Implementierung eines Computergegners für das Spiel Chamäleon Schach (ein schachähnlichen Brettspiel)

Hannes Dröse

Zusammenfassung der Arbeit ...

Inhaltsverzeichnis

Ziel dieser Arbeit

- Implementierung eines Computergegners für das Brettspiel Chamäleon Schach
- Verwendung in einer App, mobile Endgeräte -> gute Performance nötig, da beschränkte Ressourcen
- Computergegner soll so stark wie möglich sein (nachträgliches Abschwächen ist immer möglich)
- App in React Native implementiert -> nach Möglichkeit ist alles in Node.js umzusetzen
- Vergleich zwischen klassischem heuristischen Ansatz und modernem maschinellem Lernen (TD)
- Implementierung beider Algorithmen, der bessere gewinnt

Chamäleon Schach

- schachähnliches Brettspiel von meinem Opa erfunden
- 1982 entwickelt und gebaut (Holzausführung) -> Foto einfügen
- 1990/92 verlegt vom Verlag VSK (Großkopf und Schubert 2019, s. 31)
- nun Entwicklung des Spieles als Smartphone/Tablet-App durch den Enkel (mir)
- Umbenennung in Chamäleon Schach

Spielregeln

als Anhang?

- buntes Schachbrett mit vier Farben (rot, grün, gelb, blau)
- bis zu vier Spieler (rot, grün, gelb, blau)
- 4 Figuren pro Spieler, Figuren haben Farbe-Rolle-Zuweisung -> Rolle der Figur von Farbe des aktuellen Feldes abhängig (Rollen: Springer, Dame, Läufer, Turm)
- gespielt wird reihum, wer dran ist bewegt eine seiner Figuren entsprechend der Regeln (Rollen)
- Spielbrett schrumpft -> Größe wir von den äußersten Figuren bestimmt
- Ziel: der letzte Überlebende zu sein
- ...

-> Quellenangabe aus der Spielanleitung nötig?

Spieltheorie

Einordnung von Chamäleon Schach

- zugbasiert: fest definierte Spielzustände zwischen den Zügen
- zeitunabhängig
- perfect information game: keine versteckten Information, keine Zufallskomponenten
- finite:
  - Spiele haben fest definiertes Ende
  - kein Unentschieden möglich -> es gibt immer einen klaren Gewinner
- Multiplayer: 2-4 Spieler
- constant sum game:
  - was gut ist für den einen, ist schlecht für den anderen immer im gleichen Maße
  - keine Kooperationen

(vgl. Luckhart und Irani 1986)

Spielanalyse – Game Tree

- Analyse von Spielen dieser Art ohne Probleme möglich über einen Game Tree
- Game Tree
  - Darstellung aller möglichen Spielverläufe in Baumstruktur
  - Knoten sind die jeweiligen Spielzustände (Game States)
  - Kanten sind die Züge, die von einem Zustand in den flogenden führen
  - dadurch umfassende Analyse möglich
    - Fairness über den Saddle Point
    - optimale Strategie durch z.B. den Minimax-Algorithmus

(vgl. Luckhart und Irani 1986)

Beispiel TicTacToe:

- Regeln: (müssen Regeln zitiert werden? falls ja:) (Abu Dalffa, Abu-Nasser, und Abu-Naser 2019)
  - 3x3 Felder
  - 2 Spieler: X und O
  - nacheinander platzieren die Spieler ihr Symbol auf den noch freien Feldern
  - Gewinner: drei in einer Reihe (horizontal, vertikal oder diagonal)
  - Unentschieden: keine freien Felder mehr

-> Grafik Spielbaum TicTacToe für die ersten 3 Ebenen

Perfektes Spiel – Minimax-Algorithmus

- nach kompletten Aufbau des Spielbaums -> Bewertung der Endzustände möglich
- bsp: TicTacToe: X gewinnt -> +1, O gewinnt -> -1, unentschieden -> 0
- Backpropagation der Bewertungen nach Minimax-Prinzip
- Minimax:
  - policy für X -> immer die höchste Bewertung wählen (maximieren)
  - policy für O -> immer die niedrigste Bewertung wählen (minimieren)

