Hochschule Ravensburg-Weingarten

Fakultät für Elektrotechnik

Bachelorarbeit

im Studiengang Angewandte Informatik - Schwerpunkt Robotik und Automatisierungstechnik

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	Deep Reinforcement Learning from Self-Play in Imperfect-Information Games
Autor:	David Joos david.joos@hs-weingarten.de MatNr. 24316.
Version vom:	21. Mai 2018
1. Betreuer:	Prof. Dr. Ertel
2. Betreuer:	Prof. Dr. Weiss

Zusammenfassung

In dieser Arbeit wird die Umsetzung des Artikels von Heinrich and Silver (2016) und der daraus resultierende „Neural Fictitous Self-Play“ Algorithmus thematisiert. Es wird dabei auf die Implementierung neuronaler Netze, Reinforcement, Supervised Learning und den „Neural Fititious Self-Play“ Algorithmus eingangen. Die notwendigen Grundlagen zur Arbeit mit diesen Technologien wird ebenfalls behandelt und in den jeweiligen Unterkapiteln beschrieben. Für die Umsetzung wurde Python, Keras - welches auf TensorFlow aufsetzt und Numpy verwendet. Es wurde gezeigt, dass mit der Implementierung dieser Arbeit ähnliche Ergebnisse wie in der Vorgabe erreicht werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis **4**

Listingverzeichnis **4**

1 Einleitung **5**
1.1 Aufgabenstellung 5
1.2 Abgrenzung 6

2 Spieltheorie **7**
2.1 Nullsummenspiele 7
2.2 Spiele mit unvollständiger Information 7
2.3 Nash-Gleichgewicht 7
2.4 Extensiv und Normalform 8
2.4.1 Normalform 8
2.4.2 Extensivform 9
2.5 Leduc Hold'em 11
2.6 Umsetzung 13

3 Neuronale Netze **15**
3.1 Aufbau eines künstlichen neuronalen Netzes 15
3.2 Aktivierungsfunktionen 16
3.2.1 Rectifier 16
3.2.2 Softmax 16
3.3 Backpropagation 17

4 Grundlagen Maschinelles Lernen **18**
4.1 Überwachtes Lernen 19
4.2 Lernen durch Verstärkung 21
4.2.1 Q-Learning 22

5 Neural Fictitious Self-Play **24**
5.1 Grundlagen 24
5.2 Funktionsweise des NFSP 25

6 Vorgehensweise **27**

7 Praktische Umsetzung **27**
7.1 Zusammenfassung 27
7.2 NFSP Agent 28
7.2.1 Initialisierung 28
7.2.2 Lernen des Agenten 30
7.2.3 Spielen des Agenten 32
7.3 Main 33

8 Fazit **35**

9 Ausblick **37**

Literaturverzeichnis **38**

Anhang **40**

Eidesstattliche Erklärung **40**

Abbildungsverzeichnis

1 Spieltheorie: Normalform 8
2 Spieltheorie: Extensivform-sequentiell 9
3 Spieltheorie: Extensivform. Dass die Otto-Knoten eingekreist sind bedeutet, dass Otto nicht weiß in welchem dieser Knoten er sich befindet. Er hat also kein Wissen darüber welche Strategie Anna wählt. Damit lässt sich dann auch das Spiel „Schere, Stein, Papier“ wieder sinnvoll darstellen. 10
4 Aktivierungsfunktion: ReLu 17
5 Vollständig vernetztes neuronales Netz. Die Kreise stellen die Neuronen dar, die Kanten die Gewichte zwischen den Neuronen. 17
6 NFSP: Algorithmus (vgl. Heinrich and Silver, 2016, S. 4) 26
7 Ergebnis: Kurve bis 100k Iterationen 36
8 Ergebnis: Vorgabe, Kurve bis 400k Iterationen 36

Listingverzeichnis

1 Grundlegende Interaktion mit der Spieleumgebung LHE.14 2 Initialisierungsfunktion des Best-Response Netzwerkes mit Keras Chollet et al. (2015). Dense steht für „fully-connected“. „mse“ steht für „Mean Squared Error“ und entspricht der vorgebenen Kostenfunktion für das Best-Response Netzwerk (vgl. Heinrich and Silver, 2016).29 3 Initialisierungsfunktion des Average-Response Netzwerkes mit Keras Chollet et al. (2015). „categorical crossentropy“ wird in 4 beschrieben und entspricht der vorgebenen Kostenfunktion für das Average-Response Netzwerk.29 4 Initialisierung der Netzwerke und setzen der Gewichte des Target Netzwerks.30 5 Aktualisieren der Q-Werte des DQN31 6 Average-Response Netzwerk: Klassifizieren der bisherigen Aktionen des Best-Response Netzwerkes.32 7 Spiel-Funktion des NFSP Agenten.33 8 Live-Training der NFSP Agenten.34

1 Einleitung

Künstliche Intelligenz wird immer populärer, wir begegnen ihr zunehmend in alltäglichen Situationen, sei es der sprachgesteuerte Assistent oder das autonom agierende Fahrzeug. Auch medienwirksame Ereignisse, wie der Sieg von Alpha Go über den weltbesten menschlichen Spieler Lee Sedol Silver and Hassabis (2016)) rücken die entwicklung künstlicher Intelligenz immer mehr in den Fokus der Öffentlichkeit. Die möglichen Einsatzgebiete sind vielschichtig und für vielerlei Branchen attraktiv, sei es die Finanzbranche oder in der Pflege. Doch so nützlich diese Technologie erscheinen mag, so kontrovers wird sie auch diskutiert - Schlagzeilen über Gesichtserkennung in Fußballstadien, die unbescholtene Bürger belastet, als auch die KI unterstützte Verwertung personenbezogener Daten durch Konzerne wie Google oder Facebook werfen Fragen über den sinnvollen Gebrauch dieser Technologie auf. Der Vormarsch der KI liegt vor allem in der intensiven Forschung der letzten Jahre begründet. In der jüngsten Vergangenheit konnte besonders die Forschung auf dem Gebiet der künstlichen neuronalen Netze auf sich aufmerksam machen. Viele dieser innovativen Technologien werden zuerst im Umfeld einer Spieleumgebung erforscht, siehe AlphaGo von Deepmind. Nachdem KI in diversen Spielen wie Schach, Go und den Atari Arcade Spielen den Menschen mittlerweile um Längen schlägt, gibt es jedoch noch eine Domäne die einige große Herausforderungen birgt: Spiele in denen die KI nicht über die vollständige Information des Spielzustandes verfügt. Poker beispielsweise ist eines dieser Spiele. Diese Arbeit soll sich ebenfalls dem Lernen in Spielen mit unvollständiger Information widmen. Dabei wird modernste Technologie verwendet und ein lauffähiges Programm entwickelt, das eine einfache Form von Poker lernt, in dem es gegen sich selbst spielt.

1.1 Aufgabenstellung

Diese Arbeit beschäftigt sich mit dem Artikel von Heinrich and Silver (2016). Es sollen die notwendigen Technologien zur Implementierung des beschriebenen „Neural Fictitious Self-Play“ Algorithmus aufgezeigt und erläutert werden. Dabei sollen folgende Punkte Beachtung finden:

- Erläuterung der notwendigen Technologien, die zur Implementierung eines lauffähigen Programms notwendig sind.
- Entwicklung der Spieleumgebung Leduc Hold’em, welche auch im Artikel beschrieben wird.
- Entwicklung eines Agenten, der neuronale Netze zur Funktionsapproximation nutzt und anhand der Erläuterung in Heinrich and Silver (2016) lernt.

Das Ziel der Arbeit ist es, die Ergebnisse der Autoren zu reproduzieren. Da der Artikel[1] nicht auf die direkte Implementierung eingeht, sondern nur eine grobe Struktur vorgibt, soll ein Weg gefunden werden, trotz fehlender Informationen ein lauffähiges Programm darauf aufbauend zu entwickeln.

1.2 Abgrenzung

Die Bachelorarbeit konzentriert sich auf die technische Umsetzung des NFSP Algorithmus und erläutert alle notwendigen Technologien um ein lauffähiges Programm entwickeln zu können. Hierfür werden gängige Technologien und Frameworks wie Keras, TensorFlow, Python und Numpy verwendet. Diese Technologien werden als gesetzt angenommen und somit ist der Vergleich zu ähnlichen Technologien anderer Anbieter nicht Teil dieser Arbeit. Des weiteren werden sämtliche Parameter wie z.B. die Lernrate direkt dem Artikel Heinrich and Silver (2016) entnommen. Wird von den Vorgaben im Artikel abgewichen, wird dies gekennzeichnet und dient zur einfacheren Implementierung. Es ist nicht Teil dieser Arbeit, die Vorgabe[2] in irgendeiner Weise zu optimieren oder in Frage zu stellen.

Hinweise zum Umgang:
In dieser Arbeit werden diverse Wörter in Anlehnung an ihre englische Schreibweise verwendet. Dies dient zur einfacheren Zurechtfindung in zugehörigen Arbeiten, die hauptsächlich in Englisch verfasst sind. Zur weiteren Vereinfachung der Sprache sollen nachfolgend einige Punkte definiert werden:

- Von **Deep Learning** kann gesprochen werden, wenn ein neuronales Netz mit einer Anzahl n an verborgenen Schichten genutzt wird. Dabei gilt: $n > 0, \forall n \in \mathbb{N}$
- Als Agent wird ein Programm bezeichnet, das zu eigenständigem handeln in der Lage ist. In diesem Falle ein Programm, das durch die Implementierung maschineller Lernverfahren gelernte oder randomisierte Entscheidungen trifft.
- Lernen durch Verstärkung wird im folgenden als **Reinforcement Learning** betitelt. Des weiteren wird an diversen Stellen die Abkürzung **RL** verwendet.
- Überwachtes Lernen wird im folgenden als **Supervised Learning** betitelt. Des weiteren wird an diversen Stellen die Abkürzung **SL** verwendet.
- Die programmatische Implementierung erfolgte in Python. Wenn von **Code** die Rede ist, wird auf Programmcode verfasst in Python referenziert.

[1] Heinrich and Silver (2016)
[2] (ebd. Heinrich and Silver, 2016)

2 Spieltheorie

Häufig werden Spiele zur Entwicklung und Erforschung maschineller Lernverfahren herangezogen, dies hat einen einfachen Grund: In einem Realwelt-Szenario ist die Menge an möglichen Zuständen häufig zu groß und das herbeiführen der Zustände zu aufwändig. In einer vereinfachten simulierten Umgebung kann sowohl schneller als auch besser trainiert werden, da sich der Zustandsraum schneller explorieren und künstlich begrenzen lässt. Des weiteren bieten Spiele viele Eigenschaften, die auf Realwelt-Szenarien übertragen werden können. Diese Arbeit wird sich ebenfalls mit einer künstlichen Intelligenz im Umfeld eines Spieles beschäftigen. In diesem Kapitel sollen die Grundlagen vermittelt und dem Leser gängige Begriffe der Spieltheorie näher gebracht werden.

