Das Videospiel Snake mit Reinforcement Learning lösen

Ein experimenteller Vergleich von Deep Q-Learning, SARSA und PPO

**Projektbericht**

Methoden der KI

Markus Dauth

27.07.2022

**Abstract**

In dieser Projektarbeit werden die drei Reinforcement Learning Algorithmen Deep Q-Learning, SARSA und PPO anhand des Videospiels Snake verglichen. Es wird eine Trainingsumgebung implementiert, mit der ein einheitliches Training der drei Agenten durchgeführt wird. Ziel der RL-Agenten soll es sein, einen hohen Punktetand durch das Einsammeln von Äpfeln in der Snake-Welt zu erreichen.

Es erfolgt eine Diskussion möglicher Abbildungen für den State-, Action- und Reward-Space. Bei den Experimenten wurden zunächst Hyperparameter definiert, mit denen die drei Algorithmen in vier Szenarien trainiert wurden. Die Szenarien unterscheiden sich jeweils in der Spielfeldgröße. Ergebnisse der Experimente ist, dass bei Trainingszeiten von unter einer Stunde der einfache regelbasierte Agent (basierend auf dem euklidischen Abstand zwischen Schlange und Apfel) besser abschneidet als die RL-Verfahren. PPO ist der mit Abstand beste Algorithmus, während Deep Q-Learning den SARSA-Algorithmus bei großen Spielwelten schlägt.

# Inhalt

[1. Inhalt 2](#_Toc109807041)

[2. Einführung 4](#_Toc109807042)

[2.1. Erwartungen an den Leser 4](#_Toc109807043)

[2.2. Zielsetzungen 4](#_Toc109807044)

[2.2.1 Implementierung der Trainingsumgebung 4](#_Toc109807045)

[2.2.2 Implementierung der RL-Algorithmen 4](#_Toc109807046)

[2.2.3 Experimente 4](#_Toc109807047)

[2.2.4 Fragestellungen 5](#_Toc109807048)

[3. Grundlagen 5](#_Toc109807049)

[3.1. Das Spiel Snake 5](#_Toc109807050)

[3.1.1 Die Snake Welt 5](#_Toc109807051)

[3.1.2 Der beste Score 6](#_Toc109807052)

[3.2. Auswahl der Algorithmen 6](#_Toc109807053)

[3.2.1 Randbedingungen zur Auswahl der Algorithmen 6](#_Toc109807054)

[3.2.2 Gewählte Algorithmen 7](#_Toc109807055)

[3.3. Deep Q-Learning (DQN) 7](#_Toc109807056)

[3.4. SARSA (State–action–reward–state–action) 8](#_Toc109807057)

[3.5. PPO (Proximal Policy Optimization) 8](#_Toc109807058)

[3.6. Unterschiede von DQN, SARSA und PPO 8](#_Toc109807059)

[4. Methoden 9](#_Toc109807060)

[4.1. Das Videospiel Snake als Benchmark 9](#_Toc109807061)

[4.2. Vergleich der Algorithmen 9](#_Toc109807062)

[4.3. Hypothesen über die Performanz der Algorithmen 10](#_Toc109807063)

[5. Komplexität und Modelbildung 10](#_Toc109807064)

[5.1. Komplexität der Snake-Spielwelt 10](#_Toc109807065)

[5.2. Diskussion der Modellierung der Spaces 10](#_Toc109807066)

[5.2.1 Symmetrie der Spielwelt 10](#_Toc109807067)

[5.2.2 Größere State Spaces 11](#_Toc109807068)

[5.2.3 Größe der Spielwelt 11](#_Toc109807069)

[5.2.4 Modellierung des Rewards 11](#_Toc109807070)

[5.3. Modellierung des State und Action Spaces 11](#_Toc109807071)

[5.3.1 Mögliche Modellierungen 11](#_Toc109807072)

[5.3.2 Implementierung des State Spaces 12](#_Toc109807073)

[5.3.3 Implementierung des Action Spaces und Reward 12](#_Toc109807074)

[6. Implementierung 12](#_Toc109807075)

[6.1. Trainings- und Testablauf 13](#_Toc109807076)

[6.2. Klassen, Dateien und Ordner 14](#_Toc109807077)

[6.3. Implementierungen der Algorithmen 15](#_Toc109807078)

[6.3.1 Implementierung Regelbasierter Algorithmus (Euklidischer Abstand) 15](#_Toc109807079)

[6.3.2 Implementierung DQN und SARSA 16](#_Toc109807080)

[6.3.3 Implementierung PPO 16](#_Toc109807081)

[7. Aufbau der Versuche 16](#_Toc109807082)

[8. Zusammenfassung der Versuche 18](#_Toc109807083)

[8.1. Spielweltgröße 6 18](#_Toc109807084)

[8.2. Spielweltgröße 12 20](#_Toc109807085)

[8.3. Spielweltgröße 24 20](#_Toc109807086)

[8.4. Spielweltgröße 48 21](#_Toc109807087)

[8.5. Erkenntnisse 21](#_Toc109807088)

[8.5.1 Betrachtung der Trainingsläufe 21](#_Toc109807089)

[8.5.2 Betrachtung der Scores 22](#_Toc109807090)

[8.5.3 Betrachtung der Timesteps 22](#_Toc109807091)

[9. Fehlerbetrachtung 23](#_Toc109807092)

[10. Aussicht 23](#_Toc109807093)

[11. Zusammenfassung 24](#_Toc109807094)

[12. Literaturverzeichnis 25](#_Toc109807095)

# Einführung

Das Videospiel *Snake* ist ein Singleplayer-Spiel, das in den 90ern bekannt wurde, da es standardmäßig mit dem Handy Nokia 3310 ausgeliefert wurde. In diesem Projekt werden anhand von Snake die drei Reinforcement Learning (RL) Algorithmen Deep Q-Learning, SARSA und PPO experimentell miteinander verglichen. Zusätzlich wird ein regelbasierter Algorithmus als Referenz implementiert.

Es wurde eine Python-Trainingsumgebung entwickelt, in der einheitlich die vier Agenten einfach trainiert werden können. Ergebnisse von Trainingsläufen werden in einer Excel-Tabelle dokumentiert.

## Erwartungen an den Leser

Für diesen Projektbericht wird ein Grundwissen von Reinforcement-Learning vorausgesetzt. Folgende Begriffe und Konzepte sollten dem Leser bekannt sein:

* Agent und Environment
* Action- und Observation-Space
* Discrete und Continous Spaces
* Policy, Hyperparameter
* Neuronale Netze, Hidden Layers

## Zielsetzungen

### Implementierung der Trainingsumgebung

Es muss eine geeignete Trainingsumgebung erstellt werden, mit der einfach mehrere RL-Agenten trainiert werden können. Hierfür wird eine bereits vorhandene Python-Implementierung des Spiels verwendet. Der RL-Anteil wird möglichst austauschbar gestaltet. Es wird eine einfache ausschaltbare Visualisierung der Spielwelt geben.

### Implementierung der RL-Algorithmen

Unterschiedliche RL-Algorithmen werden im Laufe des Projektes implementiert und experimentell miteinander verglichen. Folgende state-of-the-art RL-Algorithmen werden untersucht:

* **Deep Q-Learning** (Deep Quality-Learning, DQN)
* **SARSA** (State-Action Reward State-Action)
* **PPO** (Proximal Policy Optimization)

Für Referenzwerte wird ein regelbasierter Algorithmus implementiert. Dieser regelbasierte Algorithmus nimmt immer die kürzeste Distanz zum Apfel (euklidischer Abstand), es sei denn, ein Hindernis ist im weg.

### Experimente

Die Algorithmen werden anhand von Experimenten verglichen. Es werden früh im Projektverlauf Testläufe durchgeführt, um Parameter für weitere Experimente zu definieren. Ziel dieser Parameterdefinierung ist es, passende Konfiguration für die RL-Algorithmen zu finden, die ein annehmbares Training durchführen (unter schlecht gewählten Parametern könnten ggf. die Agenten nicht trainieren). Zudem können sich diese Parameter zwischen den Agenten unterscheiden. Die Parameter werden so gewählt, dass die Agenten unter fairen Bedingungen trainieren (z.B. durch weniger Episoden für einen Agent, der länger pro Episode trainiert).

Es werden vier Szenarien früh im Projektverlauf definiert mit der die Algorithmen verglichen werden. Für jedes Szenario unterscheidet sich lediglich die Spielfeldgröße. Abhängig von der Spielfeldgröße sollten Unterschiede über die Performanz der Agenten sichtbar werden.

Ziel des Agenten ist es, einen möglichst großen Punktestand (gefressene Äpfel) zu erreichen. Hieraus leidet sich auch die Reward-Funktion ab: Das Essen der Äpfel wird belohnt und das Laufen in Hindernisse (Körper der Schlange oder Wände der Spielwelt) wird bestraft. Ein Vergleich von Trainingsläufen kann anhand des erzielten Punktestandes einfach vorgenommen werden. Zusätzlich wird die Anzahl der benötigten Spielzüge ausgewertet.

Für die Experimente werden die Algorithmen unter gleichen Szenarien verglichen. State-Space, Action-Space, Reward-Space und Spielfeldgröße sind identisch für alle Algorithmen. Hyperparameter (z.B. Anzahl der Trainingsepisoden, Learn Rate) sind identisch, soweit dies möglich ist.

### Fragestellungen

Folgenden Fragen sollen in diesem Projekt beantwortet werden:

1. Welcher der ausgewählten RL-Algorithmen eignet sich am besten, um das Computerspiel Snake zu lösen?
2. Unterscheidet sich die Performanz der RL-Algorithmen? Falls ja, warum?
3. Welche Spielfeldgrößen sind für die RL-Algorithmen machbar? Wie skalieren die Algorithmen mit der Spielfeldgröße?
4. Können die RL-Algorithmen einen einfachen regelbasierten Algorithmus schlagen? Können sie die maximale Punktezahl erreichen?

