Bestärkendes Lernen

M. Heckel

1 EINLEITUNG

Häufig werden neuronale Netze trainiert, indem das Netz ein Problem löst, für das die Lösung bereits bekannt ist. Dieser Ansatz wird als **überwachtes Lernen** bezeichnet. Eine typische Problemklasse, bei der dieser Ansatz zum Einsatz kommt, sind Klassifizierungsprobleme. Dabei soll ein Netz lernen, Objekte (z.B. Bilder) einer von mehreren gegebenen Kategorien zuzuordnen. Das Netz wird mit beschrifteten Datensätzen trainiert, d.h. jedes Objekt des Datensatzes wurde bereits klassifiziert. Das Netz lernt auf Basis dieser Daten, entsprechende Objekte zu klassifizieren.

Solche Netze lassen sich meist gut trainieren und erzielen gute Ergebnisse. Der Nachteil besteht allerdings darin, dass zum Training entsprechende beschriftete Datensätze notwendig sind. Bei manchen Problemstellungen ist dies nicht möglich, weshalb dieser Ansatz nicht zur Lösung aller Problemtypen verwendet werden kann.

Der Ansatz des **selbstbekräftigenden Lernen** löst dieses Problem, indem die Datensätze zum Training dynamisch während der Interaktion des Netzes mit der Umgebung gesammelt werden. Ein typischer Anwendungsbereich des selbstbekräftigenden Lernen sind Spiele: Ein Spieler interagiert mit der Umgebung, indem er Aktionen ausführt. Diese Aktionen können den Zustand der Umbegung verändern. Das Ziel besteht darin, mit Hilfe des Netzes eine gute Aktion in einem gegebenen Zustand zu wählen.

Dabei besteht allerdings das Problem, dass es häufig nicht möglich ist, eine Aktion während eines Spiels zu bewerten, da die Bewertung dieser Aktion von den folgenden Aktionen abhängt. Im Rahmen dieser Arbeit werden einige Grundlagen und Ansätze vorgestellt, dieses Problem zu lösen.

Im Rahmen der Studienarbeit sollte eine Anwendung entwickelt werden, die das Verhalten des entwickelten Netzes möglichst anschaulich demonstriert. Im Abschnitt 2 “Theorie” wurden einige der möglichen Ansätze vorgestellt. In den folgenden Abschnitten wurde das Netz auf Basis von Deep-Q-Learning implementiert. Darauf hin wurde eine grafische Oberfläche entwickelt, um die Fähigkeiten und Grenzen des Netzes anschaulicher nachvollziehen zu können. Außerdem wurden verschiedene Trainingsmodi implementiert, um diese miteinander vergleichen zu können.

2 THEORIE

2.1 Allgemeine Ansätze

Die Herausforderung beim selbstbestätigenden Lernen besteht darin, ein neuronales Netz so zu trainieren, dass es mit Hilfe dieses Netzes möglich ist, in einem gegebenen Zustand der Umgebung eine möglichst gute Aktion zu wählen. Um dies zu erreichen, gibt es zwei grundlegende Ansätze: das Lernen der **Aktion** und das Abschätzen des **Wertes** eines gegebenen Zustands.

Wenn das Netz lernen soll, welche Aktion a es in einem Zustand s wählen soll, ist es notwendig, über entsprechende Tainingsdaten zu verfügen. Allerdings ist es häufig nicht möglich, direkt abzuschätzen, wie gut eine Aktion in einem Zustand ist, da der Wert dieser Aktion von den folgenden Aktionen abhängt.

Ein gutes Beispiel dafür ist das Bauernopfer beim Schach: Ein Spieler opfert einen Bauern, um dadurch einen anderweitigen Vorteil im Spiel zu erlangen. Wenn man diese Aktion direkt abschätzen würde, wäre die Aktion ziemlich schlecht, da der Spieler ja eine Figur verloren hat, ohne dadurch (direkt) einen Vorteil zu haben. Der Vorteil wird erst durch Distanz deutlich (z.B. da der Spieler im nächsten Zug eine Figur des Gegners schlägt). Bei komplexeren Strategien kann es mehrere Züge dauern, bis der durch einen früheren Zug erzielte Vorteil zur Geltung kommt. Der einzige Zeitpunkt, an dem der Wert einer Aktion direkt angegeben werden kann, ist das Spielende, da es ab diesem Zeitpunkt nicht mehr möglich ist, durch vorherige Züge erhaltene Vorteile zu nutzen.

