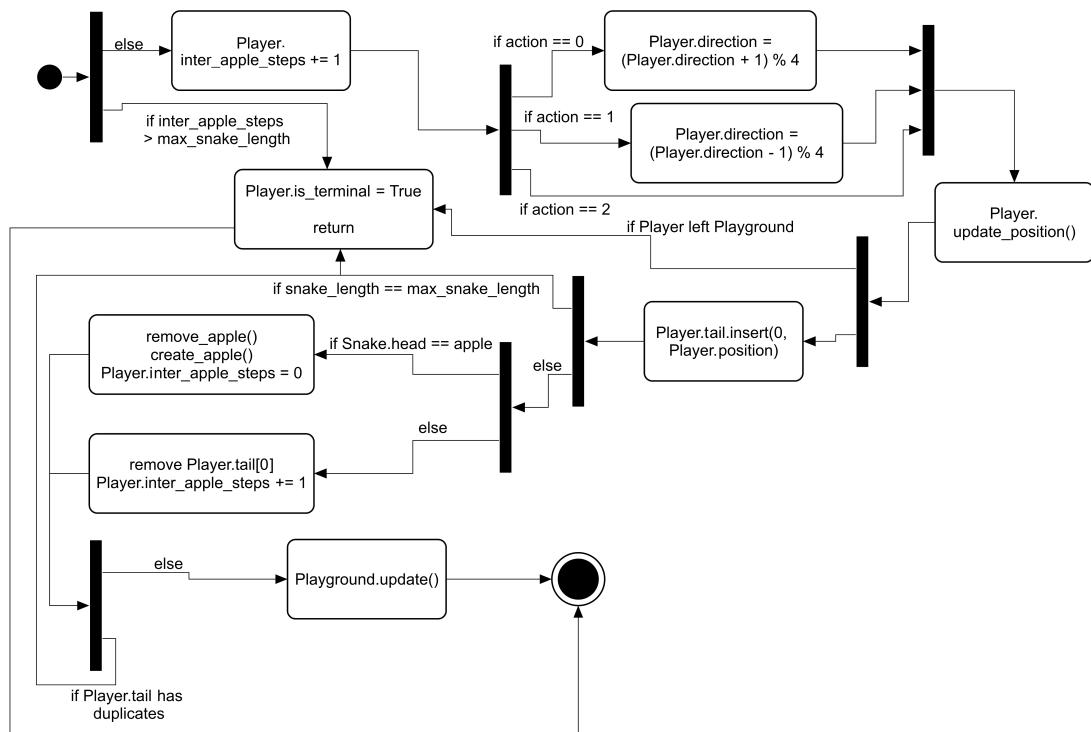


Abbildung 6.2.: Darstellung des Ablaufs der action Methode in einem Ablaufdiagramm.

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 gefordert ist (siehe Unterabschnitt 5.3.1). Eine genauere Darstellung der Implementierung des AV-Networks findet sich im Anhang (siehe Abschnitt B.3). Das AV-Network wird von allen Algorithmen und daher auch von den DQN Agenten genutzt, welcher als Nächstes thematisiert werden.

6.3. DQN

Die Implementierung des DQN Algorithmus wurde von der Quelle (Tabor, 2020) inspiriert.

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 (siehe Unterunterabschnitt 5.3.2) und learn (siehe Unterunterabschnitt 5.3.2) Methoden. Sie ist dabei als DQN-Komponente zu interpretieren. Die Memory Klasse entspricht der Memory-Komponente. Die QNetwork Klasse beinhaltet das NN zur Aktionsbestimmung (siehe Unterabschnitt 5.3.1)) und wird in Folgenden näher thematisiert.

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 Unterabschnitt 5.3.1). Es wird durch das PyTorch Framework realisiert und besteht aus den im Unterabschnitt 5.3.1 erwähnten Elementen. Die forward Methode leitet die AV (around_view) durch das AV_NET (siehe Abschnitt 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. Um die zurückgelieferten Werte für das lernen nutzen zu können, müssen sie jedoch im Memory gespeichert werden.

6.3.2. Memory

Die Klasse Memory besteht aus einer Reihe von Tensoren, welche Daten mittels einer zusätzlichen Dimension speichern. Sollte die AV die Form (6x13x13) besitzen, so besitzt der Speicher-Tensor 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 (siehe Unterabschnitt 5.3.2) 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.

Das Memory ist ein zentraler Bestandteil der Agent Klasse, welche im Weiteren erläutert wird.

6.3.3. 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 hält der Agent jeweils eine Instanz der Memory und QNetwork Klasse.

Die Aktionsbestimmung wird dabei durch die act Methode durchgeführt, die im Weiteren erklärt wird.

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 Unterunterabschnitt 5.3.2) implementiert. Die act_test Methode ist für die Aktionsbestimmung während der Testläufe zuständig und ermittelt 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

Damit eine Verbesserung der Performance eintritt, wird im Weiteren eine Trainingsroutine implementiert.

Trainingsroutine

Die learn Methode stellt die Trainingsroutine und damit das Herzstück eines Agenten dar. Sie wird entsprechend der Darstellungen im Unterunterabschnitt 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 Memorys aufgerufen, welche einen Mini-Batch liefert. Danach werden die Schritte entsprechend der Beschreibung im Konzept (siehe Unterunterabschnitt 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 Gleichung 2.11 und Unterunterabschnitt 5.3.2 dargestellt, Q-Target ($r(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$) bestimmt (siehe 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 Fehler des DQN mit dem MSE (Mean Squared Error) bestimmt und das Q-Net aktualisiert (siehe Code Listing 6.4).

```
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 Trainingslaufs zu senken.

Neben dem DQN Agenten wurde jedoch noch gefordert (siehe Unterabschnitt 3.2.3), dass ein PPO Agent implementiert wird. Dies geschieht im Folgenden.

6.4. PPO

Die Implementierung des PPO wurde inspiriert durch die Quellen (Barhate, 2021) und (Tabor, 2020).

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

Die Agent Klasse definiert die learn Methode und ist dabei als PPO-Komponente zu interpretieren (siehe Unterabschnitt 5.3.3). Die Memory Klasse speichert die gesammelten Erfahrungen. Die ActorNetwork bzw. CriticNetwork Klassen beinhaltet das NN des Actors bzw. Critics. In der ActorCritic Klasse befinden sich zentrale Methoden zur Durchführung des Lernens, wie z.B. evaluate. Des Weiteren verbindet die ActorCritic Klasse das Actor- und Critic-Network miteinander, welche im nächsten Abschnitt näher betrachtet werden.

6.4.1. Actor und Critic

Der Actor bzw. Critic wird durch die ActorNetwork bzw. CriticNetwork Klasse aufgespannt, welche von der Module Klasse des PyTorch Frameworks erbt. Damit lassen sich ActorNetwork und CriticNetwork als NN-Bausteine benutzen. Neben dem AV-Network (siehe Abschnitt 6.2 und Unterabschnitt 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. einen Value bestimmt. Exemplarisch wird die Klasse ActorNetworks im Code Listing 6.5 dargestellt. Analog verhält es sich beim Critic mit der Ausnahme, dass dieser nicht Tensoren der Länge drei, sondern eins ausgibt. Diese stellen die Baseline Estimate Werte (siehe Unterunterabschnitt 2.3.3) 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 Unterabschnitt 6.3.1 vorgestellt wurde. Sowohl ActorNetwork als auch CriticNetwork findet man jedoch nur gekapselt in der ActorCritic Klasse vor (siehe Unterabschnitt 6.4.2).

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 ActorNetwork und CriticNetwork ein gesamt Network aufspannen, welches alle Parameter der beiden Networks hält. Eine der wichtigsten Aufgaben der ActorCritic Klasse ist das Bestimmen von Aktionen. Dieser Prozess wird im Folgenden erklärt.

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 Unterunterabschnitt 5.3.3) implementiert.

Die act_test Methode verzichtet, wie beim DQN (siehe Unterunterabschnitt 6.3.3), wieder auf Zufallselemente. Am Ende der act_test Methode (siehe Code Listing 6.6) wird 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 Trainingsablauf unerlässlich ist.

Evaluate Methode

Die evaluate Methode bestimmt die logarithmierten Wahrscheinlichkeiten von Aktion unter einer anderen bzw. älteren Policy (Wahrscheinlichkeitsverteilung) (siehe Unterunterabschnitt 5.3.3). Neben dieser, bestimmt sie auch noch die Values des Critics und ermittelt die Entropies der Policy (siehe Code Listing 6.7).

