

CARL VON OSSIEZKY UNIVERSITÄT OLDENBURG

WIRTSCHAFTSINFORMATIK  
BACHELORARBEIT

---

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

---

*Autor:*  
Lorenz Mumm

*Erstgutachter:*  
Apl. Prof. Dr. Jürgen Sauer

*Zweitgutachter:*  
M. Sc. Julius Möller

Abteilung Systemanalyse und -optimierung  
Department für Informatik

Oldenburg, 20. August 2021

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Tabellenverzeichnis</b>	<b>vi</b>
<b>Abkürzungsverzeichnis</b>	<b>vii</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Zielsetzung . . . . .	2
<b>2. Grundlagen</b>	<b>3</b>
2.1. Game of Snake . . . . .	3
2.2. Reinforcement Learning . . . . .	4
2.2.1. Vokabular . . . . .	5
2.2.2. Funktionsweise . . . . .	8
2.2.3. Arten von RL-Verfahren . . . . .	8
2.3. Proximal Policy Optimization . . . . .	10
2.3.1. Actor-Critic Modell . . . . .	11
2.3.2. PPO Training Objective Function . . . . .	11
2.3.3. PPO - Algorithmus . . . . .	16
2.4. Deep Q-Network . . . . .	17
2.4.1. DQN - Algorithmus . . . . .	18
<b>3. Anforderungen</b>	<b>20</b>
3.1. Anforderungen an das Environment . . . . .	20
3.1.1. Standardisierte Schnittstelle . . . . .	20
3.1.2. Funktionalitäten . . . . .	20
3.2. Anforderungen an die Agenten . . . . .	21
3.2.1. Funktionalitäten . . . . .	21
3.2.2. Parametrisierung . . . . .	21
3.2.3. Diversität der RL Algorithmen . . . . .	22
3.3. Anforderungen an die Datenerhebung . . . . .	22
3.3.1. Mehrfache Datenerhebung . . . . .	22
3.3.2. Datenspeicherung . . . . .	22

3.4. Anforderungen an die Statistiken . . . . .	23
3.5. Anforderungen an die Evaluation . . . . .	23
<b>4. Verwandte Arbeiten</b>	<b>25</b>
4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning	25
4.1.1. Diskussion . . . . .	26
4.2. UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene . . . . .	27
4.2.1. Diskussion . . . . .	28
4.3. Zusammenfassung . . . . .	29
<b>5. Konzept</b>	<b>30</b>
5.1. Vorgehen . . . . .	30
5.2. Environment . . . . .	31
5.2.1. Wrapper . . . . .	32
5.2.2. Spiellogik . . . . .	32
5.3. Agenten . . . . .	39
5.4. Optimierungen . . . . .	39
5.4.1. Optimierung 02 . . . . .	39
5.5. Datenerhebung . . . . .	39
5.6. Vergleich . . . . .	39
<b>6. Agenten</b>	<b>40</b>
6.1. Agenten . . . . .	40
<b>7. Implementierung</b>	<b>42</b>
7.1. Snake Environment . . . . .	42
7.1.1. Schnittstelle . . . . .	42
7.1.2. Spiellogik . . . . .	43
7.1.3. Graphische Oberfläche . . . . .	45
7.2. Agenten . . . . .	45
7.2.1. Netzstruktur . . . . .	45
7.2.2. Testung der Agenten . . . . .	47
7.2.3. DQN . . . . .	47
7.2.4. PPO . . . . .	49
<b>8. Optimierungen</b>	<b>52</b>
8.1. Optimierung 1 - Dual Experience Replay . . . . .	52
8.2. Optimierung 2 - Joined Reward Function . . . . .	53
8.3. Optimierung 3 - Odor Reward Function & Loop Storm Strategy . . .	54
8.4. Optimierung 4 - Dynamische Lernrate . . . . .	54

---

<b>9. Evaluation</b>	<b>55</b>
<b>Literaturverzeichnis</b>	<b>56</b>
<b>A. Anhang</b>	<b>58</b>
A.1. Backpropagation und das Gradientenverfahren . . . . .	58
A.2. Tensoren . . . . .	59
A.3. Convolution Neural Networks . . . . .	59
A.3.1. Convolutional Layer . . . . .	59
A.3.2. Pooling Layer . . . . .	61
A.3.3. Fully Connected Layer . . . . .	62
<b>B. Anleitung</b>	<b>63</b>
B.1. PPO Train Startargumente . . . . .	63
B.2. DQN Train Startargumente . . . . .	64
B.3. Play Startargumente . . . . .	65

# Abbildungsverzeichnis

2.1. Game of Snake . . . . .	3
2.2. Reinforcement Learning . . . . .	8
5.1. Flussdiagramm des Vorgehens . . . . .	31
5.2. Wrapper . . . . .	32
5.3. Spiellogik . . . . .	33
5.4. Spielablauf . . . . .	34
5.5. Observation . . . . .	37
7.1. BaseNet . . . . .	46
A.1. Darstellung Convolutional Computation . . . . .	61
A.2. Darstellung MaxPooling . . . . .	61

# Tabellenverzeichnis

2.1. Formelemente . . . . .	12
3.1. Zu erhebende Daten . . . . .	22
3.2. Evaluationskriterien . . . . .	23
5.1. Kodierung der Actions . . . . .	34
5.2. Channel-Erklärung der Around_View (AV) . . . . .	36
7.1. Methoden der SnakeGame Klasse . . . . .	43

# 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

# Kapitel 1

## Einleitung

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

### 1.1. Motivation

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

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



mus (DQN) und der Double Deep Q-Network Algorithmus (DDQN). Die Wahl des passenden Agenten kann großen Einfluss auf die Performance und andere Bewertungskriterien haben, (Bowe Ma (2016)), deshalb soll in dieser Ausarbeitung ein Vorgehen, welches auf einem Vergleich beruht, entwickelt werden, dass 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 Spieles ist zusätzlich zu dem oben erwähnten Mehrwert noch ein weiterer in Erscheinung getreten.

So interpretieren neue Forschungsansätze das Spiel Snake als ein dynamisches Path-finding Problem. Mit dieser Art von Problemen sind auch unbemannte Drohne (UAV Drohnen) konfrontiert, welche beispielsweise Menschen in komplexen Katastrophensituationen, wie z.B. auf explodierten Rohölbohrplattformen oder Raffinerien, finden und retten sollen. Auch kann das Liefern von wichtigen Gütern, beispielsweise medizinischer Natur, in solche Gebiete kann durch die Forschung am Spiel Snake möglich gemacht werden. (Chunxue u. a. (2019))

## 1.2. Zielsetzung

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

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

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

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

# Kapitel 2

## Grundlagen

Im folgenden Kapitel soll das benötigte Wissen vermittelt werden, welcher zum Verständnis dieser Arbeit benötigt wird. Dabei sollen verschiedene Reinforcement Learning Algorithmen, wie auch grundlegende Informationen des Reinforcement Learnings selbst thematisiert werden. Auch das Spiel Snake wird Erwähnung finden.

### 2.1. Game of Snake

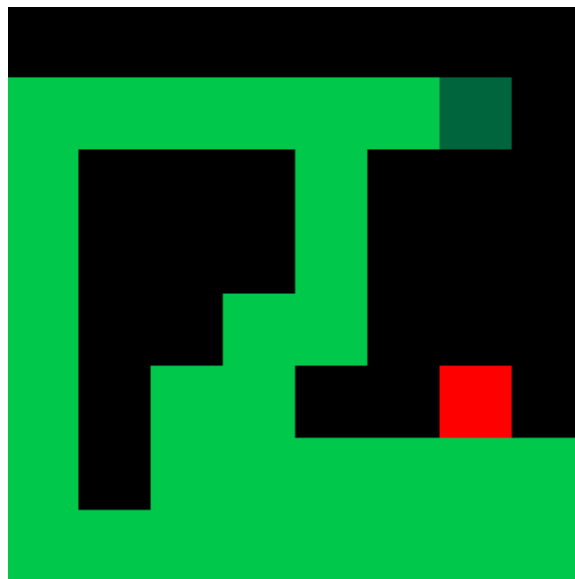


Abbildung 2.1.: Game of Snake - Abbildung eines Snake Spiels in welchem der Apfel durch das rote und der Snake Kopf durch das dunkelgrüne Quadrat dargestellt wird. Die hellgrünen Quadrate stellen den Schwanz der Snake dar.

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

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

## 2.2. Reinforcement Learning

Das Reinforcement Learning (Bestärkendes Lernen) ist einer der drei großen Teilbereiche, die das Machine Learning zu bieten hat. Neben dem Reinforcement Learning zählen das supervised Learning (Überwachtes Lernen) und das unsupervised Learning (unüberwachtes Lernen) ebenfalls noch zum Machine Learning.

Einordnen lässt sich das Reinforcement Learning (RL) irgendwo zwischen vollständig überwachtem Lernen und dem vollständigen Fehlen vordefinierter Labels (Lapan, 2020, S. 26). Viele Aspekte des (SL), wie z.B. neuronale Netze zur Approximation einer Lösungsfunktion, Gradientenabstiegsverfahren und Backpropagation zur Erlernung von Datenrepräsentationen, werden auch im RL verwendet.

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

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

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

### 2.2.1. Vokabular

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

#### Agent

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

Bei den Agenten handelt es sich gewöhnlich um die einzigen Instanzen, welche mit dem Environment (der Umgebung) interagieren. Zu dieser Interaktionen zählen das Entgegennehmen von Observations und Rewards, wie auch das Tätigen von Actions. (Sutton und Barto, 2018, S. 2ff.)

#### Environment

Das Environment (Env.) bzw. die Umgebung ist vollständig außerhalb des Agenten angesiedelt. Es spannt das zu manipulierende Umfeld auf, in welchem der Agent Interaktionen tätigen kann. An ein Environment werden unter anderem verschiedene Ansprüche gestellt, damit ein RL Agent mit ihm in Interaktion treten kann. Zu diesen Ansprüchen gehört unter anderem die Fähigkeit Observations und Rewards zu liefern, Actions zu verarbeiten. (Lapan, 2020; Sutton und Barto, 2018, S. 31 & S.2 ff.)

Actions, welche im momentanen State des Env. nicht gestattet sind, müssen entsprechend behandelt werden. Dies wirft die Frage auf, ob es in dem Env. einen terminalen Zustand (häufig done oder terminal genannt) geben soll. Existiert ein solcher terminaler Zustand, so muss eine Routine für den Reset (Neustart) des Env. implementiert sein.

#### Action

Die Actions bzw. die Aktionen sind einer der drei Datenübermittlungswege. Bei ihnen handelt es sich um Handlungen welche im Env. ausgeführt werden. Actions können z.B. erlaubte Züge in Spielen, das Abbiegen im autonomen Fahren oder das

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

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

### Observation

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

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

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

### Reward

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

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

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

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

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

## State

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

## Policy

Informell lässt sich die Policy als eine Menge an Regeln beschreiben, welche das Verhalten eines Agenten steuern. Formal ist die Policy  $\pi$  als eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen  $a$  im State  $s$  des Env. definiert. (Lapan, 2020, S. 44)

Sollte daher ein Agent der Policy  $\pi_t$  zum Zeitpunkt  $t$  folgen, so ist  $\pi_t(a_t|s_t)$  die Wahrscheinlichkeit, dass die Aktion  $a_t$  im State  $s_t$  unter den stochastischen Aktionswahlwahrscheinlichkeiten (Policy)  $\pi_t$  zum Zeitpunkt  $t$  gewählt wird. (Sutton und Barto, 2018, S. 45 ff.)