Ein analoges Problem besteht beim Lernen des Wertes v eines gegebenen Zustands s: Es ist möglich, dass ein durch eine Aktion erreichter Zustand einen schlechten Werte hätte (z.B. da der Gegner mehr Figuren im Spiel hat), dadurch allerdings ein Vorteil erzielt wird, der einige Züge später genutzt werden kann.

Im Abschnitt “Q-Learning” wird beschrieben, wie es möglich ist, einen Zustand, der kein Endzustand ist, dennoch zu bewerten.

Bei neuronalen Netzen handelt es sich um hochdimensionale Optimierungsprobleme, da der gesamte Abstand der Vorhersagen des Netzes zu den gewünschten Ergebnissen minimiert wird. Die Parameter (“Gewichten”) des Netzes sollen also so angepasst werden, dass dieser Abstand minimal wird. Das Ziel des Trainings ist es, ein gutes Minimum zu finden. Dabei kann das Problem auftreten, dass die Parameter so optimiert wurden, dass ein Minimum gefunden wurde, das gefundene Minimum allerdings nicht besonders gut ist. Um die Darstellung etwas zu vereinfachen, wurde eine Funktion mit einem Parameter verwendet ($f(x) = 3sin(x + 25) + 3cos(x + 25) - sin(0.5(x + 25)) - 20cos(0.1(x + 25)) + 30$):

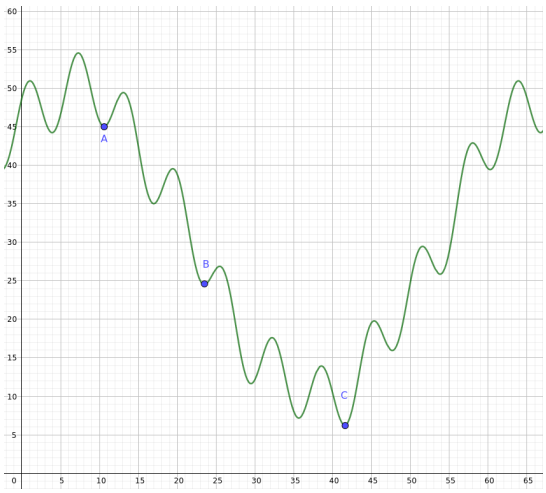

Abbildung 1: Minima einer Funktion mit einem Parameter

, ,
.

Wie man an Abbildung 5 gut sehen kann, handelt es sich bei A um ein Minimum, allerdings ist dieses Minimum global betrachtet nicht besonders gut, da B und C deutlich niedriger liegen. Wenn man allerdings von A nach einen Minimum sucht, sind die Funktionswerte der Umgebung höher, weshalb wieder A als Minimum gefunden werden würde. Die Konsequenz davon ist, dass das Netz die Minima bei B und C nicht finden würde.

Um dieses Problem zu beheben, lässt man das Netzwerk "erkunden". Dabei wird mit einer Wahrscheinlichkeit ϵ eine zufällige Aktion gewählt. Durch diesen Ansatz kann das Netz auch bessere als bereits gefundene Minima finden. Natürlich besteht auch die Möglichkeit, dass die Zufällige Aktion schlechter ist. In diesem Fall würde das Netz lernen, dass diese Aktion nicht gut war und bei der bekannten Aktion bleiben.

Das Ziel besteht an dieser Stelle darin, einen guten Wert für ϵ zu finden. Ist ϵ zu groß, werden vom Netz gelernte Eigenschaften zu wenig berücksichtigt, ist ϵ zu klein, findet zu wenig Erkundung statt und das bereits gelernte nicht optimale Verhalten wird nicht verbessert.