# Grundlagen

In diesem Kapitel werden die Regeln des Spiels Snake erklärt. Es folgt eine Begründung zur Auswahl der Algorithmen für den Vergleich dieser Projektarbeit. Die Funktionsweise von DQN, SARSA und PPO werden kurz erläutert und deren Unterschiede vorgestellt.

## Das Spiel Snake

### Die Snake Welt

Snake ist ein Singleplayer Spiel, bei dem eine Schlange zugbasiert von einem Menschen gesteuert wird. Ziel ist es, mit der Schlange so viele Äpfel wie möglich zu essen, ohne in ein Hindernis zu laufen. Als Hindernisse zählt der Körper der Schlange oder der Rand der Spielwelt.

Das Spielfeld besteht aus mehreren Kacheln (oder „Tile“). Die Spielfeldgröße variiert je nach Implementation (z.B. 18x14 oder 20x20). Eine Kachel kann leer sein oder eins von drei Objekten enthalten: Kopf der Schlange, Körper der Schlange oder ein Apfel.

Zu Spielbeginn befindet sich die Schlange in der Mitte des Spielfeldes und ein Apfel wird auf einer zufälligen Kachel erzeugt. Es kann immer nur ein Apfel gleichzeitig existieren. Die Schlange bewegt sich pro Spielzug („Step“, „Timestep“ oder „Takt“) ein Feld. Abhängig von der Eingabe des Spielers kann sich die Schlange in eine Richtung bewegen: hoch, links, rechts oder runter. Dies sind auch alle vier Aktionen, die ein Agent ausführen kann. Frisst die Schlange einen Apfel, erhöht sich die Punkteanzahl („Score“) um Eins und der Körper der Schlange wird ein Feld länger. Läuft die Schlange in ein Hindernis, wird das Spiel beendet (der Score verändert sich nicht).

Der Kopf der Schlange zeigt in eine Richtung. Üblicherweise ist die Zeit pro Zug für den Spieler begrenzt: Erfolgt keine Eingabe des Spielers, bewegt sich die Schlange geradeaus. Auf höheren Schwierigkeitsstufen hat der menschliche Spieler weniger Zeit, um rechtzeitig die richtigen Eingaben einzugeben. Bei den RL-Experimenten in diesem Projekt wird jedoch dieser zeitliche Aspekt der Züge nicht weiter betrachtet. Die RL-Agenten haben für einen Zug unendlich viel Zeit. Somit kann der Agent für jeden Zug seine optimale Aktion wählen.

Ziel der Agenten ist es, möglichst viele Äpfel zu essen (ohne einen zeitlichen Aspekt). Der Agent muss somit lernen in Richtung Apfel zu laufen und Hindernisse zu vermeiden.

Der Kopf der Schlange zeigt immer in eine von vier Richtungen. Um Verwirrungen zu vermeiden, wird im Folgenden zwischen Himmelsrichtungen („cardinal direction“) und relative Kopfrichtung unterscheiden: zeigt beispielsweise der Kopf der Schlange Richtung Osten und ein Apfel ist nordöstlich von der Schlange, dann ist die relative Richtung zum Apfel aus Sicht des Kopfes „geradeaus links“. Diese Unterscheidung wird bei der Abbildung des Observation Spaces wichtig (siehe Kapitel 5).

### Der beste Score

Der größtmögliche Score ist abhängig von der Größe des Spielfeldes. Da die Schlange pro gefressenen Apfel ein Feld größer wird, geht der Schlange im späteren Spielverlauf der Platz aus. Die Schlange besteht zu Spielbeginn aus zwei Körperteilen und dem Kopf. Beispielsweise besteht ein 10x10 Feld aus 100 Feldern. Hat die Schlange 96 Äpfel (Score = 96) gegessen, ist nur noch ein Feld frei, auf dem ein Apfel erzeugt werden kann. Frisst die Schlange diesen letzten Apfel, beißt die Schlange sich in den Schwanz (letztes Körperteil), erreicht den höchstmöglichen Highscore von 97 und das Spiel wird beendet.

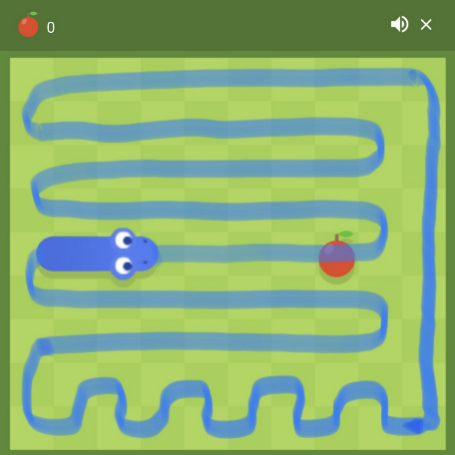
Ein wichtiger Bestandteil für einen rational handelnden Agenten ist das Vermeiden von Sackgassen. Wird die Schlange groß genug, kann sich die Schlange mit ihrem Körper „einbauen“. Es existieren Algorithmen, die immer den höchstmöglichsten Punktestand erreichen ohne einen Fehler. Abbildung 1 zeigt ein Beispiel für solch einen Algorithmus: die Schlange folgt immer der eingezeichneten Linie, unabhängig davon wo der Apfel ist. Da keine Sackgassen entstehen können und alle Felder wiederholt abgelaufen werden, frisst Schlange garantiert jeden Apfel. Dieser Algorithmus veranschaulicht auch, dass die kürzeste Distanz zum Apfel nicht immer die beste Strategie darstellt.

Abbildung 1 Optimaler Algorithmus[[1]](#footnote-1)

## Auswahl der Algorithmen

In diesem Abschnitt wird dargestellt, warum die Algorithmen DQN, SARSA und PPO für den Vergleich ausgewählt wurden. Details zur Funktionsweise der Algorithmen werden in den darauffolgenden Abschnitt erläutert.

### Randbedingungen zur Auswahl der Algorithmen

Die Auswahl an Algorithmen ist durch die Eigenschaften des Spiels eingeschränkt. Wichtig sind hierbei die Betrachtung von discrete/continous Action- und State-Spaces (siehe [1]).

* **Diskreter Action-Space**: Es gibt vier Aktionen die ein Agent pro Spielzug (Tick) ausführen kann: Der Spieler steuert, in welche Richtung die Schlange sich bewegt (links, rechts, hoch, runter)
* **Diskreter State-Space**: Das Spielfeld ist ein diskretes Gatterfeld (“Grid”), in dem sich an einer Position der Apfel oder ein Teil der Schlange befinden kann. Beispielsweise könnte sich hier ein 20x20 Feld eignen.
* **Single-Agent**: Es gibt nur eine Schlange, die gesteuert wird. Pro Tick wählt der Agent eine Aktion aus.
* **On-policy / off-policy:** Es soll jeweils mindestens ein on-policy und ein off-policy Algorithmus untersucht werden. Aufgrund der inhärenten Unterschiede zwischen on-/off-policy sollten sich unterschiedliche Verhaltensweisen beim Training der Agenten auftreten.

Weitere Randbedingungen, bedingt durch den zeitlichen Rahmen dieser Projektarbeit:

* **Einfache Implementierung**: Die Agenten sollten mit dem Snake Environment einfach zu implementieren sein.
* **Lernen ohne „Finetuning“:** Die Agenten sollen ohne viel Konfigurationsaufwand lernen.
* **State of the art**: Die Algorithmen sind bewährt und werden oft in der Forschung und Wirtschaft angewandt.

### Gewählte Algorithmen

Im Folgenden erfolgt eine Auflistung der ausgewählten Algorithmen mit einer kurzen Begründung.

**Regelbasierter Algorithmus (Euklidischer Abstand):**In der Literatur implementieren viele Vergleiche von RL-Algorithmen einen regelbasierten Algorithmus als Referenzwert. Solch ein Algorithmus wird beispielsweise in [2], [3] und [4] verwendet. Der Algorithmus soll sich dem Apfel anhand des euklidischen Abstands zwischen Schlangenkopf und Apfel nähern.

**Deep Q-Learning (DQN) :**Q-Learning ist einer der häufigsten verwendeten off-policy RL-Algorithmus bei Videospielen. Beispielsweise wurde dieser von [5] bei Snake angewandt und konnte das Spielfeld zu 46% füllen.

**SARSA:**Im Gegensatz zu Q-Learning und PPO ist SARSA on-policy. SARSA wird beispielsweise in [6] verwendet. SARSA verhält sich ähnlich wie DQN. SARSA und DQN lassen sich einfach zusammen implementieren, weswegen sich hier ein direkter Vergleich anbietet.

**PPO:**PPOist derzeit einer der besten RL-Algorithmen und schneidet meist besser als andere RL-Algorithmen ab (siehe [7]). PPO ist ein beliebter state-of-the-art on-policy Algorithmus, der in vielen Environments angewandt wird, da er meist ohne viele Parametrisierung lernt. Im Gegensatz zu DQN und SARSA (beide Temporal Difference Algorithmen) ist PPO eine Policy Gradient Methode und ist somit ein interessanter Kontrast für diesen Vergleich, da hier das Aktualisieren der Policy fundamental anders erfolgt.

## Deep Q-Learning (DQN)

Q-Learning wurde das erste Mal im Jahre 1989 vorgestellt. Paare von States und Actions werden in einer Tabelle („Q-Table“) gespeichert. Bei Deep Q-Learning wird die Tabelle durch ein neuronales Netz ersetzt. Da PPO immer mit einem neuronalen Netz arbeitet, wird hier für einen fairen Vergleich auch der „deep“ Ansatz verwendet. Eine Verwendung einer Q-Table kann sich für kleine Observation-Spaces eignen, jedoch sind mögliche Observation-Space bei Snake zu groß.