-> Grafik: Ausschnitt aus Game Tree auf unterester Ebene mit Bewertungen

- Minimax-Bewertungen bis nach ganz oben durch propagiert
  - Grafik: Bewertungen der obersten 3 Ebenen
- Analyse: Bewertung des Startzustands -> Saddle Point -> Faires Spiel?
- Minimax gibt uns Bewertungsfunktion:
  - $V^* : s \to v, s \in S, v \in \mathbb{R}$
  - S..Menge der Game States
- Strategie für perfekten Spieler:
  - alle möglichen Folgezüge bewerten
  - höchesten oder niedrigesten wählen (je nach Strategie)

Prinzip beschrieben von Shannon (1950), Name Minimax erst später z.B. Duncan Luce und Raiffa (1957) oder Luckhart und Irani (1986)

Multiplayer-Problem – MaxN-Algorithmus

- Minimax nur für 2 Spieler Spiele -> Bewertung mit skalarem Wert möglich -> Minimax-Theorem (Duncan Luce und Raiffa 1957)
- für mehr als 2 Spieler -> Bewertung als Vektor
- maxN-Algorithmus wie Minimax, nur eben Vektor mit Bewertung für jeden Spieler einzeln
- jeder Spieler maximiert nur seinen eigenen Wert im Vektor
- ansonsten genau der gleiche Ablauf

(Luckhart und Irani 1986)

Komplexität von Brettspielen

1. Problem: Rechenzeit

- Game Trees explodieren sehr schnell in ihrem Umfang -> exponentiell
- TicTacToe: Anzahl möglicher Spielverläufe: !!!Zahl suchen!!!
  - klein und überschaubar
  - Spiele haben feste Länge (5-9 Züge)
- bei komplexen Spielen nur mit Hilfe des Spielbaums abschätzbar:

  - Branching Factor b -> wieviel mögliche Züge hat eine Spieler im Durchschnitt
  - Depth d -> wieviele Züge braucht ein Spiel im Durchschnitt bis zum Ende?
  - Spielverläufe: b^d

- Schach: $b = 30..35$, $d = 80$, Spielverläufe: min. 10^{120} (Shannon-number) (Shannon 1950)
- Chamäleon-Schach: $b = 25$, $d = 40$, Spielverläufe: min. 10^{48} ????? Hier noch genauere Zahlen ermitteln,eventuell getrennt nach 2,3,4 Spieler Spielen

-> Vergleich der Komplexität als Tabelle!

2. Problem: Speicherbedarf für (s, v)

-> Tabelle mit Anzahl möglicher Game States (wieviele Game States es gibt):

- TicTacToe: !!!Zahl suchen!!!
- Schach: 10^{43} (Shannon 1950)
- Chamäleon-Schach: $\sum_{i=1}^{16} (\binom{64}{i} \times \frac{16!}{(16-i)!}) \approx 10^{28}$

Fazit: Ermittlung und Speicherung von V^* bei Chamäleon Schach nicht möglich

Also gilt es V^* so gut wie möglich zu approximieren

Klassischer Ansatz – Heuristischer Ansatz

- Standard Methode: Analyse des Game States anhand bestimmter Heuristiken (Shannon 1950) (Sutton und Barto 2018, s. 181-182)

Grundprinzip

- Heuristiken i.d.R. von Menschen festgelegt
- werden für die Spieler einzeln ermittelt und dann miteinander verglichen
- Ergebnis: Bewertung welcher Spieler gerade wahrscheinlich am gewinnen ist
- Außerdem Verwendung von beschränktem Minimax
  - nicht die direkten nächsten Zustände werden bewertet
  - sondern Game Tree wird bis zu einer gewissen Tiefe (abhängig von Ressourcen) aufgebaut
  - unterste Knoten werden bewertet
  - Ergebnis wird nach oben propagiert
  - Je weiter in der Zukunft die Bewertung erfolgt, desto genauer ist sie (weil im Terminate State ist Bewertung ja eindeutig)

-> Grafik wieder TicTacToe als Beispiel ???