2.1 Nullsummenspiele

Ein Spiel ist dann ein Nullsummenspiel wenn die Summe der Gewinne und Verluste aller Spieler genau Null enstspricht. In einem zwei Spieler Szenario ist dies dann gegeben wenn der Gewinn des einen, der Verlust des anderen ist (vgl. Holler and Illing, 2006).

$$Gewinn(Spieler1) + Gewinn(Spieler2) = 0$$

2.2 Spiele mit unvollständiger Information

Es kann zwischen Spielen mit unvollständiger und vollständiger Information unterschieden werden. Schach und Dame sind beispielsweise Vertreter der Spiele mit vollständiger Information, das bedeutet im Klartext, dass alle Spieler zu jedem Zeitpunkt vollständige Information zum Spielstand haben. Häufig ist dies beispielsweise in Kartenspielen nicht gegeben. In dieser Arbeit wird im weiteren Verlauf auf eine Pokervariante referenziert und Poker gehört naturgemäß zu den Spielen mit unvollständiger Information. Zu Beginn eines jeden Spieles werden private Karten ausgegeben, die nur der jeweilige Spieler kennt. Das bedeutet dass die Spieler zwar ihre eigenen Karten und die öffentlichen Karten auf dem Tisch kennen, jedoch keine Information zu den privaten Karten des Gegners haben. Der zu erwartende Gewinn ist somit nicht deterministisch. Ein Spieler kann also in verschiedenen Runden mit ein und derselben Karte verlieren oder aber auch gewinnen, abhängig von der Gegnerkarte. Wenn keine vollständige Information gegeben ist, muss der Spieler Wahrscheinlichkeiten abschätzen und diese in seine Entscheidungen miteinbeziehen[3].

2.3 Nash-Gleichgewicht

Ein Nash-Gleichgewicht, benannt nach Nash et al. (1950), besteht wenn es für keinen Spieler lohnenswert ist von seiner Strategie abzuweichen. Angenommen alle Spieler han-

[3](ebd. Holler and Illing, 2006)

deln rational, was bedeutet, dass jeder versuchen wird seinen Gewinn zu maximieren, dann wird jeder Spieler versuchen die best möglichste Antwort auf die Gegnerstrategie zu finden. Wenn dies erreicht ist, dann besteht ein Nashgleichgewicht und es ist für keinen Spieler mehr lohnenswert von seiner Strategie abzuweichen.

2.4 Extensiv und Normalform

Extensiv und Normalform sind Begriffe aus der Spieltheorie. Der Unterschied der beiden Formen liegt in ihrer Darstellung. Die Normalform wird gemeinhin als Tabelle oder Matrix dargestellt, die Extensivform meist als Baumdiagramm. Anhand eines Zwei-Spieler Nullsummenspiels soll deutlich gemacht werden, welche Unterschiede die zwei Darstellungen ausmachen. Als beispielhaftes Spiel wurde „Schere, Stein, Papier“ gewählt.

2.4.1 Normalform

Das Spiel „Schere, Stein, Papier“ wird von zwei Spielern *Anna*, *Otto* gespielt. Jeder der Spieler kann aus den Aktionen $\{Schere,\ Stein,\ Papier\}$ auswählen. Die gewählte Aktion wird dabei **Strategie** genannt. Wenn *Anna* gewinnt, dann verliert *Otto*. Zu je-

		Otto: Schere	Otto: Stein	Otto: Papier
Anna	Schere	0 / 0	0 / 1	1 / 0
	Stein	1 / 0	0 / 0	0 / 1
	Papier	0 / 1	1 / 0	0 / 0

Abbildung 1: Spieltheorie: Normalform

der Strategie eines Spielers sind die zu erwartenden Gewinne eingetragen, abhängig von der gewählten Gegnerstrategie. Spielt *Anna* also Strategie „Stein“ und *Otto* Strategie „Schere“ kann direkt aus der Tabelle abgeleitet werden:

$$G_{Anna}(Anna_{Stein},\ Otto_{Schere}) = 1$$

Heinrich and Silver (2016) definieren jede der Aktionen als eine **reine Strategie**. Jede reine Strategie Δ_p^i ist Teil der Strategiemenge Δ_b^i. Abhängig von der Belohnungsfunktion R und der gewählten Gegnerstrategie wird die Belohnungsfunktion für jeden Spieler i wie folgt definiert:

$$R^i(\Delta_p^i,\ \Delta_p^{-i})$$

Jede reine Strategie kann also als Spielplan betrachtet werden, der für jede Situation, mit der ein Spieler eventuell konfrontiert wird, die Belohnung bestimmt. Wird ein Spiel in mehreren Iteration gespielt, dann wird jeder Spieler eine **Mischstrategie** wählen, was bedeutet, dass vor jeder Iteration, ein Spielplan gewählt wird und dieser für diese eine Iteration gespielt wird. Die Wahl, welche reine Strategie dabei zum tragen kommt, hängt von der **gemischten Strategie** Π ab, welche eine Wahrscheinlichkeitsverteilung über alle reinen Strategien ist. Die Menge aller gemischten Strategien ist dabei Δ^i für Spieler i und somit ist $\Pi \in \times_{i \in \mathcal{N}} \Delta^i$ mit $\mathcal{N} = \{1, \ ..., \ n\}$ als Menge aller Spieler. Damit kann die erwartete Belohnung weiter formalisiert werden:

$$R^i : \times_{i \in \mathcal{N}} \Delta^i \to \mathbb{R}$$

Im weiteren Verlauf dieser Arbeit werden, wie von Heinrich and Silver (2016) ebenfalls verwendet, gemischte Strategien und reine Strategien mit großen griechischen Buchstaben beschrieben. Während Verhaltensstrategien, erläutert im nächsten Unterkapitel, kleine griechische Buchstaben bekommen.

2.4.2 Extensivform

Im Gegensatz zur Normalform lassen sich mit der Extensivform sequentielle und nicht deterministische Gewinnerwartungen darstellen(vgl. Holler and Illing, 2006). Angewendet auf das eben verwendete „Schere, Stein, Papier“ Spiel, ergibt sich somit eine Baumdarstellung 2. In dieser Darstellung lässt sich erkennen, dass sich ebenso komplexe

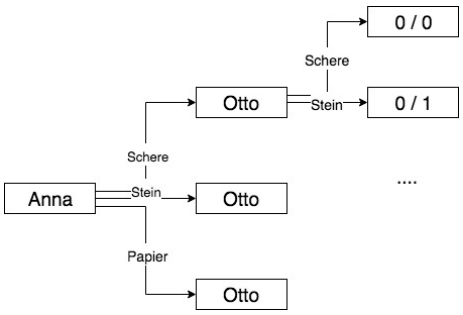


Abbildung 2: Spieltheorie: Extensivform-sequentiell

sequentielle Abläufe, als auch nicht deterministische Gewinne darstellen lassen. Beispielsweise ist es denkbar, dass Otto mit „Schere“ gegen Anna mit „Stein“ gewinnt, unter der Bedingung, dass Otto zuvor einmal mit „Schere“ gegen „Stein“ verloren hat. Allerdings lässt sich auch leicht erkennen, dass „Schere, Stein, Papier“ nur dann Sinn macht, wenn es keinem sequentiellen Ablauf folgt, da Otto sonst einen Vorteil hätte, wüsste er bereits, welche Strategie Anna spielt. Deswegen lässt sich die Extensivform dahin gehend erweitern, dass kein sequentieller Ablauf vonstatten geht - dies sei aber nur der Vollständigkeit halber erwähnt 3.

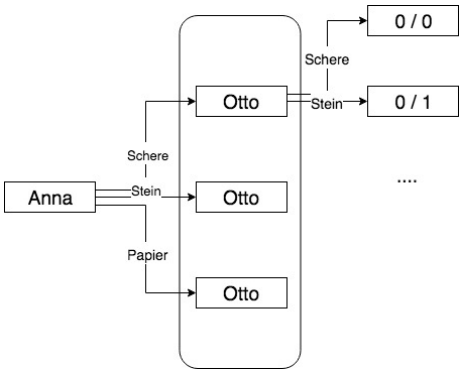


Abbildung 3: Spieltheorie: Extensivform. Dass die Otto-Knoten eingekreist sind bedeutet, dass Otto nicht weiß in welchem dieser Knoten er sich befindet. Er hat also kein Wissen darüber welche Strategie Anna wählt. Damit lässt sich dann auch das Spiel „Schere, Stein, Papier“ wieder sinnvoll darstellen.

Die Voraussetzungen für die Extensivform Darstellung, wie sie von Heinrich and Silver (2016) verwendet wird, sind folgende Komponenten:

- $\mathcal{N} = \{1,\ ...,\ n\}$ beschreibt die Menge der Spieler
- $\mathcal{S}$ ist die Menge der Zustände und somit die Menge aller Zustände im Spielbaum.
- $\forall s \in \mathcal{S}$ wird eine Menge von Aktionen $\mathcal{A}(s)$ definiert, die jedem Spieler im Knoten s zur Verfügung stehen.
- Die Spielerfunktion $P : \mathcal{S} \to \mathcal{N}$, welche jeden Zustand auf einen Spieler abbildet, sprich welcher Spieler in diesem Knoten am Zug ist.
- Für jeden Spieler i gibt es eine zugehörige Menge an Informationen $\mathcal{U}^i$ und eine Informationsfunktion $I^i : \mathcal{S} \to \mathcal{U}^i$. Diese bestimmt welche der Zustände für den Spieler nicht zu unterscheiden sind, wenn er einen Zustand auf den gleichen Informationszustand $u \in \mathcal{U}^i$ abbildet. Wenn von einem Spiel mit unvollständiger Information ausgegangen wird, dann bedeutet das, dass Spieler i zu zwei Zuständen s_{k1} und s_{k2} den selben u^i_k zur Verfügung hat, obwohl sich die Zustände unterscheiden. Am Beispiel von Poker würde dies Bedeuten, dass der Zustand des Spiels gleich ist bis auf die private Karte des Gegners von Spieler i.

In dieser Arbeit wird davon ausgegangen, dass jeder Spieler vollständige Information über alle Aktionen und Zustände hat, welche zum Informationszustand u^i_k geführt haben. Jeder Spieler kennt also die Sequenz: $u^i_1, a^i_1, u^i_2, a^i_2, ..., u^i_k$. Ist dies gegeben, ist es ein Spiel mit **perfect Recall**.

- Mit der Belohnungsfunktion $R : \mathcal{S} \to \mathbb{R}^n$ werden schließlich die Endzustände auf einen Vektor abgebildet, der die Belohnung für jeden Spieler repräsentiert.

Jeder Spieler wählt nun eine Strategie. Ähnlich der Mischstrategie in der Normalform wird diese Strategie verschiedene Zustände auf verschiedene Aktionen abbilden. Der Unterschied dabei ist, dass die Normalform nach dem einmaligen ausspielen der Aktion davon ausgeht, dass das Spiel terminiert und eine Belohnung ausgeschüttet wird. In der Extensivform kann ein Spiel jedoch mehrere Aktionen der jeweiligen Spieler beinhalten bevor das Spiel terminiert. Damit definieren wir die Verhaltensstrategie, die jeden Zustand im Spielbaum auf eine Aktion abbildet: $\pi^i(u) \in \Delta(\mathcal{A}(u)), \forall u \in \mathcal{U}^i, \Delta_b^i$ als die Menge aller Verhaltensstrategien von Spieler i. Diese bestimmt die Wahrscheinlichkeitsverteilung über alle möglichen Aktionen für einen gegebenen Zustand. Ein **Strategieprofil** $\pi = \{\pi^1, ..., \pi^n\}$ beinhaltet alle Strategien aller Spieler und π^{-i} bezieht sich damit auf alle Strategien in π bis auf π^i. Dieser Argumentation folgend wird die erwartete Belohnung von Spieler i wie folgt definiert: $R^i(\pi)$ unter der Bedingung, dass alle Spieler dem Strategieprofil π folgen[4].