## Value

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

Bei einem Verfahren wie z.B. dem Q-Learning lässt sich die Policy formal angeben:  $\pi(s) = \arg \max_a Q_\pi(s, a)$ . Dies ist die Auswahlregel der Actions  $a$  im State  $s$ . (Lapan, 2020, S.291)

### 2.2.2. Funktionsweise

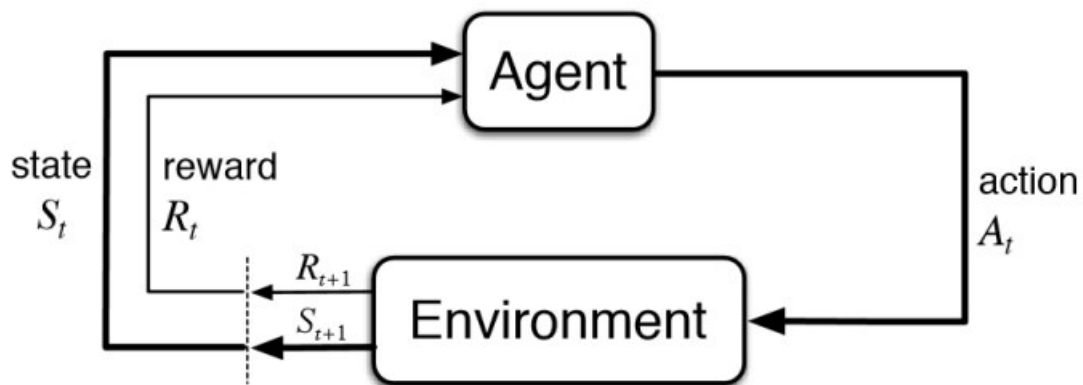


Abbildung 2.2.: Reinforcement Learning schematische Funktionsweise - Der Agent erhält einen State  $S_t$  und falls  $t \neq 0$  einen Reward  $R_t$ . Daraufhin wird vom Agenten eine Action  $A_t$  ermittelt, welche im Environment ausgeführt wird. Das Env. übermittelt den neu entstandenen State  $S_{t+1}$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

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

### 2.2.3. Arten von RL-Verfahren

Nach dem nun das Basisvokabular weitestgehend erklärt wurde, soll nun noch ein tieferer Blick in die verschiedenen Arten der RL geworfen werden.

Alle RL Verfahren lassen sich, unter gewissen Gesichtspunkten, in Klassen einordnen, welche Aufschluss über die Implementierung, den Entscheidungsfindungsprozess und die Datennutzung geben. Natürlich existieren noch viele weitere Möglichkeiten

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

### **Model-free und Model-based**

Die Unterscheidung in model-free (modellfrei) und in model-based (modellbasiert) gibt Aufschluss darüber, ob der Agent fähig ist, ein Modell des Zustandekommens der Belohnungen (Reward) zu entwickeln.

Model-free RL Verfahren sind nicht in der Lage das Zustandekommen der Belohnung vorherzusagen, vielmehr ordnen sie einfach die Beobachtung einer Aktion oder einem Zustand zu. Es werden keine zukünftigen Beobachtungen und/oder Belohnungen extrapoliert. (Lapan, 2020; Sutton und Barto, 2018, S. 303 ff. / S. 100 )

Ganz anderes sieht es da bei den model-based RL-Verfahren aus. Diese versuchen eine oder mehrere zukünftige Beobachtungen und/oder Belohnungen zu ermitteln, um die beste Aktionsabfolge zu bestimmen. Dem Model-based RL-Verfahren liegt also ein Planungsprozess der nächsten Züge zugrunde. (Sutton und Barto, 2018, S. 303 ff.)

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

### **Policy-Based und Value-Based Verfahren**

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

Basierend auf dieser Wahrscheinlichkeitsverteilung  $\pi$  wird die nächste Action  $a$  be-



stimmt. Dabei ist es offensichtlich, dass nicht immer die optimalste Action gewählt wird.

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

### On-Policy und Off-Policy Verfahren

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

On-Policy-Verfahren sind dagegen sehr wohl abhängig von aktuellen Daten, da sie versuchen die Policy indirekt oder direkt zu optimieren.

Auch hier besitzen beide Klassen ihre Vor- und Nachteile. So können beispielsweise Off-Policy Verfahren mit älteren Daten immer noch trainiert werden. Dies macht Off-Policy RL Verfahren Daten effizienter als On-Policy Verfahren. Meist ist es jedoch so, dass diese Art von Verfahren langsamer konvergieren.

On-Policy Verfahren konvergieren dagegen meist schneller. Sie benötigen aber dafür auch mehr Daten aus dem Environment, dessen Beschaffung aufwendig und teuer sein könnte. Die Dateneffizienz nimmt ab. (Lapan, 2020, S. 210 f.)

## 2.3. Proximal Policy Optimization

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

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

### 2.3.1. Actor-Critic Modell

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

Beide NN können aus mehreren Schichten bestehen, jedoch sind Actor und Critic streng von einander getrennt und teilen keine gemeinsamen Parameter. Gelegentlich werden den beiden Netzen (Actor bzw. Critic) noch ein weiteres Netz vorgeschoben. In diesem Fall können Actor und Critic gemeinsame Parameter besitzen. Das Actor- bzw. Policy-Network ist für die Bestimmung der Policy zuständig. Anders als bei Value-based RL-Verfahren wird diese direkt bestimmt und kann auch direkt angepasst werden. Die Policy wird als eine Wahrscheinlichkeitsverteilung über alle möglichen Actions vom Actor-NN zurückgegeben. 2.2.1

Das Critic- bzw. Value-Network evaluiert die Actions, welche vom Actor-Network bestimmt worden sind. Genauer gesagt, schätzt das Value-Network die sogenannte "Discounted Sum of Rewards" zu einem Zeitpunkt  $t$ , basierend auf dem momentanen State  $s$ , welcher dem Value-Network als Input dient. "Discounted Sum of Rewards" wird im späteren Verlauf noch weiter vorgestellt und erklärt.

### 2.3.2. PPO Training Objective Function

Nun da einige Grundlagen näher beleuchtet worden sind, ist das nächste Ziel die dem PPO zugrunde liegende mathematische Funktion zu verstehen, um im späteren eine eigene Implementierung des PPO durchführen zu können und um einen objektiveren Vergleich der zwei RL-Verfahren durchführen zu können.

Der PPO basiert auf den folgenden mathematischen Formel, welche den Loss eines Updates bestimmt (Schulman u. a., 2017, S. 5):

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

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

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

## Formelelemente

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

Tabelle 2.1.: Formelelemente

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

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

### Return

Der Return  $R_t$  stellt dabei die Summe der Rewards in der gesamten Spielepisode von dem Zeitpunkt  $t$  an dar. Diese kann ermittelt werden, da alle Rewards, durch das Sammeln von Daten, bereits bekannt sind. Des Weiteren werden die einzelnen Summanden mit einem Discount Factor  $\gamma$  multipliziert, um die Zeitpräferenz des Agenten besser zu steuern. Gamma liegt dabei gewöhnlich zwischen einem Wert von 0.9 bis 0.99. Kleine Werte für Gamma sorgen dafür, dass der Agent langfristig eher dazu tendiert Actions bzw. Aktionen zu wählen, welche unmittelbar zu positiven Reward führen. Entsprechend verhält es sich mit großen Werten für Gamma. (Sutton und Barto, 2018, S. 42 ff.)

### Baseline Estimate

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

### Advantage

Der erste Funktionsbestandteil des  $L_t^{\text{CLIP}}(\theta)$  2.2 behandelt den Advantage  $\hat{A}_t(s, a)$ . Dieser wird durch die Subtraktion der Discounted Sum of Rewards bzw. des Return  $R_t$  und dem Baseline Estimate  $b(s_t)$  bzw. der Value-Function berechnet. Die folgende Formel ist eine zusammengefasste Version der original Formel aus Schulman u. a.

(2017):

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

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

### Probability Ratio

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

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

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

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

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

Nun können generierte Daten mehrfach für Updates der Policy genutzt werden, was die Menge an Daten, welche nötig ist, um ein gewisses Ergebnis zu erreichen, minimiert. Der PPO Algorithmus ist durch die Umstellung auf die Probability Ratio effizienter in der Nutzung von Daten geworden.

## Surrogate Objectives

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

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

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

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

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

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

## Zusammenfassung der PPO Training Objective Function

Insgesamt ist der Actor-Loss 2.2 ein Erwartungswert, welcher sich empirisch über eine Menge an gesammelten Daten ergibt. Dies wird durch  $\hat{\mathbb{E}}_t$  impliziert. Dieser setzt sich aus vielen einzelnen Losses zusammen. Diese sind das Minimum der beiden Surrogate Objective Functions zusammen. Dies sorgt dafür, dass keine zu großen Losses die Policy zu weit von dem Entstehungspunkt der Daten wegführen (Trust Region). (Schulman u. a., 2017, S. 3f.)

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

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

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

Es ergibt sich damit der bereits oben erwähnte PPO Training Loss 2.1:

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

### 2.3.3. PPO - Algorithmus

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

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für Actor  $\theta$  und Critic  $w$  zufallsbasiert. Erstelle ein Experience Buffer  $EB$ .
2. Bestimmt mit dem State  $s$  und dem Actor-NN eine Action  $a$ . Dies geschieht durch  $\pi_\theta(s)$ .
3. Führe Action  $a$  aus und ermittle den Reward  $r$  und den Folgezustand  $s'$ .
4. Speichern von State  $s$ , Action  $a$ , Policy  $\pi_{\theta_{\text{old}}}(s, a)$ , Reward  $r$ , Folge-State  $s'$ .  
 $(s, a, \pi_{\theta_{\text{old}}}(s, a), r, s') \rightarrow EB$
5. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis N Zeitintervalle bzw. für N Spielepisoden erreicht sind.
6. Entnehme ein Mini-Batch aus dem Buffer  $(S, A, \{\pi_{\theta_{\text{old}}}\}, R, S') \leftarrow EB$ .
7. Bestimmt die Ratio  $r_t(\theta)$ . 2.3.2.
8. Berechne die Advantages  $\hat{A}_t(s, a)$ . 2.3.2.
9. Berechne die Surrogate Losses  $\text{surr}_1$  und  $\text{surr}_2$ . 2.3.2.
10. Bestimmt den PPO Loss. 2.1
11. Update die Gewichte des Actors und Critics.  $\theta_{\text{old}} \leftarrow \theta$  und  $w_{\text{old}} \leftarrow w$
12. Wiederhole alle Schritte ab Schritt 7  $K$ -mal erneut durch.
13. Wiederhole alle Schritte ab Schritt 2 erneut durch, bis der Verfahren konvergiert.

## 2.4. Deep Q-Network

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

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

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

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

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

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

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

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



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

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

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

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

### 2.4.1. DQN - Algorithmus

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

1. Initialisiere alle Hyperparameter. Initialisiere die Gewichte für das Q-NN zufallsbasiert. Setze Epsilon  $\epsilon = 1.00$  und Erzeuge leeren Replay-Buffer  $RB$ .
2. Wähle mit der Wahrscheinlichkeit  $\epsilon$  eine zufällige Action  $a$  oder nutze  $a = \arg \max_a Q(s, a)$
3. Führt Action  $a$  aus und ermittle den Reward  $r$  und den Folgezustand  $s'$ .
4. Speichern von State  $s$ , Action  $a$ , Reward  $r$  und Folgestate  $s'$ .  $(s, a, r, s') \rightarrow$  Replay-Buffer
5. Senkt  $\epsilon$ , sodass die Wahrscheinlichkeit eine zufällige Action zu wählen minimiert wird. Gewöhnlich existiert eine untere Grenze für  $\epsilon$ , sodass immer noch einige wenige Actions zufällig gewählt werden, je nach Grenze.
6. Entnehme auf zufallsbasiert Mini-Batches aus dem Replay Buffer.
7. Berechne für alle sich im Mini-Batch befindliche Übergänge den Zielwert  $y = r$  wenn die Episode in diesem Übergang endet. Ansonsten soll  $y = r + \gamma \max_{a' \in A} \hat{Q}(s', a')$ .

8. Berechne den Verlust  $\mathbb{L} = (Q(s, a) - y)^2$
9. Update  $Q_\theta(s, a)$  durch SGD (Stochastischen Gradientenabstiegsverfahren, Englisch: Stochastic Gradient Descent). Daher  $Q_{\theta_i}(s, a) \longrightarrow Q_{\theta_{i+1}}(s, a)$
10. Kopiere alle  $N$  Schritte die Gewichte von  $Q(s, a)$  nach  $\hat{Q}(s, a)$   $\theta_{Q(s, a)} \longrightarrow \hat{Q}(s, a)$
11. Wiederhole alle Schritte von Schritt 2 an bis sich eine Konvergenz einstellt.

