ShipzAI

Eine künstliche Intelligenz für das Spiel Schiffe versenken

Philipp Doll
Medieninformatik
Hochschule Düsseldorf
Essen, NRW, Deutschland
philipp.doll@study.hs-duesseldorf.de

Max van Aerssen
Medieninformatik
Hochschule Düsseldorf
Essen, NRW, Deutschland
max.vanaerssen@study.hs-duesseldorf.de

Christian Leich
Medieninformatik
Hochschule Düsseldorf
Düsseldorf, NRW, Deutschland
christian.leich@study.hs-duesseldorf.de

ABSTRACT

This project implements a reinforcement learning for the game battleships. Battleships is a classic board game where the goal is to find and destroy the enemy ships, quicker than your opponent. To train an agent this project provides a configurable battlehips OpenAI-Gym environment. The boardsize, number of ships and placement rules can be configured inside the environment. For this project a DQN- and ACKTR-agent where implemented, so the performance can be measured and compared.

CCS CONCEPTS

• Computing methodologies → Machine learning → Machine learning approaches → Neural networks

KEYWORDS

Machine learning, DQN, ACKTR, Reinforcement learning, Markov-Decision-Process, OpenAI Gym, Tensorboard, stable-baselines

ACM Reference format:

Philipp Doll, Max van Aerssen and Christian Leich. 2020. Insert ShipzAI: Eine künstliche Intelligenz für das Spiel Schiffe versenken

1 Einleitung

Das Spiel Schiffe versenken existiert in seiner Grundform wohl schon seit dem früher 20. Jahrhundert. Es besitzt einen simplen Aufbau und Spielablauf, sodass es gerne von Leuten auch nur mit Stift und Papier gespielt wird. 1977 wurde von Milton Bradley das Spiel Electronic Battleship auf den Markt gebracht. Dieses Spiel war eine erste Computerversion von Battleships, jedoch konnte diese nur anzeigen, ob es ein Treffer oder nicht war [1]. Durch den simplen Aufbau ist Schiffe versenken eins der ersten Spiele, die auf einem Computer gespielt werden konnten. Damit der Computer gegen einen Menschen spielen konnte, mussten Algorithmen entwickelt werden, womit der Computer den nächsten Zug berechnen konnte. Diese sind jedoch nur Übernahmen von menschlichen Strategien. Ein anderer Ansatz ist das Reinforcement Learning. Das Reinforcement learning gehört zu dem Bereich des Machine Learnings. Hierbei wird dem Computer nicht anhand eines Algorithmus vorgegeben, welcher Zug dieser als nächstes durchführen soll. Der Computer kann selbst Methoden und Strategien entwickeln, um das Spiel bestmöglich zu beschreiten. Durch diesen Ansatz ist es möglich, dass der Computer Strategien entwickelt, die effektiver sind als die eines menschlichen Spielers [2].

2 Motivation

Reinforcement Learning ist eine bewährte Methode, um einem Computersystem beizubringen, ein Spiel zu spielen und in diesem Spiel eigene Strategien zu entwickeln. Diese neu entwickelten Strategien erlauben es, in vielen Fällen, dass das Computersystem einem Menschen in dem Spiel überlegen ist. Ein Agent der jeden Zug zufällig auswählt, braucht bei einem 10x10 Spielfeld mit der klassischen Anzahl von Schiffen im Durchschnitt ~96 Schüsse. Durch einen Hunt Algorithmus kann die Anzahl der Schüsse auf ~65 Schüsse reduziert werden [3]. Ein durch Reinforcement Learning trainierter Agent kann in der Lage sein, ein Spiel in ~52 zu beenden [4]. Damit ist dieser besser als ein menschlicher Spieler.

3 Ziel

Es sollen Agenten trainiert werden, welche in der Lage sind, eine Partie Schiffe versenken möglichst effizient zu spielen. Diese Agenten werden dafür durch verschiedene Reinforcement Learning Algorithmen trainiert. Ziel jedes Agenten soll es sein, am Ende möglichst wenig Züge für ein Spiel zu benötigen. Dabei muss der jeweilige Agent in der Lage sein, sich auf unterschiedliche Anordnung der Schiffe, größe des Spielfelds und größe der Schiffe anpassen zu können. Die jeweiligen Agenten sollen verglichen werden. Dabei soll herausgefunden werden, welcher der Agenten letztendlich am effizientesten ist.