Da davon ausgegangen werden kann, dass jeder Spieler gewinnen will, wird jeder Spieler versuchen eine Strategie zu spielen, die seinen erwarteten Gewinn maximiert - damit kann die **best response** (*engl. Beste Antwort*) definiert werden. Spieler i wird wenn möglich in jedem Zustand eine Aktion wählen, die unter der Bedingung, dass Spieler $-i$ die Strategie π^{-i} spielt R maximiert : $b^i(\pi^{-i}) = \arg\max_{\pi^i \in \Delta_b^i} R^i(\pi^i, \pi^{-i})$. Ein Nash-Gleichgewicht ist erreicht, wenn jeder Spieler die best response zur Strategie des Gegners spielt. Formalisiert bedeutet das: $\pi^i \in b^i(\pi^{-i})$ für alle $i \in \mathcal{N}$[5] .

2.5 Leduc Hold'em

Heinrich and Silver (2016) beziehen sich in ihrem Artikel auf eine Pokervariante, die „Leduc Hold'em" (**LHE**) genannt wird. Da LHE eine Sondervariante des weit verbreiteten Texas Hold'em ist, soll vorerst auf einige Begrifflichkeiten und Regeln des Pokerspieles eingegangen werden, um anschließend die Unterschiede zwischen LHE und Texas Hold'em zu erläutern.

Begrifflichkeiten:

- **Dealer**: Der Dealer, ist jener Spieler am Tisch, der die Karten ausgibt. Der Dealer wechselt nach jeder Runde. In der Regel wird der Spieler zur linken des Dealers der neue Dealer.
- **Smallblind**: Der Spieler zur linken des Dealers muss den Smallblind erbringen. Da in der Regel um Geld gespielt wird, ist der Smallblind ein gewisser Geldbetrag, den dieser Spieler setzen muss.

[4](ebd. Heinrich and Silver, 2016)
[5](ebd. Heinrich and Silver, 2016)

- **Bigblind**: Der Spieler zur linken des Smallblinds ist in der Pflicht den Bigblind zu erbringen. Der Betrag des Bigblinds ist in der Regel das doppelte des Smallblinds. Wie auch der Smallblind, ist der Bigblind verpflichtend für den jeweiligen Spieler.

Texas Hold'em wird in Runden gespielt. Zu Anfang der ersten Runde mischt der Dealer und gibt die privaten Karten an die jeweiligen Spieler aus. Im herkömmlichen Texas Hold'em erhält jeder Spieler am Tisch zwei verdeckte Karten. Sind alle Karten ausgeteilt, beginnt die erste Wettrunde. Der Small- und Bigblind sind verpflichtend, die Spieler jedoch, welche nicht von den Blinds betroffen sind, können an dieser Stelle bereits aussteigen, falls ihre privaten Karten keinen Gewinn versprechen. In der Wettrunde hat jeder Spieler die Möglichkeit eine von drei Aktionen zu wählen:

- **Fold**: Folden bedeutet aus dem aktuellen Spiel auszusteigen. Der Spieler gibt seine Karten zurück an den Dealer und ist für dieses Spiel raus. Falls der Spieler zu diesem Zeitpunkt bereits einen Betrag gesetzt hat, ist dieser verloren.

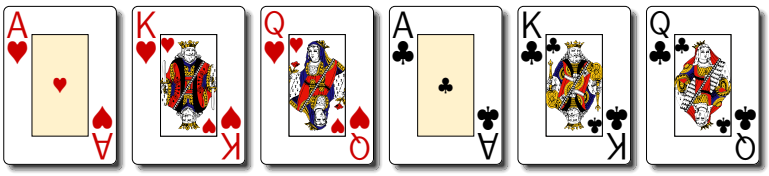
- **Call, Check**: Das bedeutet, ein Spieler möchte nicht erhöhen aber im Spiel bleiben. Ist in der aktuellen Wettrunde bereits erhöht worden, dann muss der Spieler mindestens den Einsatz bringen, der bereits vom erhöhenden Spieler gebracht wurde (Call). Ist in der Wettrunde nicht erhöht worden, muss der Spieler nichts tun (Check).

- **Raise**: Raise steht für erhöhen. Wenn ein Spieler gute Karten hat oder auch nur bluffed, dann erhöht er um einen Betrag. Dieser Betrag liegt im Ermessen des Erhöhenden, die Schwierigkeit dabei ist es nicht zu viele Gegner zu verschrecken, falls der Erhöhende gute Karten hat oder nicht zu wenig zu setzen, falls gebluffed wird.

Eine Wettrunde endet, sobald alle Spieler den selben Betrag gesetzt oder gefoldet haben. Jeder Spieler kann pro Runde nur einmal erhöhen. Erhöht Spieler 1 woraufhin Spieler 2 nochmals erhöht, kann Spieler 1 anschließend nur noch callen oder folden.
Ist die erste Wettrunde abgeschlossen, folgt die Offenlegung der Tischkarten. Dabei werden drei Karten offen auf den Tisch gelegt. Jeder Spieler kann nun aus den zwei privaten und den drei offenen Karten eine Hand bilden. Ob eine Hand gewinnt oder nicht hängt dabei vom Rang der Hand ab. Dabei werden gewisse Kombinationen aus den Karten höher gewertet als andere. Beispielsweise ist ein Pärchen niedriger als ein Drilling und so weiter. Daraufhin wird die zweite Wettrunde gespielt, die Blinds entfallen nach der ersten Runde und somit kann jeder Spieler frei entscheiden, ob er weiter spielt oder das Spiel mit Fold verlässt. Nachfolgend werden zwei weitere Runden gespielt, wobei jedes mal eine weitere Karte offen auf den Tisch gelegt wird. Zu guter letzt liegen fünf offene Karten auf dem Tisch. Im Falle, dass zwei oder mehr Spieler übrig sind, entscheidet die Höhe der Hand, wer den Pot gewinnt. Der Pot ist dabei die

Menge an Geld, die während der Wettrunden gesetzt wurde.

LHE ist eine limitierte Variante von Texas Hold'em. Bei LHE sind die Einsatzhöhen fest vorgeschrieben, will ein Spieler erhöhen, kann nur um den Betrag des Bigblinds erhöht werden. Davon ausgegangen, dass der Bigblind dem Wert „1“ entspricht, können sämtliche Spieler pro Runde jeweils auch nur um 1 erhöhen. Heinrich and Silver (2016) haben LHE noch weiter eingeschränkt: Texas Hold'em wird in der Regel mit einem 52 Karten Blatt gespielt wird, LHE in deren Artikel jedoch auf ein 6 Karten Blatt reduziert. Das führt auch dazu, dass die Anzahl der Runden beschränkt werden muss. In dieser Arbeit wird die Anzahl der Runden auf 2 beschränkt. Außerdem bekommt jeder Spieler nur ein verdeckte Karte zu Beginn. In der ersten Wettrunde wird nach den Regeln von Texas Hold'em gespielt. Ist diese Runde abgeschlossen, wird auch nur eine Karte offen auf den Tisch gelegt. Nachdem die zweite Wettrunde abgeschlossen ist und kein Spieler gefoldet hat, wird der Gewinner festgelegt. Gegeben sei ein 6 Karten Blatt:



Der Spieler, der mit seiner verdeckten Karte den gleichen Rang (beispielsweise Ass) wie die offen liegende Karte hat, ist der Gewinner. Da von jedem Rang nur zwei Exemplare im Spiel sind (drei Duplikate), ist es ausgeschlossen, dass beide Spieler die offene Karte treffen. Trifft keiner der beiden Spieler die offenliegende Karte, gewinnt derjenige der die höherwertige Karte auf der Hand hat. Haben beide Spieler den gleichen Rang auf der Hand, so ist es Unentschieden und jeder Spieler bekommt seinen Einsatz zurück.

Eine letzte Besonderheit ist, dass das Spiel **Heads-Up** gespielt wird. Das heißt, dass nur zwei Spieler im Spiel sind und somit der Dealer den Smallblind erbringen muss und der jeweilig andere den Bigblind.

2.6 Umsetzung

Das Spiel wird als „Blackbox“ implementiert. Das bedeutet, die Spieler haben keine Kenntnis über die Spielregeln oder Belohnungsfunktion. Grundsätzlich kann der Spieler nur über zwei Funktionen mit dem Spiel interagieren:

```
import leduc.env as env

# Initialisiere Spieler
spieler = spieler.Spieler()

# Initialisieren der LHE Umgebung
LHE = env.Env()

# Ruecksetzen der Umgebung nach Beendigung eines Spieles
# Hierbei muss der Dealer Index mituebergeben werden
LHE.reset(spieler.getIndex)

# Abgreifen des Spielzustandes als MDP codiert
s, a, r, s_1, t = LHE.get_state(spieler.getIndex)

# Im Spiel einen Zug ausfuehren
LHE.step(action, spieler.getIndex)
```

Listing 1: Grundlegende Interaktion mit der Spieleumgebung LHE.

Frägt ein Spieler den aktuellen Zustandes des Spiels über die Funktion „get_state“ ab, dann muss dieser der Spielerindex übergeben werden, da dies ein Spiel mit unvollständiger Information ist und jeder Spieler nur die Information erhalten darf, die ihm zusteht. Dasselbe gilt auch für die „step“ Funktion. Die Umgebung legt im Hintergrund für jeden Spieler im Spiel einen eigenen State an und verwertet die ausgeführten Aktionen, immer im Hinblick auf den Index des jeweiligen Spielers. Das führt dazu, dass kein Spieler speziell auf den Spielstand achten muss, da die Spielumgebung das im Hintergrund für den jeweiligen Spieler erledigt. Die Spieler interagieren mit der Spielumgebung also lediglich indem sie den Spielstatus abfragen, aufgrund dessen dann eine Aktion wählen und diese wieder an das Spiel übergeben.

LHE ist ein Extensivform Spiel und somit sind im Zustand s_k alle vorherig ausgeführten Aktionen und Zustände $s_1, a_1, s_2, a_2, ..., s_k$ ein Entscheidungskriterium für den Spieler. Aufgrund dessen enthält der Zustand, der über die „get_state“-Funktion abgefragt werden kann, eine Wetthistorie. Die Wetthistorie beschreibt darin die getätigten Aktionen eines jeden Spielers, zu jeder Runde. Der Zustand kann also als

$$2 \times 2 \times 3 \times 2$$

Tensor - namentlich (*Spieler*, *Runde*, *Raises*, *Aktionen*) dargestellt werden, da es ein Zwei-Spieler-Spiel, mit maximal zwei Runden, 0 bis 2 **Raise** pro Runde und jeweils zwei möglichen Aktionen ist. Der Aktionsraum besteht allerdings aus drei Aktionen `{ Fold, Call, Raise }`, da **Fold** aber zwingend zum Abbruch des Spiels führt, wird es in der Wetthistorie nicht berücksichtigt. Des weiteren müssen die Karten noch als:

$$2 \times 3$$

Tensor übergeben werden - namentlich ($Runde$, $Karten$) dargestellt. Die zwei Tensoren werden miteinander verkettet und als Array der Länge `{30}` übergeben.

3 Neuronale Netze

Das menschliche Gehirn besteht aus etwa 10 bis 100 Milliarden Nervenzellen, die in ihrer Verschaltung und gegenseitigen Aktivierung unseren Intellekt definieren. Nachdem die Wissenschaft immer sicherer wurde, dass all unsere Fähigkeiten dem Netzwerk aus unseren Nervenzellen entspringt, wurde dann von McCulloch and Pitts (1990) das erste mathematische Modell eines Neurons als zentrales Schaltelement unseres Gehirns vorgestellt. Auf Basis dieses Artikels konnten nun erste künstliche neuronale Netze nachgebildet werden. Heutzutage ist diese Technik vielschichtig einsetzbar und erlaubt es auch nichtlineare komplexe Funktionen zu approximieren[6] siehe Ertel (2009).

Der Aufbau kann sich je nach Art der Aufgabe stark unterscheiden, deswegen beschränkt sich diese Arbeit darauf, nur die notwendigen Module für die Umsetzung des NFSP's zu beschreiben.