DQN ist ein Temporal Difference Learning Algorithmus: Der Agent bewertet den Nutzen einer Aktion und nicht den Nutzen eines Zustandes.

Folgende Formel beschreibt die Funktionsweise von DQN:   


* S – aktueller Zustand (State)
* A – aktuelle Aktion
* R- Reward (Belohnung)
* S‘ – neuer Zustand
* A – zukünftige Aktion

DQN arbeitet mit zwei Policies zur Auswahl des aktuellen Zustand S und dem Folgezustand S‘ (DQN ist somit off-policy). S wird mit einer „Behavior Policy“ gewählt, während für S‘ eine „Greedy Policy“ verwendet wird, siehe maxQ(s‘,a‘). DQN macht somit eine Annahme über Belohnungen zukünftiger Zustände, jedoch kann sich die tatsächliche gewählte Strategie davon abweichen. Sieht beispielsweise DQN, dass die nächste Aktion wahrscheinlich zu einer hohen Bestrafung führt, dann führt der Agent diese Aktion mit einer geringen Wahrscheinlichkeit aus (SARSA macht dies nicht).

## SARSA (State–action–reward–state–action)

SARSA wurde 1994 das erste Mal vorgestellt. SARSA ist wie DQN ein Temporal Difference Learning Algorithmus. DQN ist on-policy, da die Policy für das Aktualisieren und die Policy für das Auswählen von Aktionen gleich sind. S und S‘ werden mit der gleichen Policy gewählt.

Folgende Formel beschreibt die Funktionsweise von SARSA:



Im Gegensatz zu DQN wird hier Q(s‘,a‘) ohne eine Annahme (maxQ) gewählt.

## PPO (Proximal Policy Optimization)

PPO ist ein on-policy Algorithmus, der 2017 das erste Mal von OpenAI vorgestellt wurde. Im Gegensatz zu DQN und SARSA ist PPO eine Policy Gradient Methode: Anstatt einzelner State-Action-Paare werden hier ein „Raum“ (Space) von Policies untersucht.

Der Agent trainiert nicht mit jedem Schritt (Timestep), sondern es werden die Erfahrungen mehrerer Schritte gesammelt, bevor eine Policy-Aktualisierung stattfindet. PPO verwendet zwei Policies: eine Policy, die aktualisiert werden soll und eine Policy mit der Proben gesammelt werden. Nur die aktuellen Übergänge („Transitions“) werden alle x Schritte für das Aktualisieren der Policy verwendet. Siehe [7] für weitere Details zur Funktionsweise von PPO.

PPO ist relativ einfach zu implementieren und generell anwendbar im Vergleich zu anderen Policy Gradient Methods, wie z.B. TRPO.

## Unterschiede von DQN, SARSA und PPO

Es folgt eine Auflistung einiger wichtige Unterschieden zwischen den Algorithmen:

* Q-Learning lernt direkt die optimale Strategie, während SARSA die nahezu optimale Strategie lernt. SARSA macht so mehr Exploration, während Q-Learning schneller seine Policy anpasst.
* Off-policy Algorithmen haben eine höhere Varianz per Sample, wodurch Probleme beim Konvergieren auftreten können.
* Off-policy Varianten vermeiden eher mögliche Bestrafungen. Ist die optimale Strategie nahe an Bestrafungen dran, versuchen Off-policy Algorithmen anfangs diese gefährlichen Situationen zu vermeiden. Ein bekanntes Beispiel hierfür das Cliff Walking Problem: <https://github.com/cvhu/CliffWalking>
* PPO trainiert nur alle X Steps, während DQN und SARSA mit jedem Step trainieren. PPO wird somit weniger Trainingszeit pro Episode benötigen.

# Methoden

## Das Videospiel Snake als Benchmark

Videospiele werden oft als Benchmark benutzt, um RL-Algorithmen zu vergleichen [8]. So dienen beispielsweise Spiele der Konsole „Atari 2600“ als Benchmark für Single-Agent RL-Algorithmen mit diskreten Action-Spaces [9].

Snake ist von der Komplexität her vergleichbar mit Atari 2600 Spielen und eignet sich aus den folgenden Gründen als Benchmark für RL-Algorithmen:

* **Wenig Zufall** (im Vergleich zu anderen Spielen): Die Äpfel erscheinen immer an einem zufälligen Ort, entweder bei Spielstart oder wenn die Schlange einen Apfel gegessen hat. Dies ist die einzige Art an Zufall im Snake Spiel. Der Startpunkt der Schlange ist in der Mitte des Spielfeldes mit dem Blickwinkel nach rechts.
* Durch die gegebene Spielwelt (Grid, Schlange, Apfel) ist das Spiel ist **vollständig beobachtbar**.
* Das Spiel ist **leicht zu implementieren**. Es gibt etliche Open-Source Repositories im Internet. Der Code für die Spiellogik inklusive grafischer Oberfläche ist meist unter 200 Codezeilen lang. (siehe z.B. [10])
* Die **Spielfeldgröße ist variabel**, wodurch auch der Action- und State-Space eingegrenzt werden kann. Durch eine kleine Spielfeldgröße sollte ein vollständiges Lernen möglich sein. Dies sollte auch ermöglichen, dass ein selbstständiges Trainieren der RL-Algorithmen möglich ist.
* Ein **Vergleich an Algorithmen ist simpel**. Anhand der erreichten Spielpunkte ist messbar, wie erfolgreich ein Algorithmus ist.
* Snake ist für RL-Algorithmen **herausfordernd**, aufgrund der großen Anzahl an möglichen Spaces. Während des Spielverlaufs wächst der Körper der Schlange, wodurch sich die Spielzustände dynamisch im Spielverlauf verändern.

Problematisch ist jedoch, dass die einzige „positive“ Aktion von Snake das Aufsammeln von Äpfeln ist. Findet der Agent nur selten Äpfel, z.B. wenn die Spielfeldgröße sehr groß ist, dann wird der Agent selten belohnt. Bei großen Spielfeldern ist es also möglich, dass der Agent nicht trainiert oder das Training sehr lange dauert.

## Vergleich der Algorithmen

Insgesamt werden vier verschiedene Algorithmen (regelbasiert, DQN, SARSA und PPO) anhand von vier festgelegten Szenarien getestet. Ein Szenario wird durch die Spielfeldgröße definiert. Die kleinstmögliche Spielfeldgröße mit der verwendeten Snake-Implementation ist 6x6. Die weiteren zu Spielfeldgrößen werden jeweils verdoppelt, wodurch sich die vier Szenarien aus den Spielfeldgrößen 6x6, 12x12, 24x24 und 48x48 ergeben.

Ziel von Snake ist, einen möglichen hohen Punktestand (Highscore) zu erreichen. Je mehr Punkte ein Algorithmus im Durchschnitt erzielt, bevor das Spiel verloren ist, desto besser ist der Algorithmus. Abhängig von der Spielfeldgröße gibt es einen maximalen Highscore: wenn die Schlange die maximale Länge erreicht hat, gibt es keine freien Felder mehr. Ein Algorithmus, der den maximalen Highscore erreicht, hat somit das Spiel gelöst. Sollten mehrere Algorithmen das Spiel lösen, dann gewinnt der Algorithmus der Schneller (in weniger Spielschritten) eine Lösung findet.

Es werden bereits vorhandene Implementierungen der Algorithmen verwendet. Für die unterschiedlichen Algorithmen fallen ggf. Codeänderungen an. Pro Spieldurchlauf wird zudem die Anzahl der Züge gemessen.

## Hypothesen über die Performanz der Algorithmen

Folgende Hypothesen lassen sich anhand von Literaturbeispielen ableiten:

1. **Regelbasierter Algorithmus:** Laut [2] hat ein Algorithmus mit der Manhattan-Distanz in großen Szenarien eine bessere Leistung erzielt als Q-Learning und SARSA. In kleineren Szenarien sollte der Regelbasierte schlechter abschneiden als die RL-Varianten, da der Action-Space klein genug ist, um bessere Taktikten zu lernen (vor allem das Vermeiden von Sackgassen).
2. **Q-Learning:** Wird wahrscheinlich schlechter ausfallen als PPO. Q-Learning bekommt bei größeren Suchräumen schneller Probleme als PPO. (z.B. ist bei [11]SARSA doppelt so gut wie Q-Learning)
3. **PPO:** Laut [2] hat PPO bessere Ergebnisse erzielt als Q-Learning und SARSA. PPO hat in vielen Experimenten am besten abgeschnitten, auch bei Szenarien, die keine Spiele sind.
4. **SARSA:** Schneidet wahrscheinlich besser als Q-Learning ab, aber schlechter als PPO

# Komplexität und Modelbildung

Es folgt eine Berechnung bezüglich der Komplexität der Spielwelt.

## Komplexität der Snake-Spielwelt

Im Folgenden wird von einem quadratischen Spielfeld mit der Größe n\*n ausgegangen. Für die Experimente wurden die Feldgrößen 6x6, 12x12, 24x24 und 48x48 ausgewählt. Jede Zelle hat vier mögliche Zustände: {Apfel, Kopf der Schlange, Körper der Schlange oder leer}. Laut [11] lässt sich die Größe des State Spaces |S| durch einfaches Zählen berechnen: |S| > n8

Bei einer Spielfeldgröße von 20x20 ergibt sich hieraus aus ein State Space von der Größe 25.600.000.000. Laut [11] sind hier Trainingsläufe mit eine Millionen Spielzügen nötig. Somit ist ein vollständiges Lernen für diese Projektarbeit ungeeignet es muss eine Abbildung der Spielwelt auf einen kleineren State Space vorgenommen werden.

## Diskussion der Modellierung der Spaces

Im Folgenden folgt eine Diskussion bezüglich unterschiedlicher Aspekte, die für die Abbildung eines State-Spaces in Frage kommen.