2.2 Markov-Entscheidungsprozess

Der Markov-Entscheidungsprozess ist eine Möglichkeit, Entscheidungsprobleme mathematisch zu formulieren: $MEP = (S, A, t, r)$ mit

- S: Menge aller Zustände, die die Umgebung annehmen kann
- A: Menge aller Aktionen, die der Agent durchführen kann
- t: Übergangsfunktion, die abhängig von einem Zustand und der in diesem Zustand gewählten Aktion den Folgezustand ausgibt ($t(s, a) = s'$)
- r: Wertfunktion, die den Wert einer Aktion in einem gegebenen Zustand ausgibt ($r(s, a) = v$)

Die Lösung des MEP ist eine Funktion $\pi(s) = a$, die zu einem Zustand die optimale in diesem Zustand wählbare Aktion ausgibt. Diese Funktion wird als **Policy** bezeichnet.

Im praktischen Teil dieser Studienarbeit wurde das Spiel "TicTacToe" betrachtet. Dieses Spiel kann als $MEP = (S, A, t, r)$ formuliert werden:

- Der Zustand des Spielfelds kann codiert werden, indem die Felder von links oben nach rechts unten zu einer Reihe zusammengefasst werden. Jedes leere Feld wird durch eine 0, jedes eigene Feld durch eine 1 und jedes gegnerische Feld durch eine 2 dargestellt. Zusätzlich gibt es einen ungültigen Zustand (z.B. 333333333). Entsprechend gilt also: $S = \{000000000, 000000001, \ldots, 012021000, \ldots, 333333333\}$
- Der Spieler kann jedes Feld als Aktion auswälen: $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- Die Übergangsfunktion der Zustände wird entsprechend der Spielregeln erstellt, z.B.: $t(000000000, 0) = 100000000$
- Die Wertfunktion kann für Zustände und Aktionen, die zu Endzuständen führen (Ein Spieler gewinnt, alle Felder sind belegt, ungültiger Zustand) angegeben werden, z.B.: $v(120102000, 6) = 42$, wenn Werte für diese Zustände definiert werden.

S, A und t sind entsprechend vollständig von der Umgebung vorgegeben Das Problem besteht darin, dass r nur teilweise durch die Umgebung vorgegeben ist. Wenn r vollständig bekannt wäre, könnte man das MEP lösen und die Funktion $\pi(s)$ aufstellen.

Im Fall von TicTacToe ist es möglich, die Funktion r vollständig aufzustellen und das MEP zu lösen. Bei komplexeren Umgebungen (z.B. Schach) wäre dieses Vorgehen praktisch nicht anwendbar. Aus diesem Grund kann man ein neuronales Netz verwenden, welches die Funktion $\pi(s)$ lernen soll.

2.3 Q-Learning

Um die Policy-Funktion $\pi(s)$ zu lernen, muss ein neuronales Netz mit Datensätzen trainiert werden, die folgende Informationen enthalten:

- s: Zustand
- a: gewählte Aktion
- v: Wert der Aktion in diesem Zustand

Ein Trainingsdatensatz kann also als Tupel (s, a, v) dargestellt werden. Dadurch tritt allerdings das im vorherigen Abschnitt beschriebene Problem auf, dass v nur für Endzustände bekannt ist. Dieses Problem lässt sich dadurch lösen, dass jedes Tupel aus Zustand und Aktion einen **Q-Wert** erhält, der mit Hilfe der Bellman-Gleichung berechnet wird: $Q(s, a) = r + \gamma(max(Q(s', a'))$ Anschaulich gesprochen wird der Q-Wert also berechnet, indem der direkte Wert (0, wenn es sich nicht um einen Endzustand handelt) mit dem Q-Wert der besten Aktion im Folgezustand addiert wird.

Da der Zug, der zum Sieg geführt hat, einen besseren Wert haben sollte als vorhergehende Züge, wird der Q-Wert der besten Aktion im Folgezustand mit einem Reduktionsfaktor γ multipliziert. Dadurch nähert sich der Q-Wert umso stärker 0 je größer die Distanz zum Endzustand ist.