3.1 Aufbau eines künstlichen neuronalen Netzes

In einem neuronalen Netz sind die Neuronen in Schichten miteinander verbunden, hier sind verschiedene Verbindungen vorstellbar. In dieser Arbeit soll ein vollständig vernetztes Netz verwendet werden. Wie schon erwähnt, ist das Neuron der Baustoff eines jeden neuronalen Netzes. Ob ein Neuron „aktiv“ wird, sprich ein Signal an die darauf folgendenden Neuron weitergibt, hängt davon ab, ob die vorhergehenden Neuronen aktiviert worden sind oder nicht. Die oberste Schicht, die lediglich das Eingangsignal weiter gibt wird im weiteren Verlauf der Arbeit als **Input Layer** bezeichnet. Die Neuronen des Input Layers fungieren somit als die eingehenden Verbindungen für die folgende Schicht, **Hidden Layer** genannt. Grundsätzlich wird in einem vollständig vernetzten Netz, jedes Neuron einer Schicht, mit jedem Neuron der darauf folgenden bzw. der vorhergehenden Schicht verbunden. Somit kann das Aktivierungspotential des Neurons i als mathematischen Modell dargestellt werden (vgl. Ertel, 2009, S. 269):

$$\sum_{j=1}^{n} w_{ij}\ x_j$$

Wobei $x_1, ..., x_j$ die Werte der vorhergehenden Neuronen darstellen, w_{ij} das Gewicht zwischen Neuron i und Neuron j .

[6] Gleichungssysteme, deren Gleichungen nicht alle linear sind, werden nichtlineare Gleichungssysteme genannt. Für solche Gleichungssysteme gibt es keine allgemeingültigen Lösungstrategien.

Das Aktivierungspotential für sich sagt jedoch nichts darüber aus, ob das Neuron aktiviert wird oder nicht. Deswegen wird eine Aktivierungsfunktion auf das Aktivierungspotential angewendet:

$$x_i = f(\sum_{j=1}^{n} w_{ij}\ x_j) \tag{1}$$

Es gibt eine Vielzahl an Aktivierungsfunktionen und wird je nach Anspruch gewählt. Ein neuronales Netz besteht somit aus dem Input Layer, n Hidden Layern und der letzten Schicht (in dieser Arbeit als **Output Layer** bezeichnet). Das Output Layer folgt den selben Prinzipien wie alle anderen Layer. Des weiteren wird von einer diskreten Zeitskala ausgegangen, das bedeutet, die Informationen pro Eingangssignal werden sequentiell durch das Netz gereicht. Sprich, Neuron i hat zum Zeitpunkt t die Ausgangswerte aller vorgehenden Neuronen des Eingangssignals $Input_t$ zur Verfügung. Die Daten werden also sequentiell abgearbeitet und können nicht parallel verwertet werden.
Ein neuronales Netz funktioniert also in dem sich die Neuronen innerhalb des Netzes gegenseitig aktivieren und somit die Eingangswerte auf die Ausgangswerte abbilden.

3.2 Aktivierungsfunktionen

Den NFSP betreffend werden im späteren Verlauf zwei unterschiedliche Aktivierungsfunktionen verwendet, zum einen die **Rectifier** und zum anderen die **Softmax** Funktion. Beide sollen in diesem Bereich dem Leser vorgestellt werden.

3.2.1 Rectifier

Ein Rectifier (*engl. Korrigieren*), erstmals eingeführt von Hahnloser et al. (2000) korrigiert die Funktionswerte in dem nur die positiven Werte als Teil seiner Ausgabe verwendet werden. Formal ausgedrückt: $f(x) = x^+ = \max(0, x)$ - das heißt für jeden Wert der nicht Teil der positiven Menge von x ist, wird eine 0 ausgegeben, andernfalls x siehe Abbildung 4.

3.2.2 Softmax

Die Softmax Funktion berechnet die Wahrscheinlichkeitsverteilung über alle Neuronen (vgl. Christopher, 2016):

$$F(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}}$$

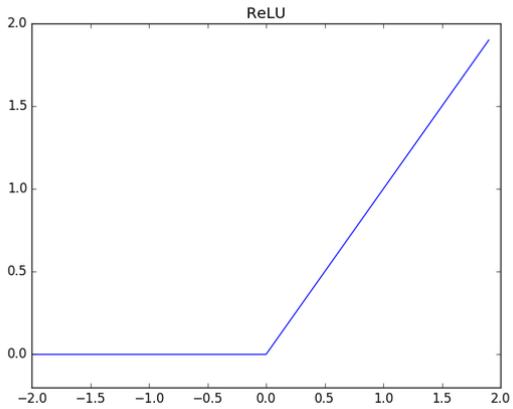


Abbildung 4: Aktivierungsfunktion: ReLu

Dabei gilt $\sum_{i=0}^{k} x_i = 1$ und ist somit eine Wahrscheinlichkeitsverteilung über alle Neuronen von x_1, ..., x_k. Bei Klassifikationsproblem ist dies nützlich, da die Softmax Funktion für jedes Ausgabeneuron eine Wahrscheinlichkeit angibt, mit der dieses Neuron der erwarteten Ausgabe entspricht.

3.3 Backpropagation

Wenn man sich das Netz in Schichten vorstellt, siehe Abbildung 5, und die zugrunde liegende Aktivierung der einzelnen Neuron berücksichtig und die Kanten als Gewichte versteht, lässt sich leicht verstehen wie das Eingangssignal auf die Ausgabe (Output Layer) abgebildet wird. Ein Netz das kein Vorwissen hat, also zufällig initialisiert wur-

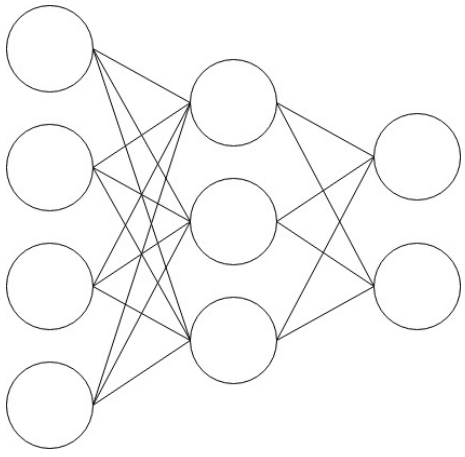


Abbildung 5: Vollständig vernetztes neuronales Netz. Die Kreise stellen die Neuronen dar, die Kanten die Gewichte zwischen den Neuronen.

de (zufällige Gewichte), wird jedes Eingangssignal auf eine zufällige Ausgabe abbilden. Will man dieses Verhalten ändern und das Netz lernen lassen, ist es nötig, den Output des Netzes mit dem erwarteten Ouput[7] zu vergleichen und dementsprechend zu

[7]Dies kann Expertenwissen oder zuvor klassifizierte Daten sein.

berichtigen. Der Vergleich zwischen der Ausgabe des Netzes und den Kontrolldaten kann verschiedene Formen annehmen und wird gemeinhin als Error oder Kostenfunktion C bezeichnet, dazu mehr in Grundlagen Maschinelles Lernen auf S. 18. Der Wert dieser Funktion ist ein Anhaltspunkt dafür, wie gut das Netz die Bewertungsfunktion approximiert hat. Ein Netz das bereits über viele Trainingsbeispiele gelernt hat, sollte einen niedrigen Wert der Kostenfunktion aufweisen. Denn es gilt immer, die Kosten und somit die Kostenfunktion zu minimieren. Da nur die Gewichte des Netzes beeinflussbar sind, gilt es die Gewichte dahingehend zu verändern, dass das Output Layer die Neuronen aktiviert, die es aktivieren soll. Um dies zu erreichen wird der **Backpropagation** Algorithmus verwendet. Dabei werden die Gewichte enstprechend dem negativen Gradienten, der über die Ausgabeneuronen summierten Kostenfunktion

$$E_p(w) = \frac{1}{2} \sum_{k \in Ausgabe} C(t_k^p, x_k^p) \tag{2}$$

für das Trainingsmuster p geändert:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}} \tag{3}$$

Es wird nun E_p durch (2) ersetzt und x_k durch (1), dann kommen darin rekursiv die Ausgaben x_i der nächst tieferen Schicht vor und darin wiederum die nächste und so weiter. Durch wiederholtes anwenden der Kettenregel (siehe Rumelhart et al., 1985) ergibt sich die Backpropagation-Lernregel (vgl. Ertel, 2009, S. 292). Wie aus der Doku hervorgeht, implementiert **Keras** den Backpropagation Algorithmus ohne zutun des Anwenders Chollet et al. (2015) weswegen auf die genauen Details nicht weiter eingegangen wird.

4 Grundlagen Maschinelles Lernen

Wenn wir lernen, sowohl absichtlich als auch nicht, definieren wir uns selbst und unsere Umwelt neu. Eine Veränderung unseres Denkens und/oder Handelns ist eingetreten und führt zu einer anderen Interaktion mit dem was wir wahrnehmen und dem was wir tun. Diese Fähigkeit ist die Grundvoraussetzung dafür, dass Tier und Mensch sich ihrer Umwelt zu gegebener Zeit anpassen und somit überleben können.

Speziell die Fähigkeit zur Assoziation, also zur Verknüpfung verschiedener Sinneseindrücke und die daraus resultierende Verstärkung von neuronalen Verknüpfungen, auch genannt Synapsen, ermöglicht dem Menschen den Lernprozess. Zur Wahrnehmung der Umwelt stehen dem Menschen seine Sinneseindrücke und Emotionen zur Verfügung.

Nachdem die Welt und die eigene Position darin bewertet wurde, führt dies zu einer Handlung. Nach ausgeführter Handlung wird wiederum bewertet, wie sich die eigene Position in Abhängigkeit der Handlung verändert hat. Ist die neue Position erstrebenswerter als die vorherige, bedeutet das, dass die Handlung in einer solchen Umgebung und zu den vorherrschenden Bedingung als sinnvoll erachtet werden kann und somit wiederholt werden will.

Was für die Menschen eine Selbstverständlichkeit darstellt, ist schwer auf Maschinen zu übertragen. Das „Wahrnehmen, Agieren, Bewerten“ Muster, wie im vorigen Absatz erläutert, stellt nicht das volle Spektrum des menschlichen Lernens dar, gehört jedoch immer zum Lernen dazu. Daraus resultierend entstand die Idee des verstärkenden Lernens. Unter die Begrifflichkeit „Lernen durch Verstärkung“ fallen eine Reihe verschiedener Methoden, beruhen jedoch alle auf dem „Wahrnehmen, Agieren, Bewerten“ Muster. Dabei spielt der „Bewerten“ Teil die größte Rolle, bewertet wird aufgrund der Belohnungsfunktion und stets gilt es die Belohnung zu maximieren. Was der Agent wahrnimmt und wie er agieren kann, hängt von der Implementierung ab. Ist die Welt wenig komplex und die Menge an möglichen Aktionen klein, ist es unter Umständen möglich für jeden Zustand eine eindeutige Aktion abzuleiten, meistens jedoch ist die Welt komplexer und die Menge der möglichen Zustände und Aktion sehr groß und die Belohnungsfunktion muss somit approximiert werden. Dafür werden zum Beispiel neuronale Netze genutzt - wie im vorherigen Kapitel beschrieben. Auf der anderen Seite ist es aber auch möglich aus bereits bekanntem zu lernen. Ist das, was man lernen soll schon bekannt, kann der eigene Lernfortschritt anhand korrekt klassifizierter Daten überprüft werden - dies gilt ebenso für maschinelles Lernen. Hier spricht man von überwachtem Lernen. Beide Vorgehensweisen sind Teil dieser Arbeit und werden zur Umsetzung des NFSP benötigt.