```
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 Fehlers benötigt. Damit dieser bestimmt werden kann, bedarf es jedoch 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 speichern. 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 Spielepisoden 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 aktualisieren. Dieser Lernprozess geschieht dabei wie im folgenden Abschnitt dargestellt.

Trainingsroutine

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 dabei 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 Unterunterabschnitt 5.3.3). Diese implementiert die Funktionalität, welche im Unterunterabschnitt 2.3.3 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

Die 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 gespeicherten Aktionen. Mit diesen und den gespeicherten alten (old_log_probs) werden daraufhin die ratios gebildet (siehe Unterunterabschnitt 2.3.3).

Zuzüglich werden mithilfe der Values, welche von evaluate stammten, die Advantages aller Erfahrungen erstellt (siehe Unterunterabschnitt 2.3.3). Dies geschieht dabei wie im Unterunterabschnitt 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 Fehler (siehe Unterunterabschnitt 2.3.3) und anschließend den Actor Fehler zu bestimmen (siehe Gleichung 2.2).

```
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 Fehler

In dem Actor Fehler befindet sich zur einfacheren Handhabung noch der Entropy Fehler bzw. Bonus mitintegriert (siehe Unterunterabschnitt 2.3.3). Zur Bestimmung des gesamten Fehlers fehlt noch der Critic Fehler, welcher mit dem MSE (Mean Squared Error) bestimmt wird. Dabei wird der Fehler zwischen den Returns und den Values des Critics ermittelt. Danach werden Actor und Critic Fehler zusammenaddiert, sodass der PPO Fehler entsteht. Dieser wird dann dazu genutzt, um das Actor_Critic NN zu aktualisieren (siehe Code Listing 6.10).

```
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.

Damit jedoch die Lernprozedur aufgerufen wird, benötigt es eine zentrale Ablaufroutine, welche Agent und Env miteinander interagieren lässt. Diese wird in den train und test Methoden implementiert.

6.5. Train Methoden

Zu Beginn werden Agent und Env zusammen mit den Datenhaltungslisten (siehe Unterabschnitt 5.5.1) erstellt. Ein Scheduler wird für die Optimierung B (siehe Unterabschnitt 5.4.2) ebenfalls erzeugt. Danach wird die Trainingsprozedur durchgeführt, welche in Abbildung 6.3 dargestellt ist.

Zu Beginn wird eine Obs bestehend aus AV (around_view) und SO (scalar_obs) durch die reset Methode (siehe Unterabschnitt 5.2.2) generiert.

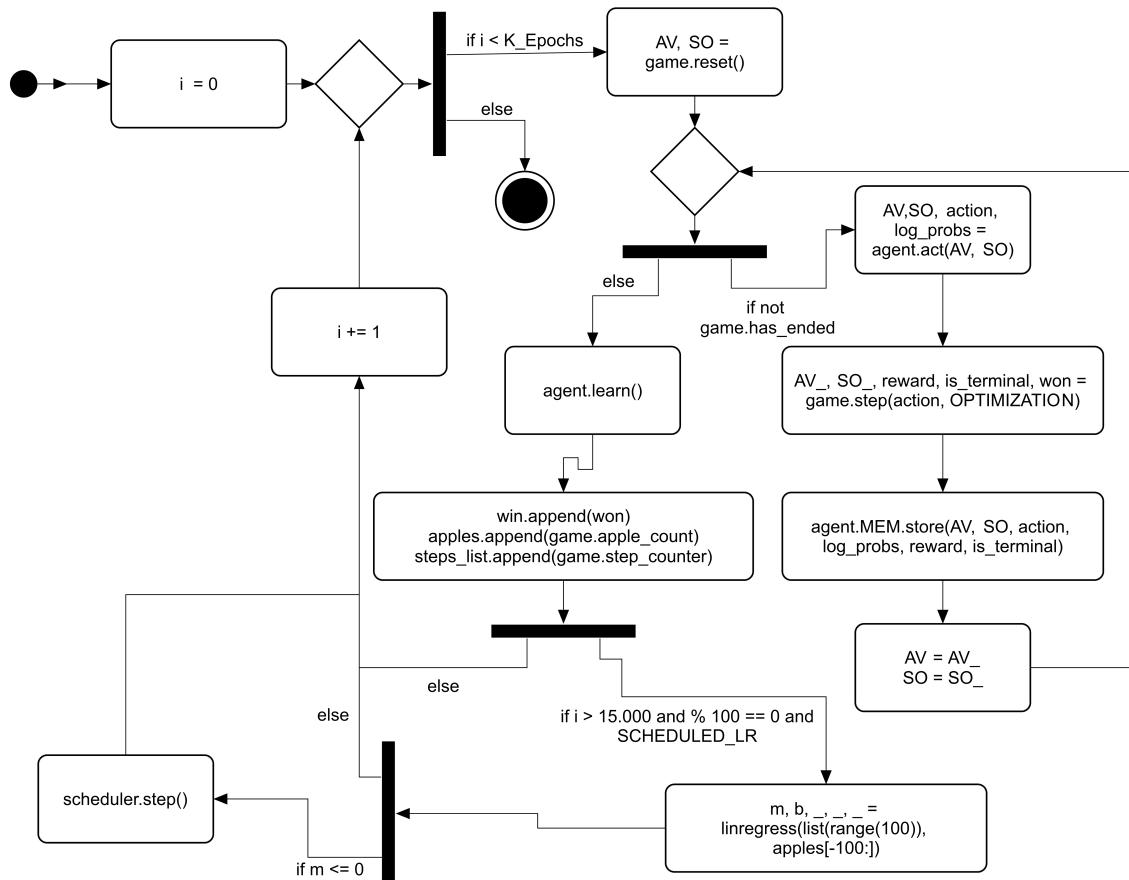


Abbildung 6.3.: Ablaufdiagramm der Train Methode.

Diese Obs wird dem Agenten übergeben, welcher mithilfe der act Methode eine Aktion und weitere benötigte Daten für den jeweiligen Algorithmus zurückgibt.

Stellvertretend für alle Algorithmen wird hier die Trainingsmethode des PPO für Beispiele herangezogen. Analog wird beim DQN verfahren. Danach wird die bestimmte Aktion mit der step Methode (siehe Unterabschnitt 5.2.2) im Env ausgeführt. Dabei kann mit dem OPTIMIZATION Hyperparameter gesteuert, welche Reward Funktion verwendet wird. Danach wird das Memory mit den entstandenen Erfahrungen aktualisiert und die alte Obs wird durch die neue ersetzt.

Sollte die Spieldaten terminieren, so wird die learn Methode des Agent aufgerufen (siehe Unterabschnitt 6.4.4 und Unterabschnitt 6.3.3) und die generierten Episodendaten werden in die Datenhaltungslisten eingefügt. 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.

Wird die Optimierung B verwendet, so wird alle 100 Spieldaten 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 Lernens umgesetzt werden, um die Lernrate nicht zu Beginn zu stark zu senken. Daher wird diese erst bei dem Überschreiten von 15.000 Epochs (Trainingsspielen) angepasst.