Bei einfachen Umgebungen können die auf diese Weise berechneten Werte als Funktion r in den MEP eingesetzt werden. Darauf hin kann der MEP gelöst und die Funktion $\pi(s)$ berechnet werden. Im Folgenden wird dieser Ansatz anhand einer einfachen Umgebung erläutert:

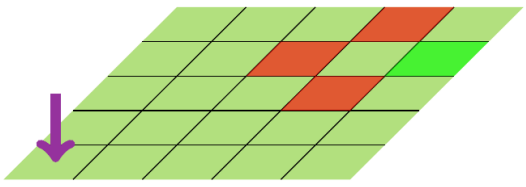

Abbildung 2: Einfache Umgebung

Der Spieler startet auf dem unteren linken Feld und kann sich in 4 Richtungen bewegen (rechts, links, nach vorn, nach hinten). Wenn der Spieler in das hervorgehobene grüne Feld läuft, hat er gewonnen. Läuft er in ein rotes Feld, hat er verloren. Im nächsten Schritt werden die Werte dieser Endfelder definiert. Gewinnen ist als +10, Verlieren als −10 definiert:

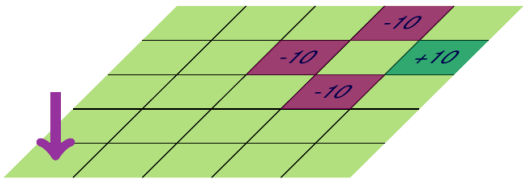

Abbildung 3: Einfache Umgebung mit Werten auf Endfeldern

Darauf hin wird die Bellman-Gleichung angewendet. Um zu ungerade Zahlen zu verhindern und dadurch die Darstellung zu verbessern wurde in diesem Beispiel nicht mit einem Faktor multipliziert, sondern der Wert des besten Folgefeldes verwendet und davon 1 subtrahiert:

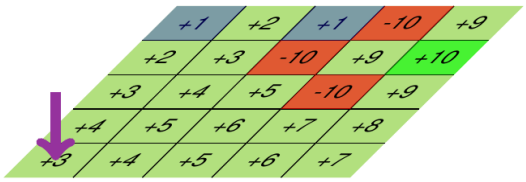

Abbildung 4: Einfache Umgebung mit Werten

Nun ist einfach zu erkennen, dass $\{r, r, r, r, v, v, v\}$ eine optimale Folge von Aktionen ist, um in der gegebenen Umgebung mit einer möglichst geringen Anzahl von Aktion zu gewinnen.

Dieses Vorgehen ist bei komplexen Umgebungen allerdings nicht geeignet, da der Aufwand für die Berechnung aller Q-Werte bei komplexen Umgebungen sehr groß ist. Um Q-Learning auch für komplexe Umgebungen verwenden zu können, kann ein neuronales Netz verwendet werden, das die Policy-Funktion $\pi(s)$ lernt. Mit der oben beschriebenen Herangehensweise kann das Netz also ein Spiel spielen. Die dabei gesammelten Tupel werden zu einem Array hinzugefügt. Der Wert für v wird dabei jeweils auf 0 gesetzt, wenn die Aktion keine direkte Belohnung erzielt hat. Am Ende eines Spiels ist der Wert der letzten Aktion bekannt. Nun können die Werte für r berechnet werden. Darauf hin wird das Netz mit diesem Datensatz trainiert.

Dieser Prozess wird so lange wiederholt bis das Netz entweder die gewünschte Qualität hat (z.B. 90% der Spiele gewinnen) oder das Netz trotz vieler Wiederholungen keine weitere Verbesserung mehr erreicht.

2.4 Deep-Q-Learning

Deep-Q-Learning ist ein auf Q-Learning aufbauender Algorithmus, der im Gegensatz zum klassichen Q-Learning ein komplexes neuronales Netz mit mehreren Schichten verwendet.

Da die im Verlauf des Spiels gespeicherten Erfahrungen stark korreliert sind (Der aktuelle Zustand hängt sehr stark von den vorher gewählten Aktionen ab), kann es sinnvoll sein, diese Korrelation der Daten aufzulösen. Dafür werden mehrere Spiele gespielt und die Erfahrungen in einen Puffer geschrieben. Danach werden die einzelnen Q-Werte berechnet und das Netz trainiert. Beim Training werden die Datensätze allerdings in zufälliger Reihenfolge verwendet, wodurch die Datensätze nicht mehr korreliert sind.