### Symmetrie der Spielwelt

Da das Spielfeld symmetrisch ist, bieten sich mehrere Möglichkeiten an, den State Space zu vereinfachen. So kann beispielsweise jeder Spielstand um 90° rotiert werden, womit ein neuer gültiger Spielstand entsteht. Wird die ungefähre Richtung des Apfels mit „links, rechts, oben, unten“ unabhängig von der Richtung des Kopfes (oder 4 Himmelsrichtungen) codiert, so kann diese Symmetrie nicht berücksichtigt werden. In diesem Fall wäre das Rotieren der Spielwelt ein komplett neuer Zustand, der trainiert werden müsste. Es eignet sich somit eine Darstellung der Spielwelt aus Richtung des Kopfes, wodurch sich die Anzahl der möglichen States um ein Viertel verkleinert.

### Größere State Spaces

[11] zeigt, dass selbst ein State Space mit 128 möglichen States nach Trainingsläufen mit 2 Millionen Spielzügen das Spiel nicht löst (mehrere Stunden Trainingsdauer). Es ist fraglich, ob eine größere Anzahl an möglichen States eine Verbesserung mit einer machbaren Trainingsdauer zulässt.

Beispielsweise können die Richtungen von Äpfeln und Hindernissen anstatt binär mit mehreren „Reichweiten“ beschrieben werden. So könnte die Position eines Hindernisses vor der Schlange mit „nah“, „mittel“ und „weit“ dargestellt werden. Es ist fraglich, ob mit solchen weiteren States das Training in akzeptabler Trainingszeit konvergiert. Zudem muss dies nicht unbedingt eine Verbesserung des Verhaltens der Schlange führen.

### Größe der Spielwelt

Die meisten Paper und Implementierungen von Snake mit RL wählen eine Spielfeldgröße von 20x20. Eine größere Spielfeldgröße würde dafür sorgen, dass die Schlange länger braucht, um einen Apfel zu essen und somit positive Rewards weniger oft auftreten, wodurch der Agent nur sehr langsam oder ggf. gar nicht lernt.

### Modellierung des Rewards

Das Aufsammeln von Äpfeln immer belohnt, z.B. mit „1“ oder „30“ (siehe Laut [5] und [12]). Das Laufen in ein Hindernis wird meist Bestraft (gleicher Wert wie das Aufsammeln eines Apfels oder größere Bestrafung). Eine Bestrafung für das Laufen in ein Hindernis kann dem Agenten früh im Spiel helfen, Hindernisse zu vermeiden. Dies hilft wiederum dem Agenten die Spielwelt mehr zu erkunden, da eine Episode somit im Durchschnitt mehr Steps läuft und wiederum ein Apfel mit höher Wahrscheinlichkeit gefunden werden kann. Um das Lernen des Agenten zu beschleunigen, wird also eine Bestrafung für das Laufen in Hindernisse implementiert.

Manches Paper (z.B. [9]) haben auch die Distanz zum Apfel im Reward: wenn sich die Schlange dem Apfel nähert, bekommt sie einen kleinen Reward. Jedoch ist fraglich, ob dies zu einem besseren Verhalten des Agenten führt. Der Agent lernt zwar schneller sich dem Apfel zu nähern, jedoch ist es im späteren Spielverlauf (wenn die Schlange länger ist) nicht immer besser direkt zum Apfel zu gehen, um Sackgassen zu vermeiden. Eine Belohnung für das Nähern zum Apfel wird in diesem Projekt nicht weiter betrachtet (siehe regelbasierter Algorithmus in Kapitel 3.1.2).

## Modellierung des State und Action Spaces

In diesem Kapitel wird erläutert, wie die Modellierung von State, Action und Rewards für die Experimente implementiert wurden.

### Mögliche Modellierungen

Im Folgenden werden drei mögliche Modellierungen der State und Action Spaces von anderen Quellen kurz dargestellt:

**1. Relative Positionen von** [11]**:**

* + Relative Position der Schlange zum Apfel + Aktueller Zustand der Schlange
  + Ein Bild, das Text enthält.

    Automatisch generierte Beschreibung
  + w = gibt an, wo sich eine Wand direkt am Kopf befindet (Straight, Left, Right)
  + q = Relative Position des Apfels (Food) und Schwanzes (Tail)
  + State Space hat somit eine Größe von 23 \* 42 = 128

**2. DDQN von** [5]**:**

Als Input für ein NN werden hier Pixel der graphischen Oberfläche des Spiels genommen. Somit wird hier ein vollständiges Lernen angestrebt. Die Trainingsläufe sind jedoch sehr lang, was für die vorhandene Hardware in diesem Projekt nicht anwendbar ist.

**3. Binäre 11 Parameter von** [12]**:**

* Richtung von Gefahr am Kopf: vorne, links, rechts
* Move Direction (links, rechts, hoch, runter)
* Richtung des Futters (links, rechts, hoch, runter)

### Implementierung des State Spaces

Für die Experimente in dieser Projektarbeit wurde die letzte Modellierungsart (3.) genommen. Durch die 11 binären Parameter ergibt sich somit ein auch ein Input-Layer für die neuronalen Netzen von 11 Bit. Dies ist ein kleiner Observation-Space, wodurch der Agent lernen sollte, sich Richtung Apfel zu bewegen. Eine mögliche Strategie für den Agent ist hiermit, dass der Agent sich mit Kopf an einem Hindernis hält (Gefahr links oder rechts sollten ‚True‘ bleiben). So kann die Schlange am Rand der Welt bleiben, womit der Agent Sackgassen vermeidet.

Die Umwandlung von der Spielweltlogik in ein Observation-Space erfolgt in der Datei „snake\_env.py“ in der Methode get\_state(), Zeile 27.

Andere State-Abbildungen sind entweder zu einfach (1. Art) oder zu komplex (2. Art). Kleine State-Spaces sind einfach zu lösen und verhindert ein Lernen von komplexeren Strategien. Zu komplexe State-Spaces benötigen zu lange Trainingsläufe.

### Implementierung des Action Spaces und Reward

Das Snake spiel lässt nur vier Aktionen zu, die sich binär abbilden lassen. Da es kein Sinn macht, dass sich der Kopf der Schlange in den eigenen Körper bewegt, kann eine Richtung immer ignoriert werden. Ist beispielsweise der Schlangenkopf in Richtung Osten ausgerichtet, macht es kein Sinn, dass der Agent sich Richtung Westen bewegt.

Zudem lässt sich die Symmetrie des Spielfeldes nutzen: egal in welche Himmelsrichtung der Kopf gerichtet ist, kann der Agent nur drei Aktionen ausführen: „geradeaus“, „rechts“ oder „links“.

Als Action-Space ergibt sich somit ein binäres Array mit drei Werten. Logik für die Umwandlung von Agenten-Aktionen in Spiellogik findet in der Klasse „snake\_env.py“ in der Methode step() ab Zeile 75 statt. Die Kodierung für die Aktionen sieht wie folgt aus:

* [1,0,0] -> Straight
* [0,1,0] -> Right Turn
* [0,0,1] -> Left Turn

Der Agent fängt eine Episode mit einem Reward von 0 an. Findet die Schlange einen Apfel, wird der Reward um 1 erhöht. Läuft die Schlange in ein Hindernis wird die Episode beendet und der Reward wird um 1 reduziert. Durch die Bestrafung lernt der Agent Hindernisse zu vermeiden, selbst wenn kein Apfel gefunden wird. Durch das Vermeiden von Hindernissen wird wiederum die Chance erhöht, Äpfel zu finden.

# Implementierung

In diesem Kapitel werden die Dateien der Implementierung kurz erläutert. Es werden anhand eines Ablaufdiagramms und eines Klassendiagramms die Abhängigkeiten zwischen den Dateien veranschaulicht. Bei der Implementierung handelt es sich um eine Python 3.9 Umgebung. Training der Algorithmen erfolgt durch PyTorch. Trainieren der neuronalen Netze erfolgt durch CUDA.

## Trainings- und Testablauf

Abbildung 3 zeigt ein Ablaufdiagramm eines Trainingslaufes. Zunächst werden alle globalen Trainingsparameter in der Datei „parameters.py“ definiert. Diese Parameter werden von allen anderen Python-Skripten benutzt. Mit dem Ausführen der Datei „start\_training.py“ werden Trainings- und Testläufe gestartet. In dieser Datei befinden sich Logik für Schleifen von Episoden und Spielschritten, inklusive Logging.

Für jede Step wird die entsprechende Methode des jeweiligen Agenten (agent\_X.py) aufgerufen. Die Agenten rufen wiederum eine Step des Environments auf („snake\_env.py“ im Ordner „/snake“). Das Environment übernimmt die Umwandlung von States für Agenten zu Spielwelt. Die komplette Spiellogik befindet sich in der Datei „snake\_game\_ai.py“. Durch das Environment kann eine einfache Änderung des Observation- und State-Spaces vorgenommen werden, ohne die eigentliche Spiellogik zu ändern. Zudem findet somit auch eine klare Trennung zwischen der Sichtbarkeiten der Agenten und der internen Spiellogik statt.

Eine Episode wird unter zwei möglichen Bedingungen abgebrochen: entweder läuft der Agent in ein Hindernis, oder der Agent hat für 100 Steps keinen Apfel gefunden (einstellbar über den Parameter „early\_episode\_end\_steps“).

Wird die maximale Anzahl an Episoden für einen Trainingslauf erreicht, wird der Lauf beendet. Das trainierte neuronale Netz wird abgespeichert und es findet eine Visualisierung, sowie ein Logging, der Trainingsdaten statt.

Nach erfolgreichem Training kann ein Testlauf gestartet werden. Siehe README.md für weitere Informationen zum Starten von Trainings- und Testläufen.