# Kapitel 3

## Anforderungen

In Kapitel 2, wurden die Grundlagen für den weiteren Vergleich der Reinforcement Learning Agenten gelegt, welche auf zwei unterschiedlichen Algorithmus-Arten (DQN und PPO) basieren.

Um diesem Vergleich durchzuführen soll ein System erstellt werden, welches diese geplanten Vergleich durchführen, festhalten und auswerten kann. Dieses soll aus einem Environment, aus mehreren Agenten beider Algorithmus-Arten, sowie aus statistischen Analysekomponenten zur Leistungsbestimmung, bestehen. Zuzüglich sollen weitere Anforderungen an die Evaluation gestellt werden, um die Vergleichbarkeit sicherstellen.

### 3.1. Anforderungen an das Environment

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

#### 3.1.1. Standardisierte Schnittstelle

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

#### 3.1.2. Funktionalitäten

Das Env soll die folgenden Funktionalitäten implementieren.

### **Aktionsausführung**

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

### **Reset**

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

### **Render**

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

## **3.2. Anforderungen an die Agenten**

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

### **3.2.1. Funktionalitäten**

Die Agenten müssen folgende Funktionalitäten implementieren.

#### **Aktionsbestimmung**

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

#### **Lernen**

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

### **3.2.2. Parametrisierung**

Das System muss die Möglichkeit besitzen, mehrere Agenten der gleichen Algorithmus-Art zu erstellen, welche sich jedoch durch verwendeten Hyperparameter und Algorithmus-Art unterscheiden. Diese Definition von Agenten ist in der Evaluation zu berücksichtigen.

### 3.2.3. Diversität der RL Algorithmen

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

## 3.3. Anforderungen an die Datenerhebung

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

### 3.3.1. Mehrfache Datenerhebung

Die Datenermittlung der muss für jeden einzelnen Agenten mehrfach durchgeführt werden, um die Validität der Messung zu gewährleisten.

### 3.3.2. Datenspeicherung

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

Das System muss daher folgende Daten speichern:

Tabelle 3.1.: Zu erhebende Daten

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

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

### 3.4. Anforderungen an die Statistiken

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

### 3.5. Anforderungen an die Evaluation

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

Tabelle 3.2.: Evaluationskriterien

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

---

Trainingszeit	Welcher Agent schafft es ein festes Ziel in der geringstmöglichen Zeit zu erreichen bzw. welcher Agent ist als erstes in der Lage durchschnittlich $x$ Äpfel zu fressen.
Spielzeit	Welcher Agent schafft es im Durchschnitt am längsten einen terminalen Zustand zu vermeiden? Auf Snake bezogen: Welcher Agent schafft es am längsten nicht zu sterben.

# Kapitel 4

## Verwandte Arbeiten

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

- Optimierungsstrategien
- Reward Function
- Evaluationskriterien

### 4.1. Autonomous Agents in Snake Game via Deep Reinforcement Learning

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

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

Der Dual Experience Replay (Splited Memory) besteht aus zwei Sub-Memories, in welchen Erfahrungen belohnungsbasiert sortiert und gespeichert werden. Mem1 besteht nur aus Erfahrungen, welche einen Reward größer als ein vordefinierter Grenzwert besitzen. Die restlichen Erfahrungen wandern in Mem2. Beim Lernen wird, zu beginn, eine überproportional größere Menge an Erfahrungen aus Mem1 ausgewählt,



um den Lernerfolg zu beschleunigen. Im weiteren Lernverlauf wird dann das Entnahmeverhältnis normalisiert (Mem1: 50% & Mem2: 50%).

Der Training Gap stellt die Erfahrungen dar, welche der Agent, zum Zwecke des performanteren Lernens, nicht verarbeiten soll. Zu diesen zählt die Erfahrung direkt nach dem Konsum eines Apfel, sodass der Agent die Neuplatzierung eines Apfels erlernt. Da der Agent auf diesen Prozess keinen Einfluss hat, könnte die Verarbeitung dieser Daten den Lernerfolg mindern, weshalb die Training Gap Strategie etwaige Erfahrungen löscht.

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

Die optimierte Reward Function, welche das Paper verwendet, besteht damit aus dem Standard Distanz Reward ohne Training Gap Erfahrungen. Letzterer bestimmt sich aus der Distanz zwischen Kopf und Apfel und der Länge der Snake, wobei der Factor der Länge wieder antiproportional den Reward beeinflusst. Sollte die Timeout Strategy auslösen, so werden die letzten  $P$  Erfahrungen entsprechend angepasst und in Mem2 verschoben.

Als maßgebliche Kriterien zur Evaluation der Leistung des DQN wurde die Performance, daher die Anzahl an gefressenen Äpfeln und die steps survived, also die überlebten Schritte herangezogen.

#### 4.1.1. Diskussion

Sowohl die Training Gap also auch Dual Experience Replay und Timeout Strategy stellen vielversprechende Optimierungen dar, welche, auf experimentellen Resultaten basierend, gute Ergebnisse erzielen konnten. Jedoch ist auch Kritik am Paper anzubringen. Das Env wird im Paper nur kurz und nicht detailliert genug vorgestellt. Aus dem Abschnitt Game Environment lässt sich jedoch schließen, dass alle Anforderungen des Env 3.1 vorhanden sind.

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

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

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

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

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

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

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

Eine weitere Optimierungsstrategie basiert auf dem Loop Storm Effect, welcher das Verhalten des Laufens um den Apfel beschreibt. Die Verfasser haben festgestellt, dass dieser Effekt zu einem sehr schlechten Lernerfolg und zu langen Trainingsdau-

ern führt. Darum haben Wu et al. einen dynamischen Positionsspeicher konzipiert, welcher Loops erkennt und diese durch das Zurückgeben einer Zufallsaktion, welche nicht auf dem Loop liegt, unterbricht. Experimentelle Ergebnisse des Papers zeigen, dass der Loop Storm Effect kaum mehr präsent ist.

Auf Basis einer Standard Reward Function, welche für das Essen eines Apfels +100 und für das Sterben -100 zurückgibt, wurden ein Versuch durchgeführt. Dem Agenten war es nicht möglich gegen die optimale Lösung zu konvergieren, aufgrund von Loop Storms und einer unzureichenden Reward Function. Hingegen war es dem Agenten mit dem Odor Effect und mit der Breakout Loop Storm Strategy möglich eine Konvergenz in 8 Millionen Epochs (hier Trainingsdurchläufe) zu verzeichnen.

Auch in diesem Paper bleibt die Performance maßgeblicher Evaluationsfaktor, wobei der Fokus auf den erreichten Reward gelegt wurde, welcher stark mit der Performance korreliert.

#### 4.2.1. Diskussion

Auch dieses Paper besitzt interessante Verbesserungen, welche nach den ersten Ergebnissen gute Resultate vorweist. Dennoch sind auch bei diesem viele der in Kapitel drei gestellten Anforderungen 3 nicht beachtet oder erfüllt worden.

Aus Graphiken geht hervor, dass die Verfasser ein Env mit einer Visualisierung besitzen. Weiterhin ist davon auszugehen, dass auch alle weiteren Anforderungen bezüglich des Env 3.5 erfüllt worden sind, da ansonsten kein Training eines Agenten möglich wäre. Dennoch wurde der Leser nur unzureichend mit Details über das Env informiert.

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

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

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

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

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

### 4.3. Zusammenfassung

Sowohl "Autonomous Agents in Snake Game via Deep Reinforcement Learning" 4.1 als auch "UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene" 4.2 bieten einige Optimierungsstrategien, welche im weiteren Verlauf dieser Ausarbeitung angewendet werden sollen. Damit die Optimierungen auf beide Agentenarten (DQN und PPO) angewendet werden können, müssen sie jedoch an die Algorithmusarten angepasst werden.

Des Weiteren ist zu bemerken, dass beide Paper nur sehr eingeschränkt die in Kapitel drei gewählten Anforderungen erfüllen. Besonders auffällig ist dabei die geringe Algorithmusdiversität. Es wurden hauptsächlich DQN verwendet und andere Algorithmen wurden entweder nur unzureichend in einen Vergleich eingebunden oder vollkommen ausgelassen.

# Kapitel 5

## Konzept

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

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

### 5.1. Vorgehen

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

Zu Beginn sei erwähnt, dass davon ausgegangen wird, dass alle für den Vergleich benötigten Komponenten, wie z.B. Environment, Agenten und statistische Analysekomponenten entsprechende der Anforderungen 3 implementiert sind.

Als erstes werden die Agenten erstellt 5.1. Für diese Initialisierung werden die Hyperparameter aus 5.3 verwendet. Mit diesem Baseline Agenten Bestand werden nun die weiteren Vergleich durchgeführt.

Für jedes Evaluationskriterien werden aus dem gegebenen Agenten-Bestand die zwei optimalen Baseline Agenten ausgewählt. Die Algorithmus-Art besitzt dabei der Auswahl der Agenten keinen Einfluss. Genauere Details zur Durchführung der Baseline Vergleiche finden sich in 5.6.

Basierend auf den beiden Sieger Agenten (Agent-01 und Agent-02) 5.1, werden nun die Optimierungen auf das Env. und die Agenten angewendet. Mit diesen optimierten Agenten (Agent-01 Optimierung A bis Agent-02 Optimierung C) werden nun die Vergleiche bezüglich jedes einzelnen Evaluationskriteriums 3.2 wiederholt.

Im letzten Schritt soll in der gesamt Evaluation der optimale Agent für jedes Evaluationskriterium ermittelt werden. Dabei können dies auch Baseline Agenten sein.

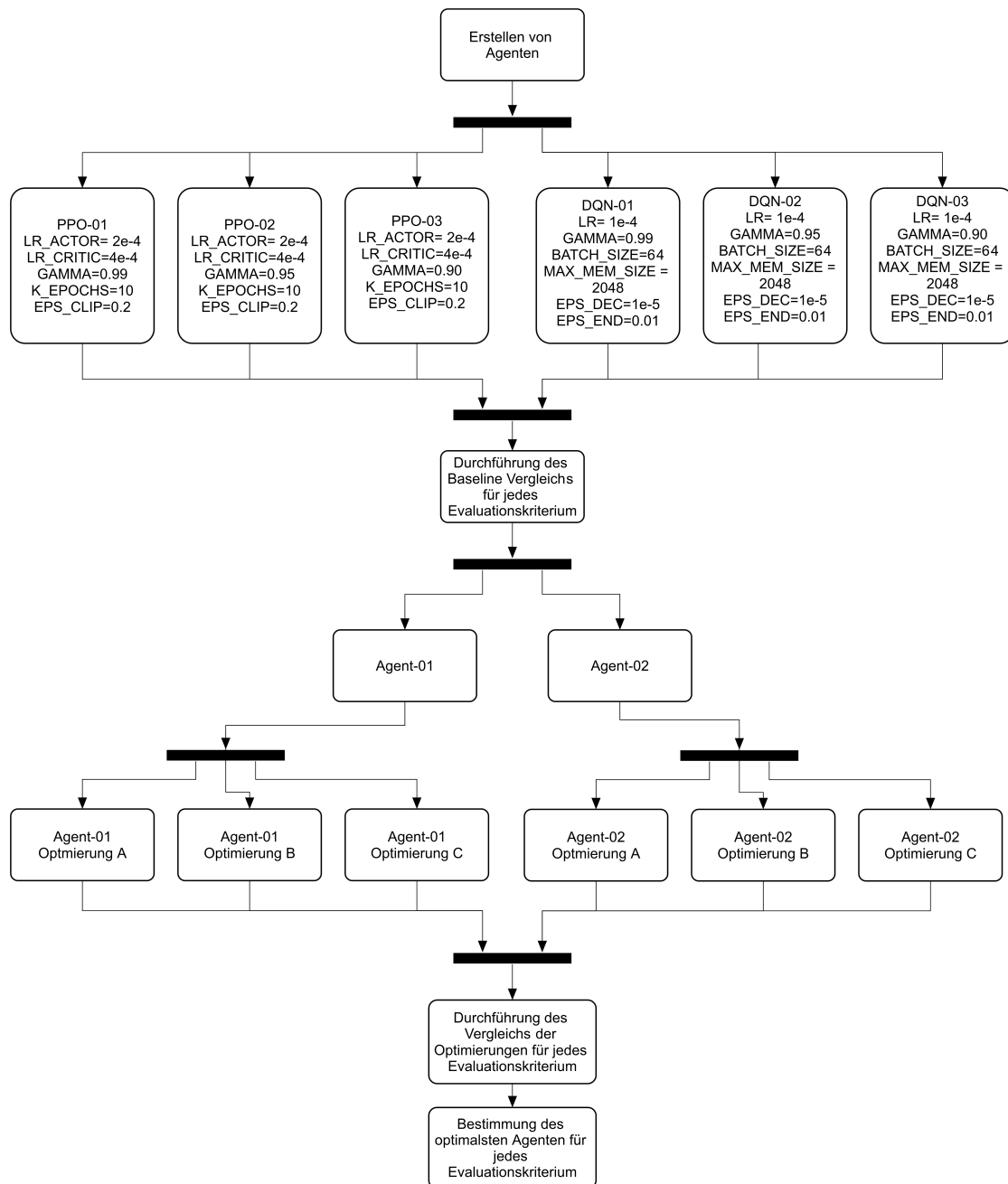


Abbildung 5.1.: Darstellung des Vorgehens.