Eine optionale Erweiterung für DQN besteht darin, ein zweites Netz zu trainieren, das den Wert eines Zustands abschätzt. Dieser Ansatz verbindet also die Vorteile des Lernens einer Aktion und des Lernes der Werte von Zuständen.

Wenn das DQN lernen soll, Videospiele zu spielen, schaltet man häufig einige convolutional-Layers vor die eigentlichen Schichten des DQN, um eine möglichst gute Vorverarbeitung der Bilddaten des Videospiels zu erreichen. Außerdem werden häufig mehrere Bilder gleichzeitig zum Trainieren verwendet, um die Spieldynamik darzustellen: Bei dem Spiel Pong ist es z.B. wichtig zu wissen, in welchem Winkel sich der Ball zwischen den Schlägern bewegt. Diese Information lässt sich allerdings nicht aus einem einzelnen Bild gewinnen. Sind hingegen zwei Bilder vorhanden, kann die Information des Winkels verwendet werden.

2.5 A3C

Der Algorithmus A3C "Asynchronous Advantage Actor-Critic" verbindet drei grundlegende Ansätze:

- **Asynchronous**:
  Es gibt mehrere Agenten, von denen jeder über eine eigene von den anderen Agenten unabhängige Kopie der Umgebung verfügt. Zusätzlich verfügt jeder Agent über eine eigene Version des neuronalen Netzes. Mehrere Agenten werden parallel mit ihrer jeweiligen Umgebung trainiert. Nach einer bestimmten Anzahl von Trainingsschritten werden die gelernten Gewichte mit einem globalen Netz synchronisiert. Dazu gibt es generell mehrere Ansätze. Der einfachste Ansatz ist das Synchronisieren durch eine "laufenden Durchschnitt". Dabei werden die jeweiligen Gewichte mit einem Faktor multipliziert und addiert. Die Summe der Faktoren ergibt 1. Beim ersten Synchronisationsschritt wird das Gewicht des globalen Netzes mit 0 und das Gewicht des Subnetzes mit 1 multipliziert, das globale Netz wird also praktisch überschrieben. Beim zweiten Synchronisationsschritt wird das Gewicht des globalen Netzes mit $\frac{1}{2}$ und das Gewicht des Subnetzes mit $\frac{1}{2}$ multipliziert. Beim n-ten Synchronisationsschritt wird das Gewicht des globalen Netzes mit $\frac{n-1}{n}$ und das Gewicht des Subnetzes mit $\frac{1}{n}$ multipliziert.
- **Actor-Critic**:
  Das Netz besteht aus einem Actor- und einem Critic-Teil. Der Actor-Teil ist für die zu wählende Aktion zuständig, der Critic-Teil für die Abschätzung des Werts eines Zustandes.
- **Advantage**:
  Die Critic gibt an, wie gut ein Zustand ist ($c(s)$), der Actor gibt an, wie gut eine Aktion in einem Zustand ($a(s, a)$). Der Vorteil durch das Wählen einer Aktion kann also wie Folgt berechnet werden: $A(s, a) = a(s, a) - v(s)$. Wenn die Umgebung erlaubt, keine Aktion durchzuführen, kann der Advantage dazu genutzt werden, um zu entscheiden, ob überhaupt eine Aktion durchgeführt wird: Ist $A(s, a) \leq 0$, wird keine Aktion durchgeführt.