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

Neben der train Methode wird jedoch auch eine Testmethode zur Erstellung von Testdaten benötigt.

6.5.1. Test Methoden

Die Testmethoden 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 Speicherpfad 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 Unterabschnitt 5.5.2 und Abschnitt 3.5). Des Weiteren wird für die Aktionsbestimmungen nun die act_test Methoden verwendet und es kommt die Funktionalität der grafischen Umsetzung hinzu, indem die render Methode des Env aufgerufen wird, sofern die GUI in den Hyperparametern nicht ausgeschaltet worden ist (siehe Unterabschnitt 5.2.2 und Abbildung 6.1). Ansonsten treten keine Unterschiede zwischen den train und test Methoden auf.

6.6. Statistik

Die Statistiken werden mit der generate_statistic Funktion erstellt. Zuerst werden alle CSV-Dateien, welche die Test- und Trainingsdaten beinhalten, eingelesen. Dabei findet auch die Mittelung der Daten aus den verschiedenen Datenerhebungen (siehe Unterabschnitt 3.3.1) statt. Diese werden danach entsprechend der Darstellung im Abschnitt 5.5 bereitgestellt. Dazu wird pro Evaluationskriterium jedem Agenten ein Pandas Dataframe (siehe <https://pandas.pydata.org/>) zugeordnet, welcher die spezifischen Daten für dieses Kriterium besitzt.

Sollten also sechs Agenten verglichen werden, so werden vier Python dicts erstellt (da vier Evaluationskriterien existieren), welche als Key den Agentennamen und als Value den jeweiligen Dataframe für das Evaluationskriterium beinhalten. Die dicts besitzen alle also die Größe sechs.

Diese dicts werden dann jeweils der make_statistics Methode übergeben, wo die eigentlichen Statistiken generiert und gespeichert werden. Dies wird mithilfe des Matplotlib Frameworks (siehe <https://matplotlib.org/>) umgesetzt.

Kapitel 7

Evaluation

In diesem Kapitel 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 erläutert.

7.1. Ergebnisevaluation der Vergleiche

In der Evaluation sollen die Ergebnisse, welche aus den Vergleichen stammten, präsentiert werden.

7.1.1. Evaluation der Baseline Vergleiche

Basierend auf dem Vorgehen (siehe Abschnitt 5.1) wird mit den Baseline Vergleichen begonnen. Bei diesen handelt es sich um die Vergleiche, welche von nicht optimierten Agenten (Baseline Agenten) durchgeführt worden sind (siehe Abbildung 5.7)). Diese Agenten wurden in einem Trainingsverlauf, entsprechend der Beschreibung im Unterabschnitt 5.5.1, trainiert. Währenddessen wurden die Trainingsdaten erhoben, die grafisch dargestellt wurden. Als nächster Schritt wurden die Testdaten ermittelt, welche im Folgenden tabellarisch ausgewertet werden mit Ausnahme der Robustheit und Effizienz. Bei diesen bietet sich eine grafische Auswertung der Testdaten an.

Performance

Die Baseline Vergleichsauswertung soll mit der Performance beginnen. Dabei ist in der Abbildung 7.1 die durchschnittliche Performance bzw. Apfanzahl der letzten 200 Epochs pro Epoch 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 konnten, liegt in der Wahl von Zufallsaktionen. Die Komplexität des Spiels Snake steigt gegen

Ende immer weiter 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 zufällige Aktionen falsche Schritte durchgeführt werden könnten. Dadurch, dass das Spiel immer vorzeitig beendet würde, könnte der DQN Agent auch seine Leistung nicht weiter steigern, weil ihm die Daten zum Lernen fehlen würden.

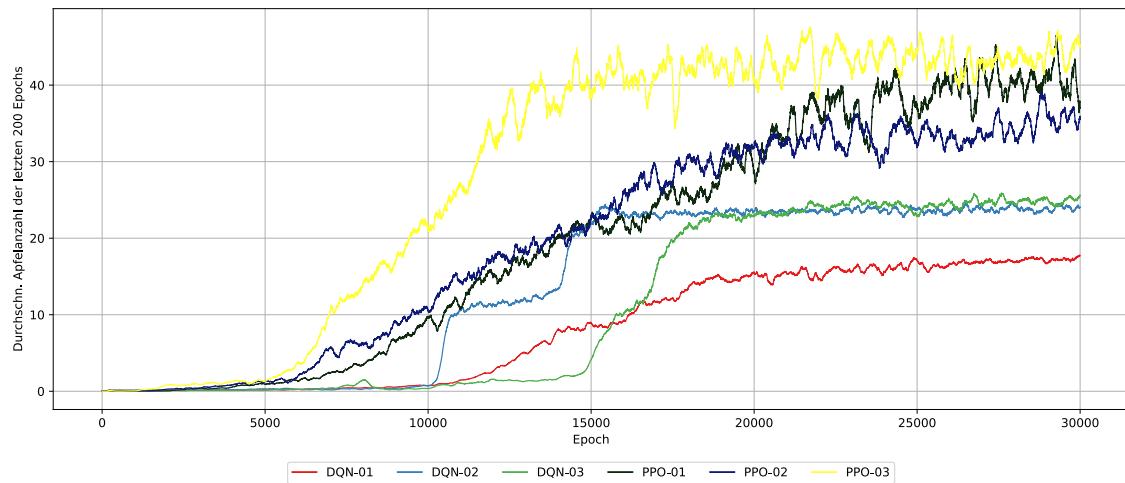


Abbildung 7.1.: Baseline Vergleich der Performance für die Trainingsdaten. Die Performance ist als Apfelanzahl pro Epoch definiert. Für einen besseren Kurvenverlauf wird die durchschn. Performance der letzten 200 Epochs abgebildet.

Im Training konnten besonders die Leistungen der PPO Agenten PPO-03 und PPO-01 überzeugen. Der PPO-03 kann dabei besonders mit seinem schnellen Lernerfolg punkten, wohingegen der PPO-01 mit seiner annähernd linearen Stetigkeit überzeugen kann. PPO-02 konnte zwar ebenfalls ein fast lineare Steigerung seiner Performance erzielen, jedoch konvergierte dieser früher als der PPO-01. Dieses Verhalten der Agenten entspricht den Darstellungen im Unterabschnitt 5.3.4.

Agent	Gemittelte Performance	Standardabweichung
DQN-01	17.7812	2.9577
DQN-02	26.4226	4.3482
DQN-03	25.4506	4.8280
PPO-01	44.4544	19.6957
PPO-02	38.7325	11.5756
PPO-03	46.4268	14.0280

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 Standardabweichungen des PPO-03 und PPO-01 zeigen jedoch, dass die Leistungen nicht konsistent sind. Besonders PPO-01 zeigte eine schwankende Performance, welche auch in den Trainingsdaten (siehe Abbildung 7.1) zu beobachten ist. Daher bleibt der PPO-03 für das Evaluationskriterium der Performance der Sieger.

Stark mit der Performance korrelierend, stellt die Siegrate das nächste zu untersuchende Evaluationskriterium dar.

Siegrate

Ähnlich wie bei der Performance verhält es auch mit dem Evaluationskriterium der Siegrate. PPO-03 und PPO-01 besitzen die besten Siegraten während des Trainings (siehe Abbildung 7.2).

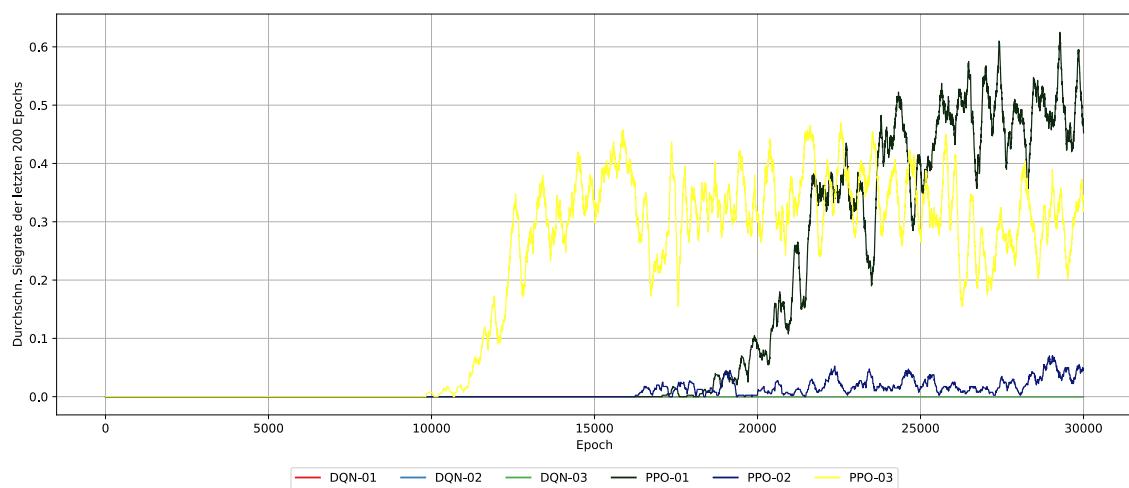


Abbildung 7.2.: Baseline Vergleich der durchschn. Siegrate. Für den besseren Kurvenverlauf wird die durchschn. Siegrate der letzten 200 Epochs abgebildet.

Die DQN Agenten sind nicht in der Lage gewesen Siege zu erreichen und fallen daher aus der Betrachtung heraus. Der PPO-02 zeigt eine deutlich geringe Siegrate, trotz ähnlicher Performances zu den anderen PPO Agenten. Bemerkenswert ist des Weiteren, dass der PPO-03 zwar ähnliche Leistungen wie der PPO-01 erreicht (siehe Abbildung 7.1), jedoch der PPO-01 eine deutlich bessere Siegraten erzielt. Dies ist möglicherweise auf den stetigen Lerncharakter des Agenten zurückzuführen (siehe Unterabschnitt 5.3.4). Auch die Testdaten in Tabelle 7.2 zeigen, dass PPO-01 der Sieger ist, wobei sich der eben beschriebene Trend aus den Trainingsdaten in den Testdaten widerspiegelt. Daher ist der PPO-01 der Sieger für das Evaluationskriterium der Siegrate.

Agent	Gemittelte Siegraten	Standardabweichung
PPO-01	0.6759	0.33306
PPO-02	0.0926	0.20478
PPO-03	0.3316	0.32751

Tabelle 7.2.: Testdatenauswertung der Baseline Siegrate

Robustheit

Die Robustheit stellt ein besonderes Evaluationskriterium dar, denn sie wird ausschließlich aus Testdaten bestimmt. Diese werden zur besseren Übersicht in eine Grafik überführt. Die Robustheit ist als durchschn. Apfelanzahl dividiert durch die Spielfeldgröße pro Quadratwurzel aus der Spielfeldgröße definiert.

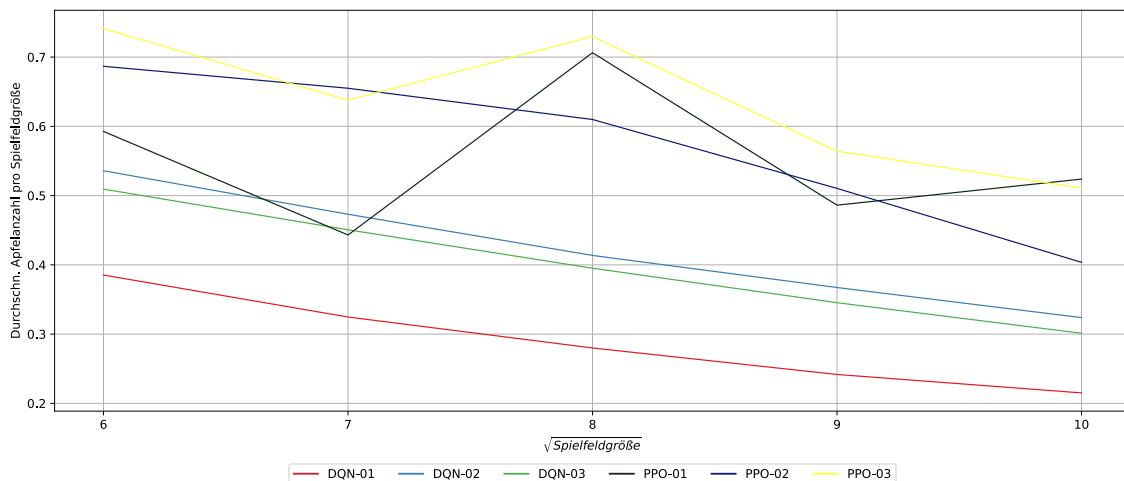


Abbildung 7.3.: Baseline Vergleich der durchschn. Robustheit. Diese wird als durchschn. Apfelanzahl dividiert durch die Spielfeldgröße pro Quadratwurzel aus der Spielfeldgröße dargestellt.

Wie in Abbildung 7.3 zu erkennen ist, zeigen die DQN Agenten keine guten Resultate. Sie 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 PPO Agenten zeigt sich ein ähnliches Bild. Ihre Leistungen steigen ebenfalls nicht mit dem sich vergrößernden Spielfeld an. Jedoch ist der PPO-03 in der Lage, die besten Ergebnisse in den unbekannten Gebieten zu erzielen.

Bemerkenswert ist des Weiteren, dass sich ein Trend abzeichnet, nachdem Agenten auf geraden Spielfeldgrößen (z.B. (6x6), (8x8) und (10x10)) bessere Leistungen erzielen können als auf ungeraden (z.B. (7x7) und (9x9)).

Aus diesem Vergleich geht dennoch der PPO-03 Agent als Sieger für das Evaluationskriterium der Robustheit hervor.

Effizienz

Die Effizienz stellt zusammen mit der Robustheit ein besonderes Evaluationskriterium dar. Sie ist als durchschn. Schrittanzahl pro Apfelanzahl definiert.

Wie in der Abbildung 7.4 zu erkennen ist, sind DQN-01 und DQN-02 diejenigen, welche die niedrigste Schrittanzahl für die jeweilige Apfelanzahl besitzt. Dies gilt jedoch nicht kontinuierlich. Denkbar wären daher die DQN Agenten in Einsatzgebieten mit wenigen Zielen aber großen Distanzen einzusetzen, sodass der Effizienzcharakter der Agenten hilft Kraftstoff bzw. Energie, am Beispiel der unbemannten Drohnen, zu sparen.

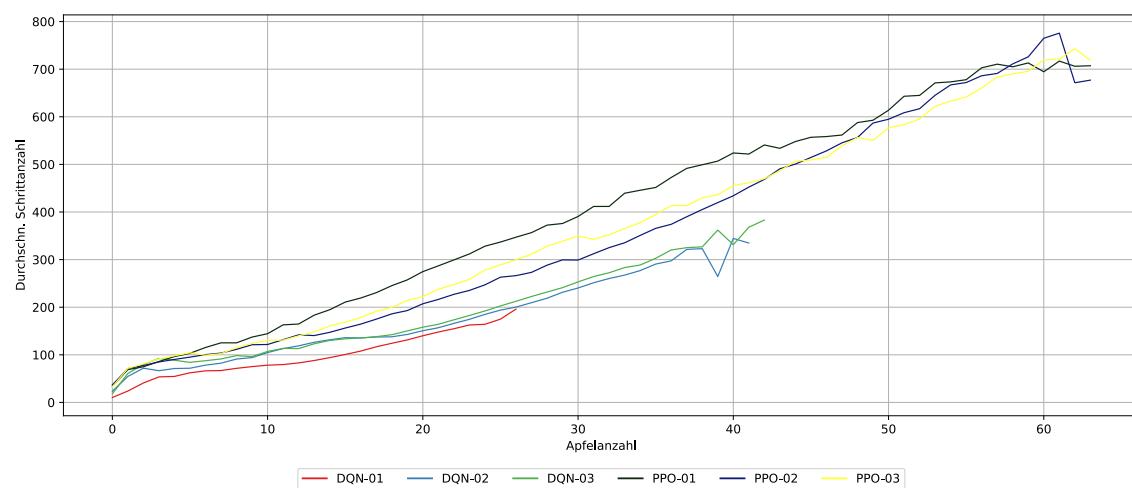


Abbildung 7.4.: Baseline Vergleich der Effizienz für die Trainingsdaten. Die Effizienz ist als durchschn. Schrittanzahl pro Apfelanzahl dargestellt.

Eine weitere Betrachtung der DQN Agenten unter dem Kriterium der Effizienz wird jedoch aufgrund der fehlenden Daten nicht durchgeführt. Da die DQN Agenten ausgeschieden sind, bleiben nur noch die PPO Agenten. Diese verfügen über den gesamten Trainingslauf eine solide Effizienz. Der PPO-02 konnte bei der Auswertung der Trainingsdaten die besten Ergebnisse erzielen, gefolgt von PPO-03. Erwähnenswert ist ebenfalls, dass die Effizienz-Differenz zwischen den Agenten PPO-01 bis PPO-03 gegeben Ende deutlich abnimmt, da durch die Gesetzmäßigkeiten des Spiels Snake, kaum effizientere Routen zum Apfel zu finden sind. Auch bei der Auswertung der Testdaten in Abbildung 7.5, wird dieser Trend deutlich.

Einzelnen Graphen in Abbildung 7.5 besitzen immer wieder Abschnitte, in welchen der Graph abbricht. Dies bedeutet, dass der Agent eine solche Anzahl an Äpfeln nie erreicht hat. Dies muss nicht negativ ausgewertet werden.

Die gute Effizienz des PPO-02 kann dabei unter anderen auf den niedrigen Gamma Wert von 0.93 zurückgeführt werden, welcher den Agenten dazu anhält, schnell viele gute Rewards zu erzielen (siehe Abschnitt ??).

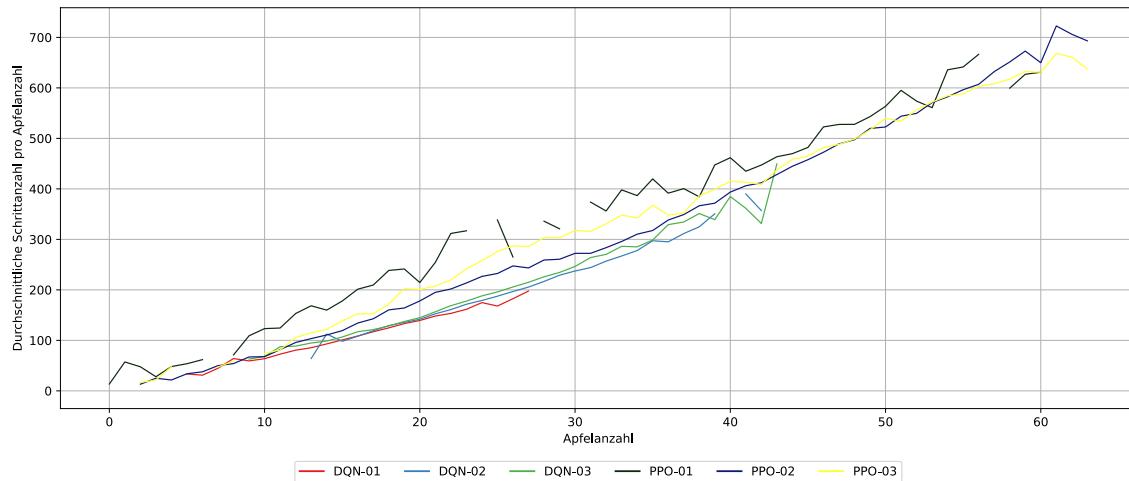


Abbildung 7.5.: Baseline Vergleich der Effizienz - Testdaten

7.1.2. Evaluation der Optimized Vergleiche

Nach der Durchführung der Baseline Vergleiche werden nun, entsprechend des Vorgehens (siehe Abschnitt 5.1), die Baseline Gewinner Agenten (siehe Abbildung ??) optimiert und dann untereinander verglichen. Dabei ist es auch möglich, sofern sich die Optimierungen nicht gegenseitig ausschließen, die Agenten mit mehr als nur einer Optimierung als zusätzlichen Parameter auszustatten. Dies wird jedoch in dieser Ausarbeitung, trotz der bestehenden Möglichkeit, nicht durchgeführt, um damit die Menge an Agenten nicht zu stark auszuweiten. Dies schafft eine bessere Übersichtlichkeit bei den Auswertungen.

Ebenfalls werden die Gewinner der Baseline Vergleiche mit in die Optimized Vergleiche eingebunden. Die Ergebnisse finden sich in den Folgenden Abschnitten.

Performance

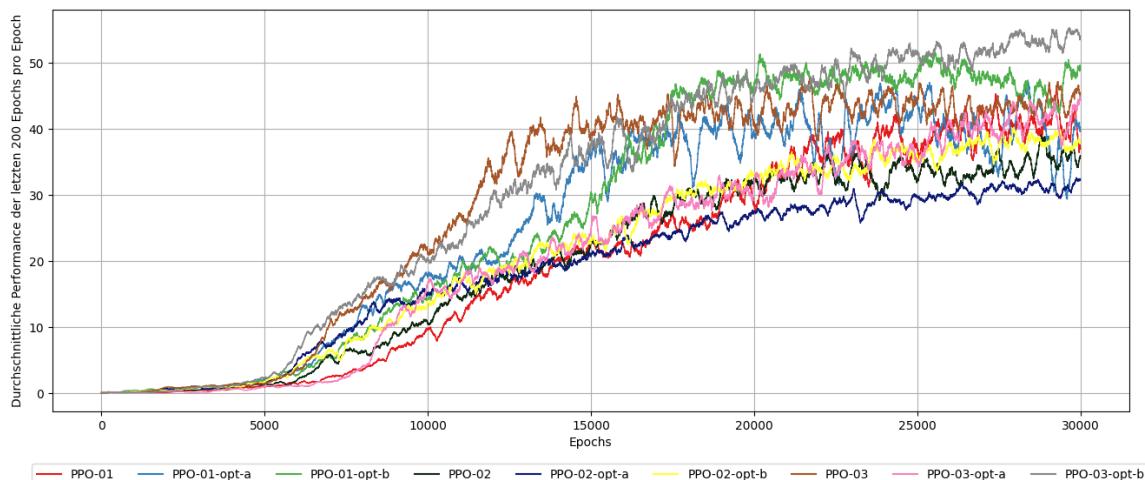


Abbildung 7.6.: Baseline Vergleich eins (oben) und zwei (unten) der Effizienz

Wie in Abbildung 7.6 zu erkennen ist, sind die Leistungen der einzelnen Agenten sehr unterschiedlich. Alle Agenten, welche auf dem PPO-02 aufbauen (Dunkelgrün, Dunkelblau und Gelb), konnten sich in diesem Vergleich nicht behaupten und schnitten schlechter als alle anderen Agenten ab. Dies könnte mit der Kurzzeitpräferenz, ausgelöst durch den niedrigen Gamma Wert, in Verbindung stehen. Die Ergebnisse der PPO-01 Agenten (Rot, Hellblau, Hellgrün) liegen im Mittelfeld, mit Ausnahme des PPO-01-opt-b, welcher das zweit beste Ergebnis erzielen konnte. Im oberen Mittelfeld sind die Agenten, welche von PPO-03 abstammen. Diese beinhalten auch den Gewinner bezüglich der Trainingsdaten. PPO-03-opt-b konnte seine Leistungen in den letzten 5.000 Trainingsspielen noch verbessern, im Gegensatz zum PPO-01-opt-b.

