Reinforcement Learning - Chessmaster

Implementierung im Rahmen des Data Exploration Project

A. Bernrieder, J. Brebeck, N. Wichter, and T. Hilbradt

Kurs: WWI18DSB

05. Juli 2020

ABSTRACT

Context. Im Rahmen dieses Projekts wird die Implementierung eines Reinforcement Learning Algorithmus zum Erlernen des Schachspiels untersucht.
Ziel der Ausarbeitung ist dabei ein funktionierendes Modell, dass in der Lage ist gegen einen Menschen zu spielen und eine angemessene Herausforderung darzustellen.
Aims. Das Projekt soll eine vereinfachte Version des Reinforcement Learnings zeigen, dass genutzt wird, um das Schachspielen zu erlernen. Dabei wird besonderer Wert auf eine Implementierung des Monte Carlo Tree Search Algorithmus gelegt, der die Basis für das RL bietet.
Methods. Die Basis des Projekts sind die Arbeiten von Deepmind an der AlphaZero AI (Deepmind, 2018), die Vorreiter im Zuge des Reinforcement Learnings waren und sind. Mit ihrer aktuellen AI, die MuZero AI, haben sie erneut Standards gesetzt. (Schrittwieser, et all, 2020)
Dieses Projekt orientiert sich in seiner Grundstruktur an diesem Vorbild, verwendet also einen Monte Carlo Tree Search zum suchen der optimalen Züge und greift für die Implementierung des Reinforcement Learnings auf einen Temporal Difference Learning Algorithmus (SARSA) zurück.
Results. Das Ziel dieses Projekts konnte in seinen Grundzügen erreicht werden, wobei unklar bleibt inwiefern es gegen menschliche Spieler dauerhaft gewinnen kann. Die Spielstärke der AI kann nicht genau ermittelt werden, aber ein anspruchvolles Spiel gegen die AI ist möglich.
Conclusions. Es kann davon ausgegangen werden, dass weitere Verbesserungen am Modellalgorithmus, insbesondere dem Monte Carlo Tree Search zur Leistungssteigerungen führen werden. Ebenso wird eine erhöhte Trainingszeit dem Modell erlauben besser zu spielen.

1. Einführung

Das Reinforcement Learning stellt eine Unterart des Machine Learnings dar. Hierbei erlernt das Programm, Agent genannt, selbständig ohne äußeres Einwirken, wie ein bestimmtes Problem zu lösen ist. Hierunter fällt das selbständige Erlernen des aufrechten Gangs eines Roboters, das Betreiben eines autonomen Helikopters und viele andere realweltliche Probleme (Kober, et all, 2013).
Jedoch kann diese Art des Lernens auch verwendet werden, um einer schwachen Künstlichen Intelligenz beizubringen ein bestimmtes Spiel zu spielen. Deepmind hat im Jahr 2016 mit AlphaGo bewiesen, dass ein Programm inzwischen menschliche Spieler auch in komplexen Spielen wie Go, einem asiatischen Brettspiel, schlagen kann, indem der damalige Weltmeister dieses Spiels Lee Sedol von dem Programm 2016 in vier von fünf Partien geschlagen wurde (Spiegel, 2016).
Seitdem haben sich die Algorithmen weiterentwickelt, sodass inzwischen selbst Echtzeitstrategiespiele, wie StarCraft II kein Problem mehr sind (Deepmind, 2019). Durch umfrangreiche Veröffentlichungen auf diesem Themengebiet, zahlreichen Tutorials und Online Vorlesungen, wie von David Silver (Silver, 2015) ist es inzwischen auch für Anfänger möglich einfache Reinforcement Learning Algorithmen zu implementieren.
Im Rahmen dieses Projekts soll dies am Beispiel des Schachspiels dargelegt werden. Dieses wurde gewählt, da es intuitiv zugänglich, also schnell erlernbar, ist, sowie eine hohe Spieltiefe bietet. In diesem Report soll beschrieben werden, welche Algorithmen für die Implementierung verwendet wurden, warum diese gewählt wurden, an welchen Vorlagen sich orientiert wurde und was ein potentieller Business Use-Case für dieses Projekt wäre.