## 5.2. Environment

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

### 5.2.1. Wrapper

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

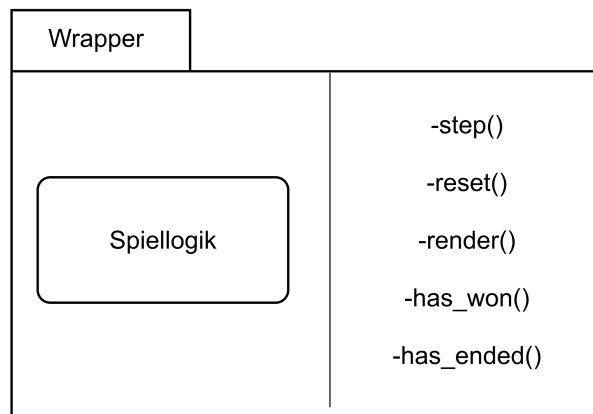


Abbildung 5.2.: Darstellung des Wrappers.

Die `step` Funktionalität 5.2 ist die Hauptmethode im Wrapper. Sie bekommt eine Aktion übergeben, welche zuvor vom Agent bestimmt wurde. Diese wird mit Hilfe der Spiellogik umgesetzt. Entsprechend der Anforderung 3.1.1 gibt diese Methode nur einen Reward und eine Obs zurück.

Reset setzt den bereits vorhandenen Spielfortschritt zurück, wobei auf die Spiellogikkomponente zugegriffen wird. Die Anforderung 3.1.2 ist damit erfüllt.

Die Render Methode ist für die Visualisierung verantwortlich und ruft Methoden in der GUI-Unterkomponente auf, welche ein genaues Bild von der Spiellogik-Komponente erhält. Die Anforderung 3.1.2 ist damit erfüllt.

Die `has_won` und `has_ended` Methodiken geben Statusinformationen über den Momentanen Spielstand zurück, welche für den Spiel- bzw. Trainingsablauf benötigt werden.

### 5.2.2. Spiellogik

Die Spiellogik besteht aus den fünf Unterkomponenten, welche in 5.2 bereits benannt wurden.

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

Die Reward-Komponente bestimmt den auszugebenden Reward nach jeder Aktionsabfertigung. Dieser berechnet sich wie in 5.2.2 angegeben. Zuzüglich wird im Rahmen eine der Optimierungen, siehe 5.6, eine weitere Reward Funktion implementiert. Diese berechnet sich wie in 5.4.1 angegeben.

In der Game-Komponente werden wichtige Spielbezogene Daten verwaltet. Zu diesen gehören das Spielfeld (ground), sowie die Form des Spielfeldes (shape) und die Position des Apfels auf dem Spielfeld. Sie beinhaltet viele Methoden, wie z.B. die action, observe, evaluate, reset, view, make\_apple.

In der Player-Komponente, welche zur Datenverwaltung dient, werden Spielerbezogene Daten verwaltet. Zu diesen zählen die Position des Kopfes der Snake, sowie ihrer Schwanzglieder, ihre Ausrichtung (Direction), ihre gelaufenen Schritte seit dem letzten Fressen eines Apfels (inter\_apple\_steps), ihr Lebensstatus (done), daher ob sie tot oder lebendig ist und weitere Farbkonstanten für die GUI.

Die Observation-Komponente beinhaltet viele einzelne Funktionen zur schrittweisen Erstellung der Observation, wie sie in 5.2.2 erklärt wird.

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

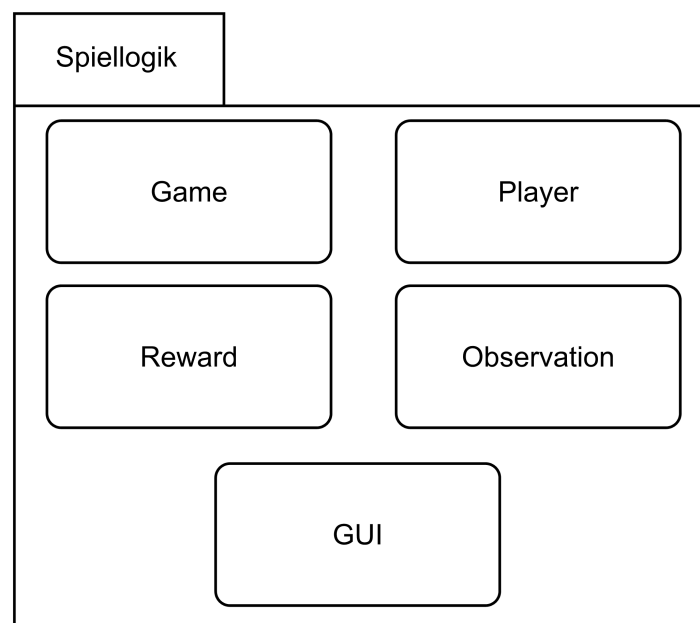


Abbildung 5.3.: Darstellung der Spiellogik mit ihren Unterkomponenten.



## Spielablauf

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

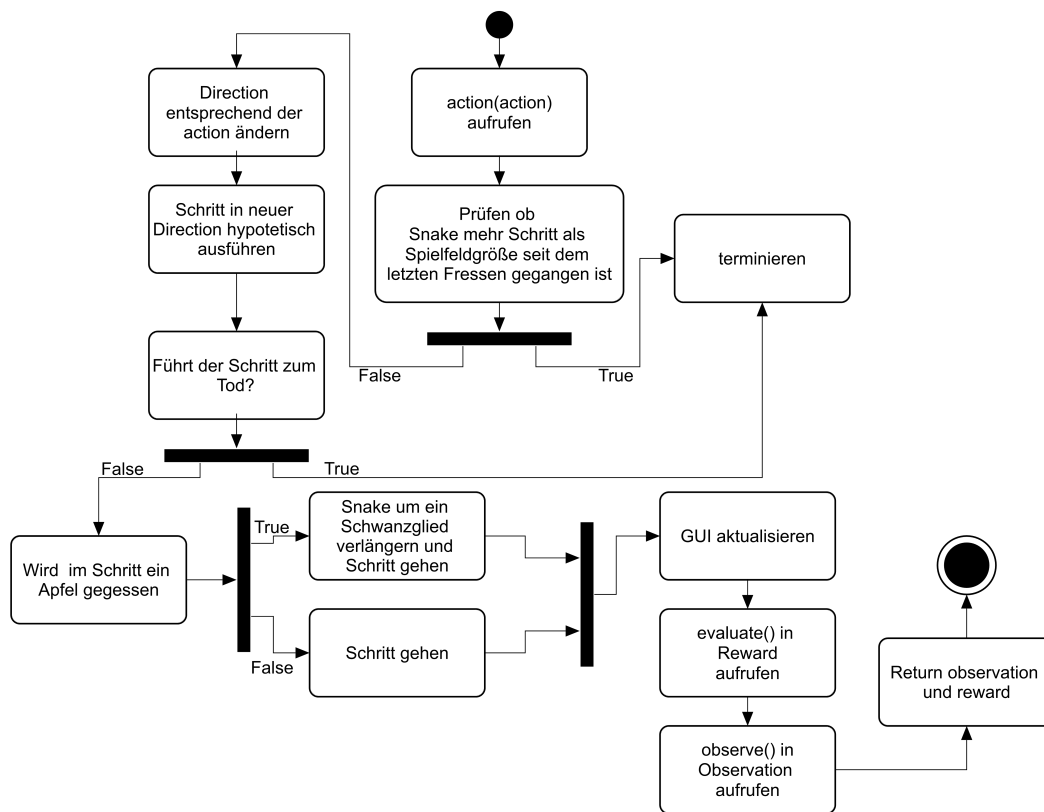


Abbildung 5.4.: Darstellung eines Schrittes in der Spielepisode.

Zu Beginn wird überprüft, ob die Snake seit dem letzten Fressen mehr Schritte als die eigentliche Spielfeldgröße gegangen ist. Die Spielfeldgröße berechnet sich dabei wie folgt  $(8, 8) = 64$ , daher ist die Spielfeldgröße 64. Sollte die Snake mehr Schritte gelaufen sein als diese Größe, so wird das Spiel terminiert, da die Snake eventuell in einer Schleife steckt und daher diese nie wieder verlassen würde.

Andernfalls wird die Aktion verarbeitet, in dem sie die direction verändert der Snake manipuliert. Das Spiel Snake besitzt in diesem Konzept drei Actions turn left, turn right oder do nothing.

Tabelle 5.1.: Kodierung der Actions

Action	Erklärung
turn left	Die Snake ändert ihre Richtung um 90° nach links. Z.B. Von N $\rightarrow$ W

turn right	Snake ändert ihre Richtung um 90° nach rechts. Z.B. Von N $\rightarrow$ O
do nothing	Die Richtung der Snake wird nicht verändert.

Entsprechend der Beispiele in 5.1 wird klar, dass die Direktion entweder nur Norden, Osten, Süden oder Westen sein kann. Als nächstes wird ein Schritt der Snake mit der aktualisierten Aktion hypothetisch durchgeführt. Dabei lässt sich feststellen, ob die Ausführung des Schrittes zum Tod der Snake führt. Sollte dies der Fall sein, so wird der Spielablauf terminiert. Dabei führt das laufen in sich selbst und das Verlassen des Spielfeldes zum Tod 2.1.

Anderenfalls wird der Schritt durchgeführt. Dabei wird zwischen zwei Fällen unterschieden. Sollte die Snake einen Apfel gefressen haben, also der Kopf der Snake und der Apfel die selbe Position einnehmen, so wächst die Snake um ein Schwanzglied. Der alte Apfel wird entfernt und ein neuer erscheint zufallsbasiert irgendwo auf einem freien Platz des Spielfelds.

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

Nach der Ausführung einer dieser beiden Fälle, wird die GUI aktualisiert mit der `update_gui` Methode. Nach diesem Schritt ist die Abarbeitung der `action` Methode abgeschlossen. Damit jedoch die der Agent den Nachfolgezustand und Reward erhält wird die `evaluate` Methode in der Reward-Komponente und die `observe` Methode in der Observation-Komponente aufgerufen.

Zum Schluss werden Obs und Reward zurückgegeben.

## Reward

Die `evaluate` Methode befindet sich in der Game-Komponente und ruft ihrerseits die `standard_reward` Method in der Reward-Komponente auf. Sie bestimmt, basierend auf dem letzten Zug, den Reward, was nach folgenden Vorbild geschieht. Der Reward ist abhängig von drei Faktoren. Dem Fressen eines Apfels, dem Sieg und dem Verlust. Sollte keiner dieser genannten Faktoren eintreten, wird ein Reward von -0.01 zurückgegeben. Dies hält den Agenten dazu an den kürzesten Pfad zum Apfel zu finden, da jeder Schritt geringfügig bestraft wird.