Zusammengefasst lassen sich grundlegende Lernverfahren in zwei Klassen unterteilen:

- Überwachtes Lernen - Supervised learning
- Lernen durch Verstärkung - Reinforcement learning

Da im NFSP sowohl überwachtes als auch unüberwachtes Lernen zum Einsatz kommt, sollen dem Leser die Prinzipen beider Verfahren näher gebracht werden.

4.1 Überwachtes Lernen

Überwachtes Lernen dient dazu, aus bestehendem Wissen Gesetzmäßigkeiten abzuleiten und diese auf neue Daten anzuwenden. Das bestehende Wissen kann in diesem Falle entweder Expertenwissen oder gekennzeichnete Daten sein. Dieses Vorgehen heißt überwacht, weil die künstliche Intelligenz auf Daten lernt, die bereits bekannt sind und

die KI ihre eigenen Aussagen kontrollieren und gegebenenfalls korrigieren kann (siehe Mohri et al., 2012)).

Im späteren Verlauf dieser Arbeit wird ein überwachter Klassifizierer implementiert werden. Klassifikation bedeutet, dass ein Eingangsvektor auf eine Menge von Klassen abgebildet wird. Für das bereits eingeführte Spiel LHE bedeutet das, der Spielzustand wird auf eine der drei möglichen Aktionen $\{Fold,\ Call,\ Raise\}$ abgebildet. Die Bewertungsfunktion $F(x)$, welche den Zustand auf die Aktionen abbildet, soll gelernt werden. Der Erfolg wird anhand einer Kostenfunktion gemessen. Da dies ein überwachter Lernvorgang ist, benötigt die Bewertungsfunktion bereits klassifizierte Daten - es werden somit folgende Punkte definiert:

- $\mathcal{S}$ ist die Menge aller Zustände. Für jeden Zustand gilt $s \in \mathcal{S}$.
- $\mathcal{A}(s_t)$ ist die Bewertungsfunktion, die für den Zustand s_t eine Wahrscheinlichkeitsverteilung über die möglichen Aktionen legt. Das heißt $\mathcal{A} : s_t \rightarrow y_{pred}$ mit y_{pred} als die approximierte Wahrscheinlichkeitsverteilung für s_t.
- y_{true} ist die korrekte Wahrscheinlichkeitsverteilung für s_t
- $C(w)$ als Kostenfunktion mit w als die Gewichte des neuronalen Netzes.

Heinrich and Silver (2016) trainieren den Klassifizierer mit folgender Kostenfunktion:

$$\mathcal{L}(\Theta^{\Pi}) = \mathbb{E}_{(s,a)\sim M}[-\log \Pi(s, a|\Theta^{\Pi})]$$

mit Θ^{Π} als die Gewichte des neuronalen Netzes Π, M als der Speicher für (s, a) Tuples die in LHE gespielt wurden, wobei s der Spielzustand und a eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen ist. Für a wird dabei die „1 aus n“ Codierung verwendet, was heißt, dass genau ein Element des Vektors einer 1 entspricht, während alle anderen Elemente 0 entsprechen. Übertragen auf die definierten Punkte:

- y_{pred} entspricht $\Pi(s, a)$ welche $\mathcal{A}(s_t)$ entspricht
- y_{true} sind gesammelt in $\mathbb{E}_{(s,a)\sim M}$
- $C(w)$ entspricht $\mathcal{L}(\Theta)$

Die Kostenfunktion entspricht der Kreuzentropiefunktion und misst wie „nahe“ die Vorhersage y_{pred} der gewollten Verteilung y_{true} entspricht (vgl. De Boer et al., 2005):

$$H(y_{true}, y_{pred}) = -\sum_{s} y_{true}(s) \log y_{pred}(s)$$

Mit dem Backpropagation Algorithmus 3.3 kann nun das Netz auf Basis der Kostenfunktion trainiert werden.

4.2 Lernen durch Verstärkung

Lernen durch Verstärkung ist ebenfalls ein Teilgebiet der künstlichen Intelligenz und dient dazu, die KI lernen zu lassen ohne auf Expertenwissen oder gekennzeichnete Daten zurück zu greifen. Da diese Daten fehlen, muss die KI im laufe des Lernprozesses eine Rückmeldung über den bisherigen Erfolg erhalten. Aufgrund dieses Erfolgs, in diesem Fall Belohnung genannt, können die bisher ausgewählten Aktionen bewertet werden. Anfangs, ohne jegliches Wissen über die Belohnung, die eine Aktion möglicherweise erbringt, wird die KI zufällige Aktionen ausführen. Nachdem dann eine Belohnung rückgemeldet wurde, kann die KI die bisherigen Aktionen bewerten. Ertel (2009) definiert auf S.316 folgende Punkte:

- Die Welt ist die Umgebung. Diese kann unterschiedlich sein, zum Beispiel das Spiel Schach oder unsere „echte“ Welt (Erde, Sonnensystem, Universum). Je nachdem in welcher Welt agiert wird, ändert sich der Zustandsraum unter umständen dramatisch. Das Schachspiel besteht beispielsweise aus allen Spielfiguren, an welcher Stelle sie sich befinden und so weiter und definieren damit den gesamten Zustandsraum während in einem Echtwelt-Szenario ungleich mehr Zustände eintreten können.
- $\mathcal{S}$ ist die Menge aller möglichen Zustände.
- $\mathcal{A}$ die Menge aller möglichen Aktionen.
- δ ist die Übergangsfunktion.
- Eine Strategie $\pi : \mathcal{S} \rightarrow \mathcal{A}$ bildet Zustände auf Aktionen ab.

Ein Agent interargiert mit seiner Umgebung[8] indem er zum Zeitpunkt t den Zustand $s_t \in \mathcal{S}$ der Welt erfasst. In einer Welt mit sehr großem Zustandsraum muss der Zustand eventuell abstrahiert werden, da die Menge an Faktoren, die den Zustand definieren, zu groß ist oder der Zustand nicht vollständig für den Agenten zu erfassen ist. Der Agent führt dann unter der Bedingung s_t eine Aktion $a_t \in \mathcal{A}$ aus. Diese Aktion hat Auswirkungen auf die Welt und den Agenten und führt in den Zustand s_{t+1} - dieser Schritt wird durch die Übergangsfunktion bestimmt $s_{t+1} = \delta(s_t, a_t)$. Der Agent bekommt durch diesen Übergang in den nächsten Zustand eine Rückmeldung r_t und steht im direkten Zusammenhang mit der gewählten Aktion zu einem bestimmten Zustand (s_t, a_t). Die Rückmeldung ist positiv wenn die gewählte Aktion unter der Bedingung eines bestimmten Zustandes zielführend war oder negativ, wenn dies nicht der Fall ist. Ist r_t positiv also $r_t > 0$ wird das gewählte Verhalten verstärkt, ist $r_t < 0$ wird das Verhalten negativ verstärkt. Ist die Rückmeldung $r_t = 0$, dann hat der Agent keine direkte Rückmeldung aus der Welt bekommen.

[8] (ebd. Ertel, 2009)

Angenommen ein Agent soll Schach lernen. Bis zum Ende des Spiels wird eine ganze Reihenfolge von Zügen ausgewählt. Der Agent wird also wahllos irgendwelche Figuren auf dem Feld verschieben und je nach dem wie die Belohnung ausfällt `{ Gewonnen, Verloren }` wird er seine ausgeführten Züge bewerten - jedoch ist nicht klar ersichtlich, welcher Zug zur Niederlage bzw. zum Sieg geführt hat, da bis zum letzten Zug die Rückmeldung $r = 0$ war. Dieses Problem ist als **Credit-Assignment-Problem** bekannt (vgl. Ertel, 2009, S. 316). Dieser Problematik kann entgegen getreten werden indem die **abgeschwächte Belohnung** *(engl. discounted reward)* eingesetzt wird:

$$\mathcal{V}^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Mit $0 \leq \gamma < 1$ als Konstante, die weiter in der Zukunft liegende Rückmeldungen umso mehr abschwächt, desto weiter diese in der Zukunft liegen. $\mathcal{V}^{\pi}$ ist dabei die Belohnung unter der Bedingung, dass jener Agent Strategie π verfolgt, diese Belohnung soll maximiert werden. Dies wird jedoch zum Problem, wenn der Agent kein Modell der Welt zur Verfügung hat und seine Aktionen, die er in der Zukunft tätigen wird, nicht von vornherein bewerten kann. In der Arbeit von Heinrich and Silver (2016) wird das Spiel LHE untersucht und in diesem Szenario hat der Agent kein solches Modell der Welt zur Verfügung. In diesem Falle kann auf einen anderen Ansatz zurück gegriffen werden und wird im folgenden Unterkapitel beschrieben.

4.2.1 Q-Learning

Q-Learning ist ein populärer Ansatz für RL-Probleme. Da der Agent nicht weiß in welchen Zustand seine Aktion a_t in Zustand s_t führt, wird eine Bewertungsfunktion eingeführt $Q(s, a)$ - die Auswahl der Aktion a_t im Zustand s_t wird formal als (vgl. Ertel, 2009, S. 324):

$$\pi^*(s) = \arg\max_a Q(s, a)$$

definiert. Die Funktion $Q(s, a)$ gibt den Wert der erwarteten Belohnung für a_t unter s_t zurück. Da die Q-Werte ohne Vorwissen mit Sicherheit nicht den wahren Q-Werten entsprechen, müssen die Q-Werte angepasst werden. Q-Learning ist ein „online“ Lernalgorithmus, das bedeutet, mit jeder neuen Erfahrung werden die Q-Werte folgendermaßen nachgebessert:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

Heinrich and Silver (2016) nutzen in ihrem Artikel den von Riedmiller (2005) vorgestellten „Fitted Q Iteration“ (im weiteren Verlauf FQI genannt) Algorithmus. Die zugrunde liegende Idee dabei ist, auf online Lernen zu verzichten, da es nur langsam konvergiert. Riedmiller (2005) hat gezeigt, dass es effizienter ist, zuerst eine Menge

an Erfahrungen zu sammeln ohne die Q-Werte zu verändern, diese Erfahrungen werden in Übergangstuples gespeichert (s_t, a_t, s_{t+1}). Diese Form von Übergangstuple kann als **Markov-Decision-Process** bezeichnet werden und wird im weiteren Verlauf als **MDP** bezeichnet. Ein MDP beinhaltet (S, A, T, r, p_0) mit S als Menge der Zustände, A Menge der Aktionen, T das Aktionsmodell, welches mit $T : S \times A \times S \rightarrow [0, 1]$ abbildet, so dasss $T(s, a, s') = p(s'|s, a)$ die Wahrscheinlichkeit ist von Zustand s und Ausführung von Aktion a in den Zustand s' zu gelangen. $r : S \times A \times S \rightarrow \mathbb{R}$ ist dabei die Belohnungsfunktion und ordnet dem Übergang von s zu s' die Belohnung zu. p_0 ist die Startverteilung und gibt zu jedem Zustand an, wie hoch die Wahrscheinlichkeit ist, in diesem Zustand zu starten (siehe Sutton and Barto, 1998). Mit den MDP's aus der Erfahrung, die der FQI ohne die Werte zu aktualisieren, gesammelt hat, können nun SL-Verfahren eingesetzt werden um von diesen gesammelten Erfahrungen zu lernen, hierbei spricht man von **Experience Replay**. Dieser Ansatz wurde durch Mnih et al. (2015) weiter verbessert und wird „Deep Q Network“ genannt. In dieser Arbeit wird der „Deep Q Network“ Algorithmus (im weiteren Verlauf DQN genannt) verwendet. Der DQN ist speziell für Deep Learning entwickelt worden und auch in dieser Arbeit wird der Algorithmus auf ein neuronales Netzwerk angewendet. Die Autoren haben eine Reihe Verbesserungen implementiert, die das Lernen stabiler und schneller machen:

- Wie der FQI verwendet auf der DQN Experience Replay.
- Es wird ein Target Netzwerk genutzt (wird im weiteren Verlauf des Kapitels erläutert).
- **Clipping Rewards**, was bedeutet, dass die maximale und minimale Belohnung auf einen festen Wert begrenzt werden.