Optimierungen – Pruning

- beim Aufbau des beschränkten Spielbaums können Optimierungen erfolgen
- Zweige müssen nicht weiterverfolgt werden, wenn absehbar ist, dass für die oberste Ebene keine Verbesserung erfolgen kann
- Beschneidung des Baumes -> pruning
- im worst-case ist Laufzeit nach wie vor $O(b^d)$
- Problem: bei Minimax (2 Spieler) sehr effektives Alpha-Beta Pruning möglich
  - reduziert die Laufzeit im best case auf $O(2b^{\frac{d}{2}})$ (Luckhart und Irani 1986)
  - funktioniert bei maxN aber nicht!
- Arten von Pruning bei maxN: (Sturtevant und Korf 2000)
  - Immediate Pruning:
    - möglich wenn es eine Maximal-Bewertung (höchstmögliche) gibt
    - sobald ein Zweig mit Maximal-Bewertung gefunden worden ist, können die anderen Zweige direkt ignoriert werden
  - Shallow Pruning:
    - möglich, wenn die Werte im Bewertungsvektor eine fest Summe v_{maxSum} haben

    - wenn der Vater schon eine vorläufige Bewertung hat, dann kann Kindzweig komplett ignoriert werden, sobald dieser eine Bewertung über $v_{maxSum} - bestScore_{parent}$ bekommt -> Vater wird diesen Weg niemals wählen

-> Grafik, die Immediate und Shallow Pruning darstellt

Implementierung für Chamäleon Schach

(Dieser Bereich ist komplett selbst erdacht, daher keine/kaum Quellen, ist das legitim? Es bezieht sich ja alles auf die vorher gezeigten Prinzipien)

Heuristik

- f: Anzahl der Figuren je Spieler
- z: Anzahl der Züge ein Spieler kann im nächsten Zug machen (starke Rollen z.B. Damen bevorzugen)

$absScore_{player} = f + z \times 0.01$

- z soll weniger Gewicht haben als die Anzahl der Figuren

Finaler Ergebnis-Vektor

- bisher nur Vektor mit absoluten Bewertungen der einzelnen Spieler
- aber: noch kein Bezug zu den Bewertungen der anderen
- Manhattan-Normalisierung des Vektors (das ist ein mathematisches Prinzip, braucht man da eine Quelle für?)
  - $relScore_{player} = \frac{absScore_{player}}{sum(absScore_{all})}$
- dadurch:
  - höchstmögliche Bewertung = 1
  - Summe der Einzelwerte im Vektor = 1
  - heißt: Immediate und Shallow Pruning sind möglich
- im Prinzip stellt der Vektor nun die Gewinnwahrscheinlichkeiten der einzelnen Spieler in Prozent dar

Ermittlung der Suchtiefe

- Frage: wie weit baut man den Game Tree auf?
- Grenzwert: maximale Laufzeit 1 Sekunde
  - in der App wird ca. 1 Sekunde gewartet, auch wenn der Computergegner schon fertig ist, weil es besser aussieht und wenn der Zug nicht sofort apruppt ausgeführt wird
- experimentell ermittelt:
  - Zufallsspiele generieren
  - für jeden Zug MaxN mit gewisser Tiefe ausführen und Laufzeit messen
  - Analyse der Ergebnisse:
    - wann ist eine höhere Suchtiefe möglich, wann nur eine geringe?
    - Vermutung: je weniger Figuren noch übrig sind, desto tiefer kann die Suchtiefen sein
  - Suchtiefen mit Hilfe der Analyse festlegen
  - erneut MaxN mit allen Spielen durchführen und Laufzeit messen
  - Optimierung der Suchtiefe bis keine Berechnung mehr länger als 1 Sekunde dauert