War es der Snake möglich einen Apfel zu fressen so wird ein Reward von +1.0 zurückgegeben, da ein Sub-Goal erfüllt worden ist. Sollte die Snake gestorben sein, durch das Verlassen des Spielfeldes oder das Laufen in sich selbst oder das zu lange Umherlaufen, so wird ein Reward von -10 zurückgegeben, um dieses Verhalten in seiner Häufigkeit zu minimieren. Hat die Snake alle Äpfel gefressen, sodass das gesamte

Spielfeld mit der Snake ausgefüllt ist, so wird ein Reward von +10 zurückgegeben, um ein solches Verhalten in seiner Häufigkeit zu maximieren.

## Observation

Die Observation, welche von der `step` Method des Wrappers 5.2.1 zurückgegeben wird, besteht aus der `around_view` (AV) und der `scalar_obs` (SO). Zur Erstellung der Obs wird die `observe` Methode in der Game-Komponente 5.2.2 aufgerufen. Diese ruft ihrerseits die `make_obs` Funktion in der Observation-Komponente auf. Mit Hilfe verschiedener Unterfunktionen wird dann die Obs generiert.

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

Das One-Hot-Encoding ist ein binäres encoding System. Sollte ein Merkmal vorhanden sein, so wird dieses mit eins codiert anderenfalls mit null.

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

Tabelle 5.2.: Channel-Erklärung der Around\_View (AV)

Channel der Matrix bzw. Erste Dimension (A x 13 x 13)	Erklärung
A = 0	Die erste Feature Map signalisiert den Raum außerhalb des Spielfelds. Nährt sich die Snake dem Rand, so würde der Ausschnitt der AV aus dem Spielfeld herausragen und den Eindruck erwecken, dass dieser größer wäre als er in Realität wirklich ist. Darum werden Felder der AV, die sich außerhalb des Spielfeldes befinden, angezeigt.
A = 1	Diese Feature Map stellt alle Schwanzglieder mit Ausnahme des Kopfes und es letzten Schwanzgliedes dar.
A = 2	In dieser Feature Map wird der Kopf der Snake dargestellt.
A = 3	Damit gegen Ende des Spiels der Agent noch freie Felder erkennen kann, wird in dieser Feature Map jedes freie und sich im Spielfeld befindliche Feld mit eins codiert.

$A = 4$	Die vorletzte Feature Map codiert das Schwanzende der Snake.
$A = 5$	In der letzte Feature Map wird der Apfel abgebildet.

Vorteilhaft an der AV ist, dass, im Gegensatz zu den verwandten Arbeiten 4.1 und 4.2, nicht das gesamte Feld übertragen wird, sondern nur der wichtigste Ausschnitt, was die Menge an zu verarbeiten Daten reduziert. Des Weiteren ergeben sich keine Probleme zwischen variablen Spielfeldgrößen und der Input-Size von Convolutional Layers A.3.1.

Ein Nachteil dieser Obs ist jedoch die Unvollständigkeit. Sollte der blaue Punkt in 5.5 außerhalb des grauen Kastens und daher außerhalb der AV liegen, so bleibt der Agent im Unklaren über den Aufenthaltsort des Apfels. Auch Informationen wie z.B. der Hunger, also die verbleibenden Schritte bis das Spiel endet, die Distanzen zu den Wänden und zu, außerhalb der AV liegenden, Schwanzteilen und die Ausrichtung (direction) der Snake, werden durch die AV nur eingeschränkt oder gar nicht geliefert.

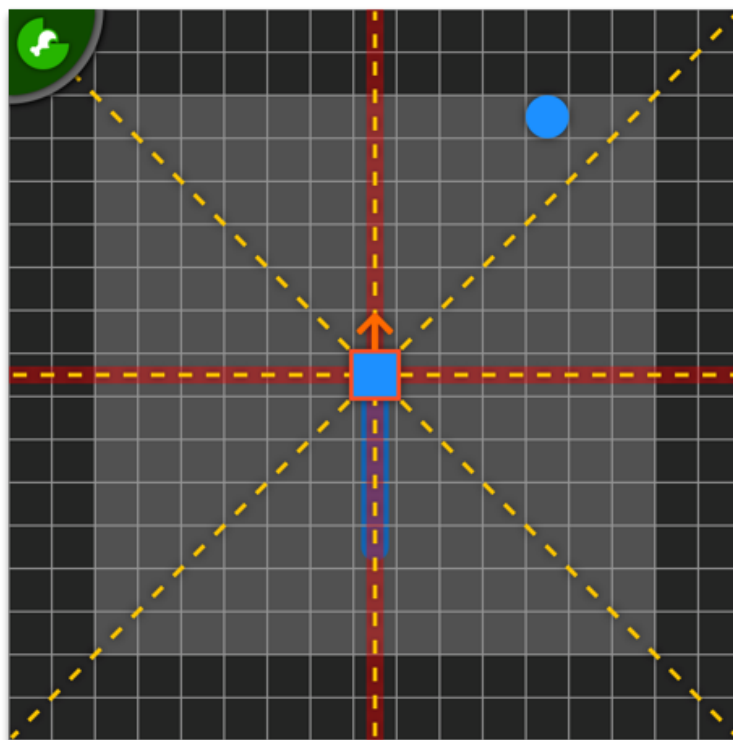


Abbildung 5.5.: Partielle Darstellung der verwendeten Observation. Das blaue Rechteck und dessen Schwanz stellt die Snake dar, wobei das rot umrandete Rechteck den Kopf darstellt. Die schwarzen Felder werden nicht von der AV abgedeckt, graue liegen innerhalb der AV. Die gelben gestrichelten Linien stellen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

Aus diesem Grund wurde die AV durch die `scalar_obs` (SO) ergänzt. Diese beinhaltet skalare Informationen und ist eine Konkatenation aus X-Ray Distanzbestimmung, Hunger- und Blickrichtungsanzeige. Zuzüglich werden der SO noch zwei Kompass für relative Positionsinformation zwischen Kopf und Apfel bzw. letztem Schwanzglied hinzugefügt. Letztere sind eindimensionale Vektoren, welche über das One-Hot-Encoding anzeigen, ob sich das gesuchte Objekt relativ zum Kopf oberhalb, unterhalb oder in der selben Zeile (Matrixsicht) befindet. Analog verhält es sich mit der vertikalen Sicht.

Die Blickfeldanzeige ist ebenfalls one-hot-encoded und stellt mit ihrem Vektor die vier Ausrichtungen Norden, Osten, Süden und Westen dar.

Der Hunger ist ein eindimensionaler Vektor mit einem einzigen Eintrag, welcher sich aus der Differenz zwischen der Anzahl der gegangenen Schritten seit dem letzten Fressen (`inter_apple_steps`) und der maximalen Schrittzahl ohne einen Apfel gefressen zu haben (`max_steps`), berechnet. Diese maximale Schrittzahl ist als Spielfeldgröße definiert. Da der Hunger bei einer großen Differenz einen kleinen Einfluss und bei einer geringen Differenz einen großen Einfluss besitzen soll, wurde diese durch eins geteilt. Nähern sich die beiden Werte an, so rückt der Wert des Bruches näher an unendlich, da der Nenner immer kleiner wird.

$$\min(2, \frac{1}{inter\_apple\_steps - max\_steps}) \quad (5.1)$$

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

In ähnlicher Weise wird mit den X-Rays Distanzbestimmungen verfahren. Bei ihnen handelt es sich um acht Distanzmesserkonturen, die in 45° Abständen ausgesandt werden, siehe 5.5. Befindet sich das gesuchte Objekt in dieser Linie, so wird die Distanz zwischen dem Kopf der Snake und dem Objekt durch eins dividiert und zurückgegeben. Es wird nach Wänden, dem eigenem Schwanz und dem Apfel gesucht. Daher wird die X-Ray Distanzbestimmung in einem Vektor der Größe 24 ( $3 * 8 = 24$ ) gespeichert.

## GUI

Die graphische Oberfläche oder auch GUI genannt kann optional ein oder ausgeschaltet werden. Beim lernen der Agenten bietet es sich beispielsweise an diese auszuschalten, da diese den Lernprozess senkt. Beim Start der GUI wird ein Fenster geöffnet, welches den momentanen Stand der Spielgeschehens anzeigt. Da Snake eine zweidimensionale Spielfläche besitzt, wird diese im Fenster dargestellt. Nach jeder Aktionsdurchführung muss die GUI mit der `update_gui` Methode aktualisiert werden, um stets den neusten Stand des Spiels zu zeigen.

## **5.3. Agenten**

In diesem Abschnitt des Konzepts sollen die Agenten vorgestellt werden. Dazu werden die Algorithmus-Arten (PPO und DQN) näher erläutert. Zuzüglich wird noch die Netzstruktur für diese Agenten beleuchtet.

### **5.3.1. Netzstruktur**

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

## **5.4. Optimierungen**

### **5.4.1. Optimierung 02**

## **5.5. Datenerhebung**

## **5.6. Vergleich**

# Kapitel 6

## Agenten

Ein zentraler Aspekt eines Vergleiches von verschiedenen RL-Agenten ist die genaue Definition der einzelnen Agenten. Basierende auf den Grundlagen 2.2.1 soll in diesem Kapitel der Begriff vervollständigt und die zu vergleichenden Agenten sollen vorgestellt werden.

Erste statistische Erhebungen haben gezeigt, dass die ausgewählten Hyperparameter einen immensen Einfluss auf das Verhalten der Agenten haben. Ein Vergleich zwischen DQN und PPO mit wahllos gewählten Hyperparametern ist folglich wenig aussagekräftig. Daher ist auch die Definition des Begriffs Agent, welcher nur zwischen DQN und PPO differenziert, unzureichend.

Angebracht wäre eine erweiterte Definition des Agentenbegriffs, welche um den entscheidenden Faktor der Hyperparameter erweitert wird. Ein Agent ist daher nicht mehr alleinig durch seine RL-Klasse (Q-Learning oder Policy Gradient) und Algorithmus-Klasse (DQN oder PPO) definiert, sondern ebenfalls durch die ausgewählten Hyperparameter.

### 6.1. Agenten

Im Folgenden werden die Agenten, welche untereinander verglichen werden sollen, vorgestellt. Dargestellt werden die RL-Klassen, Algorithmus-Klassen und Hyperparameter.

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

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

Der PPO-3 ist ein Proximal Policy Optimization Algorithmus, welcher auf der RL-Klasse des Policy Gradient basiert. Er besitzt folgende Hyperparameter:  $\gamma = 0.99$ ,  $K\text{-epochs} = 10$ ,  $\text{epsilon-clip} = 0.2$ ,  $\text{batch-size} = 64$ ,  $\text{entropy\_coefficient} = 0.001$ ,  $\text{critic-loss-coefficient} = 0.5$   $\text{lr-actor} = 1.0\text{e-}3$ ,  $\text{lr-critic} = 1.5\text{e-}3$ .

Der PPO-4 ist analog zum PPO-3 mit den folgenden Unterschieden in seinen Hyperparametern:  $\gamma = 0.95$ ,  $\text{entropy-coefficient} = 0.01$ .



# Kapitel 7

## Implementierung

Für eine Umsetzung eines solchen Vergleichs, wie er in dem Kapitel ?? beschrieben worden ist, ist es nötig eine Implementierung des Spiels Snake und der beiden Agenten, inklusive der Ablaufroutine, durchzuführen. Als Programmiersprache wurde Python (3.7) gewählt.

Python bietet im Bereich des DRL eine Vielzahl an Frameworks, welche nicht nur bei der Implementierung des Envs. helfen, sondern auch welche, die Funktionalität der Neuronalen Netzwerke bereitstellen.