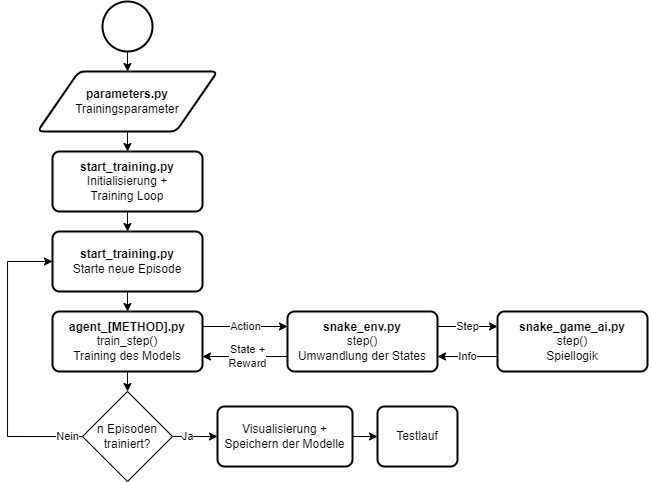


Abbildung 2 Ablaufdiagramm

## Klassen, Dateien und Ordner

Abbildung 2 zeigt ein Klassendiagramm der wichtigsten Klassen der Implementierung.

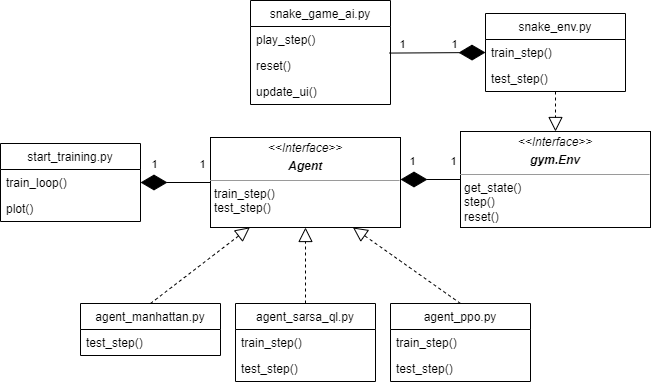


Abbildung 3 Klassendiagramm

Es folgt eine Auflistung aller Dateien und Ordner in alphabetischer Reihenfolge:

* **agent\_euclidean.py:** Implementierungen des regelbasierten Algorithmus mit euklidischem Abstand. Dieser Agent hat keine train\_step() Methode, da dieser Agent nicht trainiert werden kann. Siehe Kapitel 6.3.1 für weitere Informationen.
* **agent\_ppo.py:** Implementierung des PPO-Agenten. Siehe Kapitel 6.3.3 für weitere Informationen.
* **agent\_sarsa\_ql.py:** Beinhaltet die Implementierung der Agenten für SARSA und DNQ. Siehe Kapitel 6.3.2 für weitere Informationen.
* **/diagramme:** beinhaltet Diagramme für diesen Projektbericht.
* **Experimentenkatalog.xlsx:** Ergebnisse der Experimente
* **/logs:** Ordner für Logdateien der Trainingsläufe
* **parameters.py**: beinhaltet alle globalen Parameter für Trainingsläufe. Die meisten anderen Python-Skripte greifen hierdrauf zu.
* **README.md**: Readme-Datei für das Projekt. Beinhaltet eine Anleitung für die Installation und das Starten von Trainingsläufen
* **/saved\_models\_X:** beinhaltet die trainierten neuronalen Netze. Die NN werden nach einem Trainingslauf gespeichert.
* **start\_training.py:** Beginnt die Trainingsschleife und hat Logik für den Beginn von Episoden. Vor dem Training werden alle benötigten Initialisierungen vorgenommen. In der Trainingsschleife werden die Methoden train\_step() und test\_step() aufgerufen, womit die Agenten jeweils eine Step ausführen.
* **/snake:** beinhaltet das Snake-Environment basierend auf Gym von OpenAI. Implementiert das Interface „gym.Env“. Die Unterordner sind von OpenAI vorgeben. Das Environment wird als Python-Package gehandhabt, weswegen eine setup.py benötigt wird. Siehe <https://www.gymlibrary.ml/content/environment_creation/> für weitere Details.
* **snake\_game\_ai.py:** beinhaltet die Logik für das Snake-Spiel, mit der Agenten über das Environment interagieren können.
* **snake\_game\_manual.py:** UnabhängigesPython-Skript für das manuelle Spielen von Menschen. Der Code wurde von <https://github.com/vedantgoswami/SnakeGameAI> übernommen.

Logik von dem Python Spiel (snake\_game\_ai.py und snake\_game\_manual.py) wurden von <https://github.com/vedantgoswami/SnakeGameAI> übernommen.

## Implementierungen der Algorithmen

### Implementierung Regelbasierter Algorithmus (Euklidischer Abstand)

Um Referenzwerte für die Reinforcement-Learning-Ansätze zu erhalten, wird zunächst ein regelbasierter Algorithmus implementiert.

Der Algorithmus steuert die Schlange für jeden Spielzug Richtung Apfel abhängig von der kürzesten Distanz (Euklidischer Abstand). Zusätzlich wurde der Algorithmus so implementiert, dass er nicht in Hindernisse steuern kann. Ist ein Hindernis im Weg der kürzesten Strecke, wird versucht, die nächste bessere Richtung zum Apfel zu nehmen.

Der Algorithmus sollte somit immer einen Weg zum Apfel finden, es sei denn, dass sich der Kopf in einer Sackgasse befindet.

Der Algorithmus arbeitet mit den gleichen Observation-Space wie die RL-Algorithmen (siehe Abschnitt 5.2).

### Implementierung DQN und SARSA

Die Logik für die Agenten für DQN und SARSA befindet sich in der Datei „agent\_sarsa\_ql.py“

Die Implementierung basiert auf <https://github.com/rogerlucena/snake-ai> und wurde an das Snake-Environment angepasst. Der Agent implementiert das Interface „torch.nn.Module“ von OpenAI für die Verwendung des Gym-Environments. DQN und SARSA unterscheiden sich nur in der Auswahl der nächsten Aktion, weswegen die beiden Agenten sich viel Code teilen können. Eine Unterscheidung der beiden Agenten findet nur in Zeile 97 statt.

### Implementierung PPO

Die Implementierung basiert auf der Stable Baseline von OpenAi (siehe <https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>).  
Der Agent implementiert das Interface „torch.nn.Module“ von OpenAI für die Verwendung des Gym-Environments.

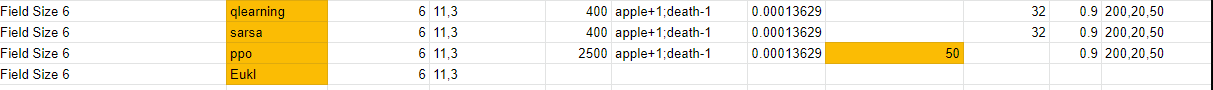
# Aufbau der Versuche

Zu Beginn der Experimente wurde untersucht, welche Hyperparameter sich für die jeweiligen Agenten eignen. Hierfür wurden andere Snake RL-Repositories und generelle Empfehlungen für Hyperparameter betrachtet. Siehe:

* <https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe>
* <https://github.com/llSourcell/Unity_ML_Agents/blob/master/docs/best-practices-ppo.md>

Alle Experimente, bei den der Agent gelernt hat, wurden in einer Excel-Tabelle („Experimentenkatalog“) dokumentiert. Folgende Bilder zeigen einen Teil der Ergebnisse für eine Spielfeldgröße von 6:

**Eingestellte Parameter:**



Dier hier dargestellten Parameter wurden für die Experimente mit den weiteren Szenarien übernommen. Diese Parameter wurden anhand etlicher Experimente festgelegt (siehe Expermentenkatalog im Code-Repository).

**Ergebnisse der Testläufe:**

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

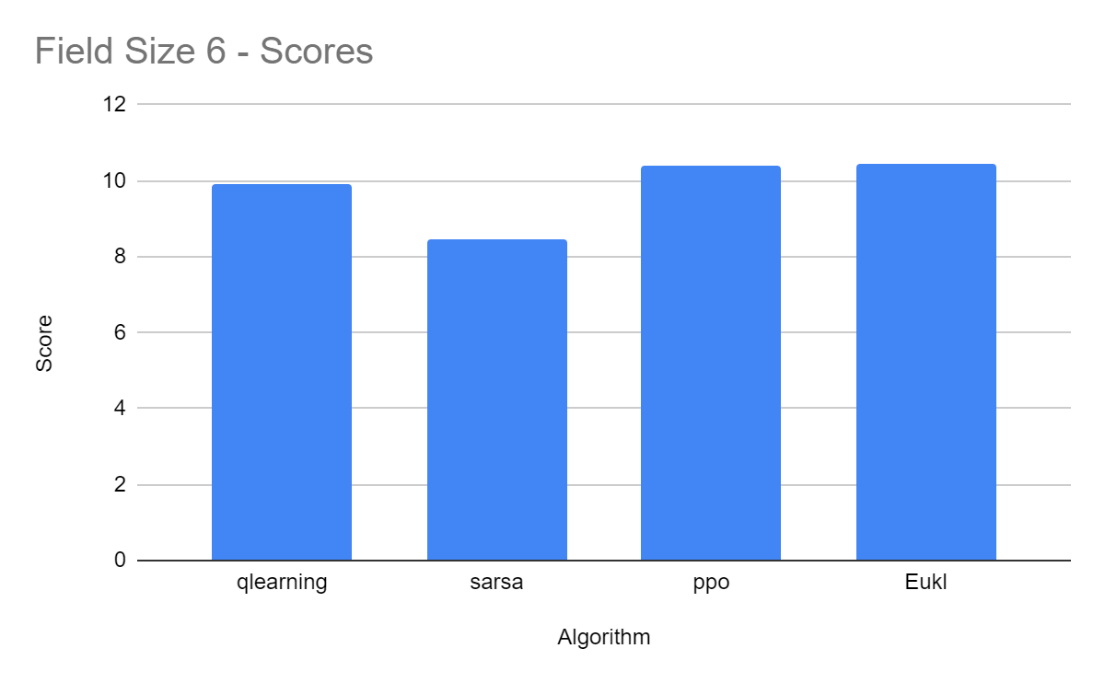
Viele der Hyperparameter von den Algorithmen überlappen sich. Ausnahmen von den betrachtenden Hyperparametern hierbei sind „update\_timestep“ von PPO (wie oft PPO seine Policy aktualisiert) und die „batch\_size“ von DQN/SARSA (mit wie vielen State-Action-Paaren die Policy aktualisiert wird). Details zu den einzelnen Hyperparametern sind aus der Datei „parameters.py“ zu entnehmen.