A3C bietet einige Vorteile gegenüber den vorher beschriebenen Ansätzen:

- Geschwindigkeit:
  Durch das parallele Trainieren mit mehreren Umgebungen kann zur Verfügung stehende Rechenleistung besser genutzt werden. A3C könnte z.B. auf einem Cluster laufen.
- Keine Korrelation:
  Da die verschiedenen Agents parallel trainieren, besteht keine Korrelation zwischen den Trainingsdaten, da diese zwar innerhalb eines Agents korreliert sind, diese Korrelation allerdings durch das Mergen verloren geht.
- Keine Aktion wählen:
  Es ist möglich, zu erkennen, ob das Wählen einer Aktion die Situation verbessert oder nicht. Wenn die Umgebung das zulässt, kann also auch keine Aktion gewählt werden

3 UMSETZUNG

3.1 Allgemeines

Um zu veranschaulichen, wie künstliche neuronale Netze trainiert werden, wurde im Rahmen dieser Studienarbeit ein Programm erstellt, mit dem das Spiel "TicTacToe" gegen ein künstliches neuronales Netz gespielt werden kann. Außerdem kann das Netz Trainiert werden, wobei einige Parameter des Trainings angepasst werden. Es werden Statistiken zur Qualität des Netzes ausgegeben.

Da sich der Fokus im Verlauf der Studienarbeit von der Implementierung eines hochwertigen Algorithmus und dem Erreichen einer möglichst hohen Gewinnquote zur anschaulichen Darstellung des Trainings verschoben hat, wurde ein DQN-ähnlicher Algorithmus implementiert.

Beim Training spielt ein Agent, der mit dem Netz verbunden ist, gegen einen Computergegner wählbarer Schwierigkeit. Dabei wird aufgezeichnet, wie viele Spiele das Netz gewonnen oder verloren hat, wie oft es unentschieden gespielt hat und wie viele Spiele es durch einen ungültigen Zug verloren hat. Diese Statistikdaten werden auch global erhoben. Zusätzlich wird die Anzahl insgesamt gespielter Spiele abgespeichert. Das Netz und die Statistikdaten können gespeichert und wieder geladen werden.

Während eines Trainingsspiels wird gespeichert, welche Aktion in welchem Zustand welchen Wert erreicht hat. Am Ende des Spiels werden die Werte wie in den vorherigen Abschnitte beschrieben berechnet. Anschließend wird das Netz mit diesen gespeicherten Datensätzen trainiert.

Zusätzlich hat der Benutzer auch die Möglichkeit, selbst gegen das Netz zu spielen. Dabei kann er wählen, ob das Netz aus diesem Spiel lernen soll oder nicht.

3.2 Programm

Das Programm wurde in der Programmiersprache Python [9] mit der Bibliothek PySide2 [10], einem Wrapper für QT [11] unter Python, erstellt. Das künstliche neuronale Netz wurde mit Keras [12] und dem TensorFlow [13] Backend erstellt.

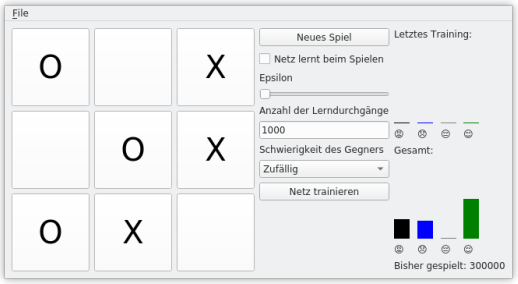

Abbildung 5: Oberfläche des Spiels

Im linken Bereich befindet sich das Spielfeld. Dort kann der Benutzer gegen das Netz spielen, wenn es nicht gerade trainiert wird. In der Menübar befindet sich das Menü "Datei". Über dieses Menü kann ein neues Netz erstellt, das aktuelle Netz gespeichert oder ein gespeichertes Netz geladen werden. Wird ein neues Netz erstellt, wenn das aktuelle Netz nicht gespeichert ist, erscheint direkt ein Speichern-Dialog, mit dem das aktuelle Netz gespeichert werden kann.