4 Task Environment

Die Umgebung wurde so programmiert, dass das Spiel über die Kommandozeile gespielt werden kann. Mit Hilfe einer Konfiguration lässt sich die Umgebung für verschiedene Zustände anpassen. Beispiele hierfür sind die Boardgröße, die Anzahl der Schiffe oder ob die Schiffe mit einer Lücke platziert werden. Für die Umgebung lassen sich folgende Eigenschaften festhalten:

Partially Observable	Der Agent kennt die Felder, die er beschossen hat. Die Positionen der gegnerischen Schiffe sind zu Beginn der Partie nicht bekannt.
Deterministic	Der nächste Zustand der Umgebung ist allein von dem aktuellen Zustand und der ausgewählten Aktion des Agenten abhängig.
Sequential	Das Ergebnis einer Aktion kann alle nachfolgenden Aktionen beeinflussen, je nachdem ob ein Schiff getroffen, beziehungsweise versenkt worden ist oder nicht.
Static	Das Spielfeld und die Position der Schiffe ändert sich nicht im Laufe eines Spiels, beziehungsweise einer Episode.
Discrete	Schiffe versenken ist ein rundenbasiertes Spiel und unterliegt somit keiner kontinuierlichen Veränderung.

Table 1: Eigenschaften der Umgebung

Die einzige Aktion, die der Agent innerhalb der Umgebung benötigt, ist der Beschuss eines Feldes. Dem Agenten stehen Percepts zur Verfügung. Sie werden über den Zustand jedes einzelnen Feldes zur Verfügung gestellt. Ein Feld kann folgende Informationen enthalten:

Wasser (encoding: W)	Auf diesem Feld befindet sich Wasser (nicht beschossen).
Treffer (encoding: X)	Ein Schiff wurde auf diesem Feld getroffen.
Gesunken (encoding: #)	Ein Schiff wurde auf diesem Feld versenkt.
Kein Treffer (encoding: 0)	Auf dieses Feld wurde geschossen und es wurde kein Schiff getroffen oder ist versunken.

Table 2: Mögliche Zustände eines Feldes

Die Zustände *Treffer*, *Gesunken* und *Kein Treffer* werden in der Kommandozeile angezeigt, wenn der Agent trainiert wird. Wenn ein Mensch spielt, wird zusätzlich der Zustand *Wasser* dargestellt.

Das Ziel eines Agenten soll sein, alle gegnerischen Schiffe zu versenken und hierfür so wenig Züge wie möglich zu benötigen.

3 Methode

Das Hauptkonzept von Reinforcement Learning wird in der Abbildung 1 dargestellt.

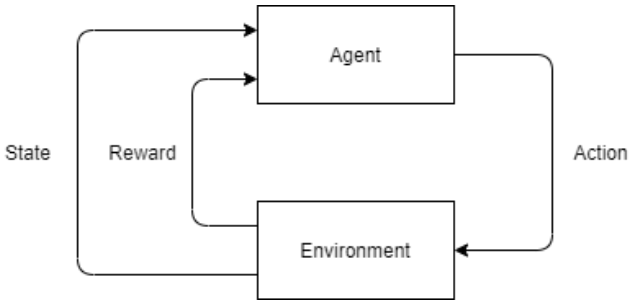

Figure 1: Reinforcement Learning Prozess

Ein Agent nimmt durch eine Action Einfluss auf die Umgebung. Im Kontext von Schiffe versenken bedeutet dies, dass ein Feld ausgewählt wird, welches beschossen wird. Die Umgebung gibt dem Agenten dann auf zwei Wegen Feedback. Zum einen gibt die Umgebung Feedback zu dem neuen State der Umgebung also z. B. Treffer, versenkt oder nicht getroffen. Zum anderen gibt die Umgebung dem Agenten einen Reward. Über diesen Reward bekommt der Agent mitgeteilt, wie gut oder schlecht seine ausgewählte Action war. Ziel des Agent ist es einen, möglichst hohen Reward zu bekommen, sprich möglichst effektiv das Spiel zu gewinnen. Das beschriebene Prinzip wird auch Markov-Decision-Process genannt. Der Markov-Decision-Process nimmt an, dass der aktuelle State einer Umgebung repräsentativ für alle States ist, die vorher in der Umgebung existiert haben.