### 7.1. Snake Environment

Zur Implementierung des Spiels Snake wurde das Framework gym von OpenAI genutzt (<https://gym.openai.com/>). Dieses bietet viele Methoden und Vorgaben in der Projektstruktur, welche das Implementieren erleichtern. So besteht das Snake Environment Package (snake\_env), aus den wesentlichen Files:

- gui
- observation
- snake\_env
- snake\_game

#### 7.1.1. Schnittstelle

Die grundlegende Schnittstelle des Snake Env. wird in dem File snake\_env bereitgestellt. In diesem wird die Klasse SnakeEnv definiert, welche durch die Vererbung der Oberklasse gym.Env zentrale Methoden übernimmt. Zu diesen gehören die step, reset, render und close Methode.

Weiterhin bietet die selbst definierte Methode post\_init die Möglichkeit zentrale Einstellungen, wie z.B. die Spielfeldgröße und GUI-Aktivierung, zu editieren. Die

step-Methode startet den Ausführungsprozess der vom Agenten ausgewählten Action, durch Aufrufen weiterer Methoden 7.1.2. Nach der Abarbeitung wird die neue Obs, der Reward, das done-flag und das has\_won-flag übermittelt, wobei letzteres zur Bestimmung des Sieges dient. Die reset-Methode startet das Spiel von neuem. Zum Schluss wird dann noch die Obs. zurückgegeben. Die render-Methode ruft die Methoden für die visuelle Darstellung auf. Die close-Methode terminiert das Programm.

Auf Basis der oben genannten Methoden wird ersichtlich, dass es sich bei der Klasse SnakeEnv um eine Wrapper-Klasse handelt, die als Schnittstelle dient.

### 7.1.2. Spiellogik

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

Der Weiteren werden die folgenden Methoden implementiert:

Tabelle 7.1.: Methoden der SnakeGame Klasse

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

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

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

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

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

Entsprechende der `Action` wird die `direction` des `Players` angepasst. Die vier Himmelsrichtungen werden dabei mit den Zahlen von `0` - `3` dargestellt, wobei `null` Norden entspricht eins Osten usw.

Nach der Manipulation der `direction` des `Players`, wird `pos`, also die vorläufige neue Position, im `Player`-Objekt angepasst. Diese Änderung wird jedoch noch nicht sofort in die Matrix übertrage, da das Eintragen on Positionen außerhalb des Spielfeldes zu Fehlern führen würde. Es muss daher erst überprüft werden, ob die neue Position des `Players` im Spielfeld liegt. Sollte dies nicht der Fall sein, so wird die Abbruchprozedur aufgerufen.

Anderenfalls wird die neue Position in `tail` eingefügt.

Zu diesem Stand der Abarbeitung ist es möglich, dass das Spiel bereits gewonnen ist. Um dies zu überprüfen, wird die Länge der Snake mit der maximal möglichen Länge, welche sich durch die Spielfeldgröße ergibt, verglichen. Entspricht die Länge Snake der maximal mögliche, so wird die Abbruchprozedur aufgerufen. Ansonsten muss als nächster Schritt die Matrix aktualisiert werden. Jedoch muss vorher festgestellt werden, ob die Snake einen Apfel gefressen hat.

Sollte sie dies getan haben, so wird die neue Position des Kopfes in die Matrix eingepflegt, ein neuer Apfel wird auf einer zufälligen freien Stelle generiert, `inter_apple_steps` wird auf `null` und die Hilfsvariable `has_grown` wird auf `True` gesetzt.

Letztere wird von der evaluate Methode verwendet, um die Höhe des rewards zu bestimmen, siehe ??.

Ist die Snake jedoch nicht gewachsen, so wird das letzte Schwanzglied aus Matrix und Liste gelöscht, um den Anschein von Bewegung zu erwecken. Zusätzlich wird die Hilfsvariable `has_grown` auf `False` gesetzt.

Zum jetzigen Zeitpunkt besteht immer noch die Möglichkeit, dass die Snake in sich selber gelaufen ist. Um dies festzustellen, wird `tail` auf Duplikate überprüft. Sollten sich Duplikate in `tail` befinden, wird die Abbruchprozedur aufgerufen.

Ansonsten wird zum Schluss noch über `tail` iteriert und die korrespondierenden Einträge der Matrix (`ground`) werden mit den Positionen von `tail` aktualisiert, wobei der Kopf und das letzte Schwanzglied mittels eines anderen Zahlenwert dargestellt werden.

### 7.1.3. Graphische Oberfläche

Im `gui` File wird eine Klasse `GUI` definiert, welche das Spiel mit Hilfe des Frameworks `pygame` (<https://www.pygame.org/>) darstellt. Dazu wird in der `GUI`-Klasse eine Oberfläche (`screen`) erzeugt, welche die Matrix `ground` darstellt, siehe 2.1. Die Methode `update_GUI`, welche von der `SnakeGame` Methode `view` aufgerufen wird, überschreibt dazu jeden einzelnen Eintrag des `pygame`-Spielfelds mit dem korrespondierenden Wert von `ground`. Spielfeld (`GUI`) und Matrix sind daher nicht direkt gekoppelt sondern müssen über `update_GUI` angeglichen werden.

Die Fenstergröße der Spieloberfläche wird dynamisch berechnet und kann über das Attribut `Particle` verändert werden, welches die Feldgröße eines einzelnen Matrixeintrags darstellt. Die `draw` Methode ist für das Generieren der einzelnen Spielfeld-Rechtecke zuständig und `reset_GUI` versetzt das Spielfeld zurück in den Ursprungszustand.

## 7.2. Agenten

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

### 7.2.1. Netzstruktur

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

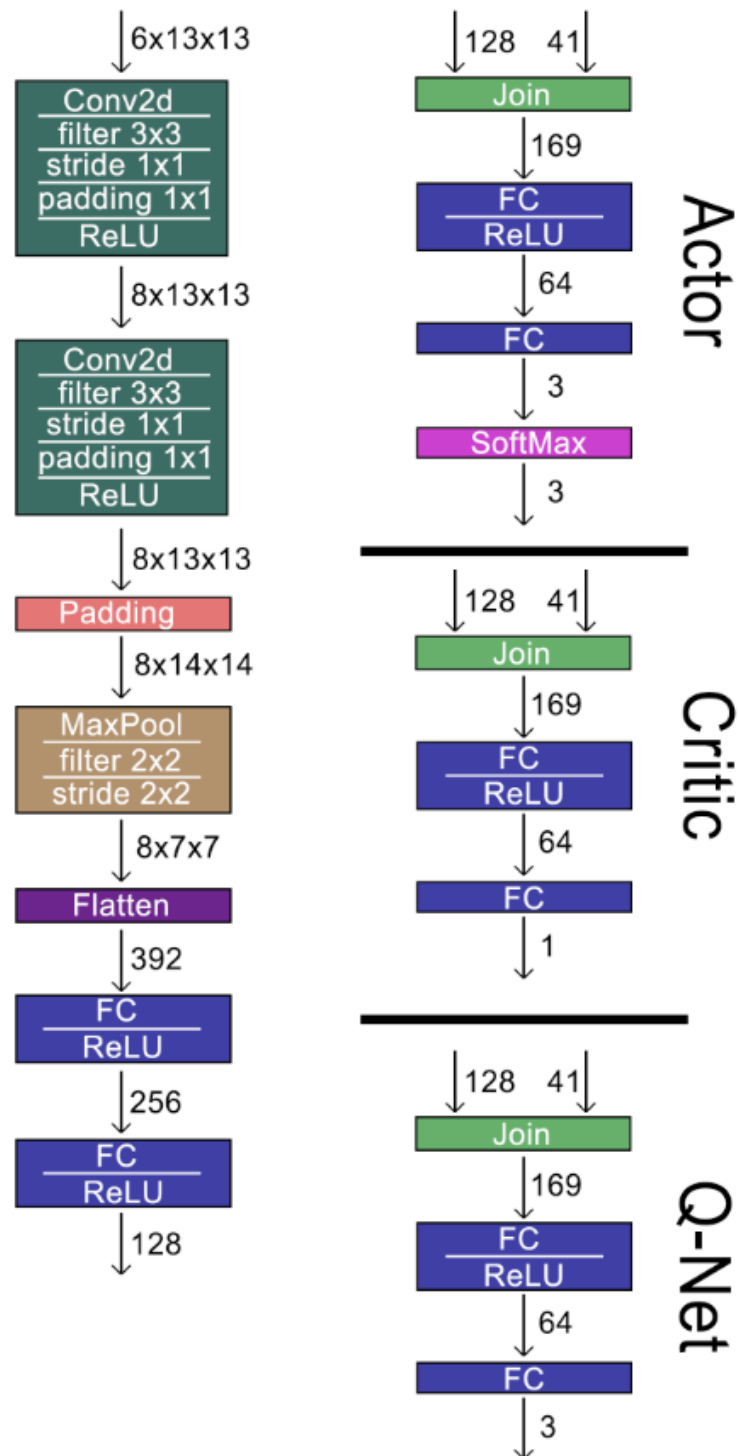


Abbildung 7.1.: Darstellung des NN, welches als Standard für den weiteren Vergleich dient.

Das Netz findet sich in den Files actor und critic der PPO Package und im q\_net des DQN Package. Diese Klassen besitzen eine forward Methode, welche die Obs (AV, SO), nach dem umwandelt in Tensoren, durch das NN propagiert und einen Output bestimmt. Dieser Prozess geschieht dabei folgendermaßen:

Zuerst wird die AV durch zwei Convolutional Layer mit einer ReLU Aktivierungs-

funktion durch propagiert. Dabei erhöht sich die Channel-Anzahl auf acht, da die AV bereits sehr stark optimiert wurde und ein Anstieg auf 16 oder 32 Channels nicht nötig ist. Die Feature Map wird während dieses Prozesses nicht minimiert, aufgrund eines Paddings. Dies soll den Informationsverlust an den Rändern minimieren. Danach werden allen Feature Maps eine Null-Zeile und Spalte hinzugefügt (Padding), damit beim Max-Pooling unter der Filtergröße und dem Stride von  $2 \times 2$ , auch die letzten Zeile und Spalte verarbeitet wird. Der Tensor besitzt nun die Form  $(8 \times 14 \times 14)$ . Nach dem max-pooling besitzt der Tensor Feature Maps der Größe  $7 \times 7$ . Nach der Einebnung (Flatten) des Tensoren zu einem eindimensionalen Tensor-Vektor wird dieser durch zwei weitere Fully Connected Layer (FC) mit einer ReLU Aktivierungsfunktion durch propagiert. Der resultierende Tensor besitzt die Größe  $1 \times 128$  und ist ein zwischen Ergebnis, da dieser nun mit der SO verbunden wird (Join).

Da das NN in beide Algorithmus-Arten verwendet wird, müssen drei unterschiedliche Netzköpfe definiert werden, siehe 7.1 rechts. Alle unterscheiden sich jedoch nur in ihrer Ausgabe. Nachdem der Joined Tensor ( $1 \times 128$ ) durch zwei weitere FC Layer mit ReLU Aktivierungsfunktion propagiert wurde, benötigt der Actor des PPO-Agenten eine Wahrscheinlichkeitsverteilung über alle Actions. Daher auch die Ausgabe von einem Tensor der Größe drei, auf welchen zur Erstellung der Wahrscheinlichkeitsverteilung SoftMax angewendet wird, siehe 7.1 rechts oben.

Die Critics der PPOs und die DQNs verwendet den rechts unten dargestellten Netzkopf (ValueNet-Head) 7.1. Beim den Critics werden Tensoren mit einem einzigen Zahlenwert zurückgegeben, wohingegen bei den DQNs Tensoren mit drei Q-Values entsprechende der Aktionsanzahl zurückgegeben werden.

### 7.2.2. Testung der Agenten

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

### 7.2.3. DQN

Der DQN Algorithmus ist einer der beiden Algorithmus-Arten, welche im Rahmen dieser Ausarbeitung, implementiert wurden. Diese Implementierung findet hauptsächlich in vier Files statt, welche im dqn Package liegen, dass wiederum zum agents Package gehört. Im dqn File wird die Agentenklasse definiert, welche die Aktionsbestimmungsmethode act und die lern Methode enthält. Im memoryDQN File wird die Memory-Klasse definiert, welche den Replay-Buffer darstellt 2.4. Die Files dqn\_train

und `dqn_play` beinhalten die eigentlichen main-Methoden, welche Agenten und Env. erstellen und den Trainings- bzw. Spielprozess umsetzt.