Im mittleren Bereich befindet sich die Steuerung. Dort kann ein neues Spiel gestartet werden. Es gib eine Checkbox zur Auswahl, ob das Netz beim Spielen lernen soll oder nicht. Zusätzlich kann mit dem Slider "Epsilon" angepasst werden, mit welcher Wahrscheinlichkeit der Agent die Ausgabe des Netzes ignoriert und einen zufälligen Zug wählt. Die Textbox "Anzahl der Lerndurchgänge" enthält die Zahl der Spiele, die das Netzt bei einem Klick auf den Button "Netz trainieren" spielen soll. Mit dem DropDown-Menü "Schwierigkeit des Gegners" kann eine von vier Schwierigkeitsstufen des Gegners, gegen den das Netz trainiert wird, gewählt werden:

- **Zufällig**:
  Es wird ein zufälliger, möglicherweise auch ungültiger Zug gewählt
- **Einfach**:
  Es wird ein zufälliger, garantiert gültiger Zug gewählt
- **Mittel**:
  Wenn der Gegner im nächsten Zug gewinnen kann, wird dies verhindert. Sonst wird ein zufälliger gültiger Zug gewählt
- **Schwer**:
  Wenn es möglich ist, in diesem Zug zu gewinnen, wird dieser Zug genommen. Kann der Gegner im nächsten Zug gewinnen, wird dies verhindert. Sonst wird ein zufälliger gültiger Zug gewählt

Im rechten Bereich des Fensters werden Statistiken angezeigt. Dabei zeigt der obere Teil die Statistiken des letzten Trainings an (ein Spiel, bei dem das Netzwerk lernt, zählt auch als Training). Der untere Teil zeigt die globale Statistik des Netzes seit dessen Erstellung an.

Der linke Balken gibt den Anteil der Spiele an, die das Netz durch einen ungültigen Zug verloren hat. Der Balken daneben zeigt den Anteil der Spiele, die das Netz verloren hat. Der Balken daneben zeigt, wie viele Spiele unentschieden ausgegangen sind. Der rechte Balken gibt den Anteil der Spiele, die das Netz gewonnen hat, an.

Darunter steht die Anzahl der von dem Netz seit der Erstellung gespielten Spiele.

3.3 Erfahrungen

Im Verlauf des Projekts gab es einige Dinge, durch deren Anpassung deutliche Verbesserungen erreicht wurden. Bei den in diesem Abschnitt verwendeten Abbildungen sind rote Schichten Eingabeschichten, cyanfarbene Schichten innere Schichten und blaue Schichten Ausgabeschichten.

3.3.1 Gesamtkomplexität des Netzes. Ein relativ komplexes Netz (8 Schichten, jeweils ca. 300 Knoten) lernte schnell, ungültige Züge zu vermeiden, wenn als Eingabevektor die Information übergeben wurde, ob ein Feld leer (0) oder belegt (1) ist. Wurde zum Eingabevektor zusätzlich die Information hinzugefügt, welcher Spieler auf ein Feld gesetzt hatte, war das Netz nicht mehr in der Lage, ungültige Züge zu vermeiden. Durch das Verringern der Komplexität (Reduktion auf 3 Schichten, jeweils ca. 50 Knoten) konnten deutlich bessere Ergebnisse erzielt werden.