-> Ausschnitt aus der Tabelle mit den Messwerten -> zeigen welche Faktoren die Laufzeit verlängern

-> finale Tabelle mit den Bedingungen und der zugeordneten Suchtiefe

Auswertung

10 Spiele gegen mich spielen -> wer gewinnt ermitteln + subjektive Bewertung des Spielverhaltens

(ist das legitim? Viel zu messen gibt es hier nicht)

Moderner Ansatz – Temporal Difference Learning

Nach wie vor gesucht möglichst gute Approximation von $V^*(s)$

Einordnung und Grundbegriffe

Maschinelles Lernen

Reinforcement Learning: (Sutton und Barto 2018, s. 1-3)

- engl.: Bestätigung oder Ermutigung (Quelle für Übersetzungen nötig?)
- ein System lernt, was in einer bestimmten Situation die beste auszuführende Aktion ist
- für ausgeführte Aktionen bekommt das System Belohnungen
- das System versucht die größtmögliche Menge an Belohnungen zu akkumulieren
- dies geschieht durch Ausprobieren (Exploration)
- und Anpassung des Verhaltens, sodass mehr und mehr Belohnungen erreicht werden können (Exploitation)

Temporal Difference Learning (Sutton und Barto 2018, s. 9-10)

- Algorithmen, die in einer unbekannten Umgebung Aktionen ausführen
- Lernen direkt aus der ausgeführten Aktion und passen ihr Verhalten an
- Lernen geschieht direkt nach der Aktion und nicht erst am Ende der Episode
- Lernen aus der zeitlichen Differenz von t zu $t + 1$

Markov Decision Process: (Sutton und Barto 2018, s. 73)

- Formalisierung von Problemen Reinforcement Problemen
- finite (Sutton und Barto 2018, s. 73)
  - klares Ende und diskrete Zeitschritte
  - alle MDPs werden immer zu finiten umgewandelt, da leichter zu lösen
- besteht aus: (Sutton und Barto 2018, s. 6, s. 47-50 und s. 53)
  - agent: der Algorithmus, der Entscheidungen treffen und daraus lernen soll
  - environment: alles, was zum System gehört, aber nicht zum Agenten
  - reward signal:
    - $R(s, a)$ gibt dem Agenten im Zustand s, wenn er die Aktion a ausführt eine Belohnung
    - die Belohnung ist ein skalarer Zahlenwert -> positiv ist gut, negativ ist schlecht
    - repräsentiert das Ziel des Agenten
    - steht für die direkte kurzfristige Belohnung
    - wird auch reinforcement function genannt

  - value function:
    * $V(s,a) = R(s,a) + V(s',a')$ ist die erwartete Summe aller Belohnungen von jetzt bis zum Ende der Episode
    * repräsentiert das langfristig erwartete Ergebnis
    * ist genau unsere Funktion V^*, die wir approximieren wollen, da sie ja auch vorhersagt, wie das Spiel enden wird, wenn diese Aktion ausgeführt wird
  - policy:
    * die Strategie, wie der Agent aus den nächsten möglichen Aktionen eine auswählt und durchführt
    * hatten wir auch schon bei Minimax und maxN -> Spieler wählt immer den Zug mit der besten Bewertung

- wichtige Begriffe:
  - episode: kompletter Durchlauf eines MDP vom Startzustand bis zum Endzustand (terminate state)
  - state s: aktueller Zustand der Environment, nächster state: s'
  - action a: Aktion, die der Agent im aktuellen Zustand ausführen kann, actions im nächsten state: a'
  - afterstate: (Sutton und Barto 2018, s. 136-137)
    * häufig ist die Action a untrennbar mit dem daraus resultierendem nächsten State s verbunden
    * dann kann man das vereinfachen, indem man statt State-Action Paaren (s,a), einfach von sog. Afterstate s' spricht
    * also: s -> aktueller Zustand, s' -> Folgezustand
      · $(s,a) \equiv s'$
      · $(s',a') \equiv s''$

Hinweis zum Discounting Factor γ: (ist dieser Hinweis nötig?)

- in meisten Formeln werden nachfolgende Values mit einem Discounting Factor versehen
- $V(s,a) = R(s,a) + \gamma V(s',a')$
- dadurch können zukünftige Rewards weniger stark gewichtet werden als der aktuell erhaltene Reward $R(s,a)$
- Sutton und Barto (2018), s. 253-254 zeigt aber, dass selten Sinn macht, vor allem wenn die Value Function nicht als Tabelle dargestellt werden kann
- daher hier in allen Formeln ignoriert

$TD(0)$-Algorithmus

- einfachste Form der TD-Algorithmen
- wir haben eine Value Function $V(s,a)$, die zunächst zufällige Ergebnisse produziert
- Agent bewegt sich durch die Episode und lernt mit jedem Schritt aus den Ergebnissen des vorherigen
- update der Value function nach jeden Schritt:
  - $V(s,a) < -V(s,a) + \alpha(R(s,a) + V(s',a') - V(s,a))$

  - Value function wird mit dem Reward und dem Ergebnis der Value Function geupdated

- Zeigen am Beispiel TicTacToe:
  - Bewertung der nächsten States -> der beste wird ausgewählt
  - nun ist Gegner am Zug, wählt seinerseits den besten Zug
  - Korrektur der Bewertung aus s' und der Bewertung v' ' des Gegners
  - Bsp: Bewertung für $(s, a) = 0.3$, Bewertung für $(s', a') = -0.4$
  - Anpassung der Funktion, sodass in Zukunft (s, a) näher an -0.4 ist
- $TD(0)$ -> wir lernen nur aus dem unmittelbar ausgeführten Schritt
- $TD(\lambda)$ -> wir lernen nur aus weiteren folge Schritten

(Sutton und Barto 2018, s. 119-121)

Neuronale Netze

- benötigt für die Value Function: Funktion, die lernfähig ist
- eine elegante Variante: Neuronale Netze
  - Funktions-Approximator für beliebige auch nicht-lineare Funktion

... entsteht gerade noch

Implementierung für Chamäleon Schach

Allgemein

- nur Verwendung von afterstates s', anstatt (s, a)
- reward signals und Ergebnisse der value function sind 4D-Vektoren
  - ein Wert pro Spieler: (Rot, Grün, Gelb, Blau)
- Algorithmus lernt indem es gegen sich selbst spielt (self-play) (vgl. Ghory 2004, s. 23)
  - beste Variante, einfach umzusetzen
  - Training ist am besten, wenn Gegner gleich stark
  - kein Optimierung nur auf eine Strategie

Reward Signal

- zwei Hauptansätze: (Harmon und Harmon 1997)
  - Minimum Time to Goal
    - wenn nicht terminate state: -1
    - wenn terminate state: 0
    - -> Agent wird umso mehr bestraft, je länger er braucht
  - Pure Delayed Reward and Avoidance Problems
    - wenn nicht terminate state: 0
    - wenn terminate state: +1

    - $*$ -> Agent will einfach nur gewinnen
- für Chamäleon Schach -> 2. Ansatz

$$R(s) = \begin{cases} (0,0,0,0) & \text{, für } s \text{ kein terminate state} \\ (1,0,0,0) & \text{, für Spieler Rot hat gewonnen} \\ (0,1,0,0) & \text{, für Spieler Grün hat gewonnen} \\ (0,0,1,0) & \text{, für Spieler Gelb hat gewonnen} \\ (0,0,0,1) & \text{, für Spieler Blau hat gewonnen} \end{cases}$$

Value Function

- wichtig: Value function gibt den immer den erwarten Reward ohne den gerade erhaltenen an
    - $V(s) = R(s) + V(s')$
- daher: wenn terminate state, dann $V(s) = (0,0,0,0)$
- weil Spiel vorbei -> keine weiteren Rewards möglich

$$V(s) = \begin{cases} R(s) + V(s') & \text{, für } s' \text{ kein terminate state} \\ R(s) + V(s') & \text{, für } s' \text{ ist terminate state} \end{cases}$$

$$V(s) = \begin{cases} 0 + V(s') & \text{, für } s' \text{ kein terminate state} \\ R(s) + 0 & \text{, für } s' \text{ ist terminate state} \end{cases}$$

$$V(s) = \begin{cases} V(s') & \text{, für } s' \text{ kein terminate state} \\ R(s) & \text{, für } s' \text{ ist terminate state} \end{cases}$$

- im Falle eine terminate states kommt die Bewertung also von der Reward Function
- im Falle eines nicht terminate states kommt die Bewertung vom Neuronalen Netz

Policy

- policy im Live-Betrieb genau wie bei maxN: $\pi^*(s) = \arg\max_{s'} V(s')$
- also immer Wert mit der besten Bewertung nehmen
- wird auch greedy policy (Sutton und Barto 2018, s. 27) oder optimal-policy (Harmon und Harmon 1997) genannt
- Problem unsere Funktion startet mit komplett zufälligen Werten
- daher zufällig manche sehr gute Züge sehr schlecht bewertet -> werden nicht genommen und nie geupdated (also verbessert)
- deshalb immer mal einen Zufallszug ausführen, statt immer den vermeintlich besten
- nennt sich ϵ-greedy policy (Sutton und Barto 2018, s. 28)
- $\pi(s)$ -> ohne Stern heißt ϵ-greedy, mit Stern optimal
- Wahrscheinlichkeit von 10% für Zufallszug

Architektur des Neuronalen Netzes

- Input Layer: 384 Neuronen + 2 Neuronen = 386 Neuronen
  - 8x8 Felder: (64 x 6 Neuronen = 384 Neuronen)
    - Status: frei, deaktiviert, Figur auf Feld -> 2 Neuronen
    - Spieler: 4 Spieler -> 2 Neuronen (wenn keine Figur auf Feld -> (0,0))
    - Figur-Art: 4 verschiedene -> 2 Neuronen (wenn keine Figur auf Feld -> (0,0))
  - Spieler am Zug: 4 spieler -> 2 Neuronen
- Hidden Layer 1:
  - Aktivierungsfunktion: Sigmoid
  - Neuronen: testen -> 20, 40, 80, 160, 320
- Output Layer:
  - Aktivierungsfunktion: Softmax -> Werte zwischen [0,1], Summe aller Werte ist 1 -> Gewinnwahrscheinlichkeit (wie bei heuristisch)
  - Neuronen: 4 Spieler -> 4 Neuronen (Gewinnwahrscheinlichkeit je Spieler in Prozent)

Trainingsalgorithmus

$w \leftarrow initNN()$

while infinite

$s \leftarrow$ begin new game

while s not terminate state

$s' \leftarrow \pi^*(s)$

$v \leftarrow V(s')$

$w \leftarrow trainNN(w, s, v)$

$s \leftarrow \pi(s)$

-> in regelmäßigen Abständen gegen maxN antreten lassen und schauen, wer gewinnt

wieviele Spiele spielen sie gegeneinander ?

Auswertung

-> Graph Trainingsdurchläufe zu Gewinnen gegen maxN in Prozent

-> wird TD irgendwann besser als maxN

-> TD gegen mich, Gewinnstatistik und Subjektiver Eindruck

Ausblick

- Champion wird implementiert und in die App eingefügt
- viel Raum für weitere Versuche:
  - heuristisch:
    - andere Bewertungen:
      - Brettgröße
      - Rolle der Figuren -> Rollenkombinationen ?
      - Ausdehnung über das Feld
  - TD-Learning:
    - verschiedene Netz-Topologien
    - andere Aktivierungsfunktionen
    - andere Kodierung der Input-Werte
    - anderer Gegner statt Self-Play -> gegen Algo, gegen Database, gegen Random

---

FRAGEN:

- muss jedes Skript gezeigt und erklärt werden? Oder reichen auch nur die Endergebnisse bzw. dass ich erkläre, was ich gemacht habe?
- müssen Ergebnisse reproduzierbar sein? (seedable Pseudo-Zufallsgeneratoren)
- Ich nutze natürlich exzessiv die Datenstrukturen und Funktionen aus der App für die Spiellogik. Muss die nochmal komplett erklärt und gezeigt werden?
- Generell: Skripte und Code anhängen oder einbetten?
- muss ich Beweise nochmal führen oder kann ich auf das entsprechende Paper verweisen?
- was soll english, was soll deutsch sein?
- veröffentlichung? auch auf english?
- Bücher bekommen ?
- Auch andere Arten der Zitatvermerke erlaubt?
  - laut Vorgabe: Sut18
  - schöner finde ich: Sutton 2018

Quellen

Abu Dalffa, Mohaned, Bassem S Abu-Nasser, und Samy S Abu-Naser. 2019. „Tic-Tac-Toe Learning Using Artificial Neural Networks“.

Chaslot, Guillaume, Sander Bakkes, Istvan Szita, und Pieter Spronck. 2008. „Monte-Carlo Tree Search: A New Framework for Game AI.“ In *AIIDE*.

Duncan Luce, R, und Howard Raiffa. 1957. „Games and decisions: Introduction and critical survey“. *New York, Jone W iley & Sons, lnc* 1: 958.

Fridenfalk, Mikael. 2014. „N-Person Minimax and Alpha-Beta Pruning“. In *NICOGRAPH International 2014, Visby, Sweden, May 2014*, 43–52.

Ghory, Imran. 2004. „Reinforcement learning in board games“. *Department of Computer Science, University of Bristol, Tech. Rep* 105.

Großkopf, W., und L. Schubert. 2019. *Gesellschaftsspiele: Zum Mitmachen, Nachmachen und Selberbauen*. Selbstverlegt.

Harmon, Mance E, und Stephanie S Harmon. 1997. „Reinforcement Learning: A Tutorial.“ WRIGHT LAB WRIGHT-PATTERSON AFB OH.

Luckhart, Carol, und Keki B Irani. 1986. „An Algorithmic Solution of N-Person Games.“ In *AAAI*, 86:158–62.

Nissen, Steffen, und others. 2003. „Implementation of a fast artificial neural network library (fann)“. *Report, Department of Computer Science University of Copenhagen (DIKU)* 31: 29.

Patterson, J., und A. Gibson. 2017. *Deep Learning: A Practitioner's Approach*. O'Reilly Media. https://books.google.de/books?id=qrcuDwAAQBAJ.

Rashid, T., und F. Langenau. 2017. *Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python*. Animals. O'Reilly. https://books.google.de/books?id=b9N3DwAAQBAJ.

Rey, G. D., und K. F. Wender. 2011. *Neuronale Netze: eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. Aus dem Programm Huber: Psychologie-Lehrbuch. Huber. https://books.google.de/books?id=CJSrbwAACAAJ.

Shannon, Claude E. 1950. „XXII. Programming a computer for playing chess“. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41 (314): 256–75.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, u. a. 2017. „Mastering chess and shogi by self-play with a general reinforcement learning algorithm“. *arXiv preprint arXiv:1712.01815*.

———. 2018. „A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play“. *Science* 362 (6419): 1140–4.

Sturtevant, Nathan R, und Richard E Korf. 2000. „On pruning techniques for multi-player games“. *AAAI/IAAI* 49: 201–7.

Sutton, R. S., und A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press. https://books.google.de/books?id=6DKPtQEACAAJ.

Tesauro, Gerald, und Terrence J Sejnowski. 1988. „A'neural'network that learns to play backgammon“. In *Neural information processing systems*, 794–803.