- S: Alle möglichen States
- A: Alle möglichen Actions
- R: Reward Verteilung bei aktuellem State und ausgeführter Action (s, a)
- P: Transition probability. Die Wahrscheinlichkeit welchen State die Umgebung, nach einer ausgeführten Action auf dem aktuellen State, besitzt.
- ɣ: Discount factor. Hyperparameter, welcher von außen gesetzt werden kann. Beschreibt wie wichtig zukünftige rewards für den aktuellen State sind.

Ziel ist es eine optimale Policy π^* zu finden, die angibt welche Aktion bei einem aktuellen State auszuführen ist.

$$\pi^* = \max\left(\sum_{t<0} \gamma^t r^t\right)$$

Mathematisch definiert ist diese Policy π^* durch das Maximum aus der Summe aller Rewards. Dabei wird der discount factor benutzt, um zukünftige Rewards zu gewichten [10].

3.1 Deep Q-Learning Network (DQN)

DQN steht für Deep Q-Learning Network. Es ist eine Methode die im Bereich des Deep Reinforcement Learnings verwendet wird. Der Agent wird durch diese Methode repräsentiert. Das DQN basiert auf dem Q-Learning.

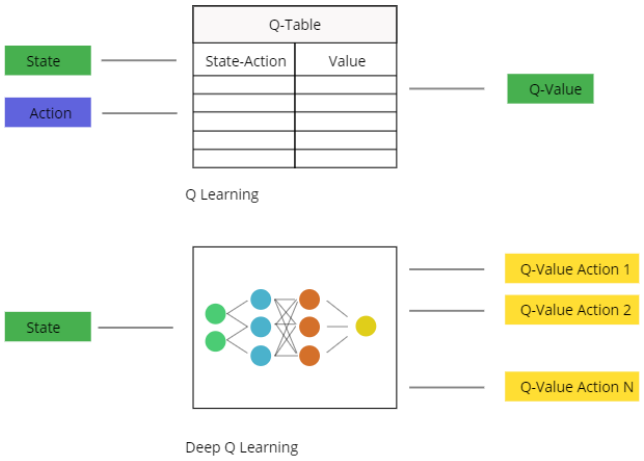

Figure 2: Unterschied Q Learning und Deep Q Learning

Für das DQN werden folgenden Funktionen benötigt:

- value function: V(S) Der erwartete Reward in einem bestimmten State
- Q-value function: Q(s,a) Zusätzlich zu dem State wird nun die Action mitbetrachtet. D. h. welcher Reward ist zu erwarten in einem bestimmten State und einer ausgewählten Action.
- Optimale Q-value function: Q*(s,a) Q(s,a,)

Die optimale Q-value function wird approximiert durch ein Deep Q-learning Network. Für das DQN wird die Q-value function um den Parameter Θ erweitert. Θ repräsentiert die Gewichtungen die genutzt wird, um die optimale Q-value function zu approximieren. Durch diese Funktion kann die Action ausführen werden, von der in diesem State der beste Reward erwartet wird. DQN gehört in den Bereich des Deep Reinforcement learning, da Θ ein Deep Neural Network (DNN) ist. Dabei kann Θ z.B ein Dense Neural Network sein oder auch ein Convolutional Neural Network (CNN). In dem Anwendungsfall von Schiffe versenken wird kein CNN benötigt [11].

3.2 Actor Critic using Kronecker-Factored Trust Region (ACKTR)

ACKTR ist ein "Actor-Critic"-Modell, welches ein Kronecker-Produkt verwendet, um sowohl den Actor als auch den Critic zu optimieren [5]. Hierbei wird ein neuronales Netz zur Schätzung der "Policy" und ein weiteres neuronales Netz zur Schätzung der "Advantage" Funktion verwendet [5]. Ebenfalls wird eine "Trust-Region" Optimierung verwendet, um die Konvergenz zu verbessern [5]. ACKTR skaliert gut mit großen neuronalen Netzen [5]. Die Performance des ACKTR ist höher als die von vergleichbaren Modellen wie A2C, PPO oder TRPO [6]. Dies kann auch der folgenden Grafik entnommen werden:

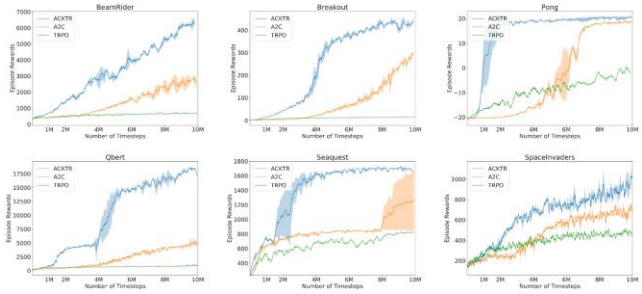

Figure 3: Vergleich der Performance von ACKTR, A2C und TRPO [5].

ACKTR erreicht mit weniger Episoden einen höheren "Reward" verglichen mit A2C und TRPO. Dies kann Grafik 2 entnommen werden:

		ACKTR		A2C		TRPO (10 M)	
Domain	Human level	Rewards	Episode	Rewards	Episode	Rewards	Episode
Beamrider	5775.0	**13581.4**	**3279**	8148.1	8930	670.0	N/A
Breakout	31.8	**735.7**	**4094**	581.6	14464	14.7	N/A
Pong	9.3	**20.9**	**904**	19.9	4768	-1.2	N/A
Q-bert	13455.0	**21500.3**	**6422**	15967.4	19168	971.8	N/A
Seaquest	**20182.0**	1776.0	N/A	1754.0	N/A	810.4	N/A
Space Invaders	1652.0	**19723.0**	**14696**	1757.2	N/A	465.1	N/A

Table 3: Vergleich der Rewards nach einer Anzahl von Episoden [5].

Wie der Tabelle 2 entnommen werden kann, liefert das ACKTR-Modell deutlich höhere Rewards bei weniger Episoden und ist somit ein sehr performantes Modell.

4 Training

Die Agenten wurden für ein 5x5 Feld mit zwei Schiffen der Länge 2 und einem Schiff der Länge 3 trainiert. Die Schiffe wurden mit einer Lücke zu jedem anderen Schiff in jeder Episode neu angeordnet.
Das Training der Agenten wurde hierbei in zwei Schritte aufgeteilt. In erster Linie war es wichtig, dass der Agent nur valide Aktionen ausführt. Eine valide Aktion bedeutet, in diesem Fall ein Feld zu beschießen, dass noch nicht beschossen worden ist. Damit der Agent lernt ein Feld nicht doppelt zu beschießen, wurden verschieden Rewards vergeben.

1. Es wird ein positiver Reward vergeben, wenn ein "neues" Feld beschossen wird.

2. Es wird ein Negativer Reward vergeben, wenn ein Feld doppelt beschossen wird. Das Spiel wird im Anschluss beendet.
3. Es wird ein positiver Reward ausgeschüttet, wenn ein Spiel erfolgreich beendet wird und somit alle Felder beschossen wurden.

Der Agent wurde hierbei mit **1.000.000** Episoden trainiert. Das Resultat sieht hierbei wie folgt aus:

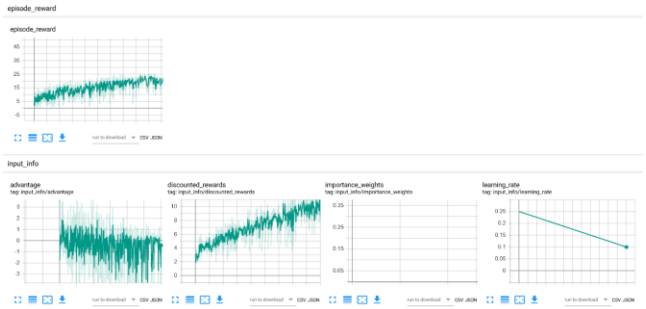

Figure 4: Screenshot des Tensorboards nach dem Training eines Agenten.

Mit dem daraus resultierenden Model wird im zweiten Schritt der Agent darauf trainiert, ein Spiel mit so wenig Zügen wie möglich zu beenden. Damit der Agent im Laufe des Lernens besser wird, wurden folgende Rewards vergeben:

1. Positiver Reward, wenn ein Schiff getroffen worden ist.
2. Je weniger Runden benötigt worden sind, desto größer ist der erhaltene Reward.
3. Bei einem Fehlversuch wird kein Reward ausgegeben.
4. Wenn ein Feld doppelt beschossen wird, wird ein negativer Reward erteilt und das Spiel abgebrochen.

Hier wurde ebenfalls mit **1.000.000** Episoden trainiert. Für das Training wurde ein Rechner mit folgender Hardware genutzt:

- CPU: Intel Xeon E3-1231 v3
- RAM: 16gb RAM DDR3
- GPU: Nvidia Geforce GTX 1080

Es wurde nur auf der CPU trainiert. Die Anbindung der GPU für das Training hat keinen performance Vorteil gegeben.

5 Test

Nachdem die Agenten trainiert wurden und das beste Modell gespeichert worden ist, wurden die Agenten getestet. Es wurden jeweils 100 Episoden gespielt, dabei benötigen die Agenten im Durchschnitt ca. 14-20 Züge.

6 Implementierung

Die Agenten wurden mit Hilfe von OpenAI Gym [7] und dem Reinforcement Learning Algorithmen Framework Stable Baselines, welches auf dem Framework OpenAI Baselines basiert [8], trainiert. Darüber hinaus wird noch die Library NumPy verwendet [9].
Eine der wichtigsten Funktionen, damit ein Agent trainiert werden kann, ist die step Funktion der Gym-Umgebung. Der Agent ruft diese Methode in jedem Zug eines Spiels auf und übergibt eine Aktion, bzw. einen Zug. Innerhalb der step Funktion wird überprüft, ob ein Feld bereits beschossen worden ist, anschließend wird auf das ausgewählte Feld geschossen und es wird der Reward berechnet. Des Weiteren wird die Observation aktualisiert und überprüft, ob das Spiel beendet ist. Die Informationen über die Zustände der Felder und der Schiffe werden aktualisiert und gespeichert. Diese Informationen werden an den Agenten als *after_shot_state*, *reward*, *done*, sowie *info* übergeben. Basierend auf den Werten des *after_shot_states* und des *Rewards* wird die nachfolgende Aktion bestimmt.
Damit die Umgebung weiß, welche Aktionen und Observations erlaubt sind, wird dort der *action_space* und der *observation_space* definiert. In dem Fall von Schiffe versenken basiert der *action_space* auf der Anzahl der Felder. Der *observationspace* hingegen wird durch die Anzahl der Felder repräsentiert, multipliziert mit der Anzahl der Zustände. Daraus resultiert für ein 5x5 Feld ein *action_space* von 25 Aktionen und ein *observation_space* von 100, da 4 Zustände zulässig sind.
OpenAI Gym bietet verschiedene Typen von spaces an. Die am häufigsten verwendeten sind der Discrete- und der Box-Space. Normalerweise würde sich für Schiffe versenken ein *action_space* des Typs Box anbieten. Dieser besteht, wie das Spielfeld, aus einem zweidimensionalen Array. Es musste jedoch ein Discrete-Space gewählt werden, da der DQN-Algorithmus nur mit diesem funktioniert.

7 Ergebnis

Sowohl der DQN- als auch der ACKTR-Agent wurden erfolgreich auf valide Spielzüge trainiert. Es werden zwar gelegentlich noch invalide Spielzüge ausgeführt, jedoch deutlich weniger im Vergleich zu einem untrainierten bzw. random Agenten. Ebenfalls konnten die beiden Agenten im Anschluss erfolgreich darauf trainiert werden, ein Spiel mit möglichst wenig Zügen zu beenden. So benötigt der ACKTR-Agent im Durchschnitt ca. 14-16 Spielzüge. Der DQN-Agent benötigt im Durchschnitt ca. 16-20 Spielzüge. Es ist zu erkennen, dass der ACKTR-Agent, bei gleichem Training, bessere Ergebnisse erzielt als der DQN-Agent. Der ACKTR-Agent benötigt für den Durchlauf von 1.000.000 Spielen ca. 25 Minuten, der DQN-Agent benötigt ca. 50 Minuten. Grund dafür ist, dass der ACKTR-Agent multi threaded genutzt werden kann, der DQN-Agent ist nur single threaded [8].

8 Fazit

Die beiden Agenten konnten erfolgreich mit OpenAI Gym und Stable Bassselines umgesetzt werden. Die Performance der Agenten liegt mit 14-16, bzw. 16-20 Zügen bei einem Spielfeld mit 25 möglichen Zügen, knapp über dem Level der "Hunt" Methode, ca. 12-14 Züge. Jedoch sorgt der feste "Action-Space" von OpenAI Gym dafür, dass doppelte bzw. invalide Spielzüge möglich sind. Somit muss ein Agent in 2 Schritten trainiert werden. Dies ist zwar erfolgreich, jedoch werden einige Spiele immer noch durch invalide Spielzüge beendet. Es ist somit nicht zu 100% gelungen, invalide Spielzüge zu verhindern. Da die Agenten immer doppelt trainiert werden müssen, können diese nicht sehr gut auf Änderungen reagieren. So müssen bei einer Änderung der Spielfeldgröße oder einer Anzahl der Schiffe, die Agenten von Beginn an neu trainiert werden. Es kann davon ausgegangen werden, dass die Agenten besser performen würden, wenn die Schiffspositionen nicht rein zufällig wären, sondern auf einem Sample aus menschlich positionierten Schiffen trainieren würde. Dadurch könnten die Agenten Muster in der Positionierung erkennen und diese nutzen.

9 Future Work

Wie im Fazit bereits erläutert, sorgt die Limitierung von OpenAI Gym für invalide Spielzüge. Hier wäre es interessant weitere Lösungsansätze zu entwickeln, sodass die Agenten nicht mehr doppelt trainiert werden müssen und sich so eventuell besser an neue Änderungen anpassen können. Wäre es möglich, invalide Züge auszuschließen, könnten die beiden Agenten deutlich effizienter trainiert werden, da somit keine Spiele abgebrochen werden müssten. Des Weiteren könnte man beobachten, wie sich unterschiedliche Agenten bei unterschiedlichen Boardgrößen verhalten. Interessant wäre hierbei, ob es erkennbare Muster oder Unterschiede beim Suchen der Schiffe existieren. Diese Erkenntnisse könnten dazu genutzt werden, einen Agenten, der auf einem 5x5 Spielfeld trainiert wurde, auf ein 10x10 Feld zu übertragen.

REFERENCES

[1] BoardGameGeek, LLC: https://www.boardgamegeek.com/boardgame/4122/electronic-battleship
[2] Sutton, Richard S. and Andrew G. Barto. "Reinforcement Learning: An Introduction."
[3] Nick Berry "Battleship" http://www.datagenetics.com/blog/december32011/
[4] Sue He "Deep Reinforcement Learning–of how to win at Battleship" http://www.ccri.com/2017/08/25/deep-reinforcement-learning-win-battleship/
[5] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse and Jimmy Ba "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation"
[6] Khuong N. Nguyen, Jaewook Yoo and Yoonsuck Choe "Speeding Up Affordance Learning for Tool Use, Using Proprioceptive and Kinesthetic Inputs"
[7] Open source project OpenAI Gym https://gym.openai.com/
[8] Open source project StableBaselines https://github.com/hill-a/stable-baselines
[9] Open source project NumPy https://numpy.org/
[10] François-Lavet, Vincent, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau. "An Introduction to Deep Reinforcement Learning." Found. Trends Mach. Learn. 11 (2018): 219-354.
[11] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.