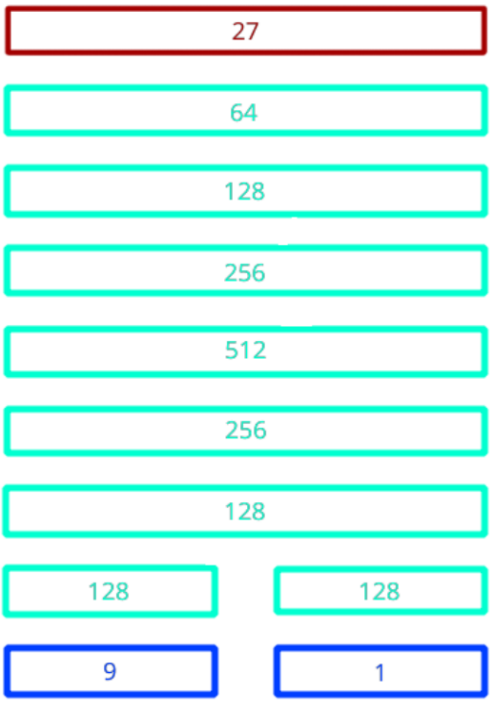

Abbildung 6: Komplexes Netz

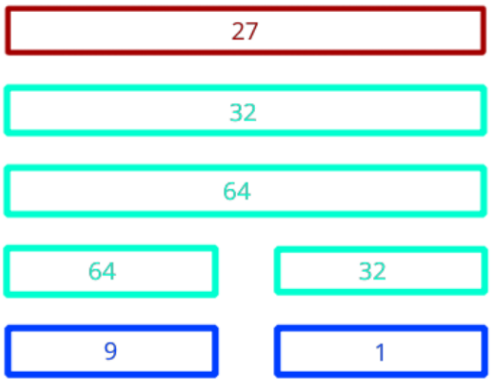

Abbildung 7: Einfaches

3.3.2 Eingaben aufteilen und in verschiedenen Schichten verwenden. Durch das Aufteilen des Eingabevektors in zwei Teilvektoren, die an verschiedenen Stellen in das Netz gegeben werden, wurden die Ergebnisse deutlich verbessert. Dabei wurde der aus 27 Elementen bestehende Eingabevektor (9 für leere Felder, 9 für Spieler 1, 9 für Spieler 2) in einen Eingabevektor für die leeren Felder und einen Eingabevektor für die Spieler aufgeteilt:

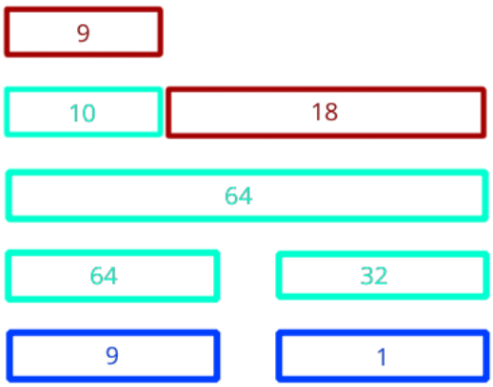

Abbildung 8: Netz mit aufteteilten Eingabevektor

3.3.3 A3C-Ansatz verwerfen. Aufgrund des Fokus auf die Implementierung des A3C-Algorithmus am Anfang der Studienarbeit, basierte das Layout des Netzes auch auf dem in A3C vorgeschlagenen Layout. Es gab also eine Ausgabe für den Wert und eine Ausgabe für die zu wählende Aktion. Da der Wert allerdigs nicht verwendet wurde, konnte er entfernt werden, was zu einer Verbesserung des Netzes führte.

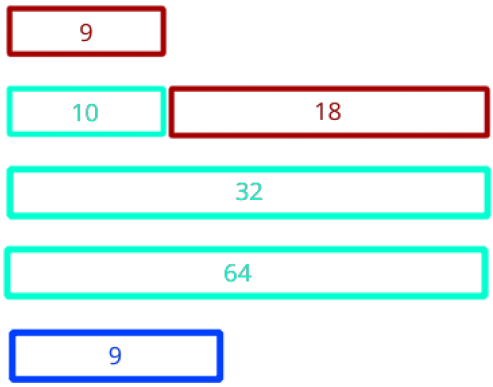

Abbildung 9: Netz nach Entfernen des Value-Teil

4 AUSBLICK

4.0.1 Bessere Gegenspieler. In der aktuellen Version kann zwischen vier verschiedenen Gegnern gewählt werden

4.0.2 Netz gegen sich selbst trainieren.

5 QUELLEN

[1] Asynchronous Methods for Deep Reinforcement Learning (https://arxiv.org/pdf/1602.01783.pdf)
[2] An Introduction toMarkov Decision Processes (https://www.cs.rice.edu/~vardi/dag01/givan1.pdf)
[3] Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0)
[4] Simple Reinforcement Learning with Tensorflow: Part 2 — Policy-based Agents (https://medium.com/@awjuliani/super-simple-reinforcement-learning-tutorial-part-2-ded33892c724)
[5] Simple Reinforcement Learning with Tensorflow: Part 3 — Model-Based RL (https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-3-model-based-rl-9a6fe0cce99)
[6] Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond (https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df)
[7] Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf)
[8] Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C) (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2)
[9] Python
[10] PySide2
[11] QT
[12] Keras
[13] TensorFlow