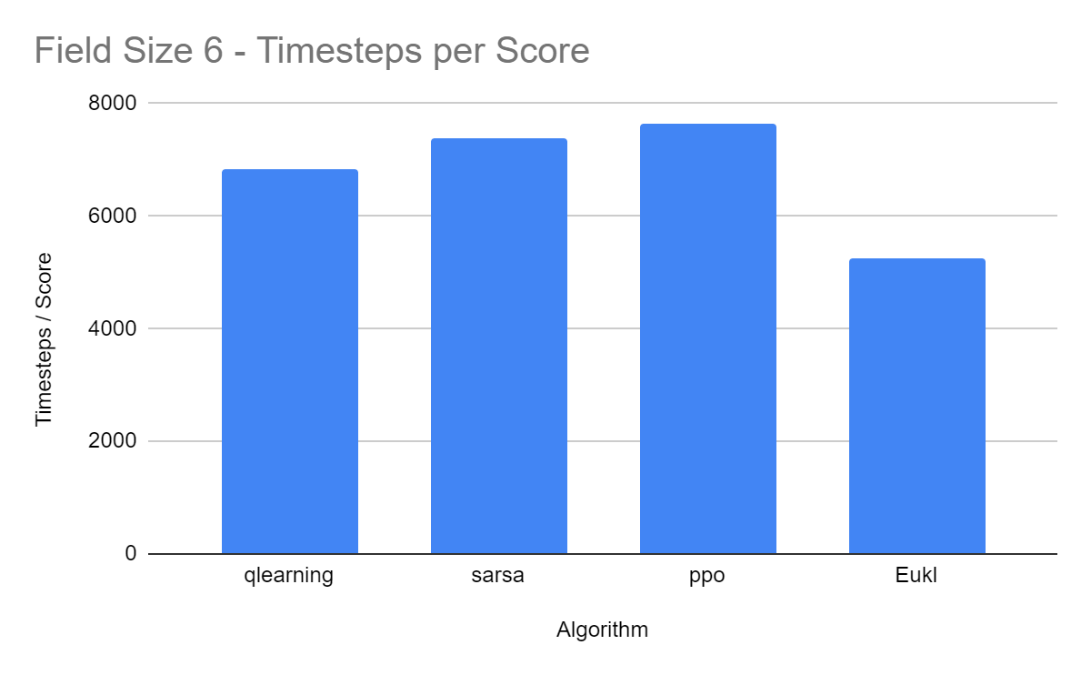
Bei den Experimenten wurde darauf geachtet, dass die Agenten unter möglichst ähnlichen Bedingungen verglichen werden. Problematisch war hierbei die Trainingsdauer. DQN und SARSA haben eine ca. fünf Mal höhere Trainingsdauer pro Episode als PPO. Um dies Auszugleichen, wurde die Episodenanzahl angepasst, damit die drei RL-Algorithmen eine ungefähre gleiche Trainingszeit haben.

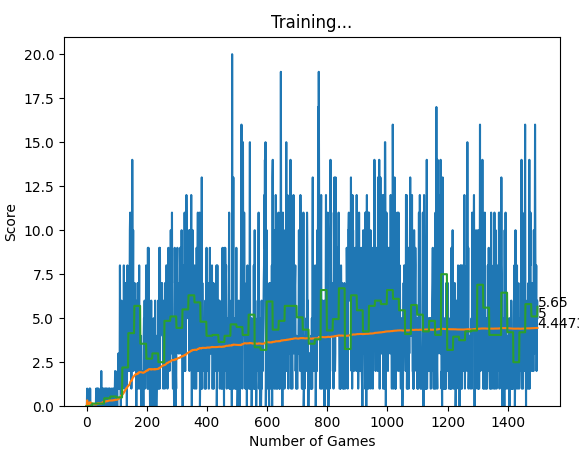
# Zusammenfassung der Versuche

Folgende Bilder fassen die Ergebnisse der Experimente zusammen. DQN und SARSA wurden jeweils mit 400 Episoden trainiert, während PPO mit 2500 trainiert wurden. Eine Analyse der Ergebnisse folgt anschließend in Abschnitt 8.5.

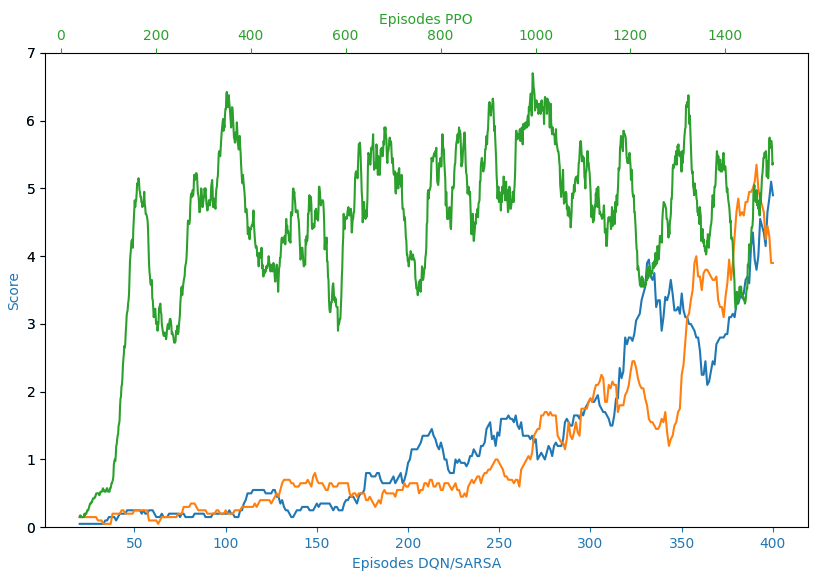
## Spielweltgröße 6



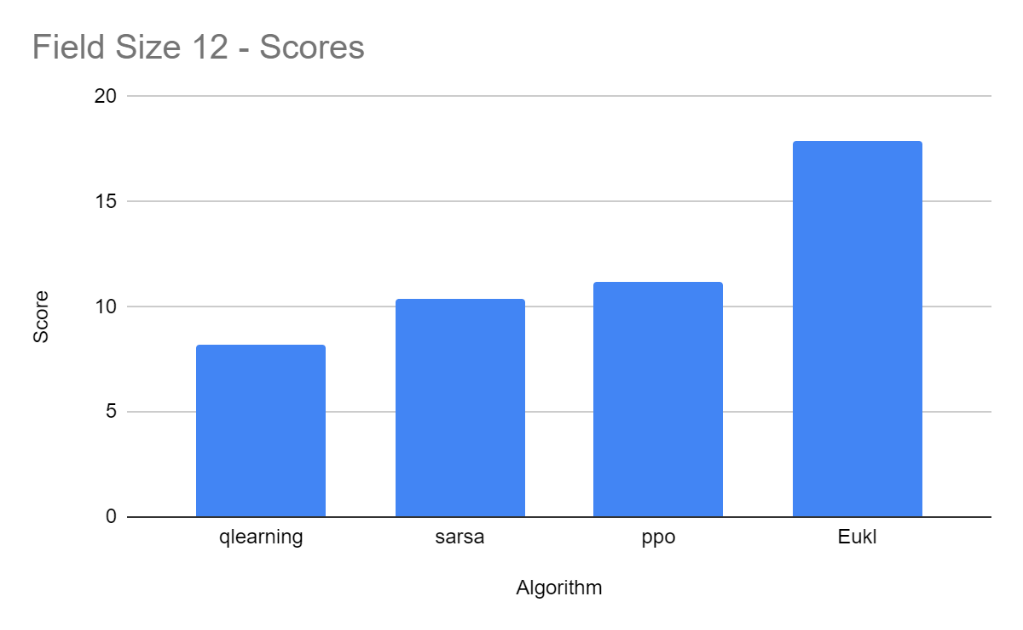


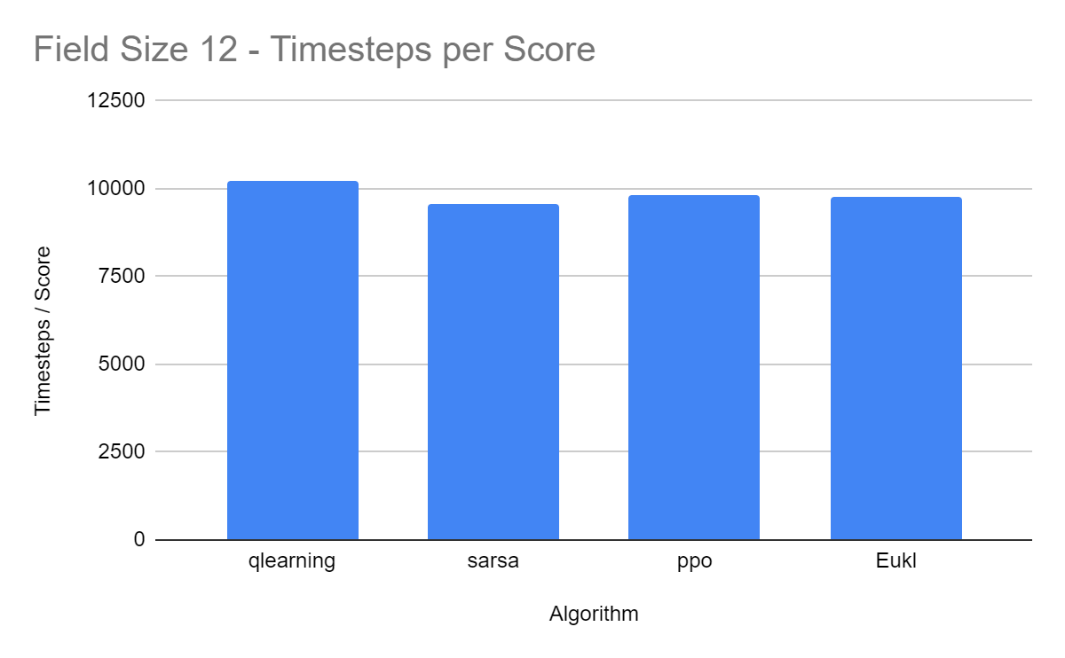
**Beispiel Auszug für das Trainieren von PPO bei einer Spielfeldgröße von 6:**   
- Orange: Average Score   
- Grün: Moving Average von 20   


**ROC Kurve – Trainingslauf Spielweltgröße 6**   
- Blau: DQN (400 Episoden, 540 Sekunden)   
- Orange: SARSA (400 Episoden, 512 Sekunden)   
- Grün: PPO (1400 Episoden, 766 Sekunden)

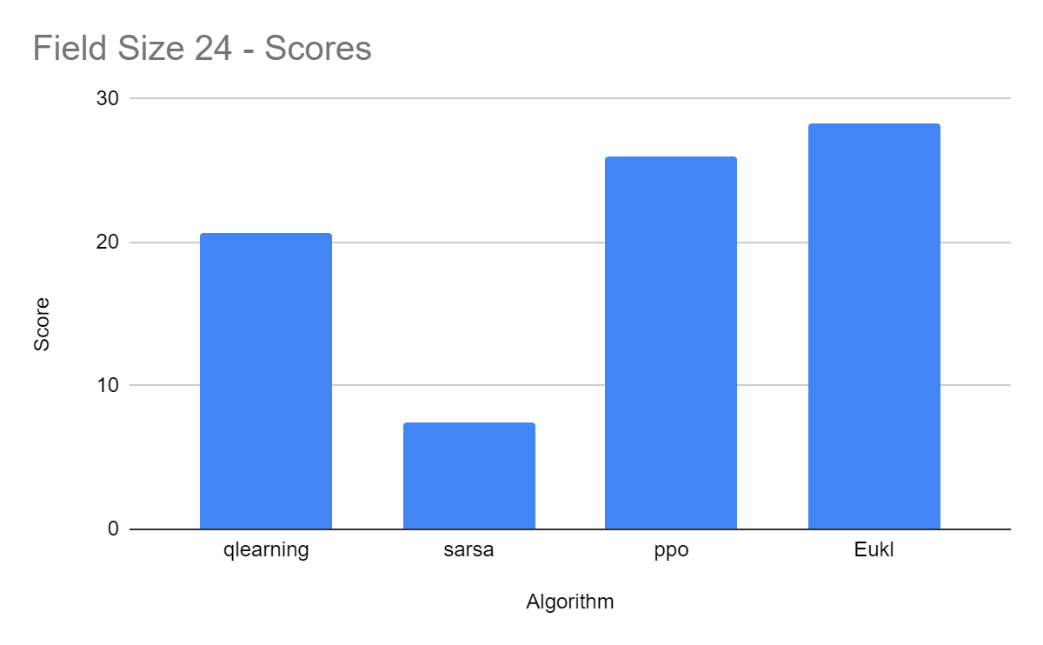


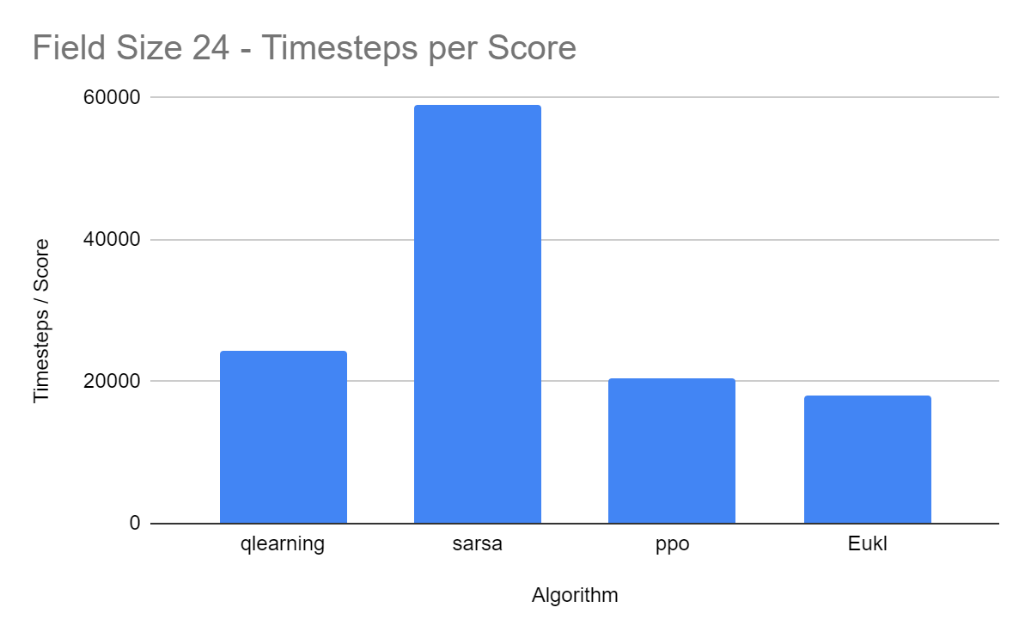
## Spielweltgröße 12





## Spielweltgröße 24





## Spielweltgröße 48

Kein Lernen: Alle Experimente hatten einen durchnittlichen Score von <1, da Äpfel zu selten gefunden werden. Um ein Training in diesem großen Spielfeld zu ermöglichen, gibt es mehrere Optionen:

* Trainingsläufe mit mehr Episoden: Dadurch werden zwar am Anfang nicht öfters Äpfel gefunden, aber nach genug Trainingsläufen müsste der Agent erkennen, dass er sich Richtung Äpfel bewegen muss.
* Pro Episode mehr als 100 Steps erlauben: Derzeit wird eine Episode beendet, wenn für 100 Steps kein Apfel gefunden wird. Wird dieses Limit erhöht, findet der Agent (mit Glück) im Durchschnitt mehr Äpfel.
* Reward anpassen: Man kann den Reward so anpassen, dass der Agent belohnt wird, wenn er sich dem Apfel nähert. Dies ist jedoch problematisch, da sich das Nähern des Apfels nicht immer die beste Aktion sein muss (z.B. beim Vermeiden von Sackgassen).

## Erkenntnisse

Im Folgenden erfolgt eine Analyse der Ergebnisse der Experimente.

### Betrachtung der Trainingsläufe

Auffällig ist, dass keiner der Agenten konvergiert. Keine der Trainingsläufe lief lange genug, dass eine Konvergenz erkannt wurde. Dies ist Zeichen dafür, dass die Trainingsläufe zu kurz waren. Alle Trainingsläufe haben weniger als eine Stunde trainiert. Im Rahmen des Projekts war es leider nicht möglich, längere Trainingsläufe durchzuführen.

DQN und SARSA trainieren pro Episode ca. fünf Mal so lange wie PPO. Dies liegt daran, dass die Policies von DQN und SARSA mit jedem Step aktualisiert werden, während dies bei PPO nur alle 50 Steps geschieht. Das Training wurde im Rahmen dieser Projektarbeit mit einer RTX 3070 Ti durchgeführt, aber aufgrund der hohen Häufigkeit von Tensor-Berechnungen, ist das Training immer noch langsam verlaufen.

Je größer die Spielwelt, desto langsamer lernt der Agent. Da die Spielwelt größer wird und die RL-Agenten sich anfangs zufällig verhalten, werden nur selten Äpfel gefunden.

### Betrachtung der Scores

Folgende Tabelle fasst die Ergebnisse der Scores zusammen:

|  |  |  |  |
| --- | --- | --- | --- |
| **Agent** | **6x6** | **12x12** | **24x24** |
| DQN | 9.923 | 8.213 | 20.673 |
| SARSA | 8.4547 | 10.376 | 7.401 |
| PPO | 10.207 | 11.226 | 26.0287 |
| Regelbasiert | 10.4274 | 17.875 | 28.27 |

Bei Trainingsläufen mit einer Spielfeldgröße von 48x48 lernen die RL-Algorithmen nicht (Score von <1).

Die Algorithmen haben wie folgt abgeschnitten:

* Der Regelbasierte Algorithmus schneidet immer am besten ab. Daraus wird deutlich, dass die RL-Agenten auch immer nur versuchen, sich den Apfel zu nähern. Komplexere Strategien werden von den RL-Algorithmen gar nicht erst versucht zu trainieren, da sich ein Nähern des Apfels (ohne in Hindernisse zu laufen) schon schwierig genug für die gewählten Trainingszeiten ist.
* Der PPO-Agent belegt konsistent den zweiten Platz. Er kommt sehr nah an die Scores des Regelbasierten Agenten. Überraschend ist hierbei, dass PPO selbst bei einer Spielfeldgröße von 24x24 fast mit dem regelbasierten mithalten kann.
* DQN ist bei den Szenarien 6x6 und 24x24 besser als SARSA. Auffällig ist, dass SARSA bei der Spielfeldgröße von 24x24 deutlich schlechter abschneidet. Ein Grund hierfür könnte die on-policy Eigenschaft von SARSA sein. Im Vergleich versucht DQN als off-policy Methode Kollisionen eher zu vermeiden, wodurch Trainingsläufe im Durchschnitt länger laufen und somit wiederum Chance erhöht Äpfel zu finden. SARSA als off-policy Algorithmus versucht „zufälliger“ sich heranzutasten, wodurch öfters in Hindernisse gelaufen wird.

Weitere Erkenntnisse:

* Keine der Agenten kommt nah an die optimale Lösung ran (Spielfeldgröße n²). Bei dem regelbasierten Algorithmus ist das zu erwarten, da er keine Sackgassen erkennen kann.
* Größere Feldgrößen führen zu besseren Scores. Durch die größere Spielwelt nimmt zum einen der Körper der Schlange einen relativen kleineren Anteil der Spielwelt ein und zum anderen sind Wände der Welt weiter vom Startpunkt der Schlange entfernt. Hindernisse sind somit seltener anzutreffen

### Betrachtung der Timesteps

Eine Analyse der Timesteps gestaltet sich als schwierig. Über die Timesteps kann nicht festgestellt werden, ob ein Agent besser agiert oder nicht. Dies liegt zum einen anderen, dass der kürzeste Weg zum Apfel nicht die optimale Strategie sein muss (siehe Kapitel 3.1.2), zum anderen erhöht sich die durchschnittliche Timestep pro Score im späteren Spielverlauf, da die Schlange größer wird. Ein Agent mit einem höheren Score erreicht somit auch eine höhere Timestep pro Score. Wiederum kann sich auch die Timestep pro Score erhöhen, wenn der Agent Hindernisse vermeidet, aber keine Äpfel findet (siehe SARSA bei 24x24).

Eine Betrachtung von Timesteps zur Analyse der Rationalität eines Agenten ist somit nicht geeignet.

# Fehlerbetrachtung

Es folgt eine Analyse über mögliche Fehler, die in dieser Projektarbeit durchgeführt wurden.

**Trainingszeiten**Für eine aussagekräftigere Beurteilung der Algorithmen müssen längere Trainingsläufe durchgeführt werden. Bei Szenarien in mit großen Spaces werden oft Trainingsläufe mit 100.000 oder 1.000.000 Episoden durchgeführt, die oft für Stunden oder Tage trainieren. Die größten Trainingslauf im Rahmen dieser Projektarbeit hatte 10.000 Episoden mit einer Dauer von ca. 3411 Sekunden.

**Implementationen**Für die Implementierungen der Algorithmen, sowie der Snake-Welt wurden vorhandene GitHub-Repositories herangezogen. Es ist fraglich, ob sich in diesen Repositories nicht möglicherweise Fehler befinden. Die Komplexität des Codes ist bei RL-Algorithmen meist sehr hoch, wodurch sich eine Überprüfung der Richtigkeit als schwierig erweist.

**Auswahl der Hyperparameter und deren Parametrisierung**Die Hyperparameter sind manchmal schwierig zu vergleichen, aufgrund der Unterschiedlichen RL-Algorithmen. Beispielsweise wurde für den Parameter „ppo\_update\_timestep“ ein Wert von 50 für die Szenarien festgelegt, aber es ist fraglich, ob ein anderer Wert vielleicht besser abschneidet. Für jeden Parameter müssten nochmal mehrere Trainingsläufe vorgenommen werden, um bessere Hyperparameter für Experimente zu definieren. Die ausgewählten Hyperparameter-Werte für die durchgeführten Experimente basieren auf Erfahrungen von Anderen in anderen Environments und einer geringen Auswahl von selbstdurchgeführten Trainingsläufen.

**Zufälligkeit der Snake-Welt**Durch die Zufälligkeit des Apfels können Trainingsverläufe stark unterschiedlich ablaufen, trotz gleicher Rahmenbedingungen. Da in großen Spielwelten nur selten Äpfel gefunden werden, kann ein seltenes finden von Äpfeln im frühen Trainingsverlauf zu deutlich schlechten Ergebnissen führen im Vergleich zu einem Trainingsverlauf, der viele Äpfel findet. Um die Agenten fairer zu vergleichen, müssten die Trainingsszenarien öfters unter gleichen Bedingungen wiederholt werden.

# Aussicht

Dieses Projekt dient als gute Grundlage für den Vergleich von RL-Algorithmen in einer Spielwelt mit Zufall. Aufbauend auf dieser Arbeit können folgende Erweiterungen vorgenommen werden:

* **Längere Experimente:** wie bereits besprochen, würden längere Trainingsläufe zu aussagekräftigeren Ergebnissen führen.So könnte Agent eine Taktik lernen, um Sackgassen zu vermeiden. Durch längere Trainingsläufe sollten die RL-Algorithmen in der Lage sein, den regelbasierten Algorithmus zu schlagen
* **Anpassung von Observation-Space und Reward:** Im Rahmen dieser Projektarbeit konnte nur eine Art von Observation-Space (11 Bit) untersucht werden. Interessant wären hier andere Darstellungen der States, wie beispielweise durch das Ergänzen der Distanz von Hindernissen und Äpfel. Zudem können andere Rewardfunktionen getestet werden. In Kombinationen mit längeren Trainingsläufen und größeren neuronalen Netzen könnte auch ein vollständiges Lernen möglich sein.
* **Auswahl anderer Algorithmen:** Es gibt moderne Varianten von DQN, SARSA und PPO, die implementiert werden können, wie beispielsweise Double Q-Learning, Expected SARSA oder Parallel PPO. Zudem können auch komplett andere Arten von RL-Algorithmen untersucht werden, wie beispielsweise Asynchronous Advantage Actor Critic (A3C).
* **Effizienter Trainingsumgebung**: Python ist eine langsame Programmiersprache, wodurch sich ein Austausch der Snake-Spiellogik durch eine schnellere Sprache (z.B. C++ oder Rust) für schnellere Trainingsläufe lohnen würden. Auch könnte der Agent für DQN und SARSA optimiert werden.
* **Verwendung von Experimentmanager:** Das händische Loggen in einer Excel-Tabelle ist aufwändig. Durch einen Experimentmanager (z.B. Weights & Biases) können Trainingsläufe durch Automatisierung schneller durchgeführt werden. Zudem kann somit auch eine bessere Visualisierung und Analyse der Trainings- und Testläufe erfolgen.

# Zusammenfassung

In dieser Projektarbeit wurden die RL-Algorithmen DQN, SARSA und PPO in Kontext des Videospiels Snake miteinander verglichen. Es wurden die Unterschiede der RL-Algorithmen dargestellt. Es wurde diskutiert, wie Observation-, Action- und Reward-States für die Snake-Umgebung umgesetzt werden können.

Es wurde eine Trainingsumgebung entwickelt, mit der effizient die unterschiedlichen Agenten trainiert und getestet werden können. Als Referenz wurde ein regelbasierter Algorithmus auf Basis des euklidischen Abstands implementiert. Für die Experimente wurden die ausgewählten Observation- und Action-Spaces implementiert.

Die Ergebnisse der Experimente entsprechen den zu erwarteten Ergebnissen. Die drei RL-Methoden haben aufgrund der geringen Trainingszeit schlechter abgeschnitten als der einfache regelbasierte Algorithmus. PPO hat von den RL-Algorithmen am besten abgeschnitten. In kleinen Spielwelten war SARSA besser als DQN, jedoch hat SARSA schlecht mit größeren Spielwelten skaliert.

# Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | O. Delalleau, M. Peter, E. Alonso und A. Logut, „arxiv.org,“ Ubisoft La Forge, 23 Dezember 2019. [Online]. Available: https://arxiv.org/pdf/1912.11077.pdf. [Zugriff am 26 April 2022]. |
| [2] | R. S. Bonnici und e. al, „Exploring Reinforcement Learning: A Case Study Applied to the Popular Snake Game,“ 1 Januar 2022. [Online]. |
| [3] | „Automated Snake Game Solvers via AI Search Algorithms,“ [Online]. Available: https://cpb-us-e2.wpmucdn.com/sites.uci.edu/dist/5/1894/files/2016/12/AutomatedSnakeGameSolvers.pdf. |
| [4] | „Snake-AI-GameSolver,“ [Online]. Available: https://github.com/neelgajjar/Snake-game-AI-Solver. |
| [5] | A. Finnson und V. Molno, „Deep Reinforcement Learning for Snake,“ 2019. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1342302/FULLTEXT01.pdf. |
| [6] | O. H. Gyeol, 08 Juli 2017. [Online]. Available: https://han-gyeol.github.io/fun/2017/07/08/Game-AI-Snake-Game/. [Zugriff am 26 April 2022]. |
| [7] | „Proximal Policy - OpenAI,“ 20 Juli 2017. [Online]. Available: https://openai.com/blog/openai-baselines-ppo/. [Zugriff am 26 Juli 2022]. |
| [8] | M. Muller-Brockhausen, A. Plaat und M. Preuss, „Reliable validation of Reinforcement Learning Benchmarks,“ Leiden Unversity, 2 März 2022. [Online]. Available: https://arxiv.org/pdf/2203.01075.pdf. |
| [9] | „Mnih, Vlad; et al,“ Deepmind, 1 Januar 2013. [Online]. Available: https://www.deepmind.com/publications/playing-atari-with-deep-reinforcement-learning. |
| [10] | W. Urooj, „edureka,“ 26 März 2022. [Online]. Available: https://www.edureka.co/blog/snake-game-with-pygame/. [Zugriff am 26 April 2022]. |
| [11] | B. Ma, T. Meng und J. Zhang, „Exploration of Reinforcement Learning to SNAKE,“ 2016. [Online]. Available: http://cs229.stanford.edu/proj2016spr/report/060.pdf. |
| [12] | „GeeksForGeeks Blogpost,“ 1 Februar 2022. [Online]. Available: https://www.geeksforgeeks.org/ai-driven-snake-game-using-deep-q-learning/. |

1. <https://towardsdatascience.com/training-a-snake-game-ai-a-literature-review-1cdddcd1862f> [↑](#footnote-ref-1)