Das **Target Netzwerk** ist ein eigenes neuronales Netz, initialisiert mit den selben Gewichten als das DQN Netzwerk selbst $\Theta^{DQN} \rightarrow \Theta^{target}$. Wird kein Target Netzwerk eingesetzt, gilt:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))]$$

Die Autoren Mnih et al. (2015) haben jedoch aufgezeigt, dass das DQN-Netzwerk stabiler lernt, wenn die Vorhersage der erwarteten Belohngen des Zustandes s_{t+1} durch das Target Netzwerk getätigt wird. Und die Lernregel wie folgt abgeändert wird:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a[r_{t+1} + \gamma \max_{a_{t+1}} Q_{target}(s_{t+1}, a_{t+1}) - Q(s_t, a_t))]$$

Das Target Netzwerk wird mit einer Verzögerung nach dem Schema $\Theta^{DQN} \rightarrow \Theta^{target}$ aktualisert. Wieviel später das Target Netzwerk aktualisiert werden muss oder sollte ist nicht genau definiert und muss je nach Anspruch anders gewählt werden. Heinrich and Silver (2016) aktualsieren das Target Netzwerk alle 300 Iterationen, wobei eine Iteration einer Aktualisierung des DQN Netzes entspricht.

5 Neural Fictitious Self-Play

5.1 Grundlagen

Fictitious Play (*engl. Fiktives Spiel*), eingeführt von Brown (1951), ist ein populäres Model der Spieletheorie zum lernen in Spielen. Fictitious Play bedeutet, dass die Spieler eines Spieles iterativ gegeneinander spielen. Bei jeder Iteration versuchen die Spieler, die (in 2.4.2 eingeführte) best response auf die Strategie des Gegners zu finden. Brown hat gezeigt, dass die Mischstrategien der Spieler in gewissen Spielen (z.B. Zwei-Spieler-Nullsummenspiel) gegen ein Nash-Gleichgewicht konvergieren. Trotz seiner Popularität fand Ficititous Play nur spärlichen Einsatz in großen Anwendungen, was seiner Abhängigkeit von der Normalform geschuldet ist, denn Realwelt-Szenarien lassen sich meist nur durch die Extensivform beschreiben. Des weiteren beruht Fictitious Play auf der Idee, alle $s \in \mathcal{S}$ zu durchlaufen um die Konvergenz zu erreichen. Die Autoren Leslie and Collins (2006) führen mit **generalised weakened ficititous play** einen Algorithmus ein, der die best response nur approximiert und somit nicht den vollen Spielbaum durchlaufen muss. Dies ist besonders in sehr großen Zustandsräumen, die nicht mit dynamischer Programmierung rekursive durchlaufen werden können, unabdingbar. Das macht es vor allem passend für maschinelles Lernen. Heinrich et al. (2015) stellt **Full-Width Extensive-Form Fictitious Play (XFP)** vor welcher es ermöglicht, Verhaltensstrategien (Extensivform) der Agenten zu aktualisieren.
Ein wichtiger Punkt dabei ist die **Realisierungswahrscheinlichkeit** Von Stengel (1996) die unter der Annahme, dass es ein Spiel mit perfect Recall (siehe 2.4.2) ist, die Realisierungswahrscheinlichkeit eines jeden Zustandes wie folgt definiert: $x_{\pi^i}(s_t^i) = \prod_{k=1}^{t-1} \pi^i(s_k^i, a_k^i)$. Die Realisierungswahrscheinlichkeit fungiert als Bindeglied zwischen Extensiv und Normalform und führt, wenn man der Argumentation von (ebd. Von Stengel, 1996) und Kuhn (2016) folgt, dazu, dass zur Normalform Mischstrategie: $\hat{\sigma} = \lambda_1\hat{\pi_1} + \lambda_2\hat{\pi_2}$ das Extensivform Equivalent als:

$$\sigma(s,a) \propto \lambda_1 x_{\pi_1}(s)\pi_1(s,a) + \lambda_2 x_{\pi_2}(s)\pi_2(s,a) \quad \forall s,a \tag{4}$$

dargestellt werden kann. Die Gleichung 4 beschreibt einen Weg, Erfahrungen der Agenten zu sammeln ohne rekursiv über alle Spielzustände zu gehen. (vlg. Heinrich and Silver, 2016, S. 3). Heinrich et al. (2015) stellt mit **Fictitious Self-Play**(FSP) einen Algorithmus vor, der mit Experience Replay und maschinellen Lernverfahren arbeitet. 4 beschreibt eine Form, in der FSP Erfahrungen sammeln kann. Genauer gesagt, die FSP Agenten generieren Daten im Spiel gegen sich selbst, speichern die Erfahrung als MDP $(s_t, a_t, r_{t+1}, s_{t+1})$ in einem Speicher M_{RL}, welcher für Reinforcement Learning vorgesehen ist. Die Daten in M_{RL} beinhalten das Verhalten des Gegners, da s_{t+1} nur dann

erreicht werden kann, wenn der Gegenspieler ebenfalls agiert hat. Die Erfahrungen, des eigenen Verhalten (s_t, a_t) wird in M_{SL} gespeichert, passend für einen Supervised Learning Ansatz.

Eine approximierte Lösung der MDP's in M_{RL} stellt somit eine approximierte best response dar, während die Daten in M_{SL} die durschnittliche Strategie des Agenten darstellt (vgl. Heinrich et al., 2015).

5.2 Funktionsweise des NFSP

NFSP kombiniert FSP mit neuronalen Netzen. Im folgenden soll die Funktionsweise detailiert erläutert werden, doch vorab werden zuerst einige Punkte definiert (vgl Heinrich and Silver, 2016, S.3):

- Das **Best-Response-Netzwerk**(BR-Netzwerk) approximiert aus den Daten in M_{RL} die best response, durch **off-policiy**[9] Reinforcement Learning (DQN).
- Das **Average-Response-Netzwerk**(AR-Netzwerk) approximiert die durschnittliche Strategie aus M_{SL} durch überwachtes Klassifizieren.
- $\beta^i(\pi^{-i})$ ist die best response Strategie des Spielers i unter der Bedingung, dass Spieler $-i$ die Strategie π^{-i} verfolgt und wird durch das BR-Netzwerk beschrieben.
- π^i ist die durschnittliche Strategie des Spielers i und wird durch das AR-Netzwerk beschrieben.

Im Algorithmus, siehe Abbildung 6, wird jeder Spieler durch einen separaten NFSP Agenten gesteuert. Ein jeder NFSP Agent interagiert mit seinen Mitspielern und lernt dabei vom Spiel gegeneinander. Die Erfahrungen, welche die Agenten sammeln wird in zwei Speichern abgelegt (M_{RL}, M_{SL}). Der Agent trainiert das BR-Netzwerk (DQN 4.2.1), welches auf Basis des M_{RL} Q-Werte lernt, die wiederrum die best response repräsentieren.
Es wird ein weiteres neuronales Netz, AR-Netz, trainiert, welches den Durchschnitt der bisher getätigten best responses aus M_{SL} durch Klassifizierung approximiert: $\pi = \Pi$.

Währrend die Agenten spielen, wählen diese ihre Aktionen aus einem Mix der zwei Strategien β^i und π^i aus (vgl. Heinrich and Silver, 2016):

$$W\ddot{a}hle\ Strategie\ \sigma = \begin{cases} \epsilon - greedy(Q), & mit\ Wahrscheinlichkeit\ \eta \\ \Pi, & mit\ Wahrscheinlichkeit\ 1-\eta \end{cases}$$

[9] *Off-policy* Lernen heißt, dass der Agent keiner speziellen Strategie folgt, da er im nächsten Schritt wiederrum der Strategie folgt, die den größten erwarteten Gewinn verspricht. Q-Learning ist ein off-policy Lernverfahren.

Initialize game Γ and execute an agent via RUNAGENT for each player in the game
function RUNAGENT(Γ)
Initialize replay memories $\mathcal{M}_{RL}$ (circular buffer) and $\mathcal{M}_{SL}$ (reservoir)
Initialize average-policy network $\Pi(s, a \mid \theta^{\Pi})$ with random parameters θ^{Π}
Initialize action-value network $Q(s, a \mid \theta^{Q})$ with random parameters θ^{Q}
Initialize target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
Initialize anticipatory parameter η
for each episode **do**
Set policy $\sigma \leftarrow \begin{cases} \epsilon\text{-greedy}\,(Q), & \text{with probability } \eta \\ \Pi, & \text{with probability } 1-\eta \end{cases}$
Observe initial information state s_1 and reward r_1
for $t = 1, T$ **do**
Sample action a_t from policy σ
Execute action a_t in game and observe reward r_{t+1} and next information state s_{t+1}
Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in reinforcement learning memory $\mathcal{M}_{RL}$
if agent follows best response policy $\sigma = \epsilon$-greedy (Q) **then**
Store behaviour tuple (s_t, a_t) in supervised learning memory $\mathcal{M}_{SL}$
end if
Update θ^{Π} with stochastic gradient descent on loss
$\mathcal{L}(\theta^{\Pi}) = \mathbb{E}_{(s,a)\sim\mathcal{M}_{SL}}\left[-\log \Pi(s, a \mid \theta^{\Pi})\right]$
Update θ^{Q} with stochastic gradient descent on loss
$\mathcal{L}\left(\theta^{Q}\right) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{M}_{RL}}\left[\left(r + \max_{a'} Q(s', a' \mid \theta^{Q'}) - Q(s, a \mid \theta^{Q})\right)^2\right]$
Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
end for
end for
end function

Abbildung 6: NFSP: Algorithmus (vgl. Heinrich and Silver, 2016, S. 4)

Vor einem jeden Spiel, legt sich ein Agent also auf eine der zwei Strategien fest und bleibt bei dieser bis das Spiel terminiert ist. Grundsätzlich könnte jeder Spieler die Strategie π spielen, daraufhin berechnet jeder Spieler die best response $\beta^i(\pi^{-i})$ gegen die Gegnerstategie π^{-i} - doch dabei würde kein Spieler sein eigenes best response Verhalten lernen. Deswegen wird die Strategie gewechselt, so dass die durchschnittliche best response gelernt werden kann und die Gegner dennoch in der Proportion $\sigma \equiv (1-\eta)\hat{\pi} + \eta\hat{\beta}$ gegen die durchschnittliche Strategie dieses Spielers spielen. η ist dabei der **Anticipatory Paramater** (siehe Heinrich and Silver, 2016) oder (vgl. Shamma and Arslan, 2005).

6 Vorgehensweise

Der Artikel von Heinrich and Silver (2016) ist eine aktuelle Veröffentlichung und es gibt keine Referenzarbeiten auf die man sich beziehen könnte. Die Vorgehensweise ist also direkt aus dem Artikel abzuleiten, wobei es schwierig wird zu validieren wie exakt die Vorgaben umgesetzt wurden. Das Projekt wurde in drei Module unterteilt:

- LHE Spielumgebung: Da das Spiel losgelöst von den Agenten funktionieren soll, wird es abgekoppelt von den Agenten entwickelt. Dabei wird die Schnittstelle zur Interaktion mit dem Spiel dem bekannten **OpenAI Gym** von Brockman et al. (2016) nachempfunden.
- Der NFSP Agent: Der Agent enthält alle notwendigen Technologien wie die neuronalen Netze und eine „Spiel"-Funktion, in der er auf Basis eines s_t auf eine a_t abbildet und diese zurück gibt.
- Die Main Funktion: Hier sollen alle Spieler und die Spieleumgebung initialisiert werden. Hier werden grundlegende Parameter definert wie z.B. wer der Dealer ist etc. Außerdem wird in dieser Funktion jede Runde des Spieles gestartet und die Anzahl der Iterationen festgelegt.