2. Implementierung des Algorithmus

Die Basis für dieses Projekt bietet die Arbeit von Arjan Groen in seinen Kaggle Notebooks (vgl. ArjanGroen, 2019), sowie die Videoreihe zum Thema von David Silver. Prinzipiell werden zwei Algorithmen implementiert, die aufeinander aufbauen, bzw. sich gegenseitig bedingen um ein komplettes Modell zu erstellen. Diese wären zum einen ein Monte Carlo Tree Search Algorithmus, sowie zum anderen ein Temporal Difference Algorithmus, im speziellen der SARSA-Algorithmus, der mit Hilfe eines Convolutional Neural Networks (CNN) ausgeführt wird. Wissenschaftlich basiert diese Auswahl auf den Untersuchungen von Ilhan und Etaner-Uyar, 2017.
Für dieses Projekt werden folgende Python-Bibliotheken verwendet:

1. Python-Chess
2. Tensorflow
3. Keras
4. weitere wie Numpy

2.1. Monte Carlo Tree Search

Der Monte Carlo Tree Search (MCTS) ist ein Algorithmus, der genutzt werden kann um in einem Spiel die optimalen Züge zu finden, um den Spieler zum Erfolg zu führen. Der hier verwendete Algortihmus basiert auf den Beschreibungen von Klassert, 2019.
Ein MCTS besteht aus vier Phasen, die iteraktiv wiederholt werden, bis ein Endergebnis ausgegeben wird. Das Endergebnis ist hierbei der optimale Zug vom jetzigen Spielstatus aus. Ein Baum besteht dabei aus Nodes, die speichern, welchen Spielzustand sie haben (aktuelles Schachbrett), welchen Wert sie haben (wie gut ist dieser Zug), sowie wie oft der Algorithmus bereits diesen Node besucht hatte. Die Child-Nodes stellen dabei den weiteren Spielverlauf dar, also können die verbindenden Kanten als die ausgeführten Züge hin zu einem neuen Spielzustand beschrieben werden. Vor der ersten Iteration des MCTS wird eine Parent-Node erstellt, sowie einmalig die Phasen 2-4 des Algorithmus ausgeführt. Dabei wird eine Basis für die iterative Wiederholung geschaffen.
Diese Phasen sind:

1. Selection
   In dieser Phase wählt der Baum die Child-Node, die den optimalen Wert hat, also die größte Wahrscheinlichkeit, dass von diesem Zug ausgehend der Agent das Spiel gewinnt. Hierfür sind unterschiedliche Algorithmen denkbar, jedoch wird in diesem Projekt der Upper Confidence Bound (UCB) Algorithmus verwendet. Dieser bietet den Vorteil, dass der für Tree-Search Algorithmen übliche Exploration / Exploitation Trade-Off vermieden wird. Der Trade-Off bezieht sich darauf, dass andere Algorithmen dazu neigen immer wieder die selben Wege einzuschlagen, also im MCTS immer die selben Child-Nodes zu wählen, da sie den höchsten Wert aller Nodes haben. Da beim Schach jedoch eine langfristige Taktik, die kurzfristig eventuell niedrigere Ergebnisse liefert, ebenfalls erfolgreich sein kann, ist es wünschenswert auch andere Spielzüge durchzuspielen.
   Mathematisch sieht der UCB Algorithmus folgendermaßen aus:

$$UCB(Node_i) = \bar{v_i} + c * \sqrt{\frac{ln(N)}{n_i}} \quad (1)$$

   Der Durchschnittswert der Node wird wie folgt beschrieben:

$$\bar{v_i} = \frac{v_i}{n_i} \quad (2)$$

   v = Wert der Node
   c = Konstanter Parameter (Feintuning der Exploration / Exploitation)
   N = Anzahl der Besuche der Parent-Node von $Node_i$
   n_1 = Anzahl der Besuche von $Node_i$

   Für den c Parameter wird der Wert 2 gewählt, nach dem Beispiel von Levin, 2017.
   Durch diesen Algorithmus wird dafür gesorgt, dass oft besuchte Nodes bei der nächsten Iteration zugunsten einer noch nicht so oft besuchten Node vernachlässigt werden. Falls eine Node bisher nicht besucht wurde, also $n_1 = 0$ ist wird der UCB-Wert dieser Node auf einen sehr hohen Wert gesetzt, damit alle Nodes einmal besucht werden. Bei gleichen UCB-Werten wird eine zufällige Node besucht. Im Zuge der Selection wird für jede Child-Node der UCB-Wert berechnet und anschließend die Node mit dem höchsten Wert besucht, bzw. zurückgegeben.
   Die Selection-Phase startet bei der Root-Node und wird rekursiv solange ausgeführt, bis eine Leaf-Node gewählt wird.

2. Expansion
   Sobald eine Leaf-Node gefunden wurde wird diese erweitert. Das bedeutet, dass von dem Spielzustand dieser Node ausgehend für jeden möglichen Zug eine weitere Child-Node erstellt wird. Beispiel: Falls nur ein einzelner Bauer auf dem Schachfeld wäre (in Startposition) werden zwei Child-Nodes erstellt. Dabei wird die erste Node den Zug um ein Feld vorwärts enthalten und die zweite die mit zwei Feldern vorwärts.

3. Rollout
   In dieser Phase wird zufällig eine Child-Node aus der vorherigen Phase gewählt, in diesem Projekt einfach die erste, von der ausgehend ein ausspielen simuliert wird. Hierbei bieten sich mehrere Möglichkeiten. Zum Einen, ob das Spiel gespielt werden soll, bis ein Sieger feststeht, oder nur eine bestimmte Anzahl an Schritten ausgeführt wird. Diese Auswahl ist bedingt durch die zur Verfügung stehenden Hardware-Ressourcen. Ferner bleibt die Methode des Playouts. Arjan Groen wählt hierfür ein Monte Carlo Playout, was jedoch zwecks der Vereinfachung in diesem Projekt verworfen wurde. Nach Levine kann auch ein zufälliges Ausspielen, das heißt mithilfe von Zufallszügen erfolgreich sein, da es im Mittel die richtigen Ergebnisse ausgibt (vgl. Levin, 2017).

4. Backpropagation
   Abschließend wird betrachtet, welches Ergebnis das Rollout hatte (also der Material-Balance nach dem Spiel). Das Ergebnis des Rollout wird bei jeder Node, einschließend der Parent-Node, zum bisherigen Wert der Node addiert. Dabei wird die Anzahl der Besuche ebenfalls um $N + 1$ erhöht. Es werden also die Werte der jeweiligen Parent-Nodes aktualisiert.

Die Phasen des MCTS werden eine bestimmte Anzahl an Iterationen ausgeführt, bevor ausgehend von der Root-Node bestimmt wird, welche Child-Node den höchsten Wert, also die höchste Wahrscheinlichkeit auf Erfolg hat. Prinzipiell kann davon ausgegangen werden, dass das Ergebnis genauer wird, je mehr Iterationen durchgeführt werden, jedoch steigt der Rechenaufwand dabei signifikant an, da Schachspiele bereits nach wenigen Zügen eine hohe Anzahl an möglichen weitern Zügen bieten. Nach 2 Zügen liegt die Zahl bereits bei 72.084 möglichen Zügen.(vgl. Klein, 2011)
Es muss also ein Mittel gefunden werden zwischen ausreichend Iterationen und sparsamer Verwendung der Ressorucen.

2.2. Temporal Difference Learning (SARSA)

An dieser Stelle kommt der Reinforcement Learning Algorithmus ins Spiel. Hier wird dieser mit einem Value-Network implementiert. Das bedeutet, dass der Algorithmus mit Hilfe von State-Action-Reward-State-Action (SARSA) Temporal Difference (TD) Learning lernt.

Der Temporal Difference Algorithmus ist ein model-free Reinforcement-Learning Algorithmus. Das bedeutet, dass der

Agent durch tatsächliche Erfahrung lernt und nicht mit Hilfe einer Übergangstabelle, in welcher Zwischenwerte aus den vorherigen Iterationen gespeichert werden.
Zu Beginn besitzt der Agent kein Wissen über den State, also dem Zustand des Spielbretts, den Reward ausgeführter Aktionen oder die Transitions, alle erlaubte Züge aus dem jeweiligen State. Der Agent interagiert lediglich mit seiner Umgebung. Dabei führt er anfangs zufällige, später informierte Aktionen durch (vgl. Kumar, 2019). Dabei wird, durch das aktualisieren des bereits bekannten Wissens, ein neuer State erlernt. Das bedeutet, dass der Algorithmus sich kontinuierlich verbessert. Für die Auswahl der verschiedenen States wird die SARSA-Methode verwendet.

SARSA ist eine on-policy Temporal Difference Kontroll Methode. Das bedeutet, dass die Methode die nächste Action, für den jeweiligen State, während des lernen durch folgen einer Policy auswählen kann. Eine Policy is in dem Fall, Tuple aus State und Action. Diese Policiy verändert sich, im Gegnsatz zu anderen Methoden, bei der SARSA-Methode nicht.
Das Ziel ist es das $Q_\pi(s, a)$ für die momentane Policy π zu schätzen für alle dazugehörigen state-action Tuple (s, a). Dies wird erreicht durch die in Formel 3 dargestellte TD Aktualisierungsregel.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (3)$$

α = Lernrate
r = Reward
γ = Gewichtungsfaktor $\gamma \in [0, 1]$

Das Update wird für jede Transition durchgeführt, solange ein nicht-terminaler Zustand für s_t besteht. Dabei wird eine Aktion a aus dem State s, mit der Hilfe der Policy aus Q, abgeleitet. Die ausgewählte Aktion wird durchgefürt, dabei wird der Reward r beobachtet, daraus folgt dann der neue State s'. Danach wird wieder eine neue Aktion a' aus dem State s' gewählt.

Das Ziel des Netzwerks ist es, im Falle des Chessmaster Agent aus einem gegebenen Spielbrett zu prognostizieren welchen Wert die zugehörigen Nodes haben werden, sodass die Aktion mit dem besten Ergebnis von der jetzigen State gewählt werden kann.

Der MCTS soll also darauf beschränkt werden von der Root-Node aus die möglichen Züge darzustellen (Expansion), wonach das Neuronale-Netz für jedes Child-Node den Wert prognostiziert. Dadurch fallen alle übrigen MCTS-Phasen weg, wodurch der Agent in Echtzeit spielen kann.
Hierbei wurde das erstellte Convolutional Neural Network von Arjan Groen übernommen. Input ist das Schachbrett einer Node. Output ist der prognostiziert Wert dieses Bretts.

2.3. Training

Nach der Erläuterung der verwendeten Algorithmen soll hier knapp auf das Training des Agenten (Chessmaster) eingegangen werden.
Der Agent trainiert als weißer Spieler und wird hier auch bessere Ergebnisse erzielen, kann jedoch auch als schwarzer Spieler spielen. Als Trainingsgegner wird ein Myoptic, also ein kurzsichtiger Algorithmus verwendet. Dieser wählt basierend auf allen möglichen Zügen immer den Zug, der beim Gegenspieler den Schaden maximiert.

Um die Performance zu verbessern wurden bestimmte Hyperparameter gesetzt. So dauert ein Spiel maximal 60 Züge, der MCST läuft pro Spielzug mit 128 Iterationen und die simulierte Tiefe im Rollout ist auf maximal 30 Züge festgesetzt.

3. Business Use Case

Das erlernte Modell kann als Trainingspartner für Schachspieler genutzt werden. Sodass man auch ohne einen menschlichen Schachpartner seine Taktiken üben kann. Der wirtschaftliche Aspekt ist hierbei durch das vermarkten als Trainingsprogramm für Schachanfänger und Schachprofis erfüllt. Des Weiteren können die entwickelten Algorithmen Grundlage für weitere AIs für andere Spiele sein, da diese leicht übertragbar sind. @REST?

3.1. Elo-Zahl

Die Bewertungsgrundlage der Stärke eines Schachspielers wird durch die Elo-Zahl wiedergegeben. Diese wird bei Spielern ohne eine Elo-Zahl nach anfänglichen Spielen geschätzt. Nach jedem Spiel lässt sich die neue Elo-Zahl eines Spielers anhand einer festgelegten Formel ermitteln. Dazu muss zuerst der Erwartungswert für den Sieg von einem Spieler bestimmt werden.

$$E_A = \frac{1}{1 + 10^{(R_A - R_B)/400}} \quad (4)$$

E_A = Erwartungswert Spieler A
R_A = Elo-Zahl Spieler A
R_B = Elo-Zahl Spieler B

$$R'_A = R_A + k * (S_A - E_A) \quad (5)$$

R'_A = neue Elo-Zahl Spieler A
k = Gewichtungsfaktor $k \in [10, 40]$
S_A = erreichte Punkte von Spieler A
(Sieg 1, Unentschieden 0,5 und Niederlage 0)

Der gewinnende Spieler bekommt dabei immer genauso viele Punkte gutgeschrieben, wie der verlierende Spieler einbüßt. Bei einem unentschieden verliert der Spieler mit der höheren Elo-Zahl einen Teil seiner Punkte.