### Aktionsauswahlprozess

Zur Bestimmung der nächsten Action wird der `act` Methode die momentane Obs übergeben. Die Bestimmung der Actions durch den DQN Agenten ist maßgeblich vom  $\epsilon$  Wertes abhängig 2.4.1, welcher das Verhältnis zwischen der Wahl einer zufälligen und einer NN-basierten Action bestimmt. Dabei wird folgendermaßen vorgegangen 2.4.1. Zuerst wird ein Zufallswert *rand* zwischen null und eins generiert, welcher mit  $\epsilon$  verglichen wird. Wenn  $rand < \epsilon$  so wird die Action mittels des NN bestimmt. Anderenfalls wird eine zufällige Action gewählt. Zur NN-basierten Bestimmung der Action werden aus der Obs zwei Tensoren generiert, welche zum vordefinierten Device (CPU oder GPU) geschickt werden. Danach folge die Prozedur, welche in 7.2.1 dargestellt ist.

### Trainingsprozess

Der Trainingsprozess wird über die `learn` Methode ausgeführt und stellt sich dabei wie folge dar:

Zuerst wird überprüft, ob im Memory genügend Experiences (Exp) gespeichert sind, um einen Mini-Batch mit der zuvor definierten `batch_size`, zu extrahieren. Sollte dies nicht der Fall sein, wird die Methode terminiert. Anderenfalls wird ein Mini-Batch aus zufälligen Exp. ohne Duplikate gebildet.

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

Danach folgt die Bestimmung aller Q-Values der States (*s*) des Mini-Batch, wobei der Q-Value der zum State (*s*) gespeicherten Action entnommen und als `q_eval`, gespeichert wird. Anschließend werden alle Q-Values der Nachfolge-States (*s\_next*) bestimmt und als `q_next` gespeichert. Die Q-Values terminaler States werden anschließend auf null gesetzt, da die Discounted Sums of Rewards bis zum Ende der Spielepisode null entspricht, siehe 2.4.

Um nun `q-target` zu bestimmen, werden die maximalen Q-Values der *s\_next* bestimmt, diskontiert und mit dem, im Mini-Batch gespeicherten, Reward addiert, siehe 2.10. Nachfolgend wird der MSE (Mean Squared Error) zwischen `q_eval` und

q\_target bestimmt. Zuletzt wird der Fehler Mit Hilfe des PyTorch Frameworks mit Hilfe von Backpropagation und Gradient Descent A.1 zurück propagiert und entsprechend die Parameter des NN angepasst.

### Main-Methode

Die Hauptmethode oder auch main Methode genannt implementiert den eigentlichen Spiel und Trainingsablauf. Die main Methode des DQN ist im dqn Package im File dqn\_train und dqn\_play zu finden. Letzteres wird für das reine visuelle Spielen genutzt und wird daher nicht weiter betrachtet.

Um den Spiel- und Trainingsablauf durchzuführen, wird in dem File dqn\_train eine Methode train\_dqn definiert, welche wichtige Hyperparameter des Ablaufs, wie z.B. die Anzahl der zum Training zu absolvierenden Spiele (N\_ITERATIONS), die Lernrate (LR) und die Spielfeldgröße (BOARD\_SIZE), die Batch Size (BATCH\_SIZE), die Maximale Größe des Speichers (MAX\_MEM\_SIZE), Epsilon Decrement (EPS\_DEC) und Epsilon End (EPS\_END) übergeben bekommt.

Zu Beginn werden Datenhaltungslisten apples, wins, dtime, eps, steps\_list initialisiert, welche für jedes absolvierte Spiel die, dem Namen der List entsprechenden, Werte speichert. Nach der Erstellung des Agenten und Environments startet der Spielverlauf.

Dabei wird wie in 2.2.2 vorgegangen. Der Agent erhält eine Obs, bestimmt seine Action und diese wird sogleich im Env ausgeführt. Danach werden die neue Obs sowie Reward und weitere Statusinformationen, wie z.B. das done-Flag usw., ausgegeben. Diese Daten werden im Memory für das sich anschließende Training gespeichert. Dieses wird wie in 7.2.3 durchgeführt. Danach werden die oben genannten Datenhaltungslisten aktualisiert und die Prozedur beginnt von neuem. Wenn diese N\_ITERATIONS Spiele alle absolviert wurden werden die Erhobenen Daten aus den Listen in einem CSV-File gespeichert und die Methode wird terminiert.

### 7.2.4. PPO

Der PPO Algorithmus stellt die zweite Algorithmus-Art dar. Seine Implementierung findet im ppo Package, welches die Files actor, actor\_critic, critic, memoryPPO, ppo, ppo\_play und ppo\_train beinhaltet. actor und critic implementieren die entsprechenden NN, actor\_critic stellt eine Verwaltungs-Klasse dar, welche den Optimizer und die act (choose\_action) und die evaluate Methode beinhaltet, wobei letztere für den Trainingsprozess benötigt wird. MemoryPPO definiert das Memory (Replay Buffer), welches Erfahrungen speichert und im ppo File wird die Agentenklasse mit der learn Methode erstellt. ppo\_train und ppo\_play besitzen analoge Funktionalitäten wie in 7.2.3 eingangs erwähnt.



## Aktionsauswahlprozess

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

## Trainingsprozess

Der Trainingsprozess ist in der Agentenklasse implementiert, welche sich im `ppo` Package befindet. Zu Beginn wird aus dem Memory ein oder mehrere Mini-Batch/es generiert. Um den Return 2.3.2 zu erhalten, werden die einzelnen in Rewards diskontiert. Sollte ein terminalen Zustand vorhanden sein, werden die discounted Rewards auf null gesetzt, da diese dem zu erwarteten Reward bis zum Ende der Spielepisode entsprechen. Da die Episode terminiert, können keine weiteren Rewards mehr gesammelt werden.

Um ein gleichmäßigeres Lernen zu unterstützen, werden die Rewards nach dem sie zu einem Tensoren umgewandelt worden sind, normalisiert. Danach wird die folgende Prozedur (`K_epochs`) mal ausgeführt, um das NN zu trainieren. Danach terminiert die `learn` Methode

Zuerst wird die `evaluate` Methode der ActorCritic-Klasse aufgerufen welche sich im `actor_critic` File befindet. Dieser werden die Obs aus dem Mini-Batch bzw. Mini-Batches übergeben. Zusätzlich werden noch die alten korrespondierenden Actions übergeben, welche zur Bestimmung von  $\pi_{\theta}(a_t|s_t)$  2.3.2 dient. `Evaluate` gibt neben  $\pi_{\theta}(a_t|s_t)$  auch noch die State Values 2.3.2 und die Entropie der neuen Policy  $\pi_{\theta}$  aus, siehe 2.3.2.

2.3.2 entsprechend werden als nächstes die Probability Ratios über alle Erfahrungen des Mini-Batch/es bestimmt. Daraufhin folgt die Bestimmung der Advantages, welches entsprechend zu 2.3.2 geschieht. Nach der Berechnung beider Surrogates Loss wird die Surrogate Objective Loss gebildet 2.3.2.

Als nächstes erfolgt die Bestimmung des Value Loss 2.8 und die des Entropy Loss, wobei letzterer durch die `evaluate` Methode bereits bestimmt wurde.

Zum Schluss werden alle Losses entsprechend 2.3.2 zusammengefügt. Die NN's werden daraufhin durch mit Hilfe des PyTorch Frameworks durch Backpropagation und Gradient Descent A.1 angepasst.

Da der PPO mit unterschiedlichen Policy arbeitet, wird vor der Terminierung der

Trainingsmethode die alte Policy  $\pi_{\theta_{\text{old}}}$  durch die neu erzeugte  $\pi_{\theta}$  ersetzt. Trainiert wird immer auf der neuen Policy  $\pi_{\theta}$  und die Daten werden stets auf der alten  $\pi_{\theta_{\text{old}}}$  erhoben.

### **Main-Methode**

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

# Kapitel 8

## Optimierungen

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

### 8.1. Optimierung 1 - Dual Experience Replay

Die Idee für Dual Experience Replay oder auch hier Splited Memory genannt, stammt aus der Arbeit „Autonomous Agents in Snake Game via Deep Reinforcement Learning“ und wurde in 4.1 bereits analysiert und diskutiert.

Diese Optimierung zielt darauf ab, den Replay Buffer (Memory) zu zweiteilen, sodass Mem1 und Mem2 entstehen. In Mem1 werden ausschließlich Erfahrungen gespeichert, welche einen Reward vorweisen können, der größer als ein vordefinierter Grenzwert ist. In Mem2 werden alle übrigen Erfahrungen gespeichert. Diese Aufteilung zielt darauf ab, dass zu Beginn ein größerer Anteil an guten Erfahrungen für das Lernen verwendet wird, um das Lerntempo zu erhöhen. Den Verfassern schwebt ein Verhältnis von (80% guten und 20 % schlechteren Erfahrungen vor). Dieses Verhältnis normalisiert sich über die Trainingszeit, daher zum Verhältnis von (50% zu 50%).

Wie bereits in 4.1 angesprochen, ist die erfahrungsorientierte Aufteilung schlecht bis gar nicht umsetzbar, da die Rewards des PPO im Nachhinein noch diskontiert werden müssen. Dafür muss die komplette Episode an Erfahrungen in ihrer ursprünglichen Reihenfolge vorhanden sein, was nach dem Sortieren nicht der Fall wäre. Daher scheidet ein Sortieren Rewards aus.

Alternativ wurde vorgeschlagen, dass die Sortierung nicht belohnungsbasiert, sondern Episoden basiert geschehen kann. Es werden daher die besten Episoden, gemessen an ihren Scores, in die Memories einsortiert.

Zu diesem Zweck wird eine weitere Klasse `SplitMemory` in den Memory Files des DQN und PPO definiert, welchen diese beschriebene Funktionalität bereitstellt. Diese enthält drei Instanzen der ursprünglichen Memory-Klasse, `Mem1`, `Mem2` und ein temporären Memory. In letzteren werden die Erfahrungen temporär eingelagert bis die Episode terminiert ist. Dann wird anhand des gesamt Scores der Episode die Zuteilung in `Mem1` oder `Mem2` erfolgen.

Beim DQN Memory wird zusätzlich darauf geachtet, dass beim Einfügen einer ganzen Episode an Erfahrungen die Ring-Buffer Eigenschaften des Memory erhalten bleiben. Beim Memory des PPO ist dies nicht nötig.

## 8.2. Optimierung 2 - Joined Reward Function

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

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

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

$dis$  stellt die Distanz gemessen mit der Euklidischen Norm dar,  $len$  ist die Länge der Snake und  $size$  ist die Größe des Spielfeldes (bei einer Spielfeldform von 8x8 ergibt sich einer Größe von 64).

Da die Training Gap Strategy zu Problemen beim PPO führen würde, wird diese nicht beachtet. Die Timeout Strategy wird in abgewandelter Form, wie in 4.1.1 diskutiert, implementiert. Dies geschieht nach folgender Formel:

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

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

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

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

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

### 8.4. Optimierung 4 - Dynamische Lernrate

Die vierte Optimierung versucht das Lernen des Agenten durch das stetige Anpassen der Lernrate (LR) zu verbessern. Mit Hilfe eines vom PyTorch bereitgestellten Schedulers lässt sich die Lernrate während des Trainingsprozess manipulieren. Dabei wird diese Optimierung in der main Methode implementiert.

Die Lernrate wird immer dann mit 0.95 multipliziert und als neue LR gesetzt, sobald keine Performance Steigerung in den lernen 200 Schritten erfolgt ist.