Sämtlicher Programmcode wird in Python geschrieben und es werden gängige Technologien wie TensorFlow, Keras, Python und numpy genutzt. Es empfiehlt sich gängige „Best-Practice" wie **Git** und **Lean Developement** zu nutzen. Dadruch soll redundante Arbeit vermieden und die Code-Qualtität konstant hoch gehalten werden.

Wenn alle Module sinngemäß implementiert wurden, soll der gesamte Prozess rekapituliert, die gewonnenen Daten analysiert und die Ergebnisse validiert werden. Die Daten sollen anhand der Vorgabe von Heinrich and Silver (2016) verglichen werden.

7 Praktische Umsetzung

Hinweis: Die Entwicklung der Spielumgebung „Leduc Hold'em" wird bereits in 2 behandelt. Da LHE „straight forward" implementiert wurde, wird nicht näher auf die Besonderheiten des Code's eingegangen. Möchte der interessierte Leser einen Einblick darüber erhalten wie LHE im Detail umgesetzt wurde, wird auf den Anhang verwiesen.

7.1 Zusammenfassung

In diesem Kapitel wird aufgezeigt wie die zur praktischen Umsetzung des Artikels Heinrich and Silver (2016) notwendigen neuronalen Netze initialisiert, trainiert und genutzt werden. Zur Implementierung dieser wird Keras verwendet, das Aufgrund seiner

Syntax einen einfachen Einstieg in das Thema maschinelles Lernen mit neuronalen Netzen bietet. In einigen Abschnitten wird auf das Objekt „np“ referenziert, damit ist *numpy* gemeint und wird zu Beginn jeder File mit „import numpy as np“ importiert. Auf einige Parameter wird kein Bezug genommen, da sie von den Autoren[10] in ihrem Artikel festgelegt werden und im Projekt in der „Config.ini“-File zu finden sind. Der interessierte Leser findet den vollständigen Code im Anhang. Außerdem wird auf die Interaktion zwischen den Agenten eingegangen, wie diese mit der Spielumgebung LHE interagieren und wie der Ablauf eines Spieles vonstattengeht. Die Implementierung hält sich nahe am Pseudocode der Autoren[11], muss jedoch abgewandelt werden, da im Pseudocode nicht auf die Generierung von Spieldaten eingegangen wird.

7.2 NFSP Agent

7.2.1 Initialisierung

Der NFSP Agent soll über **3** neuronale Netze verfügen:

- Das Best-Response Netzwerk und ein Target Netzwerk zum lernen, welches mit den selben Gewichten initialisiert wird, wie das Best-Response Netzwerk selbst.
- Das Average-Response Netzwerk

Heinrich and Silver (2016) testeten die NFSP Agenten mit nur einem Hidden Layer, welches 64 verdeckte Neuronen enthält. Das Input Layer soll einen Vektor mit 30 Elementen entgegen nehmen, denn der Zustand des Spieles beinhaltet 30 Elemente, die sowohl die Wetthistorie als auch die Karten beschreiben. Der Zustand soll auf drei Aktionen abgebildet werden, welche wiederrum von drei Ausgabeneuronen repräsentiert werden (siehe 2). Als Aktivierungsfunktion für das Output Layer wird die Identität von $f(x) = x$ (im Code „linear“) gewählt, da dies ein Nullsummenspiel ist und die Belohnung sowohl negativ als auch positiv sein kann. Auf **Clipping Reward** 4.2.1 wird verzichtet, da dazu keinerlei Information aus dem Artikel[12] zu entnehmen ist. Die Aktivierungsfunktion „relu“ wurde bereits in 3 vorgestellt und wird für die Hidden Layer verwendet.

[10] Heinrich and Silver (2016)
[11] (ebd.)
[12] Heinrich and Silver (2016)

```
def _build_best_response_model(self):

        input_ = Input(shape=self.s_dim, name='input')

        hidden = Dense(self.n_hidden, activation='relu')(input_)

        out = Dense(3, activation='linear')(hidden)

        model = Model(inputs=input_, outputs=out, name="br-model")

        model.compile(loss='mse', \
                optimizer=self.sgd_br, \
                metrics=['accuracy', 'mse'])

        return model
```

Listing 2: Initialisierungsfunktion des Best-Response Netzwerkes mit Keras Chollet et al. (2015). Dense steht für „fully-connected“. „mse“ steht für „Mean Squared Error“ und entspricht der vorgebenen Kostenfunktion für das Best-Response Netzwerk (vgl. Heinrich and Silver, 2016).

Das Average-Response Netzwerk (siehe 3) ist dem Best-Response Netzwerk nachempfunden, legt jedoch eine andere Aktivierungsfunktion über das Output Layer. Der Output soll eine Wahrscheinlichkeitsverteilung über die möglichen Aktionen legen, woraufhin „Softmax“ 3 gewählt wurde.

```
def _build_avg_response_model(self):

        input_ = Input(shape=self.s_dim, name='input')

        hidden = Dense(self.n_hidden, activation='relu')(input_)

        out = Dense(3, activation='linear')(hidden)

        model = Model(inputs=input_, outputs=out, name="ar-model")

        model.compile(loss='categorical_crossentropy', \
                optimizer=self.sgd_ar, \
                metrics=['accuracy', 'ce'])

        return model
```

Listing 3: Initialisierungsfunktion des Average-Response Netzwerkes mit Keras Chollet et al. (2015). „categorical crossentropy“ wird in 4 beschrieben und entspricht der vorgebenen Kostenfunktion für das Average-Response Netzwerk.

In der „_init_“-Funktion werden die Netze gebaut und die Gewichte gesetzt. Keras ermöglicht eine sehr leserliche und simple Implementierung dieses Vorgangs.

```
# build average strategy model

self.avg_strategy_model = self._build_avg_response_model()

# build dqn aka best response model

self.best_response_model = self._build_best_response_model()
self.target_br_model = self._build_best_response_model()
self.target_br_model.set_weights(self.best_response_model.get_weights())
```

Listing 4: Initialisierung der Netzwerke und setzen der Gewichte des Target Netzwerks.

Sind die Netze initialisiert, können sie mit der Funktion „predict“ dazu genutzt werden, Zustände auf Aktionen abzubilden.

7.2.2 Lernen des Agenten

Das Best-Response Netzwerk lernt die Q-Werte über die Daten im M_{RL}. Diese Daten sind als MDP gespeichert und in der Form ($State$, $Action$, $Reward$, $State_{next}$, $Terminated$) vorzufinden. Terminated sagt aus, ob dieser Zustand $State$ ein finaler Zustand ist und nach $State_{next}$ nichts mehr kommt, da das Spiel vorbei ist. Es wird ein „Batch Learning“ Verfahren (vgl. Mnih et al., 2015) angewandt, da dies zu stabilerem Lernen und schnellerer Konvergenz führt. Die Idee dahinter ist, dass aus den gesammelten Daten eine randomisierte Menge (Batch) herausgenommen wird. Der negative Gradient wird dann über alle Elemente im Batch berechnet und das Netz aufgrund dieses Gradienten aktualisiert (siehe5).

```
def update_best_response_network(self):
        """
        Trains the dqn aka best response network through
        replay experiences.
        """

        if self._rl_memory.size() > self.minibatch_size:

                s_batch, a_batch, r_batch, s2_batch, t_batch = \
                        self._rl_memory.sample_batch(self.minibatch_size)

                target = self.target_br_model.predict(s_batch)

                for k in range(self.minibatch_size):
                        if t_batch[k]:
                                target[k][0][np.argmax(a_batch[k])] = \
                                        r_batch[k]
                        else:
                                Q_next = \
                                  np.max(self.target_br_model.predict \
                                  (np.reshape(s2_batch[k], (1, 1, 30))))
                                target[k][0][np.argmax(a_batch[k])] = \
                                  r_batch[k] + self.gamma * Q_next

                self.best_response_model.fit(s_batch, target,
                        batch_size=self.minibatch_size,
                        epochs=2,
                        verbose=0,
                        callbacks=[self.tensorboard_br])

                self.iteration += 1

                self.update_br_target_network()

                self.epsilon = self.epsilon ** 1/self.iteration
```

Listing 5: Aktualisieren der Q-Werte des DQN

Damit folgt das Best-Response-Netzwerk der DQN Lernregel wie in 4.2.1 beschrieben. Da zur Exploration der Zustände ϵ-greedy zum Einsatz kommt, die mit der Warscheinlichkeit ϵ eine Zufallsaktion wählt oder mit $1 - \epsilon$ die Aktion wählt, welche die höchste Belohnung verspricht, muss am Ende eines jeden Updates, ϵ kleiner werden, da der Agent zunehmend Aktionen wählen soll, die vielsprechend sind. Die „fit“-Funktion implementiert den Backpropagation Algorithmus auf Basis der gegebenen Kostenfunktion („loss“ siehe 2). Die „update_target_network“-Funktion passt alle 300 Schritte die Gewichte des Target Networks, dem Best Response Network an.

Das Average-Response Netzwerk soll aus den Daten des M_{SL} das bisherige Best-Response Verhalten ableiten (siehe 6).

```
def update_avg_response_network(self):
        """
        Trains average response network with mapping action to state
        """
        if self._sl_memory.size() > self.minibatch_size:
                s_batch, a_batch = \
                  self._sl_memory.sample_batch(self.minibatch_size)

                self.avg_strategy_model.fit(s_batch, a_batch, \
                batch_size=128, \
                epochs=2, \
                verbose=0, \
                callbacks=[self.tensorboard_sl])
```

Listing 6: Average-Response Netzwerk: Klassifizieren der bisherigen Aktionen des Best-Response Netzwerkes.

7.2.3 Spielen des Agenten

Jeder Agent kann über die „play“-Funktion mit dem Spiel interagieren. Dabei wählt er für das nächste Spiel (allerdings bereits in der Main-Funktion, da er für ein gesamtes Spiel bei seiner Strategie bleibt) mit der Wahrscheinlichkeit η die β Strategie oder mit $1 - \eta$ die π Strategie, jeweils repräsentiert duch das Best-Response oder das Average-Response Netzwerk. In der Spiel-Funktion, wird abgefragt welche Strategie gewählt wurde und nach dieser gehandelt. Dabei gilt $policy == a$, dann soll die Average-Response Strategie gespielt werden (siehe 7).

```
def play(self, policy, index, s2=None):
        if s2 is None:
                s, a, r, s2, t = self.env.get_state(index)
                self.remember_for_rl(s, a, r, s2, t)
                if t:
                        return t
        else:  # because it's the initial state
                t = False
                if not t:
                        if policy == 'a':
                                a = self.avg_strategy_model \
                                  .predict(np.reshape(s2, (1, 1, 30)))
                                self.env.step(a, index)
                        else:
                                a = self.act_best_response \
                                  (np.reshape(s2, (1, 1, 30)))
                                self.env.step(a, index)
                                self.remember_best_response(s2, a)
                self.update_strategy()
                return t
```

Listing 7: Spiel-Funktion des NFSP Agenten.