4. Ausführen

Um das Modell auszuführen, wurde ein vereinfachtes Modul geschrieben, chessmaster.py. Dieses bietet die Möglichkeit ein Modell zu trainieren und anschließend gegen es zu spielen oder gegen das bereits Bestehende zu spielen.
Für eine graphische Übersicht ist die Verwendung eines Jupyter Notebooks (Graphical.ipynb in der Abgabe) empfehlenswert.

```
from chessmaster import Chessmaster

cm = Chessmaster()

# train model
cm.train_agent(iterations = 10000)

# play against model
cm.play()
```

Es besteht auch die Möglichkeit mithilfe der game.py auf erweiterte Funktionalitäten zuzugreifen, wie das Spielen gegen einen reinen MCST, oder die Auswahl der Spielerfarbe.

5. Ergebnisse

Das trainierte Modell ist in der Lage ...
Um diese Ergebnisse in Zukunft zu verbessern bieten sich bereits einige Bereiche an. Diese liegen insbesondere in der Verbesserung des MCTS, sowie dem gegnerischen Agenten (Myopic Agent). Der MCST sollte seine Ergebnisse verfeinern können, indem statt einem Zufallsplayout im Rollout ein Monte Carlo Playout oder ein andere gewählt wird. Ebenso kann die gewählte Zahl der Iterationen erhöht werden, was jedoch einen erhöhten Ressourcenaufwand bedeutet. Auch die Wahl eines intelligenteren Gegners, z.B. der Stockfish AI, sollte zu einer Leistungssteigerung beitragen. Final kann davon ausgegangen werden, dass bei einer höheren Trainingsepochenanzahl (100.000 und mehr) deutlich bessere Ergebnisse erzielt werden können.

References

[1] Arjan Groen, 2019
Kaggle Notebooks 1-5, abgerufen Mai-Juli 2020
https://www.kaggle.com/arjanso/reinforcement-learning-chess-1-policy-iteration

[2] Deepmind, 2018
Authors: David Silver, Thomas Hubert, Julian Schrittwieser, Demis Hassabis
https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

[3] Deepmind 2019
https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

[4] Ilhan and Etaner-Uyar, 2017
Monte Carlo Tree Search with Temporal-Difference Learning for General Video Game Playing
IEEE Conference on COmputational Intelligence and Games 2017
https://ieeexplore-ieee-org.ezproxy-dhma-2.redi-bw.de/stamp/stamp.jsp?tp=&arnumber=8080453&tag=1

[5] Klassert, 2019
Monte-Carlo tree search
https://hci.iwr.uni-heidelberg.de/system/files/private/downloads/297868474/report_robert-klassert.pdf

[6] Kober J., Bagnell A., Peters J.
Reinforcement Learning in Robotics: A Survey
https://www.ias.informatik.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf

[7] Levine John, 2017
Monte Carlo Tree Search
https://www.youtube.com/watch?v=UXW2yZndl7U

[8] Schrittwieser, et all, 2020
Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model
https://arxiv.org/pdf/1911.08265.pdf

[9] Silver David, 2015
UCL Course in RL
https://www.davidsilver.uk/teaching/
https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=2&t=0s

[10] Spiegel, 2016
Software schlägt Go-Genie mit 4 zu 1
https://www.spiegel.de/netzwelt/gadgets/alphago-besiegt-lee-sedol-mit-4-zu-1-a-1082388.html

[11] Sutton, 2005
6.4 Sarsa: On-Policy TD Control
http://incompleteideas.net/book/ebook/node64.html

[12] Klein, 2011
Wie berechenbar ist das Schachspiel?
http://www.sfbux.de/wp-content/uploads/artikel/berechenbarkeit.pdf

[13] Vaibhav Kumar, 2019
Reinforcement learning: Temporal-Difference, SARSA, Q-Learning Expected SARSA in python
https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e