Insgesamt schnitten die Agenten mit der Optimierung B immer besser ab als die mit der Optimierung A (PPO-01-opt-a, PPO-02-opt-a, PPO-03-opt-a) und die nicht optimierten (PPO-01, PPO-02, PPO-03). Optimierung A stellte sich unter dem Gesichtspunkt der Performancesteigerung sogar als hinderlich heraus, da die Leistungen der optimierten Agenten entweder ungefähr gleich blieben (siehe die Agenten von PPO-01 und PPO-03) oder schlechter wurden (siehe die Agenten von PPO-02).

Agent	Durchschnittliche Performance	Standardabweichung
PPO-01	44.4544	19.6957
PPO-01-opt-a	45.8125	18.1183
PPO-01-opt-b	45.6213	18.7242
PPO-02	38.7325	11.5756
PPO-02-opt-a	34.8827	6.9273
PPO-02-opt-b	42.5969	12.4416
PPO-03	46.4268	14.0280
PPO-03-opt-a	48.7674	14.9580
PPO-03-opt-b	56.1117	12.3773

Tabelle 7.3.: Testdatenauswertung der Performance

Auch die Testdaten (siehe Tabelle 7.3) geht hervor, dass der PPO-03-opt-b der Sieger dieses Vergleiches ist. Mit einer Standardabweichung, welche sich insgesamt im Mittelfeld befindet (siehe Tabelle 7.3), von 12.3773 zeigt sich jedoch, das der PPO-03-opt-b dieser Leistung nicht kontinuierlich erzielen kann. Jedoch bleibt der PPO-03-opt-b, selbst unter Einbeziehung der Standardabweichung der Gewinner im Evaluationskriterium der Performance.

Als nächsten Evaluationskriterium wird die Sieg-Rate, entsprechend der Priorisierung (siehe Abschnitt 5.1), thematisiert.

Siegrate

Die Sieg-Rate spiegelt ein ähnliches Bild wieder wie bei der zuvor thematisierten Performance (siehe Abschnitt 7.1.2). Der PPO-03-opt-b konnte auch hier bei den Trainingsdaten die besten Ergebnisse zeigen.

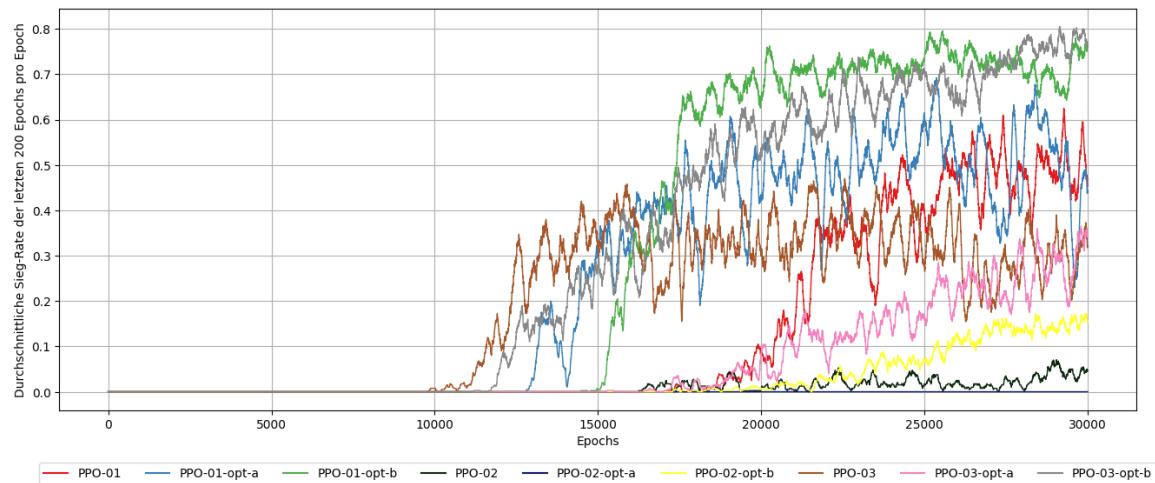


Abbildung 7.7.: Optimized Vergleich eins (oben) und zwei (unten) der Siegrate

Die PPO-02 Agenten konnten wieder keine guten Leistungen erreichen. Interessant ist die Tatsache, dass die PPO-03 Agenten, mit Ausnahme des PPO-03-opt-b, bei den Sieg-Raten eher im unteren Mittelfeld liegen und die Sieg-Raten der PPO-01 Agenten, mit Ausnahme des PPO-01-opt-b, im oberen Mittelfeld. Dies stellt einen gegensätzliches Verhalten zur Performance dar. Wie die Auswertung der Testdaten (siehe Tabelle 7.4) zeigt, ist der PPO-03-opt-b auch hier der gesamt Sieger. Mit einer Sieg-Rate von 0.8466 übertrifft er selbst den Zweitplatzierten PPO-01-opt-b, welcher nur eine Sieg-Rate von 0.7186 erreichte.

Agent	Durchschnittliche Sieg-Rate	Standardabweichung
PPO-01	0.6759	0.3331
PPO-01-opt-a	0.6551	0.3377
PPO-01-opt-b	0.7186	0.3011
PPO-02	0.0926	0.2048
PPO-02-opt-a	0.0001	0.0071
PPO-02-opt-b	0.2354	0.2496
PPO-03	0.3316	0.3275
PPO-03-opt-a	0.5273	0.3479
PPO-03-opt-b	0.8466	0.2482

Tabelle 7.4.: Testdatenauswertung der Optimized Sieg-Raten

Damit ist der PPO-03-opt-b der Sieger des Evaluationskriteriums der Sieg-Rate. Als nächstes folge die Betrachtung der Robustheit, entsprechend der Priorisierung (siehe Abschnitt 5.1).

Robustheit

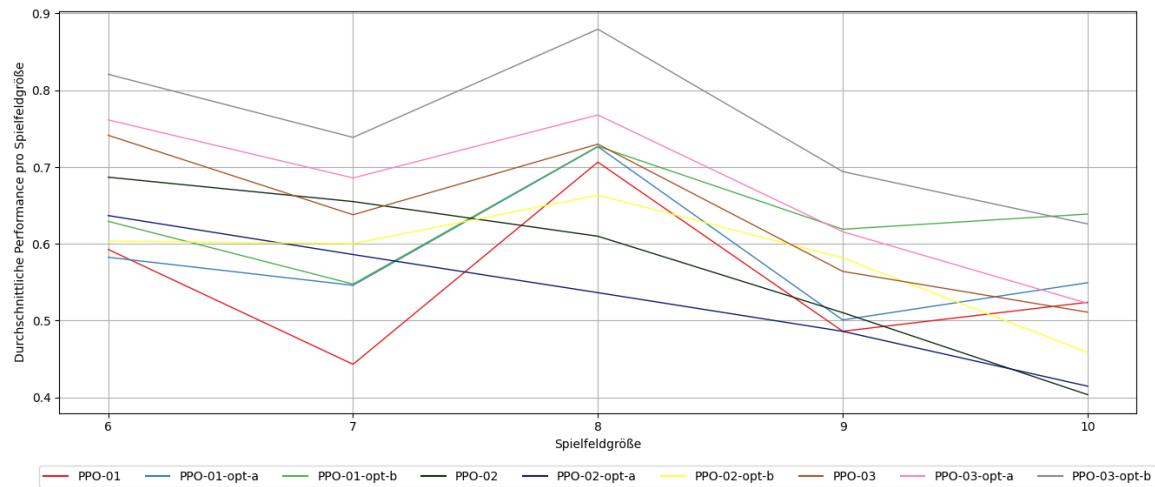


Abbildung 7.8.: Optimized Vergleich eins (oben) und zwei (unten) der Siegrate

Nach der Testauswertung (siehe Abbildung 7.8) hat sich der PPO-03-opt-b als der robusteste Agent dargestellt. Dabei zeigt die Abbildung 7.8 die durchschnittliche Performance für jede Spieldfeldgröße. Von allen Agenten wies er die kontinuierlichste Performance vor. Sowohl auf größeren als auch auf kleineren Spieldeldern zeigt er eine gute Leistung und damit eine solide Robustheit.

Es ist daher festzuhalten, dass der PPO-03-opt-b der Sieger für das Evaluationskriterium der Robustheit ist. Interessant ist des Weiteren, dass die Leistungen auf ungeraden Spieldfeldgrößen (7, 9) nicht so gut ist, wie auf geraden Spieldfeldgrößen (6, 8, 10). Es kann daher angenommen werden, dass die Strategie der Agenten in irgendeiner Form von der Spieldfeldgröße abhängig ist, obwohl diese dem Agenten nicht in der Observation direkt übergeben wird (siehe Abschnitt ??).

Effizienz

Die Abbildung 7.9 zeigt die durchschnittliche Schrittanzahl der einzelnen Agenten für jede Apfanzahl. Die Effizienz ist dabei als Schritte pro Apfel definiert. Wie in Abbildung 7.9 zu erkennen ist, benötigt der PPO-02-opt-a die wenigsten Schritte um die dargelegte Anzahl an Äpfeln zu sammeln. Dieser scheidet jedoch aufgrund seiner unzureichenden Daten aus. Der Sieger der Trainingsdaten ist damit der PPO-03-opt-a, gefolgt von dem PPO-03-opt-b. Insgesamt ist jedoch zu bemerken, dass alle Agenten ähnliche Effizienzen zeigen.

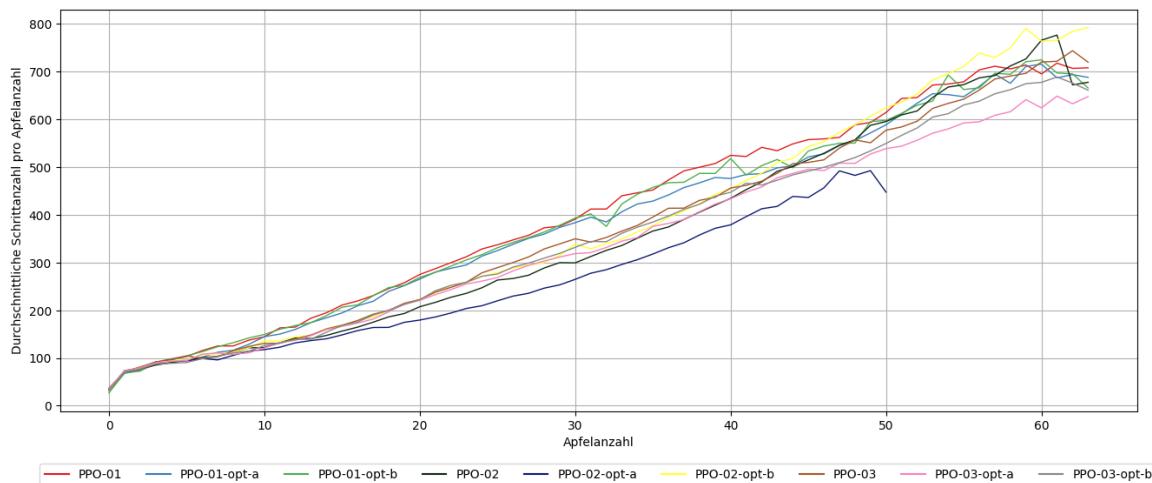


Abbildung 7.9.: Optimized Vergleich der Effizienz - Trainingsdaten

Die Auswertung der Testdaten zeigt ein ähnliches Bild (siehe Abbildung 7.10). Auch hier erweist sich der PPO-03-opt-a als der effizienteste, gefolgt von dem PPO-03-opt-b.

Wie bereits in Abschnitt 7.1.1 erwähnt, besitzen die einzelnen Graphen immer wieder Abschnitte, in welchen der Graph abbricht. Dies bedeutet, dass der Agent eine solche Anzahl an Äpfeln nie erreicht hat.

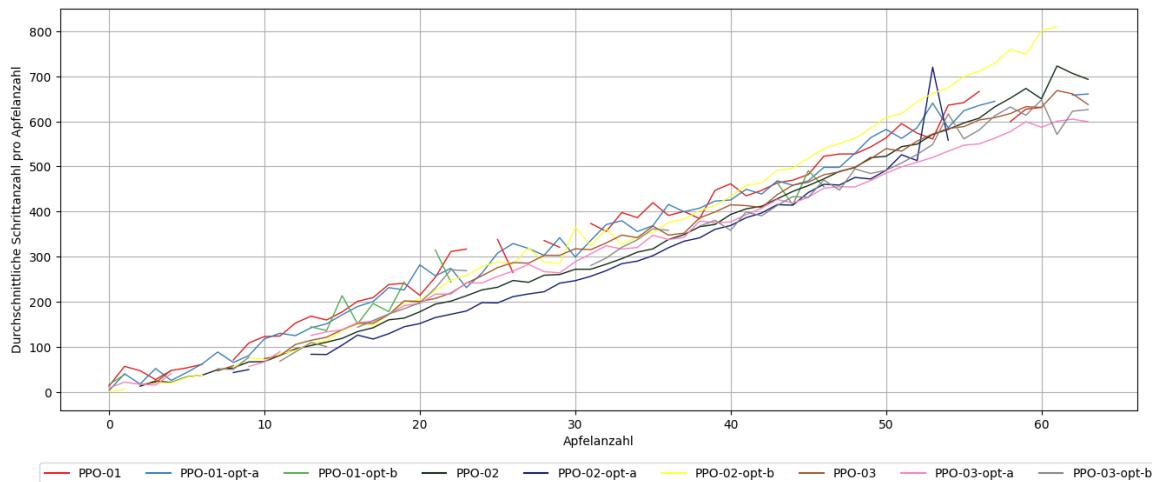


Abbildung 7.10.: Optimized Vergleich der Effizienz - Testdaten

Der Sieger dieses letzten Vergleichs ist der PPO-03-opt-a.

7.1.3. Bestimmung des optimalen Agenten

In diesem Abschnitt soll nun der optimale Agent ermittelt werden. Basierend auf den Ergebnissen der Vergleiche stellt sich heraus, dass der PPO-03-opt-b Agent der optimale Agent ist. Er liefert die besten Resultate in den Evaluationskriterien der Performance, Sieg-Rate und Robustheit. Einzig in der Effizienz belegt er den

zweiten Platz. Basierend auf der Priorisierung (siehe Abschnitt 5.1) ist damit der PPO-03-opt-b der gesamt Sieger.

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 Abschnitt 2.1 implementiert. Diese Anforderung kann daher als erfüllt angesehen werden. Eine Darstellung der Implementierung findet sich im Abschnitt des Konzepts (siehe Abschnitt 5.2) und in der Implementierung (siehe Abschnitt 6.1).

Ebenfalls wurde die Anforderung der Standardisierten Schnittstelle (siehe Abschnitt ??), welche zu einer Normung und damit zu einer einfacheren Benutzung des Environment führen soll, erfüllt. Wie in dem Abschnitt ?? 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 stan-dardisierte Schnittstelle auf.

Auch sind die funktionalen Anforderungen, welche den geregelten Ablauf im En-vironment garantieren, beachtet worden. So wird die Aktionsausführung (siehe Ab-schnitt ??) durch die action Methode in der SnakeGame Klasse durchgeführt, welche von der Schnittstellenmethode step aufgerufen wird. Die reset und render Anforde-rungen (siehe Abschnitte ?? und ??) werden durch die gleichnamigen Schnittstel-lenmethode abgedeckt (siehe Abschnitte ?? und 6.1).

7.2.2. Anforderungsevaluation der Agenten

Der nächste großer Anforderungsbereich behandelt die Agenten. Zu diesem Zweck wurden die Anforderungen der Aktionsbestimmung und des Lernens aufgestellt. Die-se stellen die grundlegenden Funktionen der Agenten dar. Wie im Konzept darge-stellt ist, wurde sowohl der DQN als auch der PPO Agent mit einer Aktionsauswahl-methode (siehe Abschnitte ?? und ??) und einer learn Methode (siehe Abschnitte

?? und ??) ausgestattet. Diese implementieren das geforderte Verhalten aus den Anforderungen (siehe Abschnitt ??).

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

Diese fordert den Vergleich verschiedener Algorithmen und dabei insbesondere des PPO und DQN Algorithmus. Mit dieser Forderung kann die entwickelte Methodik (siehe Abschnitt 5.1) besser bewertet werden. Diese Anforderung kann ebenfalls als erfüllt angesehen werden, da sowohl ein DQN (siehe Abschnitt ??) als auch ein PPO Agent (siehe Abschnitt 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 Abschnitt 7.1) und in dem Konzept (siehe Abschnitt ??) 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 Abschnitt ??). Dies soll die Validität der statistischen Untersuchung steigern. Wie im Abschnitt der Datenerhebung (siehe Abschnitt ??) 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 Abschnitt ??). Dies wurde darüber hinaus in dem Abschnitt der Datenerhebung des Konzepts (siehe Abschnitt ??) und in der Implementierung (siehe Abschnitt 6.5) dargestellt.

7.2.4. Anforderungen an die Statistiken

Insgesamt sind die Anforderungen zu den Statistiken (siehe Abschnitt 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 6.6 beweisen. Auch die Grafiken in der Evaluation der Ergebnisse (siehe Abschnitt 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 beantwortet wurde. Anschließend wird ein Ausblick auf weitere Schritte in diesem Forschungsgebiet gegeben.

8.1. Beantwortung 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, Sieg-Rate 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). Bei der Erstellung der Agenten besitzt der Anwender prinzipiell eine große Freiheit, nach welchen Gesichtspunkten er die Agenten definiert. In dieser Ausarbeitung wurden die Hyperparameter so ausgewählt das stets ein langsamer, ein schneller und ein Hybrid aus den vorher erwähnten Agent erstellt wurde. Dies sollte eine große Bandbreite an Möglichkeiten abdecken.

In dieser wurden die vordefinierte Agenten erst mehreren Baseline verglichen 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, entsprechend 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.

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

[Barhate 2021] BARHATE, Nikhil: *Minimal PyTorch Implementation of Proximal Policy Optimization.* <https://github.com/nikhilbarhate99/PPO-PyTorch>. 2021

[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] MNIIH, 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>
- [Tabor 2020] TABOR, Phil: *Youtube-Code-Repository*. <https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PP0/torch>. Dezember 2020
- [Wei u. a. 2018] WEI, Zhepei ; WANG, Di ; ZHANG, Ming ; TAN, Ah-Hwee ; MIAO, Chunyan ; ZHOU, You: Autonomous Agents in Snake Game via Deep Reinforcement Learning. In: *2018 IEEE International Conference on Agents (ICA)*, 2018, S. 20–25

Anhang A

Anhang

A.1. Backpropagation und das Gradientenverfahren

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

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

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

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

A.2. Tensoren

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

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

A.3. Convolution Neural Networks

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

A.3.1. Convolutional Layer

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

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

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

Die Funktionsweise der Convolutionalen Prozedur stellt sich wie folgt dar:

Jeder einzelne Kernel wird mit der Eingabe entsprechend A.1 multipliziert und addiert. Ist dieser Berechnungsschritt abgeschlossen, bewegt sich das 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.

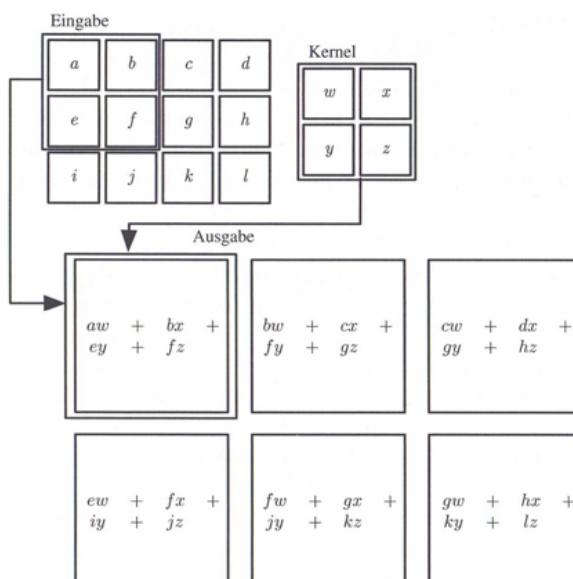


Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

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.)

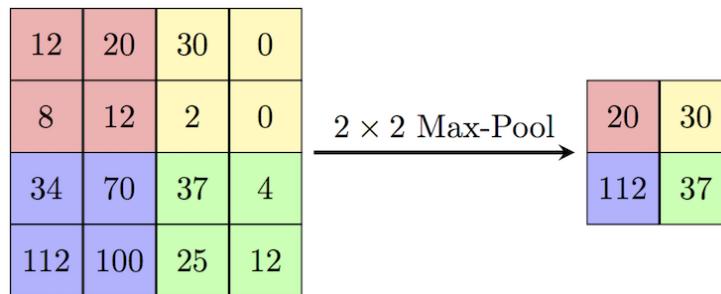


Abbildung A.2.: Darstellung des Max-Poolings. <https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png>

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 ?? entspricht. Sollte ein Merkmal erkannt worden sein, wie z.B. die Position des Apfels, so wird diese in den dazugehörigen Channel eingepflegt.

Code Listing B.1:

```
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 List namens grad_list erstellt. In dieser werden Faktoren gehalten, welche den Grad der Sucherlinie (siehe ??) angeben. Dabei entsprechen die Faktoren den Himmels- bzw. Nebenhimmelsrichtungen in der folgenden Reihenfolge ($0^\circ = \text{N}$, $45^\circ = \text{NO}$, $90^\circ = \text{O}$, $135^\circ = \text{SO}$, $180^\circ = \text{S}$, $225^\circ = \text{SW}$ und $270^\circ = \text{W}$, $315^\circ = \text{NW}$).

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.

Code Listing B.2:

```

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.

Code Listing B.3:

```
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.

Code Listing B.4:

```
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.5: 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 ?? 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 ??
7. EPS_CLIP (Float) → Clip Faktor, welcher St Standard bei 0.2. Siehe ??
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 ??
6. MAX_MEM_SIZE (Int) → Maximale Größe des Memory.
7. EPS_DEC (Float) → Der Absenkungsfaktor von Epsilon ??
8. EPS_END (Float) → Der Endwert von Epsilon ??
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 ??), B (siehe ??) 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