# **Kapitel 9**

## **Evaluation**

# Literaturverzeichnis

- [Baker u. a. 2019] BAKER, Bowen ; KANITSCHIEDER, Ingmar ; MARKOV, Todor M. ; WU, Yi ; POWELL, Glenn ; MCGREW, Bob ; MORDATCH, Igor: Emergent Tool Use From Multi-Agent Autocurricula. In: *CoRR* abs/1909.07528 (2019). – URL <http://arxiv.org/abs/1909.07528>
- [Bowe Ma 2016] BOWEI MA, Jun Z.: *Exploration of Reinforcement Learning to SNAKE*. 2016. – URL <http://cs229.stanford.edu/proj2016spr/report/060.pdf>
- [Chunxue u. a. 2019] CHUNXUE, Wu ; JU, Bobo ; WU, Yan ; LIN, Xiao ; XIONG, Naixue ; XU, Guangquan ; LI, Hongyan ; LIANG, Xuefeng: UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene. In: *IEEE Access* PP (2019), 08, S. 1–1. – URL <https://ieeexplore.ieee.org/abstract/document/8787847>
- [Goodfellow u. a. 2018] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning. Das umfassende Handbuch*. MITP Verlags GmbH, 2018. – URL [https://www.ebook.de/de/product/31366940/ian\\_goodfellow\\_yoshua\\_bengio\\_aaron\\_courville\\_deep\\_learning\\_das\\_umfassende\\_handbuch.html](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](https://www.ebook.de/de/product/37826629/maxim_lapan_deep_reinforcement_learning.html). – ISBN 3747500366
- [Mnih u. a. 2016] MNIH, Volodymyr ; BADIA, Adrià P. ; MIRZA, Mehdi ; GRAVES, Alex ; LILICRAP, Timothy P. ; HARLEY, Tim ; SILVER, David ; KAVUKCUOGLU, Koray: Asynchronous Methods for Deep Reinforcement Learning. In: *CoRR* abs/1602.01783 (2016). – URL <http://arxiv.org/abs/1602.01783>

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



# Anhang A

## Anhang

### A.1. Backpropagation und das Gradientenverfahren

Nachdem nun alle Agenten-Klassen vorgestellt sind, man sich vielleicht der eine oder andere Leser frage, wie denn nun das eigentliche Lernen vonstattengeht. Die dem Lernen zugrunde liegenden Verfahren sind das Backpropagation und das Gradient Descent. Dabei wird häufig fälschlicherweise angenommen, dass sich hinter dem Begriff Backpropagation der komplette Lernprozess verbirgt. Dem ist jedoch nicht so. Der Backpropagation-Algorithmus oder auch häufig einfach nur Backprop genannt, ist der Algorithmus, welcher zuständig für die Bestimmung der Gradienten in einer Funktion. Häufig wird ebenfalls angenommen, dass Backprop nur für NN anwendbar sind, den ist jedoch nicht so. Prinzipiell können mit dem Backprop-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. Basierend auf dieser Funktion lässt sich eine zweite Funktion definieren, die Loss Function oder Kostenfunktion oder Verlustfunktion usw. Diese gibt den Fehler wieder und soll im Optimierungsverlauf minimiert werden, um optimale Ergebnisse zu erhalten. Diese Fehlerfunktion zu minimieren müssen die Gewichte des NN soweit angepasst werden, der die Fehlerfunktion geringe Werte ausgibt. Ist diese für alle Daten, mit welchen das NN jemals konfrontiert wird geschafft, so ist das NN perfekt angepasst (Goodfellow u. a., 2018, S. 225ff.).

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

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

## A.2. Tensoren

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

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

## A.3. Convolution Neural Networks

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

### A.3.1. Convolutional Layer

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

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

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

Die Funktionsweise der Convolutionalen Prozedur stellt sich wie folgt dar:

Jeder einzelne Kernel wird mit der Eingabe entsprechend A.1 multipliziert und addiert. Ist dieser Berechnungsschritt abgeschlossen, bewegt sich das Eingabequadrat um dem sogenannten Stride weiter nach rechts (in der Grafik wird ein Stride von eins verwendet). Sollte ein Stride von zwei verwendet werden, so würde die Ausgabe  $bw + cx + fy + gz$  nicht existieren und die resultierende Feature Map wäre kleiner. Daher wird der Stride gerne dazu verwendet, um die Feature Map Size und damit den Berechnungsaufwand zu senken. Des Weiteren lässt sich der Stride nicht nur in der Horizontalen sondern auch in der Vertikalen anwenden.

Nicht in A.1 abgebildet ist das sogenannte Padding. Bei diesem wird an den Feature Maps weitere Nullzeilen bzw. Nullspalten angefügt. Dabei kann an allen vier Seiten oder an speziell ausgewählten Zeilen bzw. Spalten hinzuaddiert werden. Wie in A.1 zu erkennen ist, hat die Feature Map die Form (3x4), jedoch ist der Output nur noch von der Form (2x3). Die Durchführung der Convolution Prozedur sorgt für eine Verkleinerung, welche durch Padding verhindert werden kann. (Goodfellow u. a., 2018, S. 369 ff.) Die Ausgaben in A.1 bilden eine Feature Map, welche mit allen weiteren Feature Maps zusammenaddiert werden müsste um einen Output Channel zu bilden.

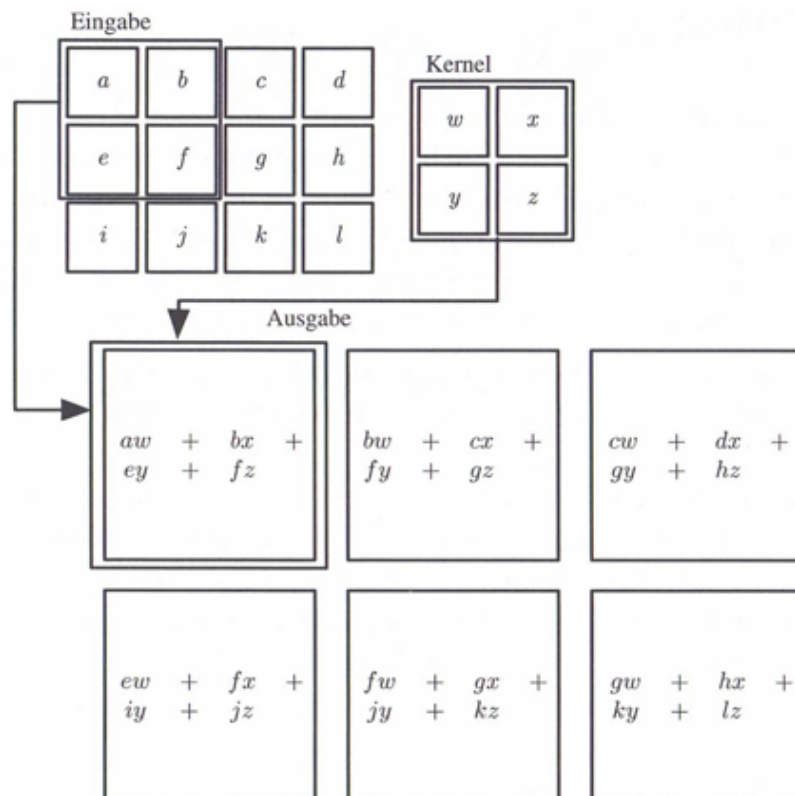


Abbildung A.1.: Darstellung der Berechnung einer 2-D-Faltung bzw. einer convolutionalen Prozedur. (Goodfellow u. a., 2018, S. 373)

### A.3.2. Pooling Layer

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

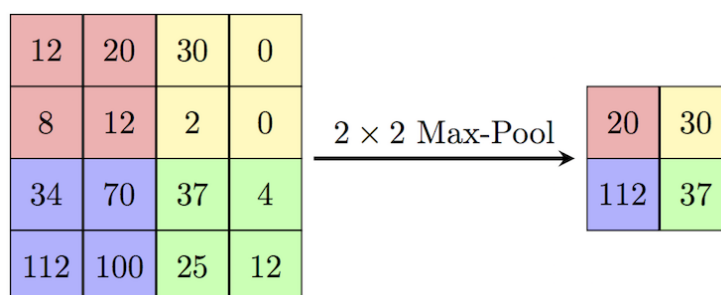


Abbildung A.2.: Darstellung des Max-Poolings. <https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png>

### A.3.3. Fully Connected Layer

Die Fully Connected Layer (FC) sind die grundlegendsten Elemente eines NN. Ein solches Layer besteht aus zwei Teilen, die beide einer Initialisierung benötigen. Die Gewichte des FC sind als Tensor mit der Form (Anzahl der Output Features, Anzahl der Input Features) initialisiert. Zusätzlich 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

## 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.

### B.1. PPO Train Startargumente

1. "PPO" → Algorithmus-Art
2. `N_ITERATIONS` (Int) → Anzahl l der zu spielenden Spiele
3. `LR_ACTOR` (Float) → Lernrate der Actor-NN
4. `LR_CRITIC` (Float) → Lernrate des Critic-NN
5. `GAMMA` (Float) → Abzinsungsfaktor 2.1
6. `K_EPOCHS` (Int) → Gibt die Anzahl der Lernzyklen eines Batches bzw. Spieles an. Siehe 2.3.3
7. `EPS_CLIP` (Float) → Clip Faktor, welcher St Standard bei 0.2. Siehe 2.3.2
8. `BOARD_SIZE` (Tuple of Ints) → Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
9. `PATH` (String) → Path an dessen Stelle ein neuer Ordner erstellt wird mit dem erlernten Modell sowie der Trainingsstatistik und den Trainingsdaten. Wenn der Standard Path verwendet werden soll: übergeben.

10. DO\_TOTAL\_RUN (Boolean) → Wenn False, wird nur bis zu dem Zeitpunkt gelernt, an dem der Agent unter den zehn letzten Spielen sechs Siege erreichen hat. Ansonsten wird ein kompletter Trainingsrun durchgeführt.
11. GPU (Boolean) → Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Trainingsprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

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

## B.2. DQN Train Startargumente

1. "DQN" → Algorithmus-Art
2. N\_ITERATIONS (Int) → Anzahl l der zu spielenden Spiele
3. LR (Float) → Lernrate der Q-NN
4. GAMMA (Float) → Abzinsungsfaktor 2.1
5. BATCH\_SIZE (Int) → Größe des zu entnehmenden Batches 2.4.1
6. MAX\_MEM\_SIZE (Int) → Maximale Größe des Memory.
7. EPS\_DEC (Float) → Der Absenkungsfaktor von Epsilon 2.4.1
8. EPS\_END (Float) → Der Endwert von Epsilon 2.4.1
9. BOARD\_SIZE (Tuple of Ints) → Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
10. PATH (String) → Path an dessen Stelle ein neuer Ordner erstellt wird mit dem erlernten Modell sowie der Trainingsstatistik und den Trainingsdaten. Wenn der Standard Path verwendet werden soll: übergeben.
11. DO\_TOTAL\_RUN (Boolean) → Wenn False, wird nur bis zu dem Zeitpunkt gelernt, an dem der Agent unter den zehn letzten Spielen sechs Siege erreichen hat. Ansonsten wird ein kompletter Trainingsrun durchgeführt.
12. GPU (Boolean) → Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Trainingsprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

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

## B.3. Play Startargumente

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

1. "DQN/ "PPO" → Algorithmus-Art
2. PATH (String) → Path der Model Datei.
3. N\_ITERATIONS (Int) → Anzahl l der zu spielenden Spiele.
4. BOARD\_SIZE (Tuple of Ints) → Spielfeldgröße bzw. Spielfeldform. Z.B. "(8, 8)"
5. PRINT\_STATS (Boolean) → Wenn True, werden Informationen über den abgeschlossenen Spielverlauf auf der Konsole erscheinen.
6. HAS\_GUI (Boolean) → WENN True wird die GUI 7.1.3 dargestellt.
7. GPU (Boolean) → Wenn True und eine CUDA-fähige Grafikkarte vorhanden ist, wird der Spielprozess auf der Grafikkarte ausgeführt.

So könnte ein Start mittels Kommandozeile aussehen:

```
Path_to_File\Bachelor-Snake-AI\src\python main_play.py "PPO"  
"Path_to_Model" 300 "(8, 8)" True True 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