Da ϵ-greedy verwenden findet, wird die „predict“ Funktion des Best-Response Netzwerkes ausgelaget, die überprüft ob es eine randomisierte Aktion oder die vielversprechentste wählt. Außerdem wird nur dann das Best-Response Verhalten in M_{SL} gespeichert, wenn die Best-Response Strategie auch gespielt wird (vgl. Heinrich and Silver, 2016) und siehe Abbildung 6).

7.3 Main

In der Main Funktion werden die Agenten und die Spielumgebung initialisiert. Die Spielumgebung verwaltet die Zustände, Belohnungen und die Regeln des Spieles. Es hat sich jedoch herausgestellt, dass es schwierig ist, den Spielablauf innerhalb der Umgebung zu regeln. Dies wurde in die Main-Funktion ausgelagert (siehe 8).

```
def train(env, player1, player2):
  eta = float(Config.get('Agent', 'Eta'))
  players = [player1, player2]
  dealer = random.randint(0, 1)

  for i in range(int(Config.get('Common', 'Episodes'))):
    if dealer == 0:
    dealer = 1
    else:
    dealer = 0
    # Set dealer, reset env and pass dealer to it
    env.reset(dealer)

    lhand = 1 if dealer == 0 else 0
    policy = np.array(['', ''])
    # Set policies sigma
    if random.random() > eta:
      policy[dealer] = 'a'
    else:
      policy[dealer] = 'b'
    if random.random() > eta:
      policy[lhand] = 'a'
    else:
      policy[lhand] = 'b'

    # Observe initial state for dealer
    d_s = env.get_state(dealer)[3]

    terminated = False
    first_round = True
    d_t = False
    l_t = False

    while not terminated:
       actual_round = env.round_index
     if first_round and not d_t:
       d_t = players[dealer].play(policy[dealer], dealer, d_s)
       first_round = False
     elif not first_round and not d_t:
       d_t = players[dealer].play(policy[dealer], dealer)
     if not l_t:
       l_t = players[lhand].play(policy[lhand], lhand)
     if actual_round == env.round_index and not d_t:
       d_t = players[dealer].play(policy[dealer], dealer)
     if d_t and l_t:
       terminated = True
```

Listing 8: Live-Training der NFSP Agenten.

Die „while“-Schleife in Zeile 34 startet ein ganzen Spiel, dies ist erst terminiert ist, wenn einer der Spieler zwischenzeitlich folded oder beide Wettrunden zu Ende gespielt wurden und der Gewinner feststeht.

8 Fazit

Auf Basis der Vorgabe wurde ein lauffähiges Programm entwickelt, dass die notwendigen Technologien implementiert, eine Spielumgebung bereit stellt und „Neural Fictitious Self-Play“ nutzt. Das vorgebene Spiel „Leduc Hold'em“ wurde den Spezifikation entstprechend umgesetzt, wobei immer von einem „Zwei-Spieler“-Szenario ausgegangen wurde. Es ist einem jeden Spieler immer möglich, seinen aktuellen Zustand im Spiel zu erfragen und diesen in Form eines „Markov Decission Process“ zu speichern. Die Speicher für die jeweiligen Erfahrungen wurden der Vorgabe entsprechend umgesetzt und konnten zur Funktionsapproximation der Agenten genutzt werden. Die Agenten selbst implementieren die geforderten drei neuronalen Netze, wobei das Best-Response mit einem Target-Netzwerk und das Average-Response Netzwerk die Strategien der Spieler π und β darstellen.

Das Ziel war es, ein Nash-Gleichgewicht der Spielerstrategien herzustellen. Das Nash-GG sollte durch die „Exploitability“ eines Strategie-Profils gemessen werden. Der begrenzten Rechenleistung, die in dieser Arbeit zur Verfügung stand, ist es geschuldet, dass die Ergebnisse des Artikels nicht vollumfänglich reproduziert werden konnten. Die ursprüngliche Idee war es, den Spielbaum der Spielumgebung vollumfänglich zu durchlaufen, dabei die Strategie eines Spielers auf π und die des anderen auf β zu setzen. Der durschnittlich erwartete Gewinn der β Strategie stellt die „Exploitability“ dar. Da sich dieser Ansatz als zu rechenintensiv herausgestellt hat, wurde die „Exploitability“ approximiert und pro Iteration auf eine kleine Menge an randomisierten Zuständen abgebildet, was dazu geführt hat, dass diese ein verzerrtes und störungsbehaftetes Bild geliefert hat (siehe 7). Es lässt sich jedoch der gleiche Kurvenverlauf erkennen wie in der Vorgabe (siehe 8). Der Kurvenverlauf variiert stark abhängig von der verwendeten Aktivierungsfunktion im Best-Response Netzwerk. Die Wahl der Aktivierungsfunktion im Best-Response Netzwerk hat zu eigenwilligem Verhalten geführt, wurde für das Output Layer beispielsweise „relu“ genutzt, hat der Agent stets gefolded, da er recht schnell für jede Aktion den Wert 0 vorrausgesagt hatte. Da die Aktion mit der höchsten erwarteten Belohnung gewählt werden sollte, jedoch jede Aktion null Gewinn versprach kam, es zum eben beschriebenen Verhalten. Die besten Ergebnisse erzielte der Agent mit einer linearen Aktivierungsfunktion, welche den Aktionen auch negative Werte zuteilt. Des weiteren stellte sich heraus, dass Q-Werte als Target für das Average-Response Netzwerk zu einem schlechten Lernerfolg führte. Es wurden verschiedene Möglichkeiten in Betracht gezogen, die Q-Werte als Wahrscheinlichkeitsverteilung darzustellen. Jedoch

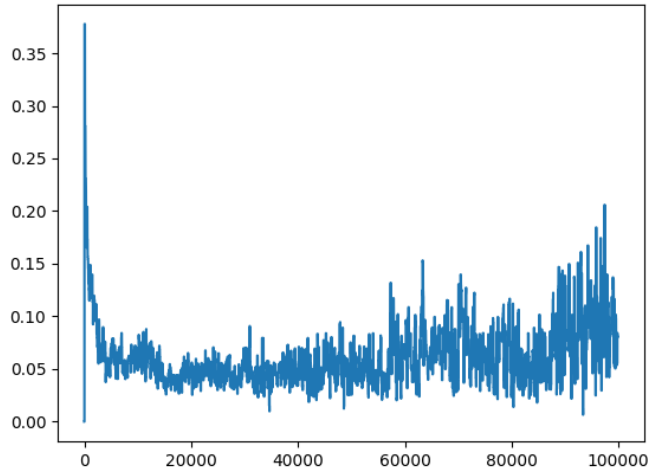


Abbildung 7: Ergebnis: Kurve bis 100k Iterationen

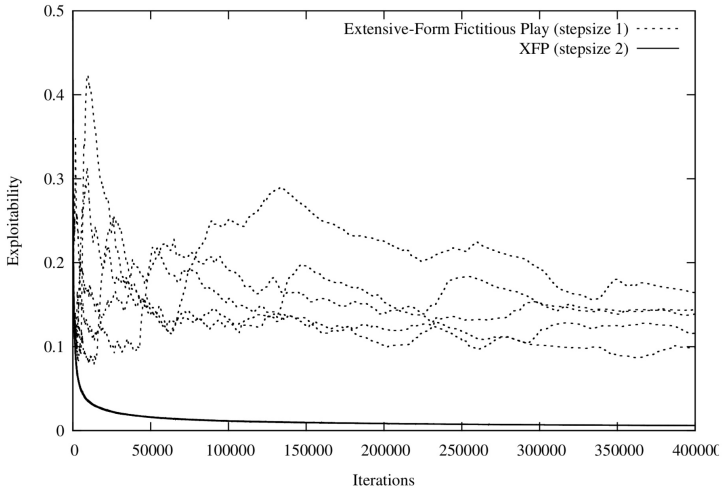


Abbildung 8: Ergebnis: Vorgabe, Kurve bis 400k Iterationen

wurden die besten Ergebnisse erzielt als die Q-Werte mit einer „1 aus k“ Codierung als Target verwendet wurden.

Reinforcement Learning ist dafür bekannt rechenintensiv zu sein und da diese Arbeit mehrere Agenten mit jeweils mehreren Netzen inkludiert, bedarf es mehr Leistung als die eines handelsüblichen Macbook Pro (mid. 2012) mit einem i7 und ohne dedizierte Grafikkarte. Es konnte jedoch gezeigt werden, dass das Programm ein ähnliches Verhalten zeigt wie in der Vorgabe und somit konnte das Ziel der Arbeit hinreichend erfüllt werden.

9 Ausblick

Abgesehen vom limitierenden Faktor Rechenleistung, kann gesagt werden, dass auch Spiele mit unvollständiger Information kein Problem mehr für maschinelles Lernen darstellt. Mit genug Rechenleistung kann man das in dieser Arbeit entwickelte Programm problemlos auf Spiele mit größerem Zustandsraum wie beispielsweise Texas Hold'em portieren. Es gibt Anbieter wie Amazon oder Google die Rechenleistung in der Cloud bereitstellen, so dass auch komplexere Welten exploriert werden können. Chiphersteller wie NVIDIA arbeiten unter Hochdruck an Chipsätzen, die für maschinelles Lernen optimiert werden. Wenn der Hardware Faktor beiseite gelegt wird, dann werden einfache Spieleumgebungen bald keine Hürden mehr für maschinelles Lernen darstellen. Natürlich sind Real-Welt Szenarion ungleich größer, doch mit fortschreitender Technologie ist es denkbar, dass Maschinen sich in Zukunft auch in der echten Welt komplizierter Probleme annehmen können, die dem agierenden Agenten nur eingeschränkte Information zu Teil werden lässt.

Literaturverzeichnis

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL `http://arxiv.org/abs/1606.01540`.

George W Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376, 1951.

François Chollet et al. Keras. `https://keras.io`, 2015.

M Bishop Christopher. *PATTERN RECOGNITION AND MACHINE LEARNING.* Springer-Verlag New York, 2016.

Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

Wolfgang Ertel. Grundkurs künstliche intelligenz. *Auflage, Wiesbaden*, 2009.

Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016. URL `http://arxiv.org/abs/1603.01121`.

Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, pages 805–813, 2015.

Manfred J Holler and Gerhard Illing. *Einführung in die Spieltheorie*. Springer-Verlag, 2006.

H Kuhn. Extensive games and the problem op information. *InH. KuhnandA. Tucker, editors, Contributions to the Theory of Games*, pages 193–216, 2016.

David S Leslie and Edmund J Collins. Generalised weakened fictitious play. *Games and Economic Behavior*, 56(2):285–298, 2006.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52(1):99 – 115, 1990. ISSN 0092-8240. doi: https://doi.org/10.1016/S0092-8240(05)80006-0. URL `http://www.sciencedirect.com/science/article/pii/S0092824005800060`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Mehryar Mohri, Ameet Talwalkar, and Afshin Rostamizadeh. *Foundations of machine learning (adaptive computation and machine learning series)*. Mit Press, 2012.

John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Jeff S Shamma and Gürdal Arslan. Dynamic fictitious play, dynamic gradient play, and distributed convergence to nash equilibria. *IEEE Transactions on Automatic Control*, 50(3):312–327, 2005.

David Silver and Demis Hassabis. Alphago: Mastering the ancient game of go with machine learning. *Research Blog*, 2016.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Bernhard Von Stengel. Efficient computation of behavior strategies. *Games and Economic Behavior*, 14(2):220–246, 1996.

Anhang

Das ganze Projekt ist der beiliegenden CD-ROM / DVD zu entnehmen.
Des weiteren kann das Projekt auch auf Github (https://github.com/DoktorDaveJoos/Neural-Ficititious-Self-Play-in-Imperfect-Information-Games) gefunden werden.

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: